Informatik 2
Einführung in Java und UML

Prof. Dr.-Ing. Holger Vogelsang
holger.vogelsang@hs-karlsruhe.de

Inhaltsverzeichnis

Inhaltsverzeichnis

Inhaltsverzeichnis

- Organisation:
  - Christian Meder: Android, Datenstrukturen
  - Holger Vogelsang: Einführung Java, UML
  - → natürlich eine gemeinsame Prüfung

- Gemeinsame Übungen für alle Themen in Gruppen, Anmeldung dazu im Ilias
- Programmiertutorium für Einsteiger ohne Anmeldung
- Tutorium ohne Anmeldung

Roter Faden

Java und UML
Schritte, Software
Objekt-orientierung
Klassendiagramme
Ausnahmen
Funktionale Programmierung
Weitere Beispiele
Klassen
Überladen
Vererbung
Gener. Klassen
Aufzähl-typen
Regeln
Über-schreiben
Typinfo., I/O
Annota-tionen
Laufzeit-typinfo.
Ein-, Ausgabe
Entwurf
Prinzipien
Verbindung von Modulen
Spring

Java

- Christian Ullenboom:
  - „Java ist auch eine Insel“, Rheinwerk Verlag GmbH (auch frei als „Open-Book“ unter http://openbook.rheinwerk-verlag.de/javainsel/)
  - „Java 8 – Mehr als eine Insel“, Rheinwerk Verlag GmbH
- D. Ratz, J. Scheffler, D. Seese, J. Wiesenberger: „Grundkurs Programmieren in Java“, Hanser-Verlag
- R. C. Martin: „Clean Code“, mitp
- Cay S. Hortsmann, „Java 8 SE for the Really Impatient“, Addison-Wesley

Objektorientierung allgemein

- B. Lahres, G. Raýman: „Objektorientierte Programmierung“, Rheinwerk Verlag GmbH (auch frei als „Open-Book“ unter http://openbook.rheinwerk-verlag.de/oop/)

UML (Objekt- und Klassendiagramme)

- M. Jeckle, C. Rupp, J. Hahn, B. Zengler, S. Queins: „UML 2 - glasklar“, Hanser-Verlag
- C. Kecher: „UML 2.5 – Das umfassende Handbuch“, Rheinwerk Verlag GmbH

IDEs zur Java-Entwicklung

- Eclipse: http://www.eclipse.org mit zusätzlichen Plugins:
  - Findbugs: In Eclipse Help → Eclipse Marketplace, „Findbugs“ im Suchfeld eingeben und „Findbugs Eclipse Plugin” installieren
  - Checkstyle zur Überprüfung der Code-Konventionen: In Eclipse Help → Eclipse Marketplace, „Checkstyle“ im Suchfeld eingeben und „Checkstyle Plug-in” installieren
- Netbeans: http://www.netbeans.org
- IntelliJIDEA
  - freie Community-Version: http://www.jetbrains.com/idea/
  - oder Ultimate Edition mit Lizenz der Hochschule: https://ilias.hs-karlsruhe.de/goto.php?target=crs_99205&client_id=HSKA (Kursbeitritt erforderlich)
  - Plug-in:
    - Checkstyle zur Überprüfung der Code-Konventionen → Preferences → Plugins → Browse repositories..., dann nach Checkstyle suchen, anklicken und den Installieren-Button drücken.

Leicht verfügbare Modellierungswerkzeuge

- Visual Paradigm for UML
  - Frei verfügbare Community-Edition, keine Code-Erzeugung in der Community-Edition
  - Download http://www.visual-paradigm.com/
- Magicdraw
  - Frei verfügbare Community-Edition, keine Code-Erzeugung in der Community-Edition
  - http://www.magicdraw.com/
- Borland Together
  - http://www.borland.com/us/products/together/index.html
  - Lizenz bei Frau Knodel in LI 136
- UML Lab
  - Kostenlose Studentenlizenz unter http://www.uml-lab.com/de/uml-lab/academic/
- Netbeans
  - Enthält einen UML-Editor

Übersicht:
Änderungen zum Wintersemester 2016/2017 und Arbeitsaufwand

Änderungen

- keine

Arbeitsaufwand Vorlesung:

- 4 ECTS-Punkte, 4 SWS ergeben 120 Stunden Aufwand (60 Stunden Präsenz, 60 Stunden **eigenständige Arbeit**)

Arbeitsaufwand Übung:

- 3 ECTS-Punkte, 2 SWS ergeben 90 Stunden Aufwand (30 Stunden Präsenz, 60 Stunden **eigenständige Arbeit**)

Markierungen

- Expertenkapitel (nicht klausurrelevant):

- Tafel-/ Rechnerübungen:

Längeres Beispiel:

- Pacman-Beispiel:

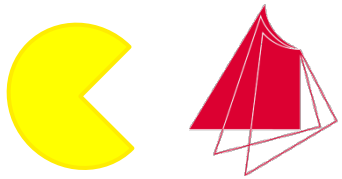
Übersicht

Historie im Informatikstudium

- Bekannt sind aus „Informatik 1“:
  - Datentypen
  - Prozedurale Elemente
  - einfache Klassen, Objekte
  - Arrays
  - Algorithmen zum Suchen und Sortieren
  - Klassen in UML

Übersicht

Durchgängiges Beispiel: Pacman-Klon

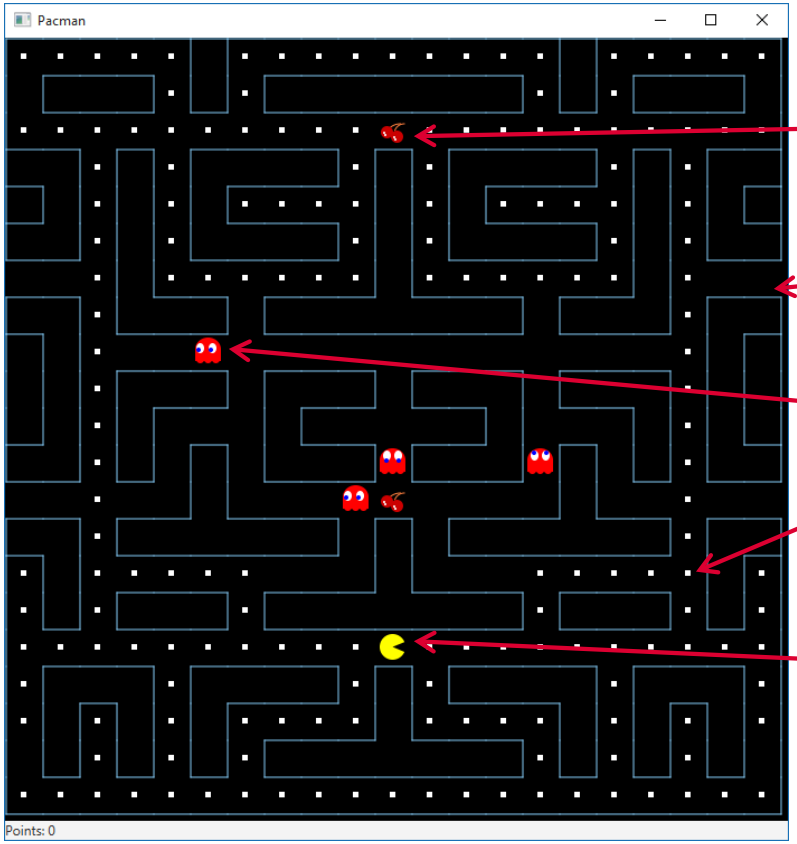


- Java und die Datenstrukturen werden anhand eines kleinen Spielfragmentes erläutert, einer Variation von Pacman:
  - nur einen Level
  - verändertes Spielfeld
  - andere Punktezählung
  - veränderter interner Aufbau, um alle wichtigen Techniken zeigen zu können
  - fehlende Spielelemente des Originals
  - teilweises anderes Verhalten der Figuren
  - Vektor- statt Pixelgrafik
  - plattformunabhängige Implementierung mit JavaFX

Übersicht

Durchgängiges Beispiel: Pacman-Klon

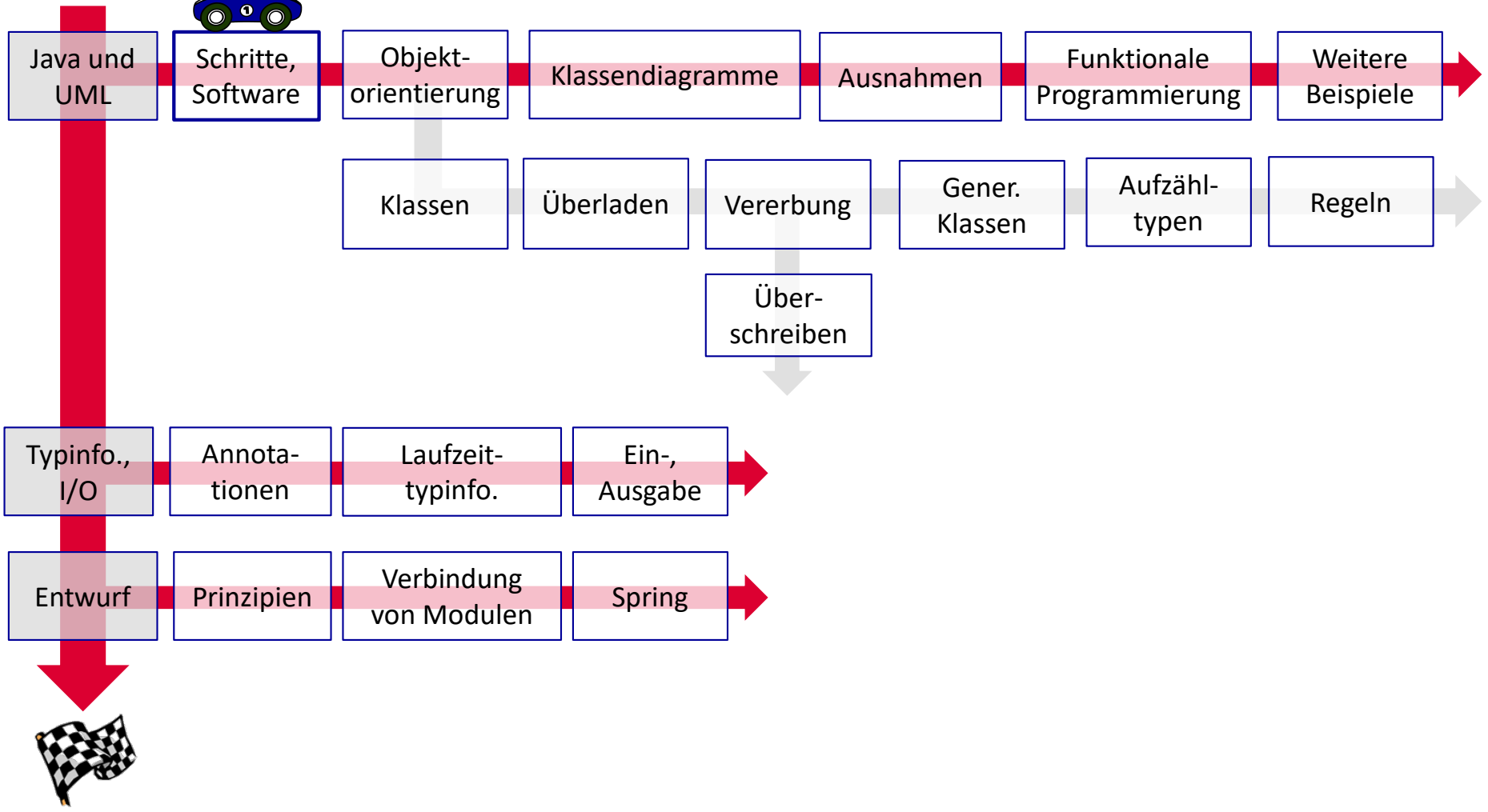
- Spielfeld:



- Spielende: Pacman wurde von einem Geist gefressen, oder alle Essensrationen sind von Pacman gefressen worden.

Arbeitsschritte und Software

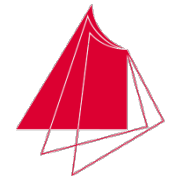
Übersicht



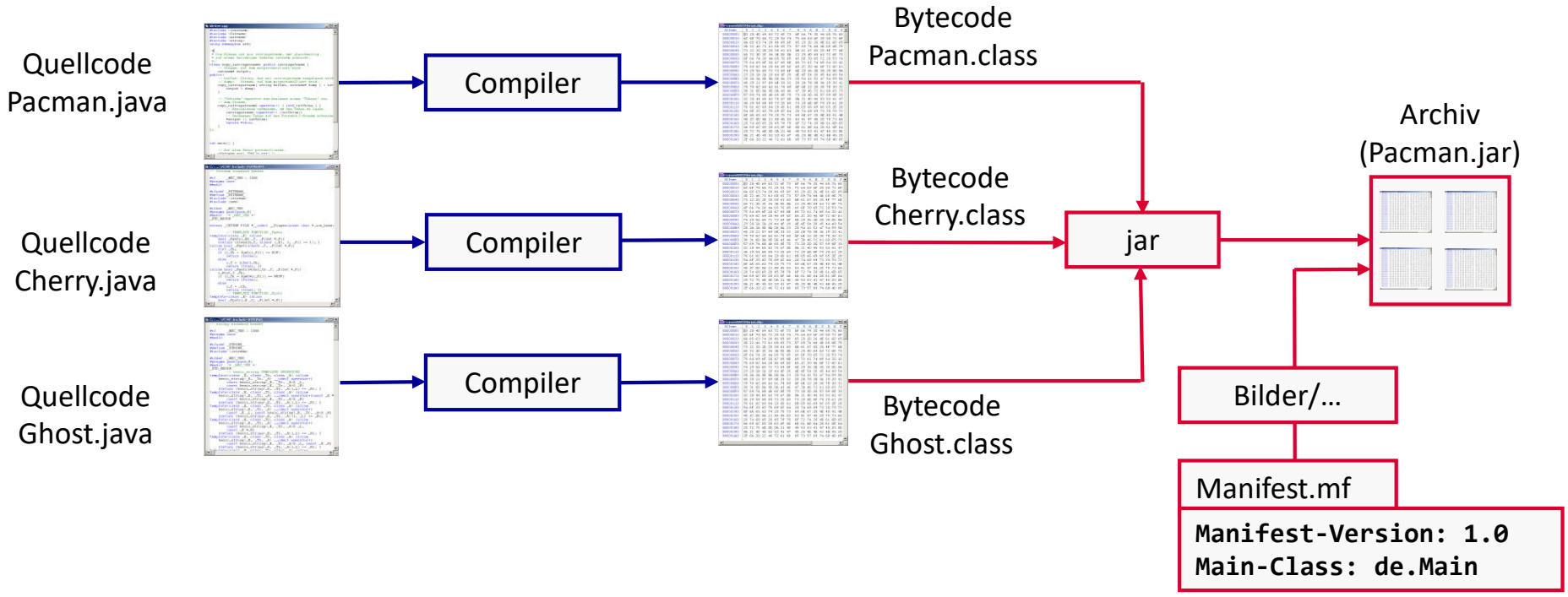
Arbeitsschritte und Software

Motivation

- *Bekannt aus Informatik 1: Schreiben einfacher Java-Programme*
- *Wie kann der Quelltext verwaltet werden?*

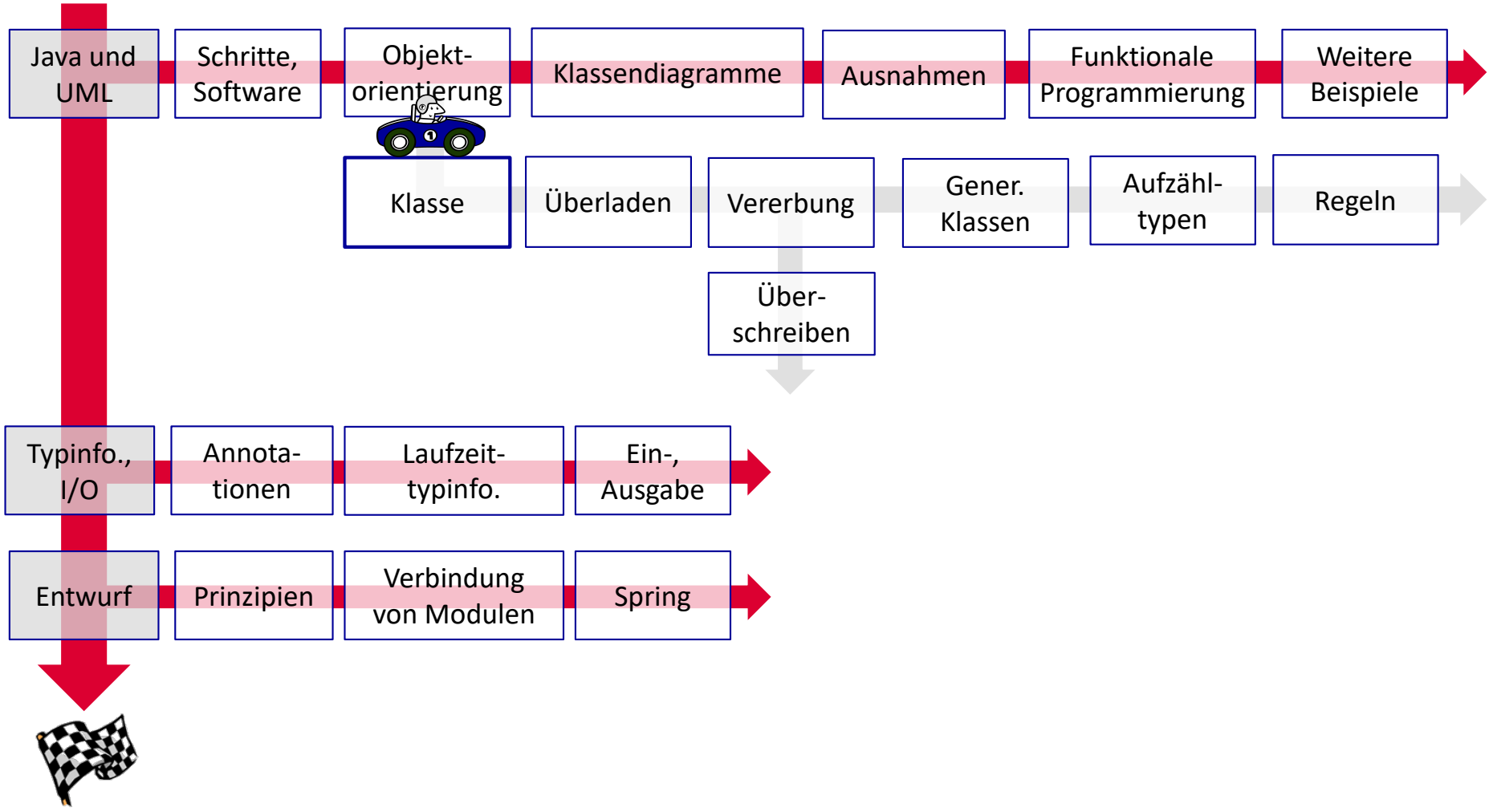


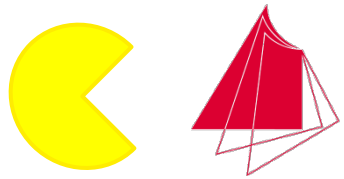
Erzeugen eines ausführbaren Programmes



Klassen und Objekte

Übersicht



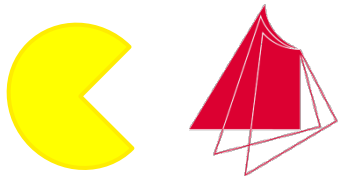


Einige Objekte im Pacman-Spiel

- Pacman
- die Geister 1 bis 4
- einzelne Kirschen
- das Spielfeld

Einige Eigenschaften von Objekten (Zustandsinformationen)

- **Pacman**: bewegt sich nach oben, öffnet den Mund
- **Geist 1**: ist gefährlich, befindet sich auf Zelle (2,3), bewegt sich nach unten und hat eine gewisse „Intelligenz“
- **Geist 2**: ist gefährlich, befindet sich auf Zelle (12,5), bewegt sich nach links und hat ebenso eine gewisse „Intelligenz“



Objekte kommunizieren (Verhalten)

- Das Spielfeld sagt zu Pacman: „Bewege Dich nach oben“

  → Quellobjekt (Spielfeld) schickt dem Ziel (Pacman) eine Nachricht.

Objekte unterscheiden sich (Identität)

- Geist 1 ist nicht Geist 2.
- Alle Objekte sind eindeutig unterscheidbar!

Definition: Objekt

Ein Objekt ist eine konkret vorhandene Einheit mit folgenden Merkmalen:

- **Identität**: Alle Objekte lassen sich eindeutig unterscheiden (z.B. durch die Referenz).
- **Zustand (Daten)**:
  - Alle Attributwerte des Objektes. Der Typ der Attribute wird durch die Klasse festgelegt.
  - Der Zustand ist nur durch das Objekt selbst veränder- und sichtbar, nicht aber von außen.
  - Beziehungen zu anderen Objekten.
- **Eigenschaften (Properties)**:
  - Sie können von außen abgefragt werden.
  - Sie werden unter Umständen aus Daten berechnet, sind aber selbst keine Daten (z.B. Alter = Datum - Geburtsdatum).
- **Verhalten**: Festgelegt durch Methoden der Klasse des Objektes.

Definition: Klasse

Eine Klasse ist die Definition der Attribut-, Eigenschaftstypen und Operationen einer Menge von Objekten.

Definition: Attribut (auch Instanzvariable genannt)

- Ein Attribut ist eine Eigenschaft eines Objektes.
- Es kann ein Datenelement oder ein berechneter Wert (=abgeleitetes Attribut) sein.
- Ein Attribut kommt in allen Objekten der Klasse vor.
- Der Wert des Attributs kann in den Objekten unterschiedlich ausfallen.
- Ein Attribut kann nicht ohne das zugehörige Objekt leben.
- Ein Attribut hat keine Identität.

In der Vorlesung wird der Begriff „Attribut“ immer für ein Datenelement verwendet.

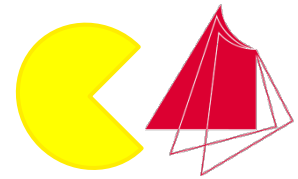
Definition: Operation

- Eine Operation legt fest, welche Funktionalität ein Objekt bereitstellt.
- Unterstützt ein Objekt eine bestimmte Operation, so sichert es einem Aufrufer zu, dass es bei einem Aufruf die Operation ausführen wird.
- Durch die Signatur der Operation wird die Syntax des Aufrufs vorgegeben (Typen der Parameterwerte).
- Die Operation gibt Zusicherungen darüber, welche Resultate die Operation haben wird.
- Die Operation beinhaltet keine Implementierung. Sie beschreibt „lediglich“ Schnittstelle (Signatur) und Zusicherung nach außen.

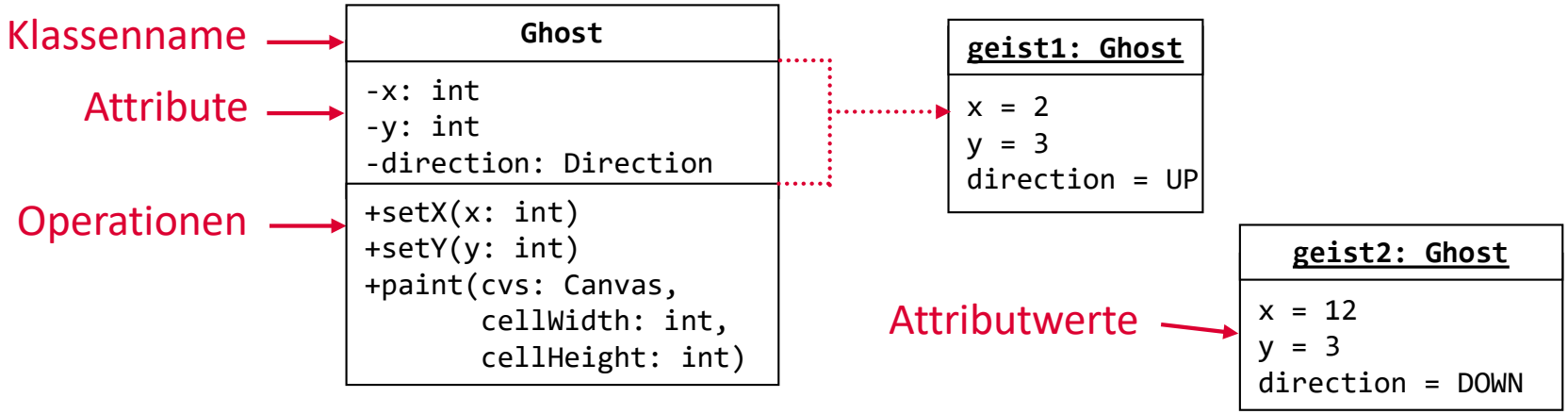
Definition: Methode

- Eine Methode ist die konkrete Implementierung einer Operation.
- Während Operationen die Funktionalität nur abstrakt definieren, sind Methoden für die Realisierung dieser Funktionalität zuständig.

- In der Vorlesung wird die strenge Kategorisierung von Methode und Operation nicht immer aufrechterhalten.
- Hier wird häufig „Methode“ als Synonym für beide Begriffe verwendet.
- „Operation“ wird dort verwendet, wo es speziell auf den reinen Signaturcharakter ankommt.

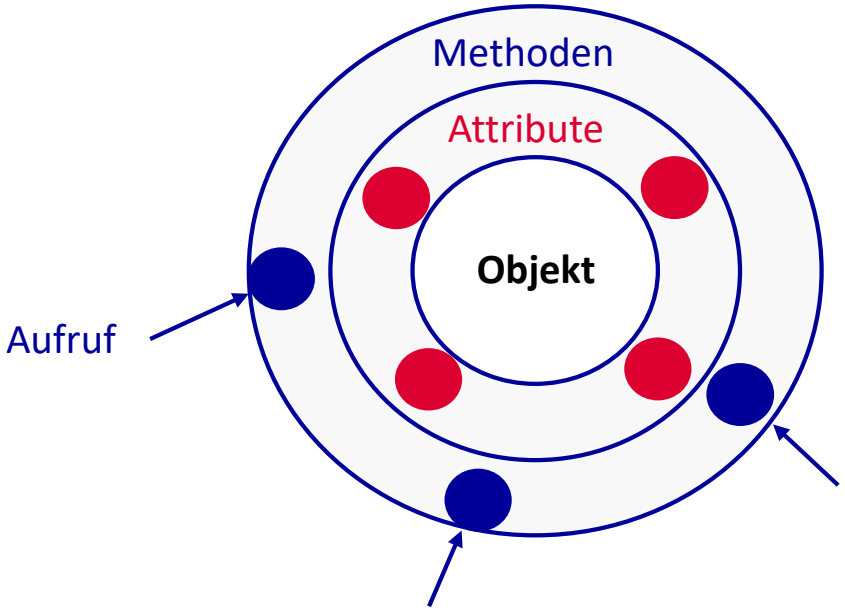


Beispiel zu Klasse und Objekt (unvollständiges Beispiel)



Idee der Kapselung (Information hiding)

- Die Implementierung der Datenstrukturen und Algorithmen innerhalb einer Klasse wird gegenüber den Aufrufern „versteckt“. Sie kann nach außen unsichtbar verändert werden.



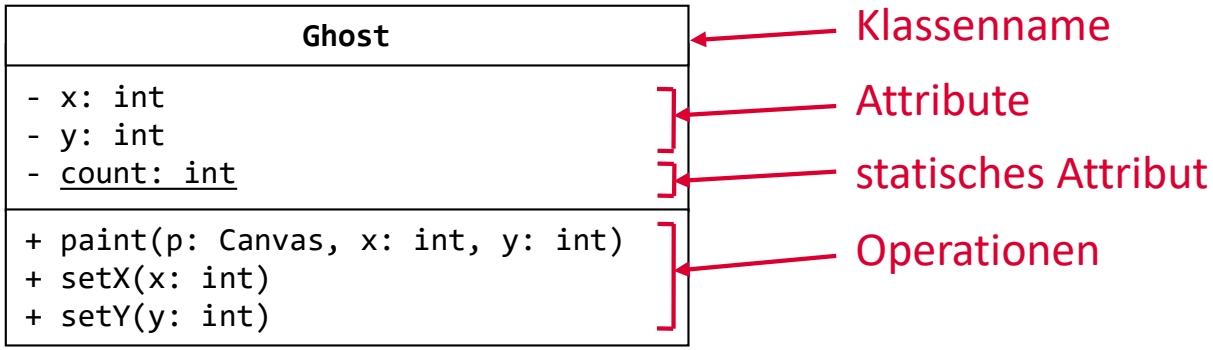
- Vorteile der Kapselung:
  - Interne Zustände und deren Abhängigkeiten bleiben nach außen verborgen → Leichtere Konsistenzhaltung der Daten.
  - Bei Zustandsänderung müssen häufig noch andere Operationen ausgeführt werden. Beispiel:
    - Im Pacman-Spiel wird die Farbe einer Figur geändert. Die Farbe ist ein Attribut der Figur.
    - Danach muss die Figur neu gezeichnet werden.
    - Bei sauberer Kapselung wird durch die Farbänderung automatisch das Neuzeichnen ausgelöst.

- Funktioniert die Kapselung immer? Beispiel:
  - Im Pacman-Spiel werden einzelne Figuren auf dem Spielfeld platziert.
  - Die Figuren haben Attribute wie „Farbe“, „Bewegungsrichtung“ und „Position“.
  - Der aktuelle Spielstand soll auf Festplatte gespeichert werden → Zugriff auf die Attribute zum Speichern notwendig.
- Konsequenz:
  - Jede Figur müsste selbst speichern → Verteilung der Speicherfunktionalität auf viele Klassen → sehr unschön.
  - Eine Klasse speichert, muss aber auf die privaten Attribute der Figuren zugreifen → Aushebelung der Kapselung?
  - Nein: **Kapselung ist immer auf einen Aufgabenbereich beschränkt.**
  - Speicherung ist eine andere Aufgabe, darf also auf die privaten Daten zugreifen.
  - Das gilt aber nicht für Klassen, die für das Zeichnen verwendet werden.

Zugriffsrechte

- **`private`**: Auf private Attribute oder Methoden dürfen nur Methoden der eigenen Klasse zugreifen.
- **`public`**: Auf öffentliche Attribute und Methoden dürfen alle Methoden anderer Klassen zugreifen.
- **`protected`**: für Vererbung (kommt später ...)
- <keine Angabe>: Paketrecht (kommt später …)
- Die Rechte gelten auf Klassenebene: Ein Objekt einer Klasse darf auf private Attribute eines anderen Objektes derselben Klasse zugreifen!

Vereinfachte Darstellung von Klassen in UML:



Attribut (Angaben in eckigen Klammern sind optional):

[***Sichtbarkeit***][/]***Name***[:***Typ***][***Multiplizität***][=***Vorgabewert***] [{***Eigenschaft***}]

- ***Sichtbarkeit***/Zugriffsrecht:
  - **#**: geschützt (protected)
  - **-**: privat (private)
  - **+**: öffentlich (public)
  - **~**: Paket (package)

- **`/`**: Das Attribut ist eine Eigenschaft. Es wird berechnet und muss somit nicht gespeichert werden.
- ***`Name`***: Name des Attributs. Aufgrund von Einschränkungen vieler Programmiersprachen sollte man auf Umlaute usw. verzichten.
- ***`:Typ`***: Typ des Attributs
- ***`Multiplizität`***: Anzahl Ausprägungen des Attributs
  - kann als Array betrachtet werden
  - Angaben sind minimale und maximale Anzahl oder die genaue Anzahl:
    - **`[2]`**: exakt zwei Werte
    - **`[1..2]`**: ein oder zwei Werte
    - **`[1..*]`**: mindestens ein Wert bis beliebig viele Werte
    - **`[0..*]`** bzw. **`[*]`**: beliebig viele Werte
- ***`=Vorgabewert`***: Initialwert für das Attribut
  - muss zum Typ des Attributs passen
  - bei Multiplizität größer als 1: Aufzählung in der Form **`{1, 2, 3, 4}`**

- **{*Eigenschaft*}**: besondere Merkmale des Attributs (Auswahl)
  - **{readOnly}**: Der Wert darf nach der Initialisierung nicht mehr verändert werden (eine Konstante).
  - **{subsets <*Attributname*>}**: Der Wert ist eine Untermenge der Werte, die in dem Attribut **<Attributname>** erlaubt sind.
  - **{union}**: Vereinigung aller Attributwerte, die mit **subsets** spezifiziert wurden.
  - **{redefines <*Attributname*>}**: Redefiniert ein Attribut seiner Basisklasse
  - **{ordered}**: Die Attributwerte müssen geordnet vorliegen. Duplikate sind nicht erlaubt.
  - **{bag}**: Die Attributwerte müssen nicht geordnet vorliegen. Duplikate sind erlaubt.
  - **{seq}** bzw. **{sequence}**: Die Attributwerte müssen geordnet vorliegen. Duplikate sind erlaubt.
  - **{unique}**: Die Attributwerte müssen nicht geordnet vorliegen. Duplikate sind nicht erlaubt.
  - **{composite}**: Das Attribut wird durch Komposition an die Klasse gebunden. Es ist so selbst für die Zerstörung seines Inhalts verantwortlich.

Operation (Angaben in eckigen Klammern sind optional):

[*Sichtbarkeit*]*Name*([*Parameterliste*])[:*Rückgabetyp*] [{*Eigenschaft*}]

- *Sichtbarkeit*/Zugriffsrecht: wie bei Attributen
- *Name*: wie bei Attributen
- *Parameterliste* mit dem folgenden Aufbau:

  [*Übergabemodus*] *Name* :*Typ* [*Multiplizität*][=*Vorgabewert*][{*Eigenschaft*}]

  - *Übergabemodus* (unvollständig):
    - **in**: Der Parameter wird von der Operation nur gelesen. Fehlt der Modus, so wird **in** angenommen. In Java nur für primitive Datentypen möglich.
    - **out**: Der Parameter wird von der Operation nur geschrieben. Umsetzungsmöglichkeit in Java nur für Objekte (Besonderheit **String** → **StringBuffer**, …)
    - **inout**: Der Parameter wird von der Operation gelesen und geschrieben. Umsetzungsmöglichkeit in Java nur für Objekte.

- ***Name***: Name des Parameters
- ***:Typ***: Typ des Parameters
- ***Multiplizität***: Anzahl Ausprägungen des Parameters, die übergeben werden. Die Angaben erfolgen wie bei Attributen. Umsetzungsmöglichkeit in Java: Arrays
- ***=Vorgabewert***: Wert, den der Parameter erhält, wenn er nicht übergeben wird. Umsetzung in Java nicht direkt möglich.
- ***{Eigenschaft}***: siehe Attribute

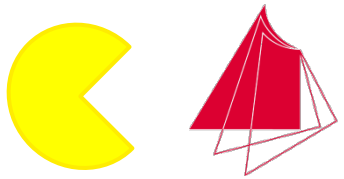
- ***:Rückgabetyp***: Typ des Rückgabewertes, fehlt dieser, so wird **`void`** angenommen. Die Angabe **`void`** ist nicht erlaubt.
- ***{Eigenschaft}***: Angabe spezieller Merkmale des Rückgabetyps, siehe Attribute.
- Und was ist mit Ausnahmen (Exceptions)? UML kennt sie nicht. Ausweg in Form eines Eigenschaftswertes (Tagged Value):

  `{raisedException=`*`NameDerAusnahme`*`}`

  Andere, eigene Eigenschaftswerte sind auch erlaubt → sinnvoll bei MDA.

Klassen und Objekte

Klassen in UML (Pacman-Beispiel)



- Klasse für eine Zelle des Spielfeldes (unvollständig):

Cell
- border: int - food: boolean - exit: boolean
+ paint(cvs: Canvas, x: int, y: int) + isFood(): boolean + isExit(): boolean + getBorder(): int

- Weitere Klassen kommen erst später, weil sie Vererbung bzw. Beziehungen zwischen Klassen benötigen.

Typen von Klassen

- Statisches Typsystem (C++, C#, Java, …):
  - Der Typ von Variablen und Parametern wird im Quelltext festgelegt.
  - Vorteile:
    - bessere Optimierung durch den Compiler möglich
    - saubere Programmstruktur, da Typen direkt im Quelltext ersichtlich sind
    - gute Unterstützung durch IDEs, da diese die Variablentypen erkennen
    - frühzeitige Fehlererkennung durch den Compiler
- Dynamisches Typsystem (z.B. JavaScript):
  - Variablen können beliebige Daten aufnehmen.
  - Der Variablentyp steht erst zur Laufzeit fest.
  - Eigenschaften („Vorteile“):
    - Werte können von beliebigem Typ sein → automatische Konvertierung
    - Ducktyping: Wenn es watschelt wie eine Ente, wenn es schwimmt wie eine Ente, wenn es quakt wie eine Ente, dann behandeln wir es wie eine Ente.

Stark und schwach typisierte Programmiersprachen

- Stark typisierte Sprache: Überwacht das Erstellen und den Zugriff auf alle Objekte so, dass sichergestellt ist, dass Referenzen/Zeiger immer auf Objekte verweisen, die auch die Spezifikation des Typs erfüllen, der für die Variable deklariert ist. Beispiel: Java
- Schwach typisierte Sprachen: Zeiger können auf Objekte verweisen, ohne dass das Objekt notwendigerweise die Spezifikation des Typs der Variablen erfüllt. Beispiel: C++

Syntax von Klassen in Java

- Eine Klasse kapselt Attribute, Operationen und eventuell Operatoren als eine Einheit.
- Beispiel zu Attributen und Methoden:

```
public class Cell {
  private int border;
  private boolean food;
  private boolean exit;

  public int getBorder() {
    return border;
  }
}
```

- Hinweis: Statt „Methode“ wird häufig auch der Begriff „Funktion“ verwendet.
- Statt „Attribut“ werden auch häufig „Instanzvariable“ oder „Variable“ verwendet.

- Alle Attributzugriffe sollten immer über Methoden mit gleichem Namen mit vorangestelltem **get** bzw. **set** erfolgen → Kapselung!

  Beispiel:

```
public class Cell {
  private int border;
  private boolean food;
//...
  public int getBorder() {
    return border;
  }
  public boolean isFood() {
    return food;
  }
  public void setBorder(int nBorder) {
    border = nBorder;
  }
  public void setFood(boolean nFood) {
    food = nFood;
  }
}
```

falls der Lese- und Schreibzugriff erlaubt sein soll → Namensvergabe nach Java-Konvention

Klassen und Objekte
Beispiel (Vektor, Version 1)

- Aus der Mathematik bekannt: Vektoren und Matrizen.
- Klasse **`Vector`**, hier eingeschränkt auf Länge 3 mit **`double`**-Zahlen (noch unvollständig und gefährlich → kommt später besser):

```
package de.hska.iwii.i2;

public class Vector {
  private double[] values =
                      new double[ 3 ];

  public void setValue(int index,
                       double value) {
    values[ index ] = value;
  }

  public double getValue(int index) {
    return values[ index ];
  }
}
```

Vector
- values: double[3] {bag}
+ setValue(index: int, double: value) + getValue(index: int): double

```
package de.hska.iwii.i2;

public class VectorTest {
  public static void main(
                        String[] args) {
    Vector v1 = new Vector();
    v1.setValue(0, 2.0);
    System.out.println(v1.getValue(0));
  }
}
```

Definition: Konstruktor

Konstruktoren dienen der gezielten Initialisierung eines Objektes bei dessen Erzeugung.

- Es kann mehr als einen Konstruktor in einer Klasse geben.

  Beispiel (**`Vector`**, Version 2):

```
public class Vector {
  private double[] values;

  public Vector(int size) {
    values = new double[ size ];
  }

  public void setValue(int index, double value) {
    values[ index ] = value;
  }
  public double getValue(int index) {
    return values[ index ];
  }
}
```

- Ein Konstruktor darf auch einen anderen aufrufen:

```
public class Vector {
  // ...
  public Vector(int size) {
    this(size, 0.0);
  }

  public Vector(int size, double initValue) {
    values = new double[ size ];
    for (int i = 0; i < size; ++i) {
      values[ i ] = initValue;
    }
  }
  // ...
}
```

- Ohne die Angabe eines Konstruktors wird immer automatisch der Defaultkonstruktor (ohne Parameter) erzeugt (sonst nicht).
- Ein Konstruktor wird immer automatisch aufgerufen, wenn ein Objekt der Klasse erzeugt wird.

- Skalarprodukt (immer noch Version 2), fehlerhaft:

```
public class Vector {
  // ...

  public double getScalarProduct(Vector second) {
    double result = 0.0;
    for (int i = 0; i < values.length; ++i) {
      result += values[ i ] * second.values[ i ];
    }

    return result;
  }

  // ...
}
```

- Probleme:
  - Eine Methode oder ein Konstruktor wird mit einer beliebigen Anzahl Werte desselben Typs aufgerufen.
  - Lösungen:
    - Alle Werte kommen in ein Array. Das Array wird übergeben.
    - Java unterstützt die Übergabe einer variablen Parameteranzahl.
- Beispiel Konstruktor des Vektors mit variabler Anzahl **`double`**-Werte:

```
public class Vector {
  private double[] values;

  public Vector(double... initValues) {
    values = new double[initValues.length];
    for (int i = 0; i < initValues.length; ++i) {
      values[ i ] = initValues[ i ];
    }
  }
  // ...
}
```

- Aufrufe:

```
Vector v1 = new Vector(1.0, 2.0, 3.0, 4.0); // Länge 4
Vector v2 = new Vector(1.0, 2.0);           // Länge 2
Vector v3 = new Vector();                   // Länge 0
```

- Intern werden die Parameter in einem Array abgelegt:
  - Die Anzahl der Parameter kann mit ***`array`*`.length`** ermitteln werden.
  - Die Parameter werden mit Array-Zugriffen ausgelesen.
- Das funktioniert auch für Methoden.

- Klassen kapseln Daten und arbeiten selbst auf ihren eigenen Daten:
  "Don't ask for the information that you need to do something;
  rather, ask the object that has that information to do the job for you."
- Beispiel (**so nicht**):

```
public class Article {
  private double price;

  public double getPrice() {
    return price;
  }
  // usw.
}
```

```
// Implementierung einer Preiserhöhung in einer anderen Klasse
public void increasePrice(Article article, double percentage) {
  article.setPrice(article.getPrice() * (1 + percentage / 100.0));
}
```

Problem: Lesen, Manipulation und Schreiben von **`Article`**-Daten. Die **`Article`**-Operation gehört in die Klasse!

- Beispiel (**so ist es ok**):

```
public class Article {
  private double price;

  public double getPrice() {
    return price;
  }
  // usw.
  // Implementierung einer Preiserhöhung
  public void increasePrice(double percentage) {
    price *= (1 + percentage / 100.0));
  }
}
```

Klassen und Objekte

Statische Attribute und Methoden

- Bisher: Jedes Attribut einer Klasse existiert in jedem Objekt der Klasse.
- Manchmal gewünscht: Auch „globale“ Attribute, die nur einmal für eine Klasse existieren → Alle Objekte einer Klasse teilen sich dieses Attribut:

```
static Typ Attribut-Name;
```

- Zugriff auf statische Attribute:
  - Statische Methoden:

```
static Typ Methode(Parameter);
```

  - Aufruf einer statischen Methode einer Klasse auch ohne ein konkretes Objekt.
- Statische Attribute können als globale Attribute innerhalb einer Klasse betrachtet werden.

- Beispiel kommt nachher im Rahmen der Einführung von Referenzen.

Klassen und Objekte

final-Parameter und final-Werte

- **`final`**-Parameter und **`final`**-Werte können nicht verändert werden → Konstante!

  Beispiel:

```
public void add(final int arg) {
  // ...
}
```

- **`final Vector vector`**: Unveränderliche Referenz auf ein Vektor-Objekt, dessen Inhalt aber verändert werden kann.

Parameterübergabe (primitive Datentypen)

- Werte primitiver Datentypen werden immer per Wert übergeben. Es wird eine lokale Kopie erzeugt. Änderungen am Wert innerhalb einer Methode wirken nicht nach außen.

  Beispiel:

```
public void doSomething(int xx){
  xx = 2;
}

public int doSomethingElse(){
  int x = 21;
  doSomething(x);
  System.out.println(x);
  return 0;
}
```

  Ausgabe: 21

Parameterübergabe (Objekte)

- Objekte werden nicht übergeben. Statt dessen werden Kopien von Referenzen auf Objekte übergeben werden. Damit sind Änderungen nach außen sichtbar. Beispiel:

```
public void doSomething(Vector v){
  v.setValue(0, 42);
}

public int doSomethingElse(){
  Vector v = new Vector(3);
  doSomething(v);
  System.out.println(v.getValue(0));
  return 0;
}
```

Ausgabe: 42

- Zusatznutzen einer Referenzkopie: Das Anlegen einer Objekt-Kopie kann zeitaufwändig sein.
- Hinweis: Die übergebenen Objekte sollte möglichst nicht verändert werden. Die Rückgabe eines Wertes ist ein sauberer Weg → andere Lösung: „Value Objects“ (kommt gleich)

- Es gilt dasselbe wie bei Übergaben:
  - Werte primitiver Datentypen werden als Kopie und
  - Objekte per Kopie der Referenz auf das Objekt zurückgegeben.

Klassen und Objekte

Parameterübergabe bei Objekten: „Value Objects“

- Es werden nur Referenzen auf Objekte übergeben, keine Kopien der Objekte.
- Vorteile:
  - Effizienter als Kopie der Objekte
  - Einfacher Mechanismus
- Problem:
  - Die aufgerufene Methode kann das Objekt verändern, ohne dass es der Aufrufer erfährt!
- Verhinderung des Problems: Übergabe unveränderlicher Objekte als „Wert-Objekte“ („Value Objects“)
  - Alle Attribute sind Konstanten.
  - Es gibt keine Methoden, die das Objekt verändern.
  - Bei jedem Veränderungsversuch wird ein neues Objekte erstellt, das alte aber unverändert beibehalten.
  - Konsequenz: Erst einmal ein gewisser Mehraufwand zur Laufzeit → führt in bestimmten Situationen („Multithreading“, kommt im dritten Semester) aber zu deutlichen Vereinfachungen
- Später im Semester: unveränderliche Datenstrukturen

- Beispiel: Geld

```
public class Money {
  private Currency currency;  // Währung, als enum definiert.
  private int value;  // Summe
  public Money(int value, Currency currency) {
    this.value = value;
    this.currency = currency;
  }

  // Zwei Geld-Beträge in einer Währung addieren.
  public Money add(Money other) {
    // Nur bei gleicher Währung
    if (this.currency == other.currency) {
      return new Money(this.value + other.value, this.currency);
    }
    // Fehlerbehandlung, wie auch immer...
  }
  // ...
}
```

- Wenn Geldbeträge addiert oder subtrahiert werden, wird immer als Ergebnis ein neues Objekt erzeugt. Es wird niemals ein Objekt verändert.

- Beispiel unveränderlicher Vektor (Projekt **`Vector 2 (Value Objects)`**):

```
public class Vector {
  // Achtung: Die Werte im Array sind immer noch veränderlich!
  private final double[] values;

  public Vector(int size, double initValue) {
    values = new double[ size ];
    for (int i = 0; i < size; ++i) {
      values[ i ] = initValue;
    }
  }

  public Vector(double... initValues) {
    values = new double[initValues.length];
    for (int i = 0; i < initValues.length; ++i) {
      values[ i ] = initValues[ i ];
    }
  }
```

Attribut darf im Konstruktor einmal verändert werden, obwohl es final ist.

```
  /**
   * Vektor unverändert lassen, neuen Vektor erzeugen!
   * @param index Index des zu verändernden Wertes.
   * @param value Neuer Wert am Index.
   * @return Neuer Vektor mit verändertem Wert.
   */
  public Vector setValue(int index, double value) {
    // Die Kopie des Vektors würde man durch die
    // clone-Methode erstellen --> noch nicht bekannt.
    // Diese Lösung hier weist Mängel auf --> mehr dazu später.
    Vector copy = new Vector(values);
    copy.values[ index ] = value;
    return copy;
  }

  public double getValue(int index) {
    return values[ index ];
  }
}
```

Klassen und Objekte

Parameterübergabe bei Objekten: „Value Objects“

- „Value Objects“ sind im JDK nicht sonderlich verbreitet.
- Es gibt aber andere Klassenbibliotheken, die dieses Entwurfsmuster einsetzen → kommt später im Datenstruktur-Kapitel.
- Die Vorteile von „Value Objects“ werden aber häufig erst im Zusammenhang mit Multithreading deutlich.

Für die primitiven Datentypen existieren Wrapper-Klassen

Typ	Größe	Wertebereich	Wrapper
`boolean`		`true`, `false`	`Boolean`
`char`	16 Bit	`'\u0000'`, bis `'\uFFFF'`	`Character`
`byte`	8 Bit	-2^7 bis 2^7-1	`Byte`
`short`	16 Bit	-2^{15} bis $2^{15}-1$	`Short`
`int`	32 Bit	-2^{31} bis $2^{31}-1$	`Integer`
`long`	64 Bit	-2^{64} bis $2^{64}-1$	`Long`
`float`	32 Bit	2^{-149} bis $(2-2^{-23})\cdot 2^{127}$	`Float`
`double`	64 Bit	2^{-1074} bis $(2-2^{-52})\cdot 2^{1023}$	`Double`
`void`			`Void`

Eigenschaften der Wrapper

- Sie kapseln einen Wert eines primitiven Datentyps (ein **`Integer`**-Objekt nimmt genau einen **`int`**-Wert auf).
- Der gekapselte Wert ist unveränderlich.
- Sie besitzen Methoden zur Konvertierung vom/in den Wrapper.

Wozu dienen die Wrapper?

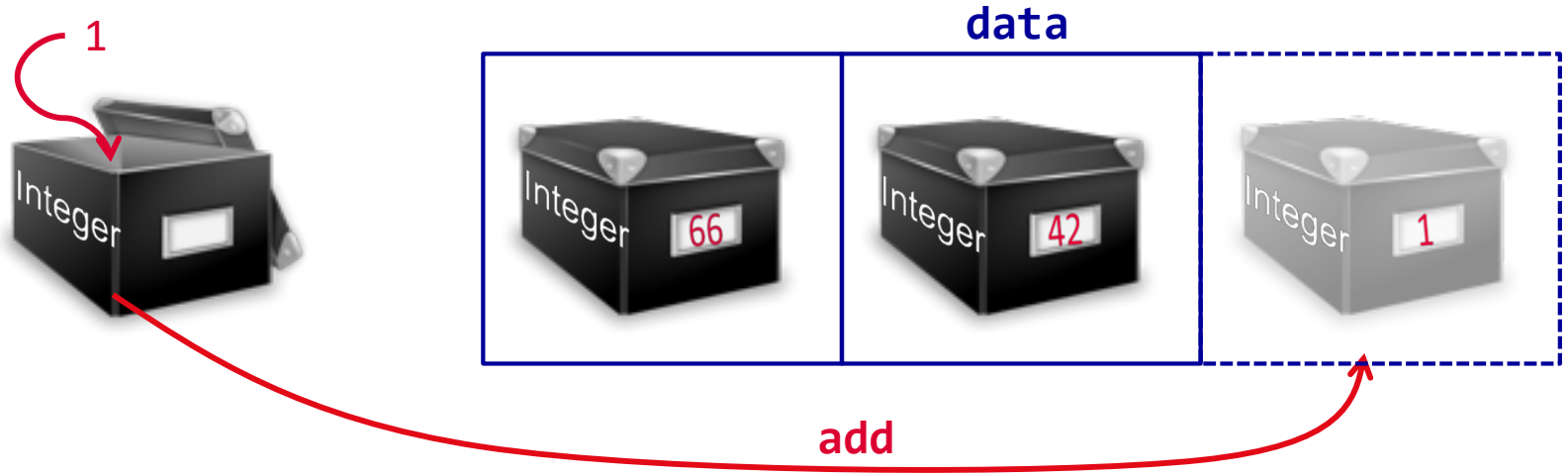
- Manche Methoden und Klassen erwarten Objekte und keine primitiven Datentypen → die Wrapper kapseln die Daten dazu.
- Beispiel aus dem 1. Semester: Eine **`ArrayList`** mit **`int`**-Werten kann so verwendet werden:

```
ArrayList<Integer> data = new ArrayList<>();  // primitive Datentypen sind
                                              // nicht möglich
data.add(500);                                // wird automatisch zu
                                              // data.add(Integer.valueOf(500));
```

Automatisches Einpacken mit Wrappern (autoboxing)

- Automatisches „Einpacken“ (siehe vorheriges Beispiel)

```
ArrayList<Integer> data = new ArrayList<>();
data.add(66);  // autoboxing
data.add(42);  // autoboxing
data.add(1);   // autoboxing
```



- Was passiert genau?
  - Für die ganze Zahlen -128 bis 127 existieren vordefinierte Wrapper-Objekte, die verwendet werden.
  - Für andere Zahlenbereiche sowie **`float`** und **`double`** werden jeweils neue Objekte erzeugt → Probleme beim Vergleichen:

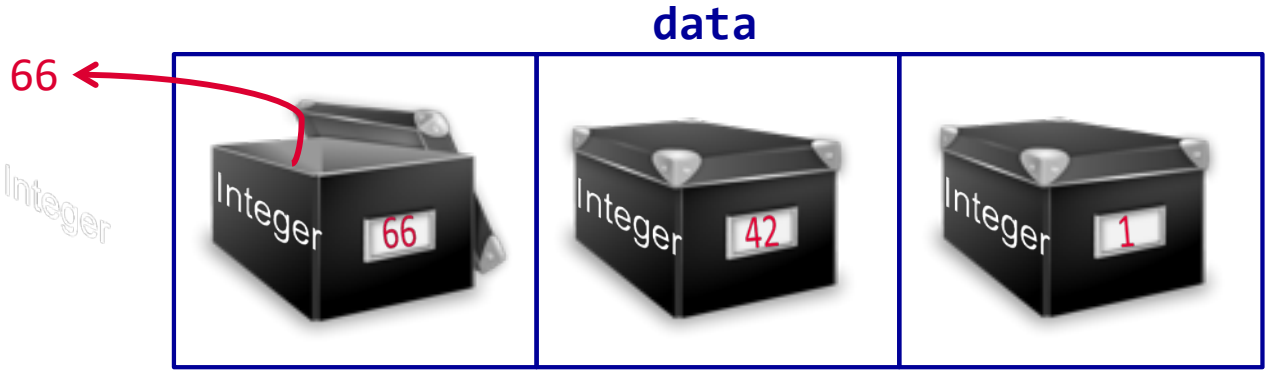
```
Integer boxedSmall1 = 127;
Integer boxedSmall2 = 127;
System.out.println(boxedSmall1 == boxedSmall2);   // true
```

```
Integer boxedSmall1 = 128;
Integer boxedSmall2 = 128;
System.out.println(boxedSmall1 == boxedSmall2);   // false
```

Automatisches Auspacken mit Wrappern (unboxing)

- Automatisches „Auspacken“

```
ArrayList<Integer> data = new ArrayList<>();
data.add(66);  // autoboxing
data.add(42);  // autoboxing
data.add(1);   // autoboxing
int value = data.get(0); // unboxing
```



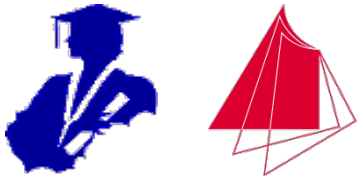
- Was passiert genau?
  - Eine **`null`**-Referenz kann nicht umgewandelt werden und führt zu einer Exception:

```
int value = (Integer) null;    // führt zu einer NullPointerException
```

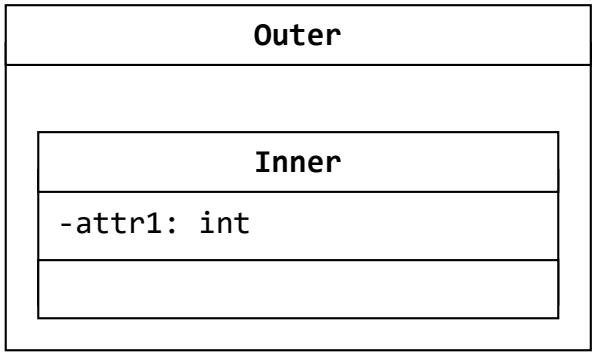
  - Es lauern Fallen bei der Umwandlung von Zahlen:

```
Integer i1 = new Integer(42);
Integer i2 = new Integer(42);
System.out.println(i1 >= i2);   // true, unboxing in int-Werte
System.out.println(i1 <= i2);   // true, unboxing in int-Werte
System.out.println(i1 == i2);   // false, Vergleich der Referenzen!
```

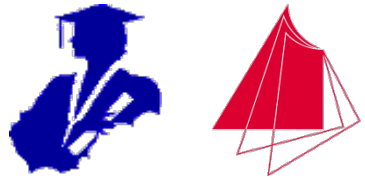
Klassen und Objekte
Innere Klassen



- Klassen lassen sich ineinander schachteln.
- Java unterstützt drei unterschiedliche Arten innerer Klassen:
  - nicht-statische innere Klassen
  - statische innere Klassen
  - anonyme innere Klassen
- Einsatzgebiete:
  - Die inneren Klassen werden hauptsächlich in der äußeren benötigt oder von dieser erzeugt.
  - Beispiel (kommt später): Iteratoren



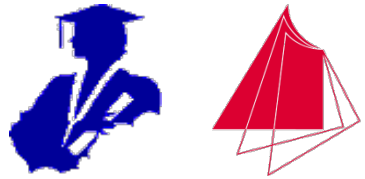

```
public class Outer {
  class Inner {
    int attr1;
  // ...
  }
  // ...
}
```



- Einsatz: Das Erzeugen eines Objektes der inneren Klasse erfolgt immer innerhalb der Grenzen der äußeren Klasse (z.B. in einer der Methoden oder im Konstruktor).
- Beispiel:

```
public class Outer {
  class Inner {
    private int attr1;
    public void method1(){}
    public void method2(){
      method1();
      Outer.this.method1();
    }
  }
  public void method1(){}
}
```

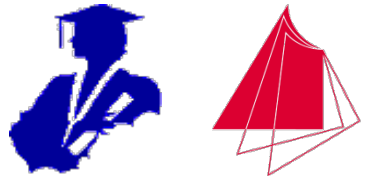
- Innere Klassen dürfen weder statische Methoden noch statische Attribute besitzen.
- **`Outer`** darf auf alle Methoden und Attribute von **`Inner`** zugreifen.
- **`Inner`** darf auf alle Methoden und Attribute von **`Outer`** zugreifen.
- Innere und äußere Klasse sind somit fest miteinander verbunden.
- Erzeugte Dateien: **`Outer.class`**, **`Outer$Inner.class`**



- Einsatz: Das Erzeugen eines Objektes der inneren Klasse kann außerhalb der Grenzen der äußeren Klasse erfolgen.
- Beispiel:

```
public class Outer {
  static class Inner {
    private int attr1;
    public void method1(){}
    public void method2(){
      method1();
    }
  }
  private int attr2;
  public void method1(){}
}
```

- Die innere Klasse ist nicht an die äußere gekoppelt und kann daher nicht auf Attribute und Methoden der äußeren Klasse zugreifen.
- Erzeugte Dateien: **`Outer.class`**, **`Outer$Inner.class`**



- Einsatz: Mit dem Erzeugen einer Klasse wird auch gleichzeitig ein Objekt angelegt. Die Klasse wird nur einmalig benötigt.
- Häufig im Zusammenhang mit dem Implementieren von Schnittstellen verwendet:
- Beispiel:

```
public class Outer {
  public void method() {
    Runnable runObject = new Runnable() {
      @Override
      public void run() {
        System.out.println("Huhu");
      }
    };
    runObject.run();
  }
}
```

- **Runnable** ist eine Schnittstelle → kommt noch genauer
- Die anonyme Klasse darf nur auf finale lokale Variablen der Methode zugreifen.
- Die anonyme Klasse darf auf Attribute und Methoden der äußeren Klassen zugreifen.
- Erzeugte Dateien: **Outer.class**, **Outer$1.class**

Klassen und Objekte

Pakete

- Probleme:
  - Was passiert, wenn zwei Hersteller Klassen mit demselben Namen erstellt haben und beide Klassen in einem Projekt zusammen benutzt werden sollen?
  - Wie können logisch zusammengehörige Klassen gruppiert werden?
- Ein Paket sollte eine Anzahl logisch zusammengehöriger Klassen, Schnittstellen, Ausnahmen, Fehler, Aufzähltypen und Annotationen beinhalten.

  Beispiel:
  - alle Klassen, die mathematische Berechnungen durchführen
  - alle Klassen, die Text formatieren
- Der Paketname entspricht einem Verzeichnis im Dateisystem.

  Beispiel:

  Klasse **`Complex`** im Paket **`de.hska.iwii.math`**: Sowohl die Quelltextdatei **`Complex.java`** als auch die Bytecode-Datei **`Complex.class`** sollten in in einem Unterverzeichnis **`de/hska/iwii/math`** liegen.
- Festlegung der Paket-Zugehörigkeit einer Klasse: Anweisung in der Klasse der Form **`package Paket-Name;`**. Diese Anweisung sollte die erste in der Datei sein.

- Verwendung einer Klasse eines anderen Pakets:
  - Variante 1: **`import Paketname.*;`** Beispiel:

```
import de.hska.iwii.math.*;              // Bindet alle Klassen des Pakets
                                         // de.hska.iwii.math ein
//...
Complex value = new Complex();
```

  - Variante 2: **`import Paketname.Klassenname;`** Beispiel:

```
import de.hska.iwii.math.Complex;        // Bindet Complex des Pakets
                                         // de.hska.iwii.math ein.
//...
Complex value = new Complex();
```

  - Variante 3: Verwendung des vollständigen Klassennamens. Beispiel:

```
//...
de.hska.iwii.math.Complex value = new de.hska.iwii.math.Complex();
```

- Nach Aussage von Oracle sollte Variante 1 vermieden werden.
- Das Paket **`java.lang`** des JDK ist immer automatisch eingebunden.

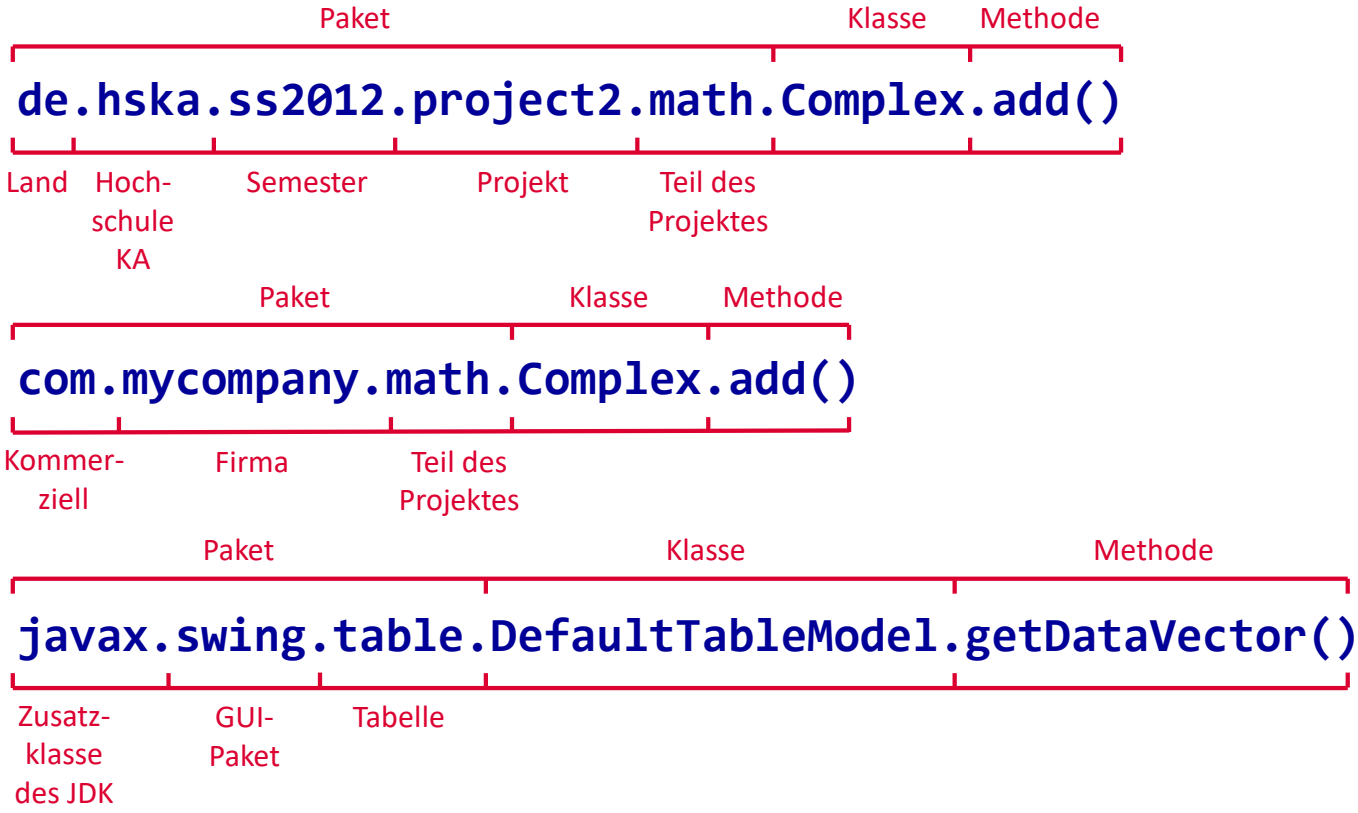
Klassen und Objekte

Pakete: Namensaufbau

- Allgemeiner Aufbau eines Namens (kann variieren):

  Organisation.Paketname.Klassenname.Methodenname

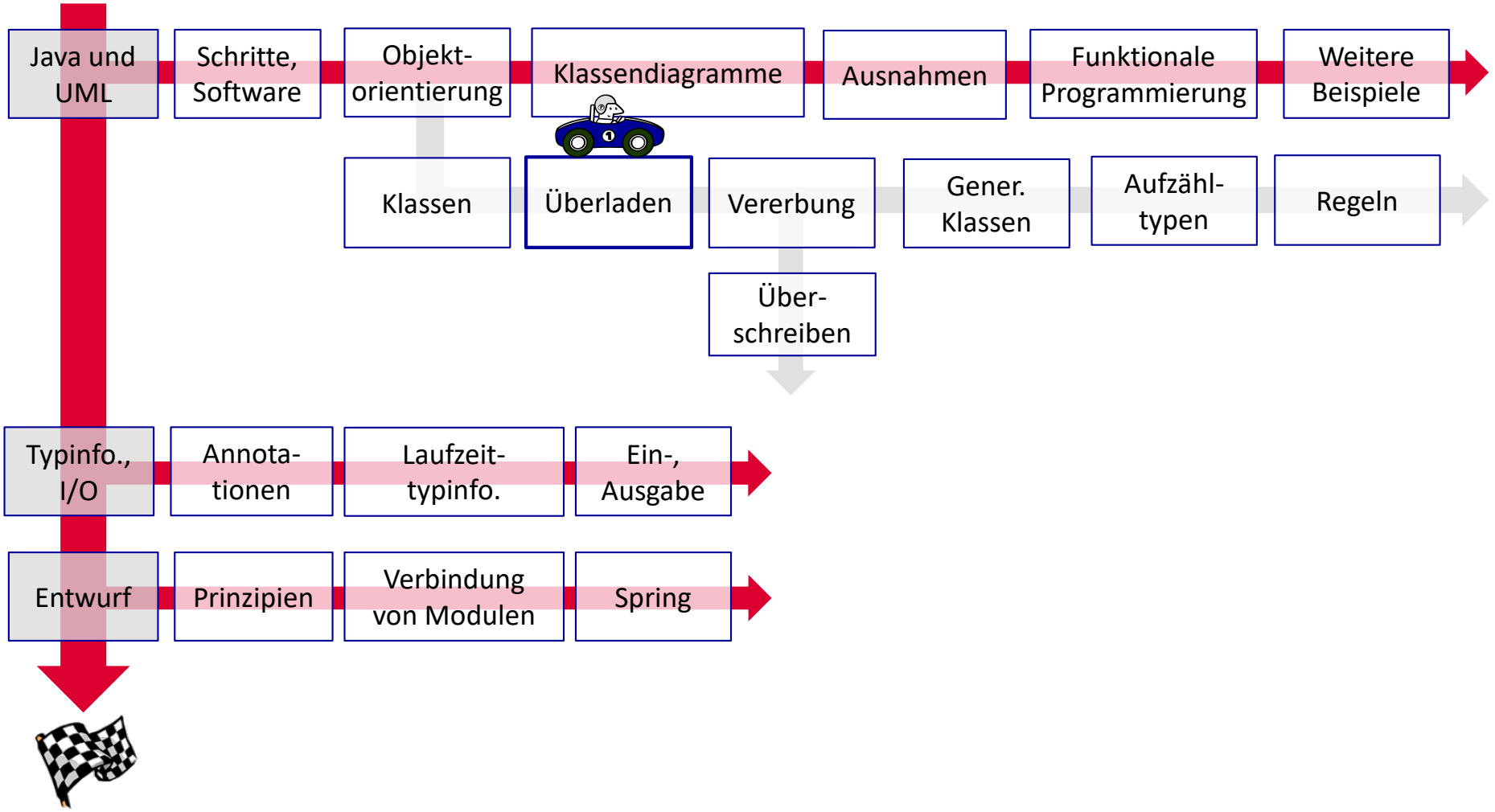
  Beispiele:



- Innerhalb eines Pakets müssen die Namen eindeutig sein: Es darf eine Klasse **`Complex`** mehrfach geben, solange sie in unterschiedlichen Paketen liegt.
- Beispiel: **`java.util.Timer`** und **`javax.swing.Timer`**
- So ganz stimmt die Aussage nicht:
  - Der komplette Paketname (der Pfad inkl. Klassenname) darf mehrfach vorkommen, wenn die Pakete durch unterschiedliche Klassenlader (Classloader) geladen werden.
  - Also: Klassenlader + Paketname muss eindeutig sein!
- Es ist mindestens ein Klassenlader in der virtuellen Maschine aktiv:
  - Er kennt mehrere Dateipfade oder jar-Dateien, aus denen er die Klassen und Schnittstellen lädt.
  - Weitere Klassenlader können erstellt und verwendet werden.

Überladen von Methoden

Übersicht



Überladen von Methoden

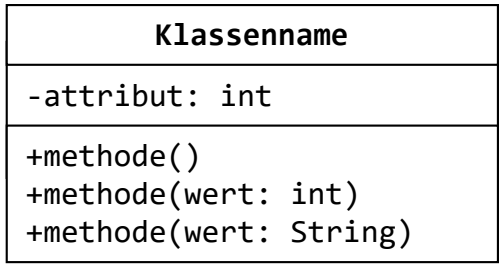
Definition: Überladen von Methoden

Eine Klasse besitzt zwei oder mehr Methoden mit demselben Namen und unterschiedlichen Parametertypen.

Die passende Methode wird durch den Compiler ermittelt, indem er auch Typkonvertierungen durchführt.

Der Rückgabetyp darf unterschiedlich sein.

Beispiel:



Überladen von Methoden

- Einsatz: Verwendung der gleichen Methode mit unterschiedlichen Parametertypen.
- Vorteil: Keine explizite Konvertierung der Argumente im Aufruf.
- Beispiel: Vereinfachter Auszug aus der bereits eingeführten Klasse **`GameController`**, die in Pacman die Figuren steuert.

GameController
`+collisionOfPacmanWith(ghost: Ghost)` `+collisionOfPacmanWith(cherry: Cherry)`

Überladen von Methoden

Auswahl der passenden Methode

Auswahlkriterien durch den Compiler:

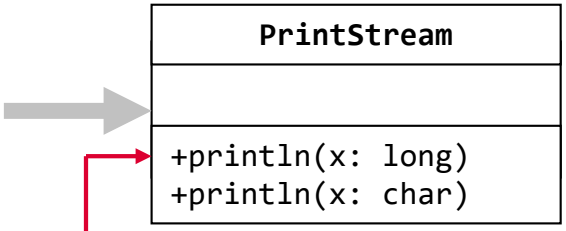
- Die Anzahl der Parameter muss zum Aufruf passen.
- Alle Übergabeparameter müssen in die Parametertypen der Methode konvertierbar sein.

  Beispiel:

```
public class PrintStream {
  public void println(long x){}
  public void println(char x){}
}
```

  Aufruf:

```
PrintStream out = ...
out.println(123);
```



int wird zu **long**

- Für alle Übergabeparameter wird der kleinste passende Typ gesucht.

  Beispiel: **char** → **int**, vor **char** → **long**.
- Ein größerer Typ wird nie in einen kleineren Typ umgewandelt.

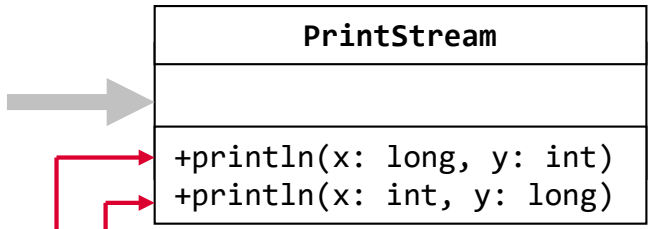
Überladen von Methoden

- Es wird die Methode mit der kleinsten Summe der „Konvertierungsabstände“ aller Parameter ermittelt.
- Existieren mehrere Methoden mit gleichem minimalen Abstand, so wird ein Übersetzungsfehler gemeldet.
- Beispiel (OK, ziemlich sinnlos):

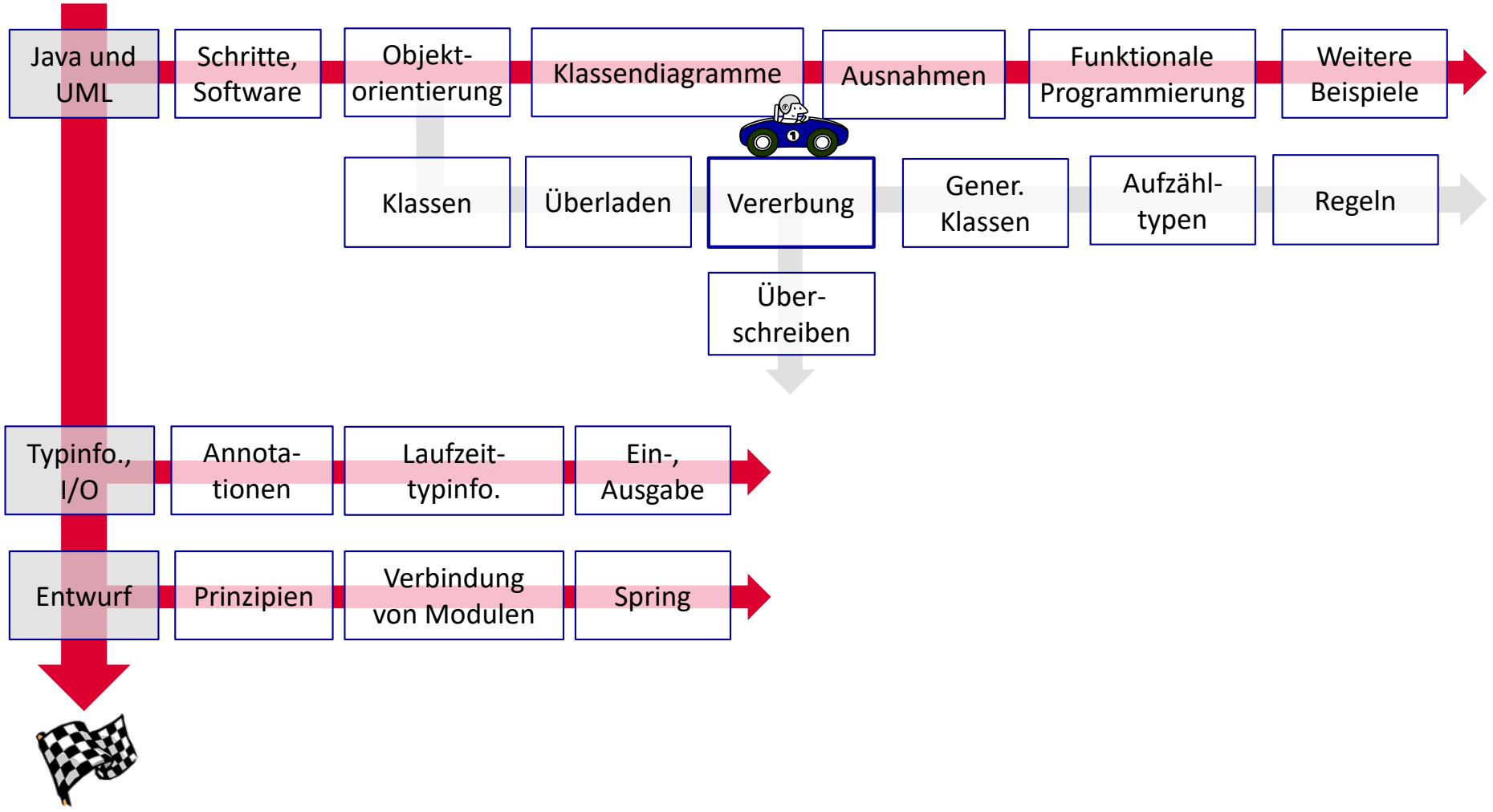
```
public class PrintStream {
  public void println(long x, int  y);
  public void println(int  x, long y);
}
```

Aufruf:

```
PrintStream pr = new PrintStream();
pr.println(123, 456 );    → Fehler !
pr.println(123L,456 );
pr.println(123, 456L);
```



Vererbung
Übersicht

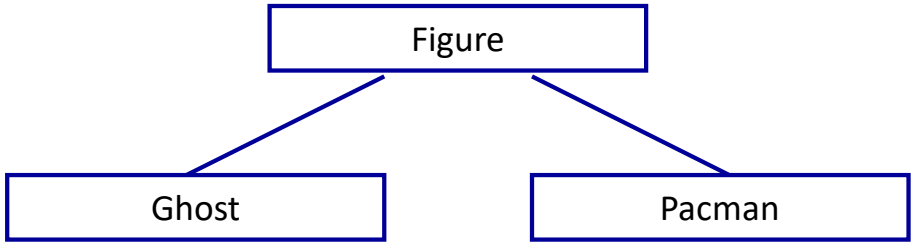


Vererbung
Motivation

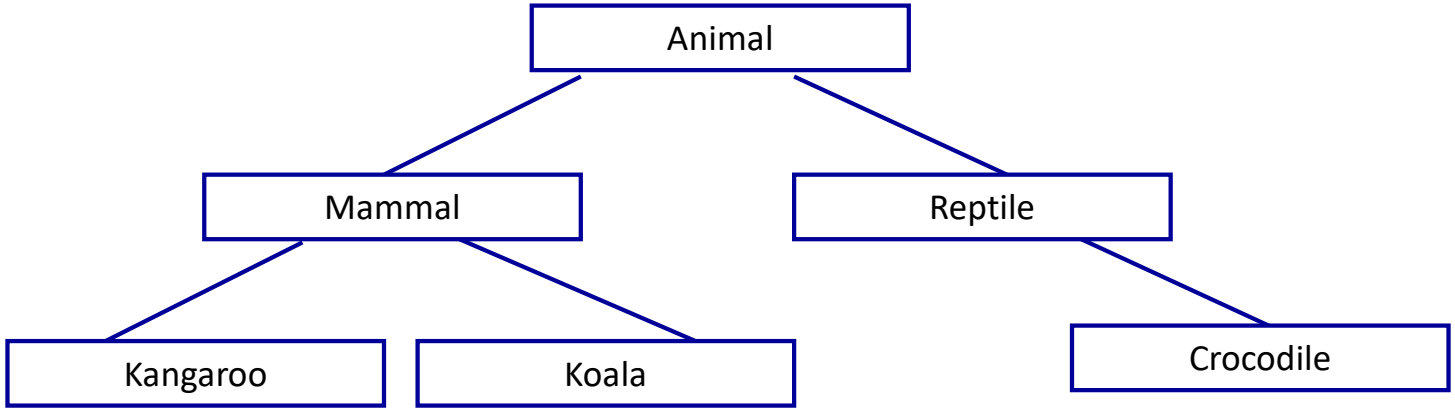
- *Wie können existierende Klassen wiederverwendet werden?*
- *Welche Methoden sollte eine Klasse immer unterstützen?*
- *Welche Arten von Beziehungen können zwischen Objekten bestehen?*

Vererbung
Motivation

- Eigenschaften, die für eine Figur gelten, gelten eventuell auch für spezielle Figuren:



- Vererbungsbeziehungen können mit der Vererbungshierarchie in der Biologie verglichen werden:



- Ein Koalabär ist ein Säugetier (Mammal) ist ein Tier.

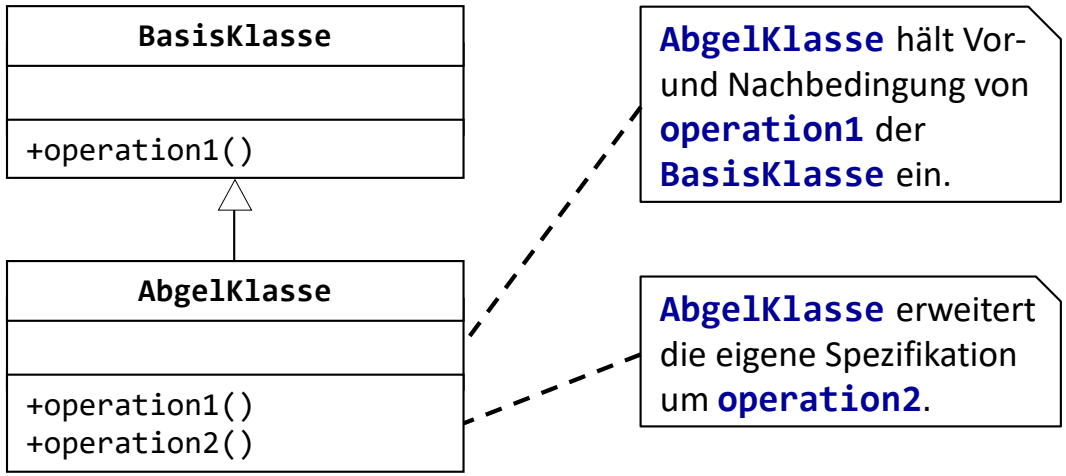
Vererbung
Begriffe

- Es gibt zwei Arten der Vererbung in Programmiersprachen:
  - Vererbung der Spezifikation: wichtiges Konzept der objektorientierten Programmierung
  - Vererbung der Implementierung: Mittel zur Vermeidung von Redundanzen mit einigen konzeptuellen und praktischen Problemen
- Beide Techniken werden häufig unter dem Begriff der Vererbung zusammengefasst.

Definition: Vererbung

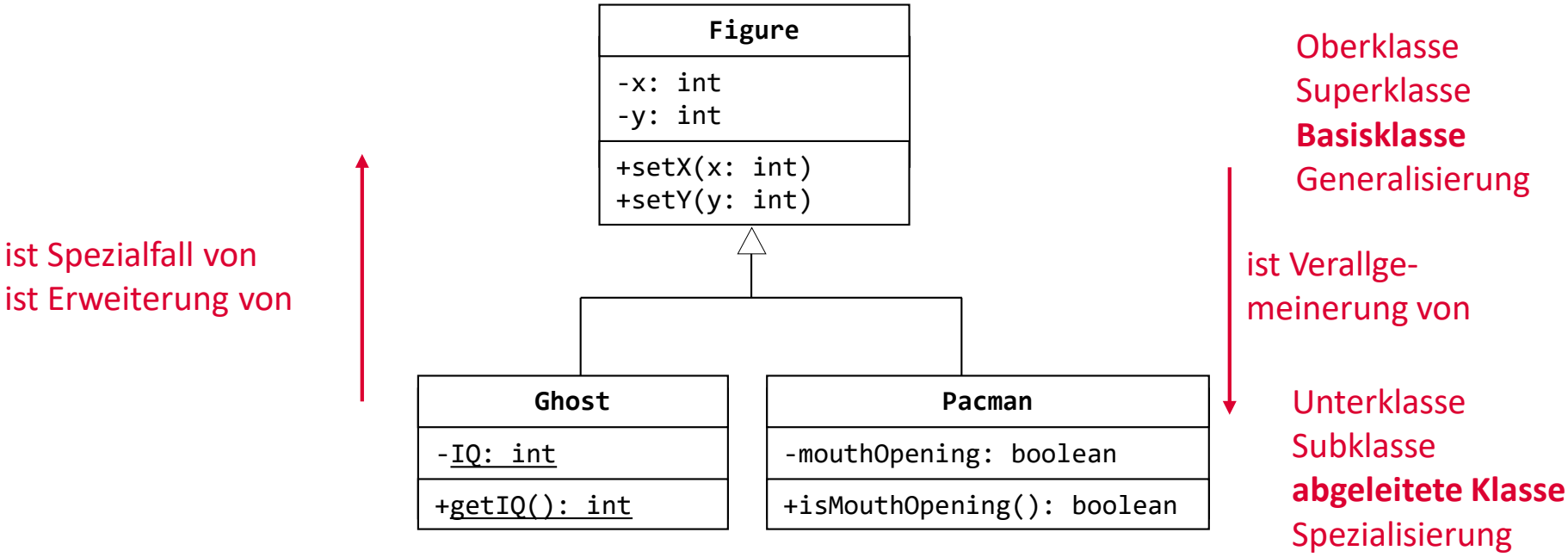
Vererbung ist eine Programmiersprachenkonzept zur Umsetzung einer Relation zwischen einer Ober- und einer Unterklasse:

Eine Klasse **AbgelKlasse** ist dann eine Unterklasse der Klasse **BasisKlasse**, wenn **AbgelKlasse** die Spezifikation von **BasisKlasse** erfüllt, umgekehrt aber **BasisKlasse** nicht die Spezifikation von **AbgelKlasse**. Die Klasse **BasisKlasse** ist dann eine Oberklasse von **AbgelKlasse**.



Vererbung
Begriffe

- Die Oberklasse vererbt ihre Spezifikation (Vor- und Nachbedingungen) an die Unterklasse.
- Durch Unterklassen wird das Verhalten der Oberklasse nicht verändert.
- Durch Vererbung (Ableitung) entsteht eine Klassenhierarchie.

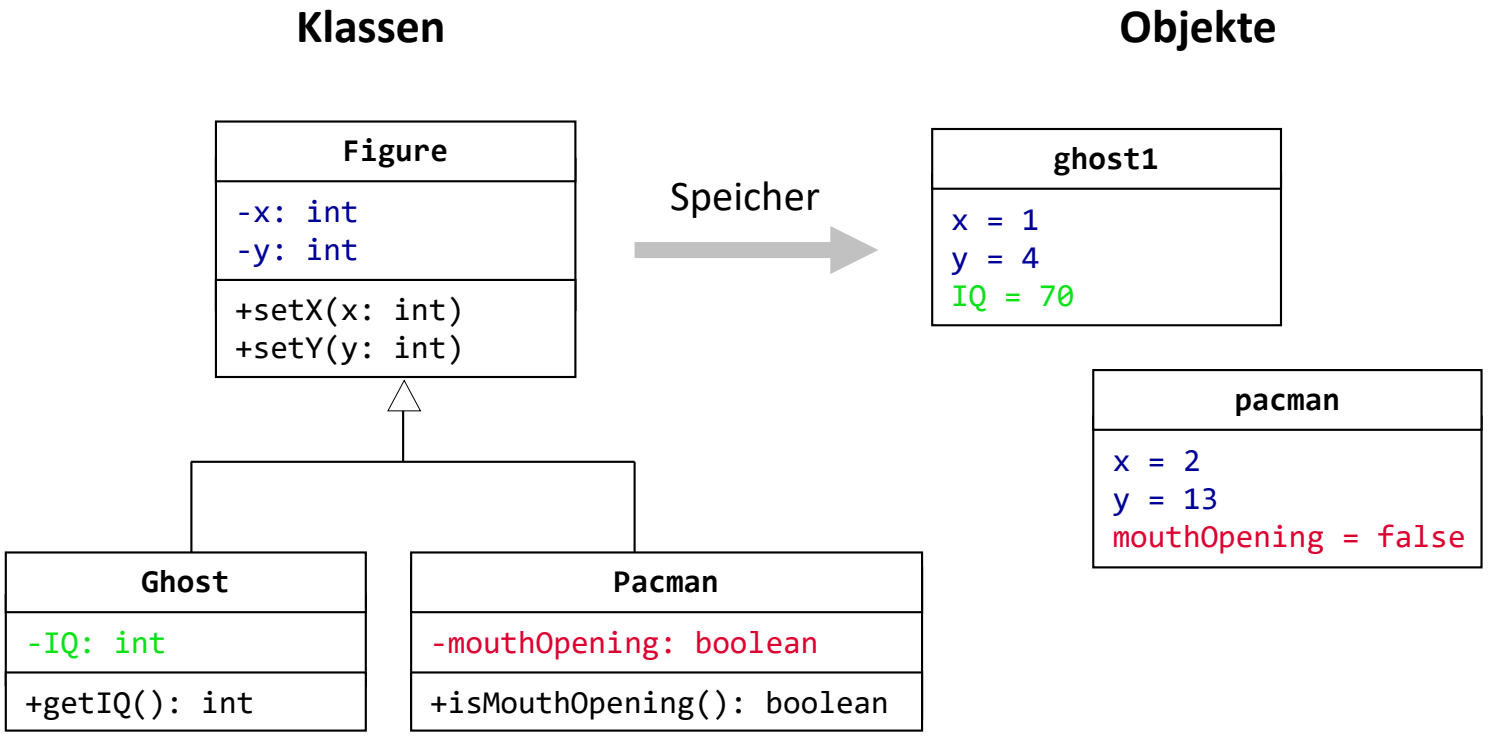


Generalisierung und Spezialisierung – Begriffe

- Prozess der Klassenbildung ist ein Abstraktionsvorgang:
  - Spezialisierung: Aus bestehenden Klassen können spezialisierte Unterklassen (abgeleitete Klassen) gebildet werden.
  - Generalisierung: Gemeinsamkeiten bestehender Klassen können in gemeinsame Oberklassen (Basisklasse) verlagert werden.
  - Ergebnis: Klassenhierarchien aus Ober- und Unterklassen.
- Oberklassen: Allgemeiner und abstrakter als Unterklassen.
- Unterklassen: Spezieller und konkreter als Oberklassen.
- Der Sprachmechanismus, der dieses Konzept unterstützt, wird Vererbung genannt.
- Wichtig: Eine Unterklasse ist ein Untertyp von Oberklasse. Damit kann die Unterklasse überall dort verwendet werden, wo die Oberklasse erwartet wird.

Vererbung
Begriffe

- Eine Unterklasse ist ein Stellvertreter für die Oberklasse.
- Sie kann Methoden der Oberklasse überschreiben (umdefinieren).

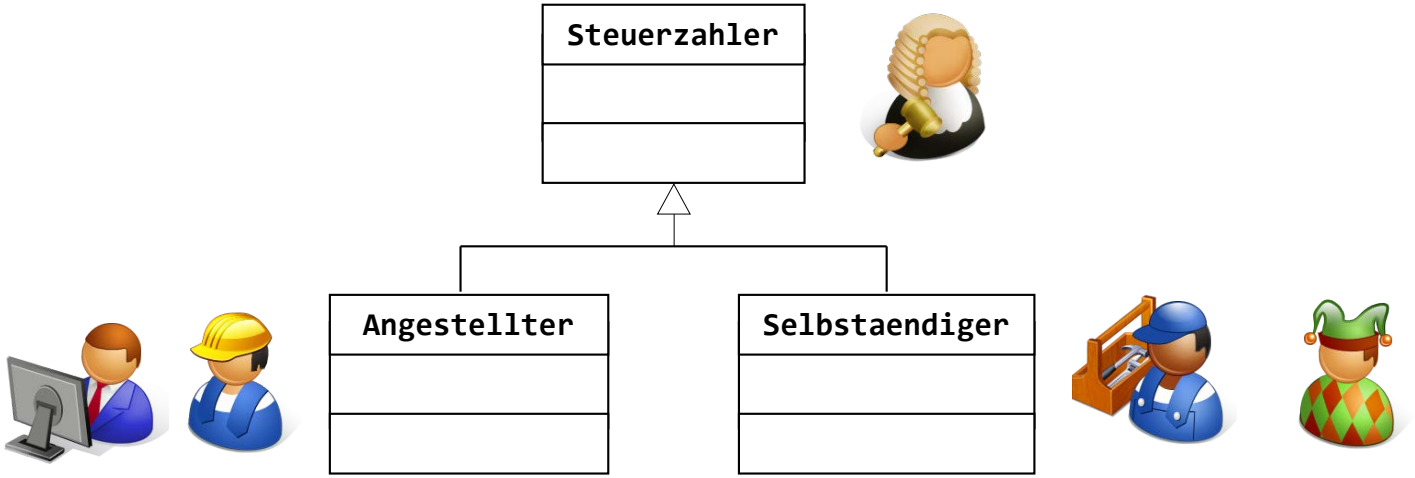


Allgemeiner Fall: Vererbung der Spezifikation

- Eine Unterklasse erbt grundsätzlich die Spezifikation ihrer Oberklasse.
- Die Unterklasse übernimmt damit alle Verpflichtungen und Zusicherungen der Oberklasse.
- Häufiger wird auch der Begriff **Vererbung von Schnittstellen** benutzt.
- Vererbung der Spezifikation: Eine Unterklasse übernimmt die Verpflichtungen (Vor- und Nachbedingungen), die sich aus der Spezifikation der Oberklasse ergeben.
- Vererbung ist mehr als die einfache Syntax zur Implementierung einer Schnittstelle.
- **Prinzip der Ersetzbarkeit**: Wenn die Klasse B eine abgeleitete Klasse der Klasse A ist, dann können in einem Programm alle Objekte der Klasse A durch Objekte der Klasse B ersetzt worden sein, und es gelten trotzdem weiterhin alle zugesicherten Eigenschaften der Klasse A.
- Java unterstützt dieses Konzept durch eigene Sprachmittel, C++ nicht → kommt gleich genauer.

Spezieller Fall: Vererbung der Implementierung

- Anwendungsbeispiel: Es gibt eine Basisklasse, die nicht vollständig durch spezialisierte abgeleitete Klassen abgedeckt wird.
- Es gibt also Objekte der Basisklasse.
- Beispiel (in der Basisklasse sind z.B. Beamte und Rentner):



- Syntaxbeispiel:

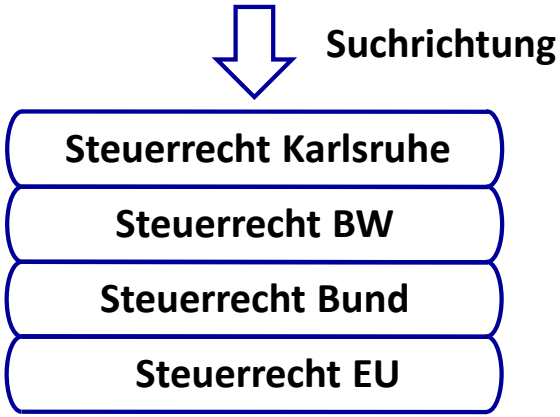
```
public class Angestellter extends Steuerzahler {
}
```

Beispiel: Erben gesetzlicher Regelungen (aus „Praxisbuch Objektorientierung“)

- Gesetzliche Regelungen werden auf verschiedenen Ebenen vorgenommen. Für eine in Karlsruhe lebende Person gilt:
  - Die europäische Union legt rechtliche Rahmenbedingungen fest.
  - Die Bundesrepublik Deutschland hat gesetzliche Regelungen für das Steuerrecht.
  - Das Land Baden-Württemberg hat wiederum eigene spezielle Regelungen.
  - Schließlich legt die Stadt Karlsruhe noch eigene Regelungen fest, zum Beispiel den so genannten Hebesatz für die Gewerbesteuer.
- „Vererbung der Regelungen“
  - Die Karlsruher-Regelung erbt von der des Landes Baden-Württemberg.
  - Diese wiederum erben die Regeln des Bundes.
  - Der Bund muss die Regeln der EU akzeptieren.

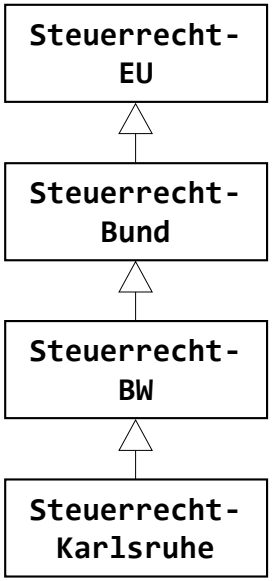
- Effekte der Vererbung der Umsetzung:
  - Beispiel: Der allgemeine Einkommenssteuersatz wird für Karlsruhe direkt aus der Regelung des Bundes übernommen.
  - Eine Änderung des Einkommensteuersatzes bundesweit führt auch zu einer Änderung in Karlsruhe.
  - In einem bestimmten Rahmen können eigene Umsetzungen in den speziellen Fällen erfolgen. Beispiel: Jede Kommune hat eigene Gewerbesteuerumsetzung.
- Die Regelungen sind hierarchisch organisiert, wobei die weiter oben liegenden Regeln jeweils weiter unten liegende überschreiben.

- Suche einer passenden Regelung anhand des Beispiels:



  - Suche eines Gesetzes durch Suchen in den Büchern „von oben nach unten“ im Stapel.
  - Vorteil:
    - Karlsruhe muss nicht den kompletten Gesetzestext der EU beinhalten.
    - Eine Änderung innerhalb der EU wird automatisch übernommen.
    - Nur kleinere Anpassungen werden lokal festgelegt → Karlsruher Recht muss mit EU-Recht übereinstimmen.

- Klassenhierarchie der Regelungen:



- Wie wird eine Klassenhierarchie implementiert?

- Beispiel:
  - **`Figure`** als allgemeine Basisklasse für eine Figur
  - **`Ghost`** als spezielle Figur

```
public class Figure {
  private int xPos;
  private int yPos;

  public Figure(int xPos, int yPos) { /* ... */ }
  public void move(int xPos, int yPos) { /* ... */ }
  //...
}
```

```
public class Ghost extends Figure {
  private boolean dangerous;

  public Ghost(int xPos, int yPos, boolean dangerous) { /* ... */ }
  public boolean isDangerous() { /* ... */ }
  public void move(int xPos, int yPos) { /* ... */ }
  //...
}
```

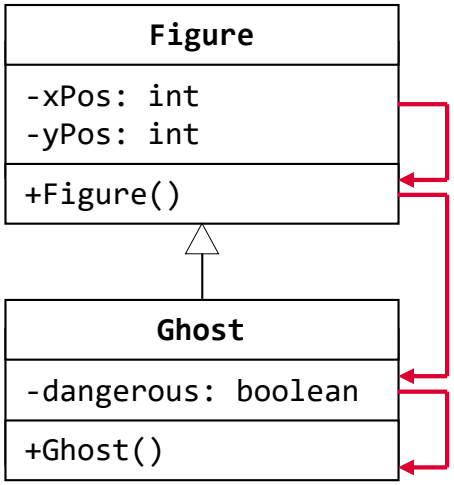
Vererbung

Vererbung der Implementierung – Konstruktorenreihenfolge

- Frage: Wie funktioniert der Konstruktoraufruf in **`Figure`** jetzt, wenn ein **`Ghost`**-Objekt erzeugt wird?
- Antwort: Der Konstruktor von **`Ghost`** kann Argumente an den Konstruktor der Basisklasse **`Figure`** weiterleiten.
- Ein Konstruktor kann auch seine eigenen Attribute initialisieren.

  Vorgehensweise:
  - Aufruf des Basisklassenkonstruktors.
  - Initialisierung der eigenen Attribute.

- Klassenelemente werden von „oben“ nach „unten“ initialisiert:
  - Erst wird der Konstruktor der Basisklasse aufgerufen.
  - Dann werden die Attribute einer Klasse in der Reihenfolge ihrer Deklaration initialisiert.
  - Abschließend wird der Konstruktor der Klasse selbst aufgerufen.
- Initialisierungsreihenfolge im Ghost-Beispiel:



Initialisierungsreihenfolge

1. Attribute der Basisklasse in der Reihenfolge ihrer Deklaration
2. Konstruktor der Basisklasse
3. Attribute der abgeleiteten Klasse in der Reihenfolge ihrer Deklaration
4. Konstruktor der abgeleiteten Klasse

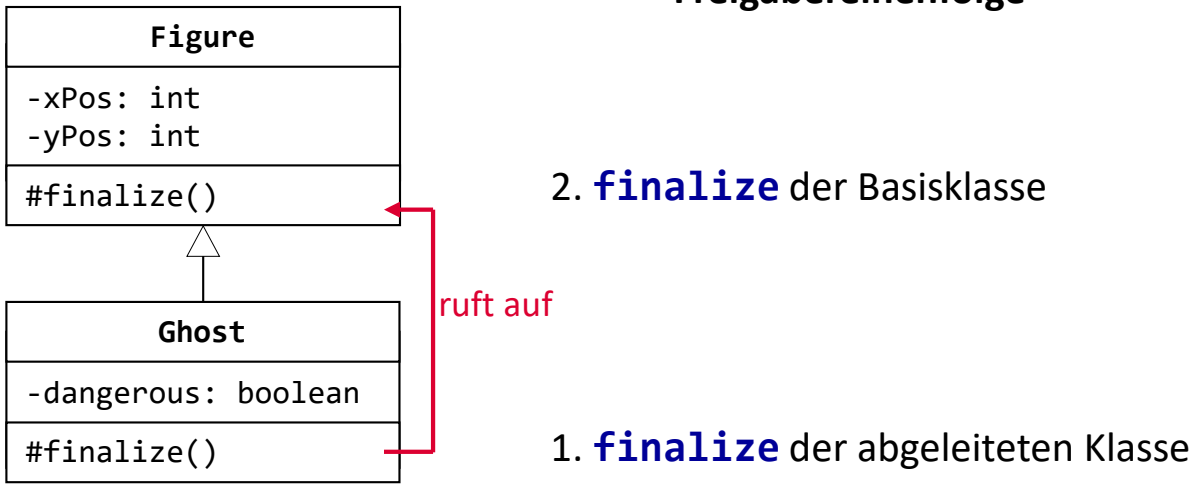
- Beispiel: **Ghost**

```
public Ghost(int xPos, int yPos, boolean dangerous) {
      super(xPos, yPos);
      this.dangerous = dangerous;
}
```

Konstruktor Basisklasse

Attribute

- Zur Erinnerung: Vor der Freigabe wird die Methode **`finalize`** aufgerufen.
  - Zuerst wird **`finalize`** der abgeleiteten Klasse aufgerufen.
  - Diese Methode ruft **`finalize`** der Basisklasse auf und führt dann eigene Aufräumarbeiten durch.
- Aufrufreihenfolge der **`finalize`**-Methoden im Ghost-Beispiel (sofern dort **`finalize`** implementiert wurde):



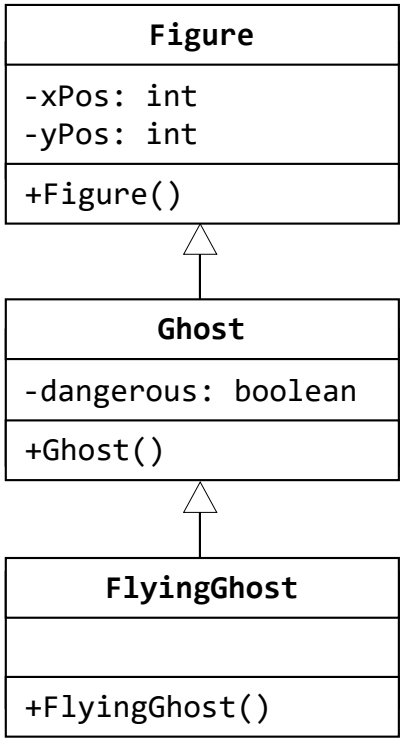
- Soll verhindert werden, dass von einer bestimmten Klasse geerbt wird, dann kann sie als **`final`** deklariert werden. Beispiel:

```
public final class Game {
  private Cell[][] cells;

  public Game() { /* ... */ }

  public void run() { /* ... */ }
  /* ... */
}
```

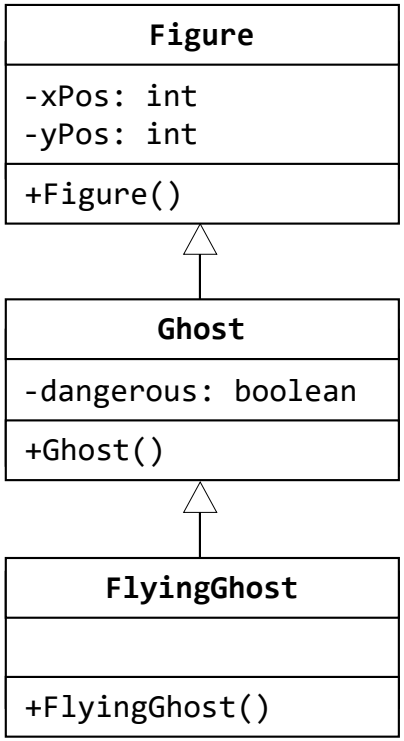
Die Klasse **`String`** ist beispielsweise als **`final`** deklariert.

Bildung komplexer Klassenhierarchien

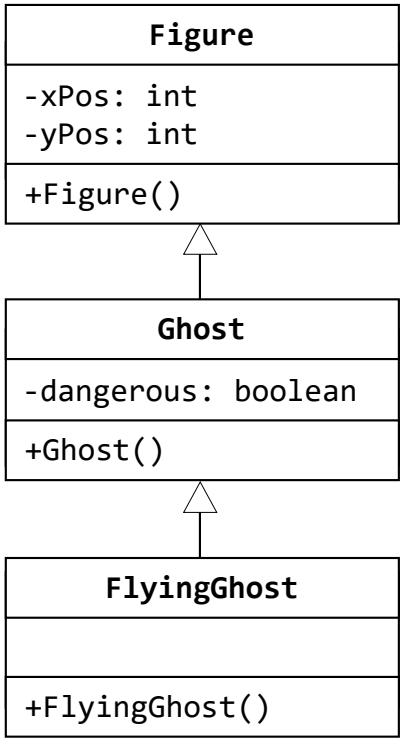
Jede abgeleitete Klasse kann die Basisklasse einer anderen Klasse sein.




```
public class Figure {
  private int xPos;
  private int yPos;

  public Figure() { /* ... */ }
  public void move(int xPos, int yPos) {
    /* ... */
  }
  public void print() { /* ... */ }
  /* ... */
}
```

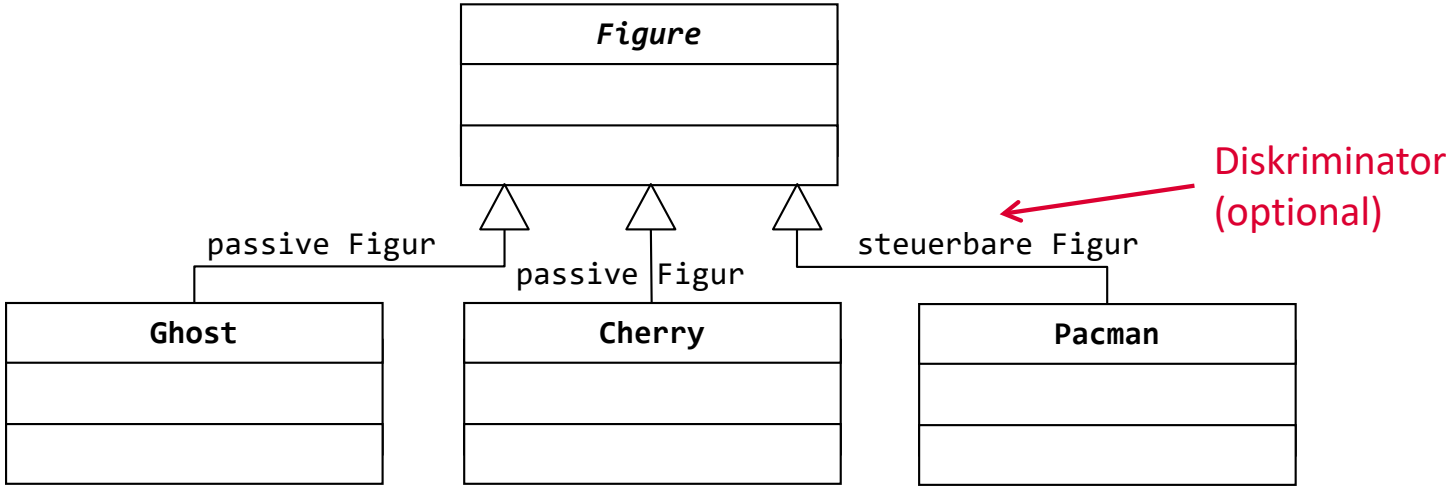



```
public class Ghost extends Figure {
  private boolean dangerous;

  public void move(int xPos, int yPos) {
    /* ... */ }
  public void print() { /* ... */ }
  /* ... */
}
```

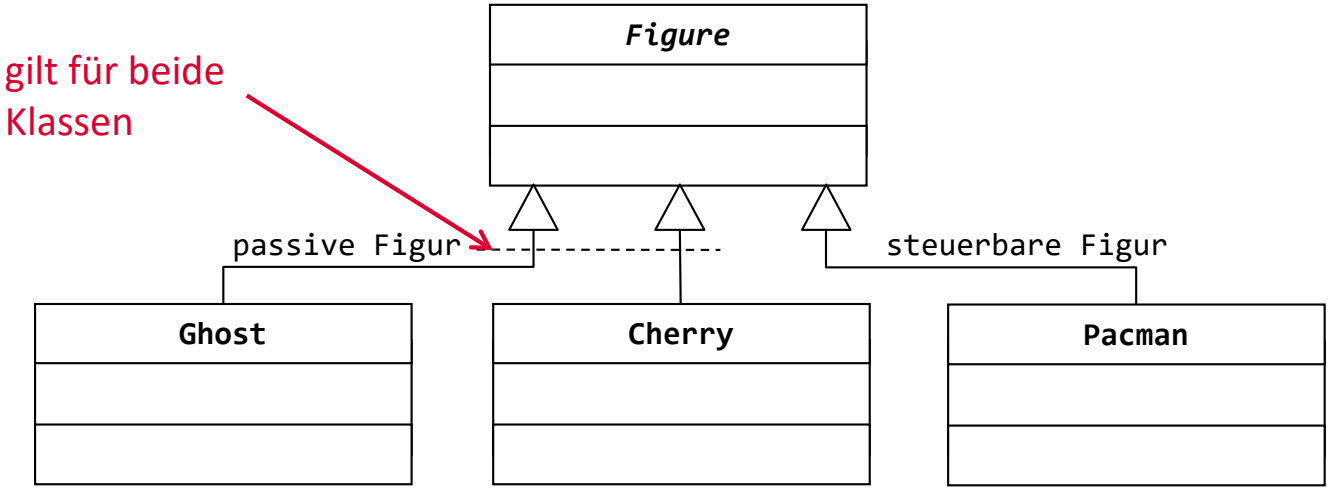



```
public class FlyingGhost extends Ghost {

  public void print() { /* ... */ }
  /* ... */
}
```

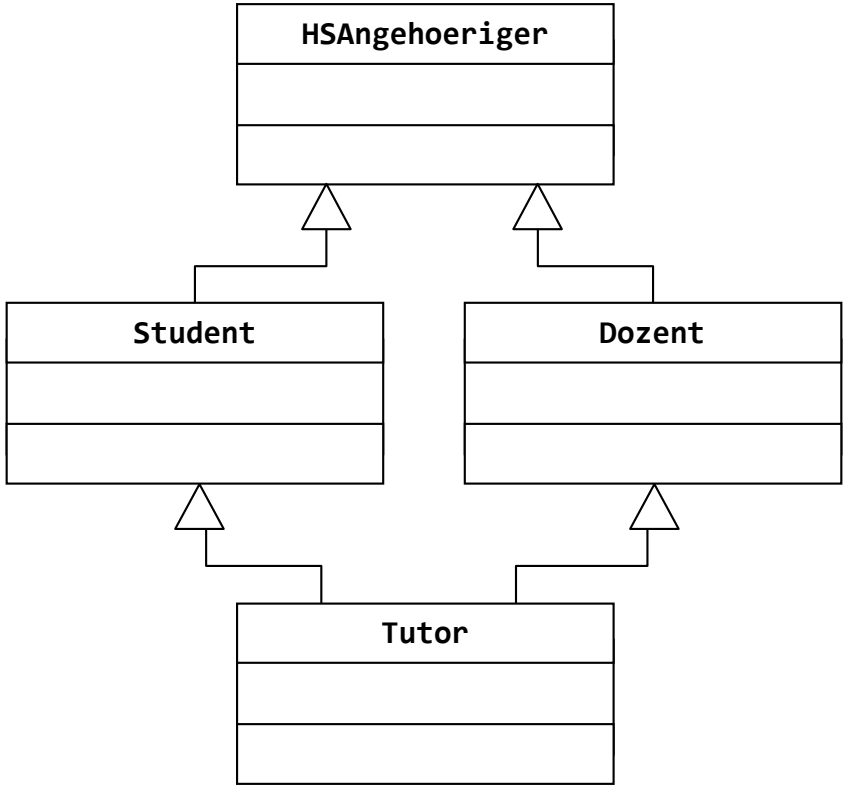
- Durch den Diskriminator können
  - Gruppen zusammengehöriger Klassen erkannt werden
  - weitere Eigenschaften einer Vererbung gegeben werden.



- Der Diskriminator kann bei Gleichheit auch so notiert werden:



- Mehrfachvererbung einer Implementierung: Eine Klasse hat mehr als eine direkte Basisklasse (in Java nicht möglich).



Vererbung

Vererbung der Implementierung – Rechte bei Attribut- oder Methodenzugriffen

- Neues Zugriffsrecht: **`protected`** für Vererbung → räumt einer abgeleiteten Klasse mehr Rechte als einer anderen Klasse ein.
- Eine abgeleitete Klasse kann
  - alle **`public`**- oder **`protected`**-Methoden aller seiner Basisklassen aufrufen.
  - alle **`public`**- oder **`protected`**-Attribute aller seiner Basisklassen lesen und beschreiben.
  - nicht auf die privaten Methoden oder privaten Attribute der Basisklassen zugreifen.
- Wird eine abgeleitete Klasse in einem Programm verwendet, so kann nur auf die **`public`**-Methoden oder Attribute dieser Klasse sowie deren Oberklassen zugegriffen werden.

  Beispiel:

```
Ghost ghost = new Ghost(3, 2, true);
ghost.move(3, 1);                          // OK
boolean dangerous = ghost.dangerous;       // Fehler, privat
```

Vererbung

Vererbung der Implementierung – Rechte bei Attribut- oder Methodenzugriffen

- Szenario:

hska.iwii.my

```
public class A {
 public    int i1;
 protected int i2;
 private   int i3;
 int       i4;
   //code
}


class B extends A {
   //code
}

class C {
   //code
}
```

hska.iwii.p1

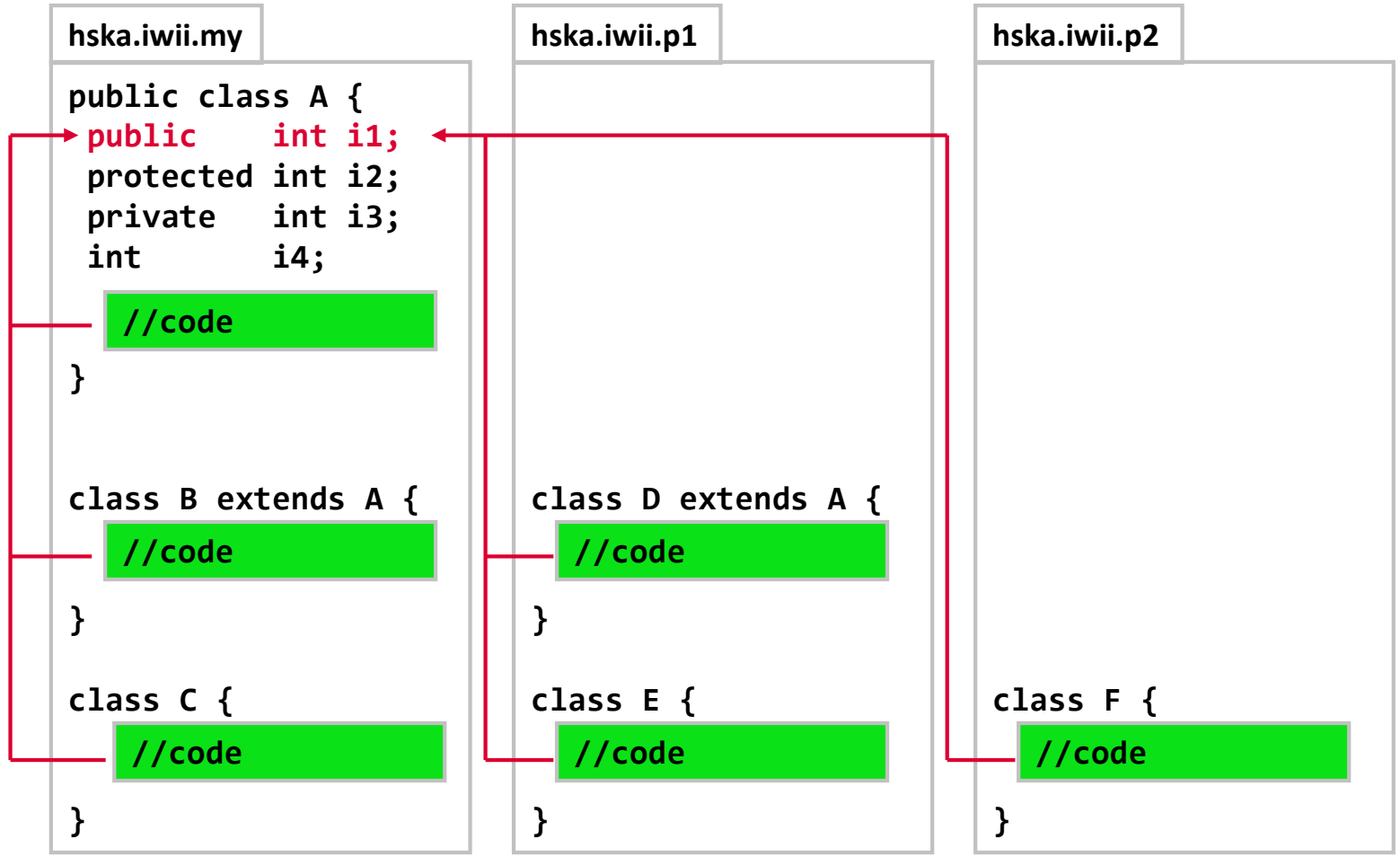
```
class D extends A {
   //code
}

class E {
   //code
}
```

hska.iwii.p2

```
class F {
   //code
}
```

Vererbung

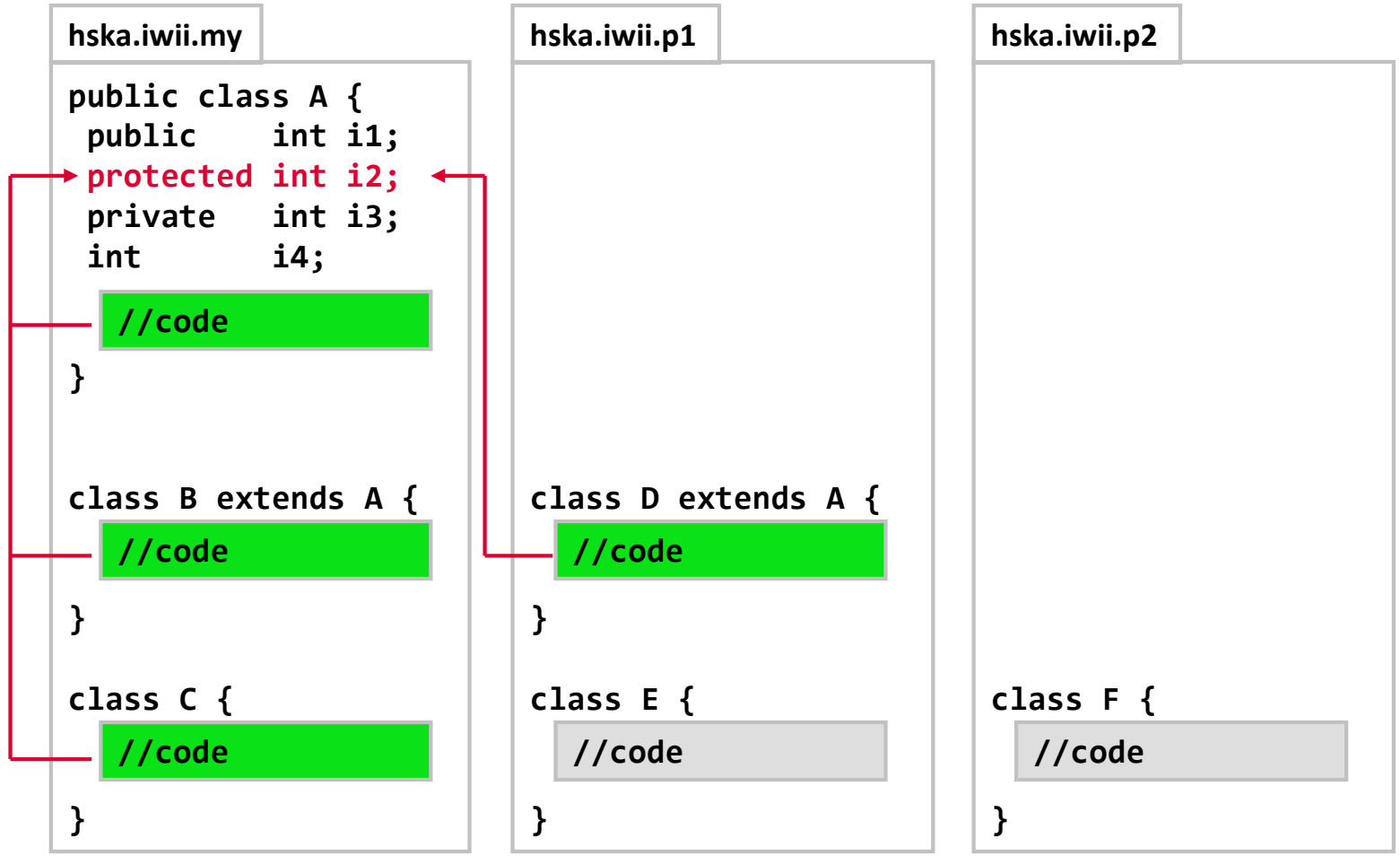
Vererbung der Implementierung – Rechte bei Attribut- oder Methodenzugriffen



`public int i1` ist in allen Klassen und Paketen sichtbar.

Vererbung

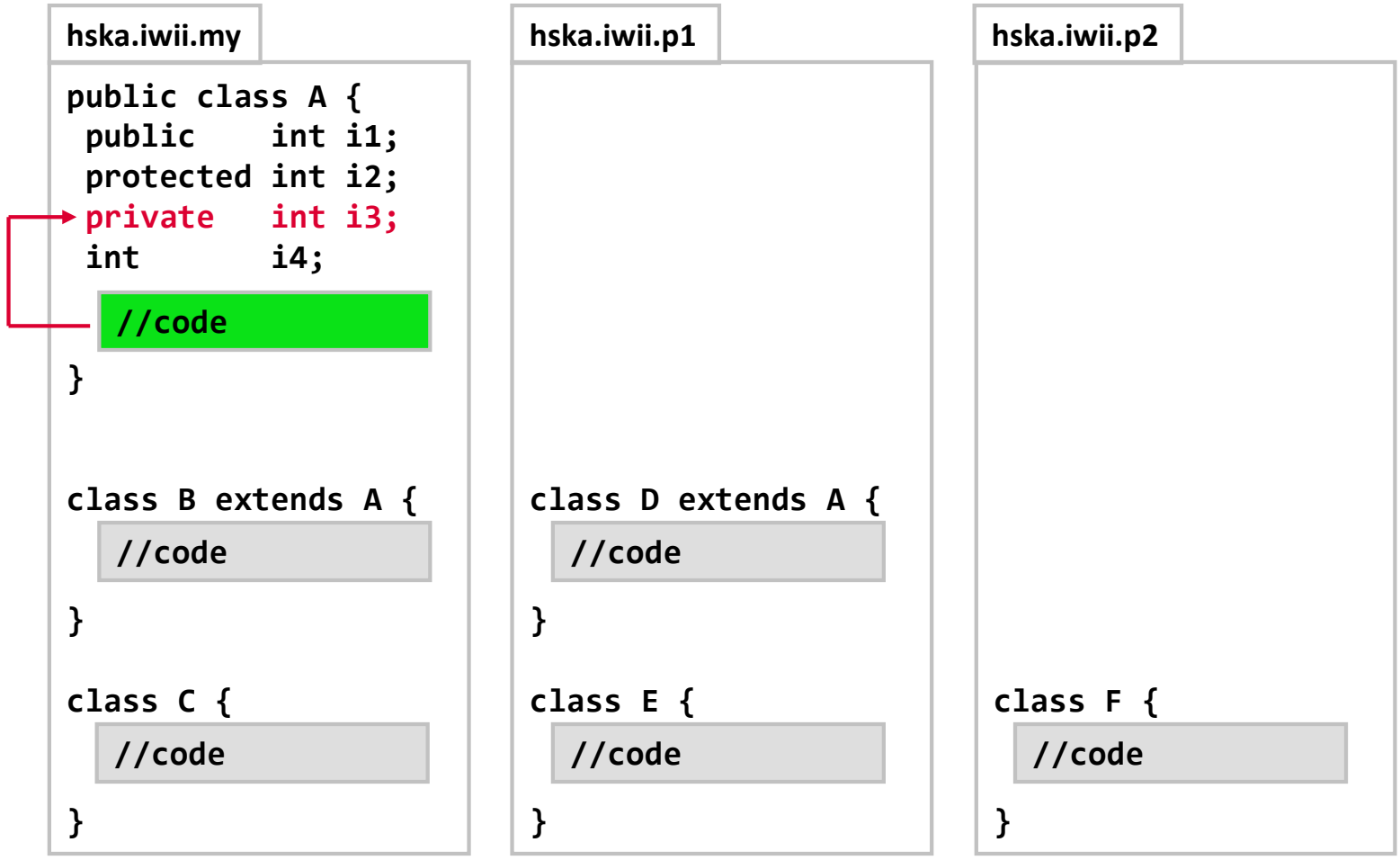
Vererbung der Implementierung – Rechte bei Attribut- oder Methodenzugriffen



`protected int i2` ist in Unterklassen und im eigenen Paket sichtbar.

Vererbung

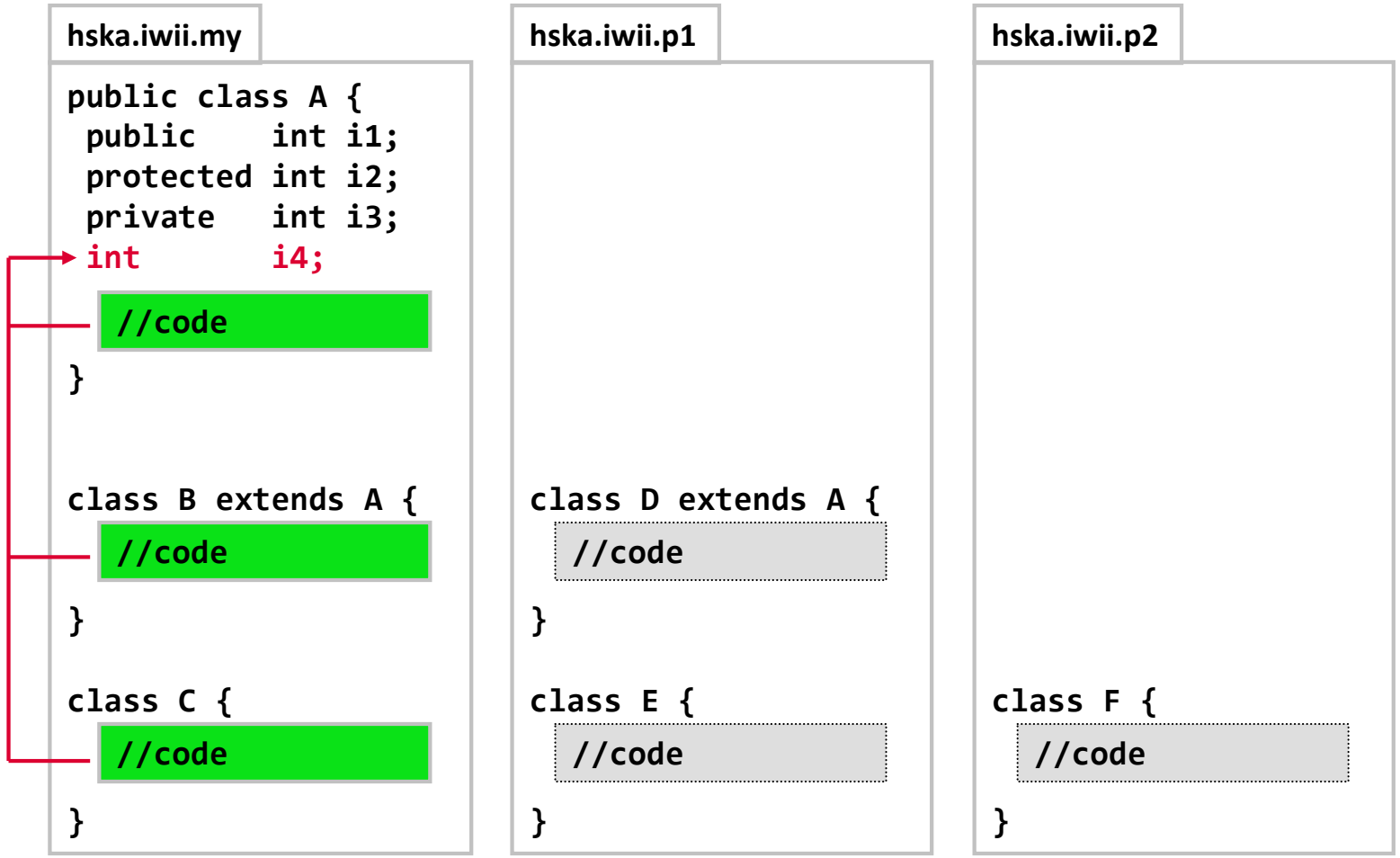
Vererbung der Implementierung – Rechte bei Attribut- oder Methodenzugriffen



`private int i3` ist nur in der eigenen Klasse sichtbar.

Vererbung

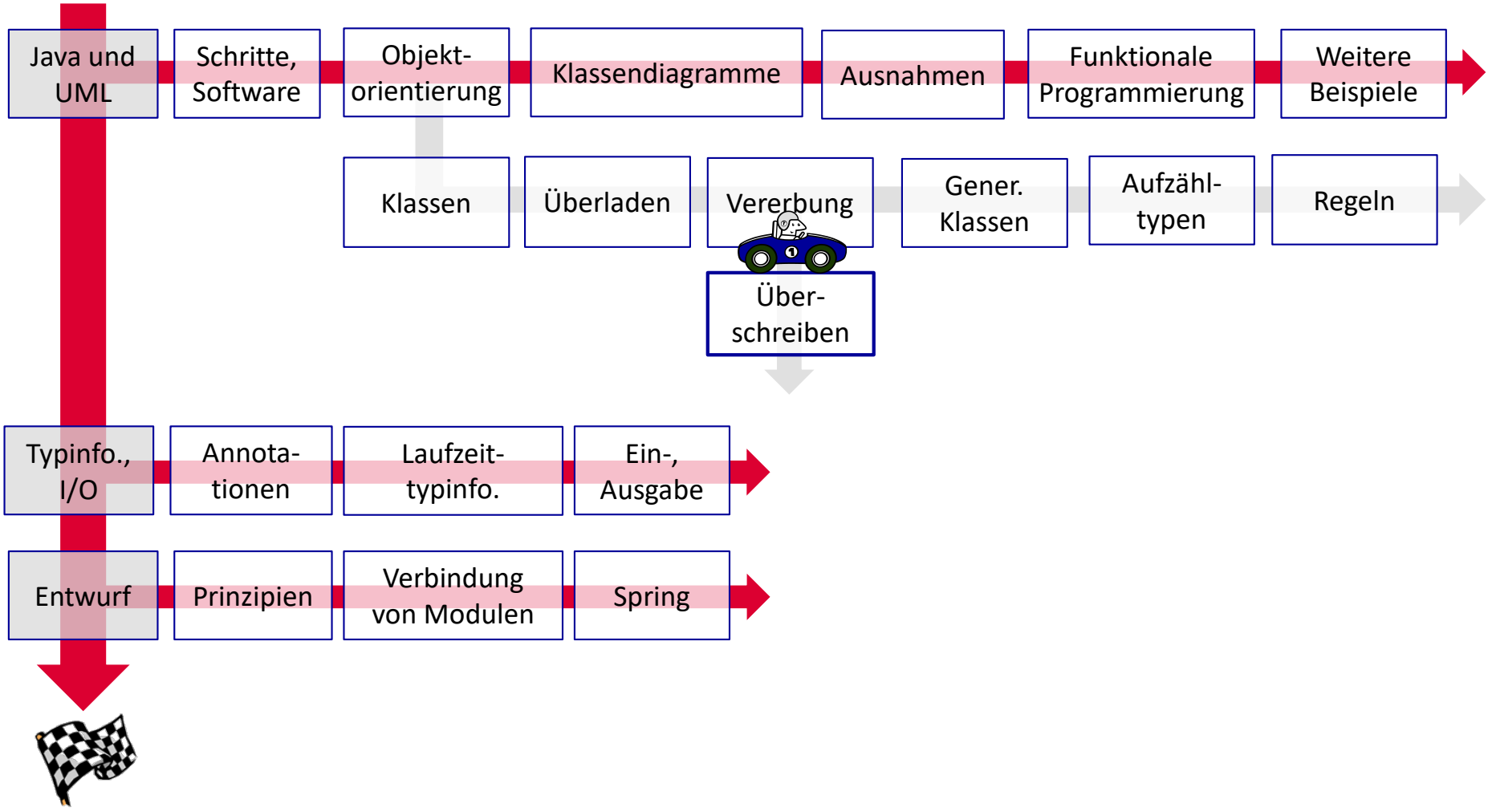
Vererbung der Implementierung – Rechte bei Attribut- oder Methodenzugriffen



- **`int i4`** ist im eigenen Package sichtbar.

Überschreiben von Methoden

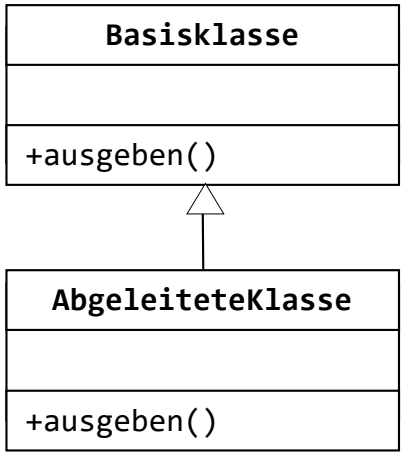
Übersicht



Überschreiben von Methoden

Grundlagen

- Idee: Eine abgeleitete Klasse besitzt eine Methode mit demselben Namen und derselben Signatur wie die Basisklasse.



- Ein häufiges Problem beim Prinzip der Ersetzbarkeit:
  - Eine Basisklassenreferenz verweist auf ein Objekt einer abgeleiteten Klasse.
  - Wie kann festgestellt werden, welcher Klasse das Objekt angehört → wichtig für den Aufruf der korrekten Methode?

Überschreiben von Methoden

Grundlagen

Definition: Überschreiben (von Methoden)

- Wenn eine abgeleitete Klasse eine Methode implementiert, für die es bereits in einer Basisklasse eine Methode gibt, so überschreibt die abgeleitete Klasse die Methode der Basisklasse.
- Wird die Operation auf einem Exemplar der abgeleiteten Klasse aufgerufen, so wird die überschriebene Implementierung der Methode aufgerufen.
- Das ist unabhängig davon, welchen Typ die Referenz hat, über die das Objekt angesprochen wird.
- Entscheidend ist der Typ des Objekts selbst, nicht der Typ der Variablen.

Überschreiben von Methoden

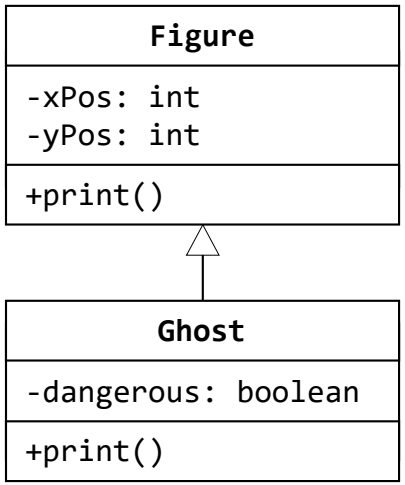
Grundlagen

- Aufruf der neuen Methode **`print`**.

```
public class Figure {
  public void print() {
    System.out.println("Figure");
  }
}
```

```
public class Ghost extends Figure {
  private boolean dangerous;
  @Override
  public void print() {
    System.out.println("Ghost");
  }
}
```

```
// …
private int test() {
  Ghost ghost = new Ghost(3, 2, true);
  Figure figure = ghost;
  ghost.print();
  figure.print();
  // ...
```



Ausgabe:

```
Ghost
Ghost
```

Verhalten

- Aufruf der Methode der abgeleiteten Klasse (auch, wenn die Methode über eine Basisklassenreferenz aufgerufen wird).
- Vorteil: Code, der die Methoden aufruft, bleibt unverändert, selbst wenn Klassen hinzukommen oder sich Klassen ändern.
- Verhalten
  - Alle Methoden, die nicht als **`final`** deklariert werden, können überschrieben werden.
  - Signatur und Name der Methode müssen in Basisklasse und abgeleiteter Klasse identisch sein.
- Überschreibende Methoden sollten mit der Annotation **`@Override`** versehen werden. Dann kann der Compiler prüfen, ob diese Methode wirklich eine andere überschreibt:

```
public class Ghost extends Figure {
  @Override
  public void print() {
    System.out.println("Ghost");
  }
}
```

Überschreiben von Methoden

Verhalten

- Beispiel für eine Methode, die nicht überschrieben werden darf:

```
public class Figure {
  public final void print() {
    System.out.println("Figure");
  }
}
```

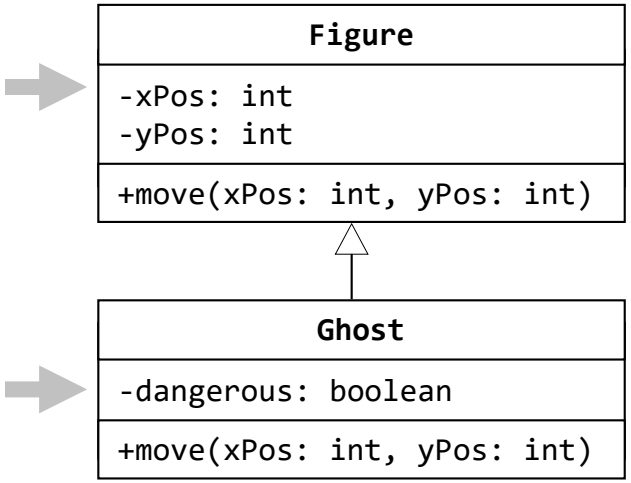
Manueller Einfluss auf das Überschreiben

- Umgehung der automatischen Methodenauswahl, expliziter Aufruf einer Methode der Basisklasse: **`super.methode(Parameter);`**

  Beispiel:

```
public class Figure {
  // ...
  public void move(int xPos, int yPos) {
    // ...
  }
}
```

```
public class Ghost extends Figure {
  // ...
  @Override
  public void move(int xPos, int yPos) {
    super.move(xPos, yPos);
    // z.B. Neuzeichnen
  }
}
```



Konstruktoren und Überschreiben

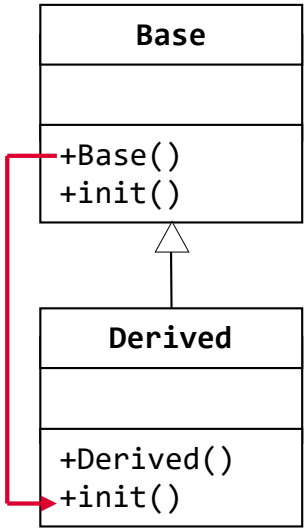
- Der Konstruktor der Basisklasse wird vor dem Konstruktor der abgeleiteten Klasse ausgeführt.
- Aufruf einer polymorphen Methode im Konstruktor der Basisklasse:
  - Im Gegensatz zu C++: Aufruf der überschriebenen Methode
  - Achtung: Die abgeleitete Klasse ist noch gar nicht initialisiert!!
  - Konsequenz: Konstruktoren sollten nur private oder finale Methoden aufrufen.

```
public class Base {
  private Vector v;
  public Base() {
    v = new Vector(3);
    init();
  }
  public void init() {
    v.setValue(1, 42);
  }
}
```

nein

ja

```
public class Derived
              extends Base {
  private MathVector mv;
  public Derived() {
    mv = new MathVector(3,true);
    init();
  }
  @Override
  public void init() {
    mv.setValue(0, 66);
  }
}
```



Hinweise zum Umgang mit dem Überschreiben

Ermittlung der „richtigen“ Methode:

- Zur Übersetzungszeit (early binding, statische Bindung):
  - Der Aufruf kann vom Compiler direkt in Bytecode umgesetzt werden, da die Methode jederzeit bekannt ist.
  - nur für Methoden möglich, die als **`final`** oder **`private`** deklariert sind
- Zur Laufzeit (late binding, dynamische Bindung):
  - Alle Methoden, die nicht final, nicht privat und nicht statisch sind ➔ Polymorphismus. Das Objekt findet selbst heraus, welche Methode aufgerufen werden soll
  - geringer Mehraufwand beim Methodenaufruf
  - Die überschreibende Methode der abgeleiteten Klasse darf die Zugriffsrechte der Methode der Basisklasse nicht einschränken.

Hinweise zum Design

- „Normale“ Methoden:
  - Alle Methoden einer Klasse, die überschrieben werden dürfen, sollten nicht **`final`** sein.
  - **`final`** sollte nur aus Design- nicht aus Geschwindigkeitsgründen hinzugefügt werden.
- Finale Methoden:
  - Methoden, die nicht überschrieben werden dürfen, sollten als **`final`** deklariert werden.

Überschreiben von Methoden

Regeln

- Wiederholung: **Prinzip der Ersetzbarkeit**
- Wenn die Klasse **`Abgel`** eine abgeleitete Klasse der Klasse **`Basis`** ist, dann können in einem Programm alle Objekte der Klasse **`Basis`** durch Objekte der Klasse **`Abgel`** ersetzt worden sein, und es gelten trotzdem weiterhin alle zugesicherten Eigenschaften der Klasse **`Basis`**.
- Der für die Basisklasse geschlossene Kontrakt mit Bezug auf Vorbedingungen, Nachbedingungen und Invarianten gilt also auch dann weiter, wenn Objekte der Basisklasse durch Objekte der abgeleiteten Klasse ersetzt werden.

- Beim Überschreiben von Methoden gilt also: Abgeleitete Klassen dürfen zwar mehr anbieten, aber nicht mehr verlangen als Exemplare ihrer Basisklassen.

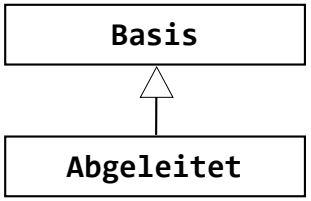
Überschreiben von Methoden

Regeln

Liskovsches Substitutionsprinzip

Formulierung 1993 von Barbara Liskov und Jeannette Wing:

- Basisklasse **`Basis`**
- **`b`** ist ein Objekt von **`Basis`**.
- Abgeleitete Klasse **`Abgeleitet`**,
  die von **`Basis`** erbt
- **`a`** ist ein Objekt von **`Abgeleitet`**.
- Es gilt:
  Sei q(**`b`**) eine beweisbare Eigenschaft von Objekten **`b`** des Typs **`Basis`**. Dann soll q(**`a`**) für Objekte **`a`** des Typs **`Abgeleitet`** wahr sein.
- Beispiele kommen gleich!



Besser verständlich als „ist-ein“-Beziehung.

Überschreiben von Methoden

Regeln

- Konsequenzen für die Vorbedingungen, Nachbedingungen und Invarianten einer abgeleiteten Klasse:
  - Schwächere Vorbedingungen:
    - Eine abgeleitete Klasse kann die Vorbedingungen für eine Operation, die durch die Basisklasse definiert wird, einhalten oder abschwächen. Sie darf die Vorbedingungen aber nicht verschärfen.
    - Falls eine abgeleitete Klasse die Vorbedingungen verschärfen würde, würde damit ohne Absprache mit den Partnern von diesen mehr verlangt als vorher.
    - Erlaubte 👍 und verbotene 👎 Maßnahmen (unvollständig):
      - 👍 Der Wertebereich der Übergabeparameter wird erweitert.
      - 👍 Eine nicht öffentliche Methode wird öffentlich überschrieben.
      - 👎 Der Wertebereich der Übergabeparameter wird verkleinert.
      - 👎 Der Übergabeparameter wird von einem Objekt der Basisklasse auf ein Objekt der abgeleiteten Klasse eingeschränkt.

Überschreiben von Methoden

Regeln

- Stärkere Nachbedingungen:
  - Eine abgeleitete Klasse kann die Nachbedingungen für eine Operation, die durch eine Oberklasse definiert werden, einhalten oder einschränken. Sie darf die Nachbedingungen aber nicht lockern.
  - Falls eine abgeleitete Klasse die Nachbedingungen lockern würde, würde diesen damit wieder ohne Absprache mit den Partnern des Kontrakts mehr geboten als vorher.
  - Erlaubte 👍 und verbotene 👎 Maßnahmen (unvollständig):
    - 👍 Der Wertebereich der Rückgabeergebnisse wird eingeschränkt.
    - 👍 Der Rückgabetyp ist ein Objekt der abgeleiteten Klasse **A** (**A** erbt von **B**), während die Methode der Basisklasse ein Objekt einer Basisklasse **B** zurückgibt.
    - 👎 Der Wertebereich der Rückgabeergebnisse wird erweitert.
    - 👎 Der Rückgabetyp ist ein Objekt der Basisklasse **B**, während die Methode der Basisklasse ein Objekt einer abgeleiteten Klasse **A** (**A** erbt von **B**) zurückgibt.

Überschreiben von Methoden

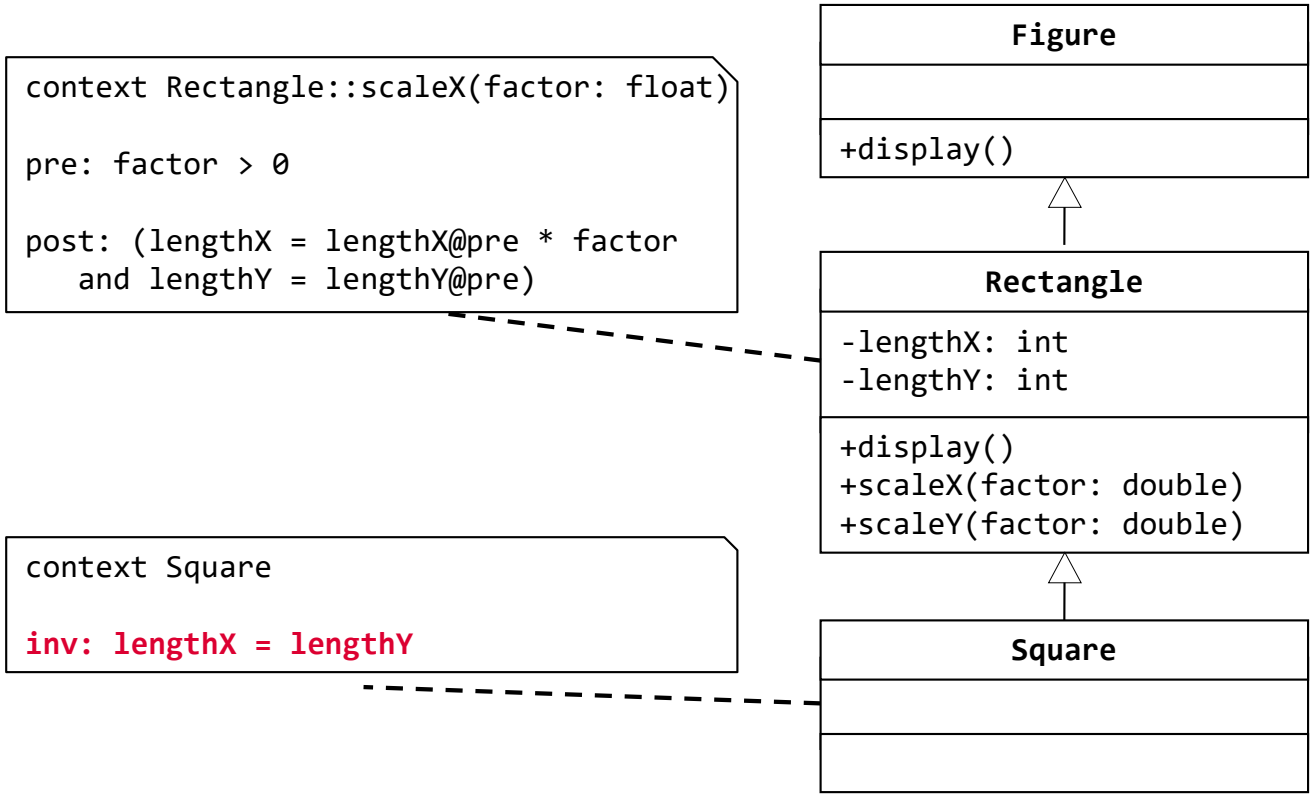
Regeln

- Invarianten:
  - Eine abgeleitete Klasse muss dafür sorgen, dass die für die Basisklasse definierten Invarianten immer gelten. Sie darf die Invarianten verschärfen.
  - Die Partner des Kontrakts müssen sich auf die zugesicherten Invarianten verlassen können.
  - Eine Verhaltensänderung darf eintreten.

Ergebnis: Design by Contract

Überschreiben von Methoden
Regeln

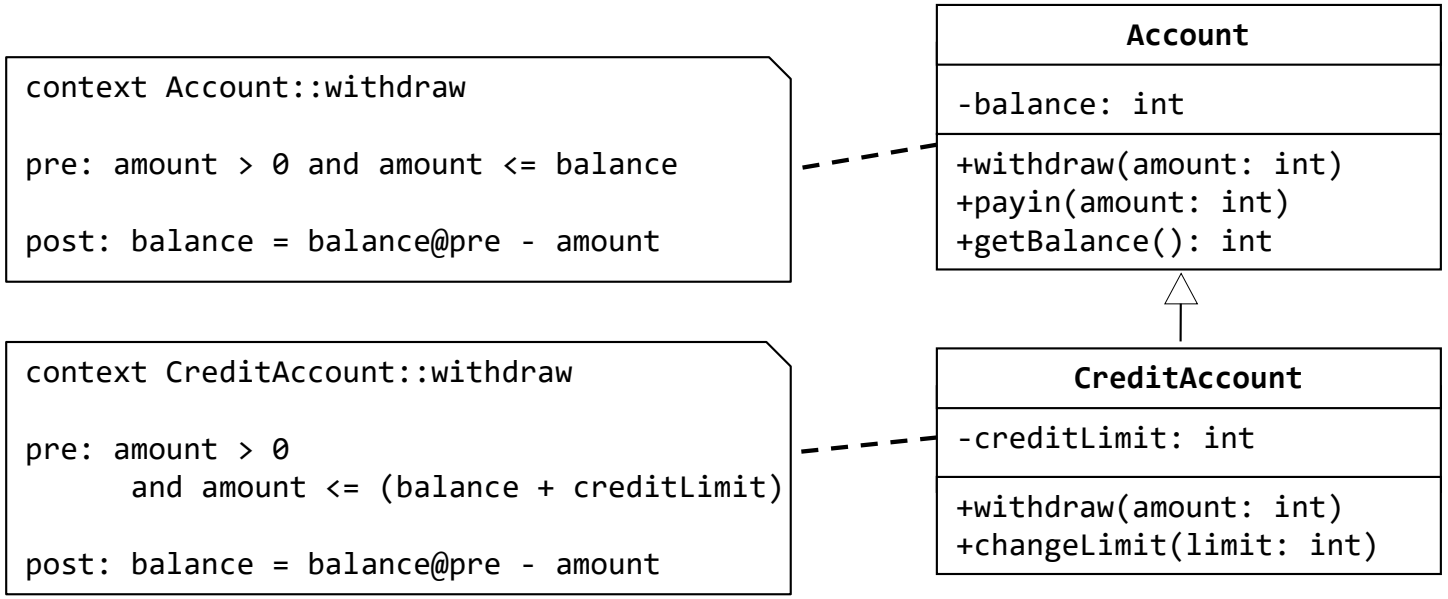
- Beispiel für die Verletzung der Ersetzbarkeit:



Ersetzbarkeit verletzt: Neue Einschränkung der Längen in der Invariante des Quadrates!

Überschreiben von Methoden
Regeln

- Beispiel für die Einhaltung der Ersetzbarkeit:



Nachbedingung wurde nicht gelockert, Vorbedingung gelockert → ok.

Überschreiben von Methoden

Regeln

- Implementierung des Kontobeispiels:

```
public class Account {
  private int balance = 0;

  public int getBalance() {
    return balance;
  }


  public void withdraw(int amount) {
    assert amount > 0
        && amount <= balance;

    balance -= amount;
  }

  public void payin(int amount) {
    balance += amount;
  }
}
```

```
public class CreditAccount extends
                                Account {
  private int creditLimit;

  public CreditAccount(int limit) {
    creditLimit = limit;
  }

  @Override
  public void withdraw(int amount) {
    assert amount > 0
        && amount <= (balance +
           creditLimit);
    balance -= amount;
  }

  public void changeLimit(int limit) {
    creditLimit = limit;
  }
}
```

Überschreiben von Methoden

Regeln

- Verwendung des Kontobeispiels:

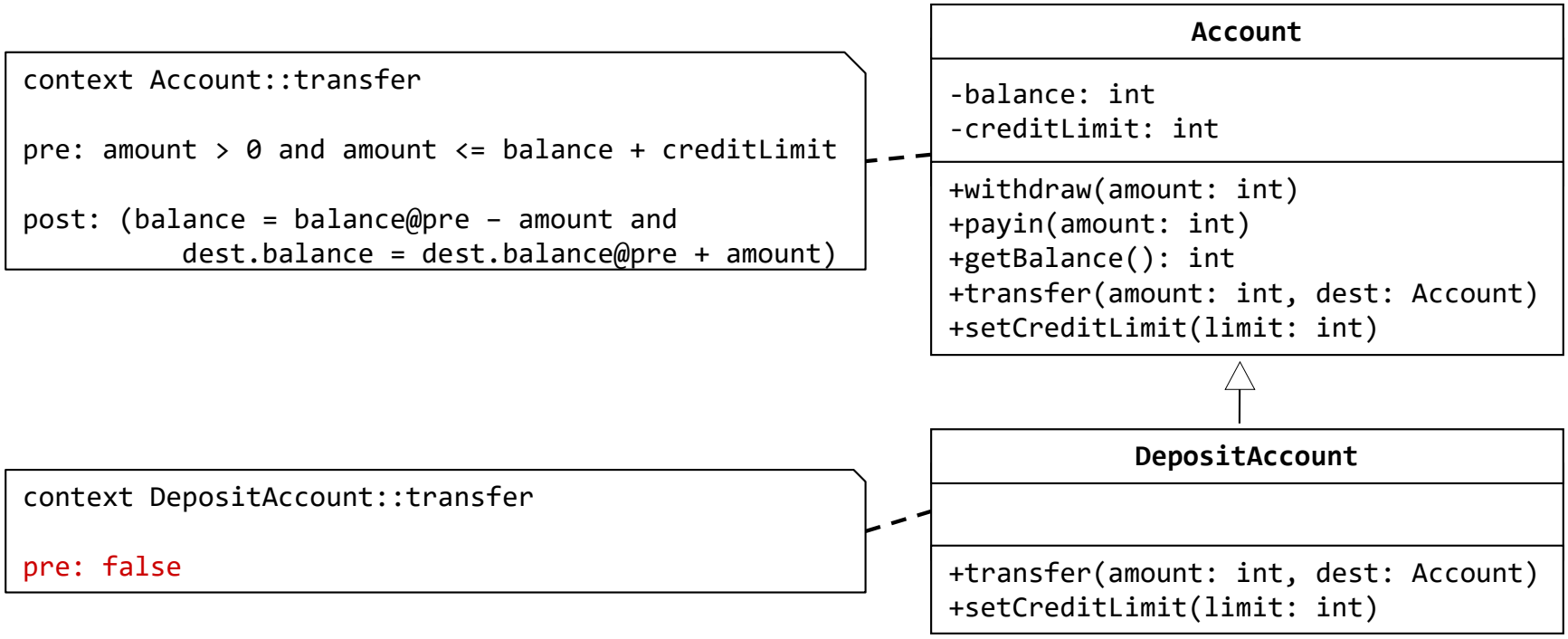
```
CreditAccount account1 = new CreditAccount(10000);
Account       account2 = new Account();

// Abheben
bank.withdraw(account1);
bank.withdraw(account2);
```

```
// Methode zum Abheben eines Festbetrags
// account verweist auf Account oder CreditAccount
public void withdraw(Account accountPtr) {
  if (accountPtr.getBalance() >= 200)
    accountPtr.withdraw(200);
}
```

- Wichtig: In der Methode **`withdraw`** sind Vor- und Nachbedingung sowie Invariante nur anhand der Klasse **`Account`** erkennbar!

Überschreiben von Methoden
Regeln

- Beispiel für die Verletzung der Ersetzbarkeit (Einschränkung der Vorbedingung):



Die Überweisung von einem Sparkonto wird verboten → Einschränkung der Vorbedingung!

Überschreiben von Methoden

Regeln

- Implementierung des Kontobeispiels (unvollständig):

```
public class Account {
  private int balance     = 0;
  private int creditLimit = 0;

  public int getBalance() {
    return balance;
  }
  public void transfer(int amount,
                       Account dest) {
    assert amount > 0
           && amount <= (balance +
              creditLimit));
    balance -= amount;
    dest.payin(amount);
  }

  public void payin(int amount) {
    assert amount > 0;
    balance += amount;
  }
}
```

```
public class DepositAccount extends
                                Account {




  @Override
  public void transfer(int amount,
                       Account dest) {
    assert false;









  }



}
```

Überschreiben von Methoden

Regeln

- Verwendung des Kontobeispiels:

```
DepositAccount account1 = new DepositAccount();
Account        account2 = new Account();

// Überweisen
bank.transfer(account1, account2);
bank.transfer(account2, account1);
```

```
// Methode zum Überweisen eines Festbetrags
public void transfer(Account dest, Account source) {
  if (source.getBalance() + source.getCreditLimit() >= 200)
    source.transfer(200, dest);
}
```

- Wichtig: In der Methode **`transfer`** berücksichtigt die Vorbedingung der Klasse **`Account`**. **`DepositAccount`** schränkt diese aber ein → Methode **`transfer`** kann nicht richtig funktionieren.

Überschreiben von Methoden

Regeln

- Problem: Die Kontraktverletzungen werden erst zur Laufzeit entdeckt → vollständige Test erforderlich.
- Lösung: Spracherweiterungen, die eine Spezifikation der Kontrakte erlauben
- Problem: Prüfungen auf Kontrakteinhaltung müssen manuell in den Code verteilt werden → sehr viel Arbeit.
- Lösung: AOP (Aspect Oriented Programming) → kommt später

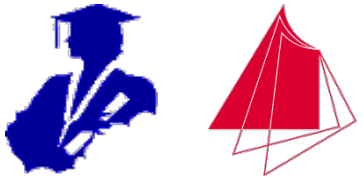
Überschreiben von Methoden

Regeln

- Wie können Kontrakte angegeben werden?
  - in UML mit OCL
  - im Quelltext mit Assertions (auch mit Hilfe von AOP)
  - Es gibt Erweiterungen objektorientierter Sprachen um Constraints, Auswahl:
    jContractor (http://jcontractor.sourceforge.net/),
    C4J (http://c4j.sourceforge.net/),
    COFOJA (http://code.google.com/p/cofoja)
- Wie sieht es bei Schnittstellen oder abstrakten Klassen aus? Dort gibt es keine Implementierung!
  - in UML mit OCL spezifizieren
  - im Quelltext dokumentieren
  - Constraint-Erweiterung verwenden

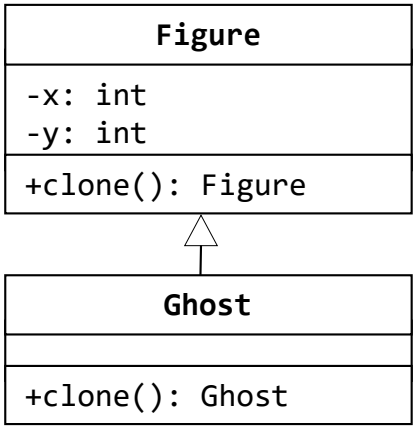
Überschreiben von Methoden

Kovarianz und Kontravarianz



Genauerer Blick auf die Vererbung bei Methodenparametern und Rückgabetypen

- Kovariante Rückgabetypen in der **`clone`**-Methode: Der Rückgabetyp in der abgeleiteten Klasse erbt vom Rückgabetyp der Basisklasse

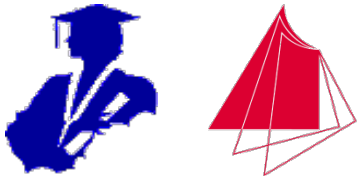


Die Klasse **T2** ist der Klasse **T1** kovariant, wenn alle Objekte von **T2** gleichzeitig Objekte von **T1** sind.
Einfacher gesagt: **T2** muss entweder **T1** oder eine abgeleitete Klasse sein.

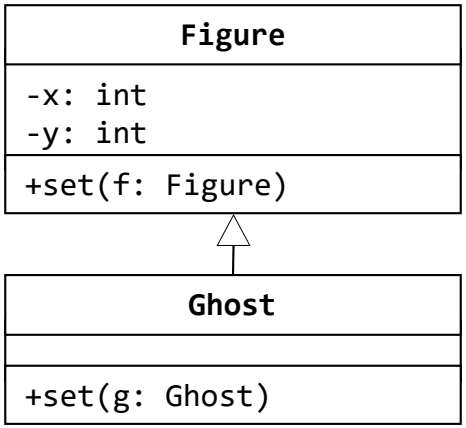
- Einschränkung der Nachbedingung in dieser Form ist erlaubt.
- Java unterstützt seit Version 5 kovariante Rückgabetypen.
- C++ hat kovariante Rückgabetypen im Standard definiert. Allerdings unterstützen nicht alle Compiler dieses Sprachmittel.

Überschreiben von Methoden

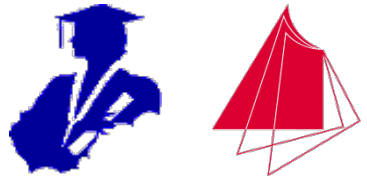
Kovarianz und Kontravarianz



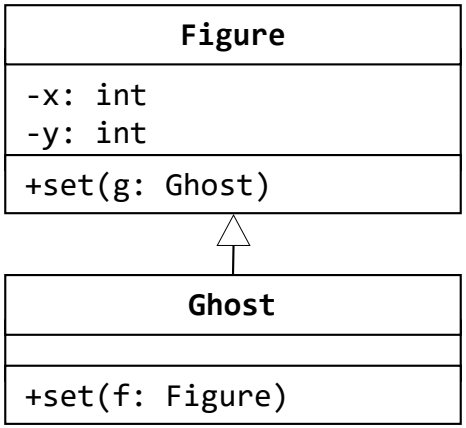
- Kovariante Übergabetypen in der **`set`**-Methode: Der Übergabegabetyp in der abgeleiteten Klasse erbt vom Übergabegabetyp der Basisklasse



  - Die Einschränkung der Vorbedingung ist **nicht** erlaubt.
  - Ausweg: In Java und C++ wird die Methode **`set`** in **`Figure`** gar nicht überschrieben, sie wird überladen!
  - Schlussfolgerung: Die Übergabetypen sind nicht kovariant.



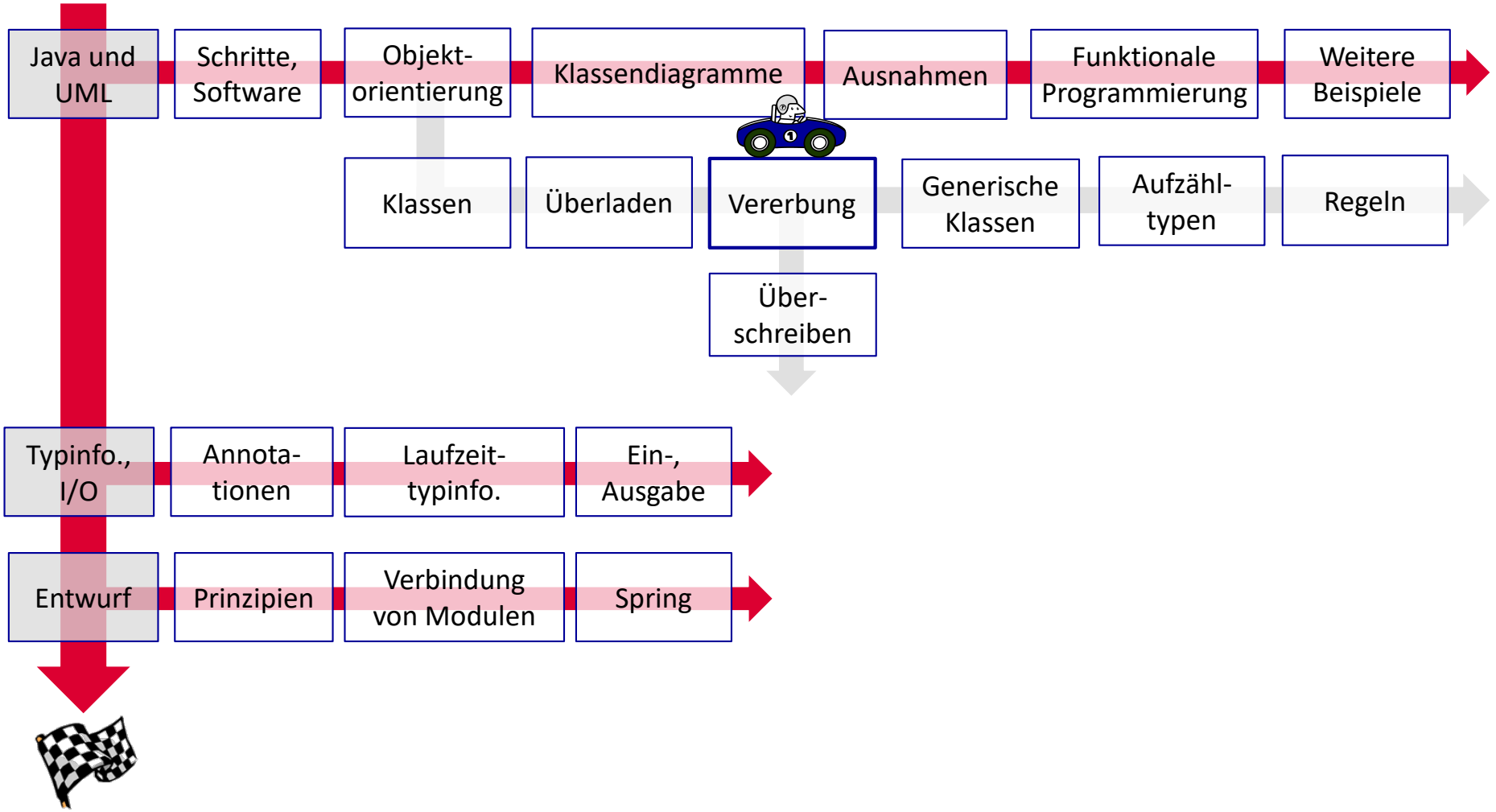
- Kontravariante Übergabetypen in der **`set`**-Methode: Der Übergabegabetyp in der abgeleiteten Klasse ist Basisklasse des Übergabetyps der Basisklasse



Die Klasse **T2** ist der Klasse **T1** kontravariant, wenn alle Objekte von **T1** gleichzeitig Objekte von **T2** sind.
Einfacher gesagt: **T1** muss entweder **T2** oder seine abgeleitete Klasse sein.

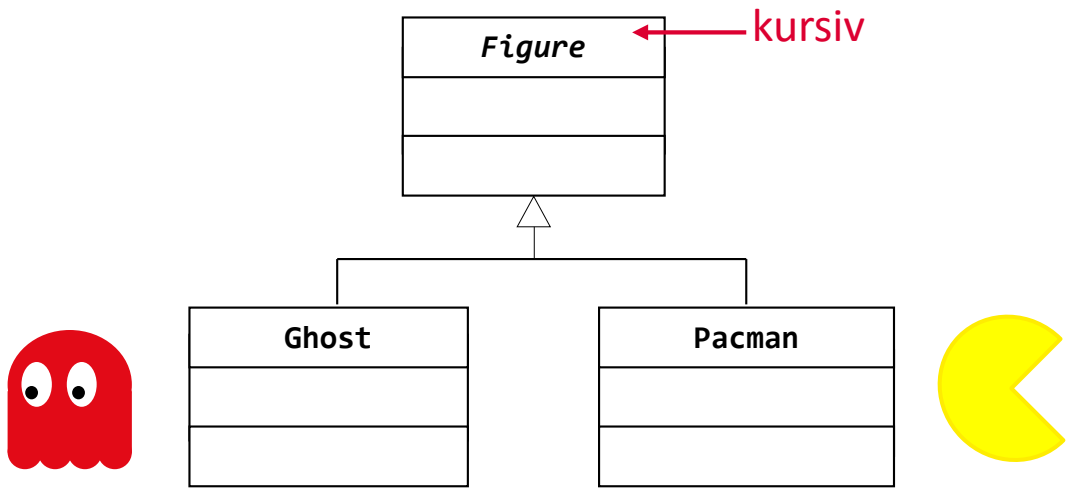
  - Aufweichung der Vorbedingung ist erlaubt.
  - Kontravariante Übergabetypen sind erlaubt.
- Kontravariante Rückgabetypen sind nicht erlaubt.

Vererbung
Übersicht



Spezialfall der Vererbung: Abstrakte Basisklasse

- Idee: Es gibt keine Objekte der Oberklasse. Alle Objekte der Basisklasse müssen durch Unterklassenobjekte repräsentiert werden.
- Beispiel: Ein Konto ist entweder ein Girokonto oder ein Sparkonto.
- Es muss eines von beiden sein, kann aber nicht in beiden Klassen gleichzeitig sein.
- Pacman-Beispiel: Es gibt keine Objekte der Basisklasse **`Figure`**.



Vererbung

Vererbung der Implementierung (abstrakte Basisklasse)

- Klassen werden mit dem Schlüsselwort **`abstract`** als abstrakt markiert, um zu verhindern, dass ein Objekt solcher Klasse erzeugt wird.
- Beispiel: **`Figure`**, von dem keine Objekte erzeugt werden dürfen.

```
public abstract class Figure {
  private int xPos;
  private int yPos;

  public void handleCollisionWith(Figure other) {
    // ...
  }
}
```

```
public class Pacman extends Figure {
  private boolean mouthOpening;

  @Override
  public void handleCollisionWith(Figure other) {
    // ...
  }
  // ...
}
```

Überschreiben bzw.
implementieren der Methode

- Auch Methoden können abstrakt sein:
  - Die Basisklasse kann z.B. keine sinnvolle gemeinsame Implementierung anbieten.
  - Die abgeleiteten Klassen müssen die Methoden dann implementieren.
  - Ein Klasse mit mindestens einer abstrakten Methode ist abstrakt.
- Beispiel: **`Figure`** ohne Implementierung des Zeichnens

```
public abstract class Figure {
  private boolean dead;

  public abstract void paint(Canvas panel);
  // ...
}
```

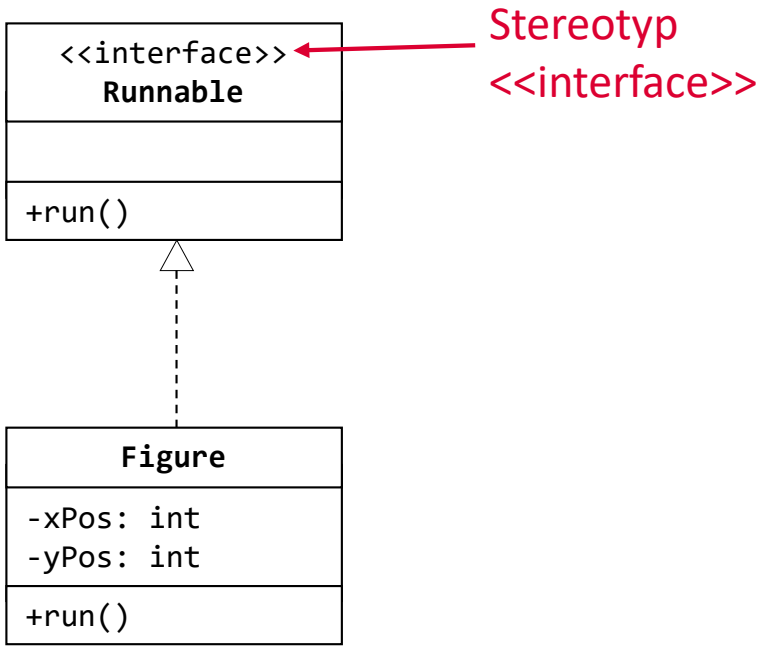
keine Implementierung

Vererbung

Vererbung der Spezifikation (Schnittstellen)

- Für die Vererbung einer Spezifikation gibt es in Java Schnittstellen (**`interface`**).
- Es werden Methodensignaturen vorgegeben, die eine erbende Klasse implementiert.
- Beispiel: Schnittstelle **`Runnable`** aus dem JDK

```
public interface Runnable {
  void run();
}
```



- Alle Methoden sind **`public`** (nicht angegeben).
- Alle Attribute sind **`public`**, **`static`** und **`final`** (nicht angegeben).

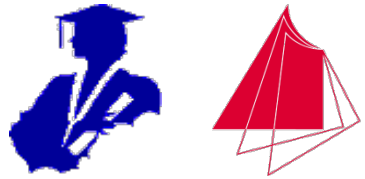
- Implementierung einer Schnittstelle:

```
public class Figure implements Runnable {
  @Override
  public void run() { /* ... */ }
}
```

- Eine Klasse darf beliebig viele Schnittstellen direkt implementieren (die Schnittstellen werden mit Kommata getrennt aufgeführt).

Vererbung

Vererbung der Spezifikation (Schnittstellen)



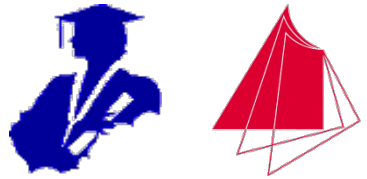
- Was passiert, wenn eine Schnittstelle um zusätzliche Methoden erweitert werden soll?
- Alle erbenden Klassen müssten angepasst werden → schwer umsetzbar, wenn weltweit tausende Projekte die Schnittstellen verwenden!
- Problem wurde bei der Erweiterung der Collections-Klassen in Java 8 behoben.
- Lösung: Schnittstellen dürfen Methoden Standard-Implementierungen geben:

```
public interface MyInterface {
  void make();
  default int makeFourtyTwo() {
    return 42;
  }
}
```

- Wenn die erbende Klasse die Methode nicht implementiert, dann wird die Implementierung der Schnittstelle verwendet.
- Tipp: Wirklich nur für eine spätere Erweiterung von Schnittstellen verwenden!

Vererbung

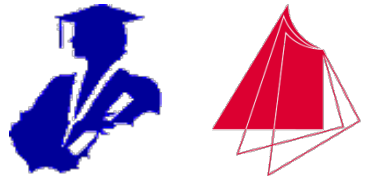
Vererbung der Spezifikation (Schnittstellen)



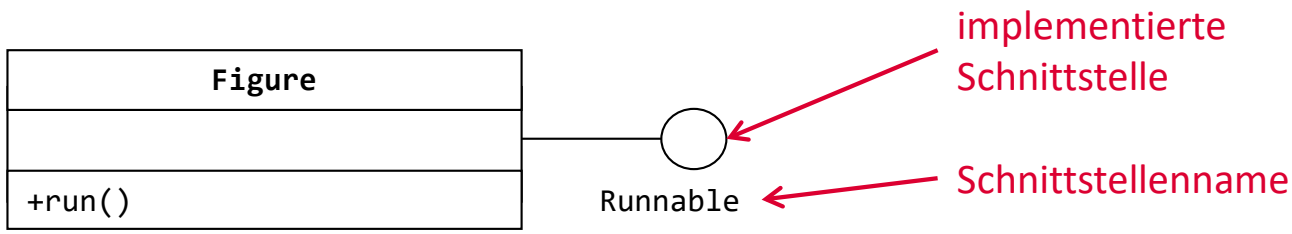
- Schnittstellen dürfen auch statische Methoden besitzen. Diese müssen eine Standard-Implementierung enthalten.

```
public interface MyInterface {
  void make();
  static MyInterface getInstance() {
    return new MyClass();
  }
}
```

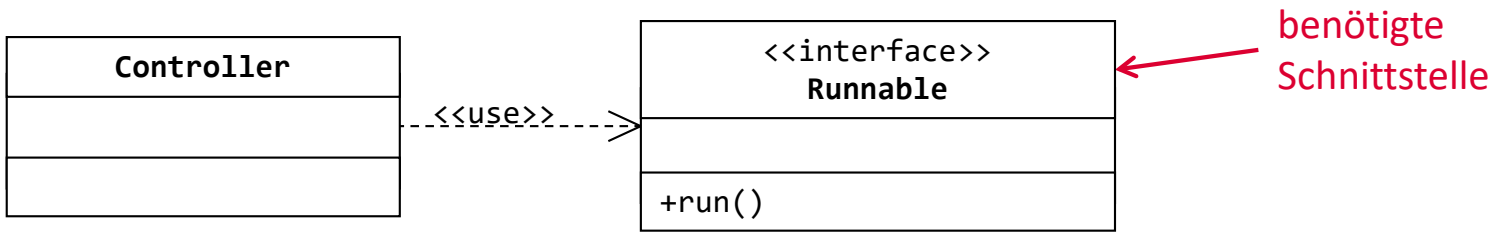
- Sinnvoll für sogenannte Fabrik-Methoden (Methoden erzeugen Objekte einer Klasse) → Einsatz kommt später im Datenstruktur-Kapitel.

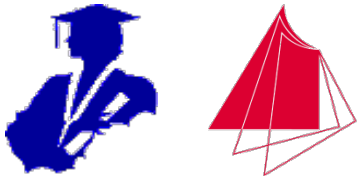


- Alternative Darstellung mit Ball-Symbol (die Klasse **`Figure`** implementiert die Schnittstelle **`Runnable`**):

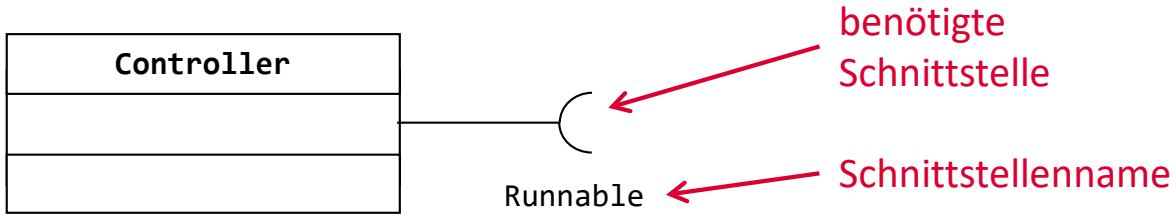


- Algorithmen und Datenstrukturen operieren häufig nur auf Schnittstellen:
  - Klassendiagramme einer Beziehung genau wie bei einer Abhängigkeit zwischen Klassen notwendig
  - Hier: Klasse benötigt eine Schnittstelle

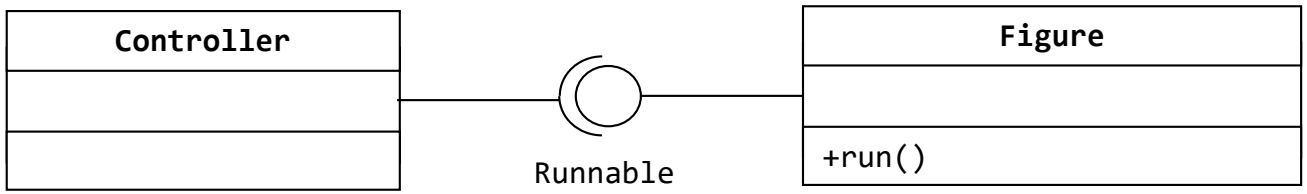




- Alternative Darstellung mit dem Ball-Symbol:



- Gemeinsame Darstellung: Schnittstelle, implementierende Klasse und Klasse, die die Schnittstelle benötigt, in der Ball-Darstellung:



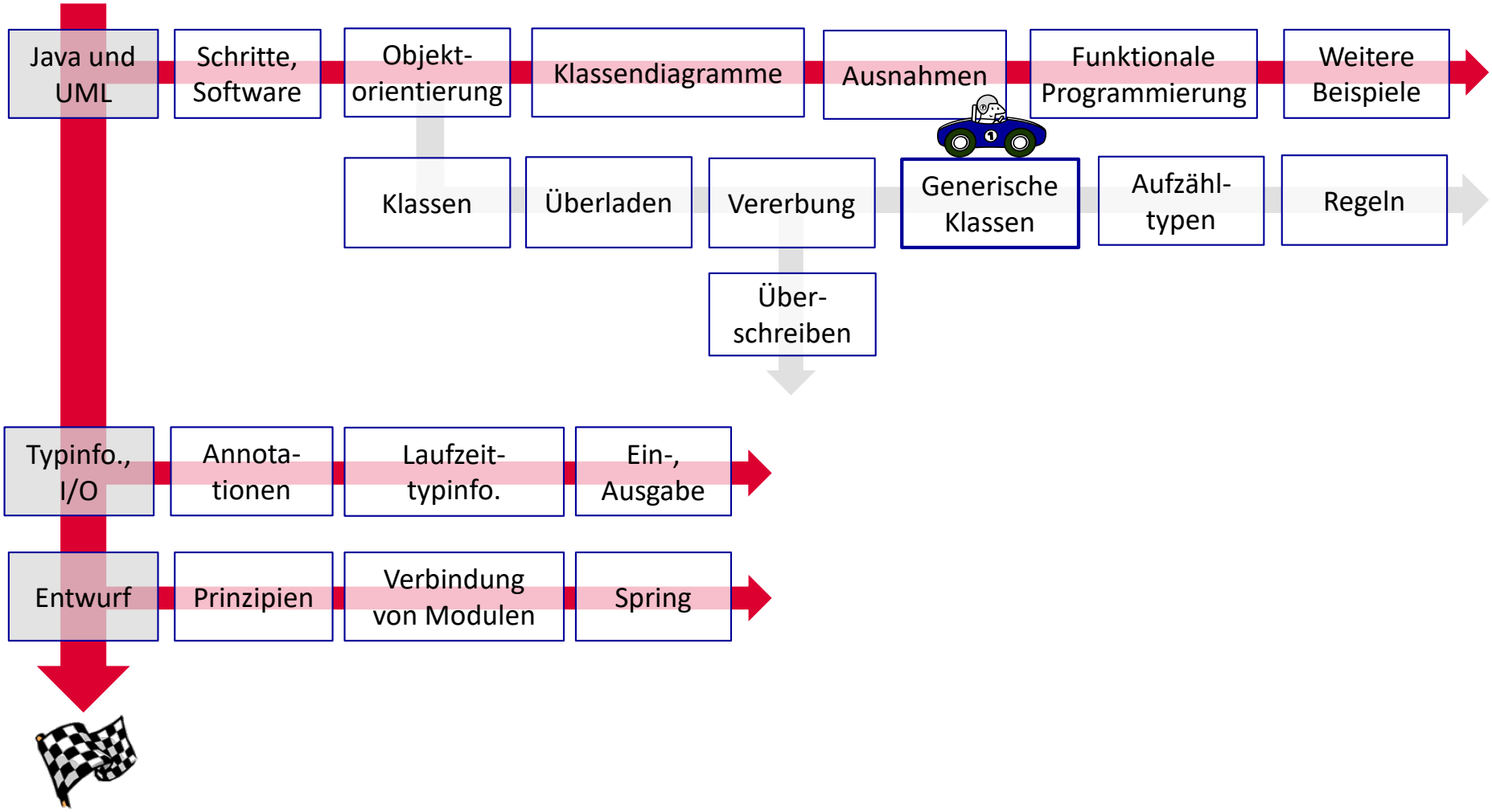
- Implementierung:

```
public class Controller {
  private Runnable[] elements;
  // ...
}
```

Vererbung
Beispiel: „Pacman“

- Ziel: Klassenhierarchie für das Zeichenprogramm oder Pacman-Figuren
- Die Algorithmen des Programms sollen auch mit noch „unbekannten“ Figuren funktionieren
  → Erweiterbarkeit!

Generische Klassen
Übersicht

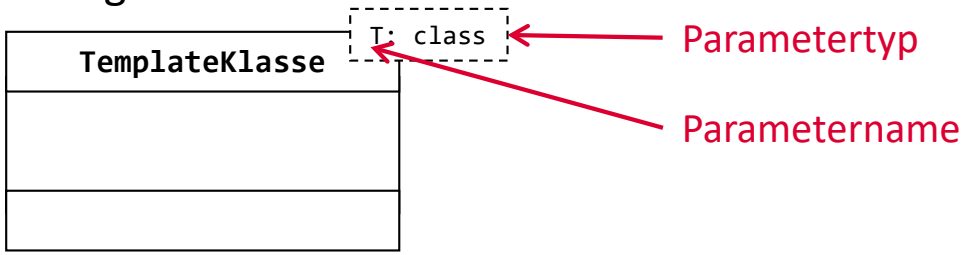


Generische Klassen

Definition: Generische Klasse

Eine generische Klasse ist eine mit formalen generischen Parametern versehene Schablone. Erst durch die Verwendung der Klasse werden die Parameter durch konkrete Klassen ersetzt. So kann zur Übersetzungszeit eine höhere Typsicherheit sichergestellt werden.

Darstellung mit UML:

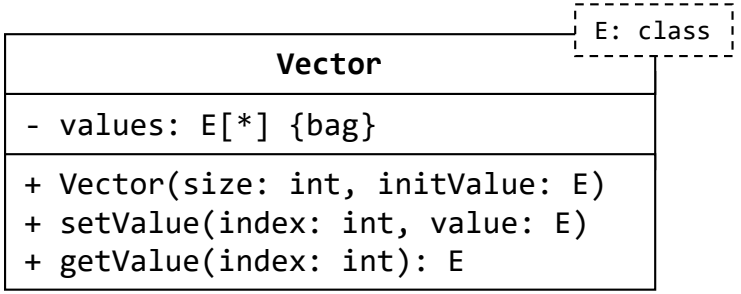


- Parameteraufbau: ***Parameter-Name***[**:*Parameter-Typ***][**=*Vorgabewert***]
  - ***Parameter-Name***: Name des Platzhalters
  - ***Parameter-Typ***: Optionale Klasse des Parameters. Ist kein Typ angegeben, kann jede beliebige Klasse **`class`** verwendet werden.
  - ***Vorgabewert***: Standardwert, falls die Typangabe fehlt

Generische Klassen
Vektor (Version 3)

- Vektor als generische Klasse:

```
public class Vector<E> {
  private E[] values;

  @SuppressWarnings("unchecked")
  public Vector(int size, E initValue) {
    values = (E[]) new Object[ size ];
    for (int i = 0; i < size; ++i) {
      values[ i ] = initValue;
    }
  }

  public void setValue(int index, E value) {
    values[ index ] = value;
  }

  public E getValue(int index) {
    return values[ index ];
  }
}
```



Generische Klassen

- Anmerkungen:
  - `public class Vector<E>`: `<E>` ist der Platzhalter für eine Klasse.
  - Der Platzhalter steht immer für den Namen einer Klasse oder Schnittstelle:
    - Primitive Datentypen werden nicht unterstützt.
    - Intern speichert die Klasse Referenzen vom Typ `Object` ab.
    - Der Platzhalter dient zum typsicheren Zugriff, der zur Übersetzungszeit geprüft werden kann.
    - Der Platzhaltertyp kann fast wie ein normaler Datentyp innerhalb der Klasse verwendet werden. Ausnahmen: Objekterzeugung, Arrays
  - Der cast-Operator im Konstruktor ist nicht geprüft. Deshalb wird die Warnung des Compilers mit `@SuppressWarnings("unchecked")` unterdrückt.
  - In Java können keine Arrays mit generischen Typen angelegt werden. Deshalb erzeugt der Vektor `Object`-Arrays → kommt noch genauer.
  - Es dürfen mehrere Platzhalter verwendet werden. Beispiel:

```
public class HashMap<K,V> { /* … */ }
```

Generische Klassen

- Eine konkrete Instanz einer generischen Klasse wird durch die folgende Schreibweise erzeugt: **`Klasse<Typ>`**
- Beispiel:

```
Vector<Double> vector1 = new Vector<>(3, 0.0);
```

Kann anhand des Typs der Referenz erkannt werden.

- Beispiel (der Compiler stellt sicher, dass die rot markierten Elemente vom Typ **`Double`** bzw. **`double`** sind):

```
Vector<Double> v1 = new Vector<>(3, 0.0);
v1.setValue(0, 2.0);
double value = v1.getValue(0);
System.out.println(value);
```

- Beispiel ohne generische Klasse (der Compiler kann keine Typprüfung vornehmen):

```
Vector v1 = new Vector(3, 0.0);
v1.setValue(0, 2.0);
Double value = (Double) v1.getValue(0);
System.out.println(value);
```

Falsche Werte (z.B. **`String`**) werden zur Laufzeit bemerkt (**`ClassCastException`**).

Generische Klassen

Umsetzung in der virtuellen Maschine

- Generische Klassen wurden erst mit Java 5 eingeführt.
- Ziel war es, den Bytecode zu Java 1.4 kompatibel zu halten.
- Wie lassen sich generische Klassen implementieren?
  - heterogen: Für jeden Typparameter wird eine neue Klasse erzeugt (C++-Ansatz).
  - homogen: Es wird nur eine Klasse erzeugt. Der Typparameter wird durch die oberste Basisklasse **`Object`** ersetzt. Die Typparameter dienen nur zur Prüfung während der Übersetzungszeit (Java-Ansatz).
- Der Compiler löscht also die Typ-Informationen („type erasure“):
  - Der Bytecode bleibt kompatibel zu Java 1.4.
  - Die generischen Typen existieren nicht mehr zur Laufzeit → führt zu Problemen:

```
Vector<Double> v1 = new Vector<>();
if (v1 instanceof Vector<Double>) // Compilerfehler, da Vector<Double> zur
                                  // Laufzeit nicht existiert!
```

Korrekt wäre:

```
Vector<Double> v1 = new Vector<>();
if (v1 instanceof Vector)         // Vector existiert
```

Generische Klassen

Umsetzung in der virtuellen Maschine

- Damit sind die Klassen trotz unterschiedlicher Typparameter identisch:

```
Vector<Double> v1 = new Vector<>();
Vector<String> v2 = new Vector<>();
if (v1.getClass() == v2.getClass()) // true
```

- Generische Klassen lassen sich auch ohne Typparameter verwenden („raw type“):

```
Vector v1 = new Vector<Double>();
Vector v2 = new Vector();
```

Die Verwendung führt aber z.B. bei **`setValue`** zu einer Compiler-Warnung.

Generische Klassen

Umsetzung in der virtuellen Maschine

- Beispiel für die Umsetzung anhand des Vektors (Ausschnitt):

Definition im Quelltext

```
public class Vector<E> {
  private E[] values;

  // ...

  public void setValue(int ind, E v) {
    values[ ind ] = v;
  }

  public E getValue(int ind) {
    return values[ ind ];
  }
}
```

Definition zur Laufzeit („fiktiv“)

```
public class Vector {
  private Object[] values;

  // ...

  public void setValue(int ind, Object v) {
    values[ ind ] = v;
  }

  public Object getValue(int ind) {
    return values[ ind ];
  }
}
```

Verwendung im Quelltext

```
Vector<String> names = new Vector<>();
names.setValue(0, "Vogelsang");
String name = names.getValue(0);
```

Verwendung zur Laufzeit („fiktiv“)

```
Vector names = new Vector();
names.setValue(0, "Vogelsang");
String name = (String) names.getValue(0);
```

- Arrays mit generischen Typen haben einige Besonderheiten:
  - Die Deklaration ist problemlos möglich:

```
private ArrayList<String>[] al;
```

  - Aber eine Initialisierung klappt nicht intuitiv:

```
private ArrayList<String>[] al = new ArrayList<>[ 10 ];
                                 // Compilerfehler
```

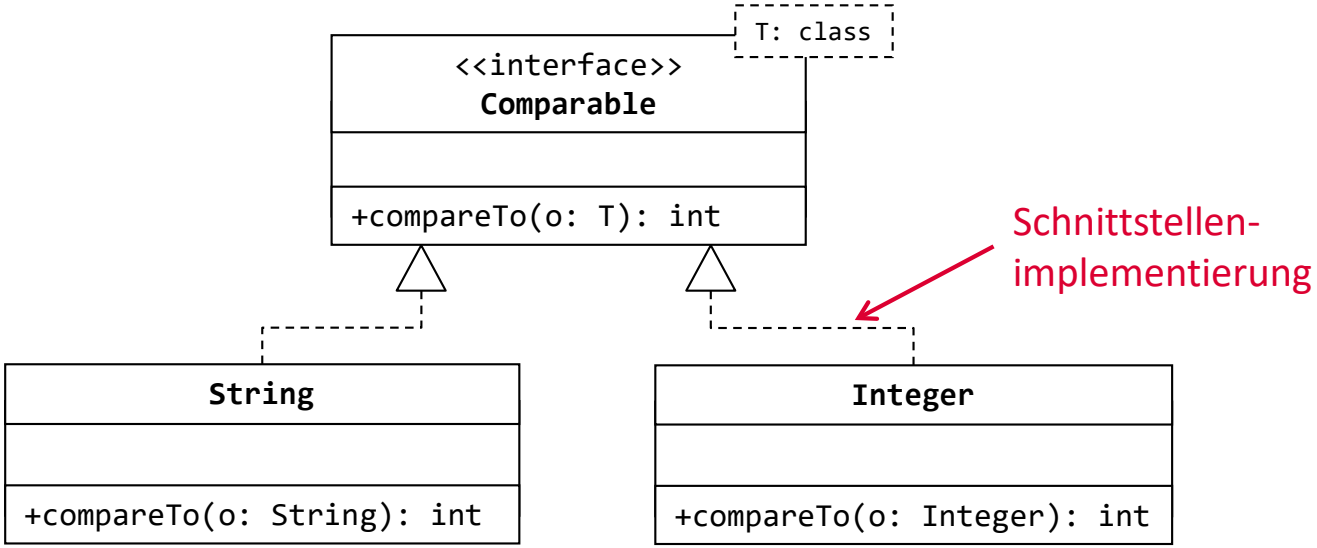
- Lösungen:
  - Verwendung des Platzhalters **?**:

```
private ArrayList<String>[] al
        = (ArrayList<String>[]) new ArrayList<?>[ 10 ];
```

  - Erstellen einer Klasse für den Inhalt der Arrayzellen:

```
public class MyList extends ArrayList<String> {
  // notwendige Konstruktoren
}
private MyList[] al = new MyList[ 10 ];
```

Auch Schnittstellen können generisch sein:



- Schnittstelle **`Comparable`** aus dem JDK:

```
public interface Comparable<E> {
  int compareTo(E other);
}
```

- Implementierung der Schnittstelle:

```
public final class String implements Comparable<String> {
  @Override
  public int compareTo(String other) {
    // ...
  }
}
```

- Einsatzbeispiel für die Schnittstelle:
  - Ein Sortieralgorithmus soll beliebige Arrays sortieren können.
  - Wie soll dieser Algorithmus wissen, welche Daten größer und welche kleiner sind?
  - Lösung: Er fragt z.B. die Daten selbst!

- Variante 1: Daten müssen die Schnittstelle **`Comparable`** implementieren.

```
public class Customer implements Comparable<Customer> {
  // ...
  private int number;
  @Override
  public int compareTo(Customer other) {
    return number - other.number;
  }
}
```

- Jetzt kann der existierende Sortieralgorithmus verwendet werden:

```
ArrayList<Customer> customers = new ArrayList<>();
// ...
Collections.sort(customers);
```

- Problem: Was passiert, wenn nach unterschiedlichen Kriterien sortiert werden soll?
- Lösung: Der Vergleich hat nichts in der Klasse **`Customer`** zu suchen. Er wird als Argument an den Algorithmus übergeben.

- Hinweis: Wenn **`compareTo`** implementiert wird, sollte auch **`equals`** überschrieben werden (kommt noch…).

- Variante 2: Separater „Vergleicher“, der die generische Schnittstelle **`Comparator<T>`** implementiert.

```
public class Customer {
  // ...
  private int number;
  // Getter und Setter
}
```

- Vergleicher für zwei Kundenobjekte:

```
public class CustomerNumberComparator implements Comparator<Customer> {
  @Override
  public int compare(Customer c1, Customer c2) {
    return c1.getNumber() - c2.getNumber();
  }
}
```

- Jetzt kann der existierende Sortieralgorithmus verwendet werden:

```
ArrayList<Customer> customers = new ArrayList<>();
// ...
Collections.sort(customers, new CustomerNumberComparator());
```

- Es wird eine „Funktion“ in einem Objekt gekapselt → geht noch eleganter → kommt später…

Generische Methoden

- „Normale“ (nicht-generische) und generische Klassen dürfen generische Methoden besitzen.
- Das wird häufig bei Hilfsklassen mit statischen Methoden eingesetzt:
- Beispiel: Methode **`asArrayList`** erzeugt aus einer variablen Anzahl Übergabewerte eine **`ArrayList`** (angelehnt an Klasse **`Arrays`** aus dem SDK)

```
public class Arrays {
  @SafeVarargs
  public static <T> ArrayList<T> asArrayList(T... a) {
    ArrayList<T> result =  new ArrayList<>(a.length);
    for (int i = 0; i < a.length; ++i) {
      result.add(a[ i ]);
    }
    return result;
  }
}
```

- Syntax: ***`Zugriffsrecht <T> Rückgabetyp Methodenname(Parameter)`***

Generische Klassen

Einschränkungen der Parametertypen

- Die Typparameter lassen sich einschränken:
  - Basisklasse: Der Parametertyp muss von einer bestimmten oder mehreren Basisklassen erben.
  - Schnittstellen: Der Parametertyp muss eine bestimmte oder mehrere Schnittstellen implementieren.
- Beispiel, in dem der Vektor nur mit Klassen, die von **`Number`** erben, verwendet werden darf (weil der Vektor z.B. Zahlenoperationen durchführt), Vektor Version 4:

```
public class Vector<E extends Number> {
  private Number[] values;

  public double getSum() { // Summe alle Werte ermitteln
    double sum = 0.0;
    for (int i = 0; i < values.length; ++i) {
      sum += values[ i ].doubleValue();
    }
    return sum;
  }
  // ...
}
```

Generische Klassen

Einschränkungen der Parametertypen

- Es können auch mehrere Einschränkungen vorhanden sein.
- Bespiel: Der Typparameter des Vektors muss von **`Number`** erben und **`Cloneable`** implementieren:

```
public class Vector<E extends Number & Cloneable>
```

- Syntax bei mehr als zwei Einschränkungen:

```
public class Klasse<E extends C1 & C2 & C3>
```

- Achtung bei Vererbung:

  `Vector<Object>` ist keine Basisklasse von **`Vector<Double>`**!

```
Vector<Object> v1 = new Vector<Double>(); // Compilerfehler!
```

  Ansonsten ließe sich z.B. ein **`String`** in dem Vektor speichern.

  Die Methode

```
public void dump(Vector<Object> v1)
```

  darf nicht mit einem **`Vector<Double>`** aufgerufen werden.

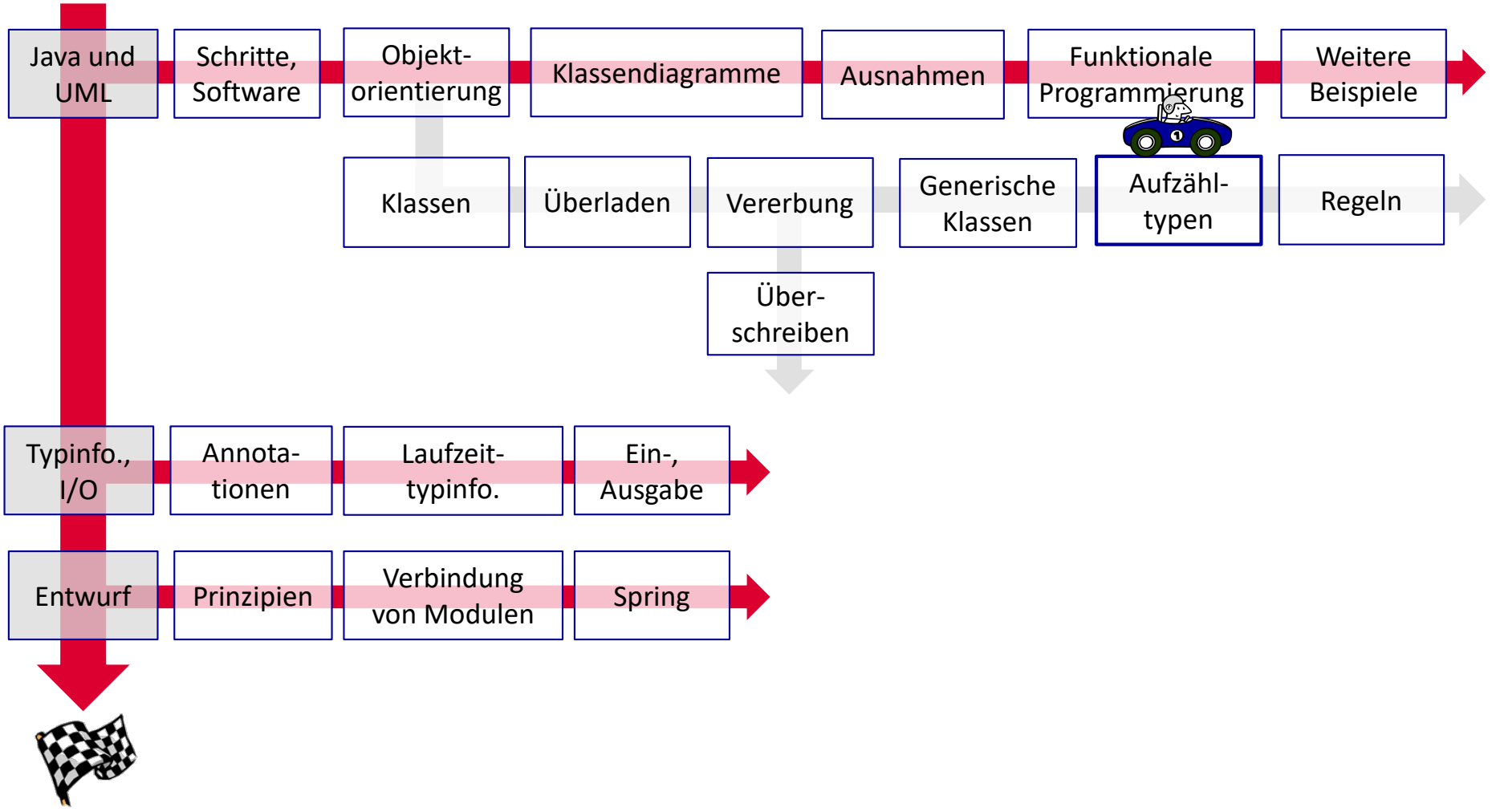
- Ausweg: Wildcards erlauben es, einen Typ unbestimmt zu lassen. Beispiel:

```
public void dump(Vector<?> v1)
```

  - **?** ist ein unbekannter Typ, nicht **Object**.
  - Die Methode **dump** darf jetzt mit jedem beliebigen Typparameter verwendet werden.

- Wildcards unterstützen auch Typeinschränkungen („upper bound wildcards“):
  - **public void dump(Vector<? extends Number> v1)**
  - Es können nur Vektoren übergeben werden, deren Inhalt von **Number** erbt.
- Einschränkung des Typparameters „nach unten“:
  - **public void dump(Vector<? super MyClass> v1)**
  - Es können nur Vektoren übergeben werden, deren Inhalt **MyClass** oder seine Basisklassen bzw. Schnittstellen sind.
  - Klassen, die von **MyClass** erben, sind nicht erlaubt.

Aufzähltypen
Übersicht



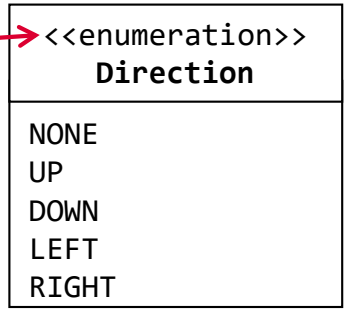
Aufzählтypen
Einführung

- Ein Aufzähltyp ist eine spezielle Klasse, von der benannte Objekte erzeugt werden.

  Beispiel für die Bewegungsrichtungen von Figuren in Pacman:

```
public enum Direction {
  NONE, UP, DOWN, LEFT, RIGHT;
}
```

Stereotyp



- Daraus erzeugt der Compiler eine Klasse, die ungefähr so aussieht:

```
public class Direction extends Enum<Direction> {
  public static final Direction NONE  = new Direction("NONE", 0);
  public static final Direction UP    = new Direction("UP", 1);
  public static final Direction DOWN  = new Direction("DOWN", 2);
  public static final Direction LEFT  = new Direction("LEFT", 3);
  public static final Direction RIGHT = new Direction("RIGHT", 4);

  private Direction(String name, int index) {
    super(name, index);
  }
  // Alle Werte ermitteln
  public static Direction[] values() { /* ... */ }

  // weitere Methoden ...
}
```

Aufzählltypen
Einführung

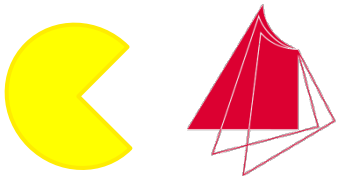
- Vorteil von Aufzählltypen gegenüber finalen `int`-Werten:
  - In einem `switch`-Block kann der Compiler testen, ob alle Werte eines Typs einen `case`-Block haben.
  - Der Benutzer kann anhand des Typs dessen Bedeutung erkennen.
  - Der Typ kann nur die erlaubten Werte annehmen.
- Was bietet die Basisklasse `Enum` noch (Auswahl)?
  - `public String name()`: Name des Aufzählwertes (z.B. `"LEFT"`)

    Beispiel: `String name = Direction.LEFT.name();`
  - `public int ordinal()`: Index eines Wertes in der Aufzählung (z.B. `3` für der Wert `Direction.LEFT`, weil `LEFT` als vierter Wert angegeben ist, Zählung zählt beginnt bei 0)

    Beispiel: `int index = Direction.LEFT.ordinal();`
  - `public static <T extends Enum<T>> valueOf(Class<T> enumType, String name)`: Ermittelt aus dem Namen den Aufzählwert

    Beispiel: `Direction dir = Enum.valueOf(Direction.class, "UP");`

Aufzähltypen
Einführung



- Ein Aufzähltyp kann in Pacman für zwei Aufgaben eingesetzt werden:
  - Richtung, in die sich eine Figur bewegt

```
public enum Direction {
  NONE, UP, DOWN, LEFT, RIGHT;
}
```

  - Position einer Zelle, an der sich ein Rahmen befindet.
    - Problem: Eine Zelle kann an mehr als einer Position einen Rahmen haben.
    - Lösung: Bitkombination als zusätzlichen Wert dem Aufzähltyp übergeben → ersetzt nicht den Index (Aufruf **`ordinal()`**).

```
public enum Direction { // noch unvollständig
  NONE(0), UP(1), DOWN(2), LEFT(4), RIGHT(8), ALL(15);
}
```

- Die zweite Variante wird auch für die Richtung verwendet, wobei **`ALL`** unsinnig ist.
- Beispiele:

`LEFT | UP | DOWN` **`RIGHT`**

Aufzählltypen

Methoden und Attribute

- Wie funktioniert die Übergabe zusätzlicher Parameter an einen Aufzählwert genau? Dieses Beispiel ergibt Übersetzungsfehler, weil der entsprechende Konstruktor fehlt:

```
public enum Direction {
  NONE(0), UP(1), DOWN(2), LEFT(4), RIGHT(8), ALL(15);
}
```

- Aufzählltypen dürfen zusätzliche Konstruktoren, Attribute und Methoden erhalten:

```
public enum Direction {
  NONE(0), UP(1), DOWN(2), LEFT(4), RIGHT(8), ALL(15);

  private int value;

  private Direction(int value) {
    this.value = value
  }

  public int getValue() {
    return value;
  }

  // weitere Methoden
}
```

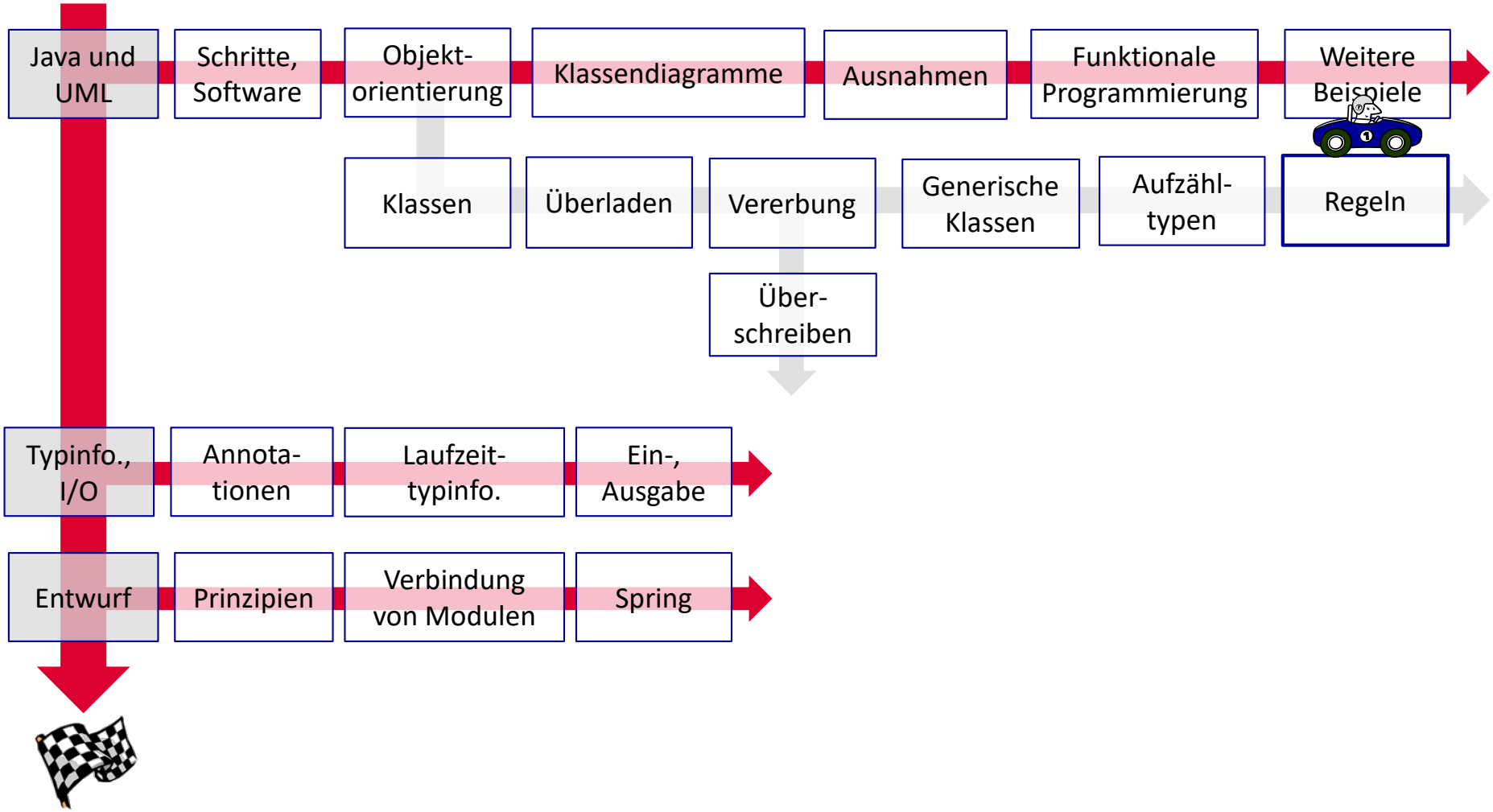
Konstruktoraufruf

Aufzählltypen

Methoden und Attribute

- Aufzählltypen dürfen auch Schnittstellen implementieren.
- Sie können aber nicht von Klassen erben, weil sie intern bereits **`Enum<E>`** als Basisklasse besitzen.

Regeln und Hinweise
Übersicht



Regeln und Hinweise

Wichtige Methoden und Operatoren einer Java-Klasse

- Welche Methoden und Operatoren haben eine besondere Bedeutung (kleine Auswahl, hier anhand der Klasse **`Ghost`**)?

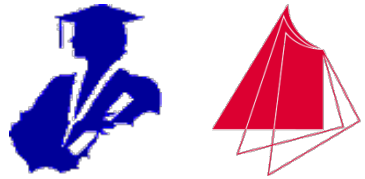
Methode/Operator	Bedeutung
`Ghost()`	Standardkonstruktor. Wird immer dann automatisch erzeugt, wenn kein eigener Konstruktor geschrieben wurde.
`protected void finalize()`	Wird aufgerufen, bevor das Objekt gelöscht wird.
`public boolean equals(Object s)`	Kommt noch…
`protected Object clone()`	Wird verwendet, um eine Kopie eines Objekts zu erzeugen, kommt noch…
`public String toString()`	Wird aufgerufen, wenn eine String-Darstellung des Objektes benötigt wird.

- Identität und Gleichheit zweier Objekte (hier anhand der Klasse **`Ghost`**):

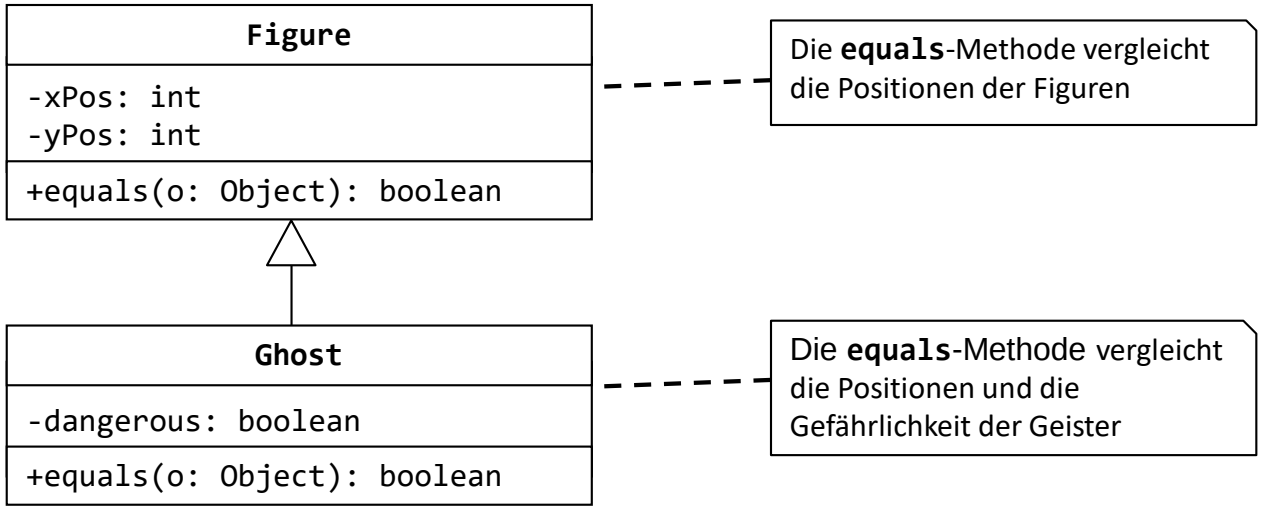
Methode/Operator	Bedeutung

`Ghost gh1 = new Ghost();`
`Ghost gh2 = new Ghost();`
`if (gh1.equals(gh2))`
	Inhaltlicher Vergleich der Objekte. Dazu sollte die Methode **`equals`** der Klasse **`Ghost`** überschrieben werden.

`Ghost gh1 = new Ghost();`
`Ghost gh2 = new Ghost();`
`if (gh1 == gh2)`
	Identität der Objekte. Dazu werden die Referenzen auf die Objekte verglichen. Sind diese identisch, so handelt es sich um dasselbe Objekt.

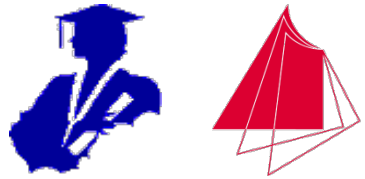


- Bedingungen für die Gleichheit zweier Objekte:
  - Die Gleichheitsprüfung ist reflexiv: A ist gleich A.
  - Die Gleichheitsprüfung ist symmetrisch: A ist gleich B ==> B ist gleich A.
  - Die Gleichheitsprüfung ist transitiv: A ist gleich B und B ist gleich C ==> A ist gleich C.
- Jede Implementierung muss diese Bedingungen einhalten!
- Problem: Wie sieht der Vergleich bei Vererbung aus?



Regeln und Hinweise

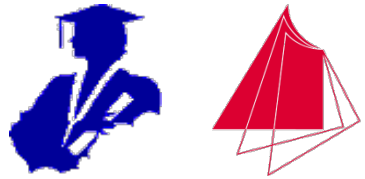
Wichtige Methoden und Operatoren – Vergleich von Objekten



- Ohne Beweis: Der Vergleich von Objekten unterschiedlicher Klassen führt zur Verletzung mindestens einer Regeln.
- Konsequenz: Nur Objekte identischer Klassen werden überhaupt auf Gleichheit überprüft.
- Beispiel:

```
public class Figure {
  private int xPos;
  private int yPos;
  // ...
  @Override
  public boolean equals(Object o) {
    if (o != null &&
        o.getClass() == getClass()) {
      Figure f = (Figure) o;
      return xPos == f.xPos
          && yPos == f.yPos;
    }
    return false;
  }
}
```

```
public class Ghost extends Figure {
  private boolean dangerous;
  // ...
  @Override
  public boolean equals(Object o) {
    return super.equals(o) &&
     ((Ghost) o).dangerous == dangerous;
  }
}
```



- Erklärung des Beispiels:
  - Die Basisklasse **`Figure`** testet mit **`getClass()`** zunächst auf identische Klassen.
  - Nur dann werden die Attribute vergleichen.
  - Die abgeleitete Klasse **`Ghost`** lässt die Basisklasse die Gleichheit feststellen und prüft dann das zusätzliche Attribut.
- Anmerkung:
  - In der Praxis ist es manchmal auch sinnvoll, nur die Basisklasse ohne Test auf Klassengleichheit die Prüfung vornehmen zu lassen → hängt von der Aufgabe ab.
  - Dann können eine Figur und ein Geist gleich sein!

Regeln und Hinweise

Wichtige Methoden und Operatoren – Kopie von Objekten

- Es gibt in Java einen definierten Weg, um Kopien von Objekten zu erzeugen.
- Warum nicht einfach direkt?

```
public class Figure {
  private int xPos;
  private int yPos;
  public Figure getCopy() {
    Figure copy = new Figure();
    copy.xPos = xPos;
    copy.yPos = yPos;
    return copy;
  }
  // ...
}
```

```
public class Pacman extends Figure {
  private int direction;

  // ...
}
```

- Sieht logisch aus, ist bei Vererbung aber falsch!
- Was passiert, wenn die Klasse **`Pacman`** von **`Figure`** erbt? Dann handelt es sich nicht mehr um ein **`Figure`**-, sondern im ein **`Pacman`**-Objekt!

- Auch das Überschreiben der Methode **`getCopy`** hilft nicht weiter.:

```
public class Pacman extends Figure {
  private int direction;

  @Override
  public Pacman getCopy() {  // kovarianter Rückgabetyp ist erlaubt
    // und jetzt?
  }
  // ...
}
```

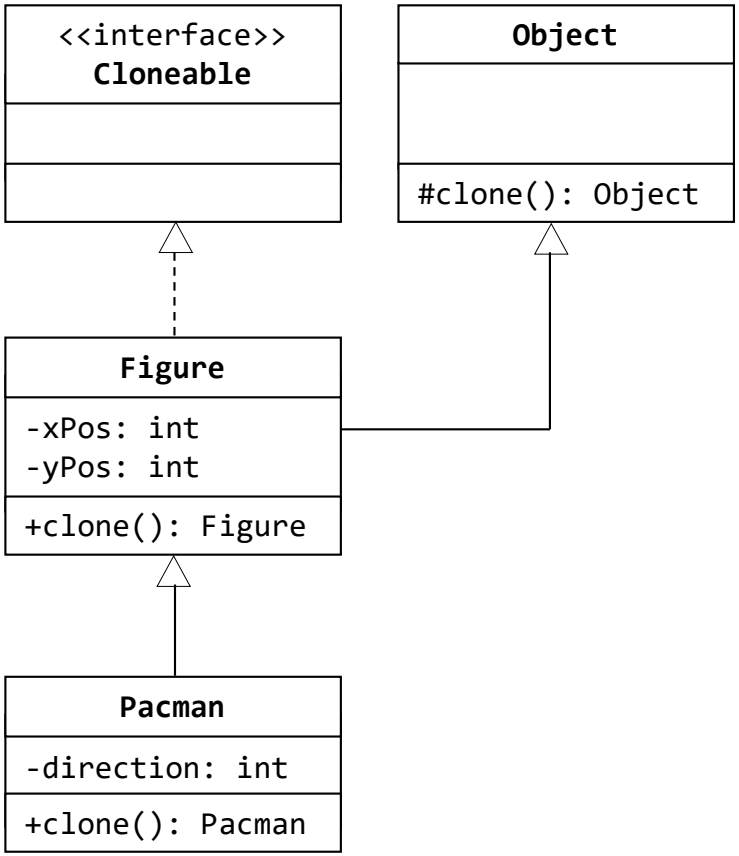
- Wie soll jetzt die Basisklasse **`Figure`** ihre eigenen Attribute kopieren?
- Den Ausweg bietet die oberste Basisklasse **`Object`** mit der **`clone`**-Methode:
  - Sie erstellt ein Objekt der korrekten Klasse.
  - Die Klasse muss die Schnittstelle **`Cloneable`** implementieren, ansonsten löst der Aufruf eine **`CloneNotSupportedException`** aus.
  - Die **`clone`**-Methode kopiert alle Attribute des Objektes, auf dem sie aufgerufen wurde.

- So sieht es dann aus:

```
public class Figure implements Cloneable {
  private int xPos;
  private int yPos;
  @Override
  public Figure clone()
              throws CloneNotSupportedException {
    return (Figure) super.clone();
  }
}
```

- **`clone`** in **`Object`** kopiert auch **`xPos`** und **`yPos`**.
- Und in Pacman:

```
public class Pacman extends Figure {
  private int direction;
  @Override
  public Pacman clone()
              throws CloneNotSupportedException {
    return (Pacman) super.clone();
  }
}
```

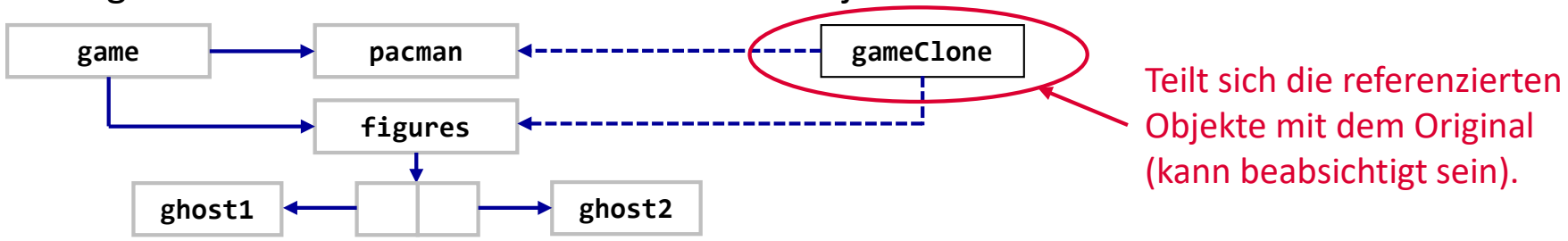


- **`clone`** erzeugt eine flache Kopie: Referenzen werden kopiert, aber nicht die Objekte, auf die die Referenzen verweisen.
- Beispiel:

```
public class GameShallowClone
                            implements Cloneable {
  private Pacman pacman;
  private ArrayList<Figure> figures =
                    new ArrayList<>();
  @Override
  public Game clone()
              throws CloneNotSupportedException {
    return (Game) super.clone();
  }
}
```

```
Pacman pacman = new Pacman(2, 3);
Ghost ghost1 = new Ghost(42, 3);
Ghost ghost2 = new Ghost(41, 13);
GameShallowClone game =
           new GameShallowClone();
game.setPacman(pacman);
game.getFigures().add(ghost1);
game.getFigures().add(ghost2);
GameShallowCone gameClone =
           game.clone();
```

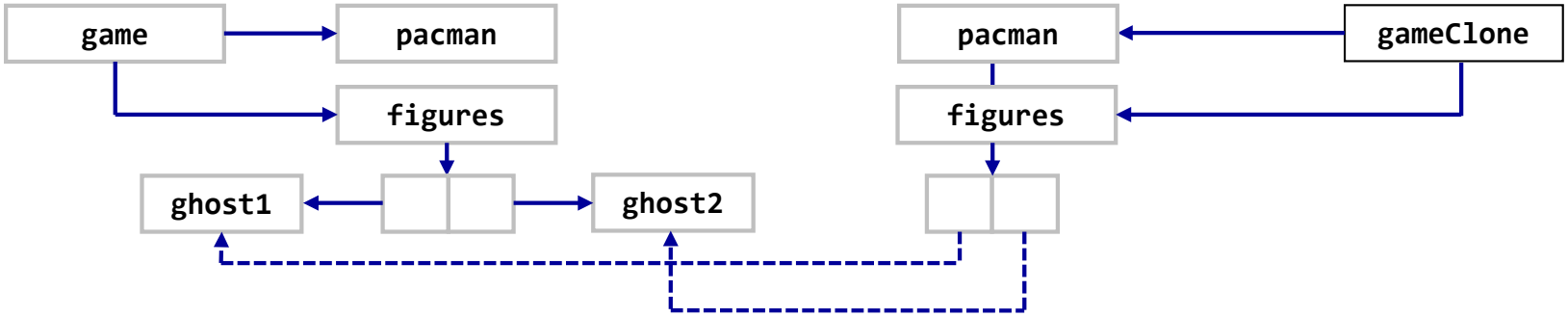
- Erzeugt beim Aufruf von **`clone`** mit Pacman-Objekt und zwei Geistern:



- Erzeugten einer tiefen Kopie (naja, fast):

```
public class GameDeepClone implements Cloneable {
  private Pacman pacman;
  private ArrayList<Figure> figures = new ArrayList<>();
  @SuppressWarnings("unchecked")
  @Override
  public GameDeepClone clone() throws CloneNotSupportedException {
    GameDeepClone clone = (GameDeepClone) super.clone();
    clone.pacman = pacman.clone();
    clone.figures = (ArrayList<Figure>) figures.clone();
    return clone;
  }
}
```

- Erzeugt beim Aufruf von **`clone`** mit Pacman-Objekt und zwei Geistern:

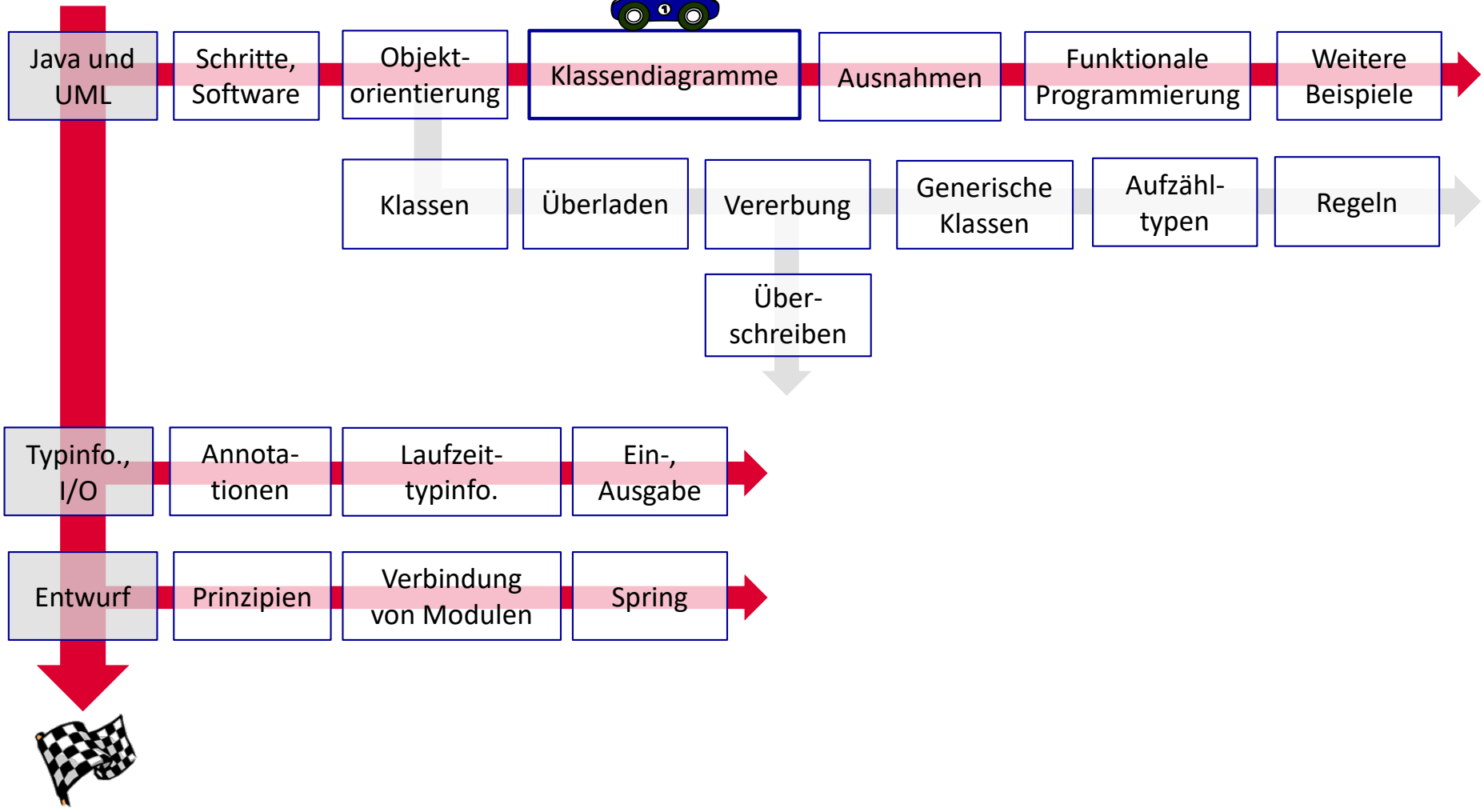


Regeln und Hinweise

Wichtige Methoden und Operatoren – Kopie von Objekten

- Die **`ArrayList`** erzeugt von ihrem Inhalt keine tiefe Kopie, siehe API-Dokumentation: „Returns a shallow copy of this ArrayList instance. (The elements themselves are not copied)“
- Um auch den Inhalt der **`ArrayList`** zu kopieren:
  - Implementierung in in der Game-Klasse implementiert.
  - Schreiben einer speziellen **`ArrayList`**, die eine tiefe Kopie erzeugt.

Klassendiagramme
Übersicht



Klassendiagramme

Beziehungen zwischen Klassen

- Zwischen Klassen können unterschiedliche Beziehungen bestehen.
- Die Beziehungen lassen sich ja nach Aufgabe in unterschiedliche Kategorien einteilen.

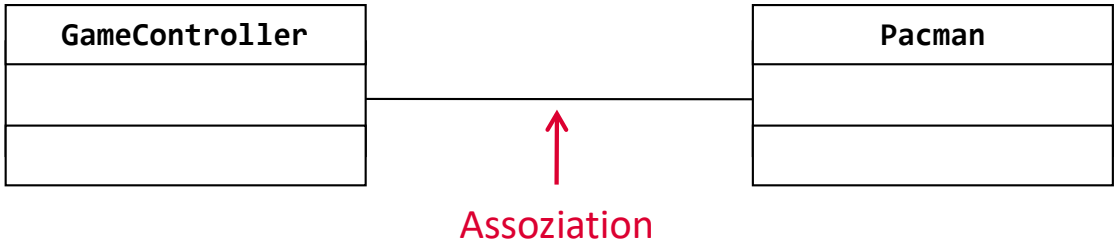
Kategorien in der UML:

- Assoziation
- Aggregation und Komposition („Teil und Ganzes“)
- Vererbung (von einer Spezifikation, von einer Implementierung) → schon bekannt

Definition: Binäre Assoziation

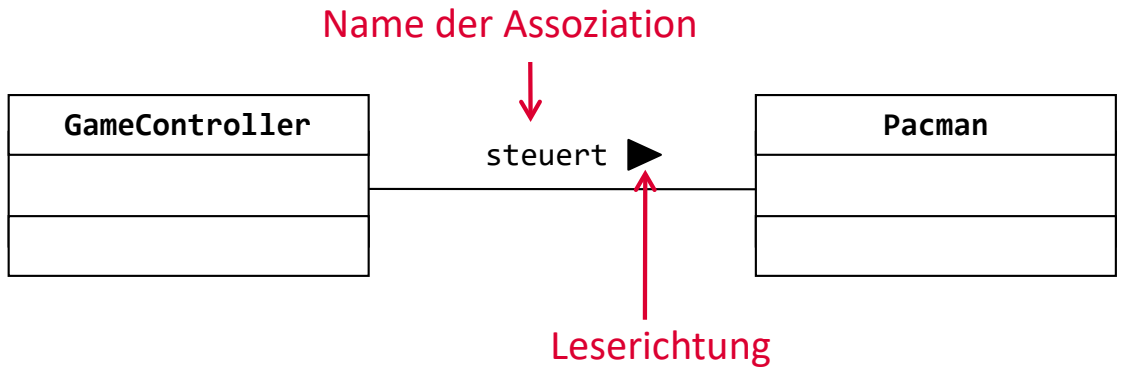
Eine binäre Assoziation beschreibt die semantische Beziehung zwischen zwei Klassen.

Beispiel:

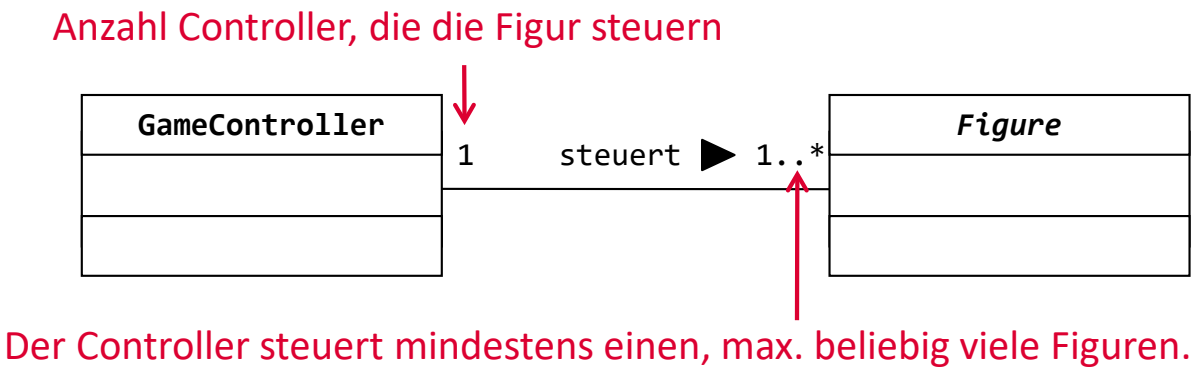


- **`Pacman`** und **`GameController`** „kennen“ sich.
- Beide Klassen können auf der jeweils anderen Operationen aufrufen.

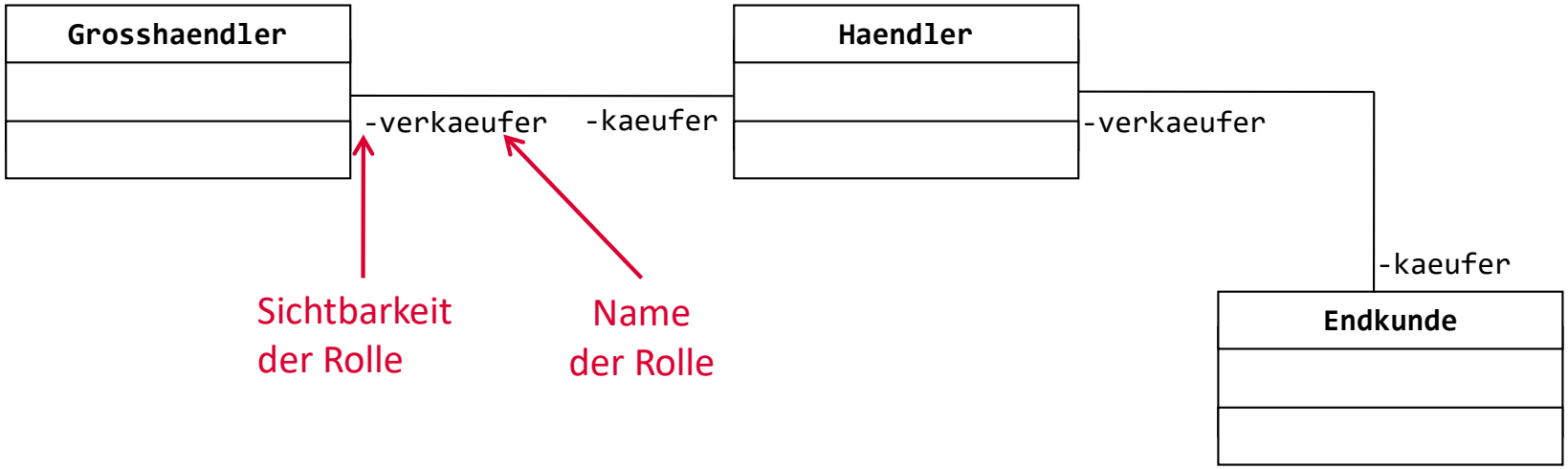
- **Leserichtung** (`GameController` steuert `Pacman`):



- **Multiplizität** (wie viele Objekte können gleichzeitig an der Beziehung beteiligt sein):



- **Rolle** (sie beschreibt, welche Funktion/Rolle eine an einer Assoziation beteiligten Klasse einnimmt):



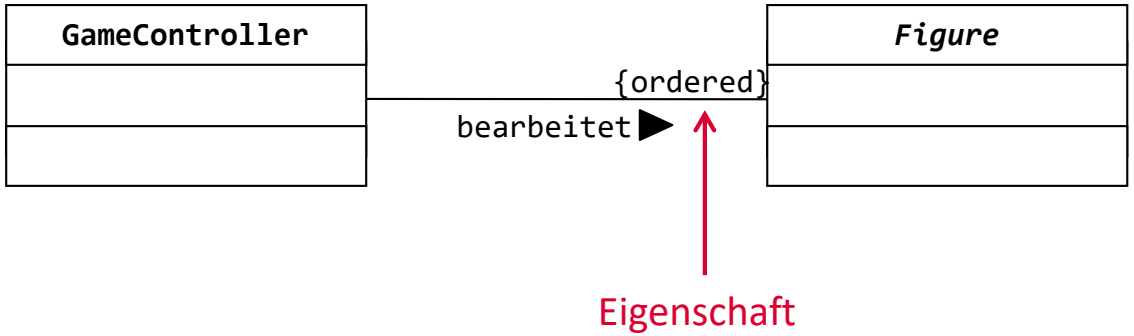
  - Zweck einer Rolle:
    - Besseres Verständnis der Assoziation
    - Name des Attributs in der Implementierung

- Implementierungsbeispiel, in dem der Name der Rolle den Namen des Attributes ergibt:

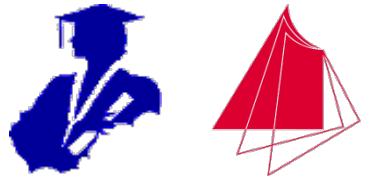
```
public class Haendler {
  private ArrayList<Grosshaendler> verkaeufer;
  private ArrayList<Endkunde>      kaeufer;

  // Methoden usw.
}
```

- Im Falle einer automatischen Code-Erzeugung ergeben sich so lesbare Attributnamen im Quelltext.
- Achtung: Eine Rolle mit privater Sichtbarkeit kann durch Getter- und Setter-Methoden von „außen“ zugänglich gemacht werden.

- **Eigenschaft** (ähnlich wie Eigenschaften von Attributen geben sie Eigenschaften von Assoziationen nähere Hinweise auf die Umsetzung der Assoziation):

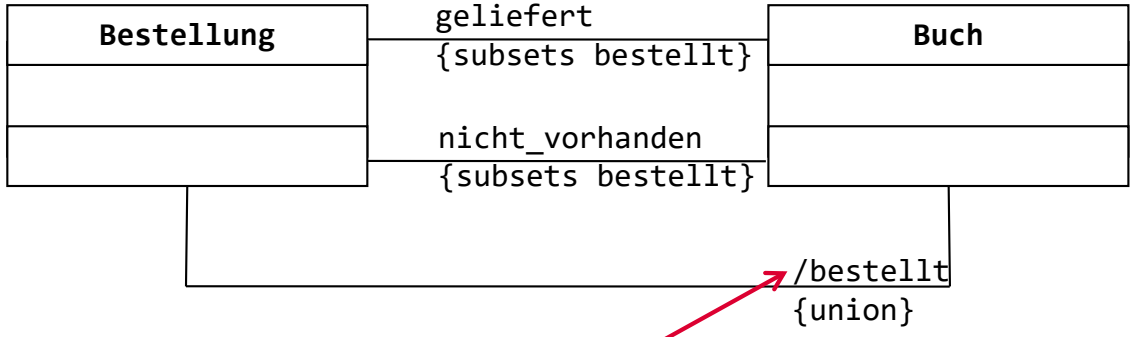


  - Der Controller benötigt die Figuren in einer bestimmten Reihenfolge (ist in der Beispielimplementierung aber nicht der Fall).

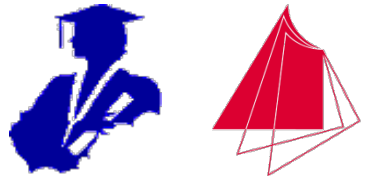


- Eigenschaftstypen (unvollständig):
  - **{subsets *<Assoziations-Ende>*}**: Die Menge der Objekte an diesem Assoziationsende ist eine Teilmenge der Objekte am Assoziationsende ***<Assoziations-Ende>***.
  - **{union}**: Die Menge der Objekte an diesem Assoziationsende ist die Vereinigung aller seiner **subsets**-Assoziationsenden.

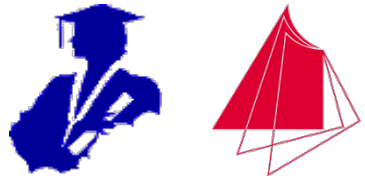
    Beispiel (die Buchbestellung setzt sich aus den bereits gelieferten und den nicht verfügbaren Büchern zusammen):



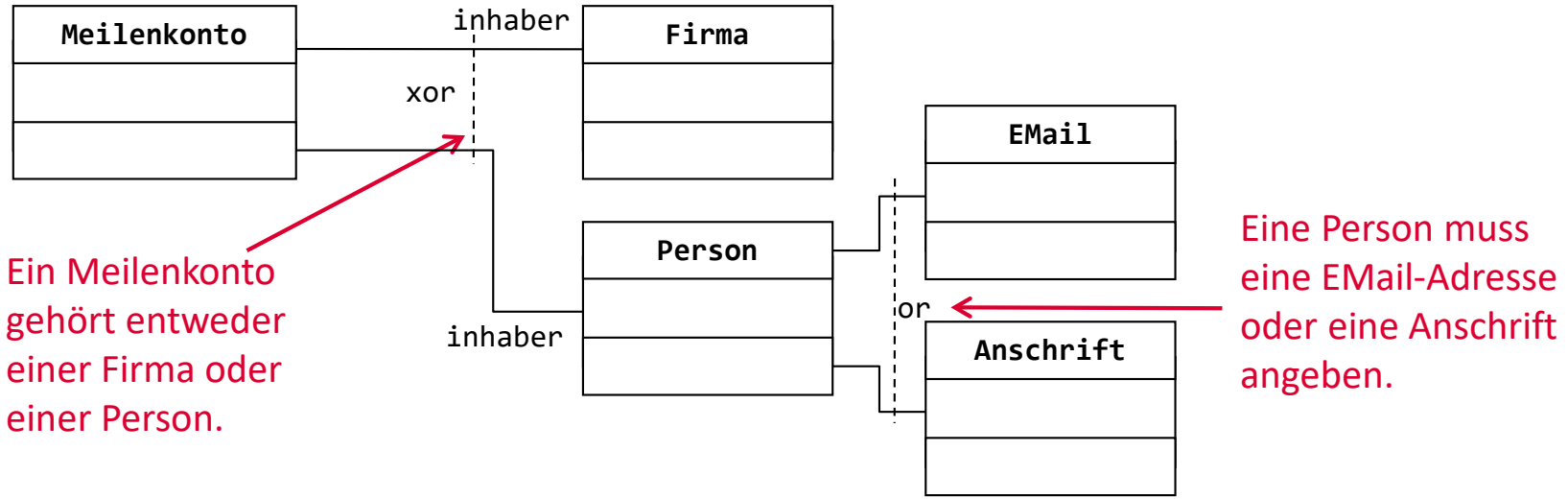
Die Menge **bestellt** ist aus allen Teilmengen abgeleitet. Es kommen keine weiteren Bücher hinzu.



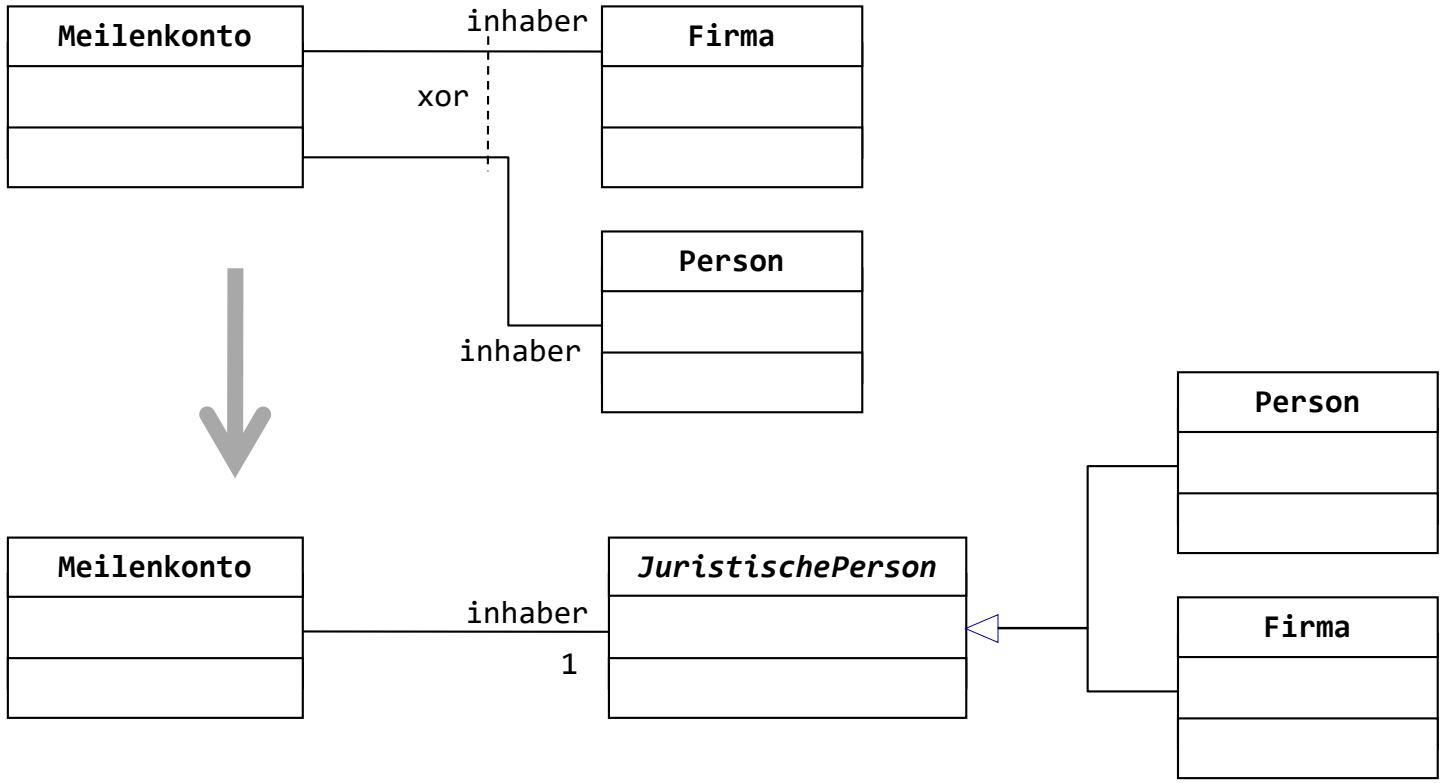
- **`{ordered}`**: Die so markierten Objekte am Ende der Assoziation liegen sortiert vor. Duplikate sind nicht erlaubt.
- **`{bag}`**: Dasselbe Objekt darf am Ende der Assoziation mehrfach erscheinen.
- **`{seq}`** bzw. **`{sequence}`**: Das Ende der Assoziation verweist auf eine geordnete Menge von Objekten. Duplikate sind erlaubt.



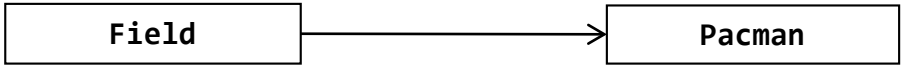
- **Einschränkung** (Teilnahme eines Objektes an einer Assoziation angeben).
  - Einschränkungstypen:
    - **{or}**: Das Objekt muss eine der beiden Assoziationen verwenden.
    - **{xor}**: Das Objekt darf nur eine der beiden Assoziationen verwenden.
    - **{and}**: Das Objekt muss an beiden Assoziationen teilnehmen → lässt sich besser durch Kardinalität 1 ausdrücken.
  - Beispiel für ein Meilenkonto bei einer Fluggesellschaft:



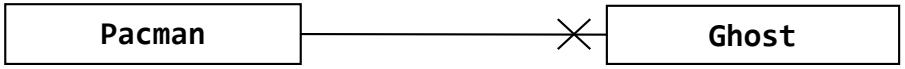
- Implementierung einer **xor**-Einschränkung:
  - Programmgesteuert überprüfen
  - Durch Vererbung und Einführung einer weiteren Klasse:



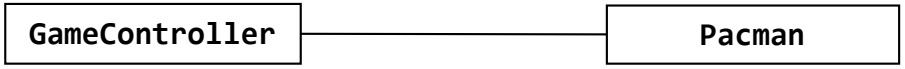
- **Navigierbarkeit** (erlaubt die Angabe, in welche Richtung eine Assoziation gelesen wird). Es werden folgende Arten unterstützt:
  - Navigierbar: Die Klasse **`Field`** kennt die Klasse **`Pacman`** am Ende der Assoziation.



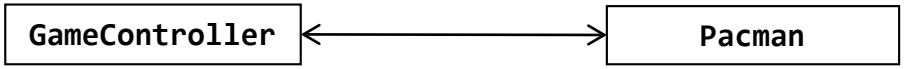
  - Nicht navigierbar: Die Klasse **`Pacman`** kennt die Klasse **`Ghost`** am Ende der Assoziation nicht.



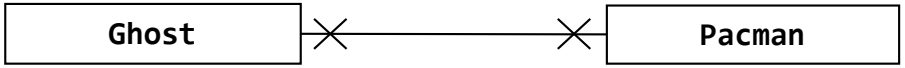
  - Unspezifiziert: Es wird keine Aussage über die Navigierbarkeit getroffen. In der Praxis wird das häufig als navigierbar interpretiert.



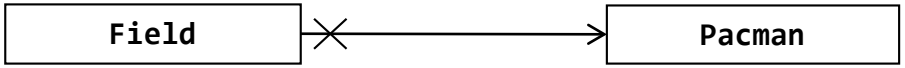
  - Bidirektionale Navigierbarkeit: Beide Klassen kennen sich gegenseitig und können so auch jeweils die Methoden des Anderen aufrufen.



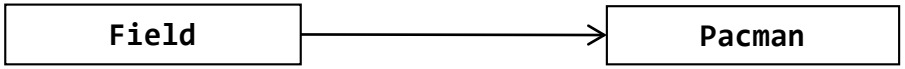
- Verbot der Navigierbarkeit: Beide Klasse dürfen sich nicht kennen. Es besteht zwar eine logische Beziehung, die aber nicht durch eine Assoziation ausgedrückt werden soll → nicht sehr gebräuchlich.



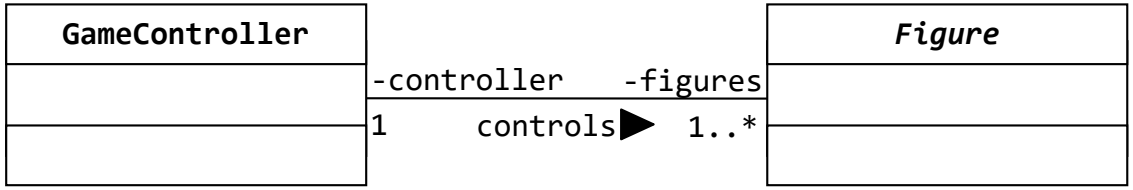
- Unidirektionale Navigierbarkeit: Die Navigation ist in nur einer Richtung erlaubt.



- Teilweise Spezifikation der Navigation: Es ist offen gelassen, ob **`Pacman`** auch das **`Field`** kennt.



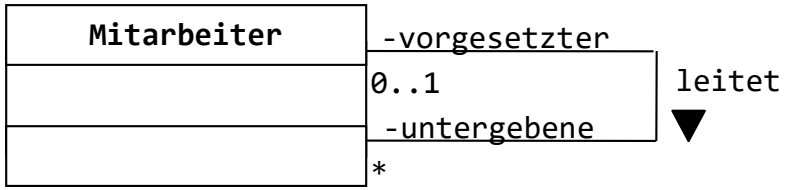
- Implementierung einer Assoziation:




```
public class GameController {
  private ArrayList<Figure> figures;
  // ...
}
```

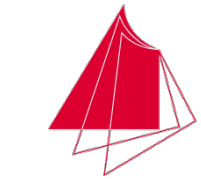
```
public abstract class Figure {
  private GameController controller;
  // ...
}
```

- **Reflexive Assoziation**: Eine Assoziation einer Klasse zu sich selbst, ansonsten handelt es sich um eine normale Assoziation.



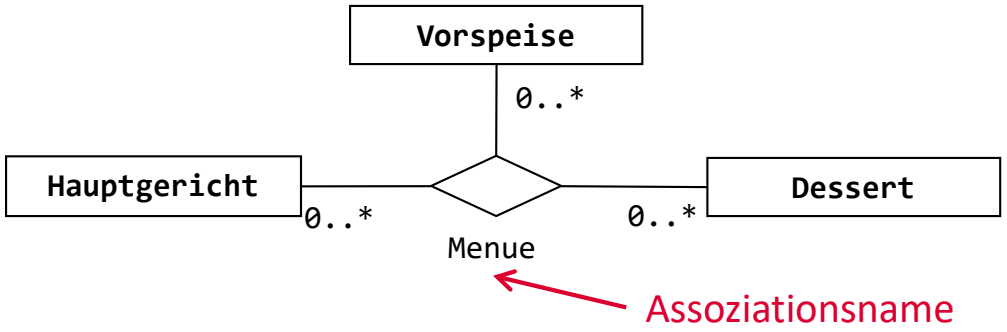
  - Der Vorgesetzte hat mindestens einen Untergebenen.
  - Jeder Untergebene hat maximal einen Vorgesetzten. Jeder Vorgesetzte kann also auch wieder Mitarbeiter sein.
  - Implementierungsmöglichkeit:

```
public class Mitarbeiter {
  private Mitarbeiter vorgesetzter;
  private ArrayList<Mitarbeiter> untergebene;

  // ...
}
```



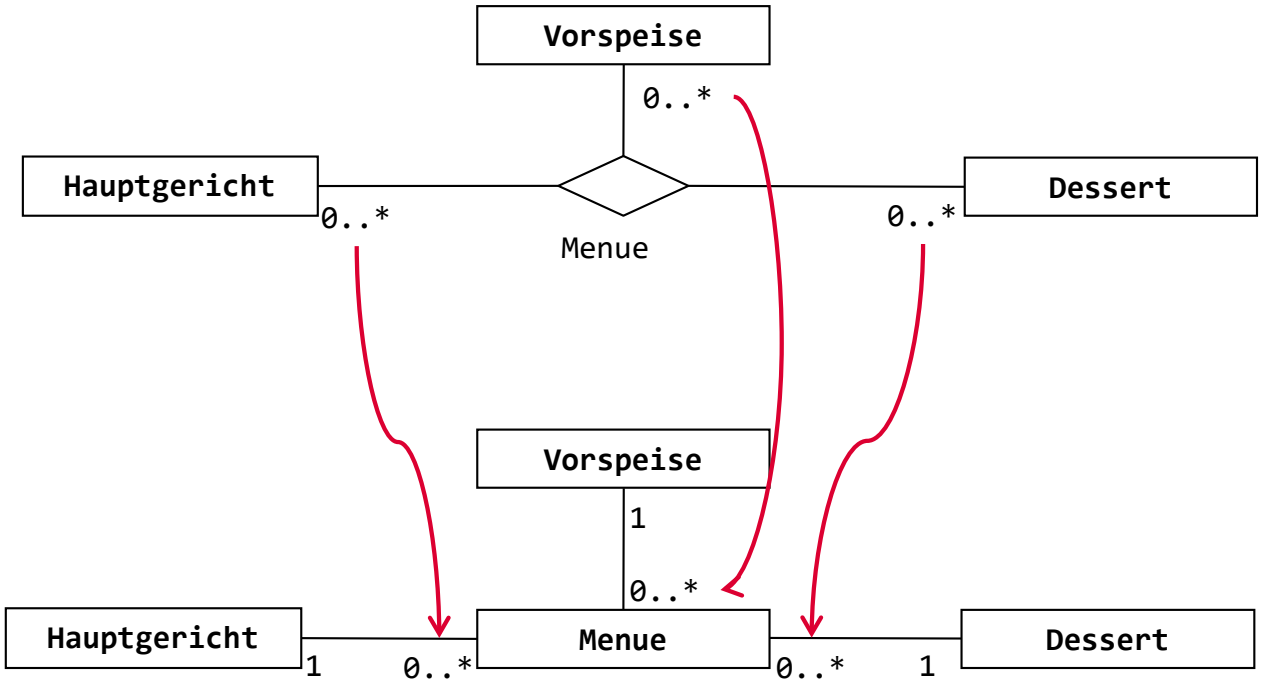
N-äre Assoziation

Eine n-äre Assoziation ist die allgemeine Form einer Assoziation zwischen n Klassen.

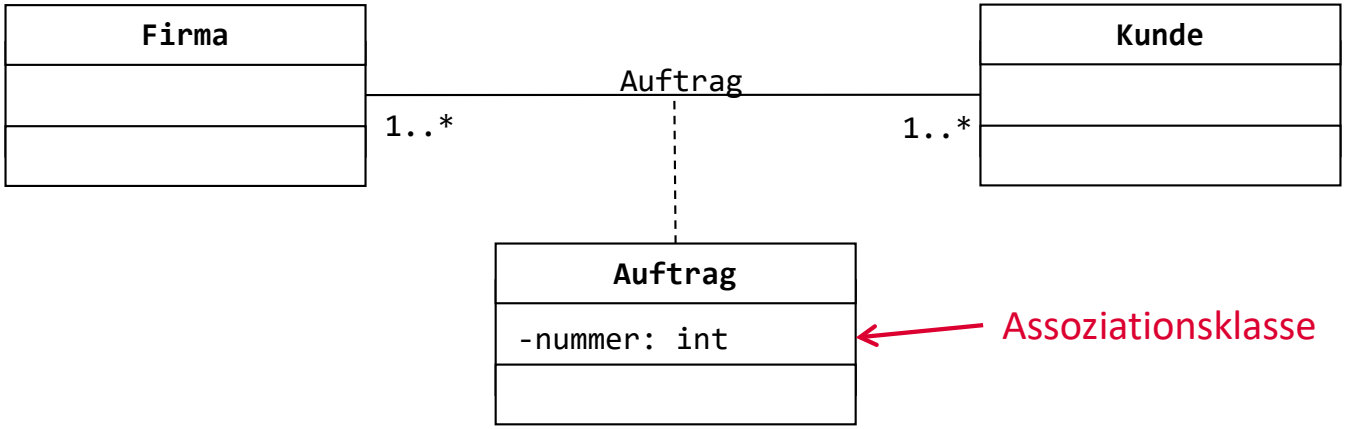


- Erklärung des Beispiels:
  - Die drei Klassen stehen in einer Beziehung namens **Menue**.
  - Nicht jedes Hauptgericht muss in einem Menü vorkommen (Multiplizität **0**).
  - Jedes Hauptgericht kann in beliebig vielen Menüs vorkommen (Multiplizität *****).
  - Wenn ein Hauptgericht in mindestens einem Menü vorhanden sein muss, würde man am Hauptgericht die Multiplizität **1..*** eintragen.

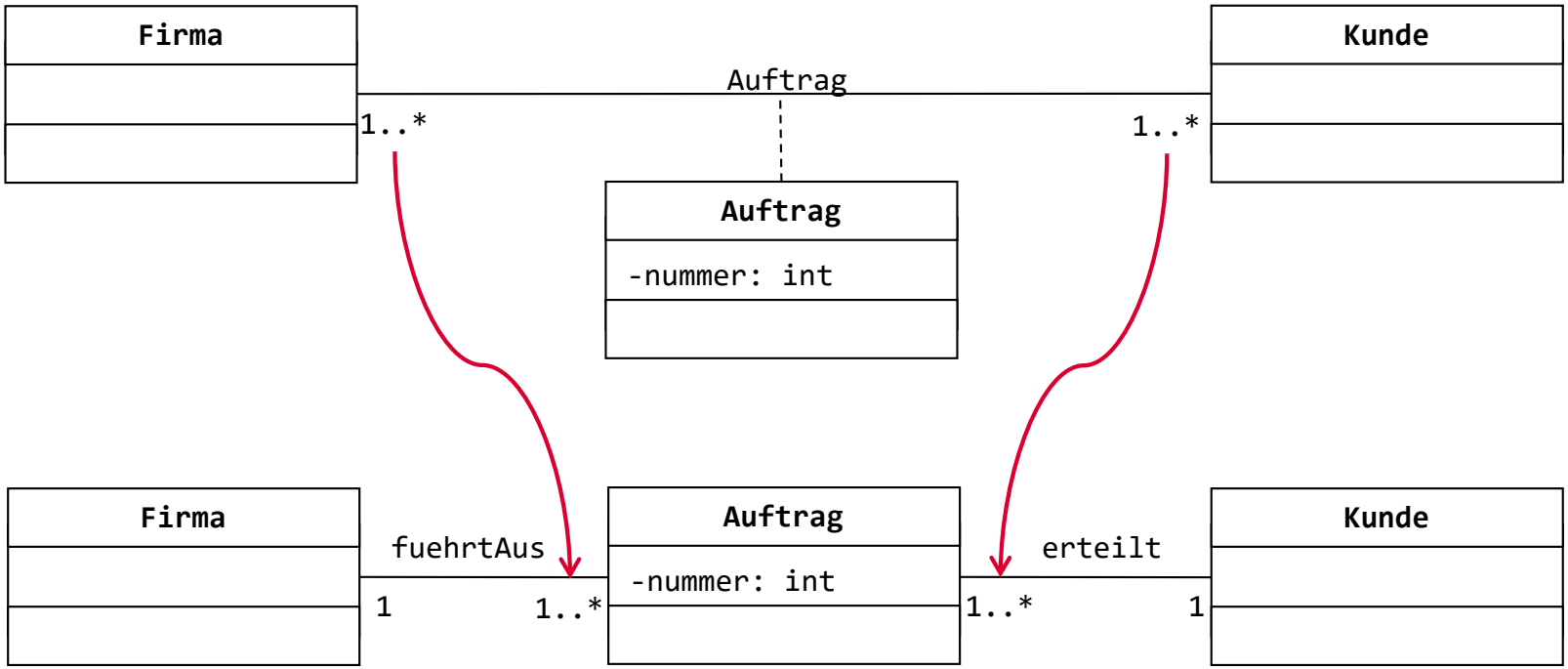
- Wie kann eine n-äre Assoziation implementiert werden?
- Ganz einfach durch Umsetzung in mehrere binäre Assoziationen:



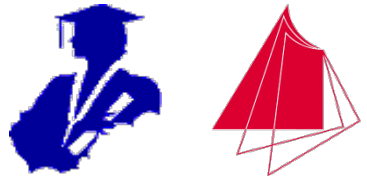
- **Assoziationsklasse**: Eine Assoziationsklasse ist eine Assoziation, die neben Attributen auch Operationen wie bei einer Klasse aufnehmen kann:
  - Die Namen der Assoziation und ihrer Klasse müssen identisch sein.
  - Einsatz: Zusätzliche Attribute können logisch keinem der Enden der Assoziation zugeordnet werden → Ablage in der Assoziationsklasse.
- Beispiel: Die Auftragsnummer kann weder der Firma noch dem Kunden zugeordnet werden.



- Implementierung: Weder C++ noch Java kennen Assoziationsklassen → Einführung einer „Zwischenklasse“:



- Der Kunde erteilt beliebig viele Aufträge.
- Eine Firma kann beliebig viele Aufträge ausführen.



- Implementierung einer Assoziationsklasse:

```
public class Auftrag {
  private int nummer;
  private Kunde kunde;
  private Firma firma;

  // ...
}
```

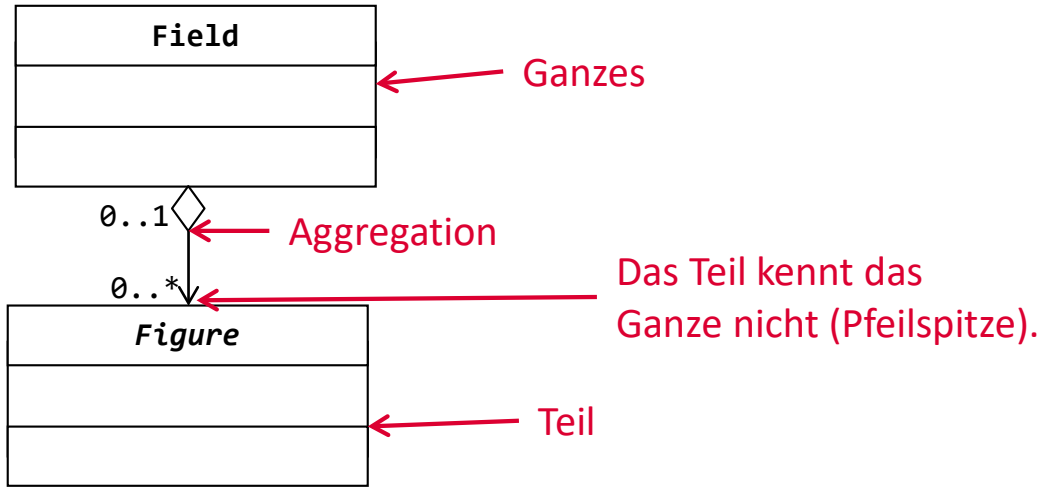
```
public class Firma {
  private ArrayList<Auftrag> auftraege;

  // ...
}
```

```
public class Kunde {
  private ArrayList<Auftrag> auftraege;

  // ...
}
```

Definition: Aggregation

Eine Aggregation ist eine spezielle Form einer binären Assoziation. Sie beschreibt eine Teil-Ganzes-Beziehung zwischen genau zwei Klassen.

- Beispiel:



Klassendiagramme

Beziehungen zwischen Klassen: Aggregation

- Eigenschaften einer Aggregation:
  - Die Lebensdauer des Ganzen ist von der Lebensdauer der Teile unabhängig.
  - Die Lebensdauer der Teile ist von der Lebensdauer des Ganzen unabhängig.
  - Teile können in mehreren Ganzen gleichzeitig verwendet werden.
  - Die Teile kennen das Ganze häufig nicht.
- Einsatzgebiete für Aggregation:
  - Das Ganze handelt als Stellvertreter für seine Teile.
  - Es nimmt Aufträge entgegen und delegiert diese an die Teile.
  - Beispiel: Pacman
    - Ganzes: Spielfeld
    - Teile: Einzelne Figuren in der Zeichenfläche
    - Auftrag: Anforderung, sich neu zu zeichnen
    - Delegation: Jedes Teil erhält den Auftrag, sich selbst neu zu zeichnen.

- Implementierung einer Aggregation:

```
public class Field {
  private ArrayList<Figure> figures;

  // ...
}
```

```
public abstract class Figure {


  // ...
}
```

- Hinweise:
  - Das Ganze erzeugt und löscht die Teile i.d.R. nicht selbst.
  - Das Ganze besitzt häufig Methoden zum Hinzufügen und Entfernen von Teilen sowie zum Besuchen aller Teile.

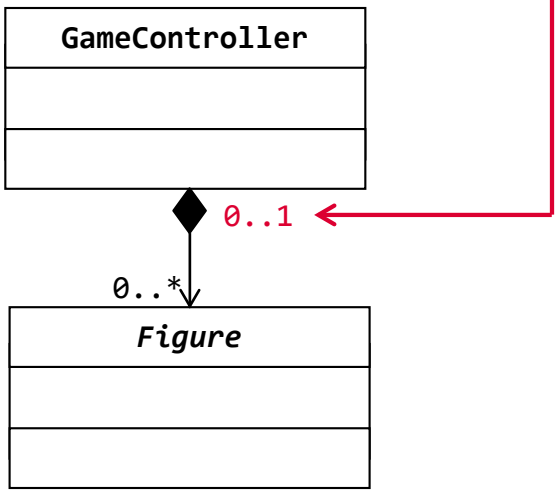
Definition: Komposition

Eine Komposition ist eine starke Form der Aggregation. Hier sind das Ganze und seine Teile untrennbar miteinander verbunden.

- Beispiel:

- Eigenschaften einer Komposition:
  - Die Lebensdauer des Ganzen ist von der Lebensdauer der Teile unabhängig. Beim Löschen eines Teils bleibt das Ganze bestehen.
  - Die Lebensdauer der Teile ist von der Lebensdauer des Ganzen abhängig: Beim Löschen des Ganzen werden die Teile auch gelöscht.
  - Teile können nicht in mehreren Ganzen gleichzeitig verwendet werden.
  - Die Teile kennen das Ganze häufig nicht.
- Einsatzgebiete für Komposition → siehe Aggregation

- Spezialfall der Komposition: Multiplizität **`0..1`**



- Figuren dürfen außerhalb und ohne Bezug zum Controller erzeugt werden.
- Sie werden dann als Teile am Controller registriert und gehören ab diesem Zeitpunkt zum Controller.
- Figuren dürften auch an einen anderen Controller übertragen werden.

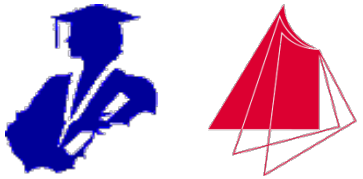
- Implementierung einer Komposition (in Java wie bei einer Aggregation):

```
public class GameController {
  private ArrayList<Figure> figures;

  // ...
}
```

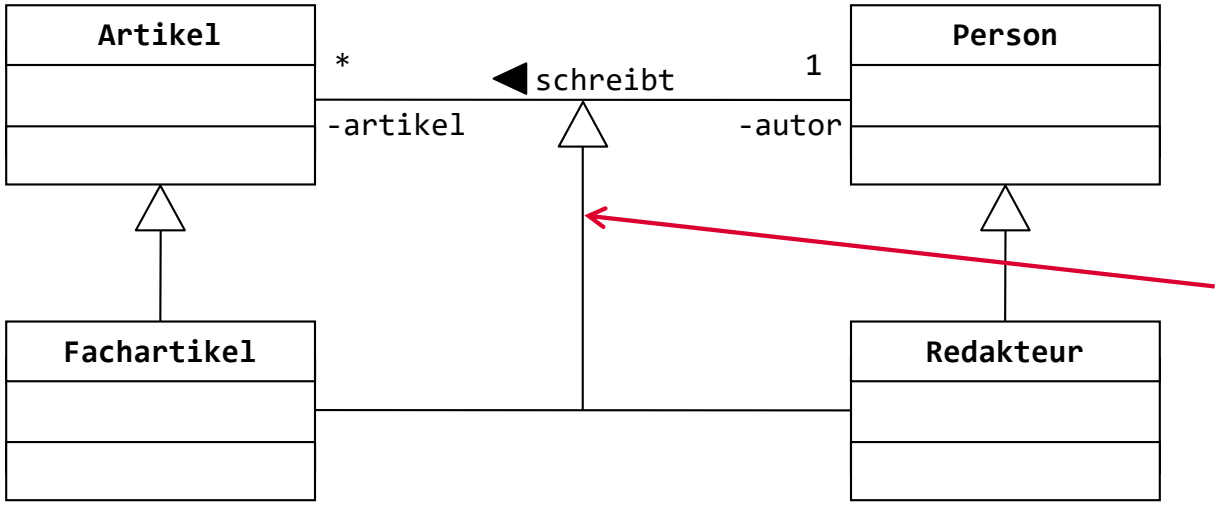
```
public abstract class Figure {


  // ...
}
```

- Hinweise:
  - Häufig ist es sinnvoll, dass das Ganze die Teile selbst erzeugt und löscht: Die Teile können nicht so versehentlich auch ohne das Ganze verwendet werden.
  - Das Ganze besitzt i.d.R. Methoden zum Erzeugen und Entfernen von Teilen sowie zum Besuchen aller Teile.

- Wann wird ein bestimmter Beziehungstyp verwendet?
  - Existenzabhängige Teil-Ganzes-Beziehung: Komposition
  - Logische Einheit, die nicht existenzabhängig ist: Aggregation
  - Kaskadierende Methodenaufrufe: Aggregation (z.B. Pacman-Spiel, in dem die Spielfläche die Aufrufe zum Neuzeichnen an die Figuren weiter leitet)
  - Rumbaugh: „Think of Aggregation as a modeling placebo“.



- Die Vererbung von Assoziationen wird meist in folgenden Szenarien verwendet:
  - Es besteht bereits eine Assoziation zwischen zwei Basisklassen.
  - Es soll ausgedrückt werden, dass die Assoziation einer abgeleiteten Klasse immer nur mit der Assoziation einer anderen abgeleiteten Klasse hergestellt werden soll.
- Beispiel:

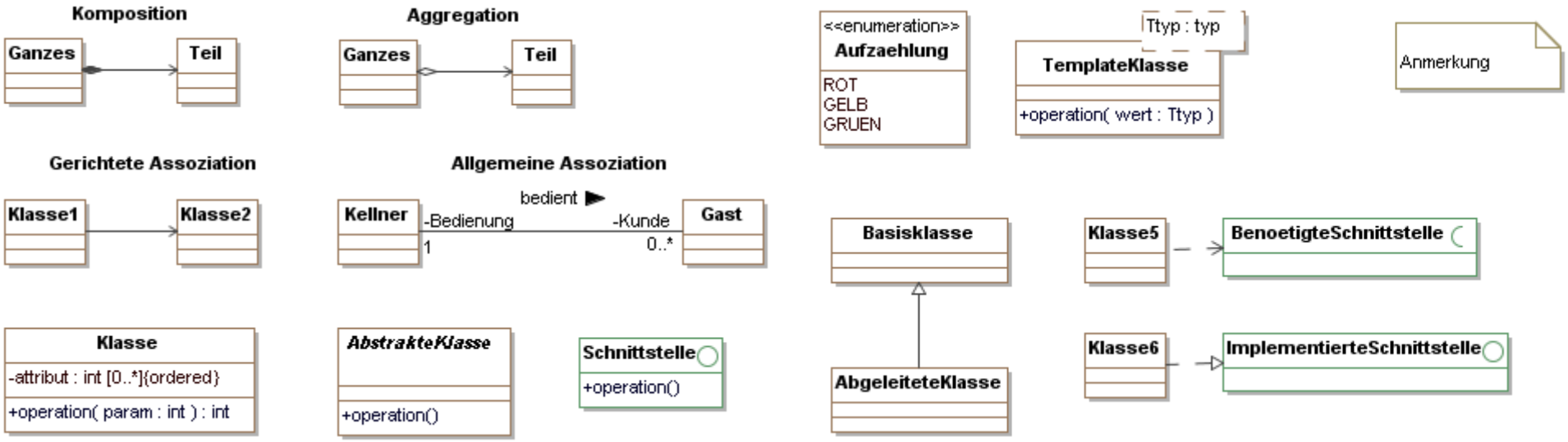


Nur der Redakteur kann Fachartikel schreiben → Eine Assoziation kann z.B. nicht von einer „normalen“ Person zum Fachartikel führen.

Klassendiagramme

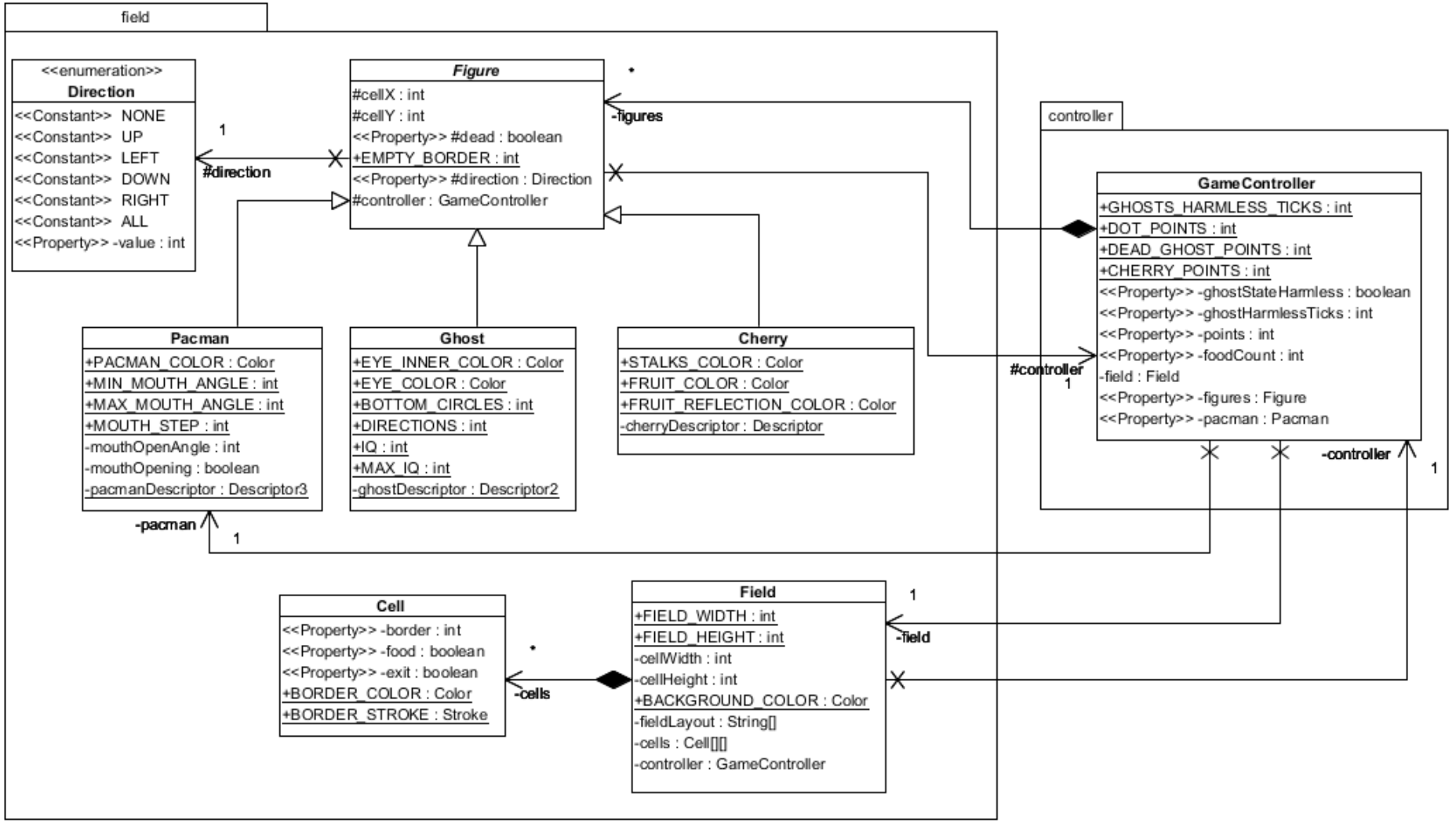
Übersicht zu Klassendiagramm (vereinfacht)

Darstellung der wichtigsten Elemente durch MagicDraw:

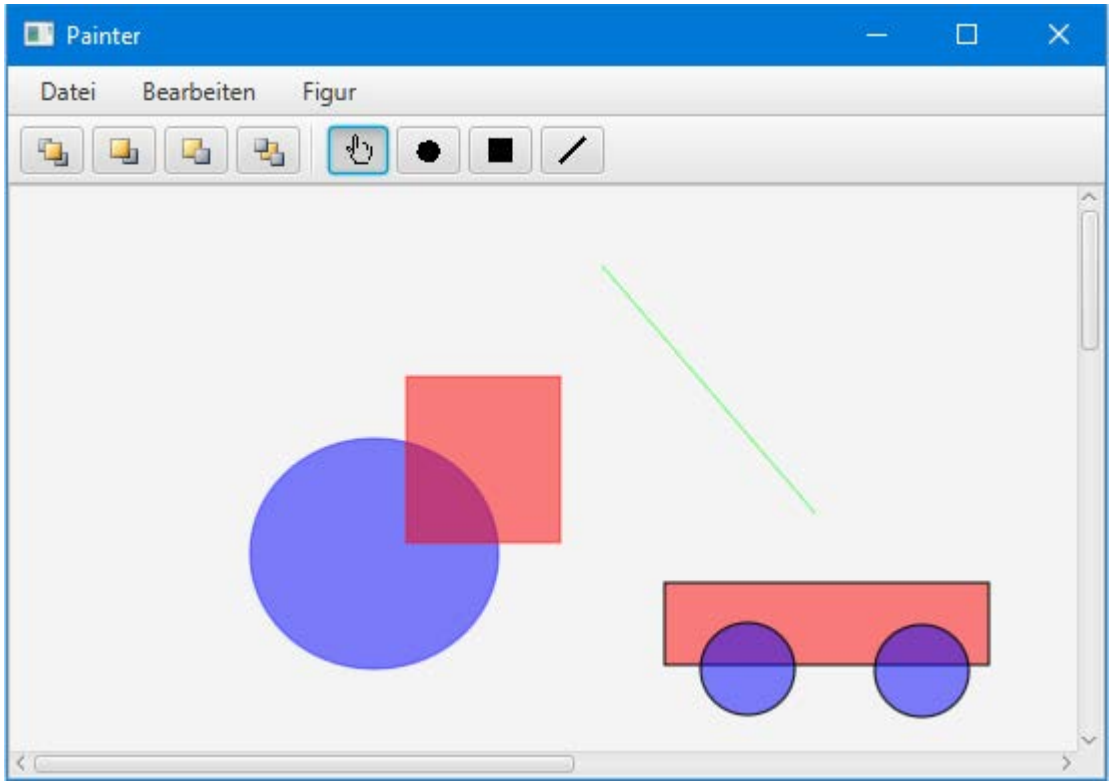


Klassendiagramme

Klassen im Pacman-Spiel (unvollständig)



- Zeichenprogramm
  - **Figuren**: Rechteck, Oval, Linie
  - **Gruppen**: Sind Figuren und bestehen aus Figuren



Klassendiagramme

Beispiel aus der Klausur WS 2012/2013

- Fiktives Flugbuchungssystem:
  - **Kunde**: Name, Anschrift, E-Mail-Adresse, Angabe (privat/geschäftlich), Login-Name, Passwort
  - **Flug**: Flugnummer, Datum, Uhrzeit, Start- und Endflughafen, Gebühr, abhängig vom Startflughafen, Preis, abhängig von der Klasse (1. Klasse, Business-Klasse, 2. Klasse). Es werden alle Sitze einer Klasse zum selben Preis angeboten.
  - **Buchung**: Jeder Kunde kann beliebig viele Flugbuchungen gleichzeitig vornehmen. Eine Buchung kann mehrere Sitzplätze in derselben Klasse umfassen. Die Zahlungsart müssen Sie nicht berücksichtigen.
  - **Reservierung**: Ist sich ein Kunde noch nicht ganz sicher, ob er einen Flug buchen möchte, dann kann er sich diesen für einen Zeitraum von 24 Stunden kostenlos reservieren.
  - **Zusatzoptionen** bei der Buchung: Zu einem Flug können optionale Leistungen hinzugebucht werden, für die aber zusätzliche Kosten anfallen (Reiserücktrittsversicherung, Sitzplatz am Notausstieg, Übergepäck, eventuell weitere (je nach Fluggesellschaft und Flugzeugtyp)).

Fehlerbehandlung mit Ausnahmen

Übersicht

Java und UML | Schritte, Software | Objekt-orientierung | Klassendiagramme | Ausnahmen | Funktionale Programmierung | Weitere Beispiele

Klassen | Überladen | Vererbung | Generische Klassen | Aufzähl-typen | Regeln

Über-schreiben

Typinfo., I/O | Annota-tionen | Laufzeit-typinfo. | Ein-, Ausgabe

Entwurf | Prinzipien | Verbindung von Modulen | Spring

Fehlerbehandlung mit Ausnahmen

Motivation

- *Welche Arten von Fehlern gibt es?*
- *Wie können Laufzeitfehler sicher gefunden werden?*
- *Wie sollten Laufzeitfehler einer Methode dem Aufrufer mitgeteilt werden?*

Fehlerbehandlung mit Ausnahmen

Fehlerklassifikation

- **Erwartete Fehler** durch Aufruf einer prinzipiell unsicheren Methode
  - Auftreten: Es muss bei jedem Aufruf mit dem Fehler gerechnet werden.
  - Beispiel: Versuch, eine Datei zum Lesen zu öffnen, die nicht existiert
  - Vermeidung: nicht möglich
  - Behandlung: ja
- **Unerwartete Fehler** durch Programmierfehler oder mangelnde Systemressourcen
  - Auftreten: Eigentlich sollte der Fehler nie auftreten.
  - Beispiel: zu wenig Hauptspeicher vorhanden, Stacküberlauf
  - Vermeidung: nur begrenzt möglich (intensive Programmtests)
  - Behandlung: nur begrenzt möglich (z.B. Datensicherung vor der Programmbeendigung)

- **Sonderfall: Erwarteter oder unerwarteter Fehler während der Fehlerbehandlung**
  - Auftreten: siehe erwarteter und unerwarteter Fehler
  - Beispiel: Die Fehlermeldung kann wegen fehlender Schreibrechte oder zu geringen freien Hauptspeichers nicht in eine Datei geschrieben werden.
  - Vermeidung: nur bei sehr einfacher Fehlerbehandlung möglich
  - Behandlung: hängt vom Fehler ab

Fehlerbehandlung mit Ausnahmen

Reaktionen auf Fehler

- **Sofort behandelbarer Fehler**
  - Auswirkung: Das Programm kann auf den Fehler reagieren und seine Arbeit (eventuell eingeschränkt) fortsetzen.
  - Reaktion: ignorieren (wenn sinnvoll möglich), alternativen Programmablauf starten
  - Beispiel: Bei fehlender Eingabedatei werden immer Standardwerte angenommen.
- **Nicht sofort behandelbarer Fehler**
  - Auswirkung: Der Fehler kann an der Stelle, an der er auftritt, nicht behandelt werden.
  - Reaktion: Fehlermeldung an den Aufrufer der Methode
  - Beispiel: Datei existiert nicht → sollen Standardwerte verwendet werden/soll der Benutzer gefragt werden? Kann eventuell an der Stelle des Fehlers nicht entschieden werden.
- **Nicht sinnvoll behandelbarer Fehler**
  - Auswirkung: sinnvolle Programmfortsetzung nicht möglich
  - Reaktion: i.d.R. Programmbeendigung mit vorheriger Datensicherung (sofern möglich)
  - Beispiel: Stacküberlauf, schwere Programmierfehler

Fehlerbehandlung mit Ausnahmen

Exceptions – Idee

- Fehlerbehandlung bisher:
  - Ein Methode `m1` stellt einen Fehler fest und liefert ein Ergebnis, das diesen Fehler beschreibt.
  - Die aufrufende Methode `m2` der Methode `m1` stellt fest, dass `m1` einen Fehler zurückliefert und beendet die eigene Arbeit mit der Rückgabe eines Fehlers.
  - usw...
- Schlechte Lösung, da sich eventuell große Teile einer Methode mit der Fehlerbehandlung befassen. Eine Trennung von Fehler- und Normalfall ist nicht vorhanden.
- Fehler werden häufig über viele Aufrufebenen hinweg nach „oben“ weitergereicht, ohne dass die Fehler direkt bearbeitet werden.
- Java bietet das Konzept der Ausnahme (Exception) zur besseren Fehlerbehandlung. Damit muss nicht der Rückgabewert einer Methode zur Fehlerübermittlung missbraucht werden.

Einsatzgebiete

- Meldung von Fehlern, die nicht lokal am Ort der Entstehung behoben werden konnten:
  - Beispielsweise kann der Autor einer Bibliothek Laufzeitfehler („Datei existiert nicht“ etc.) zwar erkennen, er ist aber selten in der Lage, diese im Kontext des aufrufenden Programms richtig zu behandeln.
  - Der Anwender der Bibliothek dagegen weiß, wie er mit den Fehlern umgehen soll.
- Die Behandlung von anderswo gefundenen Fehlern.
- Syntax:

```
try {
  // versuche, eine Aufgabe zu lösen
}
catch (Fehlerklasse fehler) {
  // behandle Fehler des Typs "Fehlerklasse"
}
```

Fehlerbehandlung mit Ausnahmen

Exceptions – Idee

- Beispiel (Zugriff auf den Vektor, angelehnt an **`ArrayList`**):

```
public class Vector<E> {
  private E[] values;
  private int size;
  // ...
  public E getValue(int index) {
    if (index >= size || index < 0)
      throw new IndexOutOfBoundsException("Index: " + index + ", Size: " + size);
    return values[ index ];
  }
}
```

```
public static void main(String[] args) {
  Vector<Double> vector = new Vector<>(3, 0.0);
  try {
    System.out.println(vector.getValue(0));
    System.out.println(vector.getValue(4));
  }
  catch (IndexOutOfBoundsException ex) {
    System.err.println("Oops" + ex.getMessage());
  }
}
```

Behandlung mehrerer Ausnahmen

- Syntax:

```
try  {
  // versuche, eine Aufgabe zu lösen
}
catch (FehlerTyp1 ex1)  {
  // behandle Fehler-Typ1
}
catch (FehlerTyp2 ex2) {
  // behandle Fehler-Typ2
}
```

- Unbehandelte Ausnahmen:
  - Beendigung der Methode
  - Auslösung der Ausnahme in der aufrufenden Methode
  - Dieses Spiel funktioniert solange, bis eine Methode die Ausnahme abfängt oder das Programm beendet wurde.

Fehlerbehandlung mit Ausnahmen

Mehrere Exceptions

- Es dürfen auch mehrere, unabhängige Fehler durch ein sogenanntes „multi-catch“ abgefangen werden:
- Syntax:

```
try  {
  // versuche, eine Aufgabe zu lösen
}
catch (FehlerTyp1 | FehlerTyp2 ex) {
  // behandle Fehler-Typ1 und Fehler-Typ 2
}
```

Fehlerbehandlung mit Ausnahmen

Mehrere Exceptions

- Ausnahmen werden in den **`catch`**-Blöcken in der Reihenfolge ihres Auftretens ausgewertet.
- Abfangen aller möglichen Ausnahmen:

```
try {
  // versuche, eine Aufgabe zu lösen
}
catch (FehlerTyp1 ex) {
  // behandle Fehler-Typ1
}
catch (Throwable thr) { // Alle unbehandelten Fehler
  // behandle alle anderen Fehler
}
```

- Ausnahmen können auch im **`catch`**-Block ausgelöst werden:

```
try {
  // versuche, eine Aufgabe zu lösen
}
catch (FehlerTyp1 ex) {
  // behandle Fehler-Typ1
  throw ex;    // Fehler weitermelden
}
```

Fehlerbehandlung mit Ausnahmen

Exceptions mit Klassenhierarchien

- Exception-Klassen können ganze Klassenhierarchien bilden.
- Im **`catch`**-Block kann auch die Basisklasse einer Ausnahme angegeben werden, um eine Ausnahme einer abgeleiteten Klasse abzufangen.
- Beispiel:

```
public class FileNotFoundException extends IOException {
  // ...
}
```

- Eine **`FileNotFoundException`** kann jetzt auch durch die Angabe ihrer Basisklasse abgefangen werden:

```
public void m() {
  try {
    // …
    throw new FileNotFoundException();
  }
  catch (IOException ex) {
    // …
  }
}
```

- Es können auch mehrere **`catch`**-Blöcke vorhanden sein, die sowohl die abgeleitete Klasse als auch die Basisklasse der Ausnahme abfangen. Achtung Reihenfolge:

```
public void m() {
  try {
    // ...
    throw new FileNotFoundException();
  }
  catch (IOException ex1) {
    // ...
  }
  catch (Throwable thr) {
    // ...
  }
}
```

- Wird erst **`Throwable`** abgefangen, dann wird der **`catch`**-Block von **`IOException`** nie betreten.

Fehlerbehandlung mit Ausnahmen

Exceptions: Hinweise

- Generelle Ausräumarbeiten nach einem erfolgreichem **`try`** oder einer Ausnahme mit **`finally`**. Beispiel:

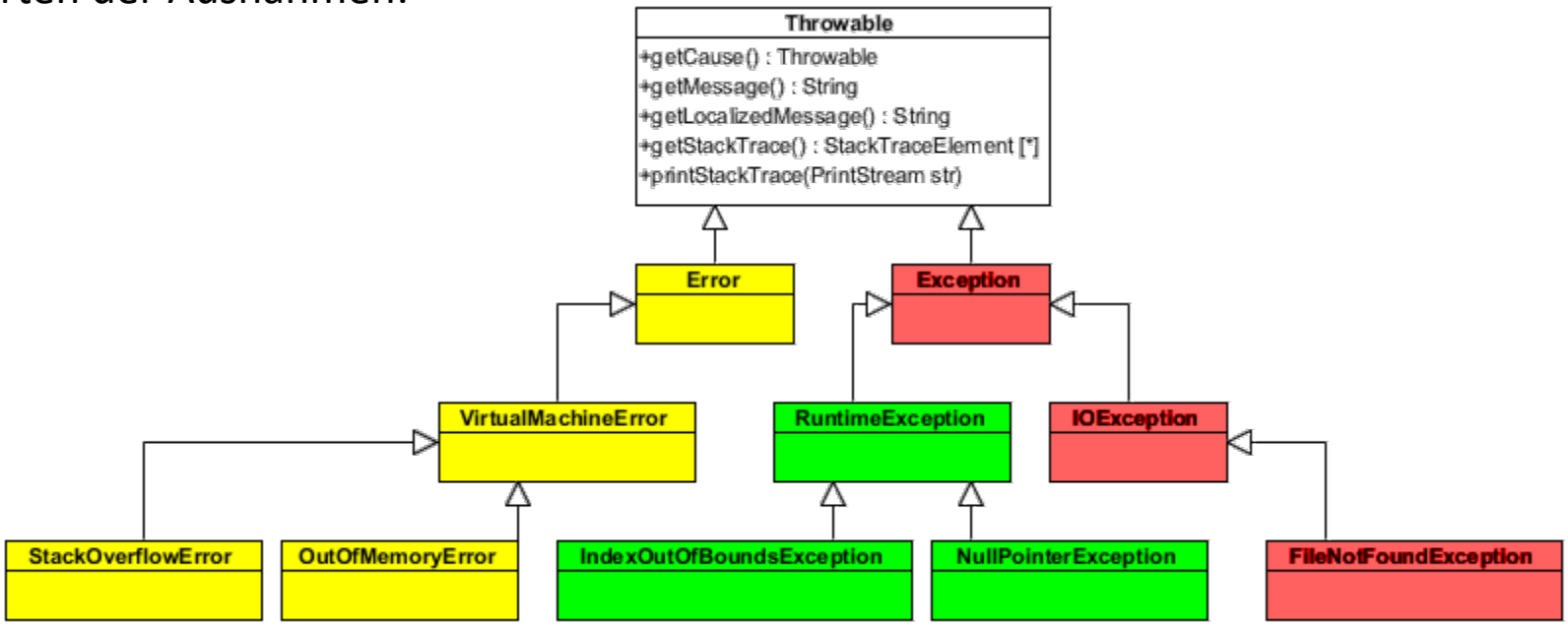
```
public void m() {
  try {
    // ...
    throw new FileNotFoundException();
  }
  catch (IOException ex1) {
    // ...
  }
  finally {
    // Wird immer betreten
  }
}
```

- Ein „echtes“ Beispiel folgt im nächsten Kapitel.
- Seit Java 7 gibt es ein erweitertes **`try`**-Konstrukt → siehe Kapitel zur Ein-/Ausgabebehandlung.

Fehlerbehandlung mit Ausnahmen

Exceptions mit Klassenhierarchien

- Arten der Ausnahmen:



- gelb: unerwartete und in der Regel nicht sinnvoll behandelbare Fehler
- grün: unerwartete und in der Regel nicht sinnvoll behandelbare Fehler („Programmierfehler“)
- rot: erwartete und sofort oder später behandelbare Fehler (erben direkt von **`Exception`**, nicht von **`RuntimeException`**)

Fehlerbehandlung mit Ausnahmen

Eigenschaften und Deklaration erwarteter Fehler

- Erwartete und sofort oder später behandelbare Fehler (rot markiert):
  - Einem Anwender einer Methode muss die Art der Ausnahme, die diese auslösen kann, angegeben werden.
  - Syntax:

    `ret-type method-name(params) throws Exceptions`

    Beispiel:

```
public int readFromFile(String name) throws FileNotFoundException,
                                             EOFException {
```

    Beispiel durch Angabe der Basisklasse **`IOException`** (**`FileNotFoundException`** und **`EOFException`** erben davon):

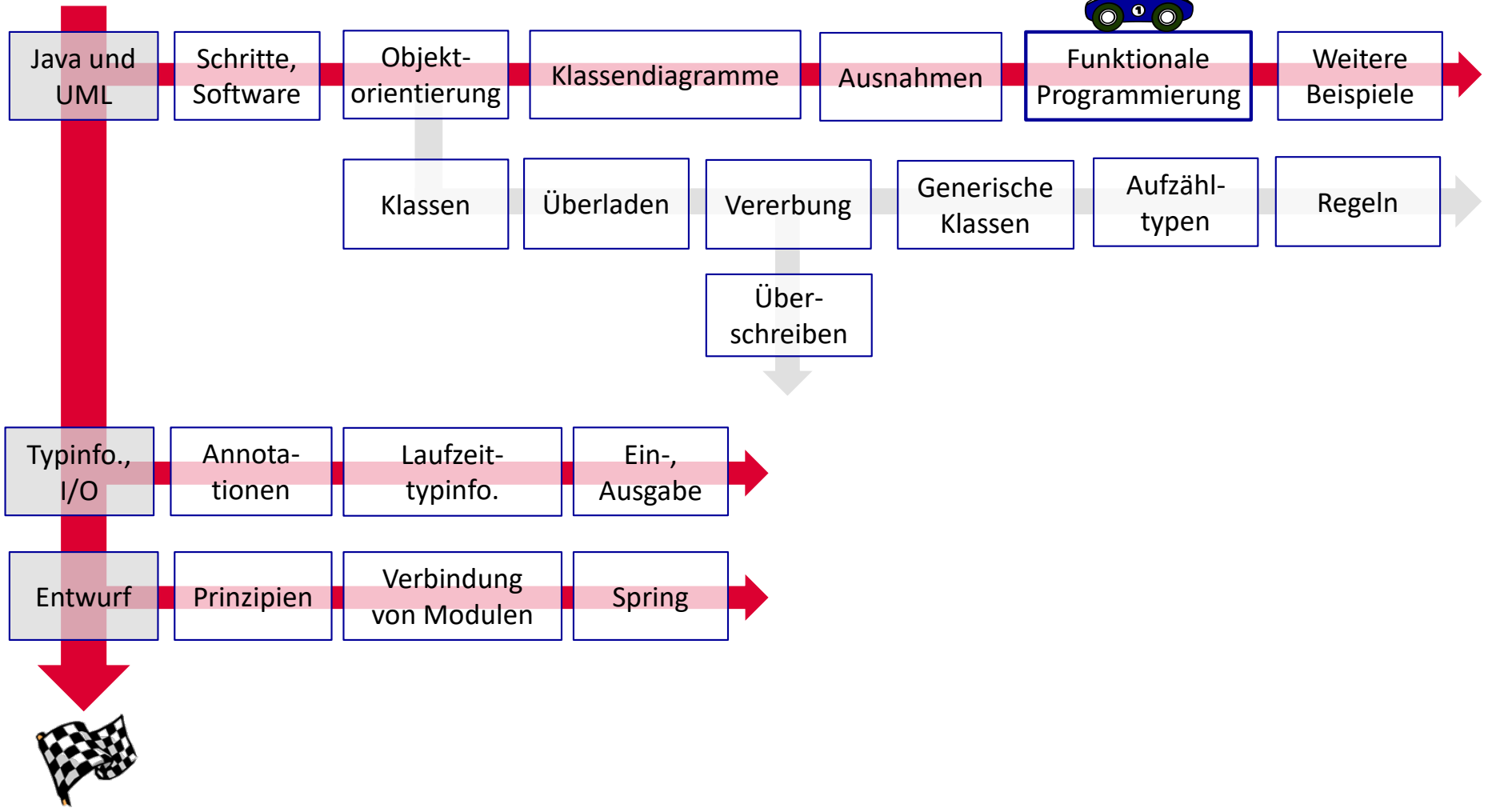
```
public int readFromFile(String name) throws IOException {
```

  - Ohne Spezifikation kann die Methode nur unerwartete oder gar keine Ausnahmen auslösen.

    Beispiel:

```
public int getValue(int index) {
```

Funktionale Programmierung
Übersicht



- Funktionale Programmierung
  - Das Programm besteht u.A. aus einer Anzahl Funktionen, die sich gegenseitig aufrufen können.
  - Syntax in LISP für einen Funktionsaufruf:

```
(Funktionsname Param1 ... ParamN)
(+ 2 4) ; ruft Funktion + auf, die die Summe von 2 und 4 zurückgibt
```

  - Rein funktionale Programmiersprachen:
    - Funktionen besitzen keine Nebeneffekte (keine Manipulation von Zuständen), geben nur Ergebnisse zurück → Funktionen im mathematischen Sinn.
  - Funktionen höherer Ordnung: Funktionen können als Parameter übergeben werden.

- Das Lambda-Kalkül war einer der ersten Ansätze, solche funktionalen Programme zu beschreiben:
  - **Funktionsabstraktion**: λ x.A (anonyme Funktion mit x als Eingabeparameter und Funktionsdefinition A), x hat keinen Typ
  - **Funktionsanwendung**: F A (Funktion F wird auf den Ausdruck A angewendet)
- Beispiele:
  - Identität: λ x.x
  - x^2: λ x.x*x, angewandt auf 3: (λ x.x*x)3 ergibt 9

- Was soll das in einer Java-Einführung????
- Seit Java 8 bietet Java einen recht eleganten Ansatz zur funktionalen Programmierung.

- Rückblick auf den „Vergleicher“ mit der generischen Schnittstelle **`Comparator<T>:`**

```
public class Customer {
  // ...
  private int number;
  // Getter und Setter
}
```

- Vergleicher für zwei Kundenobjekte:

```
public class CustomerNumberComparator implements Comparator<Customer> {
  @Override
  public int compare(Customer c1, Customer c2) {
    return c1.getNumber() - c2.getNumber();
  }
}
```

- Hinweis damals: Es wird eine „Funktion“ in einem Objekt gekapselt.

- Warum kann die Funktion nicht direkt übergeben werden?
  - Die Funktion erwartet zwei Objekte der Klasse **`Customer`**.
  - Die Funktion gibt ein Ergebnis vom Typ **`int`** zurück.
  - Java unterstützt seit Version 8 die Übergabe solcher unbenannter Funktionen, auch Lambdas genannt.
  - Die Funktionen werden zu Methoden in Klassen, die Schnittstellen implementieren.
- Beispiel **`Comparator`**:

```
Customer[] customers;
// customer füllen
```

  - Statt

```
Arrays.sort(customers, new CustomerNumberComparator());
```

  - Einfacher

```
Arrays.sort(customers,
            (c1, c2) -> c1.getNumber() - c2.getNumber());
```

  - Die Klasse **`CustomerNumberComparator`** wird überflüssig!

- Was passiert hier?

Vorhandene Methode

```
sort(T[] arg0,
     Comparator<? super T> arg1)
```

benötigt

Vorhandene Schnittstelle

```
public interface Comparator<T> {
  int compare(T arg0, T arg1);
  // ...
}
```

benötigter Rückgabetyp

benötigte Parameter

Erzeugter Aufruf mit anonymer innerer Klasse

```
sort(customers,
     new Comparator<Customer>() {
       @Override
       public int compare(Customer c1,
                          Customer c2) {
         return c1.getNumber() -
                c2.getNumber();
       }
     });
```

Methodenrumpf

übergebene Parameter

Aufruf

```
sort(customers, (c1, c2) -> c1.getNumber() - c2.getNumber())
```

- Warum benötigen im Aufruf die Parameter **`c1`** und **`c2`** keine Typangabe?

  ```
  sort(customers, (c1, c2) -> c1.getNumber() - c2.getNumber())
  ```

- Der Compiler kann die Typen selbst ermitteln:
  - **`customers`** ist vom Typ **`Customer[]`**.
  - Damit wird der generische Parameter **`T`** in der aufgerufenen Methode eine **`Customer`**-Klasse: **`sort(T[] arg0, Comparator<? super T> arg1)`**
  - Der generische Typ im **`Comparator`** muss also auch **`Customer`** oder eine Basisklasse davon sein.
  - Die **`compare`**-Methode der Schnittstelle **`Comparator`** besitzt zwei Übergabeparameter vom Typ **`T`**, also auch vom Typ **`Customer`**.
  - Damit sind die beiden Parameter **`c1`** und **`c2`** vom Typ **`Customer`**.

- Lambda-Ausdrücke werden dadurch umgesetzt, dass anonyme innere Klassen Schnittstellen implementieren.
- Die Schnittstellen dürfen nur eine abstrakte Methode besitzen. Weitere `default`- oder statische Methoden sind erlaubt.
- Solche Schnittstellen werden auch **funktionale Schnittstellen** (functional interfaces) genannt:
  - Sie erlauben die Übergabe einer „Funktion“.
  - Sie können mit der Annotation `@FunctionalInterface` markiert werden.
  - Im Paket `java.util.function` gibt es bereits sehr viele solcher Schnittstellen.
- Die übergebenen Objekte anonymer innerer Klassen werden als **Funktions-Objekte** (functional objects) oder **Funktoren** (functors) bezeichnet → Code als Objekt!

- Welche Schnittstelle wird nun zum Sortieren verwendet?
  - Sie muss zwei Parameter desselben Typs akzeptieren und einen **`int`**-Wert als Ergebniszurückgeben → generische Schnittstelle.
- Beispiel: Namen anhand ihrer Länge sortieren (Wiederholung **`Comparator`**):

```
String[] names = {"Z", "D", "Y", "B"};
Arrays.sort(names, (n1, n2) -> n1.length() - n2.length());
```

- Oder etwas länger:

```
Comparator<String> comp = (o1, o2) -> o1.length() - o2.length();
Arrays.sort(names, comp);
```

- Weiteres Beispiel: Durchlaufen einer Datenstruktur (kommt später noch genauer).
  - „Klassisch“:

```
ArrayList<String> names = new ArrayList<>();
// füllen
for (int i = 0; i < names.size(); i++) {
  System.out.println(names.get(i));
}
```

  - Immer derselbe Aufbau: Schleife, mit dem eigentlichen Inhalt
  - Warum nicht den eigentlichen Inhalt als Funktion übergeben?

```
names.forEach(n -> System.out.println(n));
```

  - **`forEach`** erwartet ein Objekt, dessen Klasse die Schnittstelle **`Consumer<T>`** implementiert, vereinfacht:

```
@FunctionalInterface
public interface Consumer<T> {
  void accept(T t);
}
```

  - **`Comsumer`**: erwartet einen Wert, gibt nichts zurück („konsumiert“)

- Es geht noch einfacher mit einer Methodenreferenz:

```
names.forEach(System.out::println);
```

- `::` leitet einen Verweis auf eine Methode (hier `println`) ein.
- Der Compiler untersucht, ob die Methode einen Wert des Typs `String` (`names` ist eine `ArrayList<String>`) erwartet und erzeugt automatisch ein Objekt der inneren Klasse.

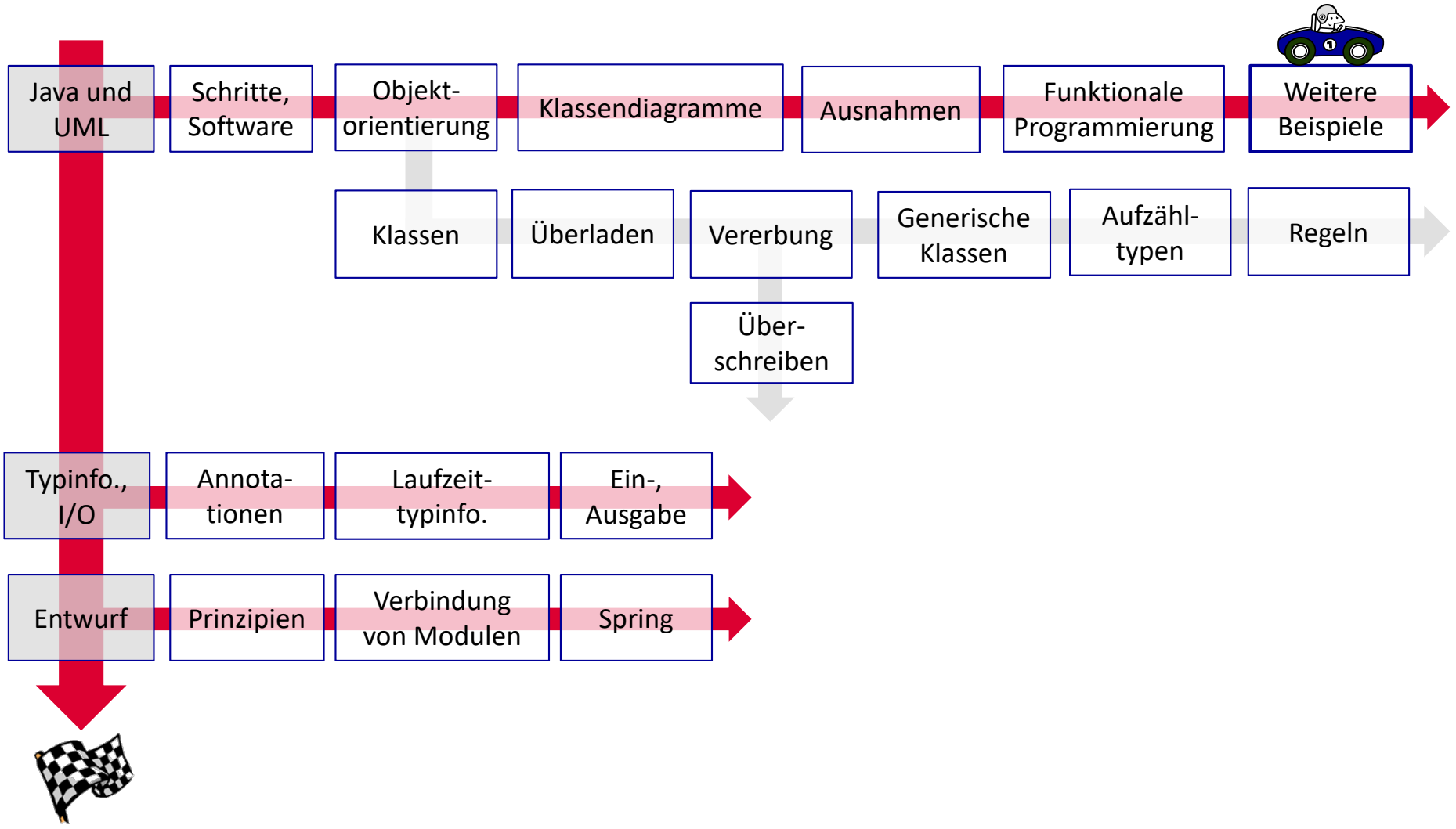
- Hinweise zur Syntax:
  - `(int x, int y) -> x + y`:
    - Typangaben sind nur erforderlich, wenn der Compiler sie nicht selbst ermitteln kann.
    - Bei mehr als einem Parameter sind Klammern links erforderlich.
  - `(x, y) -> x + y`: Der Compiler kann die Typen von `x` und `y` selbst bestimmen.
  - `x -> 2 * x`: Bei einem Parameter sind die Klammern nicht erforderlich.
  - `() -> 42`: Ohne Parameter müssen leere Klammern („burger“) gesetzt werden.
  - `System.out::println`: Methodenreferenz als Kurzform für

    ```
    x -> System.out.println(x)
    ```

  - `Customer::new`: Referenz auf einen Konstruktor

- In vielen Datenstrukturen müssen die Lamda-Ausdrücke frei von Nebeneffekten sein:
  - Sie dürfen die Datenstruktur, auf der sie aufgerufen werden, nicht verändern.
  - Sie dürfen keine anderen Daten verändern.
- Viele weitere Beispiele zu Lambda-Ausdrücken kommen in den Kapiteln zu Datenstrukturen.

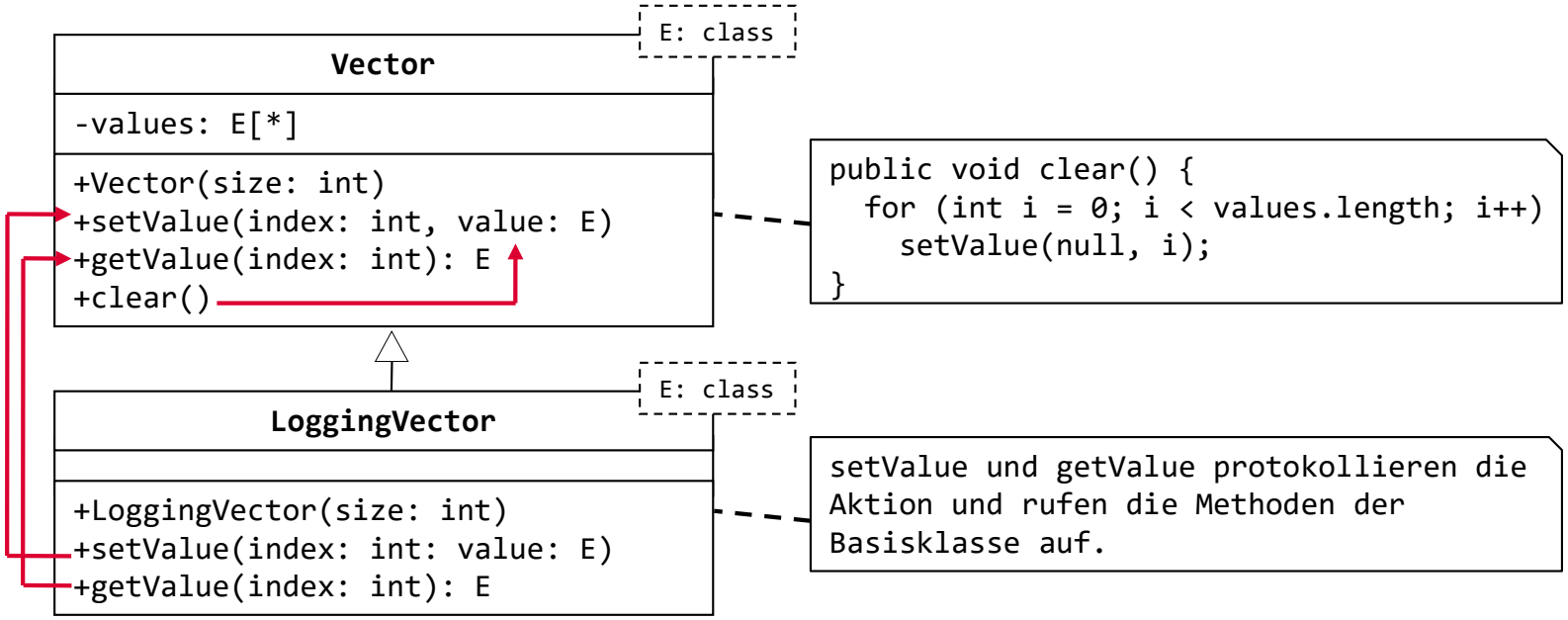
Weitere Beispiele
Übersicht



Weitere Beispiele

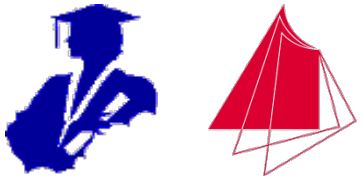
Probleme bei Vererbung – Instabile Basisklasse

- Und noch etwas gibt es bei Vererbung zu beachten, Beispiel:
  - Die Klasse **`LoggingVector`** erbt von der Basisklasse **`Vector`** (Vererbung der Implementierung).
  - **`LoggingVector`** protokolliert zusätzlich alle Aktionen von **`Vector`**.
  - Die Methode **`clear`** wird nicht überschrieben, weil sie intern **`setValue`** aufruft.

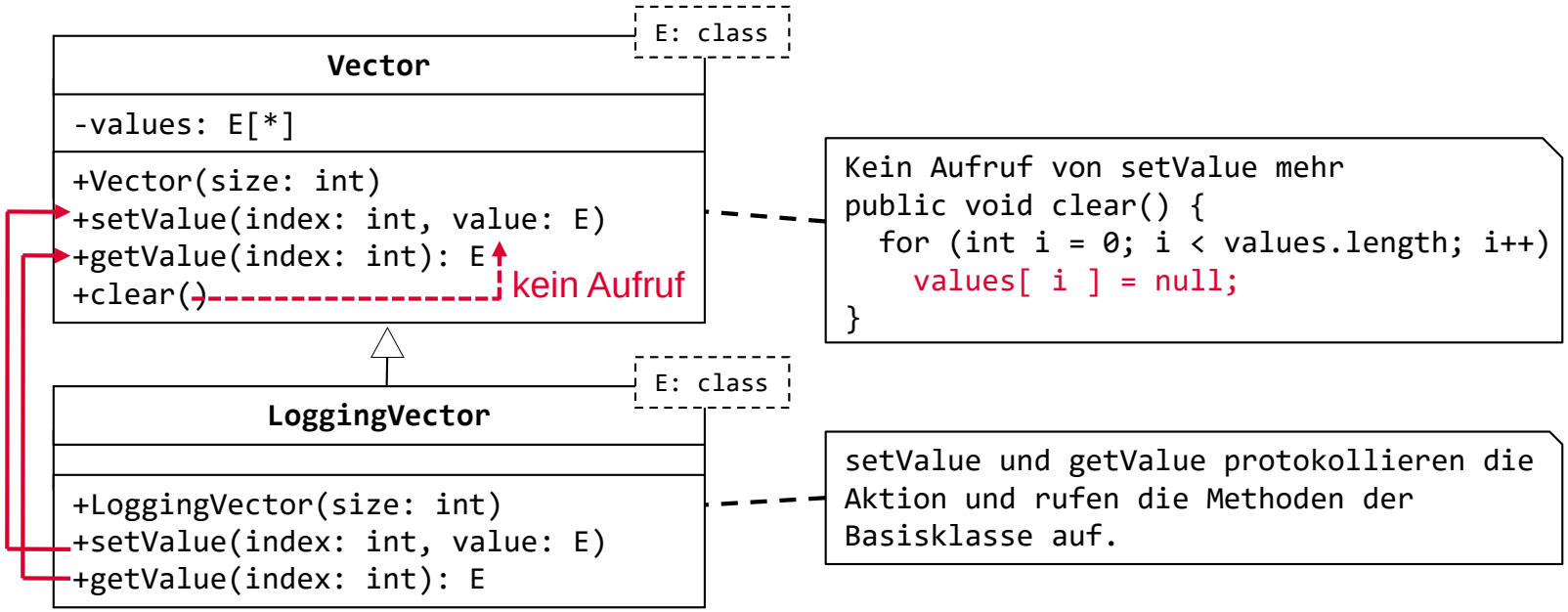


Weitere Beispiele

Probleme bei Vererbung – Instabile Basisklasse



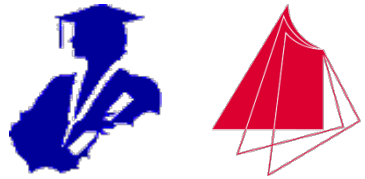
- Wo ist jetzt das Problem?
- Eine Änderung an der Basisklasse führt dazu, dass die abgeleitete Klasse nicht mehr korrekt funktioniert.



- Beim Aufruf von **`clear`** erfolgt kein Logging mehr!

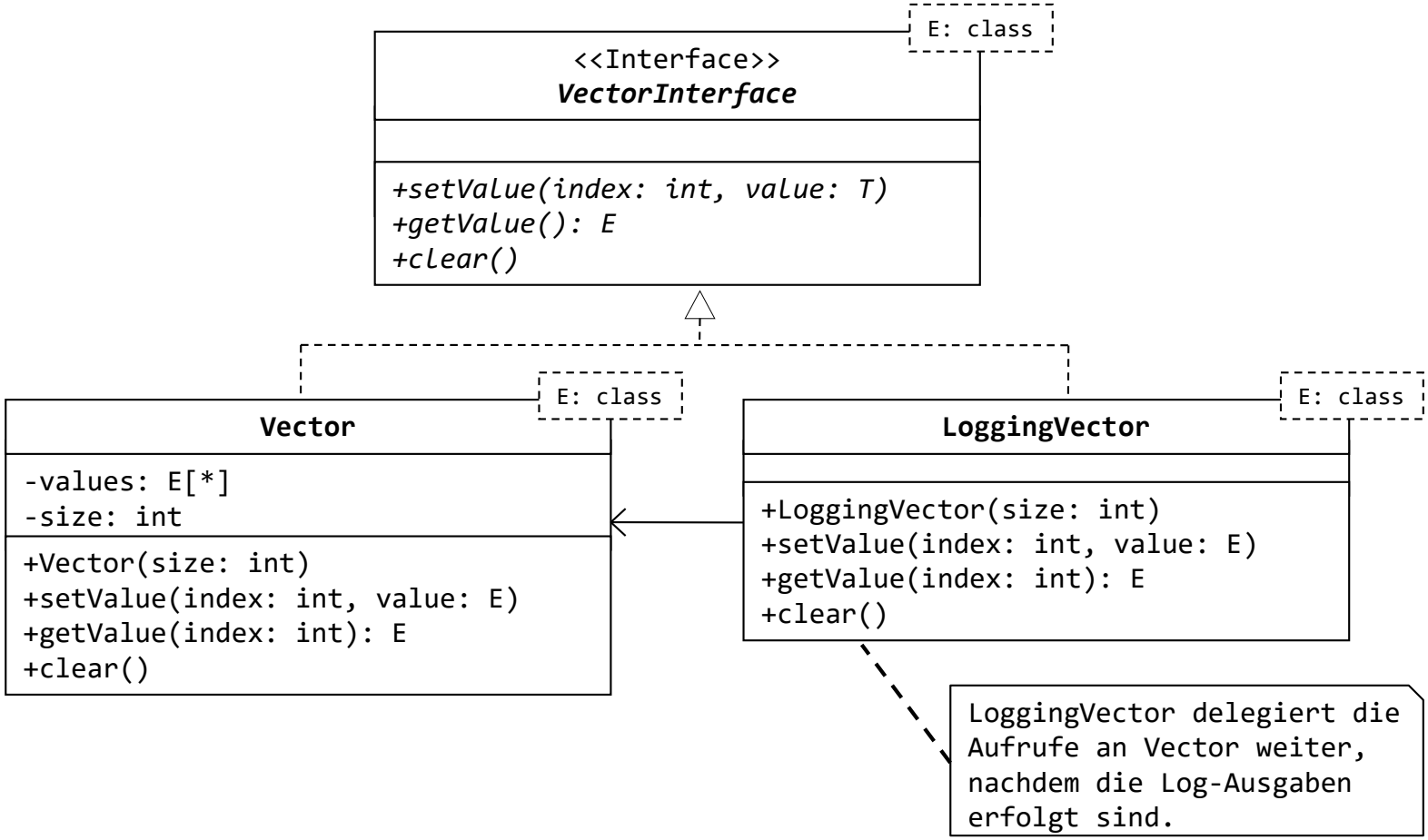
Weitere Beispiele

Probleme bei Vererbung – Instabile Basisklasse



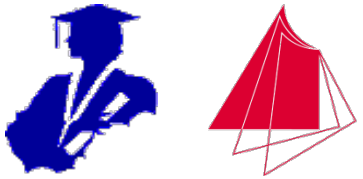
- **Problem der instabilen Basisklassen (Fragile Base Class Problem)**:
  - Anpassungen an einer Basisklasse führen zu unerwartetem Verhalten von abgeleiteten Klassen.
  - Konsequenz: Anpassungen an Basisklassen können häufig nicht vorgenommen werden, ohne den Kontext der abgeleiteten Klassen mit einzubeziehen.
  - Problem: Die Wartung objektorientierter Systeme, die häufig die Vererbung der Implementierung nutzen, wird stark erschwert.
  - Konsequenz: Vererbung der Implementierung darf nicht eingesetzt wird, wenn spätere Änderungen an den Basisklassen wahrscheinlich sind.
  - Ziel: Reine Vererbung der Spezifikation. Die Vermeidung von Redundanzen, kann auch über Delegationsbeziehungen erreicht werden.
- Anmerkung: Wenn die abgeleitete Klasse sich exakt an die Spezifikation der Basisklasse hält und die Spezifikation später auch nicht verändert wird, ist das Erben einer Implementierung problemlos möglich.
- In der Praxis: Niemand spezifiziert das Verhalten exakt…

- Lösung mit Delegation:



Weitere Beispiele

Probleme bei Vererbung – Instabile Basisklasse



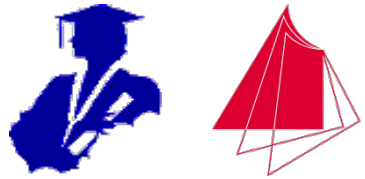
- Unvollständige Implementierung des Beispiels:

```
public class LoggingVector<E> implements VectorInterface<E> {
  private Vector<E> vector;

  public LoggingVector(int size) {
    vector = new Vector<E>(size);
  }

  @Override
  public void setValue(int index, E value) {
    // Log-Ausgaben, z.B. hier vereinfacht auf dem Bildschirm
    System.out.println("Neuer Wert " + value + " an Position " + index + " im Vektor");
    vector.setValue(value, index);
  }
  // usw.
}
```

- Änderungen an der Implementierung von **`Vector`** beeinflussen nicht die Funktionsfähigkeit von **`LoggingVector`**.
- Neue Methoden im **`Vector`** müssen auch in der Schnittstelle deklariert werden → können so nicht in **`LoggingVector`** vergessen werden.

Weitere Beispiele

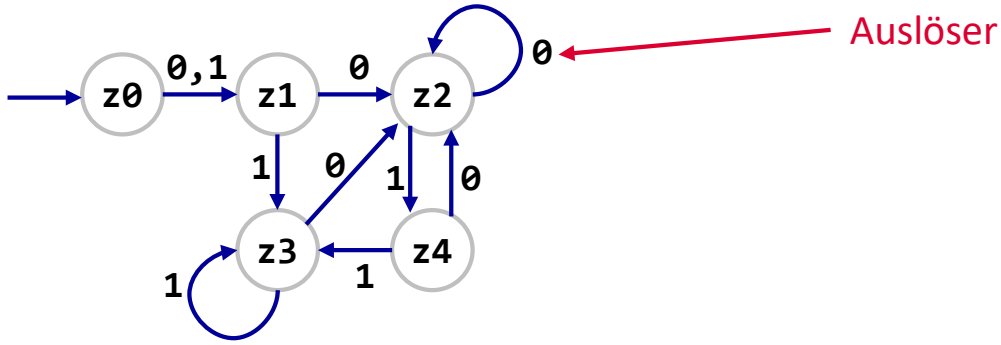
Probleme bei Vererbung – Instabile Basisklasse



- Fazit:
  - Vererbung der Spezifikation ist sicherer in Bezug auf Änderungen.
  - Vererbung der Implementierung erfordert häufig weniger Code → teilweise aber automatisch generierbar (z.B. in Eclipse).
  - Wenn sich Vererbung der Spezifikation leicht umsetzen lässt, ist diese Form der Vererbung vorzuziehen.

Weitere Beispiele
Zustandsautomat

- Aus der Vorlesung „Theoretische Informatik“ bekannt: Zustandsautomat
- Wie kann ein Automat objektorientiert mit Java umgesetzt werden?
- Beispiel aus einer Übungsaufgabe:



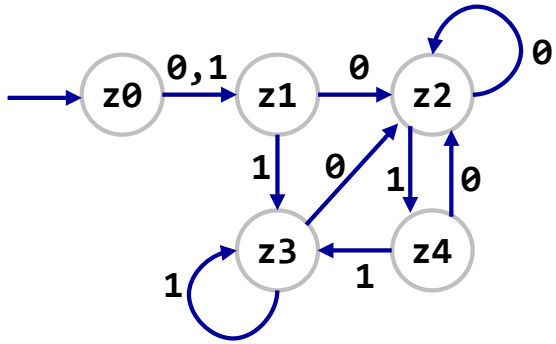
- Einige Möglichkeiten zur Implementierung:
  - Zustände und Übergänge als Tabelle beschreiben.
  - Zustände sind Klassen, die ihre Nachfolgezustände selbst bestimmen.
  - Klasse für Zustandsautomat besitzt **`switch`**/**`case`**-Block, um den neuen Zustand zu bestimmen, jeder **`case`** steht für den aktuellen Zustand

Weitere Beispiele

Zustandsautomat als Tabelle

- Umsetzung in einer Tabelle:

Zustand	Auslöser	Folgezustand
z0	0	z1
z0	1	z1
z1	0	z2
z1	1	z3
z2	0	z2
z2	1	z4
z3	0	z2
z3	1	z3
z4	0	z2
z4	1	z3



- Wird bei vielen Triggern oder Zuständen unübersichtlich.

Weitere Beispiele

Zustandsautomat manuell mit Klassen

- Schnittstelle, die alle Zustände implementieren müssen:

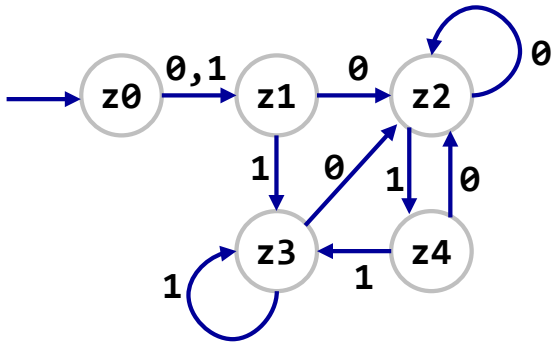
```
public interface State {
  State trigger(int value); // value ist hier 0 oder 1 → boolean wäre denkbar
}
```

- Zustände z0 und z1 als Beispiele:

```
public class Z1 implements State {
  @Override
  public State trigger(int value) {
    switch (value) {
      case 0: return States.z2;
      case 1: return States.z3;
    }
    // Bessere Fehlerbehandlung
    // wäre sinnvoll.
    return null;
  }
}
```

```
public class Z0 implements State {
  @Override
  public State trigger(int value) {
  // Unabhängig vom Wert wird in
  // den Zustand z1 gewechselt.
  return States.z1;
  }
}
```

- Die anderen Zustände sehen ähnlich aus.



Weitere Beispiele

Zustandsautomat manuell mit Klassen

- Von jedem Zustand existiert nur ein Objekt, das als öffentliches Attribut in einer separaten Klasse gespeichert ist:

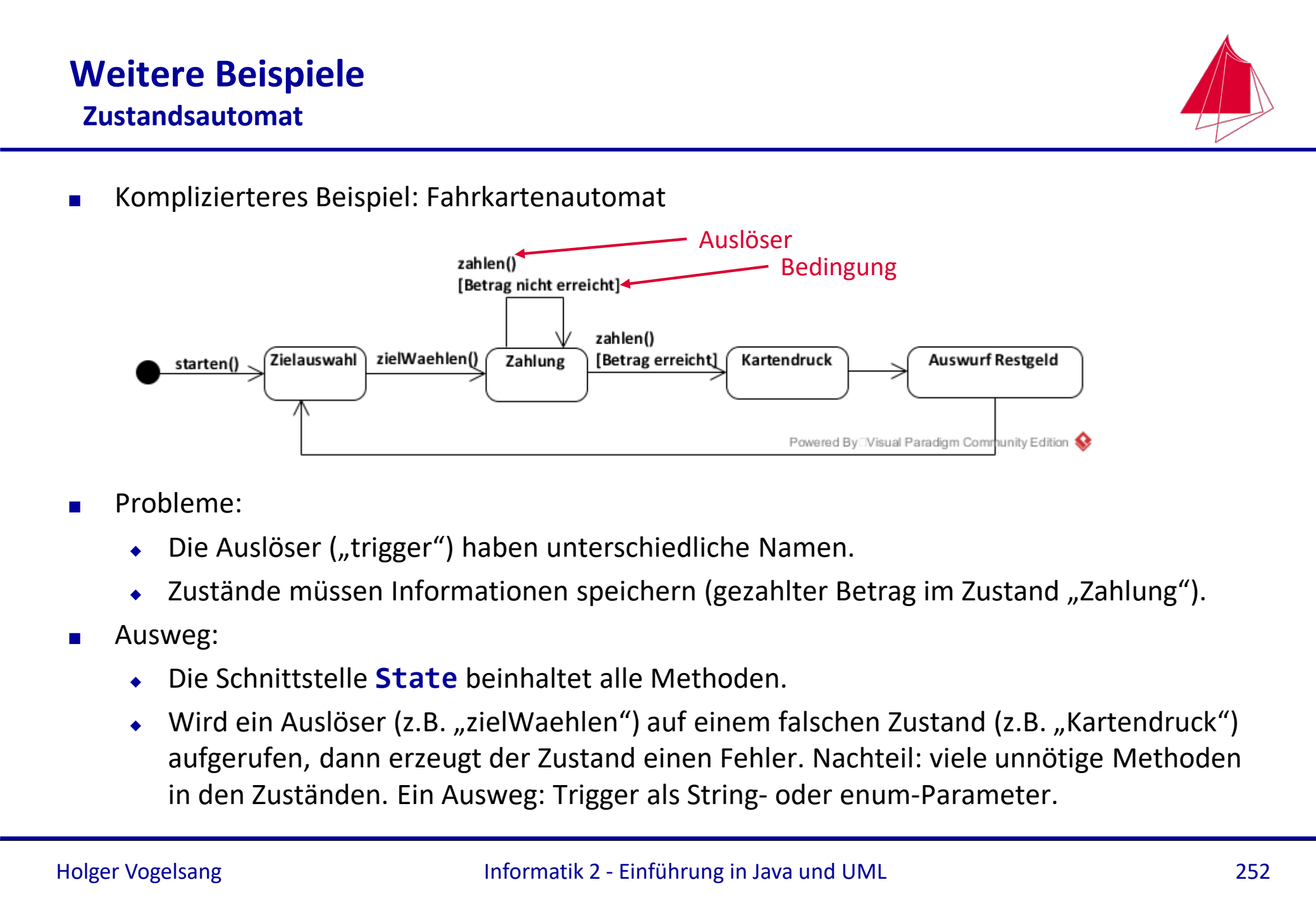
```
public class States {
  public static final Z0 z0 = new Z0();
  public static final Z1 z1 = new Z1();
  public static final Z2 z2 = new Z2();
  public static final Z3 z3 = new Z3();
  public static final Z4 z4 = new Z4();
}
```

- Dann können die Zustände direkt verwendet werden:

```
public class StateTester {
  public static void main(String[] args) {
    State state = States.z0.trigger(0).trigger(1).trigger(1).trigger(0)
                           .trigger(1).trigger(1);
    System.out.println(state.getClass().getSimpleName());
  }
}
```

- Für einfache Automaten reicht das aus.

Weitere Beispiele
Zustandsautomat



- Komplizierteres Beispiel: Fahrkartenautomat

- Probleme:
  - Die Auslöser („trigger“) haben unterschiedliche Namen.
  - Zustände müssen Informationen speichern (gezahlter Betrag im Zustand „Zahlung“).
- Ausweg:
  - Die Schnittstelle **`State`** beinhaltet alle Methoden.
  - Wird ein Auslöser (z.B. „zielWaehlen“) auf einem falschen Zustand (z.B. „Kartendruck“) aufgerufen, dann erzeugt der Zustand einen Fehler. Nachteil: viele unnötige Methoden in den Zuständen. Ein Ausweg: Trigger als String- oder enum-Parameter.