6.5. Einführung in UML

- UML = Unified Modelling Language, Entwicklung seit 1994
- standardisierte (graphische) Sprache zur objektorientierten Modellierung von (Software-)Systemen
- UML definiert eine Vielzahl von Diagrammtypen: unterschiedliche Sichtweisen des modellierten Systems
  - statische vs. dynamische Aspekte
  - unterschiedlicher Abstraktionsgrad
- UML unterstützt sowohl Objekt-Orientierte Analyse (OOA) als auch Objekt-Orientiertes Design (OOD)

6.5. Einführung in UML: Klassen, Wiederholung aus Java:

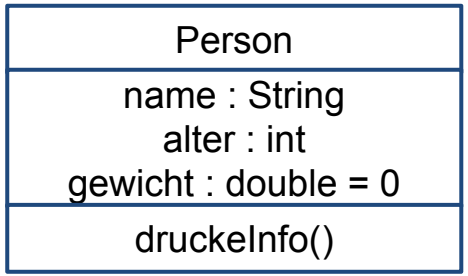
- Eine Klasse definiert für eine Kollektion gleichartiger Objekte
  - deren Struktur, d.h. die **Attribute** (nicht die Werte)
  - das Verhalten, d.h. die **Operationen**
  - die möglichen Beziehungen (**Assoziationen**) zu anderen Objekten, einschließlich der Generalisierungs-Beziehung
- Eine Klasse besitzt einen Mechanismus, um neue Objekte zu erzeugen

  Die Klasse ist der ”Bauplan“ für diese Objekte

6.5. Einführung in UML: Klassen

- Anforderungserhebung, Analyse
  Entwickler: „Was ist euch wichtig?“
  Anwender: „Der Kunde.“
  Entwickler: „Was ist denn ein Kunde, welche Merkmale sind für euch relevant?“
  Anwender: „Der Kunde hat einen Namen, eine Anschrift und eine Bonität, die wir überprüfen.“

- Design (Klasse in UML):



Implementierung (Klasse in Java):

```java
class Person {
  String name;
  int alter;
  double gewicht = 0;
  public void druckeInfo() {/*...*/}
}
```

6.5. Einführung in UML: Objekte

- Allgemeines Schema:

<Objektname>:<Klassenname>
<Attributname 1> = <Wert 1> … <Attributname n> = <Wert n>

- z.B.:

myVideo : Video
author = "R.A." length = 3.32 ytlink = "dQw4w9WgXcQ"

- Objekt− und Attributnamen klein geschrieben, Klassennamen groß geschrieben
- Die Operationen werden hier nicht angegeben, da sie für alle Objekte einer Klasse identisch sind

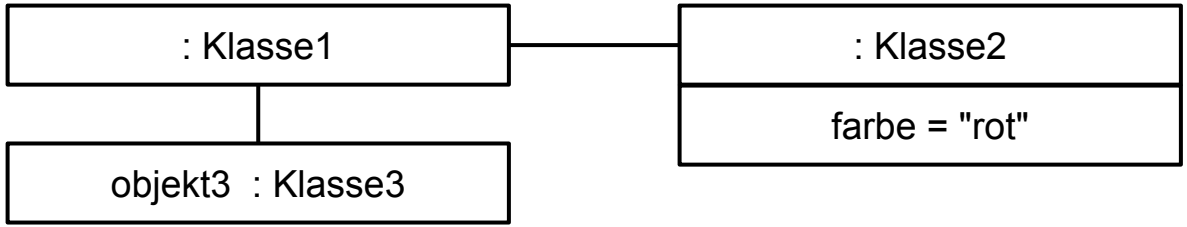
```
class Video {
  String author;
  double length = 0.0;
  String ytlink;
  /*...*/
}
Video rr;
rr = new Video("R.A.", 3.32, "dQw4w9WgXcQ");
```

6.5. Einführung in UML: Objekte

- Darstellungsvarianten:

  - Objekt ohne Klasse: Klasse geht aus Zusammenhang hervor
  - Anonymes Objekt

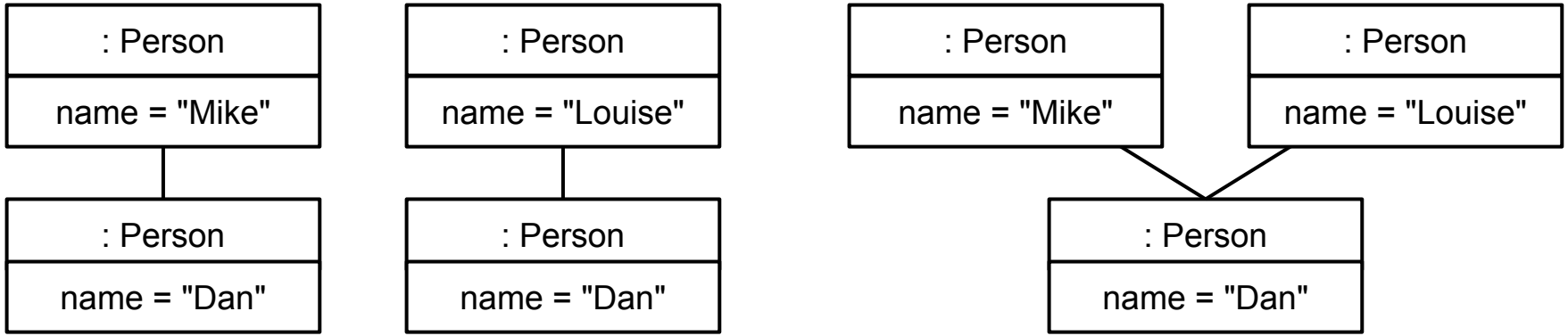
- Objektdiagramm mit Objektbeziehungen:
  Beziehungen werden durch Verbindungslinien zwischen Objekten dargestellt



6.5. Einführung in UML: Objekte

Identität und Gleichheit von Objekten

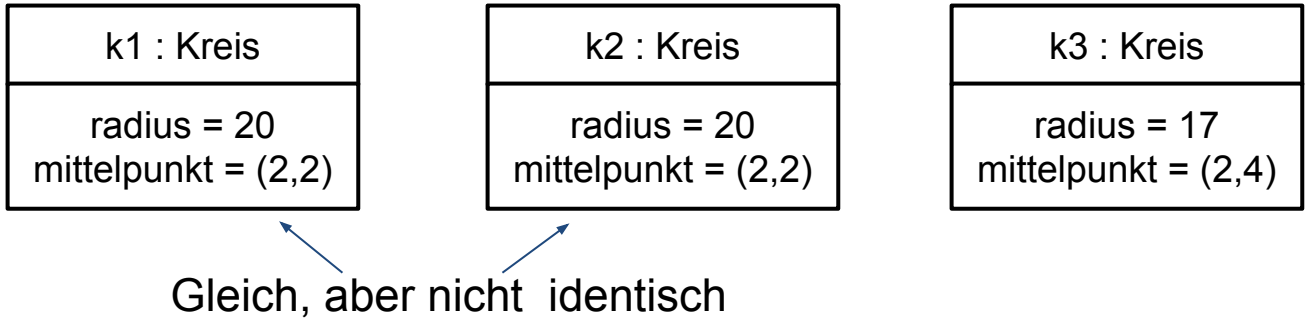
- Ein Objekt besitzt einen Zustand, ein wohldefiniertes Verhalten und eine Identität, die es von allen anderen Objekten unterscheidet
- Zwei Objekte sind gleich, wenn sie dieselben Attributwerte besitzen:



6.5. Einführung in UML: Objekte

Objektidentität

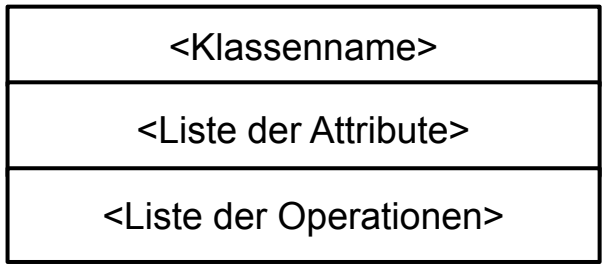
- Jedes Objekt ist per Definition, unabhängig von seinen konkreten Attributwerten, von allen anderen Objekten eindeutig zu unterscheiden.



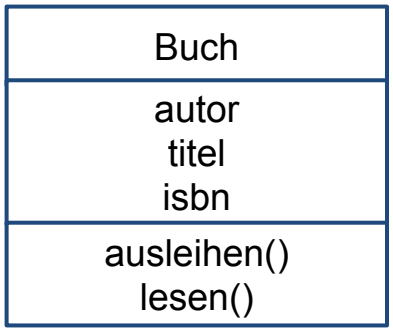
- Zur Laufzeit wahren Speicheradressen die Identität eines Objektes
- In Objektdatenbankmanagementsystem (ODBMS) werden oft künstlich erzeugte Identitätsnummern (sog. OID‘s) verwendet

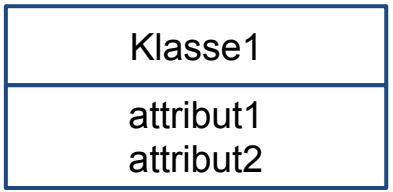
6.5. Einführung in UML: Klassen

- Allgemeines Schema:

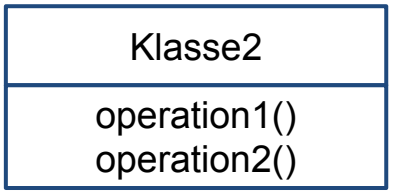


- z.B.:





Klasse ohne Operationen

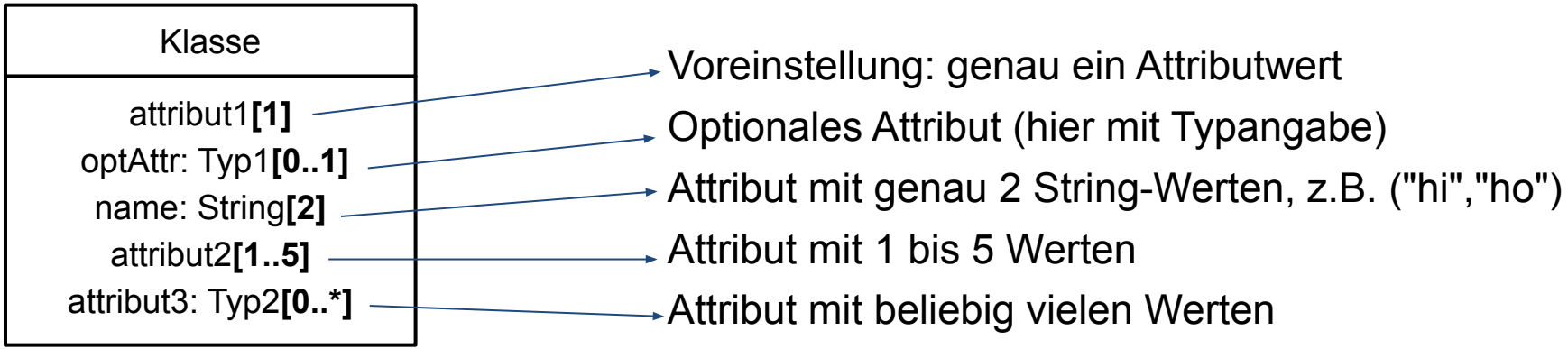


Klasse ohne Attribute

6.5. Einführung in UML: Attribute

Multiplizitäten

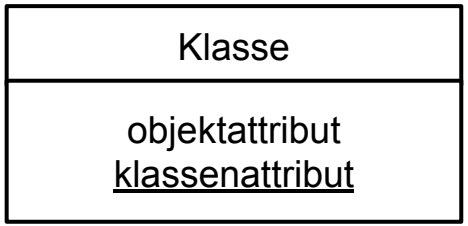
- Attribute können mit einer Multiplizität versehen werden
- Die Multiplizität gibt an, aus wie vielen Werten das Attribut bestehen kann:
  [Untergrenze .. Obergrenze (oder * für beliebig viele)]
- z.B.:



6.5. Einführung in UML: Attribute

Klassenattribute

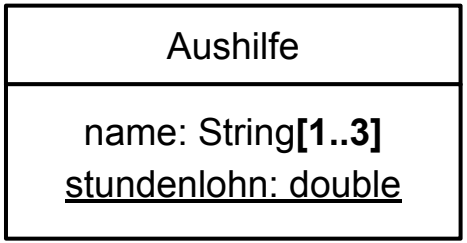
- Klassenattribute sind Attribute, für die nur ein einziger Attributwert für alle Objekte der Klasse existiert:
  - sie werden daher der Klasse zugeordnet, nicht den Objekten
  - sie existieren auch, wenn es (noch) kein Objekt der Klasse gibt
  - sie stellen oft auch Eigenschaften der Klasse selbst dar
- Klassenattribute werden durch Unterstreichen gekennzeichnet:



6.5. Einführung in UML: Attribute

In Java:

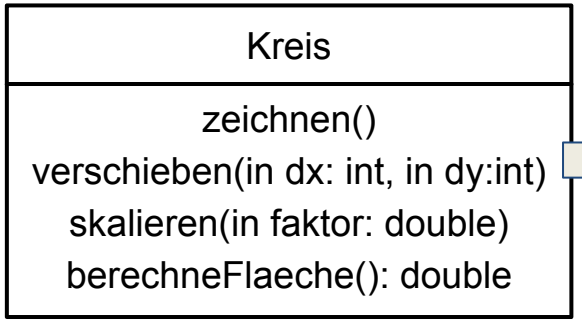
- Attribute mit Multiplizität ungleich 1 werden in Java durch Arrays (Felder) dargestellt
- in Java erfolgt bei der Deklaration eines Feldes keine Angabe, wieviel Elemente es enthalten kann
- Klassenattribute werden durch das vorgestellte Schlüsselwort static gekennzeichnet



```
class Aushilfe {
  String[] name;
  static double stundenlohn;
}
```

6.5. Einführung in UML: Operationen

- einfache Operationen und Parameter:




```
class Kreis {
     void zeichnen() {
          // Code der Operation 'zeichnen'
     }
     void verschieben(int dx, int dy) {
          // Code der Operation 'verschieben'
     }
     void skalieren(double faktor) {
          // Code der Operation 'skalieren'
     }
     double berechneFlaeche() {
          // Code der Operation 'berechneFlaeche'
     }
}
```

- In UML ist die Angabe einer Parameterliste optional
- Wenn eine Liste angegeben wird, muss sie mindestens die Namen der Parameter enthalten

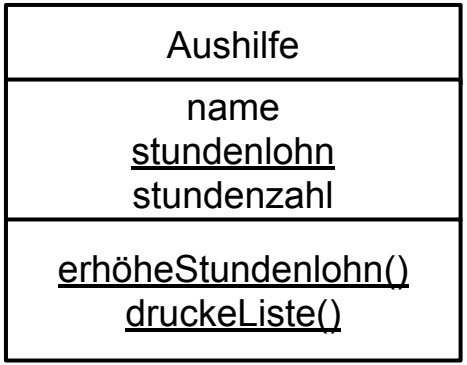
6.5. Einführung in UML: Operationen

- Operation: ausführbare Tätigkeit, die von einem Objekt über eine Botschaft angefordert werden kann
  - alle Objekte einer Klasse haben dieselben Operationen
  - Operationen können direkt auf die Attributwerte eines jeden Objekts der Klasse zugreifen
  - Synonyme: Services, Methoden, Funktionen, Prozeduren
- Drei Arten von Operationen:
  - (Objekt-)Operationen: werden immer auf ein einzelnes (bereits existierendes) Objekt angewandt
  - Konstruktoroperationen: erzeugen ein neues Objekt und initialisieren seine Attribute
  - Klassenoperationen

6.5. Einführung in UML: Operationen

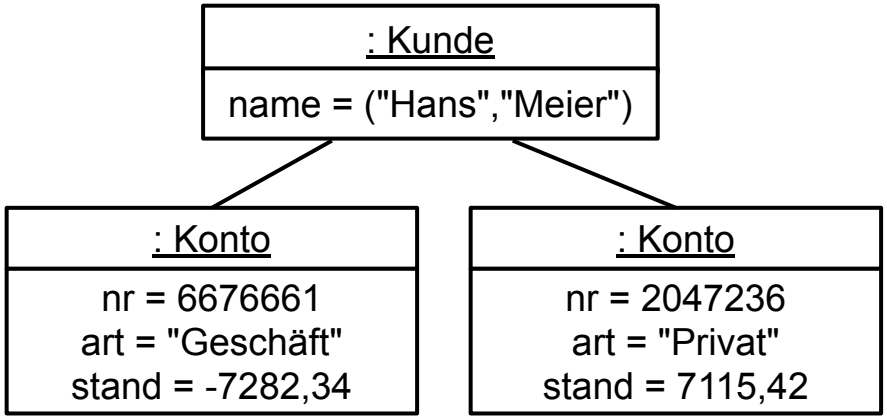
Klassenoperationen

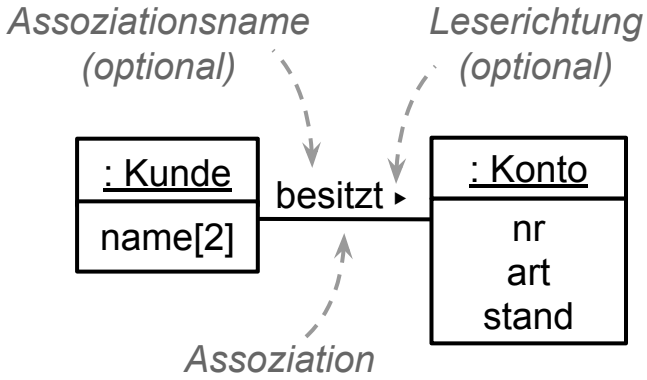
- Klassenoperationen sind der Klasse zugeordnet und werden nicht auf ein einzelnes Objekt angewendet
- Sie werden durch Unterstreichen kenntlich gemacht
- In der OOA werden Klassenoperationen in zwei Fällen benutzt:
  - Manipulation von Klassenattributen ohne Beteiligung eines Objekts, z.B. erhöheStundenlohn()
  - Operation bezieht sich auf alle oder mehrere Objekte der Klasse
    - nutzt Objektverwaltung aus
    - z.B. druckeListe():



6.5. Einführung in UML: Assoziationen

- Zwischen Objekten können Objektbeziehungen bestehen
- Assoziationen beschreiben gleichartige Objektbeziehungen zwischen Klassen
- Objektbeziehung ist Instanz einer Assoziation

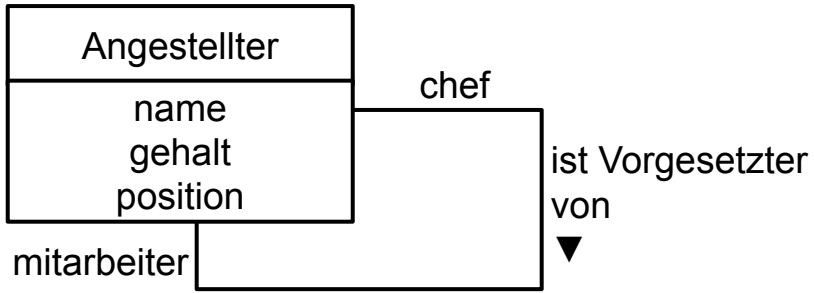




6.5. Einführung in UML: Assoziationen

Reflexive Assoziationen, Rollennamen

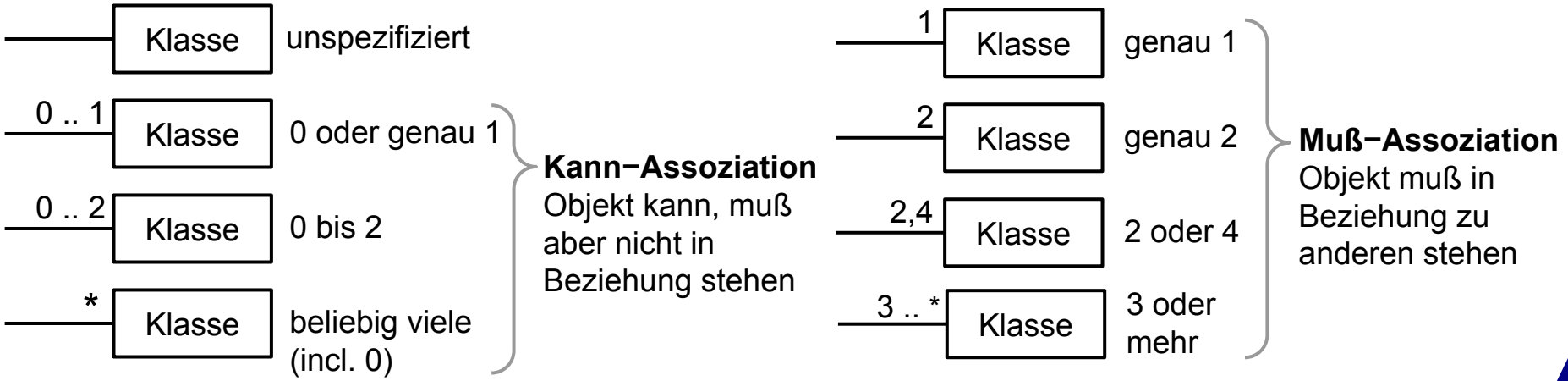
- An einer Objektbeziehung können auch zwei Objekte derselben Klasse beteiligt sein → führt zu reflexiver Assoziation zwischen Klassen
- Neben Assoziationsnamen auch Rollennamen möglich
  - Assoziationsname: Bedeutung der Assoziation
  - Rollenname: Bedeutung einer Klasse in der Assoziation
- Beispiel reflexive Assoziation:
  - Assoziationsname:
    ist Vorgesetzter von
  - Rollenname:
    chef, mitarbeiter



6.5. Einführung in UML: Assoziationen

Multiplizität von Assoziationen

- Assoziation sagt zunächst nur, dass ein Objekt andere Objekte kennen kann
- Multiplizität legt fest, wieviele Objekte ein Objekt kennen kann (oder muss)
- Die Multiplizität wird am Ende der Assoziations-Linie notiert:



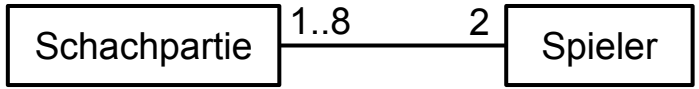
6.5. Einführung in UML: Assoziationen

Multiplizität: Beispiele

- Beispiel Bank:

  - ein Kunde muss mindestens ein Konto besitzen
  - ein Konto gehört zu genau einem Kunden
  - wenn der Kunde gelöscht wird, muss auch das Konto gelöscht werden
  - ein Kunde kann beliebig viele Depots besitzen

- Beispiel Simultan−Schach:

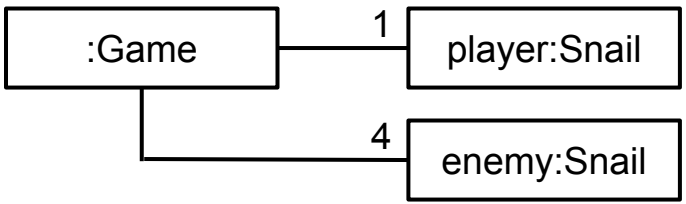


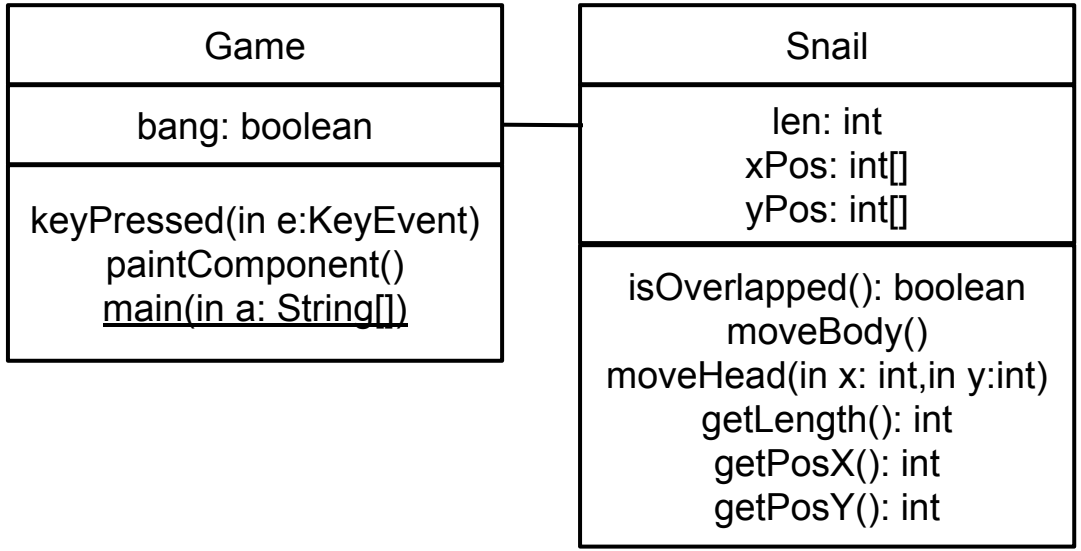
  - jede Schachpartie wird von zwei Spielern gespielt
  - ein Spieler spielt 1 bis 8 Partien gleichzeitig

6.5. Einführung in UML: Assoziationen

Multiplizität: Beispiele

- Beispiel SnailGamev1:

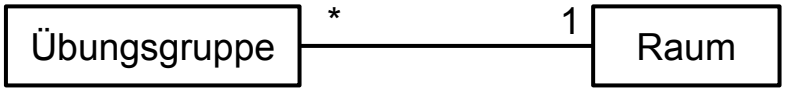




6.5. Einführung in UML: Assoziationen

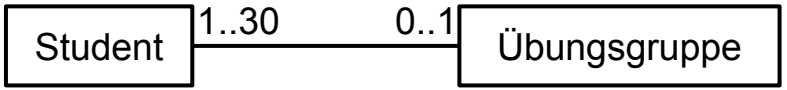
Umsetzung von Assoziationen in Java

- Muß-Assoziation




```
class Übungsgruppe {
  Raum übungsraum;
  // ...
}
class Raum {
  // ...
}
```

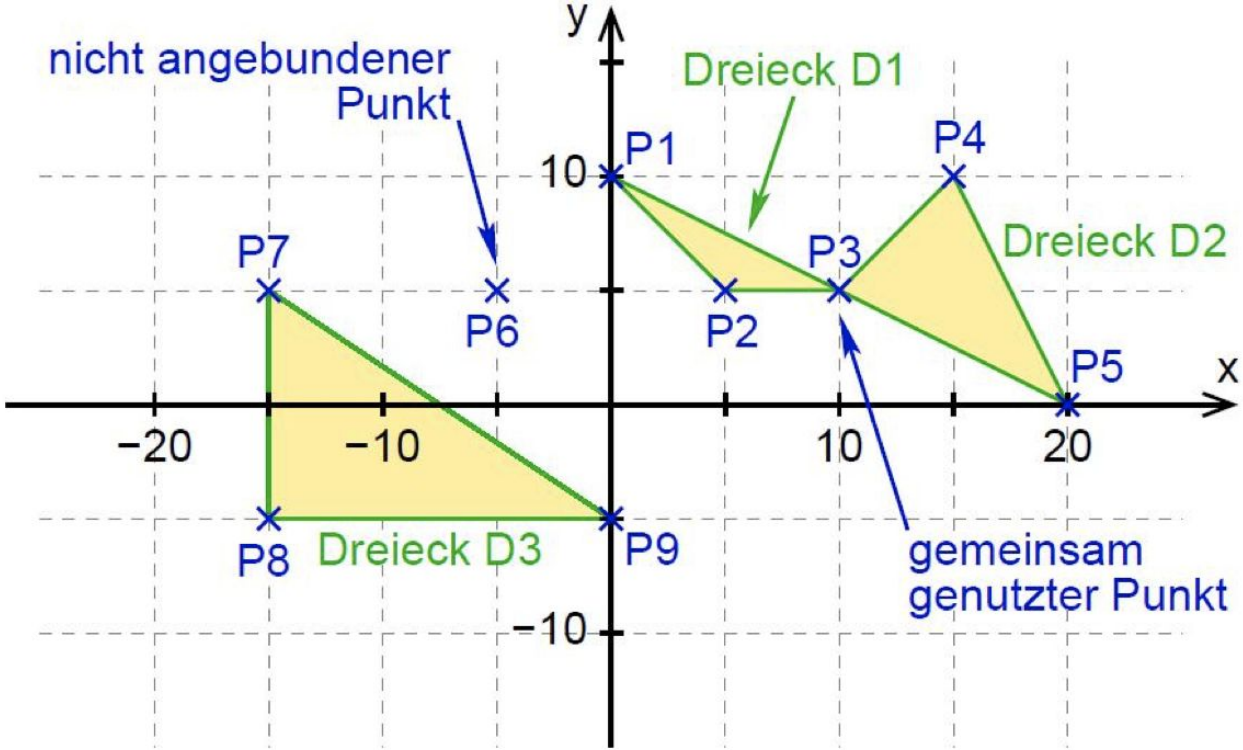
- Kann-Assoziation




```
class Student {
  Übungsgruppe gruppe;
  // ...
}
class Übungsgruppe {
  Student[] teilnehmer;
  // ...
}
```

6.5. Einführung in UML: Assoziationen

Beispiel

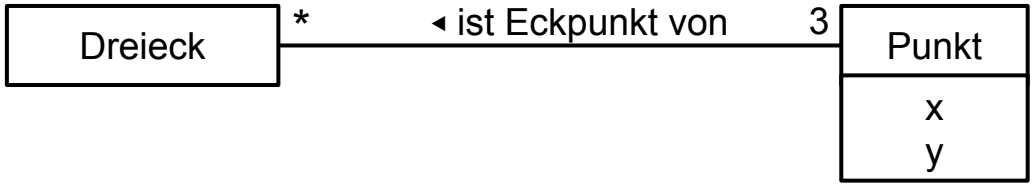


6.5. Einführung in UML: Assoziationen

Beispiel

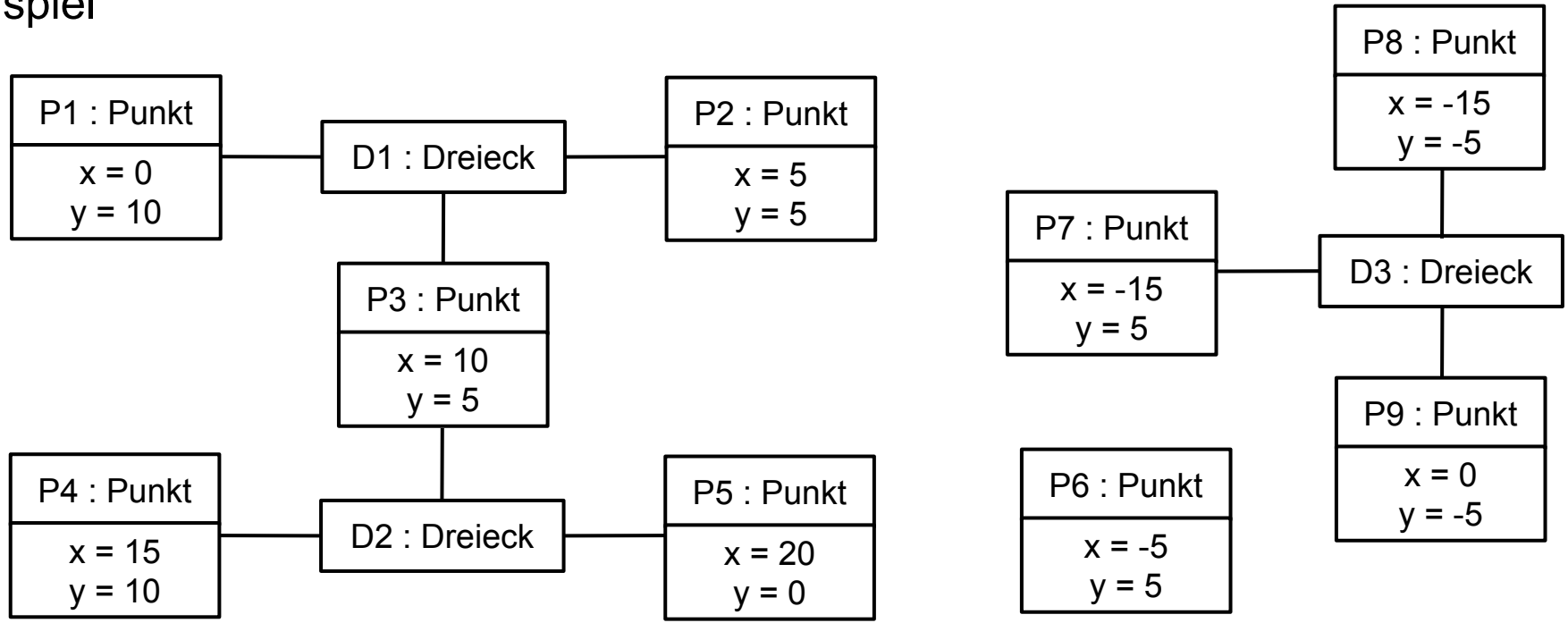
Klassendiagramm

- jedes Dreieck steht mit 3 Punkten in Verbindung
- ein Punkt ist Teil von beliebig vielen Dreiecken



6.5. Einführung in UML: Assoziationen

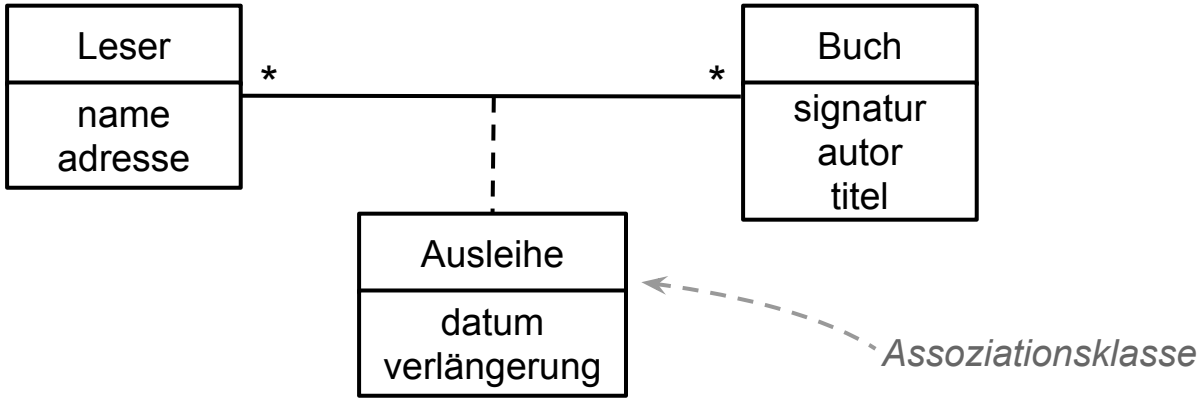
Beispiel



6.5. Einführung in UML: Assoziationen

Assoziationsklassen

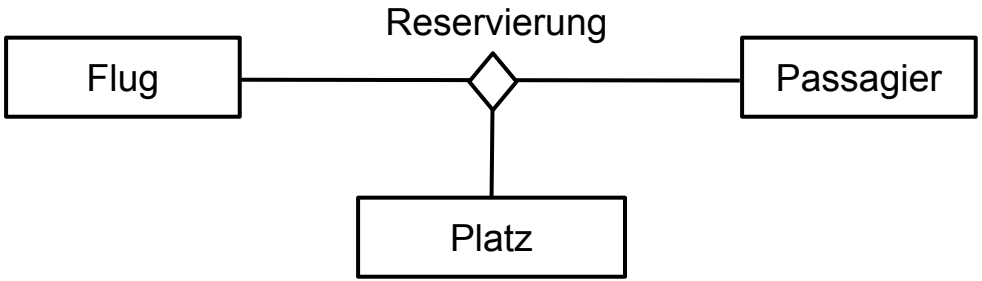
- Manchmal hat auch eine Assoziation Eigenschaften und Verhalten
- Dann: Assoziation wird explizit als Klasse modelliert
- Beispiel:



6.5. Einführung in UML: Assoziationen

Mehrstellige (n-äre) Assoziationen

- Assoziationen sind auch zwischen mehr als zwei Klassen möglich
- Darstellung am Beispiel einer ternären (dreistelligen) Assoziation:
  eine Reservierung ist eine Beziehung zwischen Passagier, Flug und Platz



6.5. Einführung in UML: Assoziationen

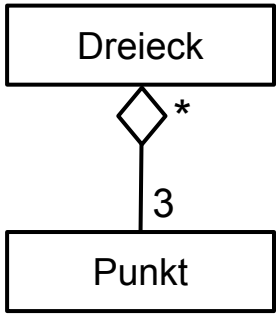
Aggregation und Komposition

- Häufige Abstraktion im täglichen Leben: Teile/Ganzes-Beziehung
  - ”besteht aus“ bzw. ”ist Teil von“
  - z.B.: ”Ein Auto besteht aus einer Karosserie, 4 Rädern, …“
- Aggregation: Teile existieren selbständig und können (gleichzeitig) zu mehreren Aggregat-Objekten gehören
- Komposition: starke Form der Aggregation
  - Teil-Objekt gehört zu genau einem Komposit-Objekt
    - es kann nicht Teil verschiedener Komposit-Objekte sein
    - es kann nicht ohne sein Komposit-Objekt existieren
  - Beim Erzeugen (Löschen) des Komposit-Objekts werden auch seine Teil-Objekte erzeugt (gelöscht)

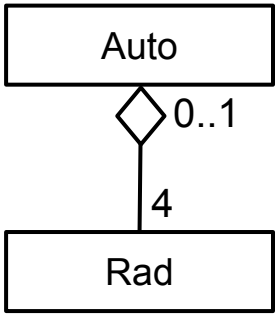
6.5. Einführung in UML: Assoziationen

Aggregation und Komposition

- Darstellung Aggregation:

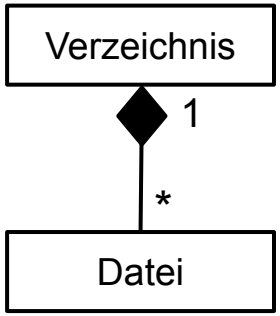


Ein Dreieck besteht immer aus 3 Punkten. Ein Punkt ist Teil von beliebig vielen (incl. 0) Dreiecken.



Ein Auto besteht u.a. aus 4 Rädern. Ein Rad ist Teil von höchstens einem Auto (es gibt auch Räder ohne Auto).

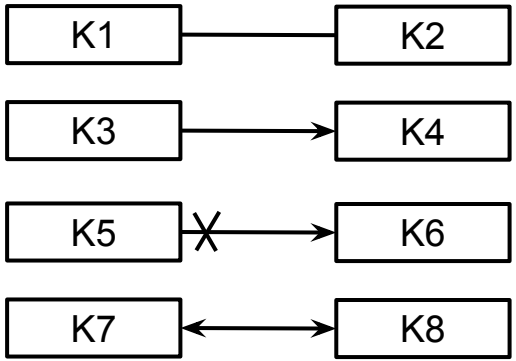
- Darstellung Komposition:



Ein Verzeichnis besteht aus beliebig vielen Dateien. Dateien stehen immer in einem Verzeichnis. Wird dieses gelöscht, so auch alle enthaltenen Dateien.

6.5. Einführung in UML: Assoziationen und Navigierbarkeit

- Im Entwurf wird zusätzlich die Navigierbarkeit modelliert:
- Assoziation von A nach B navigierbar => Objekte von A können auf Objekte von B zugreifen (aber nicht notwendigerweise umgekehrt)
- Darstellung in UML:



Navigierbarkeit ist unspezifiziert

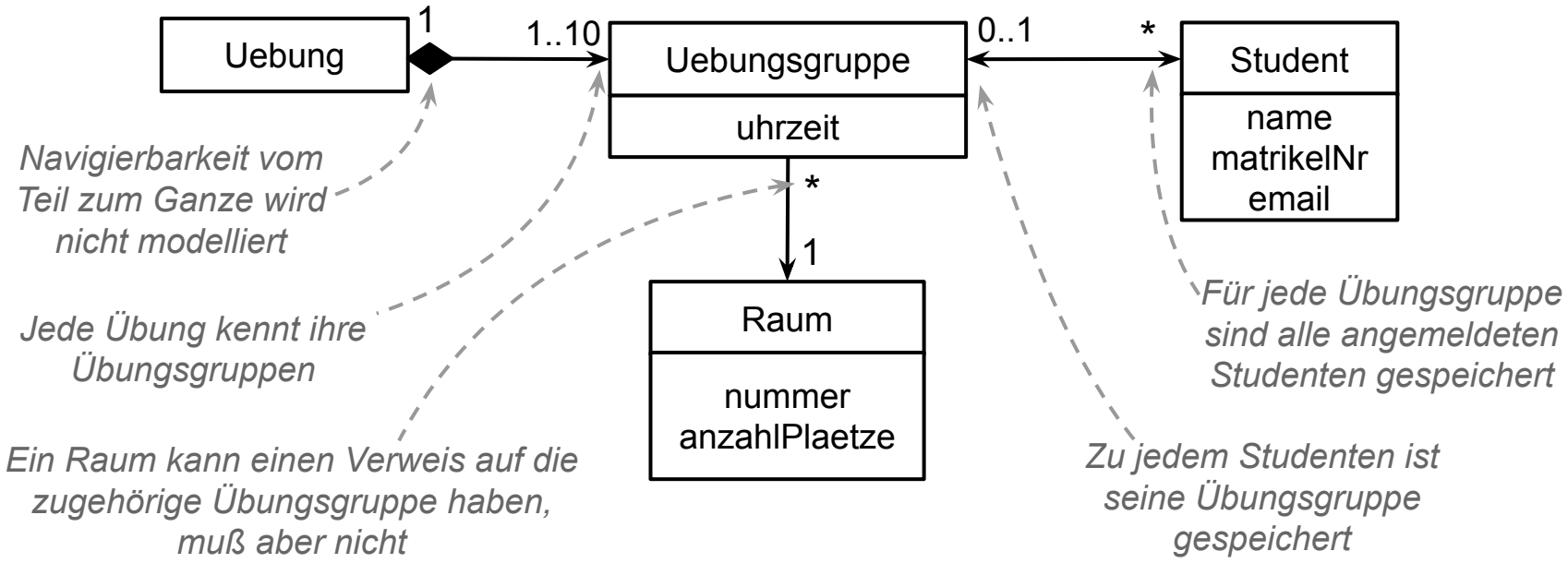
K3-Objekte können auf K4-Objekte zugreifen, keine Aussage über umgekehrte Richtung

K5-Objekte können auf K6-Objekte zugreifen, aber nicht umgekehrt

Bidirektionale Assoziation: K7-Objekte können auf K8-Objekte zugreifen und umgekehrt

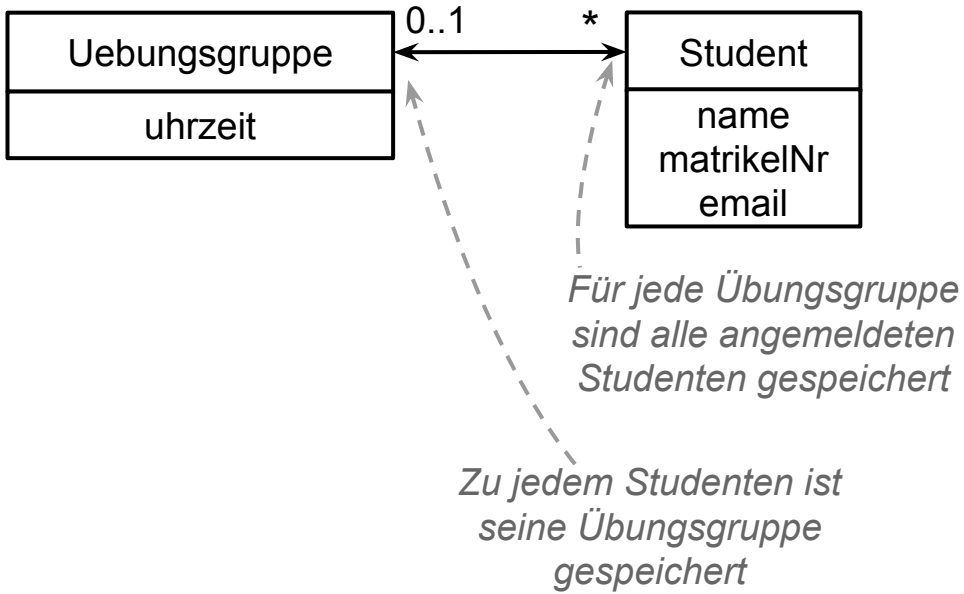
6.5. Einführung in UML: Assoziationen und Navigierbarkeit

- Beispiel:



6.5. Einführung in UML: Assoziationen und Navigierbarkeit

- Beispiel:



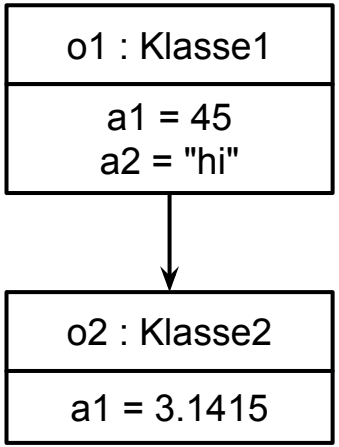

```
class Uebungsgruppe {
  Student[] teilnehmer;
}
class Student {
  Uebungsgruppe gruppe;
}
// ...
Student student = new Student();
// ...
if (student.gruppe==null) {
  // ...
} else {
  // ...
}
```

6.5. Einführung in UML: Assoziationen und Navigierbarkeit

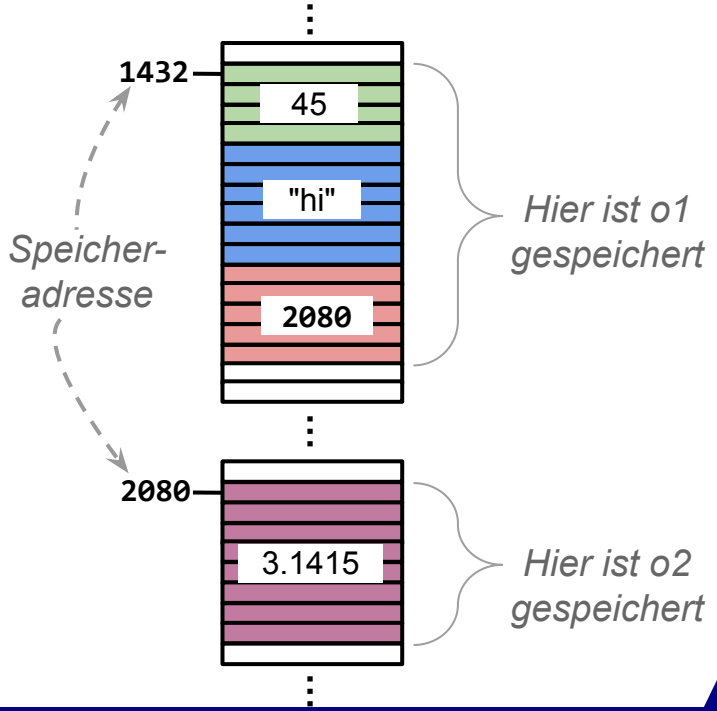
- Was bedeutet die Navigierbarkeit für die Realisierung von Assoziationen?
- K1 → K2 bedeutet, dass das K1-Objekt das K2-Objekt (bzw. die K2-Objekte) ”kennen“ muss: das K1-Objekt muss eine ***Referenz*** (auch Verweis, Zeiger) auf das K2-Objekt speichern
- In der Programmierung ist eine Referenz ein spezieller Wert, über den ein Objekt eindeutig angesprochen werden kann, z.B. Adresse des Objekts im Speicher des Rechners
- Referenzen können wie normale Werte benutzt werden:
  - Speicherung in Attributen
  - Übergabe als Parameter / Ergebnis von Operationen

6.5. Einführung in UML: Assoziationen und Navigierbarkeit

- Beispiel:

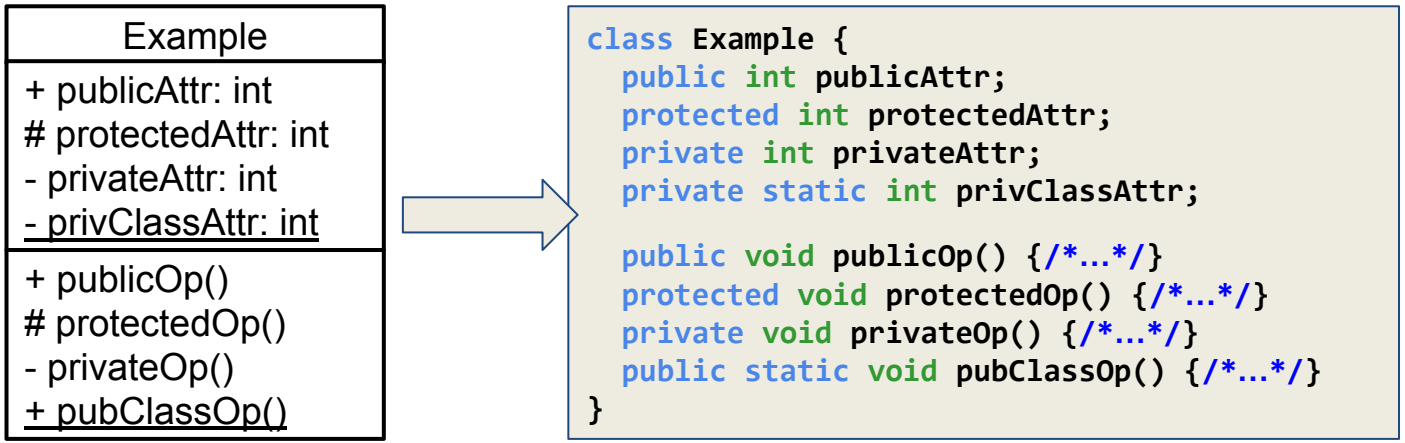



```
class Klasse1 {
  private int a1 = 45;
  private String a2 = "hi";
  public Klasse2 k2;
}
class Klasse2 {
  private double a1;
  public Klasse2(double a1) {
    this.a1 = a1;
  }
}
// ...
Klasse1 o1 = new Klasse1();
Klasse2 o2 = new Klasse2(3.1415);
o1.k2 = o2;
```



6.5. Einführung in UML: Sichtbarkeit

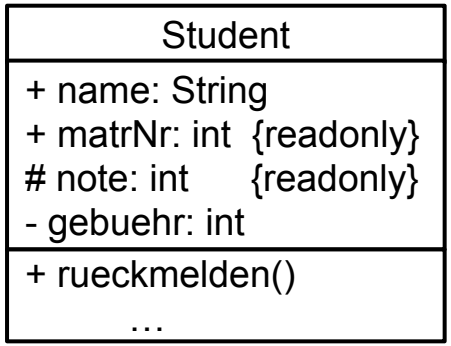
- **public** (öffentlich): sichtbar für alle Klassen (auf Attribut kann von allen Klassen aus zugegriffen werden, Operation kann von allen Klassen aufgerufen werden)
- **private** (privat): sichtbar nur innerhalb der Klasse (kein Zugriff/Aufruf durch andere Klassen möglich)
- **protected** (geschützt): sichtbar nur innerhalb der Klasse und ihren Unterklassen



6.5. Einführung in UML: Sichtbarkeit

Hinweise zu Get- und Set-Methoden

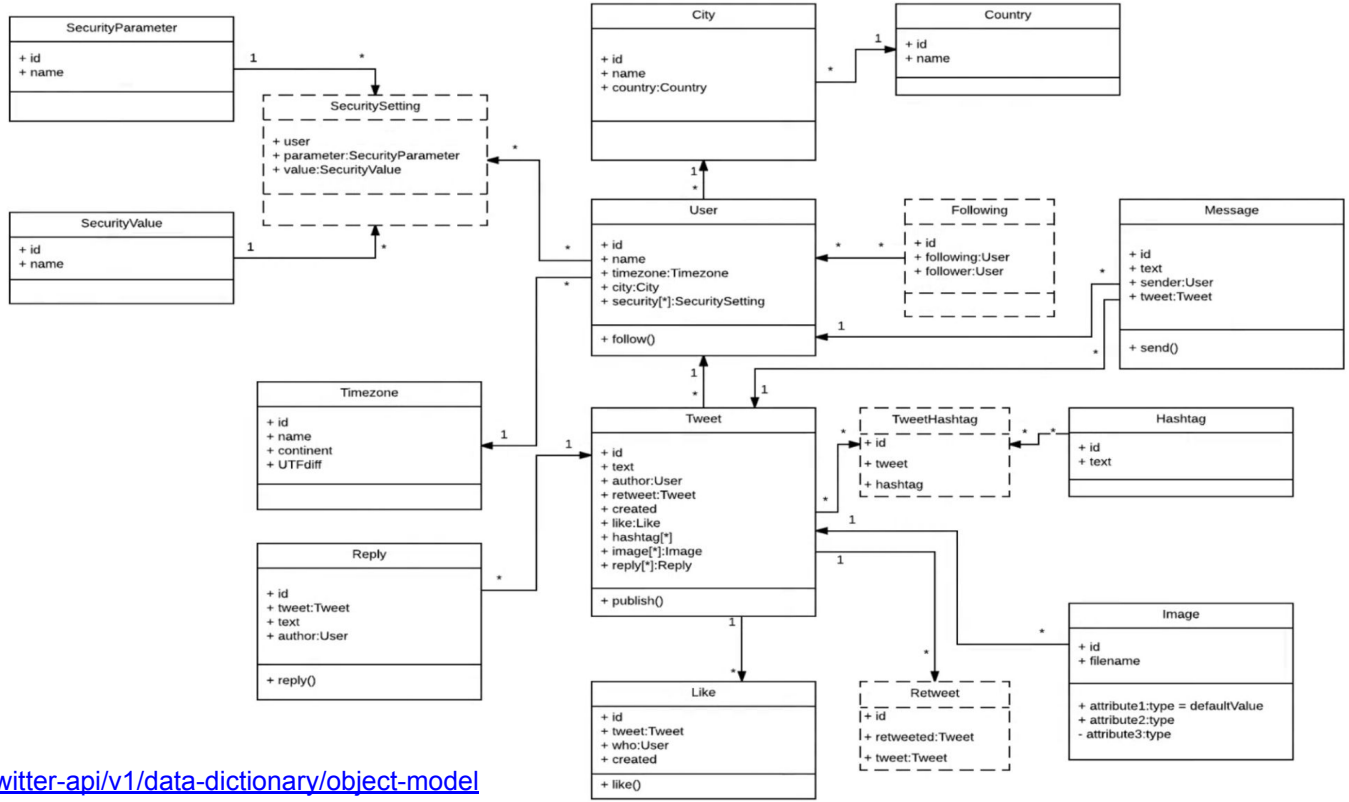
- Get- und Set-Methoden (Namenskonvention: getXxxx(), setXxxx()) werden i.d.R. nicht im Klassendiagramm dargestellt
- Die Sichtbarkeit des Attributs im Klassendiagramm bestimmt die Sichtbarkeit der Get- und Set-Methoden im Java-Code, im Java-Code ist das Attribut selbst immer private
- Bei readOnly-Attributen: Set-Methode ist private bzw. kann auch ganz fehlen




```
class Student {
  private String name;
  private int matrNr;
  private int note;
  private int gebuehr;
  // Hier keine Get- oder Set- Methoden
  // für gebuehr, da privates Attribute

  public String getName() {/*...*/}
  public void setName(String n) {/*...*/}

  public int getMatrNr() {/*...*/}
  // Hier kein setMatrNr: matrNr wird
  // bei Objekterzeugung direkt gesetzt

  protected int getNote() {/*...*/}
  private void setNote(int n) {/*...*/}
}
```

(https://ww.devcamp.com/trails/uml-foundations/campsites/class-diagrams/guides/4496d36d-ab10-475b-94b8-7a33c89bed4e)

6.6. Beispiel: Twitter



details:
https://developer.twitter.com/en/docs/twitter-api/v1/data-dictionary/object-model

Objektorientierte und Formale Programmierung
-- Java Grundlagen --

7. Vererbung und Polymorphie

7.1. Generalisierung in UML und Java

- Generalisierung: Beziehung zwischen einer allgemeineren Klasse (Basisklasse, Oberklasse) und einer spezialisierteren Klasse (Unterklasse)
  - die spezialisierte Klasse ist konsistent mit der Basisklasse, enthält aber zusätzliche Attribute, Operationen und / oder Assoziationen
  - ein Objekt der Unterklasse kann überall da verwendet werden, wo ein Objekt der Oberklasse erlaubt ist
- Nicht nur: Zusammenfassung gemeinsamer Eigenschaften und Verhaltensweisen,
  sondern immer auch: Generalisierung im Wortsinn
  - jedes Objekt der Unterklasse ist ein Objekt der Oberklasse
- Generalisierung führt zu einer Klassenhierarchie

7.1. Generalisierung in UML und Java

Beispiel Generalisierung: Angestellte, Studenten und studentische Hilfskräfte

- Modellierung als unabhängige Klassen:

Angestellter
personalnr name adresse geburtsdatum gehalt bankverbindung
druckeAdresse() überweiseGehalt()

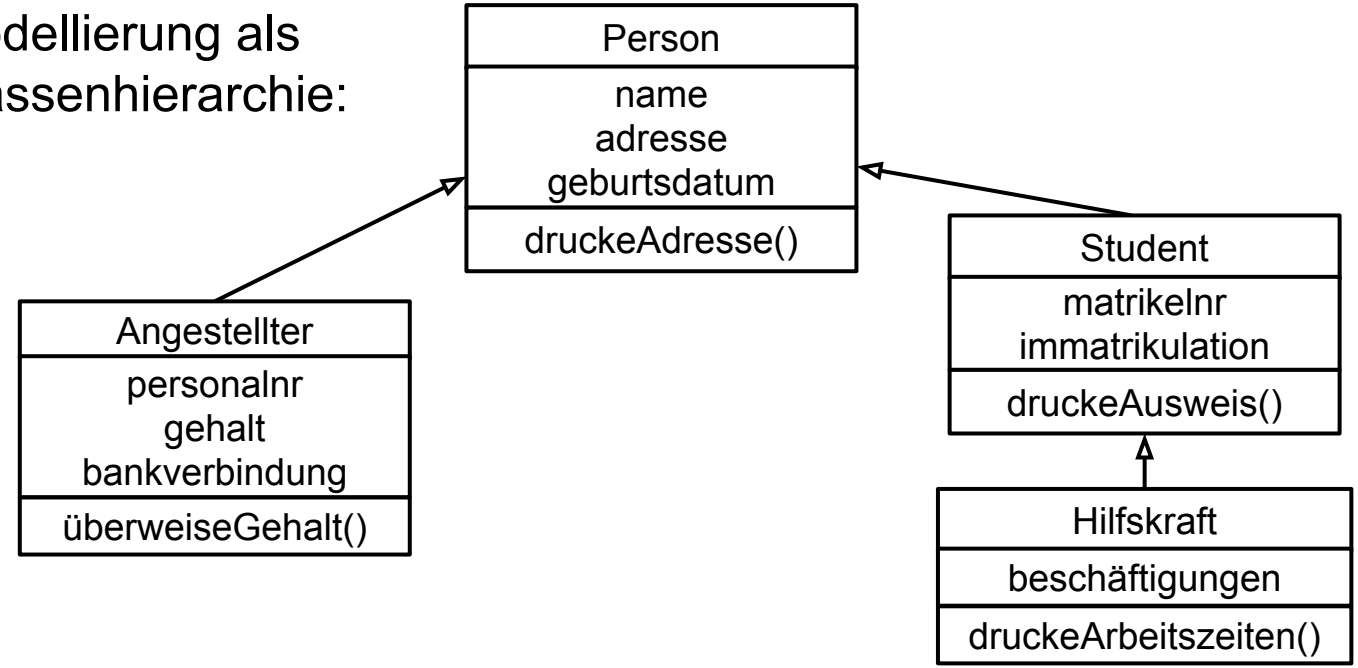
Student
matrikelnr name adresse geburtsdatum immatrikulation
druckeAdresse() druckeAusweis()

Hilfskraft
matrikelnr name adresse geburtsdatum immatrikulation beschäftigungen
druckeAdresse() überweiseGehalt() druckeArbeitszeiten()

7.1. Generalisierung in UML und Java

Beispiel Generalisierung: Angestellte, Studenten und studentische Hilfskräfte

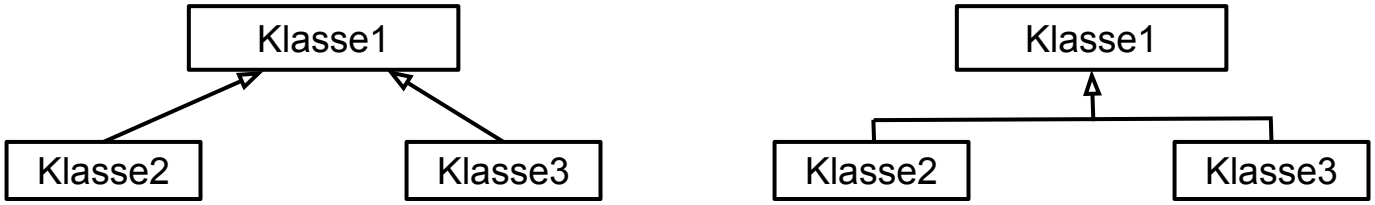
- Modellierung als Klassenhierarchie:



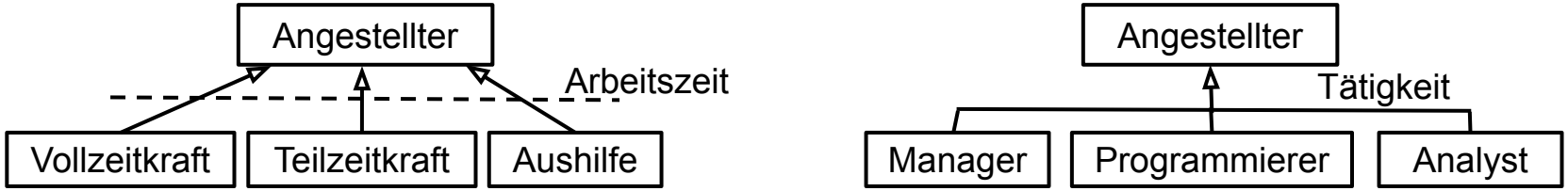
7.1. Generalisierung in UML und Java

Beispiel Generalisierung: Angestellte, Studenten und studentische Hilfskräfte

Darstellung:



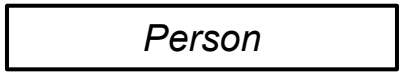
- Ein zusätzlicher Diskriminator (Generalisierungsmenge) kann das Kriterium angeben, nach dem klassifiziert wird:



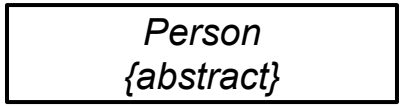
7.1. Generalisierung in UML und Java

Abstrakte Klassen und Operationen

- Im vorherigen Beispiel wurden im Modell nur Personen betrachtet, die entweder Angestellter, Student oder Hilfskraft sind
- Die neue Basisklasse ”Person“ wird daher als *abstrakte* Klasse modelliert
- von einer abstrakten Klasse können keine Instanzen (Objekte) erzeugt werden
- Darstellung:
  - Klassenname in Kursivschrift:



  - Oder für handschriftliche Diagramme:

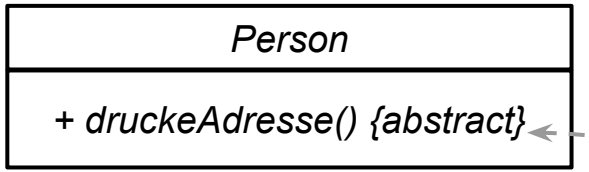



```
abstract class Person {
  String name;
  String adresse;
  Date geburtsdatum;
  public void druckeAdresse() {
    //...
  };
}
```

7.1. Generalisierung in UML und Java

Abstrakte Klassen und Operationen

- Eine abstrakte Operation einer Klasse wird von der Klasse nur deklariert, nicht aber implementiert
  - die Klasse legt nur Signatur und Ergebnistyp fest
  - die Implementierung muss in einer Unterklasse durch überschreiben der ererbten Operation erfolgen
- Abstrakte Operationen dürfen nur in abstrakten Klassen auftreten
- Darstellung:



Kursivschrift oder Zusatz {abstract}

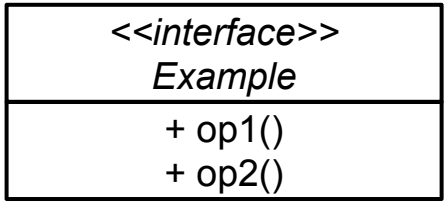
Abstrakte Operationen besitzen keinen Rumpf (d.h. keine Implementierung)

```
abstract class Person {
  String name;
  String adresse;
  Date geburtsdatum;
  public abstract void druckeAdresse();
}
```

7.1. Generalisierung in UML und Java

Interfaces (Schnittstellen)

- Eine (abstrakte) Klasse mit abstrakten Operationen
- Beschreibt eine Menge von Signaturen von Operationen
- Java erlaubt in Schnittstellen nur *öffentliche*, *unveränderliche* und *initialisierte* Klassenattribute: `public static final int M_PI = 3;`,
  `public`, **`static`** und **`final`** können dann auch entfallen: `int M_PI = 3;`
- Die Operationen einer Schnittstelle sind immer abstrakt und öffentlich
  `public` kann in Java auch entfallen
- Darstellung:




```
interface Example {
  public void op1();
  public void op2();
}
```

```
interface Example {
  void op1();
  void op2();
}
```

7.1. Generalisierung in UML und Java

Interfaces (Schnittstellen)

- Schnittstellen definieren ”Dienstleistungen“ für aufrufende Klassen, sagen aber nichts über deren Implementierung aus
- funktionale Abstraktionen, die festlegen was implementiert werden soll, aber nicht wie
- Schnittstellen realisieren damit das Geheimnisprinzip in der stärksten Form:
  - Java-Klassen verhindern über Sichtbarkeiten zwar den Zugriff auf Interna der Klasse, ein Programmierer kann diese aber trotzdem im Java-Code der Klasse lesen
  - der Java-Code einer Schnittstelle enthält nur die öffentlich sichtbaren Definitionen, nicht die Implementierung

7.1. Generalisierung in UML und Java

Interfaces (Schnittstellen)

- Schnittstellen sind von ihrer Struktur und Verwendung her praktisch identisch mit abstrakten Klassen:
  - sie können wie abstrakte Klassen nicht instanziiert werden
  - Referenzen auf Schnittstellen und auch abstrakte Klassen sind aber möglich, sie können auf Objekte zeigen, die die Schnittstelle implementieren
- Klassen können von Schnittstellen ”erben“
  - ”vererbt“ werden nur die Signaturen der Operationen
  - die Klassen müssen diese Operationen selbst implementieren
  - man spricht in diesem Fall von einer Implementierungs-Beziehung statt von Generalisierung

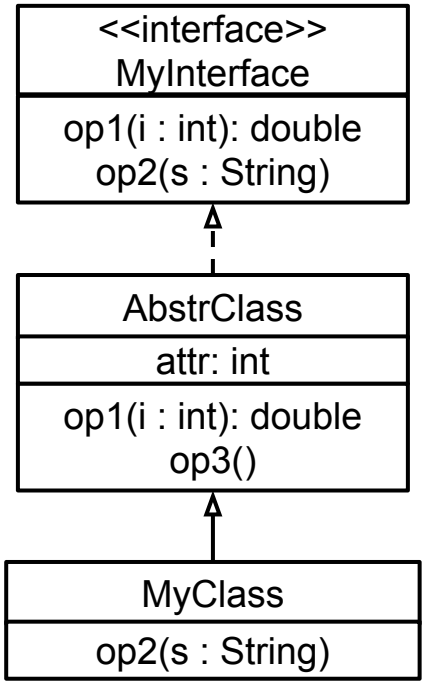
7.1. Generalisierung in UML und Java

Interfaces (Schnittstellen)

- Die Implementierungs-Beziehung **`implements`** besagt, dass eine Klasse die Operationen einer Schnittstelle implementiert
  - die Klasse erbt die abstrakten Operationen, d.h. deren Signaturen incl. Ergebnistyp: Diese müssen dann geeignet überschrieben werden
  - werden nicht alle Operationen überschrieben (d.h. implementiert), so bleibt die erbende Klasse abstrakt
  - die überschreibenden Operationen müssen öffentlich sein
  - die Klasse kann zusätzlich weitere Operationen und Attribute definieren
- Eine Klasse kann mehrere Schnittstellen implementieren (eine Art Mehrfachvererbung, die auch in Java erlaubt is)

7.1. Generalisierung in UML und Java

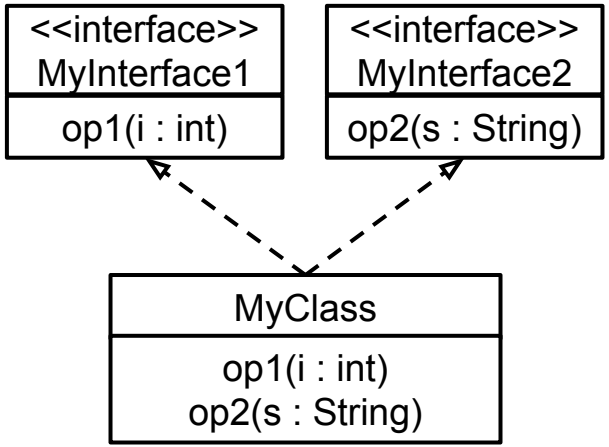
Interfaces (Schnittstellen): Die Implementierungs-Beziehung **implements**




```
interface MyInterface {
  public double op1(int i);
  public void op2(String s);
}

abstract class AbstrClass implements MyInterface {
  protected int attr;
  public double op1(int i) { /* ... */ }
  public void op3() { /* ... */ }
}

class MyClass extends AbstrClass {
  public void op2(String s) { /* ... */ }
}
```

7.1. Generalisierung in UML und Java

Interfaces (Schnittstellen): Implementierung mehrerer Schnittstellen

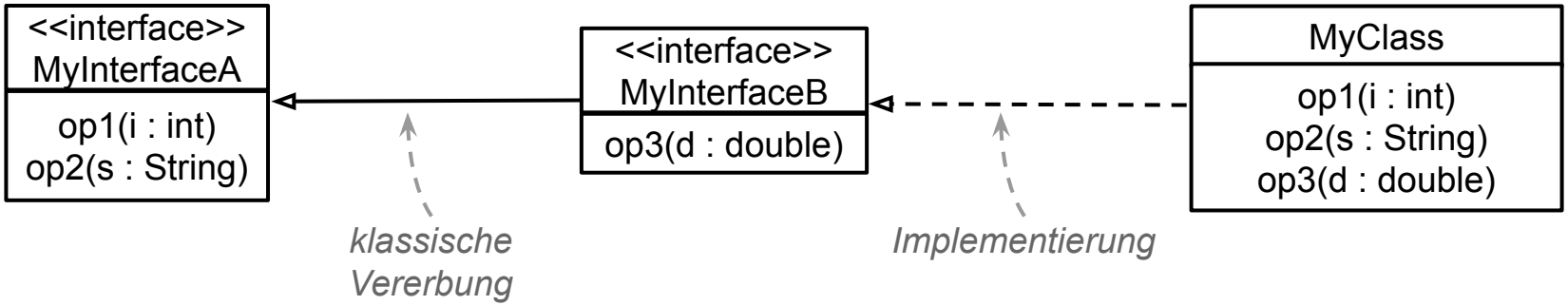



```
interface MyInterface1 {
  public void op1(int i);
}

interface MyInterface2 {
  public void op2(String s);
}

class MyClass implements MyInterface1, MyInterface2 {
  public void op1(int i) { /* ... */ }
  public void op2(String s) { /* ... */ }
}
```

7.1. Generalisierung in UML und Java

Interfaces (Schnittstellen): Vererbung

- Schnittstellen können von anderen Schnittstellen erben (analog zu Klassen)
- Darstellung in UML und Java wie bei normaler Vererbung
- Die implementierende Klasse muss die Operationen ”ihrer“ Schnittstelle und aller Ober-Schnittstellen implementieren:



7.1. Generalisierung in UML und Java

Vererbung

- Eine Unterklasse übernimmt (erbt) von ihren Oberklassen
  - alle *Attribute* und *Klassenattribute*, auch deren Anfangswert wenn definiert
  - alle *Operationen* und *Klassenoperationen*, d.h. alle Operationen einer Oberklasse können auch auf ein Objekt der Unterklasse angewendet werden
  - alle Assoziationen
- Die Unterklasse kann zusätzliche Attribute, Operationen und Assoziationen hinzufügen, aber ererbte *nicht löschen*
- Die Unterklasse kann das Verhalten *neu definieren*, indem sie Operationen der Oberklasse *überschreibt (*d.h. eine Operation gleichen Namens neu definiert)