TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

Objektorientierte Modellierung

Zur objektorientierten Modellierung

Wir geben hier nur eine kurze Einführung; OO-Modellierung ist Gegenstand der Vorlesung SE 2.

Die Analyse einer Problemstellung führt zu einem *Modell* des Ausschnitts der Welt, der zur Lösung geeignet ist.

Das Modell beschreibt die wichtigen Eigenschaften des zu entwickelnden Systems. Es orientiert sich zunächst an der Anwendung und nicht der Realisierung.

Aspekte der Modellierung

Bei objektorientierter Modellierung ist zu klären:

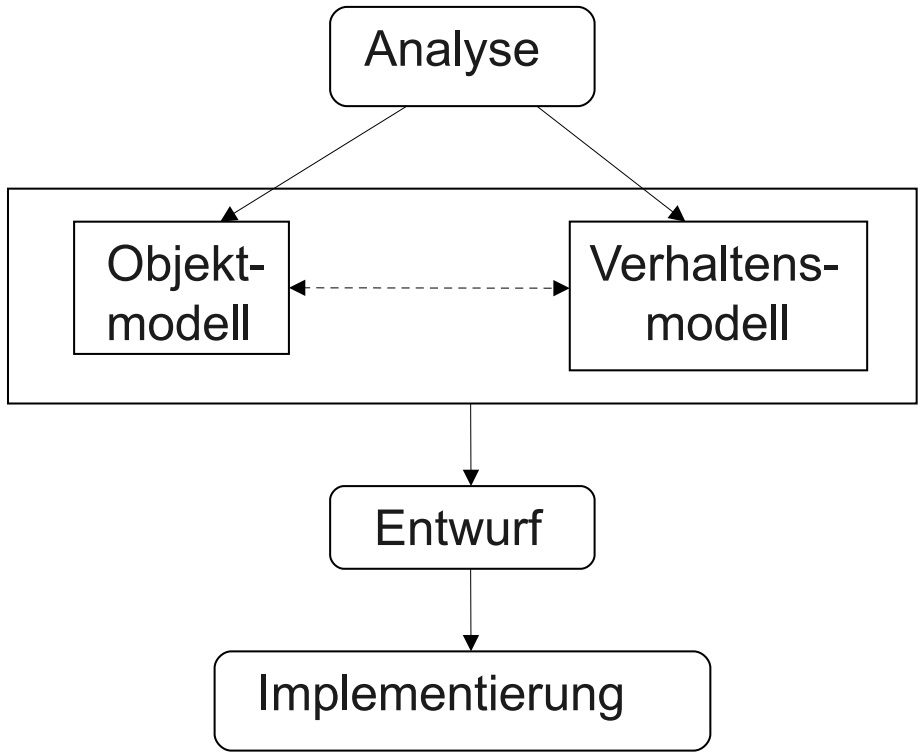
1. Welche Objekte werden benötigt?
2. Welche Beziehungen gibt es zwischen den Objekten?
3. Welche Eigenschaften besitzen die Objekte?
4. Wie lassen sich die Objekte klassifizieren?
5. Wie werden die Objekte angewendet?
6. Was ist das Verhalten der Objekte?

Das *Objekt-/Klassenmodell* liefert die Antworten zu 1-4.

Anwendungsfälle beantworten Frage 5.

Das *Verhaltensmodell* klärt Frage 6.

Objektorientierte Analyse in Umfeld der objektorientierten Softwareentwicklung



Begriffsklärung: Objekt-, Verhaltensmodell

Das **Objektmodell** beschreibt

- die relevanten Klassen von Objekten,
- welche Dienste/Nachrichten/Operationen bereitgestellt werden,
- die Beziehungen zwischen den Objekten.

Das **Verhaltensmodell** beschreibt

- die Wirkungsweise der Methoden,
- die möglichen Zustände von Objekten,
- das Ablaufverhalten und die Interaktion zwischen den Objekten.

Objektorientierte Analyse: Ein Beispiel

Aufgabe:

Entwickle ein Informationssystem für eine Universität.

Grober Leistungsumfang:

Universitäten bestehen aus Studierenden und Kursen mit folgenden Beziehungen:

- Jede(r) Studierende hat einen Namen, eine eindeutige Matrikelnummer und belegt eine Reihe von Kursen.
- Jeder Kurs hat einen Titel und eine maximale Teilnehmerzahl.

Typische **Anwendungsfälle** (engl. *use cases*) sind:

- Die Universität kann Studierende im- und exmatrikulieren.
- Studierende können angebotene Kurse belegen.
- Man kann eine Liste aller Studierenden der Universität sowie einzelner Kurse ausgeben.

Ermitteln des Verhaltens

Wir verfolgen einen *top-down* Ansatz:

Abstraktes Verhalten der Objekte $\Rightarrow$ Implementierung

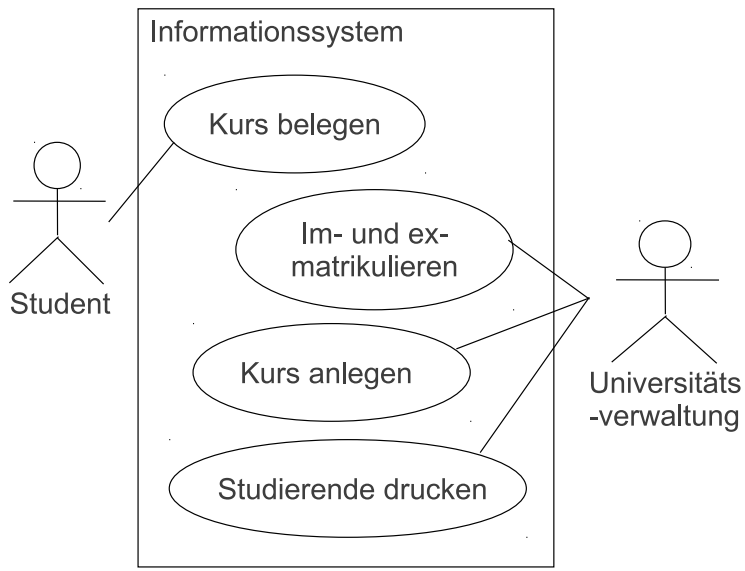
Beschreibung des Verhaltens:

- Skizzieren der wesentlichen Anwendungsfälle (*Use Cases*)
- Beschreibung der Nachrichten, die die Objekte verstehen (*Methodenschnittstelle*)

Anwendungsfälle (Use cases)

Typische Vorgänge, die mit dem zu realisierenden System durchgeführt werden.

Beispiel:



Studierende und Universitätsverwaltung sind Beispiele für **Akteure**.
Akteure können sein:

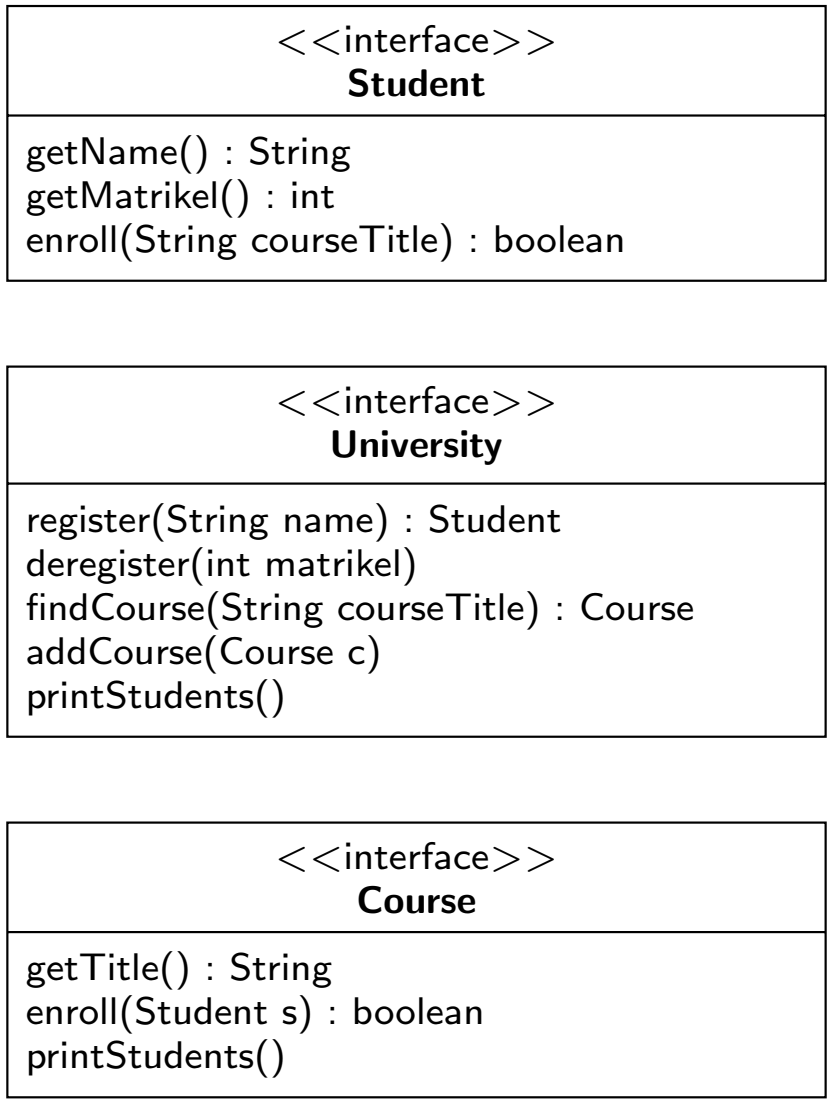
- Benutzer des Systems
- agierende Softwareteile der Systemumgebung

Methodenschnittstellen

Schnittstellenbeschreibungen sind ein wichtiger Baustein in der Modellierung und Implementierung von objekt-orientierten Systemen.

- Sie bieten klare Vereinbarungen (*contracts*) über die Interaktion von Klassen und Objekten.
- Sie klassifizieren Objekte nach ihrem Verhalten.
- Sie verbergen die tatsächliche Implementierung ($\rightarrow$ Information hiding).
- Sie verhindern (ungewollte) Abhängigkeiten zwischen Klassen.
- Sie vereinfachen die Zusammenarbeit von Entwicklerteams und ermöglichen eine klare Aufgabenteilung.
- Sie erlauben ein einfaches Verändern von Klassenimplementierungen, da nur wenig im Anwendungscode angepasst werden muss.

Beispiel


<<interface>>
Student
getName() : String
getMatrikel() : int
enroll(String courseTitle) : boolean
<<interface>>
University
register(String name) : Student
deregister(int matrikel)
findCourse(String courseTitle) : Course
addCourse(Course c)
printStudents()
<<interface>>
Course
getTitle() : String
enroll(Student s) : boolean
printStudents()

Interfaces in Java

Ein **Interface** deklariert einen Referenztyp T und beschreibt die öffentliche Schnittstelle, die alle Objekte von T haben.

Syntax:

```
InterfaceDecklaration →
        Modifikatorenliste interface ≪Bezeichner≫ extendsKlausel {
            MethodenDeklarationsListe
        }

extendsKlausel →
        extends TypListe
      | ϵ

TypListe →
        Typ , TypListe
      | Typ

MethodenDeklaration →
     TypOderVoid ≪Bezeichner≫ (FormaleParameter);
```

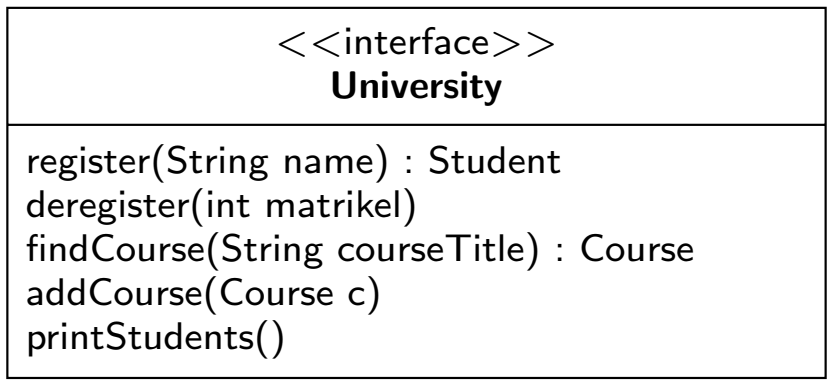
Beispiel: Deklaration von Interfaces

```
1  interface Course {
2    String getTitel();
3    boolean enroll(Student s);
4    void printStudents();
5  }
```

```
1  interface Student {
2    String getName();
3    int getMatrikel();
4    boolean enroll(String courseTitle);
5  }
```

Aufgabe I

Geben Sie eine Interface-Implementierung für Universitäten!



Aufgabe II

```
interface University {
  Student register(String name);
  void deregister(int matrikel);
  Course findCourse(String courseTitle);
  void addCourse(Course c);
  void printStudents();
}
```

Beispiel: Implementierung von Schnittstellen I

```java
import java.util.List;
import java.util.LinkedList;

class Seminar implements Course {
  private String title;
  private int capacity;
  private List<Student> students;

  Seminar(String title, int capacity) {
    this.title = title;
    this.capacity = capacity;
    this.students = new ArrayList<Student>();
  }
  public String getTitel() {
    return title;
  }
  public boolean enroll(Student s) {
    if (students.size() < capacity) {
      students.add(s);
      return true;
    }
    return false;
  }
  public void printStudents() {
    for(Student s : students) {
      System.out.println(s);
    }
  }
}
```

```java
interface Course {
  String getTitel();
  boolean enroll(Student s);
  void printStudents();
}
```

Beispiel: Implementierung von Schnittstellen II

```java
import java.util.List;
import java.util.ArrayList;

class Lecture implements Course {
  private String title;
  private String lecturer;
  private List<Student> students;

  Lecture(String title, String lecturer) {
    this.title = title;
    this.lecturer = lecturer;
    this.students = new ArrayList<Student>();
  }

  public String getTitel() {
    return title;
  }
  public String getLecturer() {
    return lecturer;
  }
  public boolean enroll(Student s) {
    return students.add(s);
  }
  public void printStudents() {
    for(Student s : students) {
      System.out.println(s);
    }
  }
}
```

```java
interface Course {
  String getTitel();
  boolean enroll(Student s);
  void printStudents();
}
```

Beispiel: Listen

In Kapitel 11 haben wir verschiedene Möglichkeiten gesehen, Listen mit Integer-Elementen zu implementieren.

- Schreiben Sie ein Interface `ListOfInt`, das den Schnittstellentyp für Listen mit Integer-Elementen definiert, das die Methoden `add`, `get`, `size` bereitstellt!
- Wie müssen die Klassen `IntList`, `DLIntList` und `ArrayList` abgeändert werden, damit sie dieses Interface implementieren?

Interface für Listen

Ein mögliches Interface für Listen mit Integer-Element ist:

```
1  interface ListOfInt {
2    void add(int i);
3    int get(int i);
4    int size();
5  }
```

Bei den Listen-Klassen muss angegeben werden, dass sie das Interface implementieren, die Implementierung selbst muss nicht verändert werden:

```
public class IntList      implements ListOfInt {...}
public class DLIntList    implements ListOfInt {...}
public class IntArrayList implements ListOfInt {...}
```

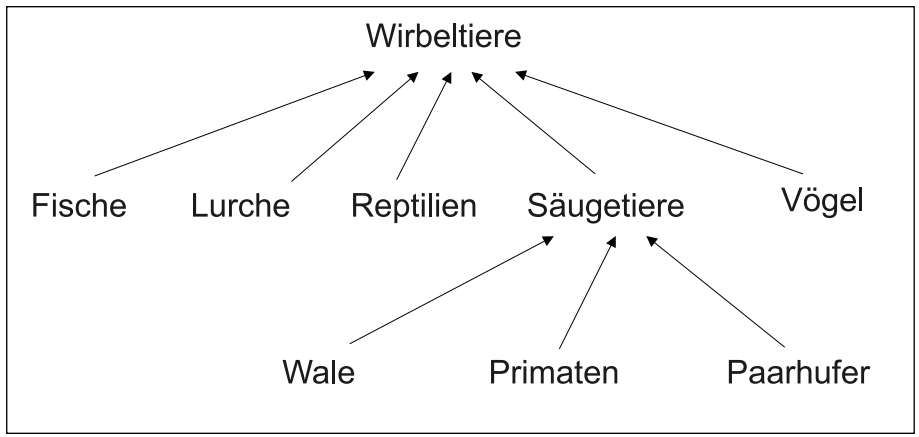
Klassifizieren von Objekten

Begriffsklärung: Klassifikation

Klassifikation ist eine zentrale Grundlage der objektorientierten Modellierung und Programmierung.

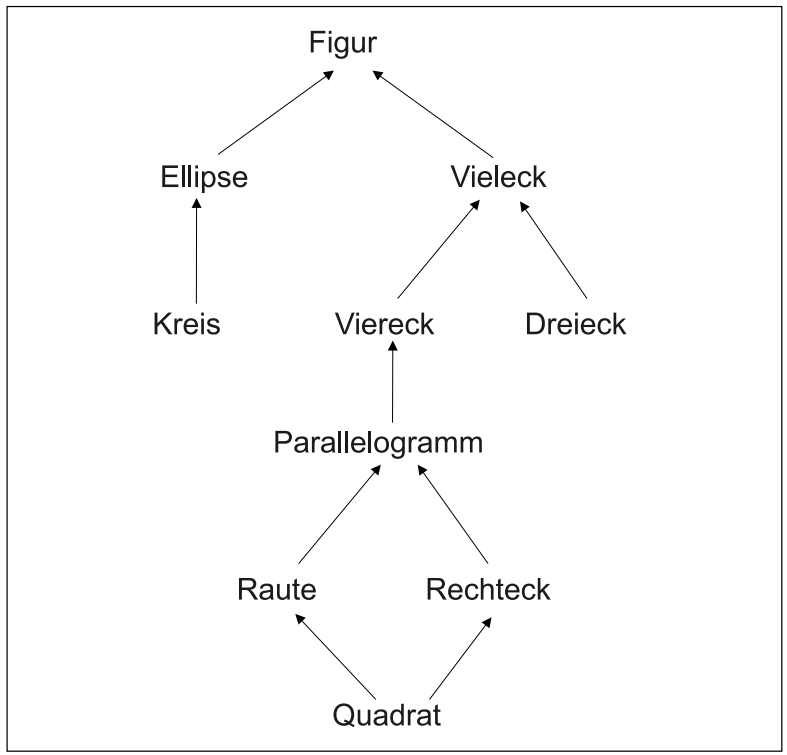
Klassifizieren ist eine allgemeine Technik, mit der Wissen über Begriffe, Dinge und deren Eigenschaften hierarchisch strukturiert wird. Das Ergebnis nennen wir eine **Klassifikation**.

Beispiele: Klassifikationen I



- Klassifikationen haben eine Richtung ("ist ein"-Beziehung) und dürfen keine Zyklen enthalten.
- Üblicherweise stehen die allgemeineren Begriffe oben, die spezielleren unten.
- Es gibt abstrakte Klassen (ohne "eigene" Objekte).

Beispiele: Klassifikationen II



Klassifikationen können baumartig oder DAG[1]-artig sein.

[1] DAG = directed acyclic graph

Klassifikation in der Softwaretechnik

Objekte lassen sich nach ihren Eigenschaften klassifizieren:

- Alle Objekte mit ähnlichen Eigenschaften werden zu einer Klasse zusammen gefasst.
- Die Klassen werden hierarchisch geordnet.

Klassifikation beruht auf:

- Schnittstellen der Klassen/Objekte
- Verhalten/Eigenschaften der Objekte

Klassifikationen und Typisierung

Ein Typ beschreibt Eigenschaften von Objekten bzw. Werten.

Ansatz:

- Realisiere jede Klasse / jeden Art einer Klassifikation im Programm durch einen Typ.

Übersicht: Typsystem von Java

Werte in Java:

- Elemente der elementaren Datentypen,
- Referenzen auf Objekte,
- der Wert `null`.

Jeder Wert in Java gehört zu mindestens einem Typ.

Typen in Java:

- die vordefinierten elementaren Basisdatentypen
- die durch Klassen deklarierten Typen
- die durch Interfaces deklarierten Typen
- Arraytypen mit Typkonstruktor `[]`

Auch jedes Objekt in Java hat (mindestens) einen Typen (Referenztyp).

In Java: Keine Unterscheidung zwischen dem Typ eines Objekts und dem Typ der Referenz auf dieses Objekt, da Unterscheidung durch Kontext klar.

Sub-/Supertypen

- Partielle Ordnung $\leq$ auf Typen
- Wenn $S \leq T$ gilt, dann sind alle Objekte vom Typ S auch vom Typ T.
- Begriffe:
  - S ist ein **Subtyp** von T, und T ist ein **Supertyp** von S.
  - Wenn $S \leq T$ und $S \neq T$, heißt S ein **echter** Subtyp von T.
  - Wenn $S < T$ und es kein U mit $S < U < T$ gibt, dann heißt S ein **direkter** Subtyp von T.

Typen als Mengen

Typen kann man als die Menge ihrer Objekte bzw. Werte auffassen.

Bezeichne $M(S)$ die Menge der Objekte vom Typ S.

Für Typen S und T gilt dann:

$$S \leq T \quad \text{impliziert} \quad M(S) \subseteq M(T)$$

Beispiel: Subtypbeziehungen

- Für die elementaren Datentypen in Java gelten z. B. die folgenden Subtypbeziehungen:

$$\texttt{float} < \texttt{double}$$
$$\texttt{int} < \texttt{long}$$
$$\texttt{long} < \texttt{float}$$

Verwendung von Subtypen

Objekte eines Typs S, der ein Subtyp eines Typs T ist ($S \leq T$), können an allen Stellen verwendet werden, an denen Objekte vom Typ T zulässig sind.

```
class Modulhandbuch {
  ...
  void addCourse(Course c) { ... }
}

// Benutzung:
Modulhandbuch mh = new Modulhandbuch();
mh.addCourse(new Lecture("Logik", "Poetzsch-Heffter"));
```

Frage

`Seminar` ist ein Subtyp von `Course`.
Welche der beiden Zuweisungen in der folgenden Methode wird vom Java-Compiler nicht akzeptiert?

```
void test(Course[] courses, Seminar[] seminars) {
    courses[0] = seminars[0];
    seminars[1] = courses[1];
}
```

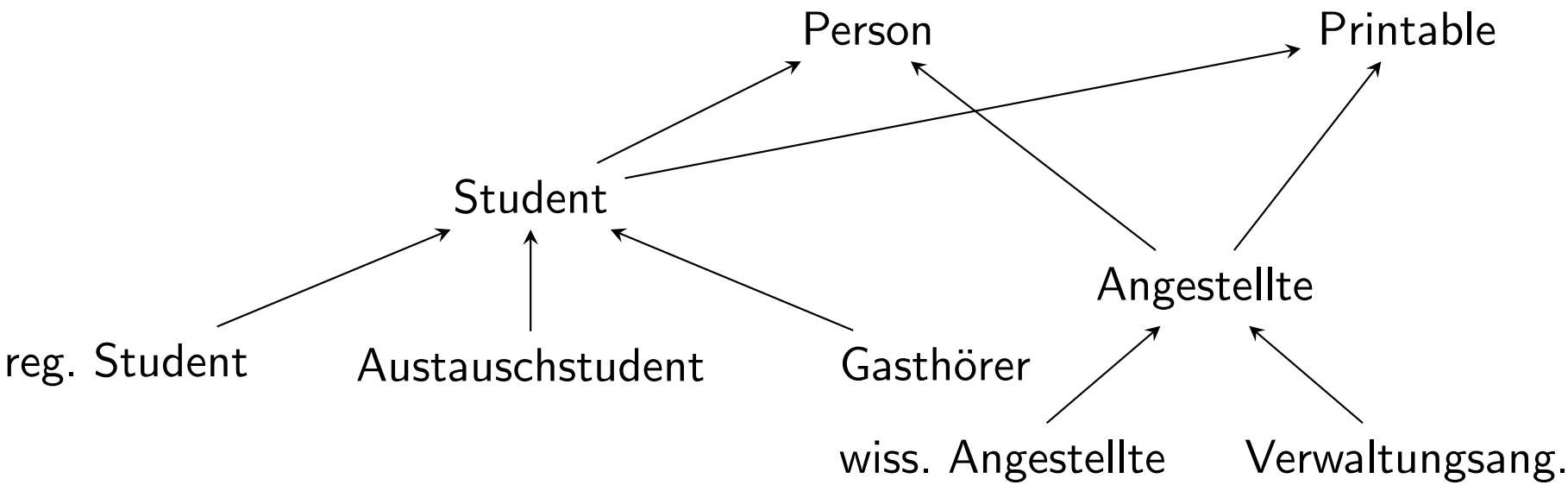
Die Klasse `Object`

Die Klasse `Object` spielt in Java eine spezielle Rolle.

Sie ist die Wurzel der Typhierarchie der Referenztypen und stellt einige wichtige Methoden (z.B. `toString()`) bereit.

- Alle Referenztypen sind Subtypen von `Object`.
- Alle Objekte (inkl. Arrays) erben die Methoden von `Object` ($\Rightarrow$ nächste Vorlesung "Vererbung").

Subtyping und Interface-Deklarationen I



Subtyping und Interface-Deklarationen II

```
interface Printable {
  void print();
}

interface Person {
  String getName();
  String getBirthdate();
}

interface Angestellte extends Person, Printable { ... }
interface Student     extends Person, Printable {... }

class WissAngestellte  implements Angestellte { ... }
class VerwAngestellte  implements Angestellte { ... }
class RegulaererStudent  implements Student { ... }
class AustauschStudent   implements Student { ... }
class Gasthoerer         implements Student { ... }
```

Subtyping und Interface-Deklarationen III

- Die Typen `Person` und `Printable` haben nur `Object` als Supertypen.
- Der Typ `Angestellter` hat `Person` und `Printable` als direkte Supertypen.
- Eine Schnittstellendeklaration erweitert also die Schnittstelle eines oder mehrerer anderer Typen.
- Methodensignaturen aus den Supertypen brauchen nicht nochmals aufgeführt werden (*Signaturvererbung*).

Dynamische Methodenauswahl

Die Auswertung von Ausdrücken vom (statischen) Typ T kann Ergebnisse haben, die von einem Subtyp sind.

```
University u = ....;

Course c = u.findCourse("Graphtheorie");
c.printStudents();
```

- Ist "Graphtheorie" eine Vorlesung oder ein Seminar?
- Wie ist der Methodenaufruf `printStudents()` auszuwerten?

Begriffsklärung: Dynamische Methodenauswahl

Die auszuführende Methode zu einem Methodenaufruf:

```
<Ausdruck>.<methodenName>( <AktParam1>,...);
```

wird wie folgt bestimmt:

1. Werte zunächst den Ausdruck aus; Ergebnis ist das *Zielobjekt*.
2. Werte dann die aktuellen Parameter aus.
3. Führe die Methode mit dem entsprechenden Methodennamen des Zielobjekts mit den aktuellen Parametern aus.

Weitere Aspekte der Subtypbildung

Zyklenfreiheit

Die Subtyprelation darf keine Zyklen enthalten.

Folgendes Fragment liefert einen Fehler beim Compilieren:

```
interface C extends A { ... }
interface B extends C { ... }
interface A extends B { ... }
```

Subtyprelation bei Arrays I

Jeder Arraytyp mit Komponenten vom Typ S ist ein Subtyp von `Object`:

$$\texttt{S[]} \leq \texttt{Object}$$

D.h. erlaubt ist

```
Object ov = new String[3];
```

Ist $S \leq T$ für Referenztypen S und T, dann ist `S[]` $\leq$ `T[]`.

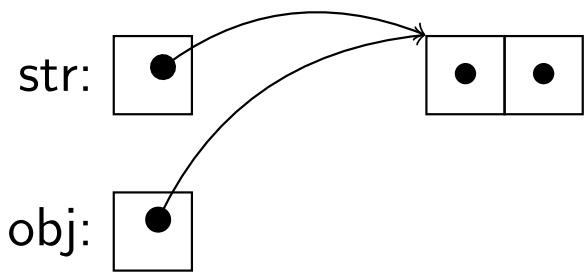
```
Person[] pv = new Student[3];
```

Subtyprelation bei Arrays II

Statische Typsicherheit ist nicht mehr gegeben:

```
1    String[] str = new String[2];
2    Object[] obj = str;
3    obj[0] = new Object(); // Laufzeitfehler: ArrayStoreException
4    int strl   =  str[0].length();
```

Nach Ausführen von Zeile 2 liegt folgender Speicherzustand vor:



Polymorphie

- Polymorphie ist die Eigenschaft, verschiedene Formen annehmen zu können.
- Die Form der Polymorphie in Typsystemen mit Subtypen heißt **Subtyp-Polymorphie**. Dabei gehören Objekte eines Subtyps auch den entsprechenden Supertypen an.
- Subtyp-Polymorphie erlaubt es z.B. *inhomogene* Arrays oder Listen zu verwenden, d.h. mit Elementen unterschiedlichen Typs (s.o.).