Klasse! Noch mehr OOP.

CS1016 Programmierung interaktiver Systeme

von Prof. Dr. Weigel

Klasse! Noch mehr OOP.

1. Klassen
2. Records
3. Enums
4. Pakete
5. Interfaces mit Default Methoden

Klassen

Beispiel: Rocket.java

```
public class Rocket {
    private Engine engine;

    public Rocket() {
        engine = new Engine();
    }

    public void start() {
        engine.start();
    }

    class Engine {
        public void start() {
           System.out.println("ppPPPPPiiiiUUUUUuuu!");
        }
    }
}
```

Top-Level Klasse

- Eine Klasse auf der obersten Ebene einer Java-Datei
- Meist nur eine Top-Level Klasse mit dem Namen der Datei

Geschachtelte Klasse

- Eine Klasse innerhalb der Top-Level Klasse.
- Zugriff nur über Rocket.Engine

Anonyme Klassen

Ähnlich wie lokale Klassen, aber ohne Namen

Erlauben gleichzeitige Deklaration und Initialisierung

→ Nur ein Objekt der Klasse erstellbar

Früher häufig verwendet, um ein Interface mit einer Funktion zu implementieren:

→ Erlaubt Übergabe von Funktionen als Parameter:
`void method(Greetings g) { ... }`

→ Im modernen Java durch Lambdas abgelöst

```
interface Greetings {
    public String greet();
}

Greetings english = new Greetings() {
    public String greet() {
        return "Hello World!";
    }
};

Greetings german = new Greetings() {
    public String greet() {
        return "Hallo Welt!";
    }
};

System.out.println(english.greet());
System.out.println(german.greet());
```

Abstrakte Klassen

Eine Klasse mit *einer* oder *mehreren* abstrakten Methoden

Es können keine Objekte von abstrakten Klassen generiert werden

```
abstract class Counter {
    int count;
    abstract void count();
    void printCounter() {
        System.out.println(count);
    }
}
```

Abstrakte Klasse

- Methode **count**() ist abstrakt
- Methode **printCounter**() ist nicht abstakt
- Keine Objekte erzeugbar (kein new Counter());
- Vererbung möglich

```
class ReverseCounter extends Counter {
    ReverseCounter(int start) {
        count = start;
    }
    void count() {
        count--;
    }
}
```

Konkrete Klasse

- Überschreibt **count**() mit eigener Implementierung
- Hat nur implementierte Funktionen
- Erlaubt das erzeugen von Objekten:
  var rc = new ReverseCounter(100);

Klasse! Noch mehr OOP.

1. Klassen
2. Records
3. Enums
4. Pakete
5. Interfaces mit Default Methoden

Records (seit Java 16)

Neue Struktur für einfache Klassen mit unveränderlichen Daten → Alle Variablen sind *final.*
Können Interfaces implementieren. Aber nicht von anderen Klassen/Records erben.

```
record Point3D(double x, double y, double z) {
    // Optional zusätzlicher Konstruktor
    public Point3D() {
        this(0, 0, 0);
    }

    // Optionale Methoden
    public double distanceTo(Point3D other) {
        return Math.sqrt(
            Math.pow(x-other.x, 2) +
            Math.pow(y-other.y, 2) +
            Math.pow(z-other.z, 2));
    }
}
```

Beispielbenutzung:

```
// Automatisch generiert
var point = new Point3D(10, 10, 1);
point.x();          // → 10

// Nur mit zusätzlichem Konstruktor
var origin = new Point3D();

// Aufruf optionaler Methoden
point.distanceTo(origin);
```

Vorteile von Records

☺ Unveränderbare (“immutable”) Objekte

☺ Automatischer Konstruktor mit allen Variablen

☺ Automatische Generierung von `toString()`, `equals()` und `hashCode()`

```
record Point3D(double x, double y, double z) {}
var p1  = new Point3D(123, 42, 777);
var p2 = new Point3D(123, 42, 777);
p1.equals(p2);      // → true
p1.toString();      // → "Point3D[x=123.0, y=42.0, z=777.0]"
```

☺ Gut für **D**aten**t**ransfer**o**bjekte zwischen Methoden: Erlaubt mehrere Rückgabewerte
Beispiel: Point3D **calculatePosition() { [...] return new** Point3D**(**x, y, z**); }**

Klassen vs. Records

Klassen

- Erlauben Änderungen von Daten
- Unterstützen Vererbung

→ Skaliert auf komplexere Datenstrukturen

- Langatmiger Syntax, um Record-Funktionalitäten zu unterstützen (Konstruktor, toString(), equals())

Records

- Objekte sind nach Erstellung unveränderlich
- Keine Vererbung

→ Gut für einfache Datenstrukturen

- Einfacherer und kurzer Syntax
- Schnell lesbar

Was nutze ich wann?

- Wenn ein Record ausreichend ist → Record
- Wenn sich Daten im Objekt ändern oder Vererbung benötigt wird → Klassen
- Kompatibilität mit älteren Java-Versionen (z.B. für Bibliotheken) → Klassen

Klasse! Noch mehr OOP.

1. Klassen
2. Records
3. Enums
4. Pakete
5. Interfaces mit Default Methoden

Enum (Aufzählungstyp)

Eine Menge an konstanten Werten

```
// Enums ohne Werte
enum Color { RED, GREEN, BLUE }

// Enums mit Werten
enum Direction {
    EAST(0), WEST(180), NORTH(90), SOUTH(270);

    private int angle;
    private Direction(final int angle) {
        this.angle = angle;
    }
    int getAngle() {
        return angle;
    }
}
```

Werte in Klammern werden zu Parametern
Auch mehrere Parameter möglich

```
var c = Color.RED;
if(c == Color.GREEN)
    [...]
```

```
var d = Direction.EAST;
int angle = d.getAngle();
```

Advanced: Intern sind enums Klassen. Vererbung ist aber nicht möglich, da diese “final” sind.

Enums für Zustände (für endliche Automaten)

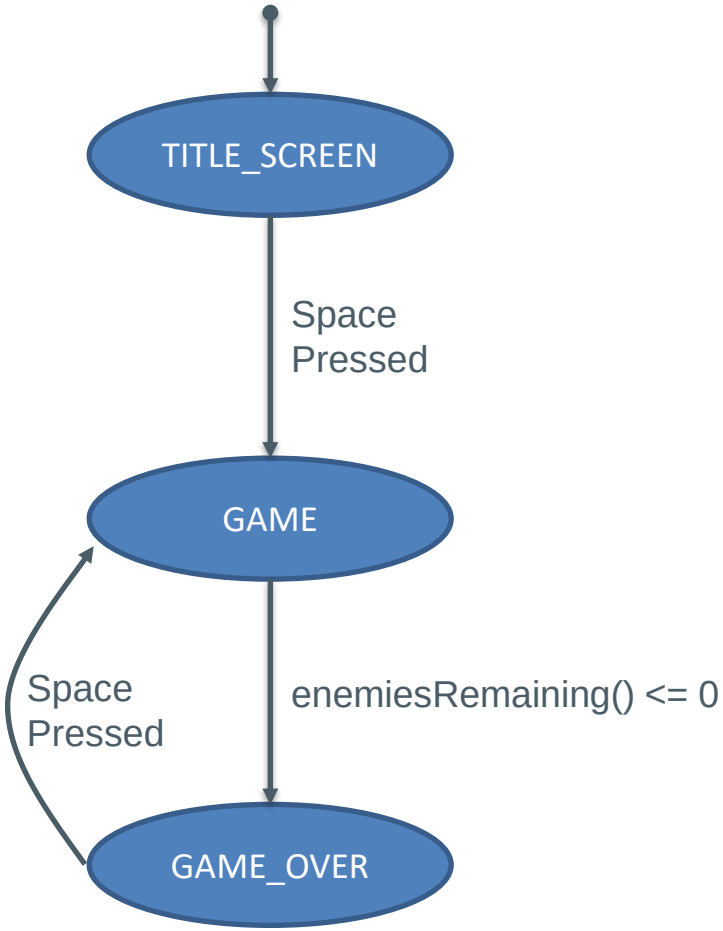
```
public enum GameState {
    TITLE_SCREEN, GAME, GAME_OVER
}
```

Setzen vom GameState:

```
GameState state = GameState.TITLE_SCREEN;
```

Auswerten im Controller:

```
switch(state) {
    case TITLE_SCREEN -> {
        view.drawTitleScreen();
    }
    case GAME -> {
        view.drawGame();
    }
    case GAME_OVER -> {
        view.drawGameOver();
    }
}
```



Klasse! Noch mehr OOP.

1. Klassen
2. Records
3. Enums
4. Pakete
5. Interfaces mit Default Methoden

Was sind Pakete?

Erlauben das Strukturieren von mehreren Dateien und Klassen

Wozu brauchen wir Pakete?

- ☺ Bündeln von einzelnen Programmkomponenten (z.B. Model, View, Controller oder Bibliotheken)
  → Können zum Beispiel als Bibliothek weitergegeben werden (s. `processing.core`)
- ☺ Es gibt keine Namenskonflikte zwischen den Paketen
  → Entwickler verschiedener Pakete müssen sich nicht absprechen
- ☺ Klassen im gleichen Paket haben spezielle Zugriffsregeln
  → Trennung interne Entwicklung und externe API

In vielen modernen Programmiersprachen verfügbar (auch namespaces genannt)

Beispiel: Foo.java

```
// “package” setzt Paketnamen der Datei fest
package de.thm.mni.lecture.pis;

// Importiert PApplet und PImage vom Paket processing.core
import processing.core.PApplet;
import processing.core.PImage;

class Foo extends PApplet {}
```

Wichtig:

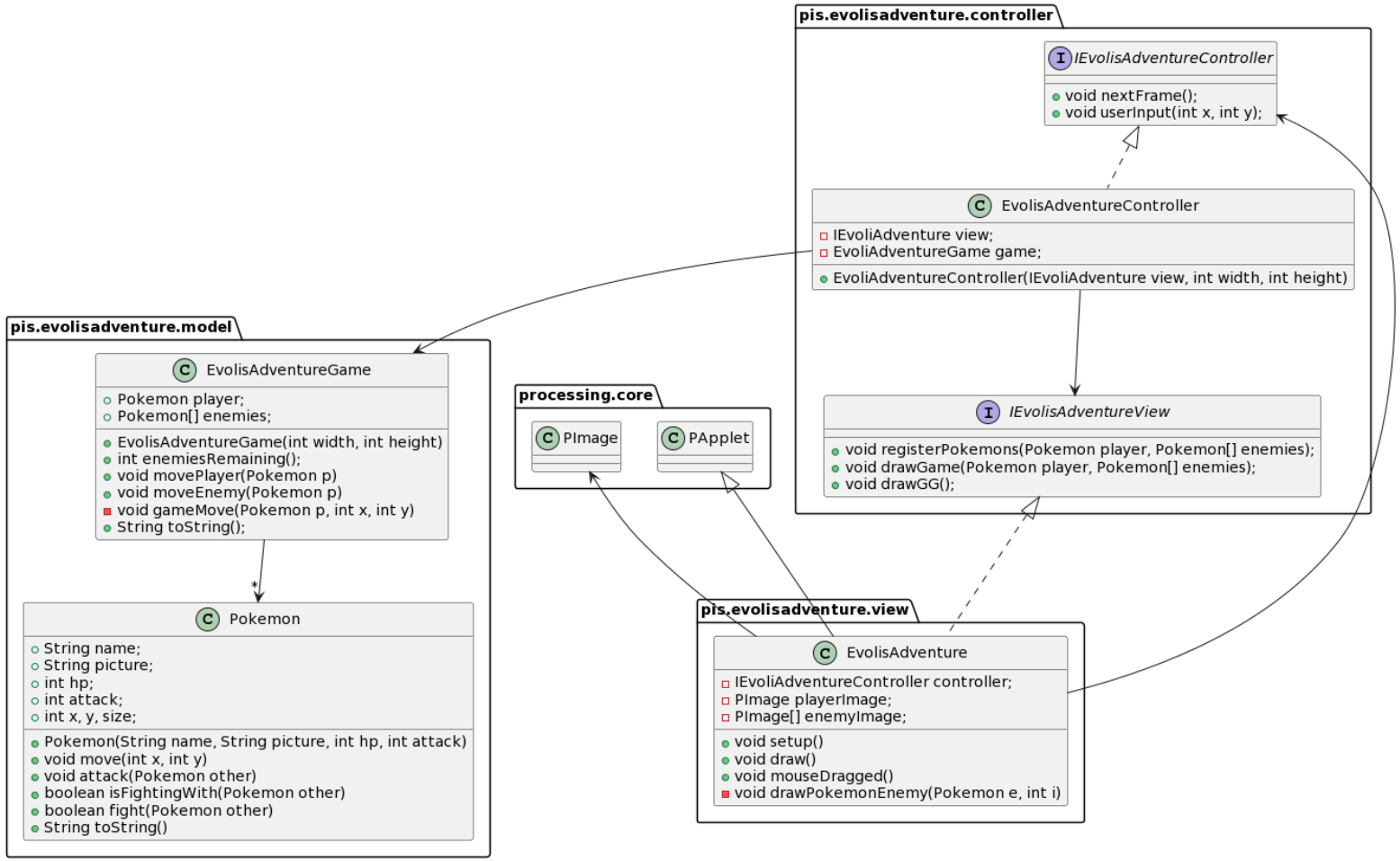
1. Der Dateipfad muss dem Paketnamen entsprechen, hier: de/thm/mni/lecture/pis/Foo.java
2. Der Paketname ist eine ID und hat keine Hierarchie, d.h.
   a) import von a.b importiert nicht a.b.c
   b) Dateien in a, a.b und a.c können ohne Konflikt die gleichen Namen verwenden

Packages in UML

Packages werden als Rechtecke um Klassen, Interfaces und Records gezeichnet

Hilft beim Visualisieren von Abhängigkeiten zwischen Paketen, z.B.:

Im Bild rechts wird klar, dass nur das Paket `pis.evolisadventure.view` von `processing.core` abhängt.



Zugriffsrechte

Modifizierer	außerhalb	abgeleitete Klasse (extends)	im gleichen Paket
public	**Ja**	**Ja**	**Ja**
protected	X	**Ja**	**Ja**
ohne Angabe	X	X	**Ja**
private	X	X	X

Klasse! Noch mehr OOP.

1. Klassen
2. Records
3. Enums
4. Pakete
5. Interfaces mit Default Methoden

Default Interface Methoden (seit Java 8)

Default Methoden

- Die Default-Methode ist ein Implementierungsvorschlag im Interface
- Eingeleitet durch das Schlüsselwort **default**
- Implementierung innerhalb eines { }-Blockes
- Hilfreich um ein sinnvolles Standardverhalten zu implementieren → **D**on't **R**epeat **Y**ourself!
- Kann von der Klasse überschrieben werden

Beispiel (für die JShell)**:**

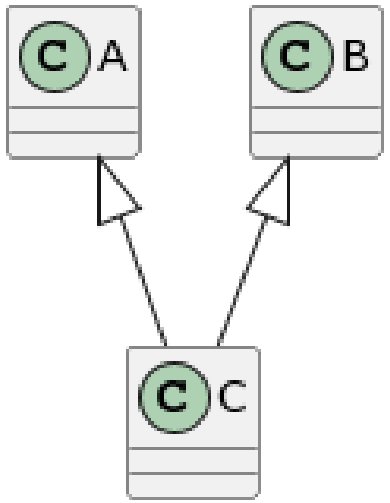
```
interface IA {
    default int method() { return 1; }
}

class C implements IA {}

var test = new C();
test.method(); // Rückgabe 1
```

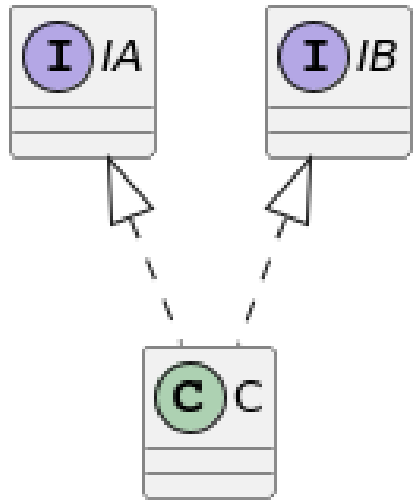
Vererbung und Implementierung

Erben von mehreren Klassen



Wird in Java **nicht** unterstützt

Implementieren von mehreren Interfaces



Wird in Java unterstützt

Mehrfachvererbung durch die Hintertür

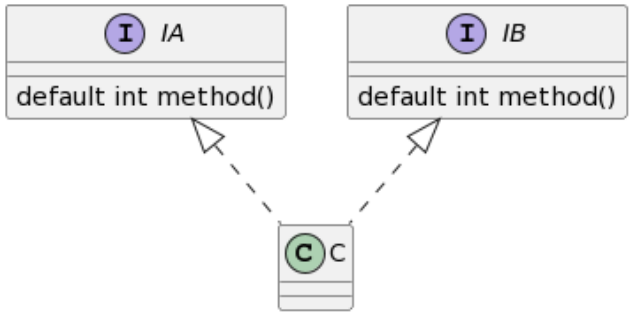
“Java omits many rarely used, poorly understood, confusing features of C++ that in our experience bring more grief than benefit. This primarily consists of [...] ***multiple inheritance*** *[...].” – James Gosling (Java Urvater)*

```
interface IA {
    default int method() { return 1; }
}

interface IB {
    default int method() { return 2; }
}

class C implements IA, IB {}

var test = new C();
test.method();
```



Mini-Quiz: Was macht der Code links?

a) Rückgabewert ist 1
b) Rückgabewert ist 2
c) Compiler-Fehler
d) Runtime-Fehler

Lösung

Da diese Mehrfachvererbung nur für Interfaces möglich ist, tritt das Problem nur bei Methoden auf

→ Einfacher als “echte” Mehrfachvererbung

Fehler beim Kompilieren:

```
jshell> class C implements IA, IB {}
|  Error:
|  types IA and IB are incompatible;
|    class C inherits unrelated defaults for
method() from types IA and IB
```

Lösung: C muss die Methode überschreiben

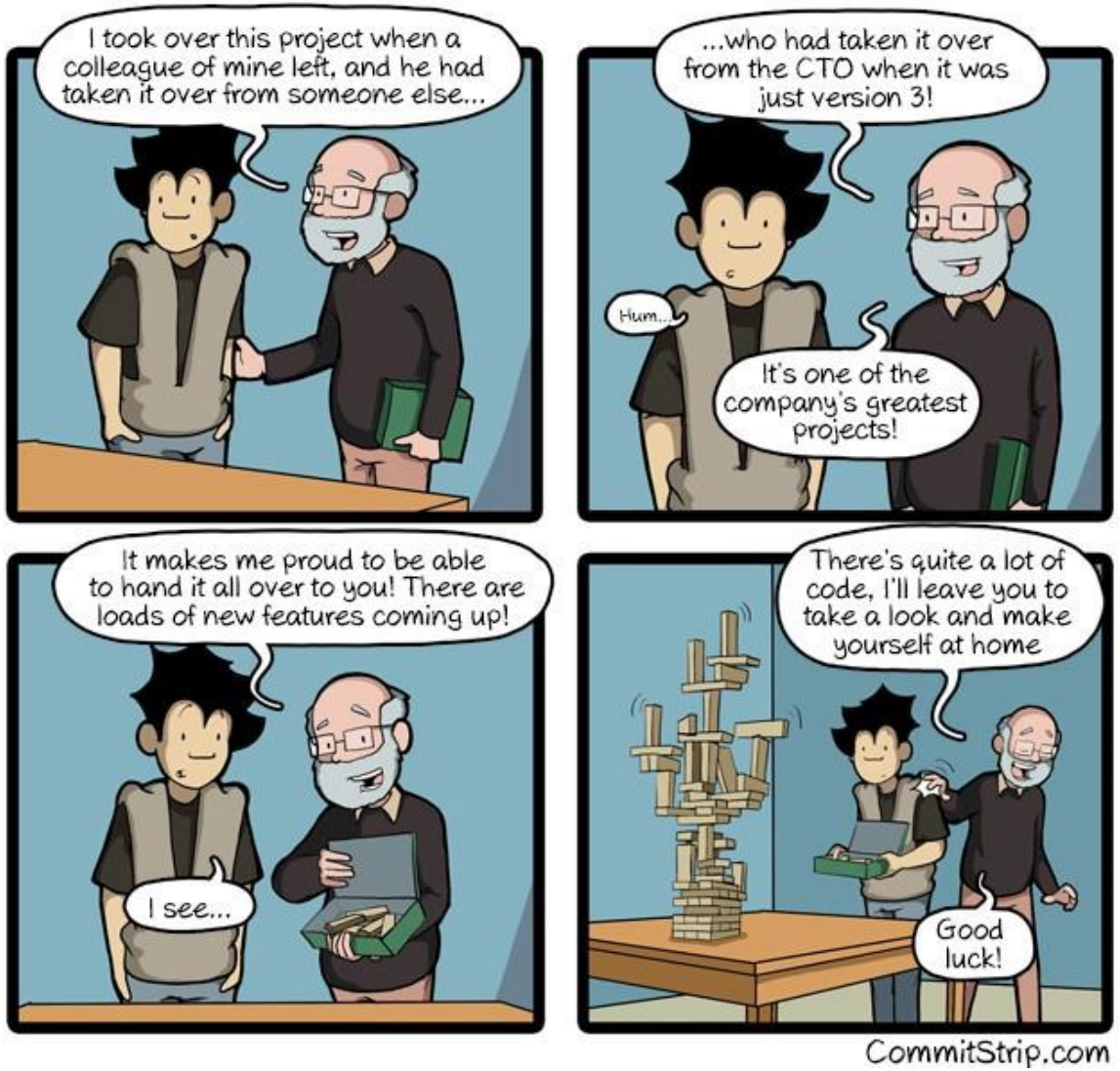
```
interface IA {
    default int method() { return 1; }
}

interface IB {
    default int method() { return 2; }
}

class C implements IA, IB {
    public int method() {
        return IB.super.method();
    }
}

var test = new C();
test.method(); // Rückgabe 2
```

Fragen?



[Bildquelle: https://www.commitstrip.com/en/2016/02/15/our-companys-greatest-project/?]

Übung: Evolis Adventure III

1. Laden von Evoli Adventure Model in der JShell
2. Aufteilen von Model-View-Controller in separate Pakete
3. Ein “Dark” View für Evolis Adventure
4. Game State als Enum
5. Nutzen von Records (“Stats” in Pokemon)