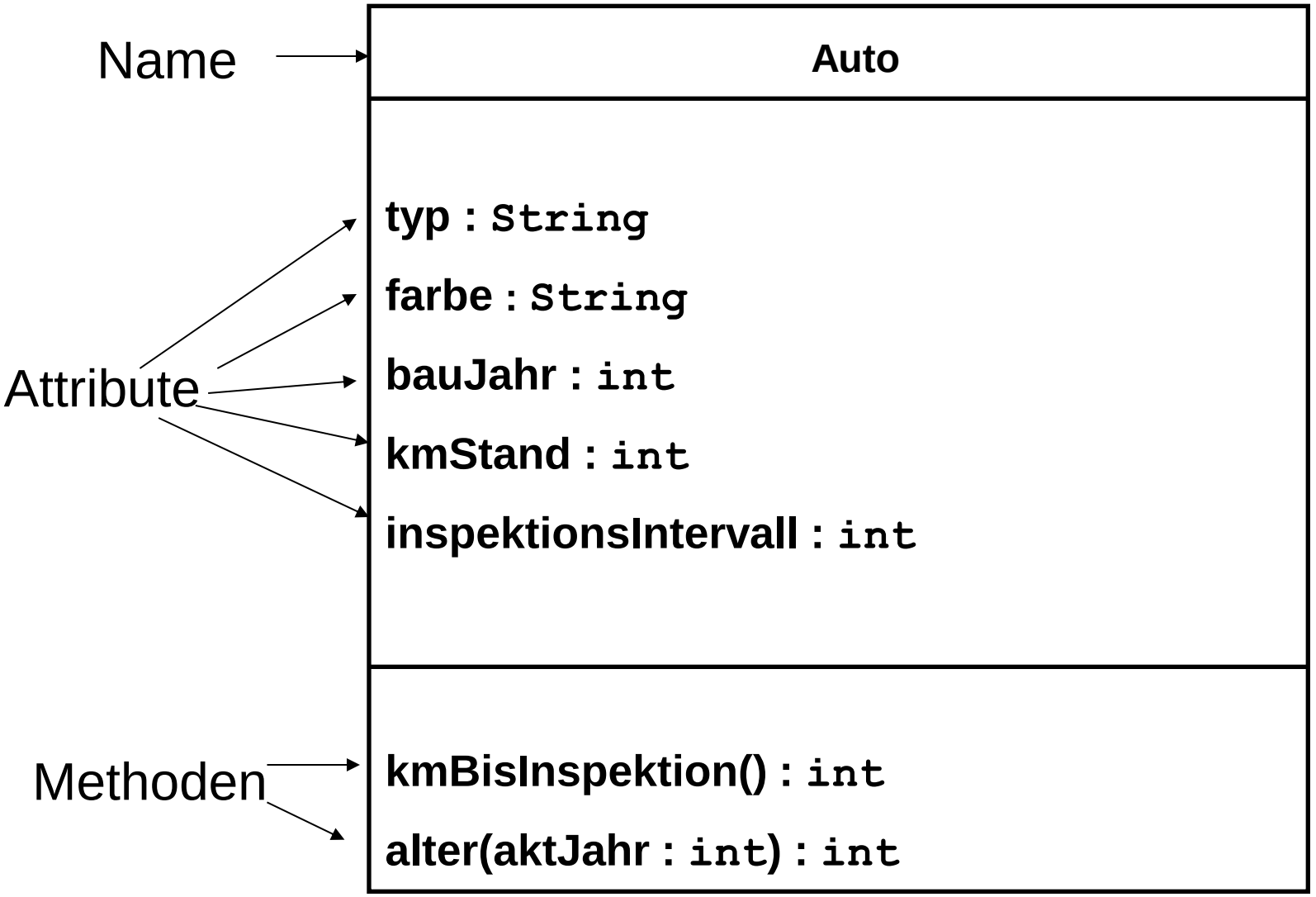
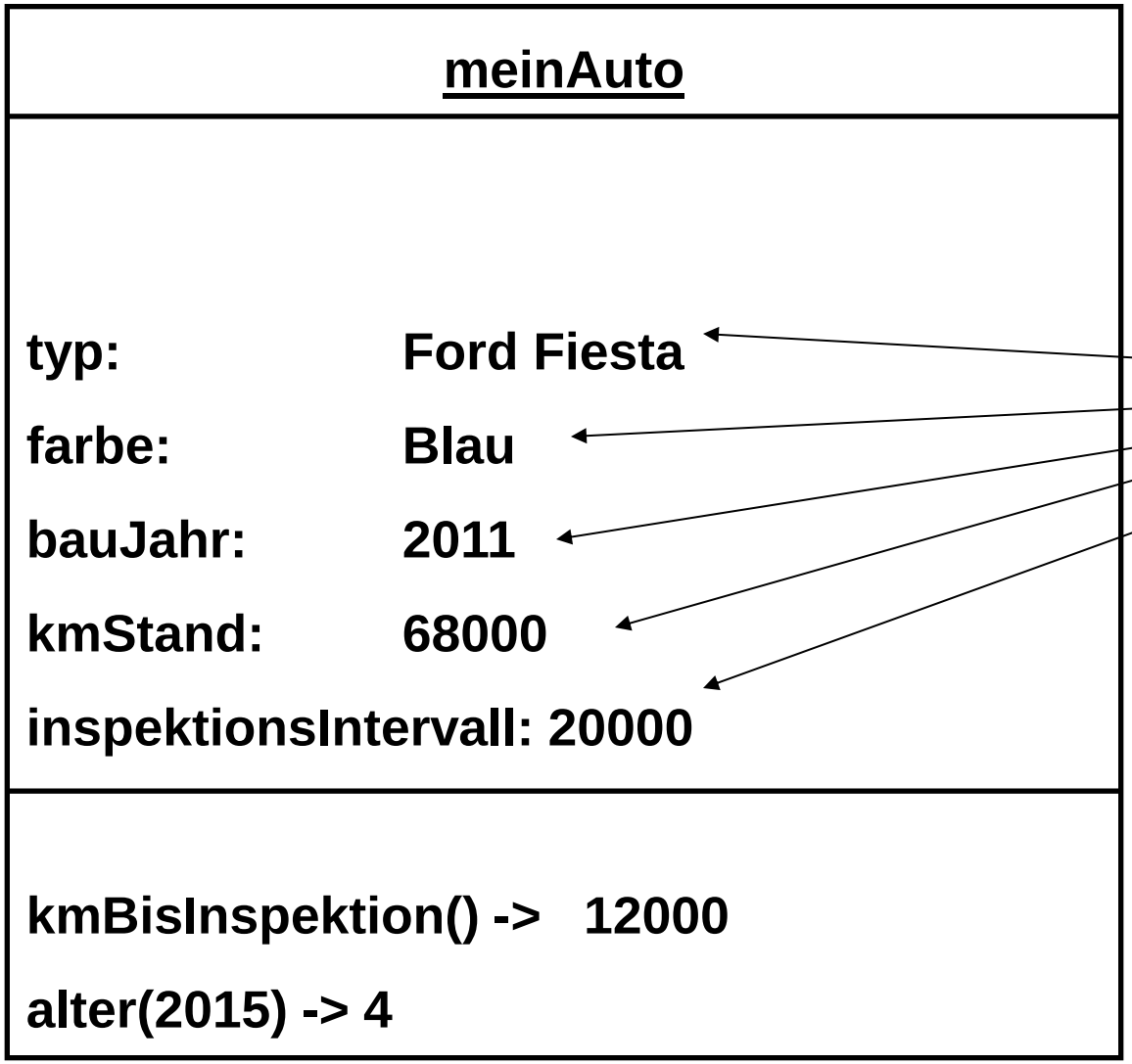
Klasse in UML



Klasse in Java

```
//Klassendefinition:
class Auto {
//Attributdeklarationen:
  String typ;
  String farbe;
  int    bauJahr;
  int    kmStand;
  int    inspektionsIntervall;
//Methodenimplementierungen:
  int kmBisInspektion() {
    return (inspektionsIntervall - (kmStand %
                inspektionsIntervall));
  }
  int alter(int aktJahr) {
    return (aktJahr - bauJahr);
  }
}
```

Objekt in UML



Objekt-ID
Referenz

Attributwerte

=

Zustand

Verhalten

Bsp. für call-by-value

```
// neue Methode in Klasse Auto
void printFarbe(int wieoft) {
  while (wieoft-- > 0) {
    System.out.println("Farbe = " + farbe);
  }
}

// Bsp. für Aufrufe von meinAuto.farbe:
int a = 3;
meinAuto.printFarbe(a);
meinAuto.printFarbe(a);
meinAuto.printFarbe(a);
```

Wie oft wird die Farbe ausgegeben?

Bsp. für call-by-reference

```
class Bsp {

     void ändereFarbe(Auto einAuto) {

       einAuto.farbe = "rot";
}
```

```
// Bsp. für Aufruf von ändereFarbe in main:
Auto meinAuto = new Auto();
meinAuto.farbe = "blau";
Bsp bsp = new Bsp();
bsp.ändereFarbe(meinAuto);
System.out.println(meinAuto.farbe);
```

Welche Farbe wird ausgegeben?

mehrere Konstruktoren in einer Klasse

```
Auto(String farbe) {
   this.farbe = farbe;
}

Auto(String farbe, int bauJahr) {
   this.farbe = farbe;
   this.bauJahr = bauJahr;
}
```

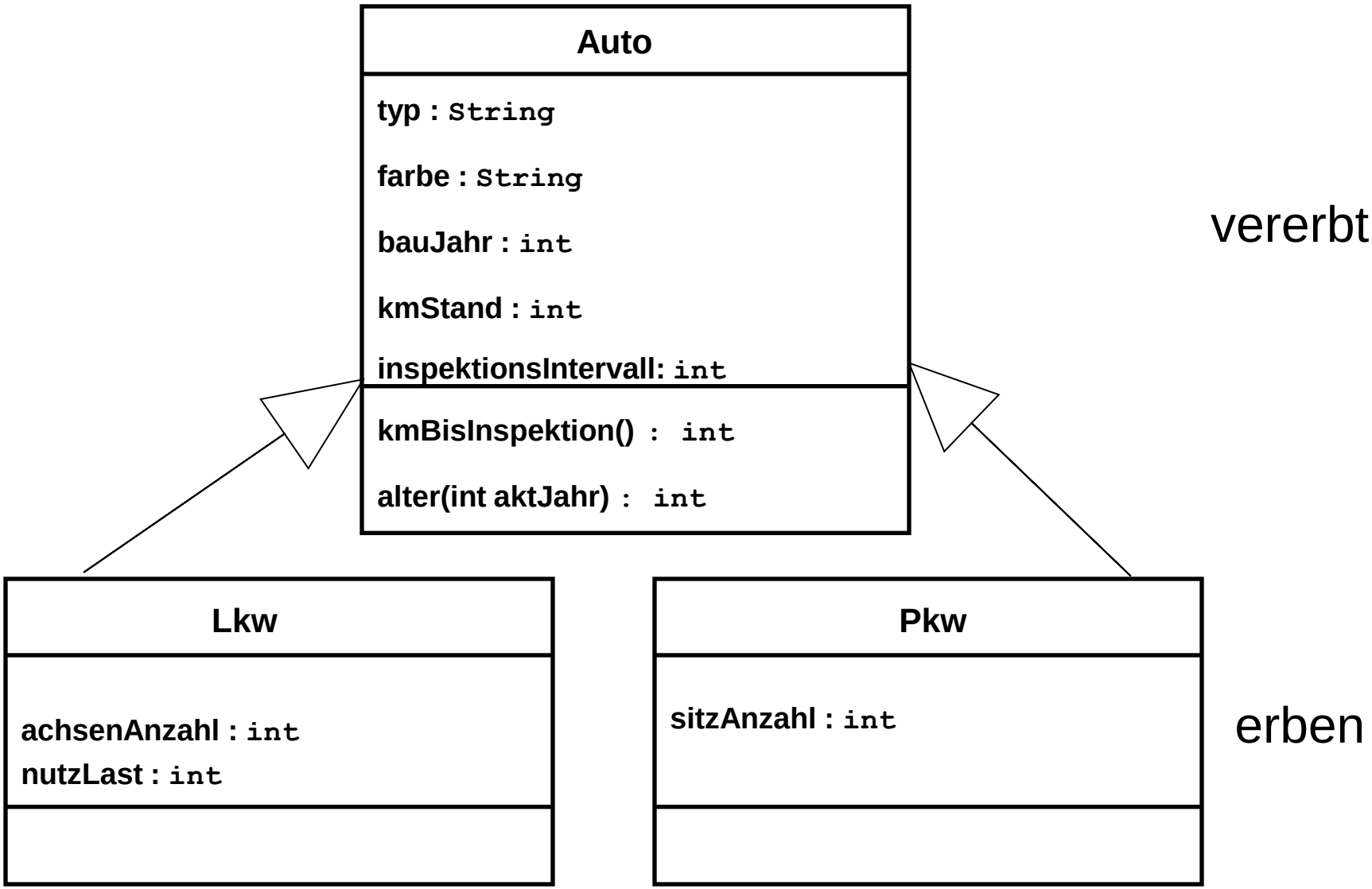
Überladung auch bei Methoden möglich

Wie sieht Konstruktor aus, der alle Attribute initialisiert?

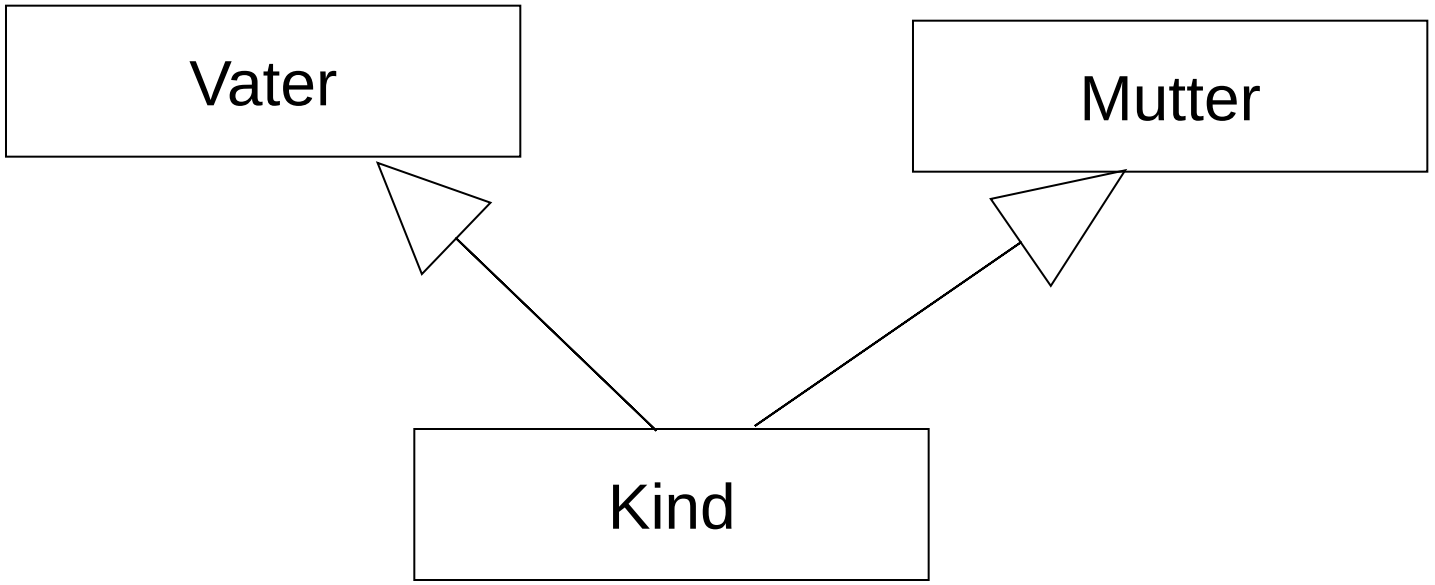
Verkettung von Konstruktoren

```
public Auto(String farbe) {
  this.farbe = farbe;
}

public Auto(String farbe, int bauJahr) {
  this(farbe);
  this.bauJahr = bauJahr;
}
```

Vererbung in UML



Mehrfachvererbung



Gibt es in Java nicht!!!

Simulation durch Interfaces -> später

Vererbung in Java

```
class Lkw extends Auto {
  int achsenAnzahl;
  int nutzLast;
}

class Pkw extends Auto {
  int sitzAnzahl;
}

//in main:
Lkw lkw = new Lkw();
lkw.achsenAnzahl = 3;
//aber auch Attribute von Auto verwendbar:
lkw.kmStand = 10000;
lkw.inspektionsIntervall = 15000;
//auch Methoden von Auto verwendbar:
System.out.println(lkw.kmBisInspektion());
```

Zuweisungskompatibilität in Param.

```
public class AutoFarben {
  String farbe (Auto auto) {
    return (auto.farbe);
  }

  public static void main(String[] args) {
    Lkw lkw = new Lkw();
    lkw.farbe = "grau";
    AutoFarben autoFarben = new AutoFarben();
    System.out.println(autoFarben.farbe(lkw));
  }
}
```

Vergleiche

Überlagern von Methoden (Polymorph.)

```
//Klassendefinition:
class Lkw extends Auto {
//neue Attribute:
  int achsenAnzahl;
  int nutzLast;
//Methodenüberlagerung alter jetzt in Monaten:
  int alter(int aktJahr) {
    return (12*(aktJahr - baujahr));
  }
}

// in main:
Lkw lkw = new Lkw();
lkw.baujahr = 1999
System.out.println(lkw.alter(2003)); // -> 48
```

Methode equals für Klasse Auto

```
boolean equals(Auto auto) {
   return (this.typ.equals(auto.typ) &&
           this.farbe.equals(auto.farbe) &&
           this.baujahr == auto.baujahr &&
           this.kmStand == auto.kmStand &&
           this.inspektionsIntervall ==
                auto.inspektionsIntervall);
  }
```

Was würde passieren, wenn man für typ und farbe auch == verwenden würde?

Referenzen würden überprüft und nicht Inhalte

-> **bei Objekten immer equals benutzen und nicht ==**

Methode toString für Klasse Auto

```
public String toString() {
  return (  " typ: " + this.typ.toString() +
                  " farbe: " + this.farbe.toString() +
                  " bauJahr: " + this.bauJahr +
                  " kmStand: " + this.kmStand +
                  " inspektionsIntervall: " +
                  this.inspektionsIntervall);
}

//in main:
Auto auto1 = new Auto("Fiesta","blau",2011,68000,20000);
System.out.println(auto1);
```

```
typ: Fiesta farbe: blau baujahr: 2011 kmStand: 68000
inspektionsIntervall: 20000
```

Ausführreihenfolge von Konstruktoren

definiere in Klasse `Auto` Konstruktor wie folgt:

```
Auto() {
  System.out.println("Konstruktor von Auto!");
}
```

definiere in Klasse `Lkw` Konstruktor wie folgt:

```
Lkw() {
  System.out.println("Konstruktor von Lkw!");
}
```

Ausgabe bei Aufruf von new `Lkw()` ist:

Konstruktor von Auto!
Konstruktor von Lkw!

Es wird die gesamte Vererbungshierarchie beginnend bei der Wurzel durchlaufen

Klasse Angestellter in Java

```
import java.util.*;

class Angestellter {
  int persNr;
  String name;
  Date eintritt;                    //aus java.util.*
  double grundGehalt;
  double ortsZuschlag;
  double zulage;

  double monatsBrutto() {
    return (grundGehalt +
            ortsZuschlag +
            zulage);
  }
}
```

Klasse Manager in Java

```
import java.util.*;

class Manager {
  int persNr;
  String name;
  Date eintritt;                          //aus java.util.*
  double fixGehalt;
  double provision1;
  double provision2;
  double umsatz1;
  double umsatz2;

  double monatsBrutto() {
    return ( fixGehalt +
             umsatz1*provision1/100 +
                   umsatz2*provision2/100);
  }
}
```

abgeleitete Klasse Angestellter in Java

```
class Angestellter  extends Mitarbeiter{

   int persNr;
   String name;
   Date eintritt;

   double grundGehalt;
   double ortsZuschlag;
   double zulage;

   double monatsBrutto() {
     return (grundGehalt +
             ortsZuschlag +
             zulage);
   }
}
```

Hier weglassen, da schon in Klasse **`Mitarbeiter`** deklariert

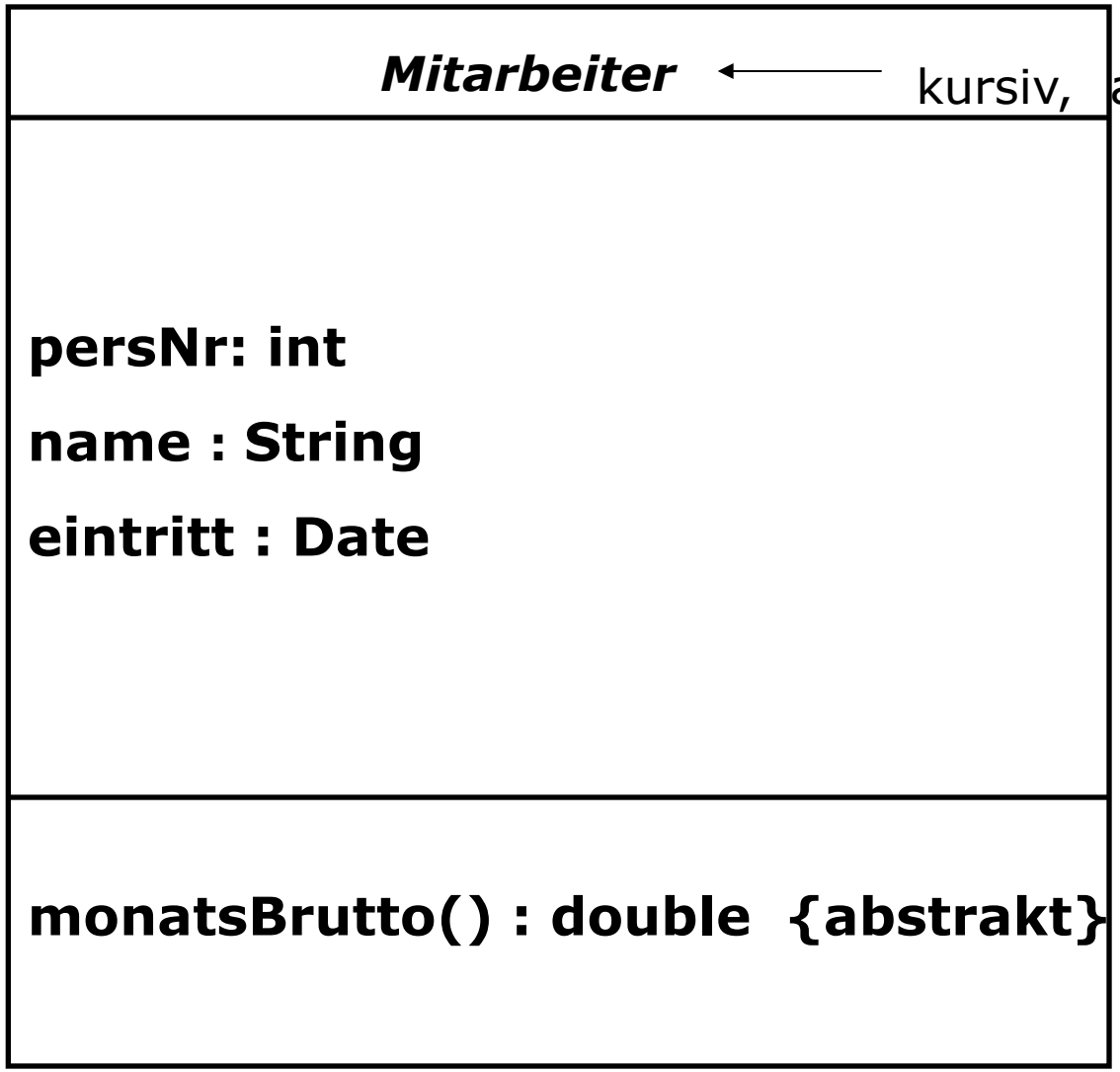
Analog für die Klasse **`Manager`**

abstrakte Klasse Mitarbeiter in Java

```
abstract class Mitarbeiter {

//gemeinsame Attribute der drei Unterklassen:
  int persNr;
  String name;
  Date eintritt;

//abstrakte Methode:
  abstract double monatsBrutto();
}
```

keine Implementierung der Methode

abstrakte Klasse Mitarbeiter in UML



kursiv, alternativ zu {abstrakt}

tiefere Ableitung

```
// Manager der Geschäftsführung:
class GFManager extends Manager {

  double gfzulage;

  double monatsBrutto() {
    return (super.monatsBrutto() +
            gfzulage);
  }
}
```

Methode der Oberklasse **Manager**

Berechnung des Gesamtbruttos

```
class GehaltsBerechnung {
  public static void main(String[] args){
    Mitarbeiter[] ma = new Mitarbeiter[100];
//Mitarbeiter-Array füllen, z.B.
//ma[0] = new Manager();
//ma[1] = new Angestellter();
//ma[2] = new GFManager();
   ...
//Bruttosumme berechnen
    double bruttosumme = 0.0;
    for (int i=0; i<100; ++i) {
      bruttosumme += ma[i].monatsBrutto();
    }
    System.out.println("Bruttosumme = "+bruttosumme);
  }
}
```

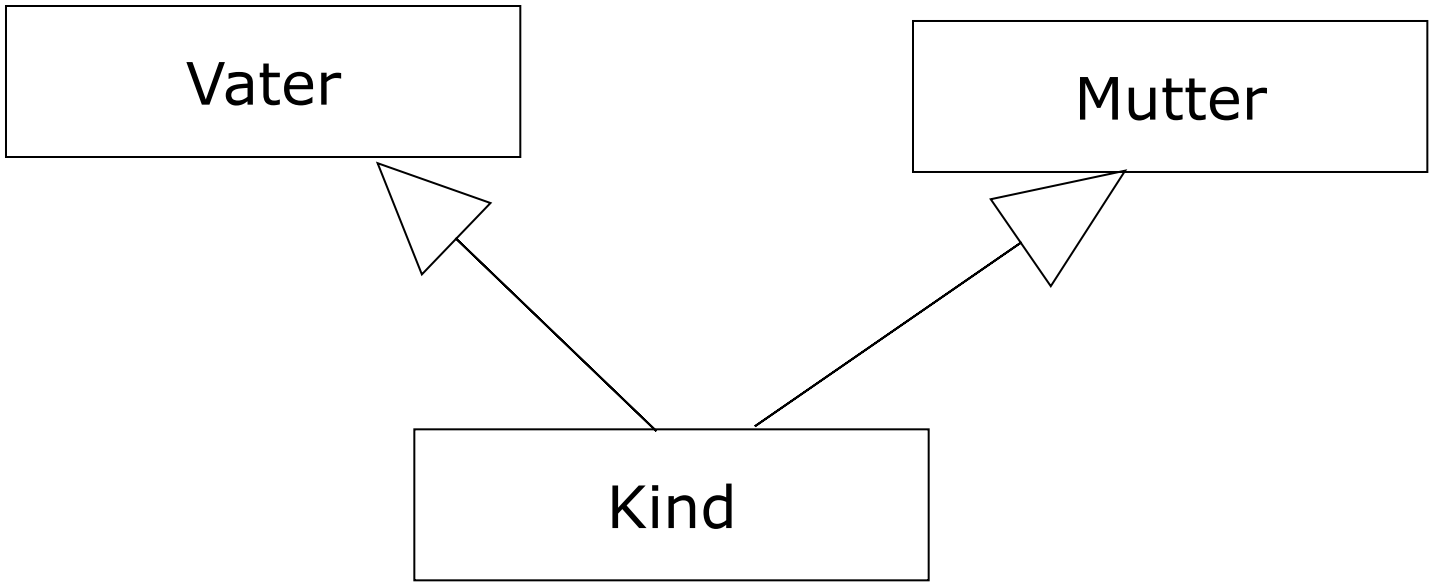
Hier wird jeweils die Implementierung der entsprechenden Unterklasse aufgerufen!

Zugriffspezifikationen

Java **UML**	**Sichtbarkeit bei anderer zugreifender Klasse**	**in Unterklasse vor-handen, falls diese**
public +	Überall	beliebige Position hat
protected #	Innerhalb des Pakets	beliebige Position hat
~ "friendly"	Innerhalb des Pakets	innerhalb des Pakets liegt
private -	Nein	Nein

echte Hierarchie & innerhalb Klasse alles aus Klasse sichtbar

Mehrfachvererbung



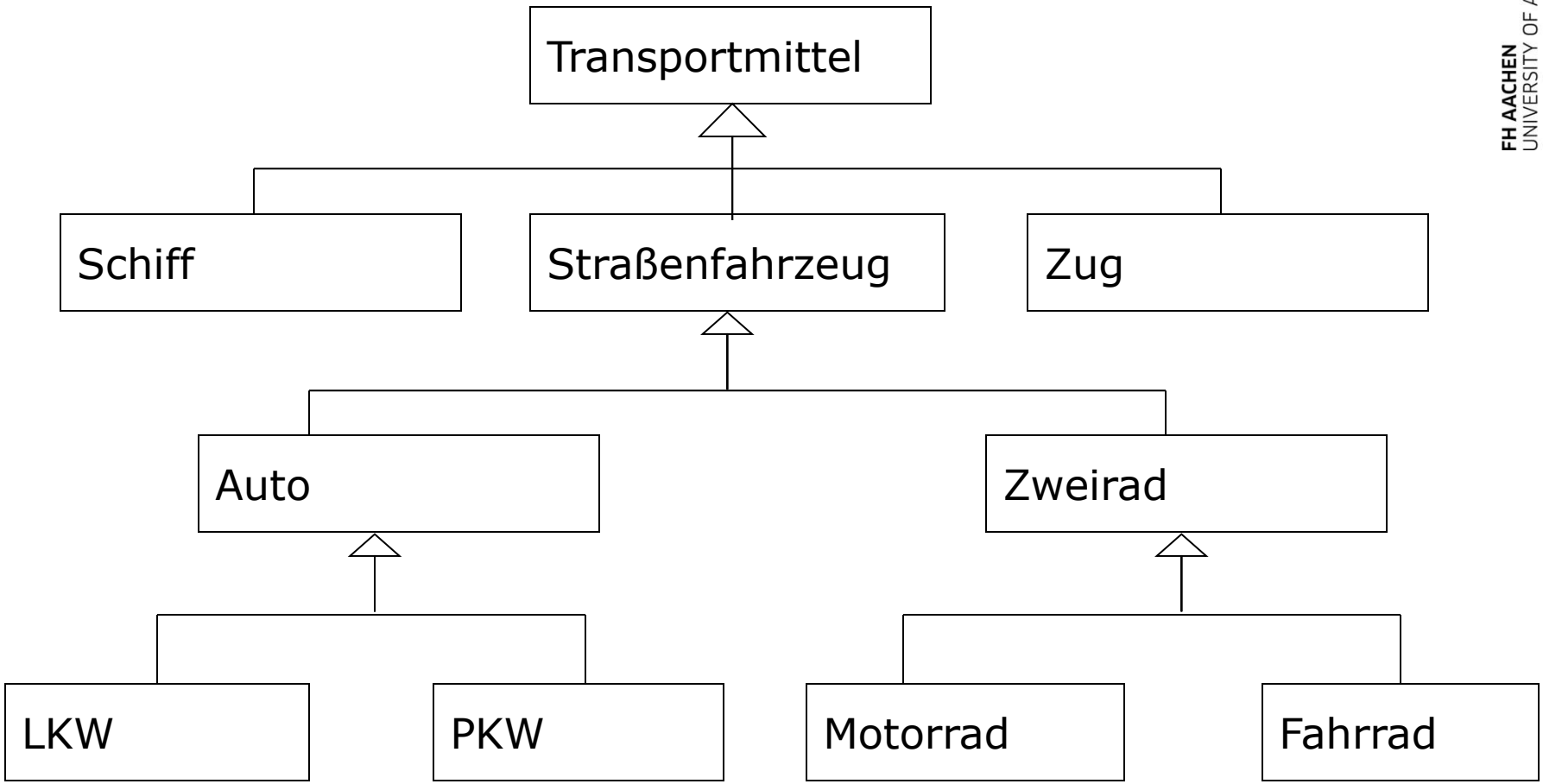
Gibt es in Java nicht!!!

Simulation durch Interfaces

Problem bei Mehrfachvererbung

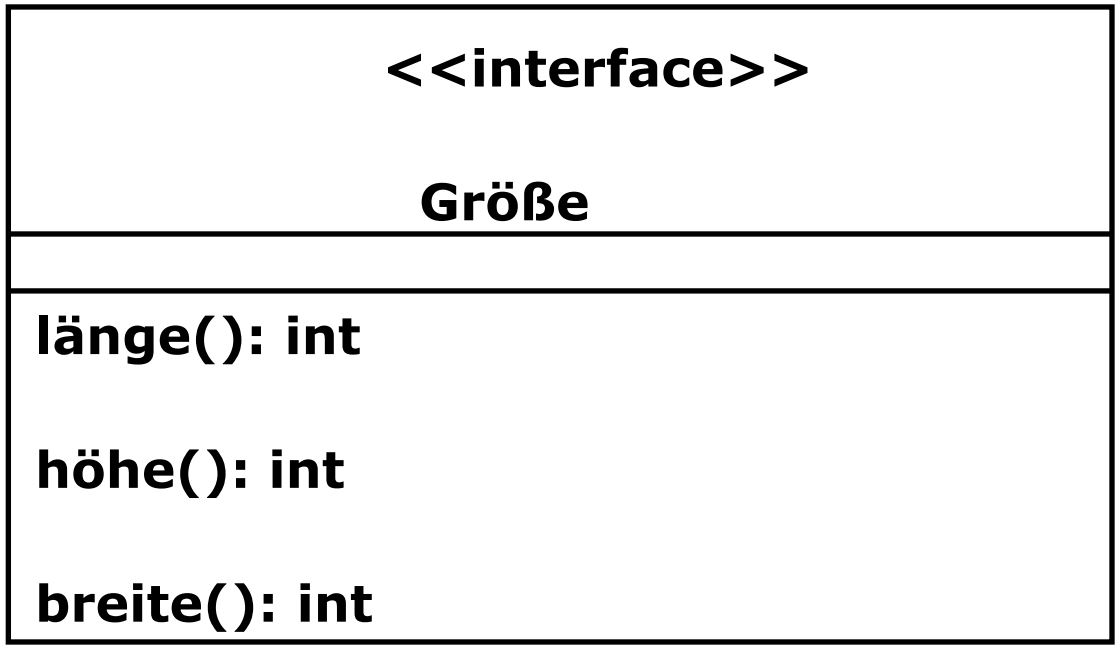
```
class Vater {
  int geburtsJahr;
  int vorgegebenesAlter(int aktJahr) {
    return (aktJahr - geburtsJahr);
  }
}

class Mutter {
  int geburtsJahr;
  int vorgegebenesAlter(int aktJahr) {
    return (aktJahr - geburtsJahr - 5);
  }
}
```

Vererbungshierarchien in Java



immer ein **Baum**

Interface in UML



Interface in Java

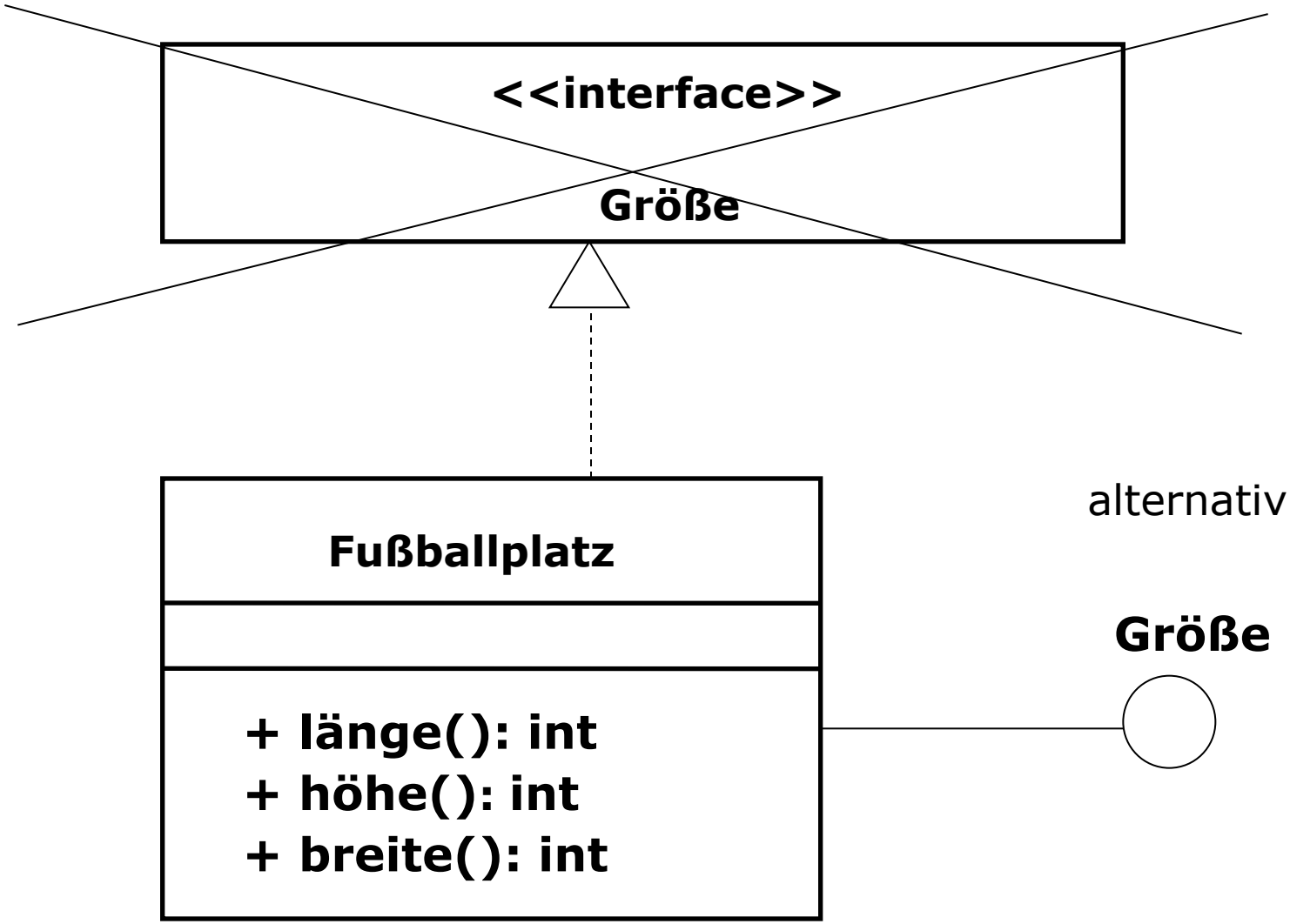
```
public interface Größe {
   public int länge();
   public int höhe();
   public int breite();
}
```

Methodendeklarationen, keine Implementierung

Prinzip des **Information Hidings**:
Es wird nur bekannt gegeben, welche Methoden zur Verfügung gestellt werden, nicht wie sie implementiert werden.

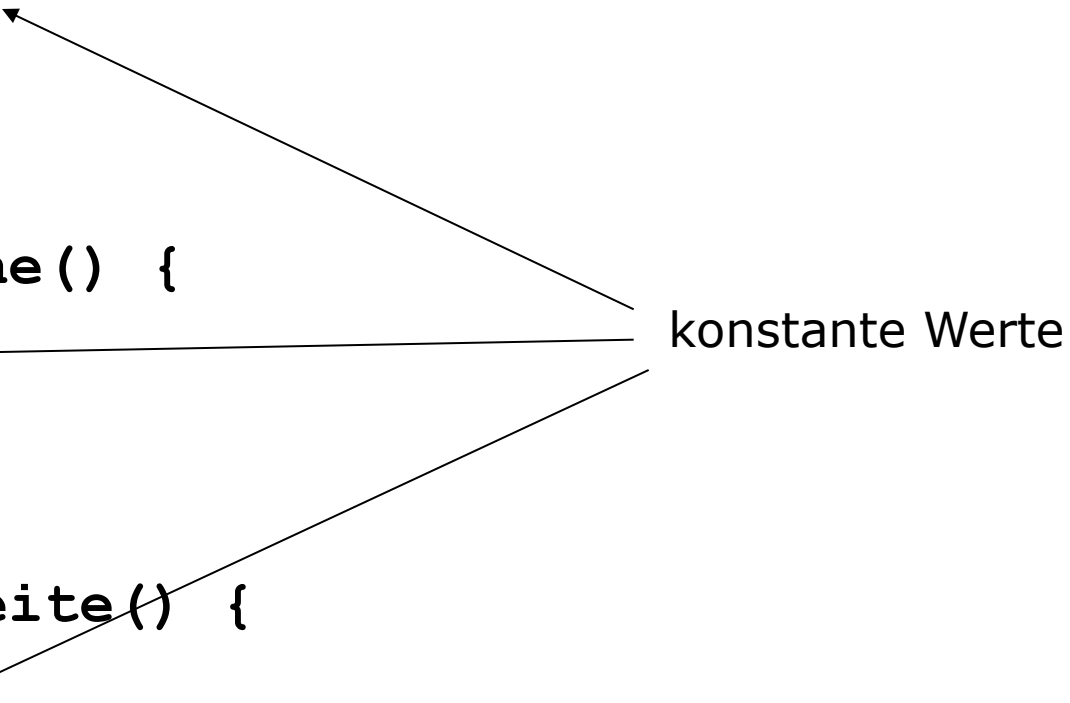
Zum Bereitstellen von Funktionalität muss das **Interface durch eine Klasse implementiert** werden.

Implementierungsrelation



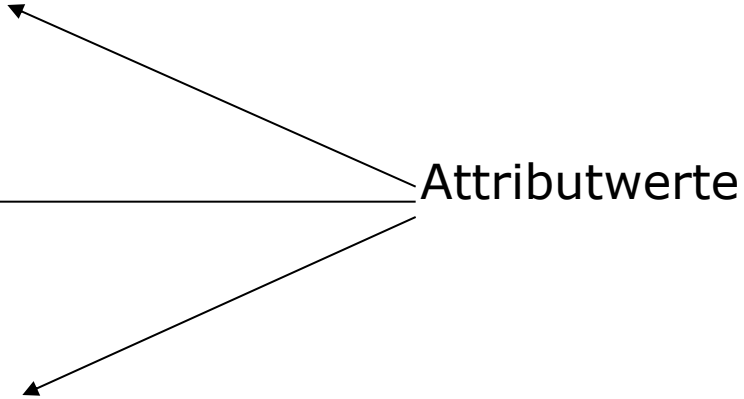
1. Implementierung von Größe

```
public class FußballPlatz implements Größe {
  public int länge() {
    return 105;
  }

  public int höhe() {
    return 0;
  }

  public int breite() {
    return 70;
  }
}
```



2. Implementierung von Größe

```
public class Schrank implements Größe {
  int länge;
  int höhe;
  int breite;

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



Teilweise Implementierung von `Größe`

```
abstract class Rechteck implements Größe {
  int länge;
//  int höhe;
  int breite;

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

Verwendung des Interface Größe

```
class InterfaceVerwendung {
  int grundflaeche(Größe g) {
    return g.länge() * g.breite();
  }
  public static void main(String[] args) {
    Schrank schrank = new Schrank();
    schrank.länge = 1;
    schrank.breite = 3;
    FussballPlatz platz = new FussballPlatz();
    InterfaceVerwendung iv = new InterfaceVerwendung();
//Nun werden sie ausgegeben:
    System.out.println("Schrank: " +
   iv.grundflaeche(schrank));
    System.out.println("Platz: " +
   iv.grundflaeche(platz));
  }
}
```

Implementierung von 2 Interfaces

```
public class Schrank2 implements Größe, Alter {
  int länge;
  int höhe;
  int breite;

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

neu hinzunehmen:

```
int bauJahr;

public int alter(int aktJahr) {
  return (aktJahr - bauJahr);
}
```

Vererbung von Interf. bei Klassenvererb.

```
public class SchrankNeu extends Schrank
                        implements Alter {

  int baujahr;

  public int alter(int aktJahr) {
    return (aktJahr - bauJahr);
  }
}
```

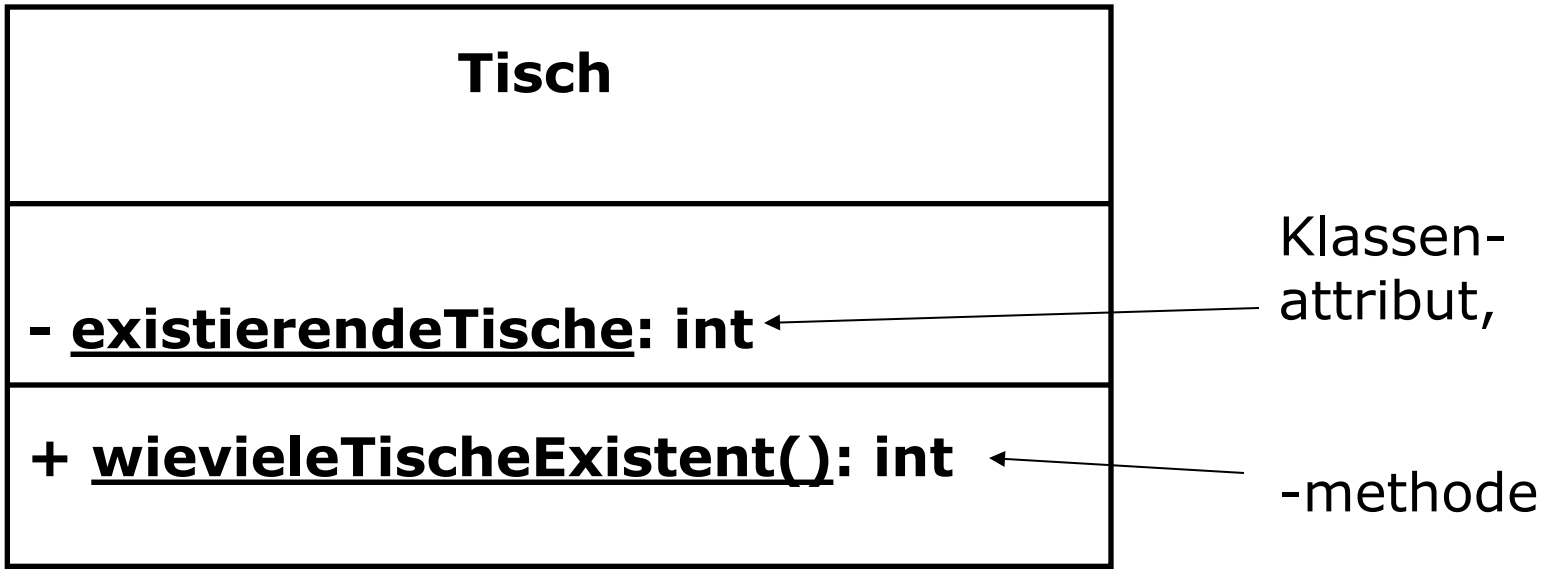
Vererben von Interfaces

```
interface EinDim {
  public int länge();
}

interface ZweiDim extends EinDim {
  public int breite();
}

interface DreiDim extends ZweiDim {
  public int höhe();
}
```

Implementierung von `DreiDim`

```
class Körper implements DreiDim {
  public int länge() {
    return 0;
  }

  public int breite() {
    return 0;
  }

  public int höhe() {
    return 0;
  }
```

Klassenattribute und -methoden (UML)



Oft auch **statisch** genannt

Klassenattribute und -methoden (Java)

```
public class Tisch {
  static private int existierendeTische = 0;
  public Tisch() {
    ++existierendeTische;
  }

  public void finalize() {
    --existierendeTische;
  }

  static public int wievieleTischeExistent() {
    return existierendeTische;
  }
}
```

enthält Anzahl der Objekte von Tisch

durch Konstruktor inkrementiert

durch Garbage Collector aufgerufen

gibt Klassenattribut aus

Dokumentationskommentar

Für Compiler gleich Blockkommentar
Findet nur bei javadoc Berücksichtigung

```
/**
* Socken sind spezielle Kleidungsstücke.
*/
public class Socke extends Kleidung {
}
```

wichtige Dokumentationskommentare

Kommentar	Beschreibung
@see Klassenname	Verweis auf eine andere Klasse
@see Methodenname	Verweis auf eine andere Methode
@version Versionstext	Version
@author Autor	Autor
@return Rückgabetext	Rückgabewert
@param Parametername oder Parametertext	Beschreibung der Parameter
@exception Exception-Klassenname oder Exceptiontext	Ausnahmen, die ausgelöst werden können
@throws Exception-Klassenname oder Exceptiontext	Synonym zu oben
{ **@link** Verweis }	Eingebauter Verweis