Tafelübung 07
Algorithmen und Datenstrukturen

Lehrstuhl für Informatik 2 (Programmiersysteme)

Friedrich-Alexander-Universität Erlangen-Nürnberg

Wintersemester 2016/2017

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Übersicht

Vererbung

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Vererbung: Grundlagen (I)

- Vererbung $\leadsto$ Relation zwischen zwei Klassen
  - Unterklasse erbt von einer Oberklasse
  - wird manchmal auch als „ist-ein-Beziehung“ bezeichnet
- eine Unterklasse...
  - übernimmt alle Methoden und Attribute ihrer Oberklasse
    - kann aber auf *private* Methoden und Attribute nicht zugreifen
  - kann eigene Methoden und Attribute definieren
    - kann damit auch Methoden ihrer Oberklasse überschreiben (s.u.)
- ein Objekt der Unterklasse ist auch ein Objekt der Oberklasse
  - an allen Stellen, an denen ein Objekt der Oberklasse erwartet wird, kann auch ein Objekt der Unterklasse verwendet werden (s.u.)

Vererbung: Grundlagen (II)

- damit möglich: Generalisierung bzw. Spezialisierung
  - Unterklasse ist Spezialisierung der Oberklasse
  - Oberklasse ist Generalisierung der Unterklasse(n)

Beispiel

- jede Eiche ist ein Baum
- aber nicht jeder Baum ist eine Eiche (denn es gibt noch Buchen, Birken, ...)

$\rightsquigarrow$ `Eiche` kann man als Unterklasse von `Baum` implementieren

- jeder Baum ist eine Pflanze
- aber nicht jede Planze ist ein Baum (denn es gibt noch Blumen, Moose, ...)

$\rightsquigarrow$ `Baum` kann man als Unterklasse von `Pflanze` implementieren

Vererbung in Java

- in Java:
  - Vererbung mittels `extends` bei der Klassen-Definition
  - jede Klasse erbt von genau einer anderen Klasse
    - falls nicht explizit angegeben, ist die Oberklasse `Object`
    - *alle* Klassen in Java erben direkt oder indirekt von `Object`

$\rightsquigarrow$ Einvachvererbung führt zu einer sog. Mono-Hierarchie

  - Zugriff auf Methoden oder Attribute der Oberklasse: `super`
    - notwendig, falls Unterklasse Teile der Oberklasse verdeckt
  - falls die Oberklasse keinen Standard-Konstruktor hat:
    - expliziter Aufruf eines Oberklasse-Konstruktors in *jedem* Konstruktor der Unterklasse notwendig

Vererbung: Beispiel

Beispiel

```
class Form {
    protected int x;
    protected int y;
    public Form(int x, int y) {
        this.x = x; this.y = y;
    }
}

class Rechteck extends Form { // jedes Rechteck ist auch eine Form
    protected int breite;
    protected int hoehe;
    public Rechteck(int x, int y, int breite, int hoehe) {
        super(x, y); // Aufruf des Konstruktors der Oberklasse
        this.breite = breite; this.hoehe = hoehe;
    }
}

class Kreis extends Form { // jeder Kreis ist auch eine Form
    protected int radius;
    public Kreis(int x, int y, int radius) {
        super(x, y); // Aufruf des Konstruktors der Oberklasse
        this.radius = radius;
    }
}
```

Hinweise zum Beispiel

Beispiel

```
class Form { /* ... */ }
class Rechteck extends Form { /* ... */ }
class Kreis extends Form { /* ... */ }
```

- `Rechteck` und `Kreis` sind Spezialisierungen von `Form`
- `Form` ist Generalisierung von `Rechteck` und `Kreis`
- Oberklasse `Form` hat keinen Standard-Konstruktor

$\rightsquigarrow$ expliziter Konstruktor-Aufruf in den Konstruktoren der Unterklassen

- Attribute `x` und `y` in `Form` haben Sichtbarkeit `protected`

$\rightsquigarrow$ Zugriff aus Unterklassen möglich

Abstrakte Klassen

- abstrakte Klasse:
  - kann nicht instanziiert werden
    - es können also keine Objekte direkt von dieser Klasse erzeugt werden,
    - sondern nur von (nicht-abstrakten) Unterklassen dieser Klasse
  - kann als statischer Typ einer Referenz-Variable verwendet werden
    - später mehr... ☺
- in Java: Deklaration einer abstrakten Klasse mittels `abstract`

Abstrakte Klassen: Beispiel

Beispiel (I)

```
abstract class Form { /* ... */ }
class Rechteck extends Form { /* ... */ }
class Kreis extends Form { /* ... */ }
```

- eine „Form“ alleine zu erzeugen, ist nicht sinnvoll

$\rightsquigarrow$ Klasse `Form` abstrakt definieren

- `Rechteck` und `Kreis` sind selber nicht abstrakt
  - können wie bekannt instanziiert werden

Beispiel (II)

Einen Baum, der weder Eiche, Buche, Birke, noch sonst irgendwas ist, gibt es nicht.

$\rightsquigarrow$ `Baum` würde man als abstrakte Klasse definieren

$\rightsquigarrow$ gleiches gilt für `Pflanze`

Abstrakte Methoden

- abstrakte Klassen können abstrakte Methoden deklarieren
  - legen lediglich die Signatur fest, nicht die Implementierung

$\rightsquigarrow$ „Methoden ohne Rumpf“

- alle Unterklassen der abstrakten Oberklasse...
  - ... müssen diese abstrakten Methoden implementieren, *oder*
  - ... sind selbst abstrakt

Sinn!?

Mit abstrakten Methoden lässt sich festlegen, dass alle Objekte (von Unterklassen) der abstrakten Klasse eine bestimmte Methode haben müssen, auch wenn die konkrete Implementierung in der abstrakten Klasse auf Grund der Generalisierung nicht möglich ist.

Abstrakte Methoden: Beispiel (I)

- *jede* Form soll eine Fläche haben
  - aber: Berechnung der Fläche ist aber abhängig von der *konkreten* Form
    - Fläche eines Rechtecks berechnet sich anders als Fläche eines Kreises

$\rightsquigarrow$ nur die Unterklassen „wissen“, wie Berechnung zu implementieren ist

$\rightsquigarrow$ in der Klasse `Form` eine abstrakte Methode `flaeche()` deklarieren

- ... und in den Unterklassen jeweils passend implementieren

Abstrakte Methoden: Beispiel (II)

Beispiel

```
abstract class Form {
    // ...
    public abstract int flaeche();
}

class Rechteck extends Form {
    // ...
    public int flaeche() {
        return this.breite * this.hoehe;
    }
}

class Kreis extends Form {
    // ...
    public int flaeche() {
        return (int)(this.radius * this.radius * 3.14159);
    }
}
```

Interfaces

- Interface:
  - „Sammlung“ von nicht-implementierten Methoden

  $\rightsquigarrow$ ein Interface deklariert ausschließlich abstrakte Methoden
  - Ähnlichkeit zu Abstrakten Klassen: ein Interface...
    - ... kann nicht instanziiert werden
    - ... kann als statischer Typ verwendet werden

$\rightsquigarrow$ Interface $\approx$ Abstrakte Klasse mit ausschließlich abstrakten Methoden

- eine Klasse kann beliebig viele Interfaces implementieren
  - und muss damit *alle* geforderten Methoden implementieren

- in Java:
  - Definition eines Interface mittels `interface` (statt `class`)
  - Implementieren eines Interface durch eine Klasse mittels `implements`

Interfaces: Sinn!?

Der Sinn von Interfaces

- Definition von Schnittstellen, die i.A. *orthogonal* zu der Vererbungs-Hierarchie sind
  - Schnittstelle soll von mehreren Klassen implementiert werden, die sich nicht (sinnvoll) gegenseitig in eine Generalisierungs-Beziehung setzen lassen
- Interface kann man als statischen Typ einer Referenz-Variable verwenden (s.u.)

$\rightsquigarrow$ man kann auf die im Interface definierten Methoden zugreifen

$\rightsquigarrow$ welche Klasse sich tatsächlich dahinter befindet, ist irrelevant

Interfaces: Beispiel 1

Beispiel 1

```
interface KannGeraeuschMachen {
    String macheGeraeusch();
}

class Hund implements KannGeraeuschMachen {
    // ...
    public String macheGeraeusch() { return "Wau!"; }
}

class Auto implements KannGeraeuschMachen {
    // ...
    public String macheGeraeusch() { return "Brumm!"; }
}
```

Hinweise zu dem Beispiel

Hund und `Auto` lassen sich eher schwer in eine explizite Vererbungs-Beziehung setzen (je nach Anwendung vermutlich keine sinnvolle gemeinsame Oberklasse), sollen aber beide die Methode `macheGeraeusch()` zur Verfügung stellen und überall dort verwendet werden können, wo lediglich diese Methode benötigt wird.

Interfaces: Beispiel 2

Beispiel 2

```
interface Sammlerstueck {
    int wert();
}

interface Fortbewegungsmittel {
    int maximalGeschwindigkeit();
}

class Oldtimer implements Sammlerstueck, Fortbewegungsmittel  {
    // ...
    public int wert() { /* ... */ }
    public int maximalGeschwindigkeit() { /* ... */ }
}
```

Hinweise zu dem Beispiel

Ein `Oldtimer` ist sowohl ein `Sammlerstueck`, als auch ein `Fortbewegungsmittel`, und soll an allen Stellen verwendet werden können, an denen ein `Sammlerstueck` oder ein `Fortbewegungsmittel` verlangt ist.

Polymorphie

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Polymorphie: Einführung

- Polymorphie: aus dem Griechischen für „*Vielgestaltigkeit*“
  - ein und dieselbe Sache kann verschiedene Gestalten haben
  - hier: derselbe Name kann verschiedene Dinge meinen
    - konkrete Bedeutung ist abhängig vom jeweiligen Kontext
- polymorphe Variablen:
  - dynamischer und statischer Typ können sich unterscheiden
- polymorphe Methoden:
  - Überschreiben einer Methode in einer Unterklasse
  - Überladen einer Methode

Haben wir alle schon benutzt...

Beispiel

```
int foo = 13 + 3;
String bar = "Hallo, " + "Welt";
```

Polymorphie im Beispiel

Der +-Operator ist polymorph, denn in der ersten Zeile meint er die Integer-Addition, in der zweiten Zeile hingegen die String-Konkatenation.

Statischer Typ $\leftrightarrow$ Dynamischer Typ

- eine Referenz-Variable hat immer zwei Typen:
  - statischer Typ:
    - Typ, der bei der Deklaration der Variable angegeben wird
    - zur Übersetzungszeit bekannt
    - ändert sich nicht
  - dynamischer Typ:
    - tatsächlicher Typ der Instanz, auf welche die Referenz verweist
    - im Allgemeinen erst zur Laufzeit des Programms bekannt
    - kann sich ändern
- „Kandidaten“ für den dynamischen Typ:
  - falls statischer Typ eine Klasse: statischer Typ oder Unterklasse
  - falls statischer Typ ein Interface: implementierende Klasse

Beispiel

Beispiel

```
interface Sammlerstueck { /* ... */ }
class Oldtimer implements Sammlerstueck { /* ... */ }
class Briefmarke implements Sammlerstueck { /* ... */ }

//====================================

Sammlerstueck objekt = null;
    // stat. Typ: Sammlerstueck, dyn. Typ: keiner

objekt = new Oldtimer();
    // stat. Typ: Sammlerstueck, dyn. Typ: Oldtimer

objekt = new Briefmarke();
    // stat. Typ: Sammlerstueck, dyn. Typ: Briefmarke
```

Zugriff auf Methoden und Attribute

Achtung

Es können nur auf die Methoden und Attribute des statischen Typs zugegriffen werden, da der dynamische Typ erst zur Laufzeit feststeht, der Compiler jedoch zur Übersetzungszeit garantieren muss, dass die entsprechende Methode bzw. das entsprechende Attribut tatsächlich existiert.

Da der dynamische Typ immer „mindestens“ der statische Typ ist, ist der Zugriff auf Methoden und Attribute des statischen Typs kein Problem, denn diese Methoden bzw. Attribute existieren garantiert auch im dynamischen Typ.

Beispiel

Beispiel

```
abstract class Tier {
    // ...
    public abstract void macheLaut();
}
class Hund extends Tier {
    // ...
    public void fresseHundefutter() { /* ... */ }
}
class Katze extends Tier { /* ... */ }


//===================================


Tier tier; // statischer Typ: Tier

tier = new Katze();
tier.macheLaut(); // in Ordnung, macheLaut() ist Methode des statischen Typs

tier = new Hund();
tier.macheLaut(); // in Ordnung, macheLaut() ist Methode des statischen Typs
tier.fresseHundefutter(); // Compiler-Fehler
                // fresseHundefutter() ist keine Methode des statischen Typs
```

Überladen von Methoden

- Überladen (*Overloading*):
  - mehrere Methoden mit demselben Namen ($\rightsquigarrow$ Polymorphie)...
  - ...aber unterschiedlicher Signatur
    - Methoden müssen sich in Anzahl/Typen der Parameter unterscheiden
    - unterschiedliche Parameternamen und/oder unterschiedlicher Rückgabetyp alleine reicht nicht
  - innerhalb derselben Klasse oder in einer Unterklasse

Beispiel

```
public static int mul(int a, int b) { return a * b; }
public static float mul(float a, float b) { return a * b; }
```

Überschreiben von Methoden

- Überschreiben (*Overriding*):
  - nur in erbenden Klassen möglich
  - Unterklasse re-definiert eine Methode der Oberklasse
    - gleicher Methoden-Name ($\rightsquigarrow$ Polymorphie)
    - gleiche Signatur

Welche Methode wird aufgerufen?

- Klassen-Methode: Methode des statischen Typs
  - sog. statische Bindung, Entscheidung zur Übersetzungszeit
- Instanz-Methode: Methode des dynamischen Typs
  - sog. dynamische Bindung, Entscheidung zur Laufzeit

Überschreiben von Methoden: Beispiel

```
class Oberklasse {
    public static void klassenMethode() {
        System.out.println("Klassen-Methode der Oberklasse");
    }
    public void instanzMethode() {
        System.out.println("Instanz-Methode der Oberklasse");
    }
}

class Unterklasse extends Oberklasse {
    public static void klassenMethode() {
        System.out.println("Klassen-Methode der Unterklasse");
    }
    public void instanzMethode() {
        System.out.println("Instanz-Methode der Unterklasse");
    }
}

//================================================

Oberklasse objekt;

objekt = new Oberklasse();
objekt.klassenMethode(); // -> Klassen-Methode der Oberklasse
objekt.instanzMethode(); // -> Instanz-Methode der Oberklasse

objekt = new Unterklasse();
objekt.klassenMethode(); // -> Klassen-Methode der Oberklasse
objekt.instanzMethode(); // -> Instanz-Methode der Unterklasse
```

UML-Klassendiagramme

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Klassendiagramme: Motivation

- bei großen Programmen:
  - viele Klassen
  - viele Beziehungen zwischen Klassen

$\rightsquigarrow$ wird irgendwann unübersichtlich...

$\rightsquigarrow$ Klassendiagramme als Werkzeug der Programmmodellierung
  - zur Dokumentation
  - zur Kommunikation (auch mit Programmier-Laien)

- es gibt Werkzeuge, die...
  - aus Code Klassendiagramme erzeugen
  - aus Klassendiagrammen Code erzeugen

Klassenkarten

- jede Klasse wird durch eine Klassenkarte dargestellt
- diese besteht aus drei „Bereichen“
  - Klassenname (immer vorhanden)
  - Attribute der Klasse (kann weggelassen werden)
  - Methoden der Klasse (kann weggelassen werden)
- zwischen Klassenkarten werden Beziehungen eingezeichnet
  - Vererbungs-Beziehungen
  - Referenzierungs-Beziehungen
  - . . .

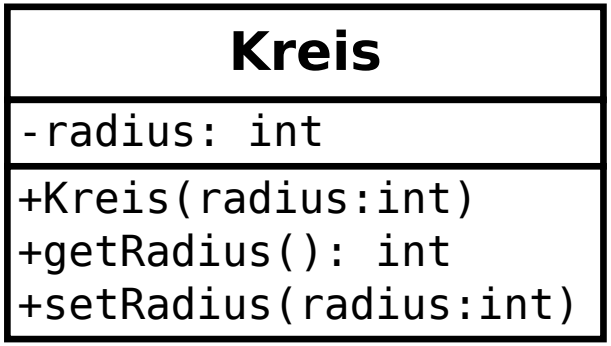
Klassenkarten: Beispiel

Beispiel

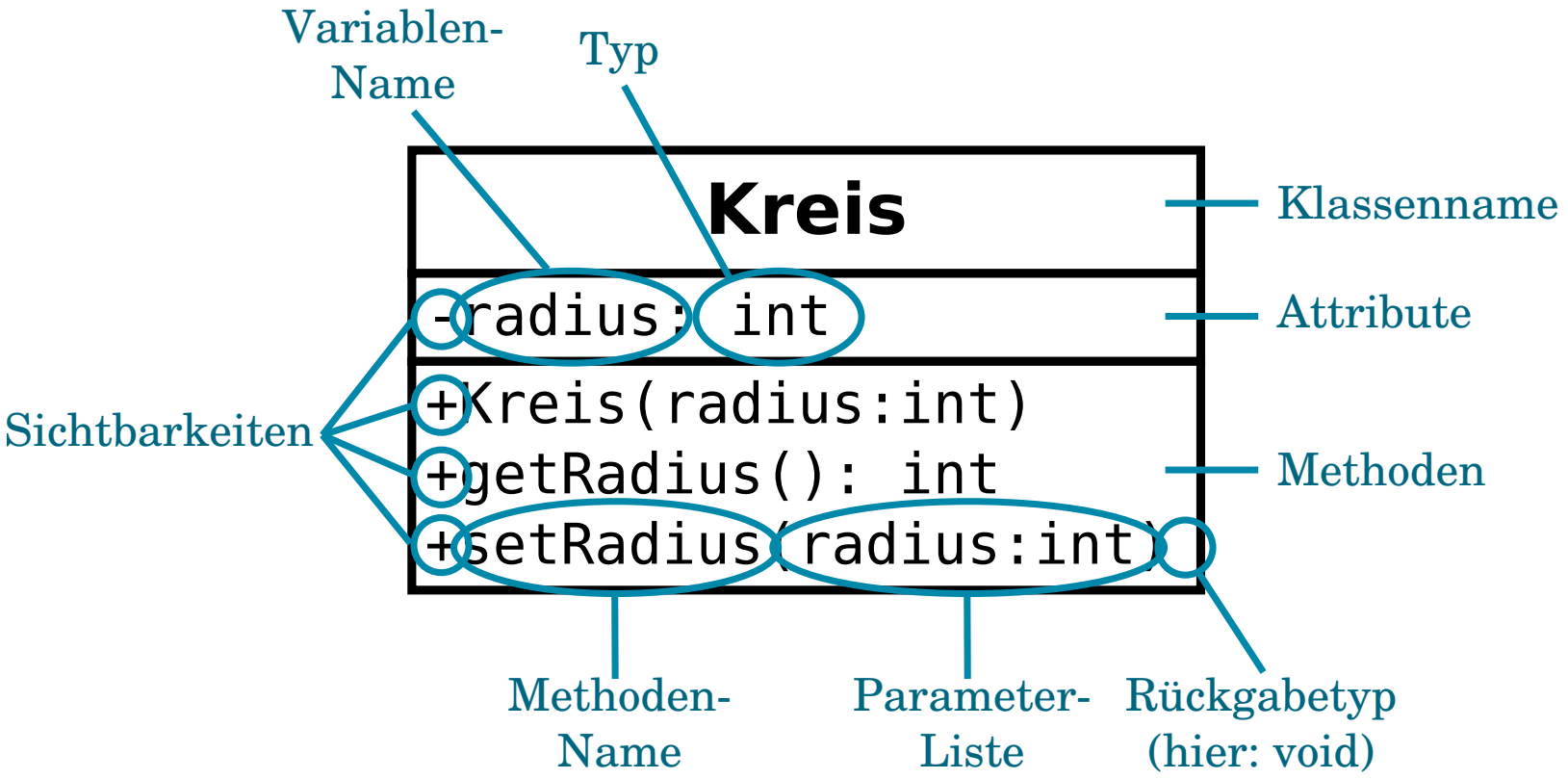
```
class Kreis {
   private int radius;

   public Kreis(int radius) { this.radius = radius; }

   public int getRadius() { return this.radius; }
   public void setRadius(int radius) { this.radius = radius; }
}
```



Im Detail


Variablen-
Name
Typ
Kreis
Klassenname
-radius: int
Attribute
+Kreis(radius:int)
+getRadius(): int
Methoden
+setRadius(radius:int)
Sichtbarkeiten
Methoden-
Name
Parameter-
Liste
Rückgabetyp
(hier: void)

Sichtbarkeiten

- vor Attributen/Methoden wird die Sichtbarkeit angegeben:
  - `+` `public`
  - `#` `protected`
  - `–` `private`
  - `~` paket-sichttbar

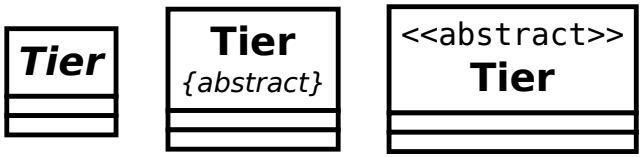
  ohne noch nicht festgelegt oder paket-sichttbar

Sichtbarkeiten in Java

Modifikator	Eigenschaft ist sichtbar aus/von			
	eigener Klasse	selben Paket	Unterklasse	überall
`public`	ja	ja	ja	ja
`protected`	ja	ja	ja	nein
ohne/paketsichtbar	ja	ja	nein	nein
`private`	ja	nein	nein	nein

Abstrakte Klassen und Methoden

- drei Möglichkeiten für die Darstellung abstrakter Klassen
  1. Klassennamen *kursiv* schreiben
  2. unter dem Klassennamen: {abstract} (Kommentar)
  3. über dem Klassennamen: «abstract» (Stereotyp)



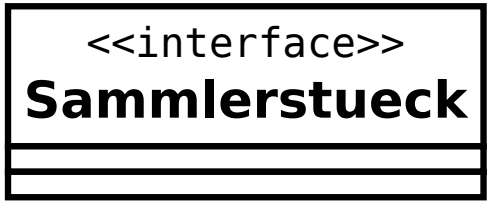
- zwei Möglichkeiten für die Darstellung abstrakter Methoden
  1. Methoden-Namen *kursiv* schreiben
  2. vor dem Methoden-Namen: «abstract»

Hinweis

In Hausaufgaben und der Klausur bitte jeweils die erste Variante vermeiden. Das macht das Korrigieren deutlich leichter und Missverständnisse unwahrscheinlicher... ☺

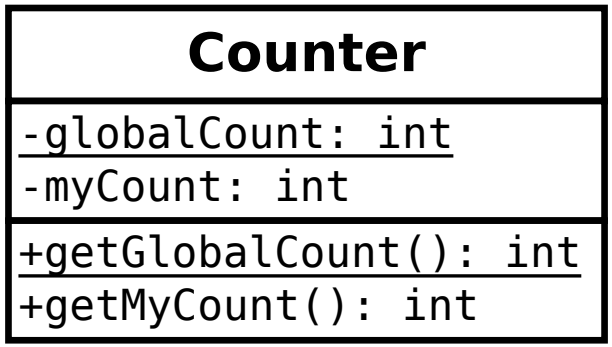
Interfaces

- Kennzeichung von Interfaces:
  - Stereotyp «interface»



Statische Methoden und Attribute

- Kennzeichnung statischer Methoden und Attribute:
  - Unterstreichen der Bezeichner



Tipp

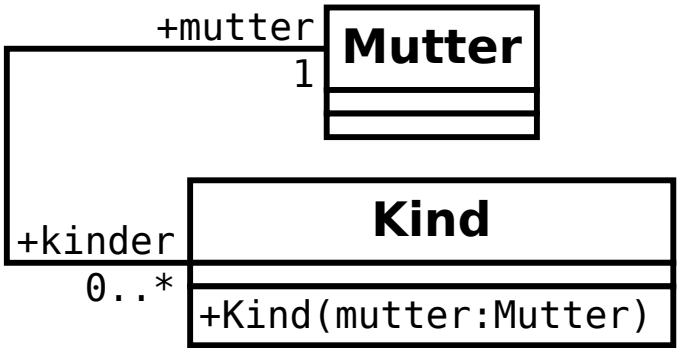
Schaut genau hin, ob es sich um eine Klassen- oder Instanzvariable handelt. Manchmal „versteckt“ sich die Unterstreichung nämlich nur knapp oberhalb des Trennstriches zwischen Attributen und Methoden.

Allgemeine Beziehungen

Beispiel

```
class Mutter {
    public Kind[] kinder;
}

class Kind {
    public Mutter mutter;
    public Kind(Mutter mutter) {
        this.mutter = mutter;
    }
}
```



- eine `Mutter` hat beliebig viele `Kinder`
- ein `Kind` hat genau eine `Mutter`
- `kinder` und `mutter` sind die sog. Rollennamen (entsprechen Attributnamen)
- die Beziehung kann einen sog. Assoziationsnamen haben

Multiplizitäten

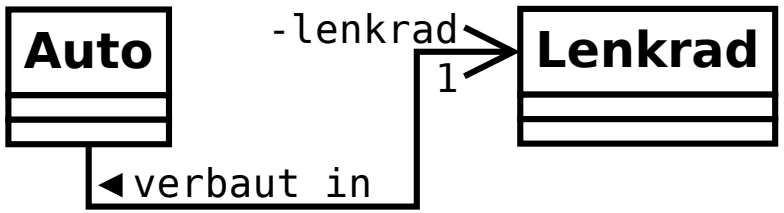
- die Multiplizität gibt an, wie viele Objekte einer Klasse...
  - ... mindestens und wie viele
  - ... maximal an der Assoziation teilnehmen
- gängige Multiplizitäten:
  - `0..1` (maximal ein Objekt)
  - 1 (genau ein Objekt)
  - `0..*` oder $*$ (beliebig viele Objekte)
  - `1..*` (beliebig viele Objekte, aber mindestens eines)
- Umsetzung in Java:
  - Multiplizitäten mit $*$: im einfachsten Fall mit Hilfe eines Arrays
  - andere: einfache Referenzen

Direkte Beziehungen

Beispiel

```
class Lenkrad { }

class Auto {
  private Lenkrad lenkrad =
      new Lenkrad();
}
```



- ein `Auto` hat genau ein `Lenkrad`
- Pfeil deutet an, dass nur `Auto` Referenz auf `Lenkrad` hat (nicht umgekehrt)
- *„verbaut in“* ist der Assoziationsname
  - hier nur im UML-Diagramm (manchmal auch als Kommentar im Code)

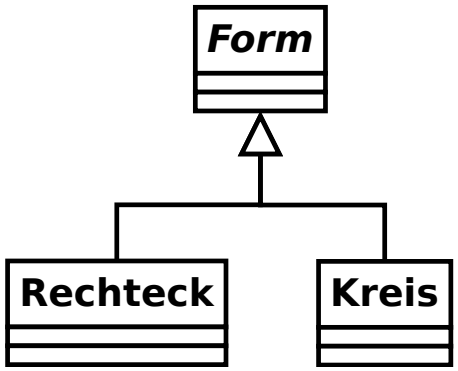
Vererbungs-Beziehung

Beispiel

```
abstract class Form { }

class Rechteck extends Form { }

class Kreis extends Form { }
```



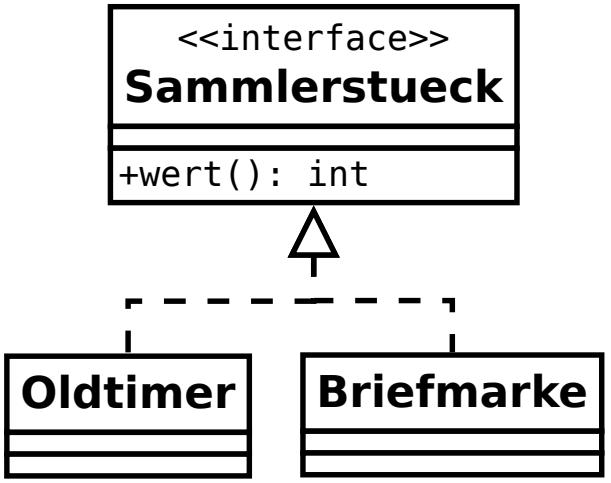
Implementierungs-Beziehung

Beispiel

```
interface Sammlerstueck {
    int wert();
}

class Oldtimer
   implements Sammlerstueck {
    // ...
}

class Briefmarke
   implements Sammlerstueck {
    // ...
}
```



Hinweise zu den Aufgaben

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

StarTrekLift

Hinweis

In dieser Aufgabe soll zu einem UML-Klassendiagramm passender, gültiger Java-Code abgegeben werden. Die Klassen sollen allesamt in einer einzigen Datei `StarTrekLift.java` abgegeben werden.
Deshalb: Nur die Klasse `StarTrekLift` darf public sein, sonst ist die Java-Datei nicht übersetzbar. Alle anderen Klassen sollen paketsichtbar sein.

StarTrekLift

Wichtig!

In dieser Aufgabe dürfen keine Klassen verschachtelt werden! (Sonst $\rightsquigarrow$ 0 Punkte!)

So **nicht!**

```
public class Foo {
    class Bar {}  // Bar und Bazz stehen innerhalb der Klasse Foo
    class Bazz {} // genau das soll *nicht* gemacht werden
}
```

Sondern so:

```
public class Foo {}
class Bar{}  // Bar und Bazz stehen neben der Klasse Foo
class Bazz{} // so soll es auch in der Hausaufgabe gemacht werden
      // (mit den richtigen Klassennamen, nicht Foo, Bar und Bazz)
```

Fragen? Fragen!

(hilft auch den anderen)

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT