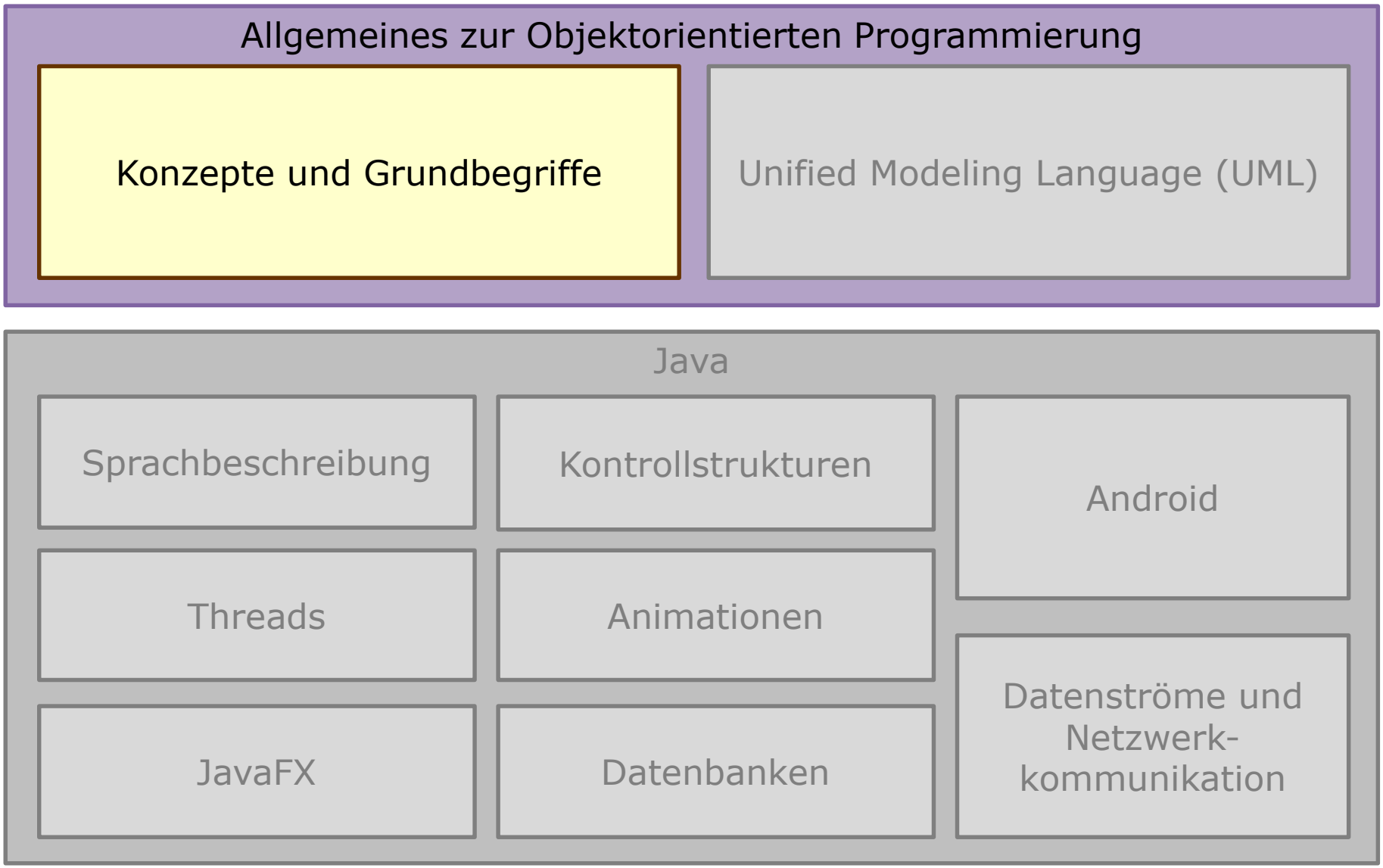
BERUFSAKADEMIE SACHSEN
STAATLICHE STUDIENAKADEMIE
DRESDEN
UNIVERSITY OF COOPERATIVE EDUCATION

Objektorientierte Programmierung

Konzepte und Grundbegriffe

Prof. Dr.-Ing. Tenshi Hara
tenshi.hara@ba-sachsen.de

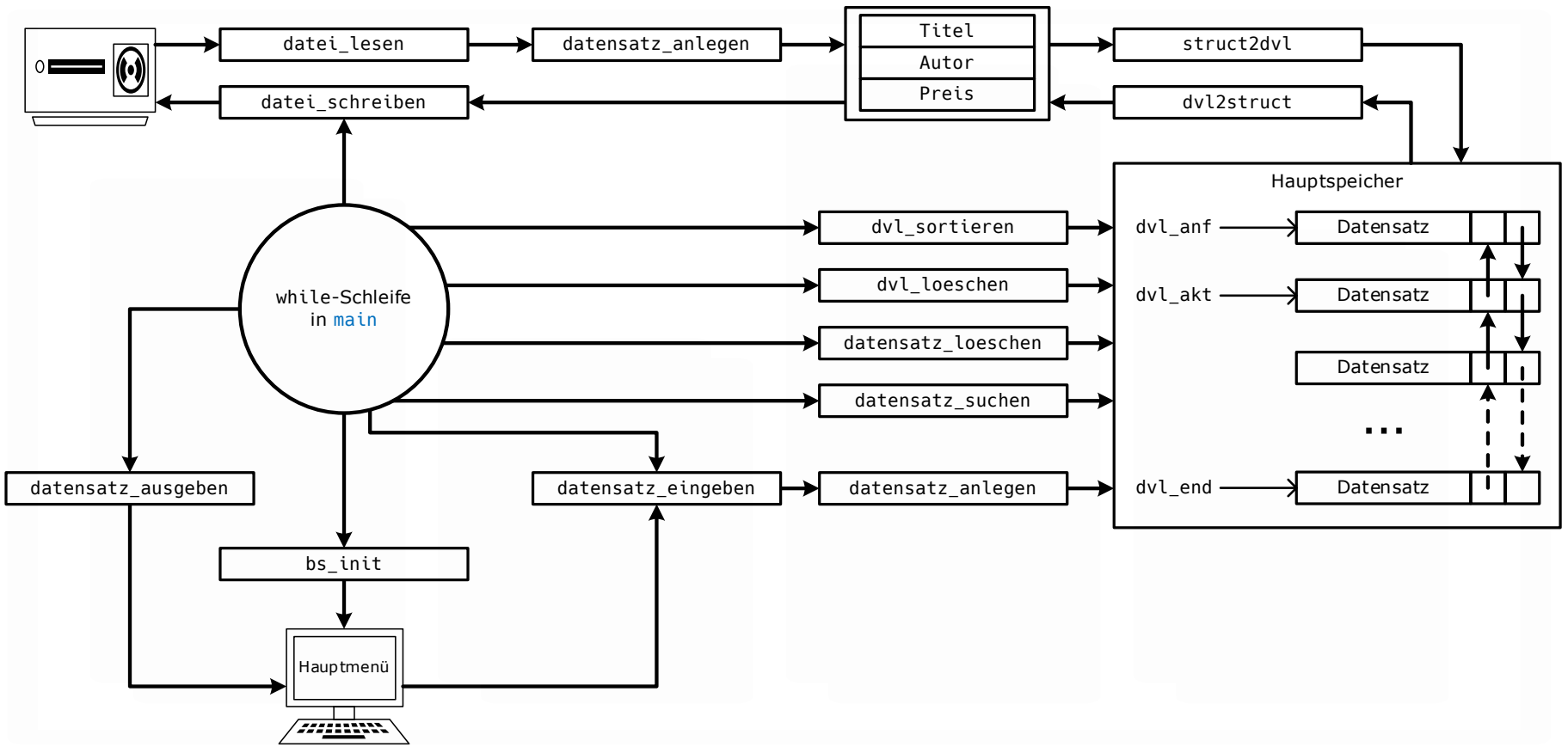
Aufbau der Lehrveranstaltung



Kritik an bisheriger Herangehensweise

- Problemlösungsbereich wird mit unterschiedlichen Konzepten beschrieben (semantische Lücke)
- wenig Unterstützung bei inkrementellen Erweiterungen (Variantenbildung)
- wenig Unterstützung für systematische Wiederverwendung
- Trennung der Datenstrukturen und Verarbeitungsstrukturen

PROZEDURALE HERANGEHENSWEISE



Was ist objektorientierte Programmierung?

- Konzept zur Entwicklung von Programmen
- Verlagerung des Schwerpunkts von den Algorithmen auf die Daten
- wichtige Prinzipien
  - Datenabstraktion
  - Modularisierung
  - Klassen und Objekte
  - Hierarchisierung (Klassenhierarchie)
  - Kapselung (Information Hiding)
  - Funktionsüberladung
  - (Mehrfach-)Vererbung und Ableitung
  - Polymorphismus

Grundidee der OOP

- Verknüpfung von Objekten mit eigener Aktionsfähigkeit
- sichere Programme durch Festlegungen, wer auf Daten zugreifen kann
- Namen von Unterprogrammen können mehrfach verwendet werden
- Wiederverwendbarkeit durch Klassen und Vererbung
- intelligente Objekte reagieren auf definierte Botschaften

Prinzipien der OOP

Definition von Objekt-Orientierung nach B. Meyer

1. objektbasierte modulare Struktur
2. Datenabstraktion
3. automatische Speicherverwaltung
4. Klassen
5. Vererbung
6. Polymorphismus und dynamisches Binden
7. multiple und wiederholte Vererbung

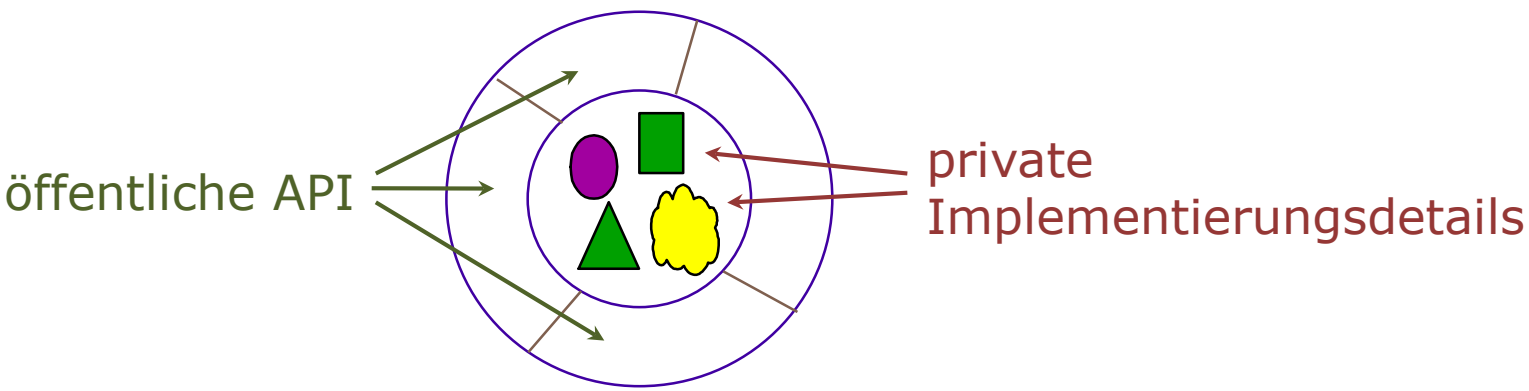
Objekte und Klassen

- Klasse (Class)
  - allgemeine Beschreibung eines oder mehrerer Objekte mit übereinstimmenden Attributen (Eigenschaften) und Methoden (Funktionalität)
  - Muster für Objekte (Bauplan für Objekte gleicher Art)
  - Konstruktor zur Definition eines Objekts
  - enthält Deklarationen der Attribute und Methoden
- Objekt (Object)
  - Abbildung eines konkreten Gebildes der Realität
  - Instanz einer Klasse, d.h. ein konkretes Exemplar einer Klasse
  - Kapselung von Daten (Attributen) und Methoden

Objekte und Klassen

Was ist ein Objekt?

- Bündel aus Variablen und dazugehörigen Methoden
- verwendet um alltägliche Objekte der realen Welt in Modell abzubilden



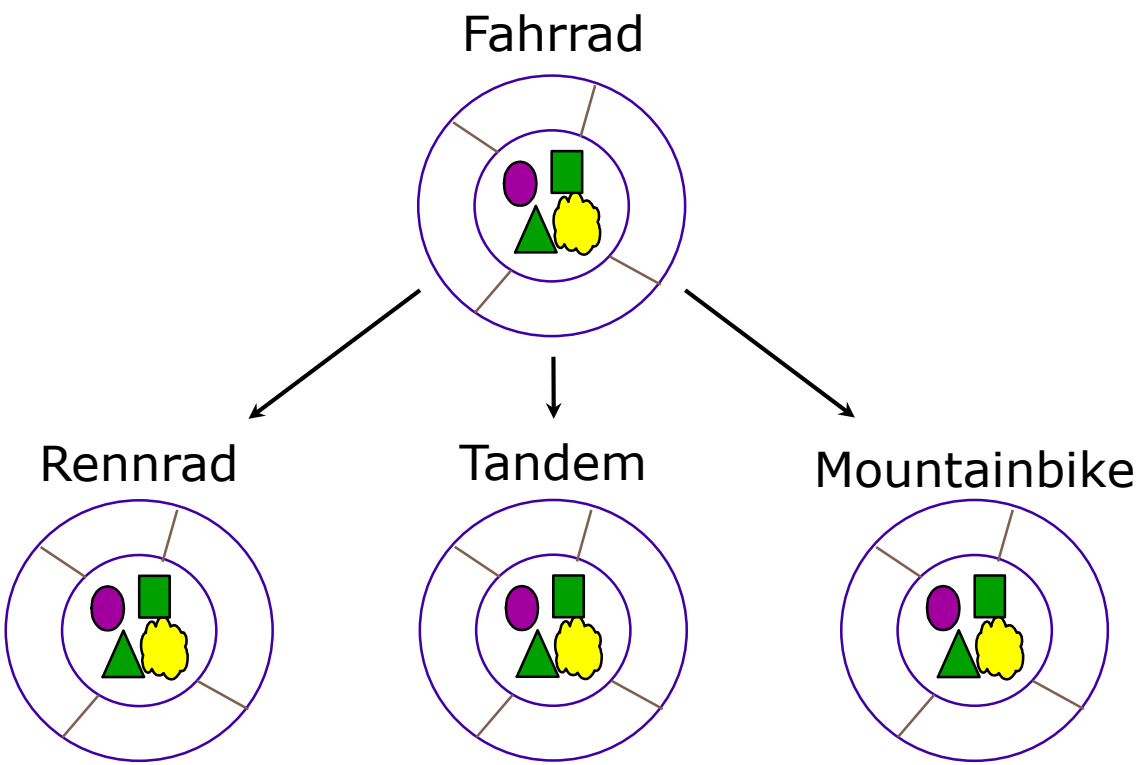
Vererbung und Ableitung

- Vererbung: Nutzung von Eigenschaften bereits vorhandener Klassen (Basisklassen) durch Ableitung
  - Eigenschaften können verändert und hinzugefügt werden
  - Ziel: wiederverwendbarer Programmcode
- Einfachvererbung: eine Klasse kann nur von einer Klasse erben (Beispiel: Mensch erbt von Primat; so arbeitet Java)
- Mehrfachvererbung: eine Klasse kann von mehreren Klassen erben (Beispiel: Panzer erbt von Waffe und Fahrzeug; geht in C++)

Vererbung und Ableitung

Was ist eine Vererbung?

- Status und das Verhalten von übergeordneter Klasse (Super-Class)
- mächtiger und natürlicher Weg, Software zu organisieren



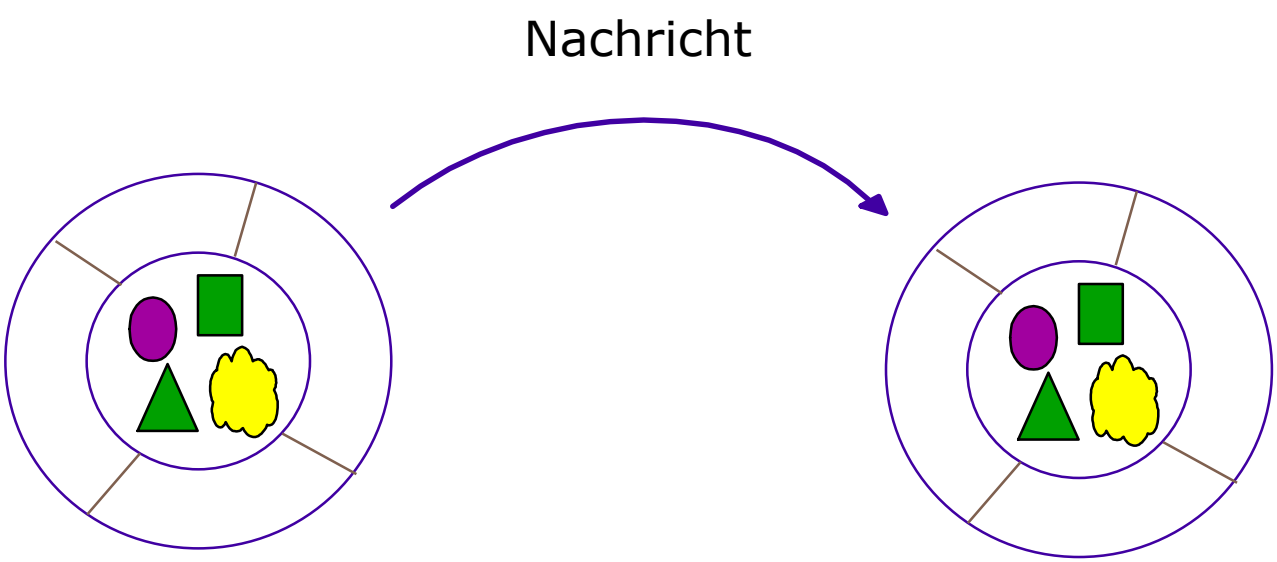
Prinzip der Kapselung

- Datendarstellung und Funktionsablauf hinter Schnittstellen versteckt
- Daten eines Objekts sollen nur durch die zugehörigen Methoden des Objekts manipuliert werden können, d.h. kein direkter Zugriff auf die Daten eines Objekts durch das Anwendungsprogramm
- wird erreicht durch die Möglichkeit, die Methoden eines Objekts „sichtbar", die Daten jedoch „unsichtbar" zu deklarieren

NACHRICHTEN

Was ist eine Nachricht? (Message)

- Objekte interagieren oder kommunizieren miteinander via Nachrichten
- Aufruf einer Methode eines Objektes ist auch eine Nachricht an das Objekt (Aufforderung)



OOP in Java – Klassen in Java

- abstrakte Beschreibung von Objekten
- es existieren keine Variablen und Methoden außerhalb von Klassen
- Superklasse aller anderen Klassen ist direkt oder indirekt `Object`
  - wenn keine Ableitung angegeben ist, wird von `Object` abgeleitet
  - Methode `clone()` erzeugt Kopie des Objekts
  - Methode `toString()` liefert String-Repräsentation
  - Methode `equals(Object o)` prüft auf dieselbe Referenz, wenn nicht anders überschrieben
  - Instanzen von `Object` können Instanzen anderer Klassen zugewiesen werden

Klassen in Java

- Java-Programme (Application oder Applet) bestehen aus mindestens einer `public`-Klasse die in eine Datei mit exakt demselben Namen wie die Klasse und der Dateierweiterung `.java` geschrieben wird
  - neben der `public`-Klasse können weitere Klassen in dieselbe Datei oder in andere Dateien geschrieben werden
  - maximal eine Klasse pro Datei darf `public` sein
- eine Klasse kann weitere „innere" Klassen enthalten
  - Beispiel des Namen einer entsprechenden `class`-Datei: `MeinProgramm$InnereKlasse.class`
  - Objekte der inneren Klasse können nur in der äußeren Klasse erzeugt werden (außer bei statischen inneren Klassen)
  - aus inneren Klassen kann auf Daten und Methoden der äußeren Klasse zugegriffen werden (auch auf private)

Anonyme Klassen in Java

- eine besondere Form von inneren Klassen sind anonyme Klassen
- die Klassendefinition wird in einem Schritt mit der Objekterzeugung verbunden

```
addWindowListener(new WindowAdapter() {
   public void windowClosing(WindowEvent we) {
      setVisible(false);
      dispose();
      System.exit(0);
   }
});
```

KAPSELUNG

Modifizierer	Klasse	abgeleitete Klasse	Paket	öffentlich
private	✓	X	X	X
package (Standard)	✓	X	✓	X
protected	✓	✓	✓	X
public	✓	✓	✓	✓

Interfaces

- in Java gibt es keine Mehrfachvererbung, dafür gibt es aber Interfaces (Schnittstellen)
  - Klassen können mehrere Interfaces implementieren
  - Interfaces dürfen nur Konstanten und Methoden-Deklarationen enthalten
  - Klassen, die Interfaces implementieren, müssen alle im Interface enthaltenen Methoden implementieren
  - alle Methoden eines Interface sind automatisch `abstract` und `public` (sie können weder `private` noch `protected` sein)
- Interfaces enthalten keine Konstruktoren und Destruktoren
- Interfaces ohne Methoden und Attribute sind Flag-Interfaces (z.B. das Interface `Serializable`)

Adapter-Klassen

Adapter-Klassen erweitern Interfaces

- können neben den vom Interface implementierten Methoden weitere Methoden besitzen
- werden häufig bei Ereignisbehandlung eingesetzt
- bei Ableitung von Adapter-Klassen müssen nur die benötigten Methoden überschrieben werden
- Beispiel für eine Adapterklasse

```
public class xxxAdapter implements xxxInterface {
   public void methode1() {}
   public void methode2() {}
   public void methode3() {}
   // …
}
```

Objekte

- elementare Datentypen sind keine Objekte
  - zu den elementaren Datentypen gibt es korrespondierende Wrapper-Klassen, um Objekte erzeugen zu können
- Objekte werden i.d.R. mit new erzeugt
  - Deklaration (Anlegen der Referenz) und Erzeugung des Objekts (Speicherbelegung) können getrennt erfolgen
  - ein Objekt existiert, solange eine Referenz auf das Objekt existiert; Freigabe erfolgt (i.d.R. automatisch) durch den Garbage Collector
  - `String`-Objekte können auch ohne `new` erzeugt werden (Beispiel: `String Text1 = "Hallo";`)
  - es können anonyme Objekte erzeugt werden, bspw.

```
g.drawPolygon(
   new int[] {50,100,150}, new int[] {150,50,150}, 3
);
```

Methoden

- es kann keine Methoden außerhalb von Klassen geben
- in Applikationen muss mindestens eine Methode `main` vorhanden sein
- `main` ist `static`, deshalb muss von der enthaltenden Klasse kein Objekt gebildet werden

```
public class AutoFahren {
   public static void main(String[] Argumente) {
      System.out.println("Hauptprogramm");
   }
}
```

„STATIC“ UND „FINAL“

- statische Methoden und Variablen (Klassenvariablen) nur 1x vorhanden
- statische Methoden arbeiten nur mit statischen Variablen und Methoden
- es muss kein Objekt gebildet werden, um auf statische Methoden und Variablen zuzugreifen
- finale Variablen dürfen nicht verändert werden, sind also Konstanten
- finale Methoden dürfen nicht überschrieben werden
- von finalen Klassen können keine weiteren Klassen abgeleitet werden (Beispiele: `java.lang.String` und `java.lang.Math`)

KONSTRUKTOREN

- erstellen eine Objektinstanz der Klasse
- haben den Namen der Klasse und sind typenlos (ohne Rückkehrwert)
- werden nicht wie andere Methoden aufgerufen, sondern automatisch beim Bilden einer Instanz aufgerufen (deshalb sind sie `public`)
- Java stellt immer einen parameterlosen Standard-Konstruktor bereit

```
public class Auto {
   public Auto() {
      System.out.println("Standard-Konstruktor");
   }
}
```

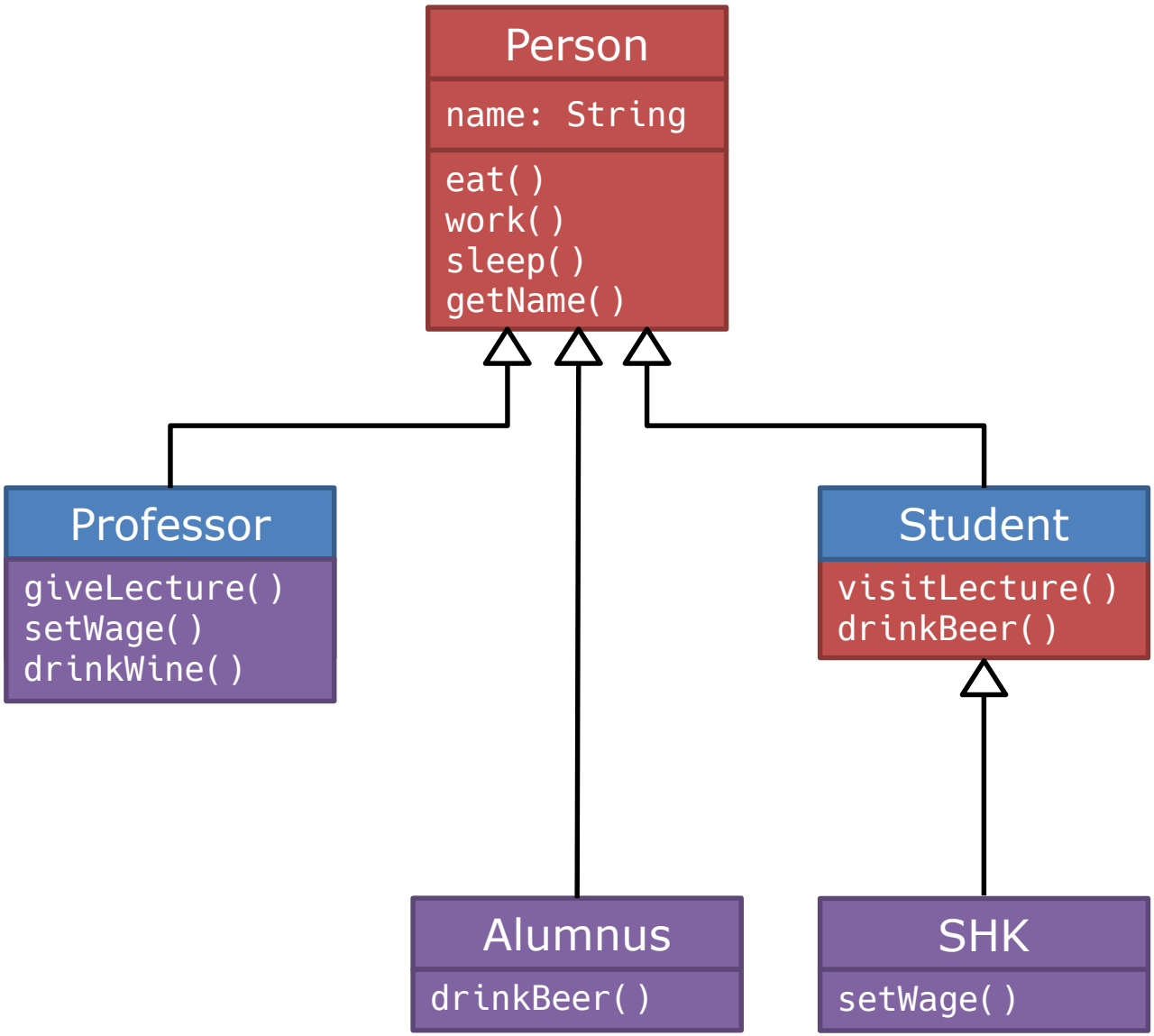
Destruktoren

- in Java gibt es keine Destruktoren
- in jeder Klasse kann eine Methode `finalize` definiert werden, die aufgerufen wird, wenn der Garbage-Collector das Objekt beseitigt
  - keine Garantie, ob und wann `finalize` aufgerufen wird
  - `finalize` wird nicht direkt aufgerufen
    (Java hindert Sie aber nicht am manuellen Aufruf;
    der Garbage-Collector ruft sie dann noch einmal auf)
  - keine garantierte Aufruf-Reihenfolge bei mehreren Objekten

```
public void finalize() {
   // …
}
```

Vererbung und Polymorphismus

Motivation: Vererbung



Warum Vererbung?

- Kodeverschmutzung durch Copy-and-Paste-Programmierung (CAPP)
  - Programm wächst über das Maß der Funktionserweiterung hinaus
  - Korrekturen und Änderungen nicht in allen Replikaten angewendet
  - undurchsichtige Dokumentation
- zunehmende Testkomplexität durch CAPP
  - ähnliche/gleiche Tests in verschiedenen Kontexten wiederholen
  - unvorhergesehene Seiteneffekte

Vererbung und Polymorphismus

elementare Techniken der Wiederverwendung

- Generalisierung und Spezialisierung mit einfacher Vererbung zwischen Klassen, konzeptuell und im Speicher
  - abstrakte Klassen und Schnittstellen
  - Merkmalssuche in einer Klasse und in der Vererbungshierarchie aufwärts
  - Überschreiben von Merkmalen
  - generische Typen zur Vermeidung von Fehlern (Nachbartypschranken)
- Polymorphie als dynamische Architektur
  - Lebenszyklen von Objekten
  - Änderungen im Speicher

Liskow'sches Ersetzbarkeitsprinzip

Ein Programm, das mit einem Objekt einer Klasse arbeiten kann, kann fehlerfrei mit jedem Objekt einer ihrer Unterklassen arbeiten.

Egal, welches Objekt einer Klasse aus einer Klassenhierarchie für die Abarbeitung eines Aufrufs genommen wird, der Aufruf muss immer funktionieren und darf nicht zum Absturz des Aufrufers führen!

Ableitung einer Klasse – Vererbung

- Quellkode der neuen Klasse kann geschrieben werden
  - ans Ende der Datei der Superklasse
  - in neue Datei mit entsprechender `public`-Klasse
- alle Attribute und Methoden der Superklasse mit Ausnahme der Konstruktoren werden vererbt
- `private`-Attribute der Superklasse werden vererbt (keine Zugriffsverletzung)

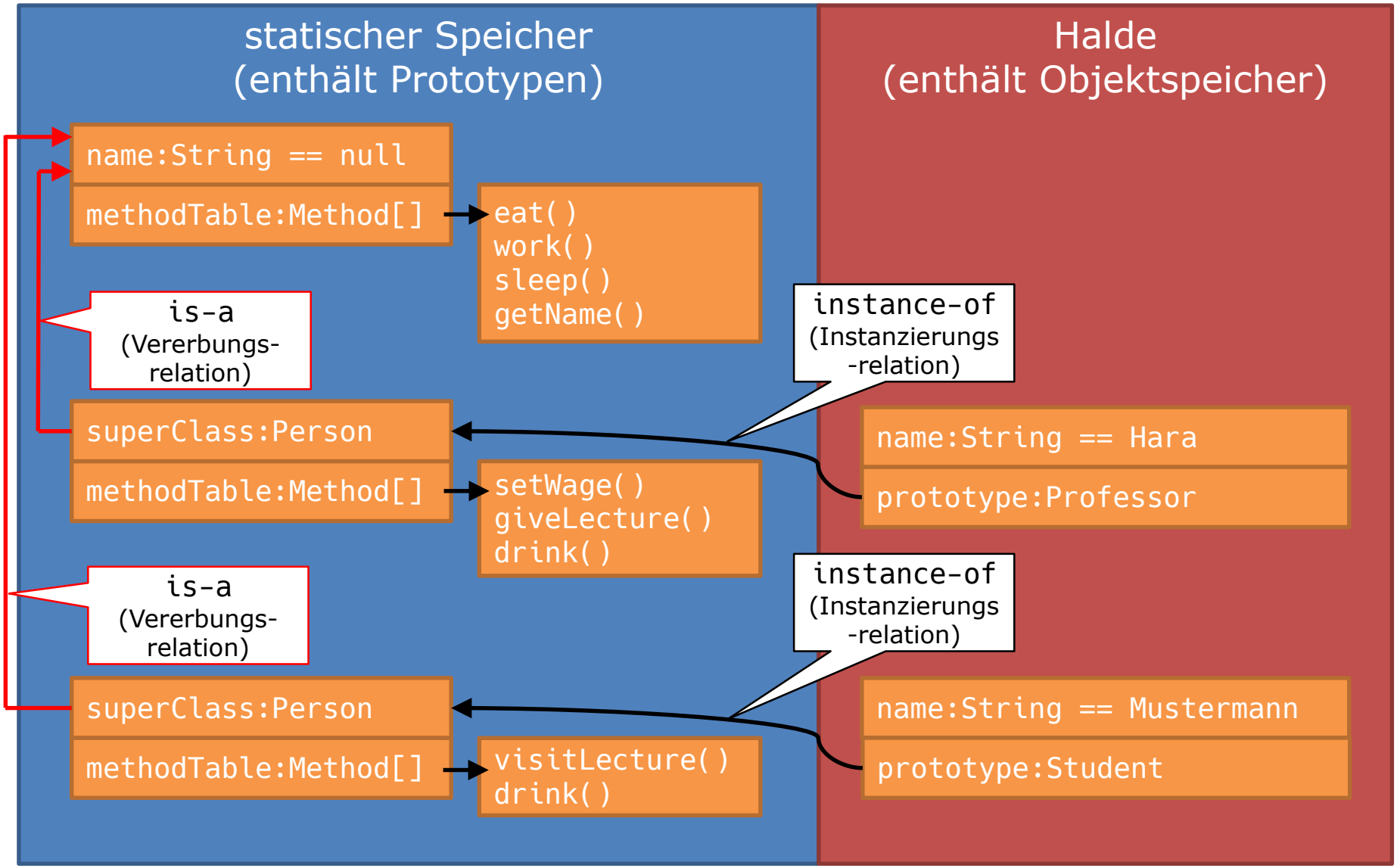
```
class Spezialisierung extends Generalisierung {
   // …
}
```

Schlüsselwort „super“

- `super` steht für das Objekt oder einen Konstruktor der Superklasse
- im Konstruktor muss `super` als erste Anweisung stehen, wenn es verwendet wird!
- nach `super` können weitere Anweisungen zur Spezifizierung dieser Konstruktoren stehen

```
// …
private int anzahl = 0;
// …
public Spezialisierung(int vor, int nach) {
   super(nach);          // Eigenschaft wird an Konstruktor
                         // der Superklasse übergeben
   this.anzahl = vor;    // Eigenschaft bleibt in der
                         // Spezialisierung verborgen
   // …
}
// …
```

Vererbung im Speicher



Methoden- und Merkmalsuche im Vererbungsbaum

- Generalisierung: Oberklassen sind allgemeiner als Unterklassen
- Spezialisierung: Unterklassen sind spezieller als Oberklassen
- Resolution: Methoden- und Merkmalsuche: Wird eine Methode oder ein Merkmal nicht in einer Klasse definiert (im Speicher gefunden), wird in der Oberklasse gesucht
- Beispiel: Suche Methode `sleep()` für `Hara:Professor`
  1. Suche in Prototypklasse der aktuellen Objektinstanz
  2. Methode wird nicht gefunden
  3. Suche in Superklasse der Prototypklasse `Professor`
  4. Methode wird gefunden und aufgerufen

Schlüsselwort „this“

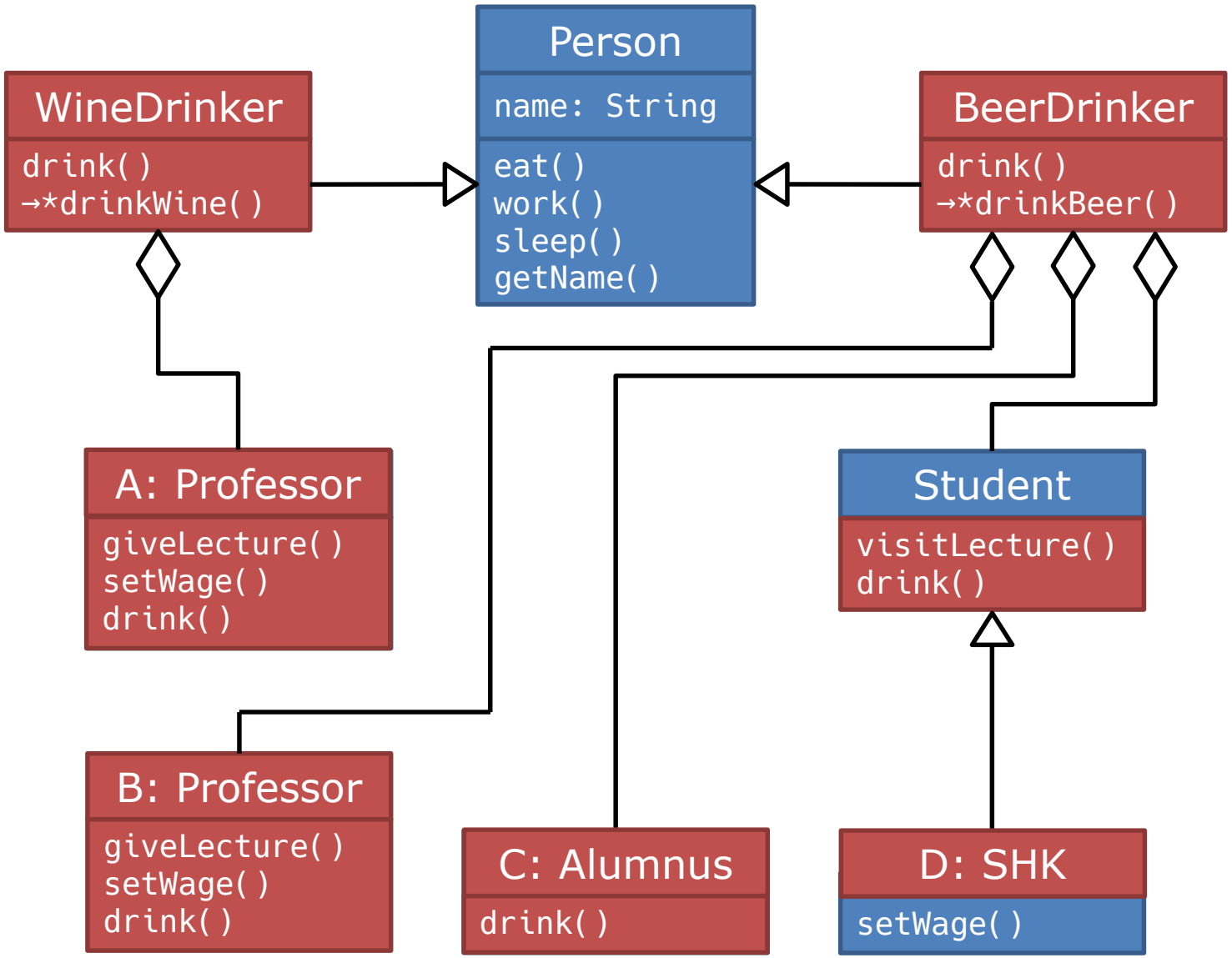
- in jedem Objekt gibt es die Referenzvariable `this`, welche automatisch an alle Methoden des Objektes übergeben wird
- Anwendungen
  - Zugriff auf die Instanz bei gleichnamigen Variablen
  - eigenes Objekt als Rückgabewert einer Methode
  - Übergabe als Parameter
  - Verkettung von Konstruktoren

```
public Test() { // Konstruktoren haben keinen Typ!
   this(i);     // Aufruf des Konstruktors dieser
                // Klasse mit einem Parameter
}
```

FUNKTIONSÜBERLADUNG

- mehrere Funktionen können in einer Klasse denselben Namen haben
- Parameterliste ist entscheidend für die richtige Auswahl der Funktion (Anzahl und Typ der Parameter)
- sinnvoller Einsatz, wenn gleiche Aufgaben für verschiedene Datenformen durchgeführt werden sollen
- in C++ können auch Operatoren (+, *, -, /) überladen werden, wenn z.B. rationale Zahlen oder komplexe Zahlen definiert werden
- in Java ist das Überladen von Operatoren nicht möglich!
  (Beispiel: +, -, *, / für rationale Zahlen)

Motivation: Polymorphismus



Polymorphismus

- gleiche Nachricht kann zu Objekten verschiedener Klassen geschickt werden
- eine Methode wird polymorph genannt, wenn sie für mehrere verschiedene Objekttypen verwendet werden kann
- Methodendefinition der aufgerufenen Methode können sich befinden
  - in der Klasse des Objektes
  - in der Superklasse (Methode wurde vererbt)
  - als Override in der Objektklasse, welcher die Methodendefinition der Superklasse überschreibt
  - als Implementierung in der Objektklasse, welche einen Prototypen der Superklasse umsetzt

Schlüsselwort „super“

- Das Schlüsselwort `super` steht für Objekt oder Konstruktor der Basisklasse (Superklasse)
- im Konstruktor muss `super` als erste Anweisung stehen, wenn es verwendet wird!
- nach `super` können weitere Anweisungen zur Spezifizierung dieser Konstruktoren stehen

```
// …
A_Klasse auto5 = new A_Klasse(); // ohne Parameter
A_Klasse auto6 = new A_Klasse(200);
A_Klasse auto7 = new A_Klasse(120, "Auto7");
// …
```

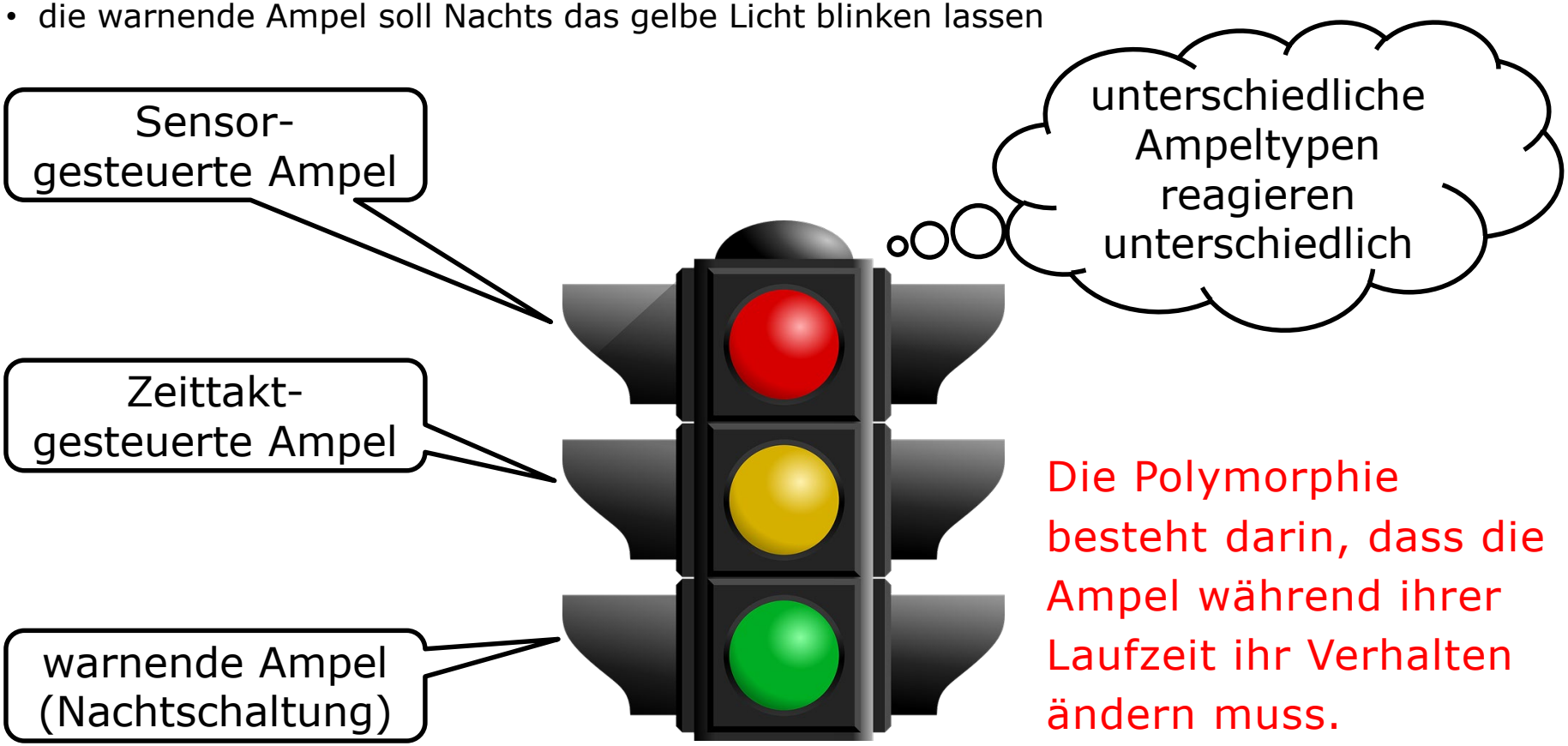
Überladen von Methoden (ad-hoc-Polymorphismus)

- Methoden können nicht nur anhand des Typs unterschieden werden
- Methoden gleichen Namens in derselben Klasse unterscheiden sich durch Typ und Anzahl der Parameter
- Überladen von Methoden wird meist für Methoden mit gleicher Funktionalität aber verschiedenem Datenraum angewendet
- Konstruktoren können und sollten überladen werden! (wichtig für Polymorphie)

Beispiel: Polymorphe Ampel

jede Ampel schaltet auf eine spezifische Art und Weise:

- die Sensor-gesteuerte Ampel schreibt vor, dass auf ein Sensorereignis mit einem Schaltvorgang reagiert werden muss
- die Zeittakt-gesteuerte Ampel schreibt vor, dass auf ein Zeitsignal geschaltet werden muss
- die warnende Ampel soll Nachts das gelbe Licht blinken lassen



Die Polymorphie besteht darin, dass die Ampel während ihrer Laufzeit ihr Verhalten ändern muss.

POLYMORPHE OBJEKTREVOLUTION

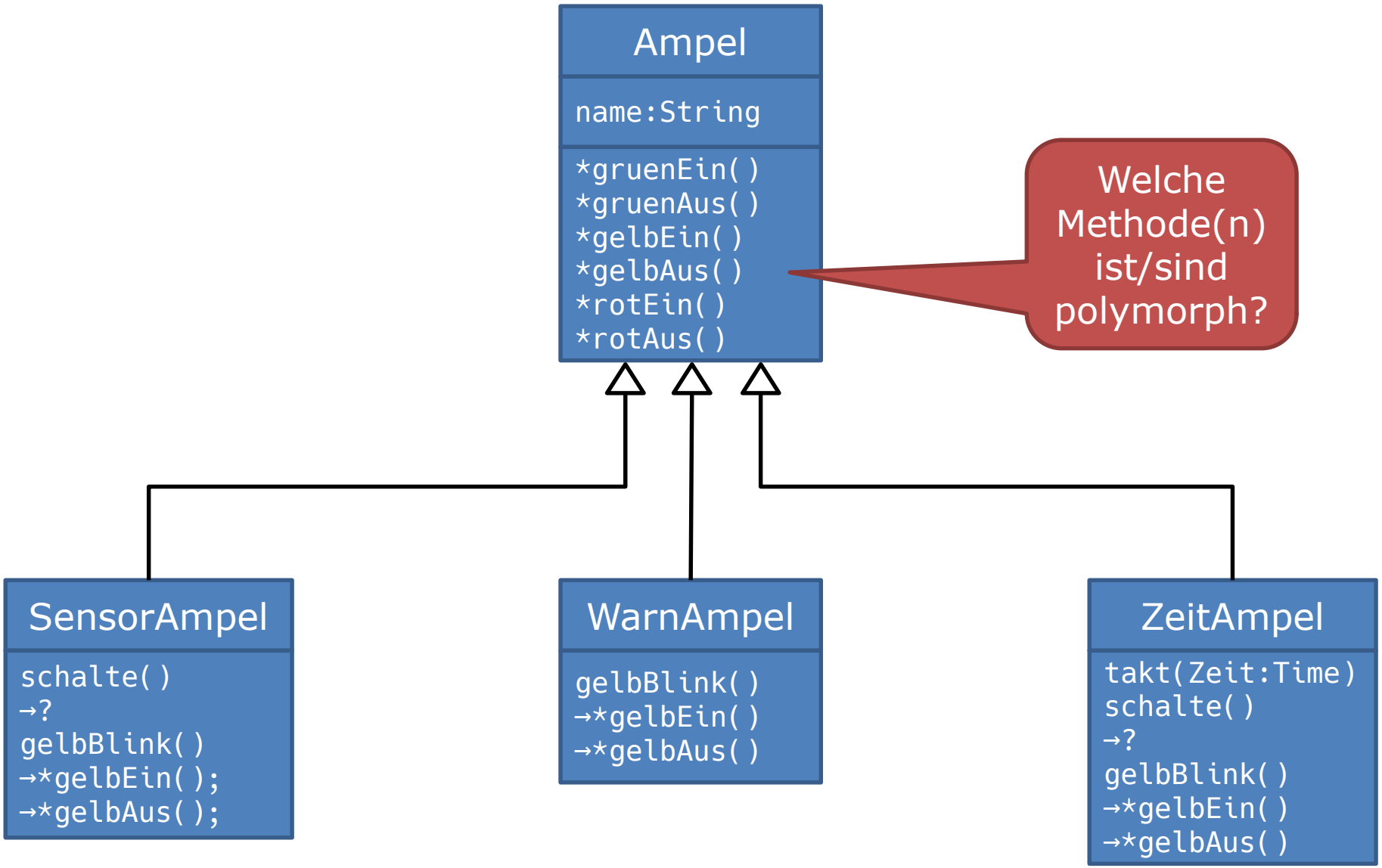
Dispatch

```
// …
Ampel gerock1;
if (Zeit>=Abend && Zeit<=Morgen)
   gerock1 = new WarnAmpel();
else if (Zeit>Morgen && Zeit<Vormittag)
   gerock1 = new SensorAmpel();
else if (Zeit>=Vormittag && Zeit<=Nachmittag)
   gerock1 = new ZeitAmpel();
else if (Zeit>Nachmittag && Zeit<Abend)
   gerock1 = new SensorAmpel();
else
   // unerlaubter Zeit-Zustand
   gerock1 = new WarnAmpel();
// …
```

Polymorphe und monomorphe Methoden

- Methoden, die nicht mit einer Generalisierung geteilt werden, können nicht polymorph sein
- Adressen monomorpher Methoden können im Speicher statisch sein
  → keine Merkmalsuche notwendig
- Adressen polymorpher Methoden müssen dynamisch ermittelt werden
  → Merkmalsuche immer notwendig

Kurzübung: polymorphe Ampel



Java am Beispiel: Autofahren

Java am Beispiel – Autofahren

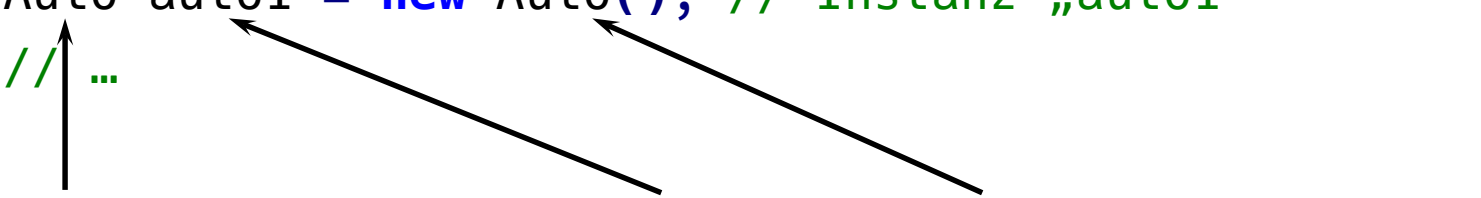
```
import java.io.*;

public class AutoFahren {
   public static void main(String Argumente[]) {
      System.out.println("Programm AutoFahren");
      /*
       * Hier wird später mit den Instanzen der
       * Klasse „Auto“ und „A_Klasse“ gearbeitet
       */
   }
}
```

Klasse „Auto“ und Instanzbildung

```
public class Auto { // in Datei „Auto.java“
   int Ps = 0;      // Attribute der Klasse
   String Name = "";
}
```

Irgendwo wird eine Instanz gebildet:

```
// …
System.out.println("Programm AutoFahren");
Auto auto1 = new Auto(); // Instanz „auto1“
// …
```



Konstruktoren – Standard-Konstruktor

```
public class Auto {
   int Ps = 0;      // Attribute der Klasse
   String Name = "";

   public Auto() { // Standard-Konstruktor
      Ps = 75; Name = "Auto";
   }
}
```

Irgendwo wird eine Instanz gebildet:

```
// …
Auto auto1 = new Auto(); // Instanzbildung
System.out.println("PS: " + auto1.Ps);
// …
```

Anwendung weiterer Konstruktoren

```
// …
Auto auto1 = new Auto();  // Instanzbildung
System.out.println("Ps= " + auto1.Ps
                   + "Name= " + auto1.Name);

// Was getan werden muss, damit folgende Zeilen funktionieren,
// folgt auf der nächsten Folie

Auto auto2 = new Auto(100); // mit 100 PS
System.out.println("Ps= " + auto2.Ps
                   + "Name= " + auto2.Name);

Auto auto3= new Auto(9, "2CV"); // Ente mit 9 PS
System.out.println("Ps= " + auto3.Ps
                   + "Name= " + auto3.Name);
// …
```

WEITERE KONSTRUKTOREN

```
public class Auto {
   // Attribute der Klasse
   // ...

   // Standard-Konstruktor
   // ...

   // 2. Konstruktor
   public Auto(int ps) {
      Ps = ps;
      Name = "Auto";
   }

   // 3. Konstruktor
   public Auto(int ps, String name) {
      Ps = ps;
      Name = name;
   }
}
```

WEITERE METHODEN

```
public class Auto {
   // Attribute der Klasse
   // ...

   // Konstruktoren
   // ...

   public void fahren() { // eigene Methode
      System.out.println("Auto fährt");
   }
}
```

Anderswo im Programm:

```
// …
auto1.fahren(); // Nachricht an Objekt „auto1“
                // zur Ausführung der Methode
// …
```

Datenkapselung

```
public class Auto {
    // Attribute der Klasse
    private int Ps = 0;
    String Name = "";
    // …
}
```

- mit dem Schlüsselwort `private` ist auf dieses Attribut nicht mehr außerhalb dieser Klasse zugreifbar
- die Klasse `Auto` lässt sich separat ohne Syntax-Fehler kompilieren
- das Kompilieren der Klasse `AutoFahren` bringt Fehler-Meldung: ... **`"Ps has private access in Auto"`**

Getter und Setter

```
public class Auto {
   // Attribute der Klasse
   private int Ps = 0;
   String Name = "";
   // …

   public int getPs() { // nur mit dieser Methode kann auf das
      return Ps;        // Attribut außerhalb der Klasse zuge-
   }                    // griffen werden

   public void setPs(int ps) {  // nur mit dieser Methode kann
      Ps = ps;                  // das Attribut außerhalb der
   }                            // Klasse geändert werden
}
```

Anwendung der Getter und Setter

```
// ...
Auto auto2 = new Auto(100); // mit 100 Ps
System.out.println("PS: " + auto2.getPs()
                   + "Name: " + auto2.Name);
auto2.setPs(150);     // Tuning auf 150 Ps
System.out.println("PS: " + auto2.getPs()
                   + "Name= " + auto2.Name);
// ...
```

Ableitung einer Klasse – Vererbung

- Quellcode der neuen Klasse kann auch in die Datei `Auto.java` geschrieben werden (ans Ende der Datei)
- neue Datei `A_Klasse.java` mit einer entsprechenden `public`-Klasse ist auch möglich
- alle Attribute und Methoden der Klasse `Auto` mit Ausnahme der Konstruktoren werden vererbt
- auch die `private`-Attribute werden vererbt (keine Zugriffsverletzung!)

```
class A_Klasse extends Auto {
   // …
}
```

Anwendung der neuen Klasse

```
// …
A_Klasse auto4 = new A_Klasse();
System.out.println("PS: " + auto4.getPs()
                   + "\nName: " + auto4.Name);
auto4.fahren();
// …
```

- Beispiel zeigt, dass die Konstruktoren (außer dem Standard-Konstruktor) nicht vererbt werden
- alles andere funktioniert

Weitere Methode „fahren“

```
// Methode der Klasse „Auto“
public void fahren(boolean elch) {
   if (elch) {
      System.out.println("Achtung Elch!");
      System.out.println("Auto stabilisiert");
   }

   System.out.println("Auto fährt");
}
```

- Aufruf der Methode `fahren` mit Parameter sieht für beide Klassen völlig gleich aus

Methode überschreiben

```
// Methode der Klasse „A_Klasse“
public void fahren(boolean elch) {
   if (elch) {
      System.out.println("Achtung Elch!");
      System.out.println("Auto kippt");
   }
   else
      System.out.println("Auto fährt");
}
```

Polymorphie (Vielgestaltigkeit)

```
// …

// Objekt der Klasse „Auto“
auto3.fahren();      // ohne Parameter
auto3.fahren(true);  // mit Parameter
auto3.fahren(false);

// Objekt der Klasse „A_Klasse“
auto4.fahren();      // ohne Parameter
auto4.fahren(true);  // mit Parameter
auto4.fahren(false);

// …
```

Konstruktoren der Klasse A_Klasse

```
public class A_Klasse {
   // … -- Attribute der Klasse

   public A_Klasse() {
      // Standard-Konstruktor
      super();
   }

   public A_Klasse(int ps) {
      // 2. Konstruktor
      super(ps);
   }

   public A_Klasse(int ps,String name) {
      // 3. Konstruktor
      super(ps, name);
   }
}
```

LITERATURANGABEN

G. Krüger: Handbuch der Java-Programmierung; Addison-Wesley Verlag, ISBN 3-8273-2201-4

E. Niedermair, M. Niedermair: Internet-Programmierung mit Java; Data Becker GmbH & Co. KG, ISBN 3-8158-2086-3

A. Niemann: Das Einsteigerseminar, Objektorientierte Programmierung in Java; bhv Verlag Bürohandels- und Verlagsgesellschaft mbH, ISBN 3-8287-1015-8

Java 2 SDK v 1.2.2, Grundlagen Programmierung; HERDT-Verlag für Bildungsmedien GmbH, Nackenheim