XI. Weiterführende Konzepte der objektorientierten Programmierung

1. Einleitung und Einordnung
2. Vererbung, dynamische Bindung und abstrakte Klassen
3. Schnittstellenbeschreibungen – *Interfaces*

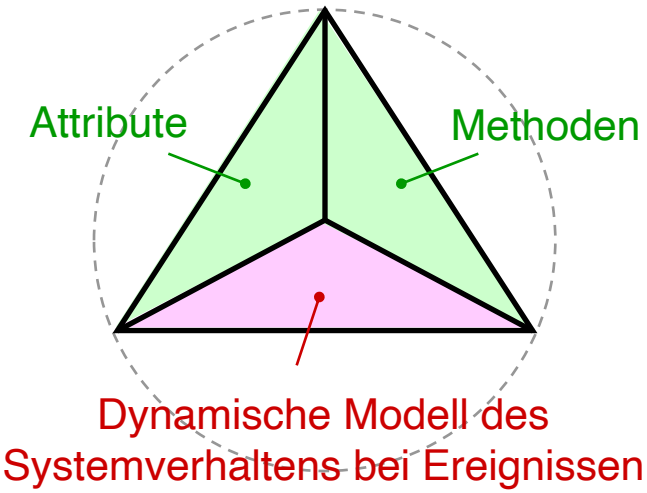
1. Einleitung, Einordnung und Beispiel

- Grundkonzepte der Objektorientierung und ihre Vorteile
- Objekte, Identität und Methoden

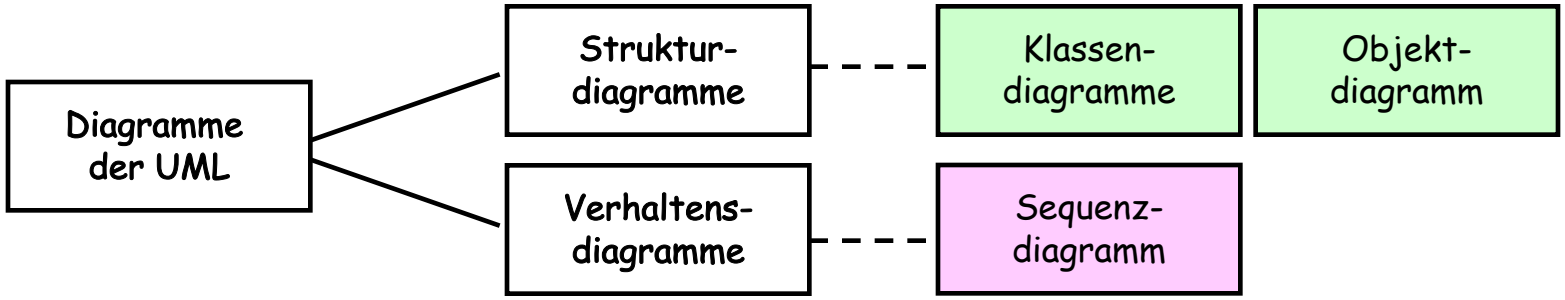
Grundkonzepte der Objektorientierung und ihre Vorteile

Aspekte der Objektorientierung

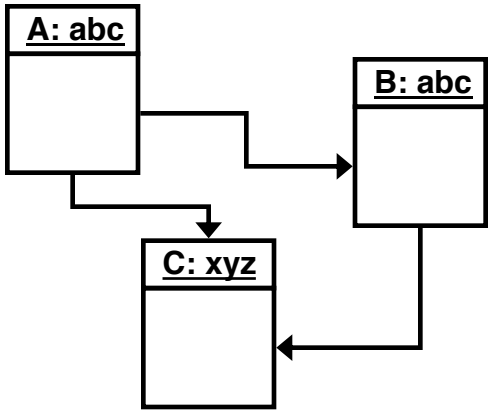
- Bei der objektorientierten Analyse (OOA) haben wir drei Aspekte betrachtet, die statische bzw. dynamische Aspekte betreffen (vgl. **Teil VII**)
  - Definition eines statischen Objektmodells
    - Daten (Attribute)
    - Operationen (Methoden)
  - Definition eines dynamischen Modells des Systemverhaltens (Reaktion auf Ereignisse)

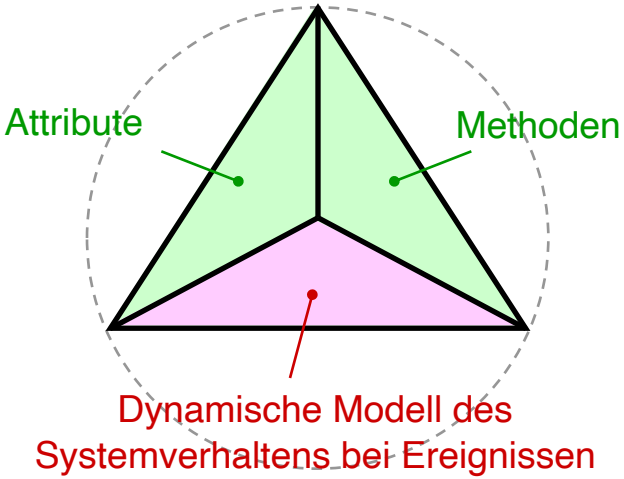


- Diese Aspekte werden auch beim Entwurf (*object-oriented design*, OOD) betrachtet – siehe beispielsweise die unterschiedlichen Diagramme in UML (vgl. **OOP Einf.**)

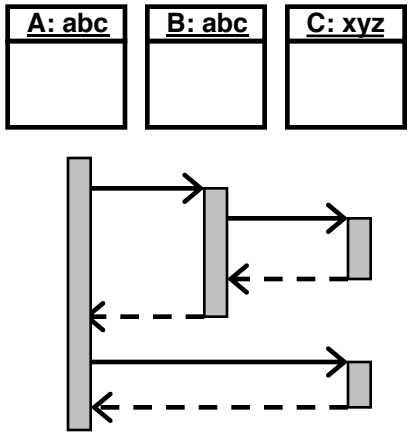


- Konzepte und Begriffe für statische Objekteigenschaften
  - Klasse (mit Klassenname)
  - Objekt (mit Identifikator)
  - Objektzustand
  - Systemzustand
  - Vererbung, Generalisierung (neu)



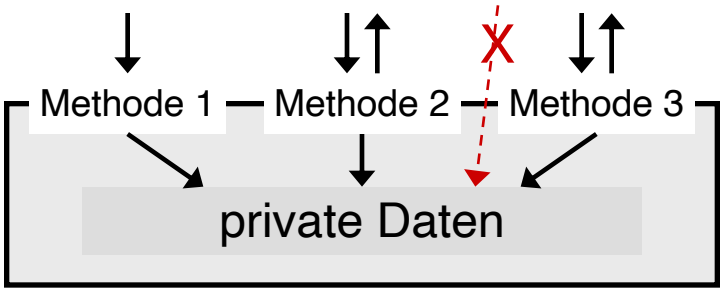


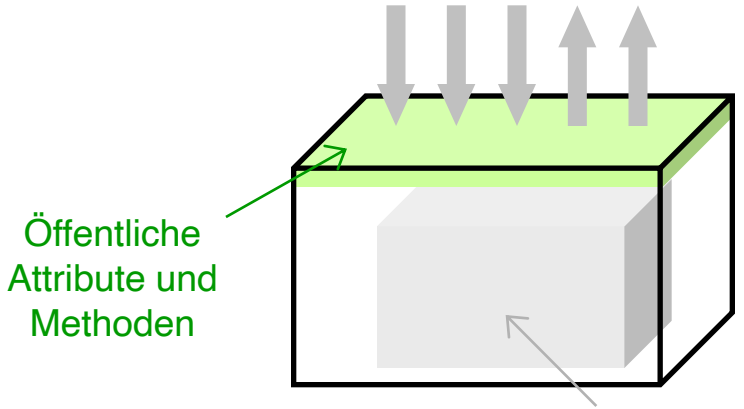
- Konzepte und Begriffe für dynamische Eigenschaften
  - Austausch von Nachrichten
  - Methoden (Aufruf, Rückgabewerte)
  - Objektlebenszyklus (dynamische Erzeugung, Elimination durch *gc*)
  - Dynamische Bindung (neu)



Strukturierungsmechanismen – Abstraktion und Kapselung

- Abstraktion und Kapselung





- Nutzen und Vorteile der Abstraktion und Datenkapselung
  - reduziert Fehler durch lokale (Projekt-) Verantwortlichkeiten
  - erleichtert die Wiederverwendung von Komponenten
  - erleichtert die Modifikation von Komponenten

- Die Datenkapselung wird in objektorientierten Sprachen konsequent umgesetzt, indem (a) Daten (Attribute) und Operationen (Methoden) in Klassen zusammengefasst werden und (b) durch verschiedene Sichtbarkeits-indikatoren deren Zugreif-/Sichtbarkeit gesteuert werden kann

- **Grundsätzliche Feststellung**: Diese Eigenschaften bzw. Vorteile werden dem Programmierer nicht geschenkt, sondern kommen nur bei einem „guten“ Entwurf der Software zum Tragen

Objekte, Identität und Methoden

Objekte, Instanzen und ihre Implementierung

- Ein objektorientiertes (Software-) System besteht aus einer Menge miteinander kommunizierender Objekte
- **Objekte**: **Sichtbare Schnittstelle + nicht sichtbare Implementierung**
  - Die sichtbare Schnittstelle umfasst
    - Extern sichtbare und zugreifbare Attribute (*public attributes*)
    - Spezifikation öffentlicher Operationen (*public operations / methods*)
  - Die (nicht sichtbare) Implementierung umfasst
    - Nur intern verfügbare Attribute (Variablen)
    - Nur intern verfügbare Operationen (*private operations / methods*)
    - Die Implementierung der öffentlichen und privaten Operationen
- Die Menge der öffentlichen und privaten Attribute (Variablen) einer Klasse – die nicht `static` deklariert wurden (in Java) – nennt man **Instanzvariablen** des Objekts
- Die Werte aller Instanzvariablen definieren den aktuellen **Objektzustand**

- Ein Objekttyp (entsprechend einer definierten Klasse, **`class`**, in Java) beschreibt die Struktur und das Verhalten von Objektinstanzen dieses Typs
- Die Typ-Beschreibung legt die sichtbare Schnittstelle und die unsichtbare Implementierung von Instanzen des Objekttyps fest
- Im Prinzip können zu einem gegebenen Objekttyp beliebig viele Instanzen erzeugt werden
  - Jede Instanz besitzt eine private Kopie der Instanzvariablen mit instanzspezifischen Werten
  - Die Operationen (Methoden) sind nur einmal beim Typ spezifiziert und implementiert; sie werden von allen Instanzen geteilt
- In vielen objektorientierten Systemen und Sprachen wird zwischen (primitiven) Datentypen und Objekttypen unterschieden
  - Die Werte primitiver Datentypen sind reine Datenwerte ohne eindeutige Identität, z.B. **`int`**, für die nur die Struktur festgelegt wird
  - Die Werte von Objekttypen sind Objekte mit eindeutiger Identität, für die sowohl die Struktur als auch das Verhalten der Objektinstanzen spezifiziert ist (das Konzept des Objekttyps ist abgeleitet von den **Abstrakten Datentypen** (ADT, *abstract data types*)

Objektidentität

- Jedes Objekt ist durch einen systemweit eindeutigen *object identifier* (OID) ausgezeichnet und dadurch eindeutig von anderen Objekten unterscheidbar
- Die Objektidentität bleibt während der Lebensdauer des Objekts stets gleich, d.h. sie ist nicht veränderbar
- Mit der Objektidentität sind die Struktur und das Verhalten des Objekts verknüpft
  - Die Struktur und das Verhalten eines Objekts sind während seiner Laufzeit nicht änderbar
  - Der Objektzustand kann verändert werden
- Objekte können mittels ihrer Identität (= OID) angesprochen werden; es können mehrere Referenzen auf dasselbe Objekt existieren
- Wichtig ist die Unterscheidung zwischen Objektgleichheit und Objektidentität:
  - **Objektgleichheit** (*object equality*): Zwei verschiedene Objekte (desselben Typs) haben die gleiche Ausprägung des internen Zustands (= gleiche Werte)
  - **Objektidentiät** (*object identity*): Es wird dasselbe Objekt referenziert

Objektmethoden

- **Methoden** sind Prozeduren oder Funktionen, die ein bestimmtes Verhalten des Objekts festlegen
- Der Aufruf einer Methode auf einem Objekt führt zur Ausführung in diesem Objekt (in dessen Objekttyp)
- Definition einer **Methode**: **Sichtbare Spezifikation + unsichtbare Implementierung**
  - Spezifikation einer Methode:
    - **Signatur** der Methode: Name der Methode + Namen und Wertebereiche von Parametern + Wertebereich der Rückgabe
    - **Vor- und Nachbedingungen** der Ausführung (optional)
      - **Vorbedingung**: Muss beim Aufruf der Methode erfüllt sein (Pflicht beim Aufrufenden)
      - **Nachbedingung**: zugesichertes Verhalten seitens des Objekts (Implementierer der Methode ist in der Pflicht)

- (Unsichtbare) Implementierung (Realisierung) einer Methode:

  Sie kann während der Ausführung …

  - lesend oder schreibend auf die Instanzvariablen des ausführenden Objektes (für das die Methode spezifiziert wurde) zugreifen
  - weitere Operationen (Methoden) auf demselben Objekt oder auf anderen Objekten aufrufen – und ggf. deren Ergebnisse weiter verwenden

- Dieselbe Methode kann bei gleichen Eingabeparametern – angewandt auf dasselbe Objekt – unterschiedliche Resultate liefern, da die Objekte durch ihre Instanzvariablen und deren Werte ein „Gedächtnis“ haben

Einordnung:

- Sichtbarkeit von Variablen und Methoden: Kontrolle der Sichtbarkeit, z.B. durch Voranstellen von Schlüsselworten wie **`public`**, **`private`** oder **`protected`** bei der Deklaration (alternativ: Deklaration in entsprechend markierten Abschnitten) – vgl. die Diskussion der Namensräume und Zugriffsrechte (**Teil VII**)
- Hinweis: Die Wirkungsweise der o.g. Schlüsselworte kann bei verschiedenen Programmiersprachen oder Entwurfsumgebungen unterschiedlich sein – hier hilft die Konsultation von Handbücher bzw. Dokumentationen

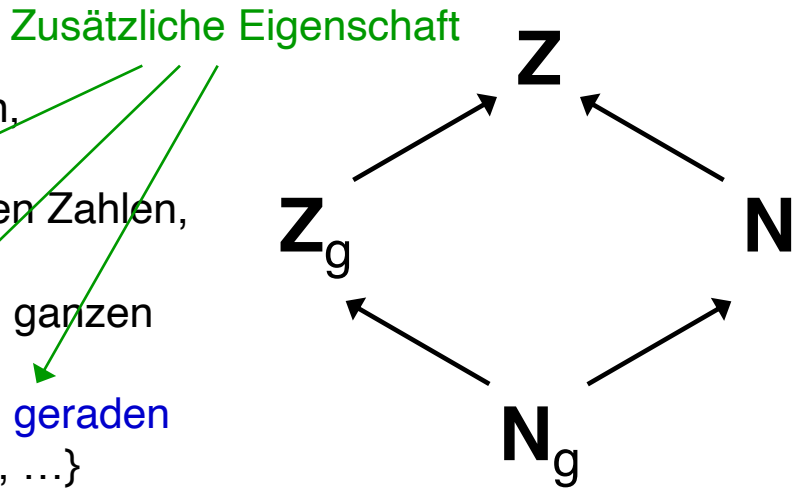
2. Vererbung, dynamische Bindung und abstrakte Klassen

- Spezialisierung, Vererbung und Subtypen
- Erweiterung bestehender Klassen
- Vererbung und Klassen-Hierarchien
- Dynamische Bindung
- Objekterzeugung bei erweiterten Klassendefinitionen
- Polymorphie und abstrakte Klassen

Spezialisierung, Vererbung und Subtypen

Spezialisierung von Typen

- Generelles Konzept der Subtypen – ein Typ ist eine Spezialisierung eines anderen Typs, dessen Elemente mehr (zusätzliche) Eigenschaften besitzen
- Im mathematischen Sinn ist der Subtyp eine Teilmenge des Supertyps
- Beispiel:
  - Typ **Z** = Menge der ganzen Zahlen, {..., -2, -1, 0, 1, 2, 3, ...}
  - Subtyp $\mathbf{Z}_g$ = Menge der geraden ganzen Zahlen, {..., -2, 0, 2, 4, 6, ...}
  - Subtyp **N** = Menge der nichtnegativen ganzen Zahlen, {0, 1, 2, 3, 4, ...}
  - Subtyp $\mathbf{N}_g$ = Menge der nichtnegativen geraden ganzen Zahlen, {0, 2, 4, 6, ...}



- Konzeptuelles **Problem**: Es existiert eine Operation *succ* auf der Menge **Z**, die den Nachfolger einer Zahl liefert; diese Operation kann nicht einfach an $\mathbf{Z}_g$ weiter gegeben (vererbt) werden, da $succ(4) \notin \mathbf{Z}_g$, d.h. man müsste $succ_{\mathbf{Z}_g}$ abändern, so dass z.B. $succ_{\mathbf{Z}_g}(4) = 6$ (!) liefert

Spezialisierung und Vererbung in Programmiersprachen

- Die Spezialisierung von Typen deutet einen **zentralen Konflikt** an, der bei objektorientierten Methoden in Programmiersprachen nicht auflösbar ist:
  - Man kann „Vererbung“ als Spezialisierung auffassen – die Elemente eines Subtyps behalten die Eigenschaften des Supertyps (sie werden vererbt); auch die Operationen bleiben unverändert erhalten (Ergebnisse können dann aus dem Subtypen heraus führen)
  - Man kann „Vererbung“ zum Zwecke der Arbeitsökonomie auffassen – das wiederholte Programmieren der gleichen Methoden wird erspart und „erbt“ diese nach Möglichkeit vom Supertyp; aus pragmatischen Gründen ist man häufig gezwungen, einige der ererbten Methoden zu modizieren
- In Java – sowie in objektorientierten Programmiersprachen allgemein – wird (technisch) die Ökonomie-Sicht realisiert
- Aus methodischer Sicht sollte man sich – als Programmierer / Software-Entwickler – an dem Spezialisierungsprinzip orientieren

Bsp.:

Autobus ist z.B. **Subtyp** von Kraftfahrzeug

Erweiterung von Klassen zur Spezialisierung

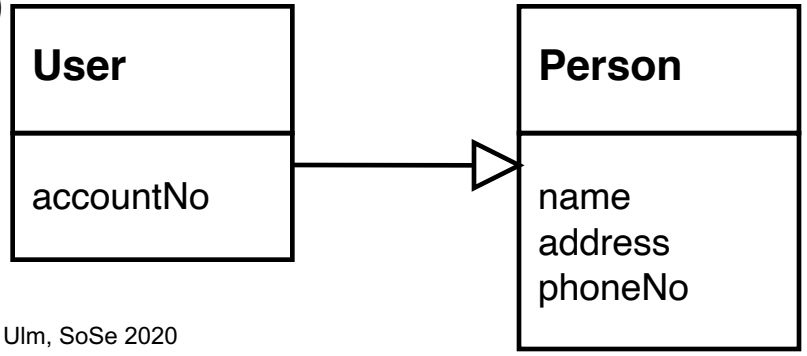
- Eine Klasse legt die Struktur von Objekten und deren Verhalten (in Form von Variablen und Methoden) fest, die in jeder Instanz enthalten sind
- Die grundlegende zusätzlichen Idee der Objektorientierung ist, dass die Klassen auch die Ähnlichkeiten zwischen Objekten unterschiedlicher Typen zum Ausdruck bringen;

  die Ähnlichkeiten werden durch

  - Erweiterung von Klassen,
  - Vererbung und
  - Dynamische Bindung (Polymorphie)

  ausgedrückt

- In UML wird die Erweiterung durch (gerichtete) Verbindungskanten in einem Graph ausgedrückt (vgl. **OOP Einf.**)



Erweiterung bestehender Klassen

Erweiterung einer Klassendefinition in Unterklassen

Deklaration von Unterklassen

- In der Anwendungs-Programmierung werden Unterklassen häufig in folgenden Situationen definiert: Es existiert eine Klasse, die durch kleine Erweiterungen oder wenige Änderungen für das aktuelle Problem angepasst werden kann (die gegebene Klasse definiert die Oberklasse für die daraus abgeleitete Unterklasse)

- Syntax für die Erweiterung einer Klasse (in Java)

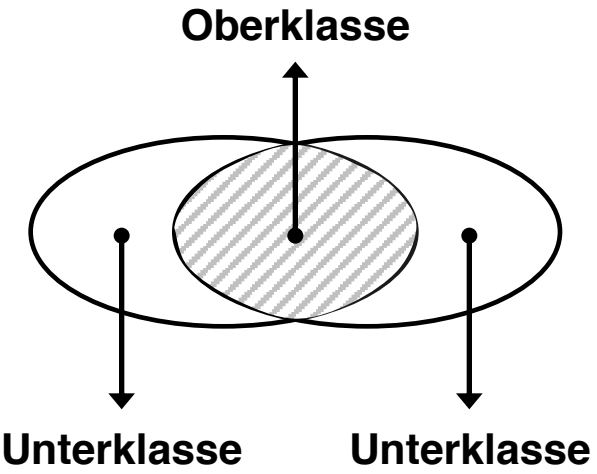
```
public class <Name_Unterklasse> extends <Name_Oberklasse> {
    ...
    ...  // Aenderungen und Erweiterungen
}
```

Erläuterungen:

- Die Klasse <Name$_{Oberklasse}$> wurde an anderer Stelle bereits definiert – und zumeist in einer separaten Datei Name$_{Oberklasse}$.java abgelegt
- Das Schlüsselwort **extends** stellt eine Beziehung der Änderungen und Erweiterungen zu der bereits existierenden Definition der Oberklasse her

Umgekehrte Vorgehensweise – Abstraktion in Oberklassen

- Wenn verschiedene Klassendefinitionen vorhanden sind, in denen einige Eigenschaften in ihrer Funktion und Bedeutung identisch sind, so kann eine **Oberklasse** als **Abstraktion** hiervon gebildet werden
  - Die gemeinsamen Eigenschaften (Attribute, Methoden) verwandter Klassen können so in einer Oberklasse definiert werden
  - Oberklassen können zu Unterklassen erweitert werden




```
class Employee {
    String name;
    String address;
    String socialSecurityNo;
    float  salary;
}
```

```
class Student {
    String name;
    String address;
    int    matriculationNumber;
}
```

```
class Person {
    String name;
    String address;
}
```

```
class Employee extends Person {
    String socialSecurityNo;
    float  salary;
}
```

```
class Student extends Person {
    int    matriculationNumber;
}
```

Vererbung, Sichtbarkeit und Zugriff auf Elemente einer Klasse

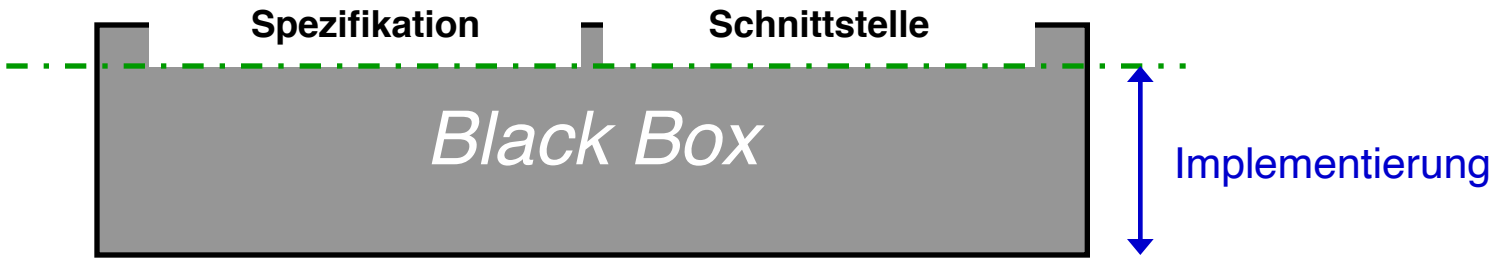
- Vererbte Variablen und Methoden aus der `Person`-Klasse können auch in der Definition von `Student` verwendet werden

  Ausnahme: Solche Elemente, die mit dem **`private`**-Modifizierer deklariert wurden, können auch nicht aus einer Unterklasse zugegriffen werden

- Die Modifizierer **`protected`** oder **`<default>`** werden verwendet, wenn eine Variable oder Methode einer Klasse zwar nicht öffentlich zugreifbar, jedoch aus deklarierten Unterklassen zugreifbar sein soll

  Hinweis: Auf ein **`protected`** Element kann auch aus jeder Klasse des gleichen Pakets (**`package`**) zugegriffen werden, **`<default>`** lässt nur Zugriffe von Subclasses innerhalb der package zu.

- Die Deklaration einer Variable oder Methode als **`protected`** ist Teil der Implementierung einer Klasse, jedoch nicht Teil der (öffentlichen) Schnittstelle der Klasse



Vererbung und Klassen-Hierarchien

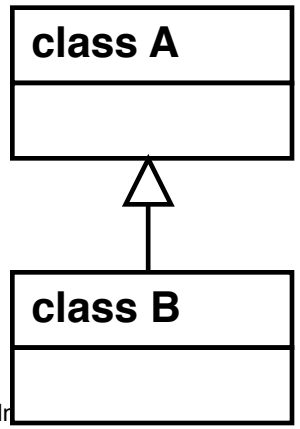
Mechanismen der Abstraktion – Vererbung, Unter- und Oberklassen

(Einfache) Vererbung (*inheritance*)

- Beim objektorientierten Entwurf werden zur Problemlösung Attribute sowie Methoden von Objekten identifiziert
- Der Begriff der **Vererbung** (*inheritance*) bezieht sich auf den Mechanismus, dass eine Klasse Teile oder die Gesamtheit ihrer Struktur und ihrem Verhalten aus einer anderen Klasse übernehmen (erben) kann
- Eine Klasse B, die Eigenschaften einer anderen Klasse A erbt, heisst Unterklasse (Subklasse) von A; ist Klasse B eine Unterklasse von A, so ist A die Oberklasse (Superklasse) von B

In **UML**:

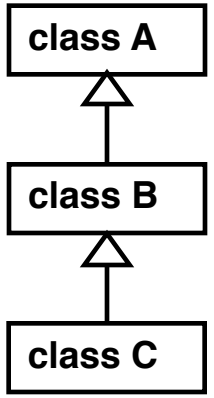
Hinweis: In anderen Sprachen, z.B. C++, werden die Klassen häufig als *derived class* bzw. *base class* bezeichnet



Oberklasse

Unterklasse

- Beispiel einer Hierarchie von 3 Klassen (mehrere Ebenen)



class A
var1 var2 op1 op2

var1 op1
var2 op2

verfügbar in Objekten mit Objekttyp **`A`**

class B extends A
var3 var4 op3 op4

var1 op1
var2 op2
var3 op3
var4 op4

verfügbar in Objekten mit Objekttyp **`B`**

class C extends B
var5 var6 op5 op6

var1 op1
var2 op2
var3 op3
var4 op4
var5 op5
var6 op6

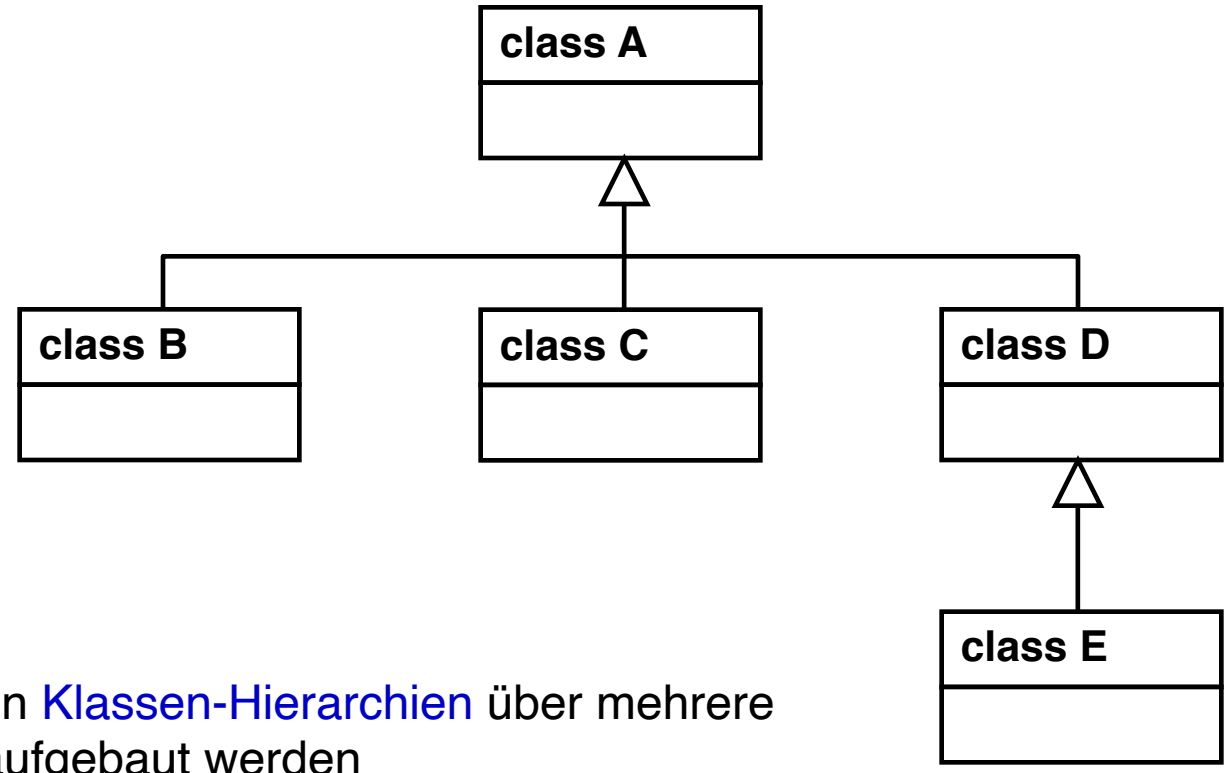
verfügbar in Objekten mit Objekttyp **`C`**

- Angabe der Vererbungs-beziehung
  - In Java: **`extends`**
  - Weitere gebräuchliche Bezeichnungen in anderen Sprachen sind z.B. **`super ...`** und **`inherits from ...`**
- **Einordnung**: Jede Instanz eines Subtyps ist auch eine indirekte Instanz aller Objekttypen, von denen der Subtyp erbt

Klassen-Hierarchien

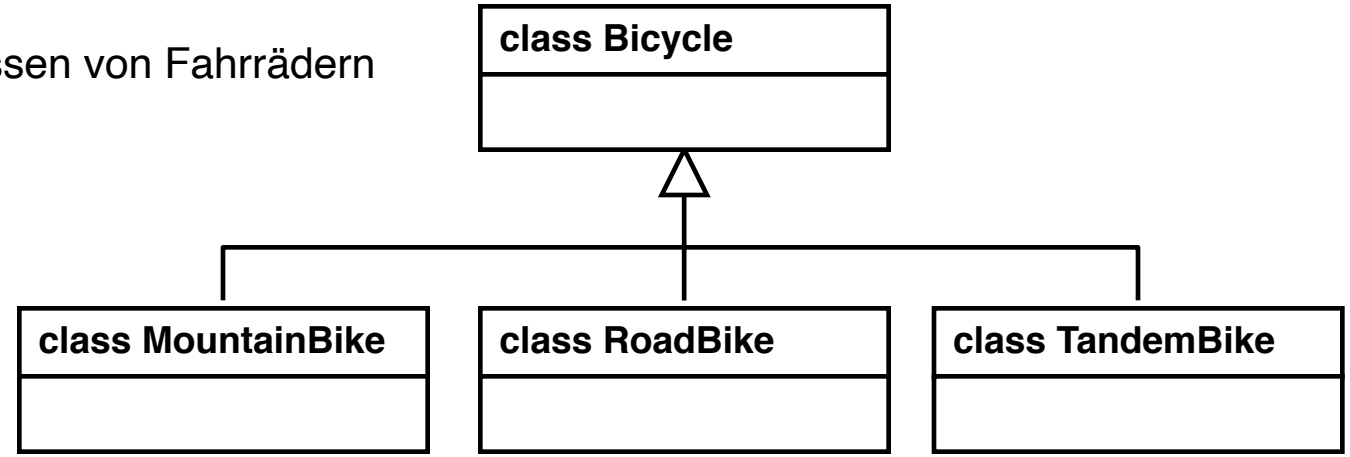
- Hierarchien und Vererbungen können sich über mehrere Ebenen erstrecken
- Es können aus einer (Super-) Klasse mehrere Unterklassen abgeleitet werden – sie werden auch Geschwisterklassen genannt; diese übernehmen manche Strukturen und Verhalten der Oberklasse

In **UML**:

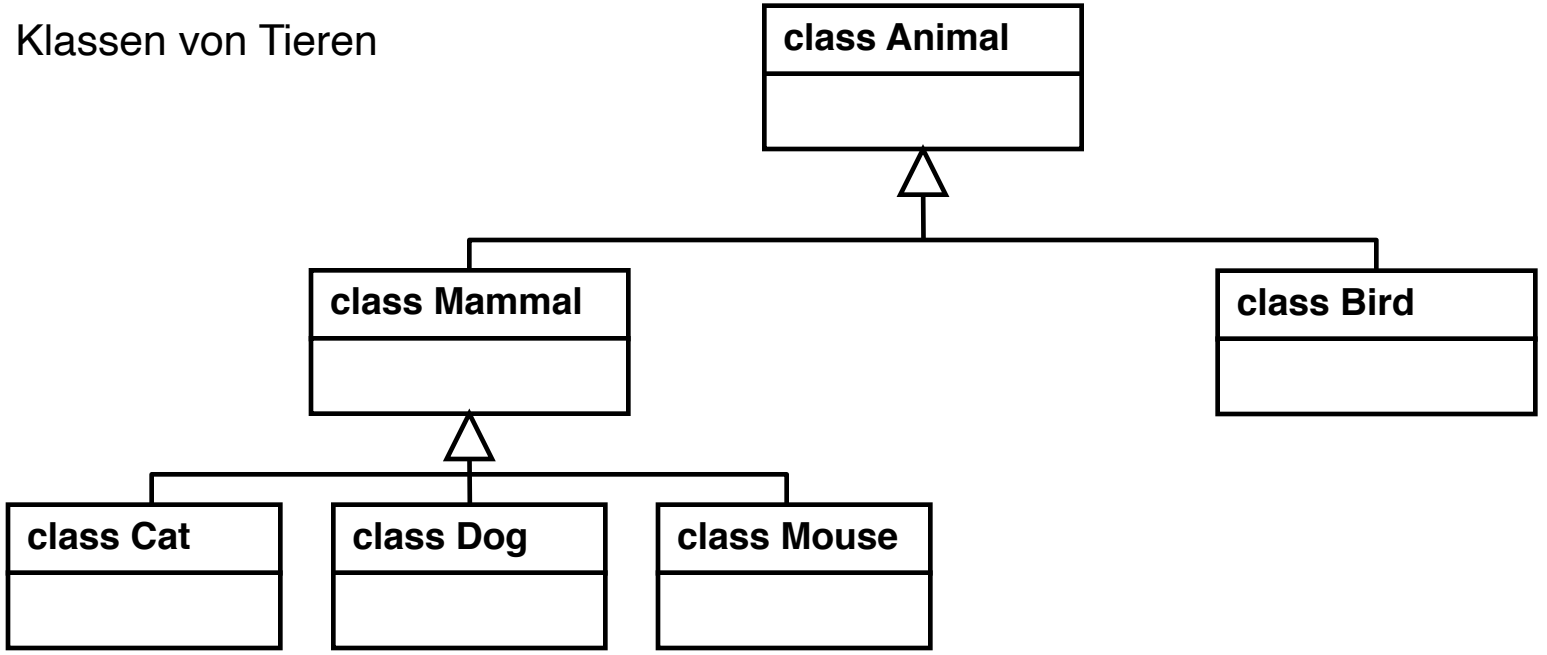


- So können Klassen-Hierarchien über mehrere Ebenen aufgebaut werden

- Beispiele
  - Klassen von Fahrrädern



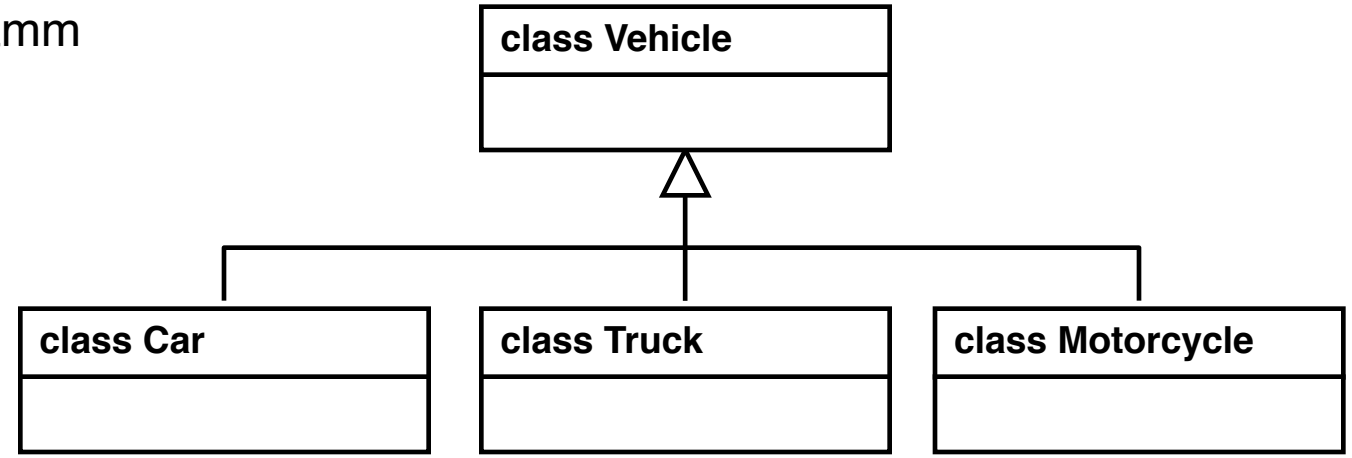
  - Klassen von Tieren



Klassenhierarchie – am Beispiel für Fahrzeuge

Hierarchie

- Diagramm



- Realisierung in Java (Code-Fragmente)

```
class Vehicle {
    int     registrationNumber;
    Person  owner;  // Annahme, dass eine Klasse Person definiert wurde

    void transferOwnership(Person newOwner) {
        ...
    }
}  // end class Vehicle

... <Fortsetzung>
```

```
...

class Car extends Vehicle {
    int numberOfDoors;
    ...
}  // end class Car

class Truck extends Vehicle {
    int numberOfAxes;
    ...
}  // end class Truck

class Motorcycle extends Vehicle {
    boolean hasSideCar;
    ...
}  // end class Motorcycle
```

Variable, Objekte und Hierarchien

- Deklaration

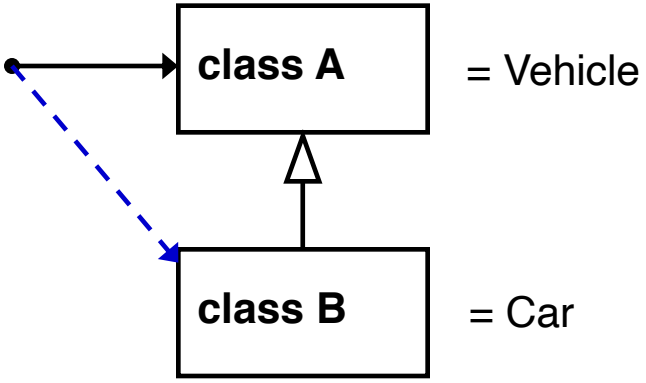
```
Car myCar = new Car();
```

- Zugriffe auf Variablen und Methoden

```
myCar.registrationNumber
myCar.owner
myCar.transferOwnership()
```

- **Wichtige Eigenschaft**: Eine Variable, die einen Zeiger auf ein Objekt der Klasse A enthält, kann auch einen Zeiger auf ein Objekt einer der Unterklassen von A enthalten

Bsp.:
```
Vehicle myVehicle  = myCar;
Vehicle myVehicle2 = new Car();
```

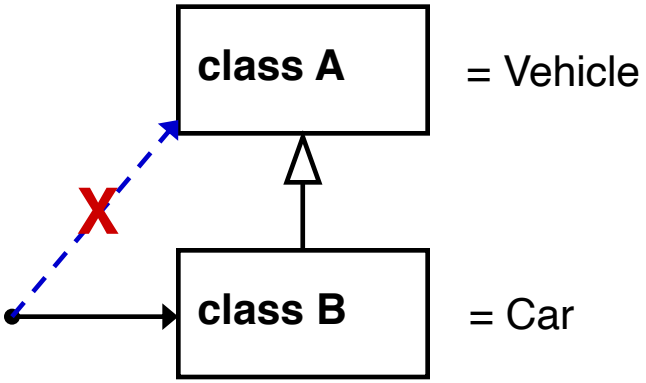


- Die Referenz von einer kleineren Eigenschafts-menge (in der Oberklasse) auf Objekte mit größerer Eigenschaftsmenge (in der Unter-klasse) ist möglich, umgekehrt ist der Zugriff nicht erlaubt

Bsp.:
```
Car myCar2;
myCar2 = myVehicle;
```
nicht erlaubt → **Uebersetzungfehler**

Begründung: `myVehicle` könnte auch ein anderes Vehikel sein, das kein Auto ist (z.B. ein Motorrad, dessen Eigenschaften nicht kompatibel sind)

Bsp.:
```
Vehicle myVehicle = myMotorcycle;
myCar2 = myVehicle;
```
inkompatibel



- Das Problem ist ähnlich der Kompatibilitäten bei der Zuweisung von Variablen aus Standardtypen (vgl. **Teil III**, implizite Typ-Konvertierung)

  Bsp.:
  ```
  short shNumber;
  int   intNumber;
  ...
  intNumber = shNumber;    // OK
  shNumber  = intNumber;   // nicht erlaubt → Uebersetzungsfehler
  ```

- Die explizite Typ-Konvertierung mittels des *casting*-Operator erzwingt eine Zuweisung (vgl. **Teil III**)

  Bsp.:
  ```
  shNumber  = (short)intNumber;
  ```

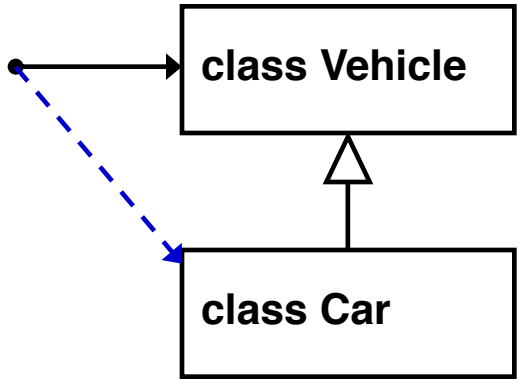
- Die explizite Typ-Konvertierung kann auch für Objekte verwendet werden

  Bsp.: `myVehicle` zeigt auf ein Auto (`car`), dann ist die folgende Zuweisung erlaubt
  ```
  myCar  = (Car)myVehicle;
  ```
  mit direktem Zugriff auf Elemente: `((Car)myVehicle).numberOfDoors`

- **Problem**: Wenn `myVehicle` auf ein Objekt vom Typ `Motorcycle` zeigt, dann ergibt `(Car)myVehicle` einen Laufzeitfehler!

Unterklassen, Unterbereiche, Typtest und Typkonvertierung

- Das Objekt `myCar` „weiß“, dass es ein Objekt der Klasse `Car` ist und nicht einfach ein `Vehicle`
- In Java kann mit dem **`instanceof`**-Operator die Zugehörigkeit zu einer bestimmten Klasse geprüft werden (das Ergebnis ist boole‘sch)



Bsp.:
```
if (myVehicle instanceof Car)
    n = ((Car)myVehicle).numberOfDoors;
```

- Ein weiteres Beispiel – für Klassen zur Spezifikation von Tieren
  - Klassendefinitionen

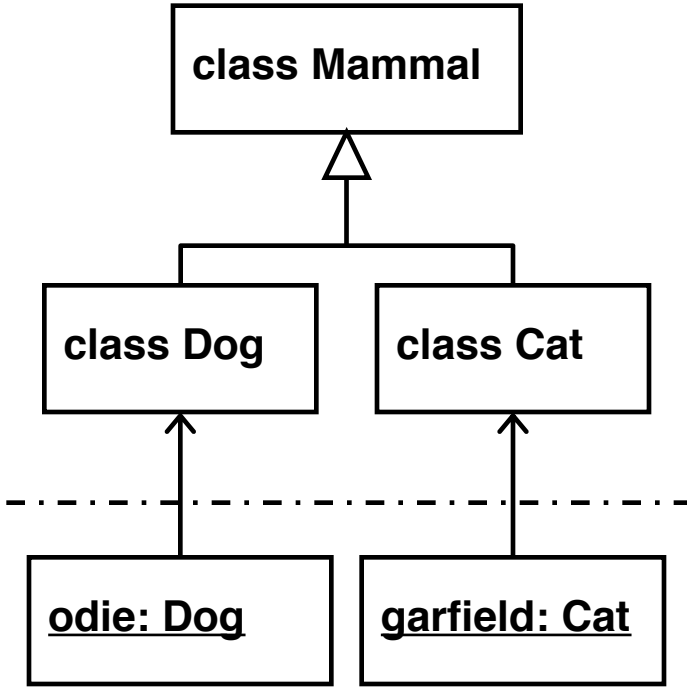
```
class Mammal {
    ...
}  // end class Mammal

class Dog extends Mammmal {
    void bark() {...}  // bellen
    ...
}  // end class Dog

class Cat extends Mammmal {
    void purr() {...}  // schnurren
    ...
}  // end class Cat
```

- Instanzen von Objekten

```
Mammal animal;
Dog    odie     = new Dog();
Cat    garfield = new Cat();
```

Es ist jetzt erlaubt, sowohl `odie` (Zeiger-Variable auf ein Objekt vom Typ `Dog`) als auch `garfield` (Zeiger-Variable vom Typ `Cat`) an die Zeiger-Variable `animal` zuzuweisen:

```
animal = odie;
animal = garfield;
```



- Zur **Absicherung der Kompatibilitäten** sollten Zugriffe auf Methoden der Objektinstanzen mit expliziter Typkonvertierung einen **Typtest** verwenden:

```
...
if (animal instanceof Dog) {
   ((Dog)animal).bark();
}
else if (animal instanceof Cat) {
   ((Cat)animal).purr();
}
...
```

Dynamische Bindung

Klassenhierarchien und Vererbung von Methoden

- Wie bereits diskutiert, wird in Java (und anderen objektorientierten Sprachen) aus pragmatischen Gründen bei der Bildung von Unterklassen kein strenges Spezialisierungsprinzip gefordert; bei der Vererbung werden Modifikationen (an den Methoden) zugelassen

- Beispiel in Java
  - Klassendefinitionen

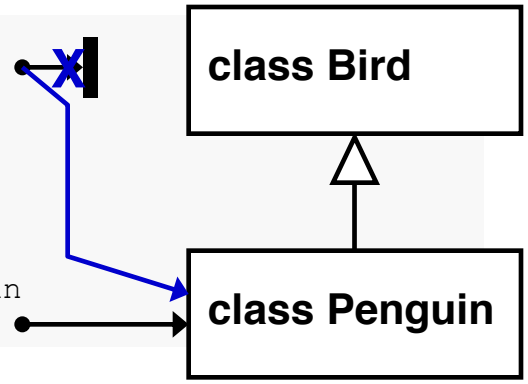
```
class Bird {
    ...
    void escape() {
        ...
        flyAway();
        ...
    }
    ...
}  // end class Bird
```

```
class Penguin extends Bird {
    ...
    void escape() {
        ...
        diveAway();
        ...
    }
    ...
}  // end class Penguin
```

Einordnung: Beide Klassen enthalten eine Methode zur Beschreibung des Fluchtverhaltens der jeweiligen Spezies (`Penguin` ist eine Unterklasse von `Bird`)

- Modifikation der Methode `escape()`, die ausgeführt werden kann

```
Bird    myBird;
Penguin jonathan = new Penguin();
...
jonathan.escape();  // spezielle Methode: diveAway
myBird = jonathan;
...
myBird.escape();    // welche Methode ist das?
```



Erläuterung: Java geht davon aus, dass ein Objekt mit **new** kreiert wird und dabei seine Attribute und Methoden erhält; diese behält das Objekt für immer, egal durch welche Variable es gerade referenziert wird

Im Beispiel heisst das, dass mit **new** `Penguin()` ein Objekt kreiert wurde, das insbesondere das spezielle Fluchtverfahren mittels Tauchen (`diveAway`) beherrscht; das bleibt so, unabhängig, ob dieses Objekt gerade von der Variablen `jonathan` oder der Variablen `myBird` referenziert wird (**dynamische Bindung**)

- **Definition (dynamische Bindung)**:

Die Sprache Java hat für Methoden eine dynamische Bindung, d.h. Methoden hängen nicht von der Klasse der (Referenz-) Variablen ab, sondern von der Klasse des Objekts, das die Variable gerade referenziert; diese können nur Objekte von Unterklassen sein.

- Der Begriff der dynamischen Bindung bringt zum Ausdruck, dass z.B. bei `myBird.escape()` die tatsächlich ausgeführte Methode nicht statisch festliegt, sondern sich zur Laufzeit immer wieder ändern kann – je nachdem, welches Objekt gerade von der Variablen `myBird` referenziert wird

  Hinweis: **Attribute** sind nicht dynamisch; wenn beide Klassen (`Bird` und `Penguin`) ein Attribut `color` besitzen, dann hat das Objekt der Unterklasse `Penguin` zwei Attribute namens `color`

```
class Bird {
    Color color;
    ...
}  // end class Bird
```

```
class Penguin extends Bird {
    Color color;
    ...
}  // end class Penguin
```

- Im Falle der mehrfach (in der Unter- und der Oberklasse) definierten Attribute liegt wiederum ein Verdeckungsproblem in den Namensräumen vor; auf die Attribute kann über spezielle Variablen (**`this`** und **`super`**) zugegriffen werden

Die Variable `super`

- Bei der Vererbung besteht manchmal die Notwendigkeit, sich explizit auf die Oberklasse zu beziehen – meistens ist dies bei Konstruktoren der Fall
- Einfaches Beispiel

```
class Rectangle {
    private double height,
                   width;

    Rectangle(double height, double width) {  // Konstruktor
        this.height = height;
        this.width  = width;
    }
    ...  // weitere Variablen und Methoden
}  // end class Rectangle
```

```
class Square extends Rectangle {
    Square(double sideLength) {  // Konstruktor
        super(sideLength, sideLength);
    }
    ...  // weitere Variablen und Methoden
}  // end class Square
```

Erläuterung: Die Seitenlängen eines Quadrats sind identisch, ansonsten sind es normale Rechtecke (damit können alle Methoden geerbt werden)

- Für die Verwendung von **`super`** als Konstruktor gelten folgende Restriktionen:
  - In einer Unterklasse darf der Konstruktor der Oberklasse selbst nicht verwendet werden, d.h. eine Aufruf `Rectangle(sideLength, sideLength)` wäre illegal; der Konstruktor darf nur über **`super`** angesprochen werden
  - Der Konstruktor **`super`**`(...)` kann nur als erste Anweisung im Konstruktor der Unterklasse verwendet werden

```
class GraphicalDice extends PairOfDice {
    public GraphicalDice() {  // Konstruktor fuer diese Klasse
        super(3, 4);           // Aufruf des Konstruktors der PairOfDice-Klasse
        initializeGraphics();  // weitere spezielle Initialisierungen ...
    }
    ...
}  // end class GraphicalDice
```

- Verwendung von **`super`** nicht als Konstruktor – namensgleiche Attribute

```
class Parent {
    int x = 3;
    ...  // weitere Variablen und Methoden
}  // end class Parent
```

```
class Child extends Parent {
    float x = 1.2f;
    ...
    float foo() {
        return this.x + super.x;  // liefert hier den Wert 4.2
    }
    ...
}  // end class Child
```

Überladen von Methoden

Überladen

- Sind Name und Rückgabewert zweier Methoden identisch, die Parametrisierung aber verschieden, spricht man vom Überladen von Methoden
- Das Überladen von Methoden kann sowohl zwischen Super- und Unterklasse als auch innerhalb einer Klasse stattfinden

```
public class Person {
    String name,
           address;

    public float computeIO() {...}
    ...
}   // end class Person

public class Student extends Person {
    int matriculationNumber;

    public float computeIO(int factor) {...}

    public float computeIO(boolean key, int factor) {...}
    ...
}   // end class Student
```

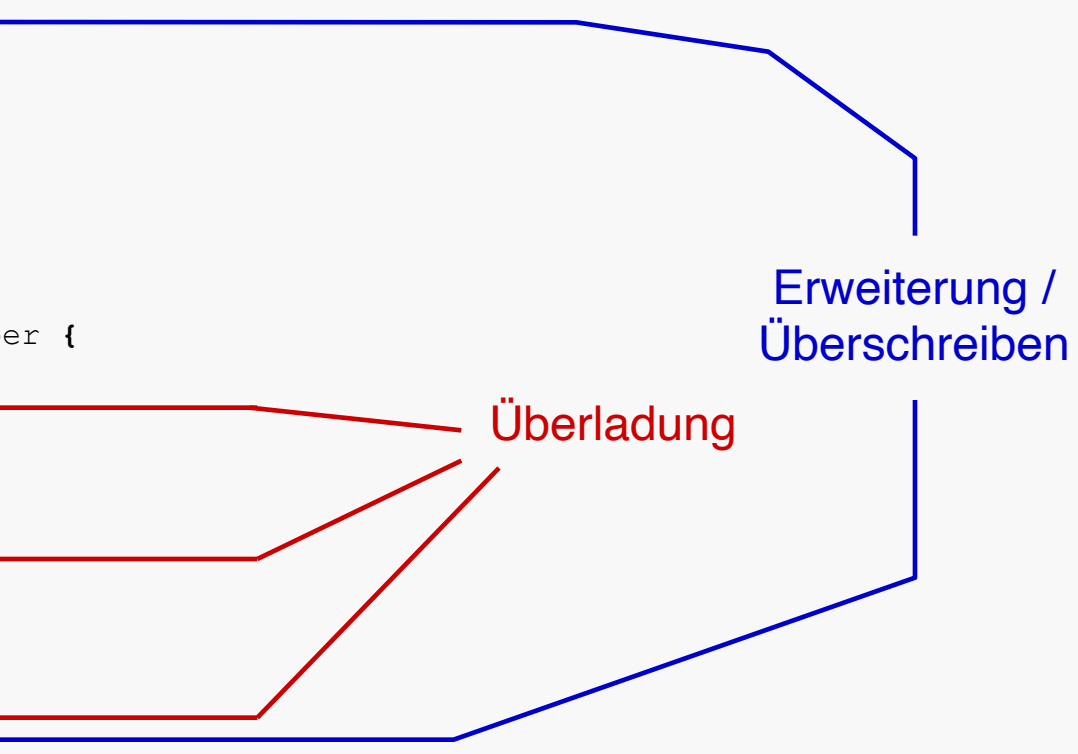
Beispiele für das **Überladen** von Methoden

Kommentar: Gleiche Namen und unterschiedlicher Rückgabetyp sind verboten!

Überschreiben vs. Überladen von Methoden

- Beispiel

```
public class Upper {
    ...
    int bFun(boolean x) {
        if (x)
            return 5;
        else
            return 6;
    }
}  // end class Upper

public class Lower extends Upper {
    ...
    int bFun(String s) {
        return s.length();
    }

    int bFun(int g) {
        return 2 * g;
    }

    int bFun(boolean y) {
        if (y)
            return -27;
        else
            return -90;
    }
}  // end class Person
```



- Überladen (*overloading*)
  - Es können mehrere Methoden gleichen Namens nebeneinander existieren, die sich in ihrer Parametrisierung unterscheiden

```
int bFun(String s) { ... }
int bFun(int g) { ... }
int bFun(boolean y) { ... }
```

  - Welche Methode aufgerufen wird, entscheidet sich zur *Compile*-Zeit anhand der deklarierten Datentypen der Parameter

- Überschreiben (*overriding*)
  - In einer Unterklasse darf eine Methode mit dem gleichen Namen und derselben Parameterliste definiert werden wie sie bereits in der Oberklasse vorhanden ist; sie überschreibt dann die in der Oberklasse definierte Methode – realisiert die Anpassung von Methoden / Eigenschaften bei der Vererbung

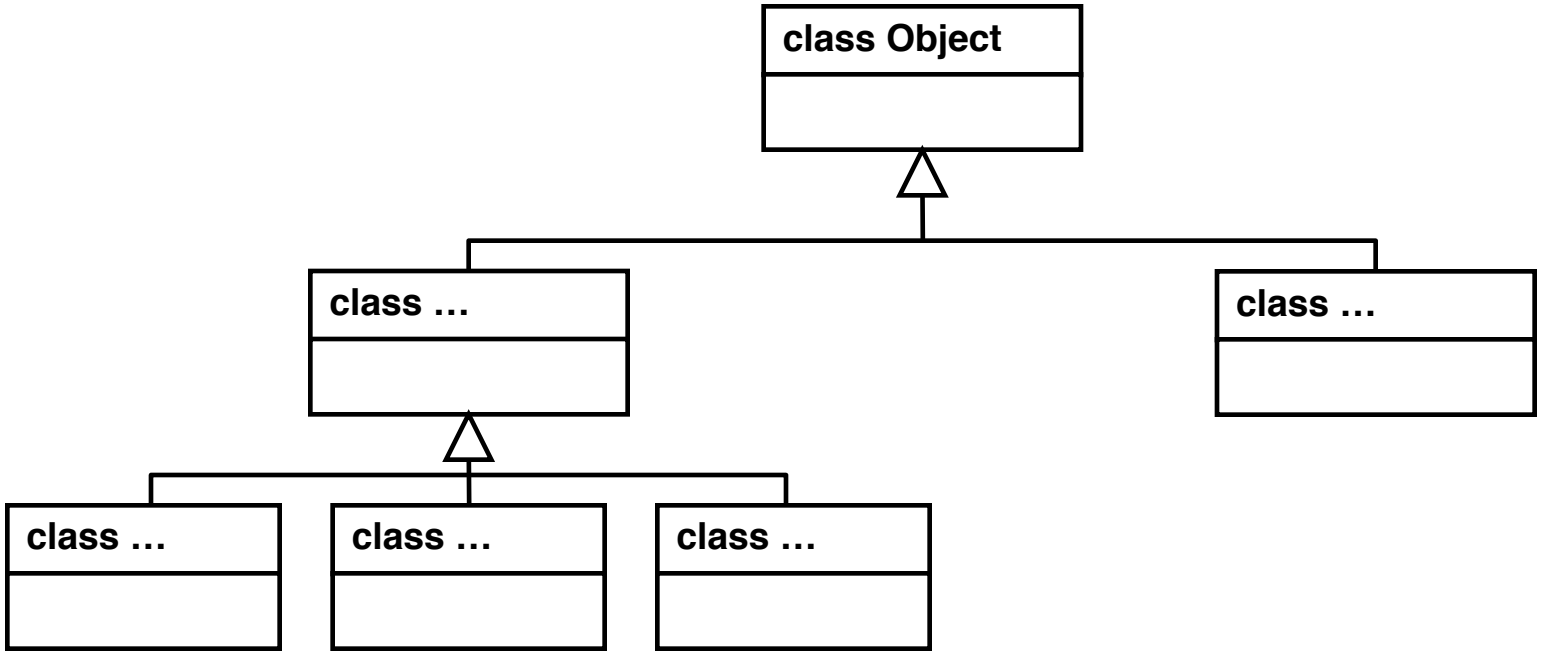
```
public class Upper {
    int bFun(boolean x) { ... }
}

public class Lower extends Upper {
    int bFun(boolean y) { ... }
}
```

  - Wird eine solche Methode über ein Objekt zugegriffen, so wird zur Laufzeit die konkrete Methode dynamisch gebunden

Die Enden der Vererbungskette – `Object` und `final`

Die ultimative Oberklasse `Object`

- Die gesamte Vererbungshierarchie in Java hängt letztendlich unter einer einzigen Oberklasse, `Object`



Erläuterung: Die Klasse `Object` ist in Java vordefiniert und enthält eine Reihe von Methoden, die allgemein hilfreich sein können, beispielsweise

```
public boolean equals(Object other) {...}
```
und
```
public String toString() {...}
```

- Aufrufe der Methoden haben folgende Funktion:
  - `a.equals(b)` liefert **`true`**, wenn die Objekte `a` und `b` „äquivalent“, d.h. inhaltlich gleich, sind
  - `a.toString()` liefert eine String-Darstellung des Objekts `a`

Erläuterung: Die Methoden werden zwar von allen anderen Klassen geerbt, sollten aber in der Praxis in jeder Klasse redefiniert werden (funktionelle Anpassung der Funktion)

Die untersten Klassen `final`

- In dem Vererbungsbaum jedes Programms gibt es am untersten Ende als Blätter Klassen, zu denen keine weiteren Unterklassen mehr gebildet wurden
- Als Programmierer einer Klasse kann man erzwingen, dass es keine weiteren Unterklassen mehr geben kann

Bsp.:
```
final class SecurityMonitor {  // hochkritischer Sicherheitsmonitor
    ...
}
```

Erläuterung: Somit ist garantiert, dass niemand eine weitere Unterklasse dieser Klasse bilden kann; dadurch wird verhindert, dass z.B. durch Viren-Programme mittels modifizierter Vererbung ein Programm umformuliert werden kann

- Soll nicht eine ganze Klasse endgültig gemacht werden, so können auch einzelne Methoden geschützt werden

Bsp.:
```
class SecurityMonitor {  // hochkritischer Sicherheitsmonitor
    ...
    final boolean checkIdentity(UserId uid, Password pwd) {
        ...
    }
    ...
}
```

- Gründe für die Abschirmung von Methoden gegen Modifikationen
  - **Sicherheit**: Schutz gegen Modifikationen einzelner Methoden in Java-Programmen, die oft über das Internet geladen werden können; Angriffe erfolgen häufig über gezielte Unterklassenbildung, die wiederum Programme so modifizieren, dass Sicherheitsmechanismen durchbrochen werden
  - **Entwurf**: In vielen Softwaresystemen ist es wichtig zu wissen, dass bestimmte Komponenten endgültig sind – und damit das Verhalten verlässlich ist

- Konstanten **`final`**: Konstanten (in den primitiven Datentypen) werden in Java ebenfalls über den **`final`**-Mechanismus realisiert

Bsp.:
```
    final float GRAVITY = 9.81f;
```

Objekterzeugung bei erweiterten Klassendefinitionen

Erweiterte Objekterzeugung

- Bisher wurden Objekte (einer Klasse) mittels eines Konstruktors erzeugt; es kann verschiedene Konstruktoren in einer Klasse geben, wird kein Konstruktor angegeben, so wird automatisch ein *Default*-Konstruktor ohne Parameter definiert
- Bei **erweiterten Klassendefinitionen** müssen die Konstruktoren der Ober- und Unterklassen aufeinander abgestimmt werden

Bsp.:

```
public class Person {
    String name,
           address;

    public Person(String name, String address) {
        this.name    = name;
        this.address = address;
    }
}  // end class Person

public class Student extends Person {
    int matriculationNumber;

    public Student(String name, String address, int matrNo) {
        super(name, address);
        this.matriculationNumber = matrNo;
    }
}  // end class Student
```

Konstruktor der Klasse `Person`, der die Anlage von Personenobjekten regelt

Konstruktor der Klasse `Student`, der den Konstruktor der Oberklasse aufruft und so die Anlage von Studentenobjekten regelt

- Konstruktoren von Unterklassen müssen als erstes einen Konstruktor der Oberklasse aufrufen
- Ist kein expliziter Konstruktoraufruf angegeben, so wird automatisch **`super()`** ausgeführt
- Alternativ kann auch ein anderer Konstruktor (vgl. **`this`**) der aktuellen Klasse aufgerufen werden, der seinerseits den Konstruktor der Oberklasse ruft

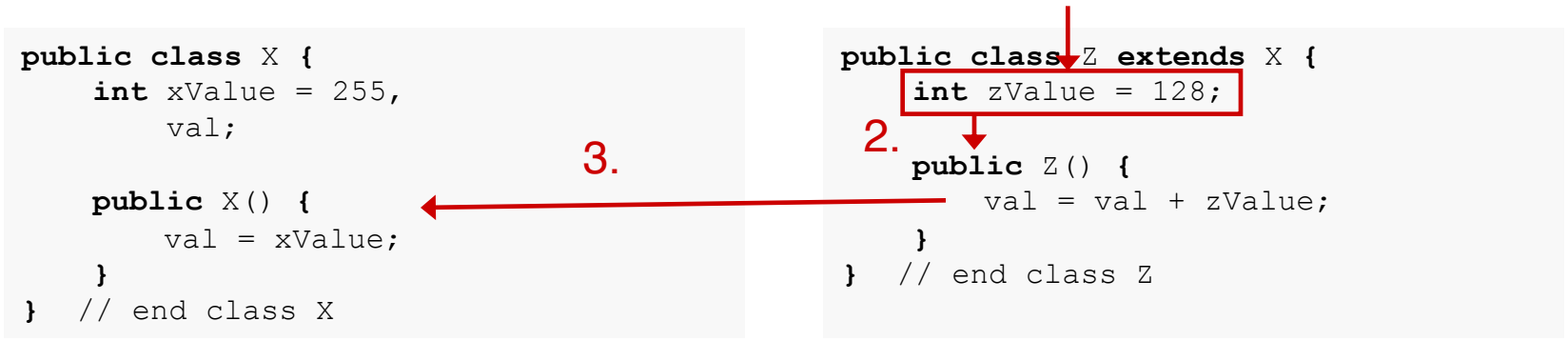
Bsp.: Noch einmal die Unterklasse `Student` (von S.45) mit zwei Konstruktoren

```
public class Student extends Person {
    int matriculationNumber;

    public Student(String name, String address, int matrNo) {
        super(name, address);
        this.matriculationNumber = matrNo;
    }

    public Student() {
        this("Standardstudent", "Ulm", 123456);
    }
}  // end class Student
```

Reihenfolge der Aktionen bei der Objekterzeugung

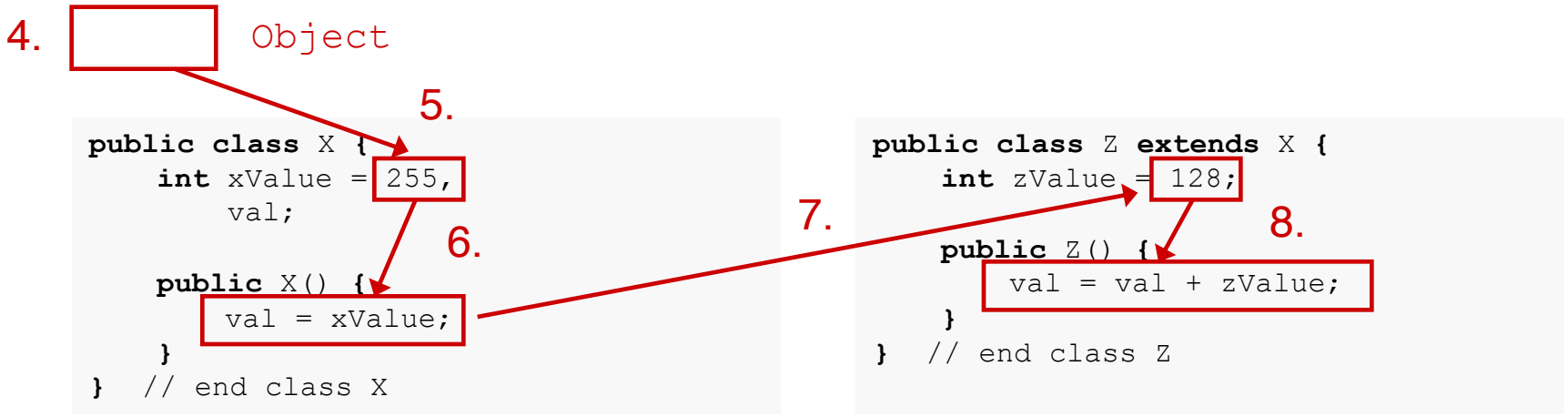
Ablauf und die Einzelschritte

- Beispiel mit zwei Klassen



Einordnung: Insgesamt werden beim Aufruf von `new Z()` acht Schritte ausgeführt

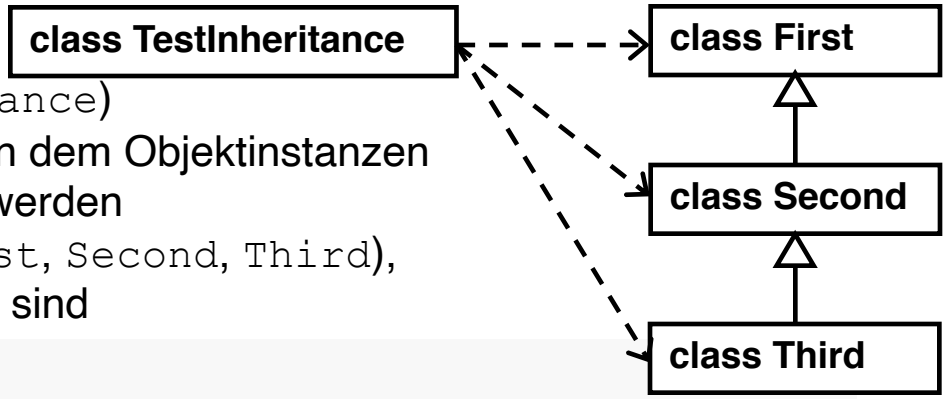
- Schritte 1-3
  1. Speicherplatz für das Objekt vom Typ `Z` wird bereitgestellt; allen Objektvariablen werden Standardwerte zugewiesen (z.B. 0 für eine `int`-Variable)
  2. Die aktuellen Parameter des Konstruktoraufrufs von `Z()` (hier leere Parameterliste) werden ausgewertet und der entsprechende Konstruktor von `Z` wird aufgerufen
  3. Da im Konstruktor `Z()` keine expliziten Aufrufe `this()` bzw. `super()` auftreten, wird implizit `super()` und damit der Konstruktor `X()` aufgerufen



- Schritte 4-8
  4. Da im Konstruktor `X()` keine expliziten Aufrufe **`this`**`()` bzw. **`super`**`()` auftreten und `X` nur noch Unterklasse von `Object` ist, wird implizit **`super`**`()` und damit der Konstruktor `Object()` aufgerufen; dieser erledigt für das neue Objekt allgemeine Verwaltungsaufgaben im Java-System
  5. Initialisierung der Variablen von `X` (hier: `xValue` mit 255)
  6. Der Rest des Rumpfes des Konstruktors von `X` wird ausgeführt
  7. Explizit angegebene Initialisierungen der Variablen von `Z` (hier: `zValue = 128`) werden ausgeführt
  8. Der Rest des Rumpfes des Konstruktors von `Z` wird ausgeführt; Initialisierung des neuen Objekts vom Typ `Z` ist damit beendet

Konkretes Beispiel und seine Implementierung (Demo: TestInheritance.java)

- Struktur und Java-Programm
  - In einer Hilfsklasse (`TestInheritance`) wird ein Hauptprogramm definiert, in dem Objektinstanzen unterschiedlicher Klasse generiert werden
  - Drei Klassen werden definiert (`First`, `Second`, `Third`), die hierarchische Spezialisierungen sind




```
class TestInheritance {
      public static void main(String[] args) {
            final int FIRST  = 1;
            final int SECOND = 2;
            final int THIRD  = 3;
            First  firstClassObject1,
                   firstClassObject2;
            Second secondClassObject1,
                   secondClassObject2,
                   secondClassObject3;
            Third  thirdClassObject1;

            firstClassObject1  = new First(FIRST);
            firstClassObject1.displayStatus();

            secondClassObject1 = new Second(SECOND);
            secondClassObject1.displayStatus();

            secondClassObject2 = new Second(SECOND);
            secondClassObject2.displayStatus();

            thirdClassObject1  = new Third(THIRD);
            thirdClassObject1.displayStatus();

            secondClassObject3 = new Second(SECOND);
            secondClassObject3.displayStatus();

            firstClassObject2  = new First(FIRST);
            firstClassObject2.displayStatus();
      }  // end main
}  // end class TestInheritance
```

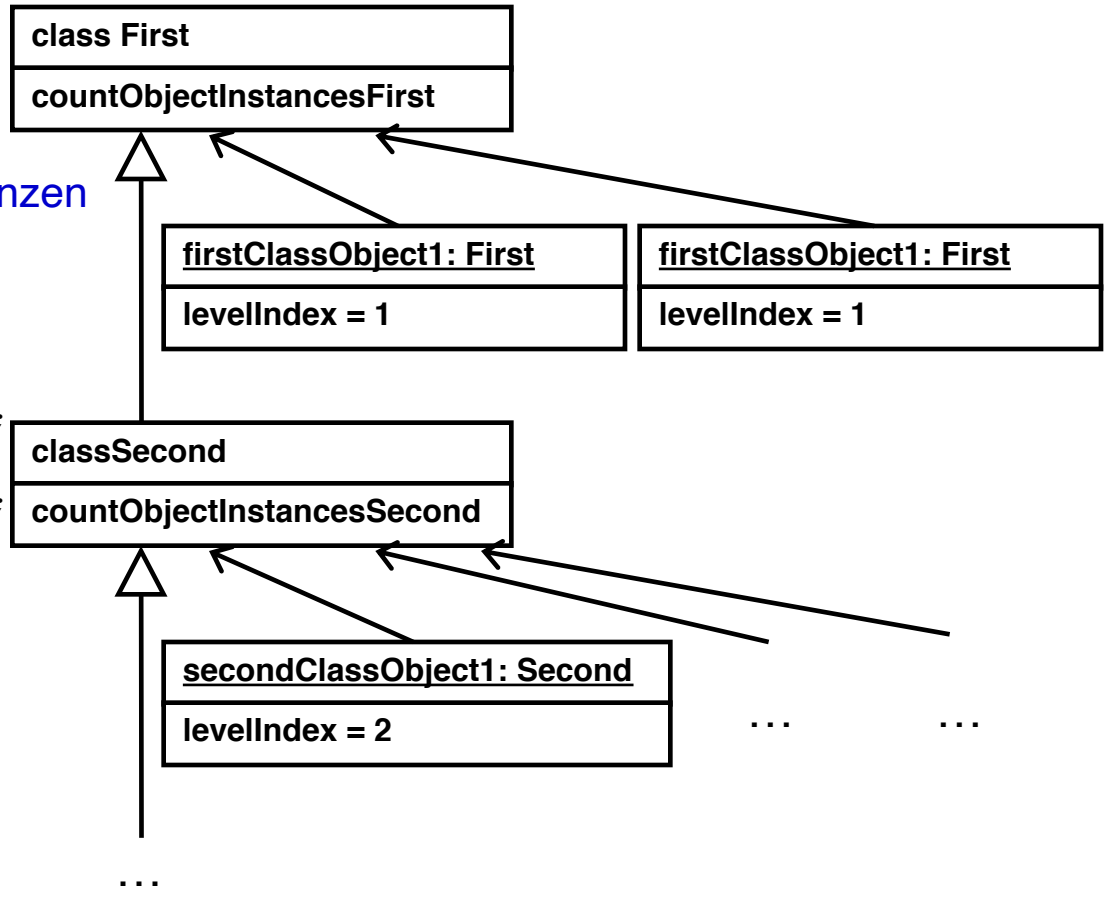
... <Fortsetzung>

```
      ... <class TestInheritance>

class First {
      private String     nameFirst;
      private int        levelIndex;
      private static int countObjectInstancesFirst;

      public First(int index) {  // Konstruktor
            this.nameFirst  = "first level";
            this.levelIndex = index;
            this.countObjectInstancesFirst++;
      }

      public int getLevelIndex() {
            return this.levelIndex;
      }

      public int getObjectInstancesFirst() {
            return this.countObjectInstancesFirst;
      }

      public void displayStatus() {
            System.out.print("object First class (idx: " + getLevelIndex() + ") ");
            System.out.print("is " + getObjectInstancesFirst() + ". object instance");
            System.out.println();
      }
}  // end class First


class Second extends First {
      private String     nameSecond;
      private int        levelIndex;
      private static int countObjectInstancesSecond;

      public Second(int index) {  // Konstruktor
            super(1);
            this.nameSecond = "second level";
            this.levelIndex = index;
            this.countObjectInstancesSecond++;
      }

      public int getLevelIndex() {  // Methode ueberdeckt die Methode aus 'First'
            return this.levelIndex;
      }

      ... <Fortsetzung>
```

```
      ... <class TestInheritance>
      ... <class First>
      ... <class Second Extends First>

      public int getObjectInstancesSecond() {
            return this.countObjectInstancesSecond;
      }

      public void displayStatus() {  // Methode ueberdeckt die Methode aus 'First'
            System.out.print("object Second class derived from First class (idx: " +
                        getLevelIndex() + ") ");
            System.out.print("is " + getObjectInstancesSecond() + ". object instance");
            System.out.println();
      }
}  // end class Second


class Third extends Second {
      private String     nameThird;
      private int        levelIndex;
      private static int countObjectInstancesThird;

      public Third(int index) {  // Konstruktor
            super(2);
            this.nameThird  = "third level";
            this.levelIndex = index;
            this.countObjectInstancesThird++;
      }

      public int getLevelIndex() {  // Methode ueberdeckt die Methode aus 'Second'
            return this.levelIndex;
      }

      public int getObjectInstancesThird() {
            return this.countObjectInstancesThird;
      }

      public void displayStatus() {  // Methode ueberdeckt die Methode aus 'Second'
            System.out.print("object Third class derived from Second class (idx: " +
                        getLevelIndex() + ") ");
            System.out.print("is " + getObjectInstancesThird() + ". object instance");
            System.out.println();
      }
}  // end class Third
```

- Erläuterung
  - Generierung von Objektinstanzen (nur für 2 Ebenen gezeigt)

```
... = new First(FIRST);

... = new Second(SECOND);

... = new Second(SECOND);

        ⋮

... = new First(FIRST);
```



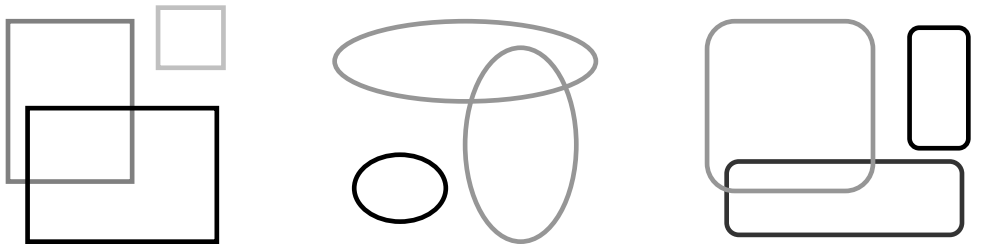
  - In den Klassenvariablen `countObjectInstancesXXX` werden die instanziierten Objekte gezählt … **Ergebnis** (beachte die Zähler für die drei Klassen)

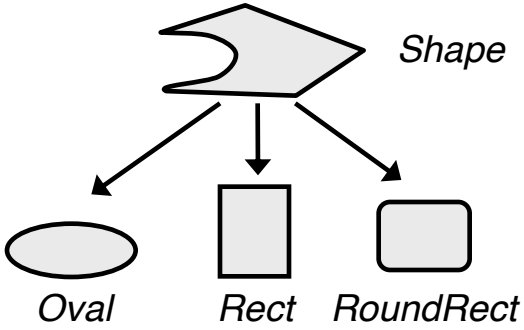
```
C:\Dokumente und Einstellungen\hneumann\Eigene Dateien\PROGRAMS\JAVA\JavaWorkObj
ectsClasses>java TestInheritance
object First class (idx: 1) is 1. object instance
object Second class derived from First class (idx: 2) is 1. object instance
object Second class derived from First class (idx: 2) is 2. object instance
object Third class derived from Second class (idx: 3) is 1. object instance
object Second class derived from First class (idx: 2) is 4. object instance
object First class (idx: 1) is 6. object instance

C:\Dokumente und Einstellungen\hneumann\Eigene Dateien\PROGRAMS\JAVA\JavaWorkObj
ectsClasses>
```

Polymorphie und abstrakte Klassen

Klassenhierarchien und Methodenaufrufe

- Beispiel-Klassen für Rechtecke (`Rectangle`), Ovale (`Oval`) und abgerundete Rechtecke (`RoundRect`) und eine Oberklasse `Shape`





- Beispiel-Implementierung in Java

```
class Shape {
    Color color;  // Farbe als Objekt mit alpha und Farbwert im sRGB Standard
                  // 'Color' ist im Package java.awt definiert

    void setColor(Color newColor) {
        this.color = newColor;
        redraw();   // zeichne ein Objekt in der neuen Farbe
    }

    void redraw() {  // oben aufgerufene Methode zum Zeichnen der Form
        ???  // welche Kommandos muessen hier stehen?
    }
    ...      // weitere Instanz-Variablen/-Methoden
}  // end class Vehicle
```

Erläuterung: Jede Form (Rechteck, Oval, …) wird anders ausgegeben, die Oberklasse kann diese Details nicht kennen (!); wenn die Methode zur Ausgabe (`redraw()`) in der Oberklasse aufgerufen wird, dann muss sie auch in dieser Klasse (oder in einer weiteren höheren Oberklasse) deklariert sein

- **Lösung**: Jede spezifische (Unter-) Klasse hat ihre eigene `redraw()`-Methode

```
class Rectangle extends Shape {
    ...
    void redraw() {  // oben aufgerufene Methode zum Zeichnen der Form
        ...  // Anweisungen zum Zeichnen eines Rechtecks
    }
    ...      // weitere Instanz-Variablen/-Methoden
}  // end class Vehicle
```

Gleiches gilt für

```
class Oval extends Shape {
    ...
}

class RoundRect extends Shape {
    ...
}
```

Polymorphie

Formalismus

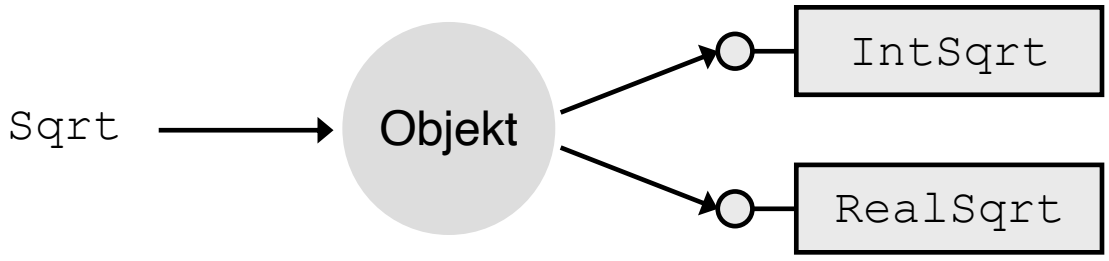
- Eine Variable `oneShape` vom Typ `Shape` (aus dem Beispiel) kann von jedem Untertyp `Rectangle`, `Oval`, `RoundRect` sein; der Aufruf

  ```
  oneShape.redraw();
  ```

  führt zum Aufruf derjenigen `redraw`-Methode, die für das Objekt geeignet ist

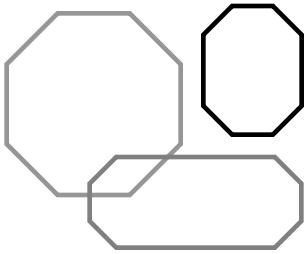
- Der Wert von `oneShape` kann sich während der Laufzeit ändern, z.B. in einer Schleife kann die Referenzvariable auf Objekte unterschiedlicher Untertypen zeigen; der jeweilige Aufruf von `redraw()` bewirkt dann den jeweiligen Aufruf der Methode aus der Unterklasse – die Methode ist **polymorph**

  Erläuterungen: Polymorphie bezeichnet die Eigenschaft, wenn die ausgeführte Aktion vom aktuellen Typ des referenzierten Objekts abhängt; die Situation entspricht der dynamischen Bindung von Methoden (S.32ff), jedoch wurden dort Modifikationen von vererbten Methoden betrachtet



Erweiterbarkeit, Implementierung und Ausführung

- Es soll eine weitere (Unter-) Klasse `BeleveledRects` definiert werden



- Die Implementierung enthält eine eigene `redraw()`-Methode
- Der Aufruf der `redraw()`-Methode in der Oberklasse `Shape` (als Teil der Methode `setColor(...)`) bleibt bei seiner Definition erhalten
- Realisierung und Ausführung:
  - Eine Instanz-Methode ist in jedem Objekt enthalten
  - Die Klasse enthält den Quell-Code der Methode
  - Die `setColor(...)`-Methode ist in der Klasse `Shape` implementiert, die `redraw()`-Methode ist in der jeweiligen Unterklasse enthalten; d.h. trotz der unterschiedlichen Klassen in denen der Quellcode enthalten ist, sind die Methoden Teil desselben Objekts

Abstrakte Klassen

Abstraktion

- *Wie wird die* `redraw()` *-Methode in der Klasse* `Shape` *definiert?*

  Antwort: Sie bleibt leer! Die Klasse `Shape` enthält nur die abstrakte Beschreibung einer geometrischen Form, daher kann sie keine konkrete Formen zeichnen – allerdings muss ein Hinweis auf die Methode enthalten sein (als Platzhalter dient der Verweis)

- Die Deklaration von `redraw()` in der Klasse `Shape` wird selbst nie direkt ausgeführt
- In einem Programm können (Zeiger-) Variablen vom Typ `Shape` deklariert werden; die konstruierten Objekte sind jedoch vom Typ der Unterklassen (hier: `Rectangle`, `Oval`, ...)

Einordnung: Die Klasse `Shape` ist eine **abstrakte Klasse**; von dieser werden keine eigenen Objekte instanziiert, sie dient ausschließlich als Basis für die Definition neuer Unterklassen – sie existiert daher nur, um die gemeinsamen Eigenschaften aller ihrer Unterklassen auszudrücken

- **Definition (Abstrakte Klasse)**:

  Eine abstrakte Methode ist eine Methode ohne Implementierung, d.h. sie hat keinen Rumpf. Die Spezifikation erfolgt durch

  ```
  abstract <Typ> <Name>(<Parameter>);
  ```

  Eine abstrakte Klasse ist eine Klasse, die als abstrakt definiert wurde. Die Spezifikationen erfolgt durch

  ```
  abstract class <Name> { <Rumpf der Klasse> }
  ```

  Sowohl abstrakte Methoden als auch abstrakte Klassen werden durch das Schlüsselwort **`abstract`** gekennzeichnet.

  Hinweis:

  - Bei den abstrakten Methoden fehlen auch die Block-Klammern **`{ . . . }`** für den Rumpf – dies bedeutet, dass sie keinen Rumpf haben und nicht nur einen leeren Rumpf!
  - Der Hauptgrund für die Verwendung ist, dass auf der entsprechenden Abstraktionsebene für die Methoden noch keine konkrete Implementierung angegeben werden kann

- Eine abstrakte Methode, z.B. `redraw()`, existiert nur für die Spezifikation der Schnittstelle der aktuellen, konkreten Versionen von `redraw()` in den Unterklassen; die abstrakte `redraw()`-Methode in `Shape` enthält daher keinen Code
- Dem Compiler kann mittels des Modizierers **`abstract`** bereits auf syntaktischer Ebene mitgeteilt werden, dass eine Methode nur abstrakt ist
- Abstrakte Klasse `Shape` in Java

```
public abstract class Shape {
    Color color;  // Farbe der allgemeinen Form

    void setColor(Color newColor) {
        this.color = newColor;
        redraw();             // zeichne ein Objekt in der neuen Farbe
    }

    abstract void redraw();  // abstrakte Methode - muss in den konkreten
                             // Unterklassen definiert werden

    ...      // weitere Instanz-Variablen/-Methoden
}  // end class Shape
```

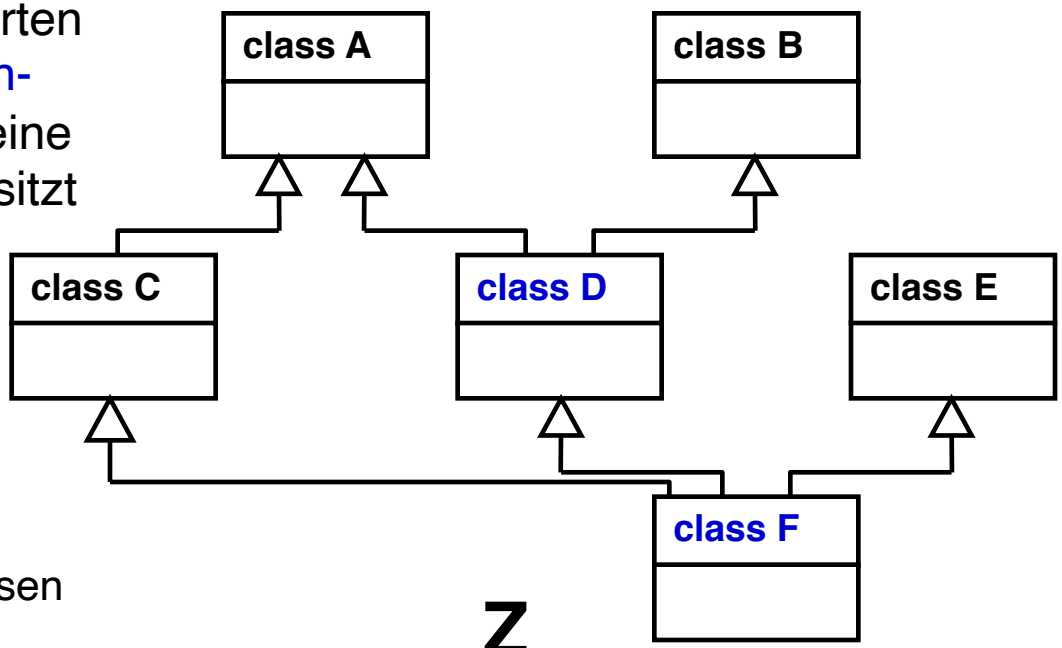
3. Schnittstellenbeschreibungen – *Interfaces*

- Mehrfachvererbung und Schnittstellen
- Anwendung – Suchen und Sortieren allgemein

Mehrfachvererbung und Schnittstellen (*Interfaces*)

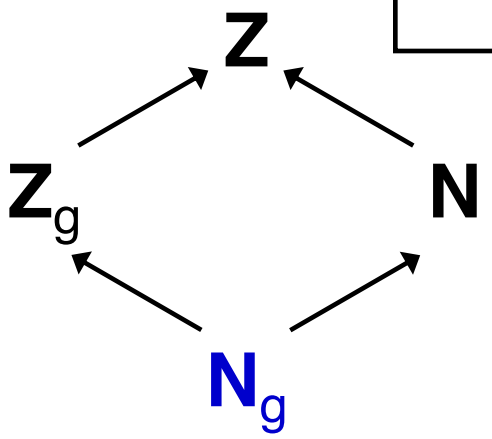
Mehrfachvererbung (*multiple inheritance*)

- Ein Problem in der objektorientierten Programmierung ist die Mehrfach-vererbung, in Situationen wenn eine Klasse mehrere Oberklassen besitzt



In dem Beispiel hat die Klasse D 2 direkte Oberklassen A und B, die Klasse F hat 3 Oberklassen C, D und E – d.h. die Klassen D und F erben von mehreren Oberklassen

- Eine entsprechende Situation war mit dem Typ der geraden natürlichen Zahlen $\mathbf{N}_g$ gegeben, der aus der Überlappung von **N** und $\mathbf{Z}_g$ hervorgegangen ist



- Allgemein machen solche Überlappungen keine Probleme; schwierig wird es, wenn in zwei solchen Oberklassen (von denen eine Klasse gemeinsam erbt) zwei Methoden mit den gleichen Namen deklariert werden

Bsp.:

```
class Car {
    ...
    void driving() {...}
}

class Boat {
    ...
    void driving() {...}
}

class AmphibianVehicle extends Car, Boat {
    ...
    void driving() {...}
}
```

Geht in Java nicht!

Einordnung: In diesem Beispiel ist unklar, welche der beiden Definitionen von `driving()` gemeint ist – auch die Hilfsmittel **`super`**`()` helfen hier nicht mehr weiter!

- Für dieses grundlegende Problem gibt es unterschiedliche Lösungsansätze (z.B. in C++); in Java wird ein rigoroser Weg gewählt: Mehrfachvererbungen sind in Java verboten!

Schnittstellen (*Interfaces*)

Definition und Verwendung

- Das grundlegende Problem der Mehrfachvererbungen ist aber durchaus sinnvoll, es taucht in der Praxis auch häufiger auf; daher bietet Java eine Ersatzlösung in Form der *Interfaces* an

- **Definition (*Interface*)**:

  Ein ***Interface*** ist eine Sammlung von Methodenköpfen ohne deren Rümpfe. Zusätzlich können noch einige Konstanten enthalten sein. Eine Schnittstelle wird wie folgt definiert:

```
interface <Name> {
    <Methoden-Koepfe>; <Konstanten>;
}
```

  Interfaces werden durch Klassen **implementiert**. Dazu muss die Klasse jeweils die im *Interface* geforderten Methoden realisieren:

```
class <Name_Klasse> implements <Name_Interface> {
    ...
}
```

  Für diejenigen Methoden der Klasse, die die *Interface*-Methoden realisieren, gilt die Zusatzbedingung, dass diese als **`public`** gekennzeichnet sein müssen.

- *Interfaces* dienen als reine Schnittstellenbeschreibungen und sagen nichts über die zugehörigen Implementierungen aus; sie stellen nur Aufforderungen an ihre Implementierungen dar, gewisse Methoden verfügbar zu machen

Bsp.:

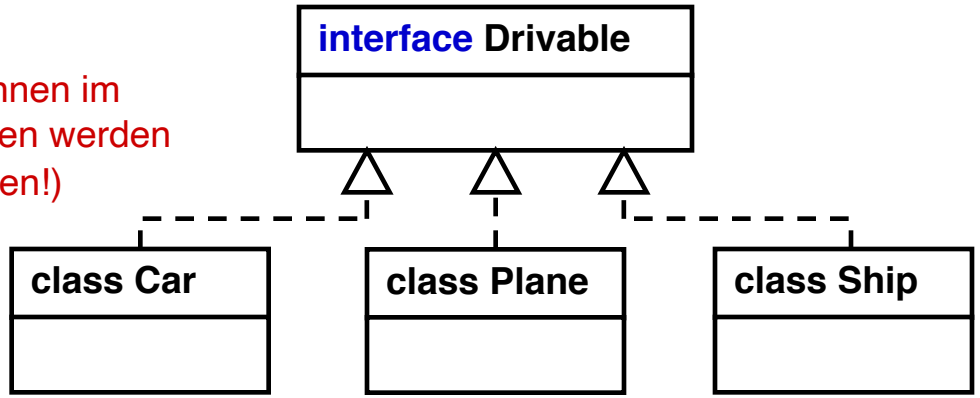
```
public interface Drawable {
    public void draw(Graphics g);
}

public class Line implements Drawable {
    public void draw(Graphics g) {
        ...  // Anweisungen zum Zeichnen von Linien
    }
    ...  // andere Methoden und Variablen
}
```

Bemerkung: Das Format des Methodenkopfes mit den Bezeichnern seiner Parameter muss zwischen *Interface* und Implementierung vollständig übereinstimmen

- Es ist unmöglich, aus *Interfaces* mithilfe des **`new`**-Operators Objekte zu generieren (wie bei den abstrakten Klassen)
- Interfaces können auch in einer Ableitungshierarchie stehen: *MyInterface2* extends *MyInterface1*

Einsatzmöglichkeiten von *Interfaces*

- *Interfaces* sind in verschiedenen Situationen nützlich
  - Klassen können zusammengefasst werden – mit sehr unterschiedlichen Implementierungstechniken, die dem gleichen Zweck dienen
  - Von einer Klasse kann die Schnittstelle bekannt gegeben werden, ohne ihre Implementierung offen legen zu müssen
- Beispiel – Ein *Interface* mit mehreren Implementierungen



`public`-Modifizierer können im `interface` weg gelassen werden (wird *default* angenommen!)

Entwurf:

```
interface Drivable {
   boolean start();              // starte Motor (erfolgreich?)
   void stop();                  // stoppe Motor
   void accelerate(float acc);   // beschleunige
   void turn(int degree);        // drehen
}  // end interface Drivable
```

Programmierung:

```
class Car implements Drivable {
    float currentSpeed = 0;
    ...
    public boolean start() {...}

    public void stop() {...}

    public void accelerate(float acc) {...}

    public void turn(int degree) {...}
}
```

- Verwendung

```
    ...
    Drivable d;
    Car      c = new Car();

    d = c;
    boolean success = c.start();
    if (success) {
        d.turn(10);
        d.stop();
        ...
    }
```

- Eine Klasse kann nur eine andere Klasse erweitern (mit Vererbung), sie kann jedoch mehrere *Interfaces* implementieren; eine Klasse kann gleichzeitig eine Klasse erweitern und mehrere Interfaces implementieren

```
class FilledCircle extends Circle
                        implements Drawable, Fillable {
    ...
}
```

Zusammenfassung

Überblick zum Vergleich von Klassen, abstrakten Klassen und *Interfaces*

	verwendbar zur Typisierung	erlaubt Vererbung	erlaubt Mehrfachvererbung	verwendbar in **new**
Klassen	☺	☺		☺
Abstrakte Klassen	☺	☺		
Interfaces	☺	☺	☺ (*)	

(*) Klassen können mehrere *Interfaces* implementieren!

Anwendung – Suchen und Sortieren allgemein

Interface `Sortable`

- Im letzten Kapitel wurde gezeigt, wie mittels Generics und eines durch ein Factory Methode **`Comparator`**.reverseOrder() erstellten Objekts eine **`ArrayList`** sortiert werden kann. Comparator ist dabei ein Interface, das unterschiedlich implementiert werden kann. Hier erfolgt eine eigene Definition eines solchen Interfaces.
- **Ziel**: Realisierung von Such- und Sortier-Algorithmen auf *Arrays* beliebiger Daten – ohne dass es jeweils eine eigene Methode (z.B. für *insert*) für jede Art von *Array* gibt

Bsp.:

```
public interface Sortable {
    boolean le(Sortable other);  // kleiner-gleich
    boolean lt(Sortable other);  // kleiner
    boolean eq(Sortable other);  // gleich
}
```

Auf der Basis des *Interfaces* `Sortable` kann ein allgemeiner Sortieralgorithmus programmiert werden:

```
public void insertElem(Sortable[] arr, int w) {
    for (int i = w; i >= 1; i--) {
        if (arr[i-1].le(arr[i]))
            break;
        swap(arr, i-1, i);
    }
}  // end insertElem
```

- Klassen, deren Objekte sortiert werden sollen, müssen als Implementierungen von `Sortable` deklariert werden

Bsp.:

```
public class Customer implements Sortable {
    private int coopNo;

    public boolean le(Sortable other) {
        return (this.coopNo <= ((Customer)other).coopNo);
    }

    public boolean lt(Sortable other) {
        return (this.coopNo < ((Customer)other).coopNo);
    }

    public boolean eq(Sortable other) {
        return (this.coopNo == ((Customer)other).coopNo);
    }
}
```

Bemerkungen:

- Die Methoden des *Interfaces* `Sortable` werden hier implementiert, wobei jede Methode einen Rumpf mit entsprechendem Block `{...}` besitzt
- Die Signatur der Methoden muss bei ihrer Implementierung identisch übernommen werden, daher muss aus dem Datentyp `Sortable` mittels des *casting*-Operators eine explizite Konvertierung in `Customer` durchgeführt werden

Java-*Interface* `Comparable` und `Comparator`

- In den Java-Bibliotheken sind die Bedürfnisse, die mit `Sortable` gelöst wurden, auch berücksichtigt worden
- Im *Package* `java.lang` gibt es ein *Interface* `Comparable`, das im Wesentlichen die Idee des *Interfaces* `Sortable` realisiert;

  anstatt dreier Operationen `le`, `lt`, und `eq` wird hier nur eine Operation `compareTo()` bereit gestellt, die – je nach Größe der beiden Objekte – die Werte -1, 0, 1 liefert
- Die Idee, bei unterschiedlichen Sortierkriterien die Vergleichoperation als zusätzliches Argument mitzugeben, wird in Java auch realisiert:

  Im *Package* `java.util` gibt es das *Interface* `Comparator`, das die Methode `compare()` vorsieht, die wiederum ähnlich wie `compareTo` arbeitet