

# Programmieren II

## Abstrakte Klassen, Interfaces



Heusch 13.8, 13.9  
Ratz 9.6

Institut für Automation und angewandte Informatik

```
final List<String> allResults = new ArrayList<String>();  
final Map<String, Integer> typeWordResultCount = new HashMap<String, Integer>();  
final Map<String, Integer> typePoints = new HashMap<String, Integer>();  
evaluation.put(type, typePoints);  
  
for (final Sheet sheet : this.sheets) {  
    final String sheetResult = sheet.getPlayerInput(type);  
    if (sheetResult.startsWith(start) && this.isValidWord(sheetResult, type)) {  
        validWordCountForType++;  
        allResults.add(sheetResult);  
    }  
}
```

# Abstrakte Klassen: Motivation

- Prinzip der Vererbung:
  - Aus existierenden Klassen können *spezialisierte Klassen* abgeleitet werden
- Prinzip der abstrakten Klassen:
  - Aus mehreren ähnlichen Klassen kann eine *gemeinsame Oberklasse* abstrahiert werden
  - Sinn und Zweck: Ausnutzung der Polymorphie!

```
public class Rectangle {  
    void draw() { /* ... */ }  
    // ...  
}  
  
public class Circle {  
    void draw() { /* ... */ }  
    // ...  
}
```

```
public class Line {  
    void draw() { /* ... */ }  
    // ...  
}  
  
public abstract class Graphic {  
    abstract void draw();  
}
```

# Abstrakte Klassen: Definition

## ■ Konzept : *Abstrakte Klasse*

- Eine abstrakte Klasse ist eine *bewusst unvollständige Oberklasse*, in der von einzelnen Methodenimplementierungen abstrahiert wird („abstrakte“ Methoden!)
- Fehlende Methodenrümpfe werden erst in abgeleiteten Unterklassen implementiert
- Die Instanziierung abstrakter Klassen ist nicht möglich
- Modifikator (modifier): Java-Schlüsselwort `abstract`

```
public abstract class Graphic {  
    abstract void draw(); // Kein Methodenrumpf!  
}
```

# Abstrakte Klassen: Beispiel (1)

```
public abstract class Graphic {
    String name;


    public String getName() {
        return this.name;
    }

    abstract void draw(); // nur Methodendeklaration
                          // ohne Methodenrumpf { ... }
}

public class Rectangle extends Graphic {
    float width, height;

    public Rectangle(String str, float w, float h) {
        this.name = str;
        this.width = w;
        this.height = h;
    }
    // geerbt: getName - ist bereits in Graphic implementiert

    void draw() {          // wird erst hier implementiert
        System.out.println("Rectangle: " + this.name);
    }
    // weitere Methoden...
}
```



# Abstrakte Klassen: Beispiel (2)

```
public class Circle extends Graphic {  
    float radius;  
  
    public Circle(String str, float r) {  
        this.name = str;  
        this.radius = r;  
    }  
    // geerbt: getName weitere Methoden ...  
    // implementiert:  
    void draw() {  
        System.out.println("Circle: " + this.name + this.radius);  
    }  
}  
public class Line extends Graphic {  
    // ...  
    // implementiert:  
    void draw() {  
        System.out.println("Line: " + this.name);  
    }  
}
```

# Abstrakte Klassen: Beispiel (3)

```
public class GraphicColl {    // Zur Speicherung einer Menge
                               // von Graphic-Objekten

    Graphic[] elems;
    int next;

    public GraphicColl(int size) {
        this.elems = new Graphic[size];
        this.next = 0;
    }

    public void add(Graphic obj) {
        if (this.next < this.elems.length){
            this.elems[this.next++] = obj;
        }
    }

    public Graphic[] get() {
        return this.elems;
    }
}
```

# Abstrakte Klassen: Beispiel (4)

```
public class Probe {  
    public static void main(String[] args) {  
        Rectangle r1 = new Rectangle("Rectangle 1", 10, 20);  
        Circle c1 = new Circle("Circle 1", 50);  
        Line l1 = new Line("Line 1");  
        Rectangle r2 = new Rectangle("Rectangle 2", 15, 15);  
        Graphic g1 = new Graphic("Graphic 1"); // Fehlermeldung!  
                                                // "Cannot instantiate the type Graphic"  
        GraphicColl coll = new GraphicColl(6);  
        coll.add(r1);  
        coll.add(r2);  
        coll.add(c1);  
        coll.add(l1);  
  
        Graphic[] elements = coll.get();  
        for (int i = 0; i < elements.length; i++)  
            if (elements[i] != null)  
                elements[i].draw(); // Polymorphie  
    }  
}
```

# Interfaces: Konzept

- Interfaces dienen dazu, *gleiche Schnittstellen* in *unterschiedlichen Klassen* zu definieren
  - Ein Interface ist eine Art Klasse (ist aber keine Klasse), die ausschließlich Konstanten und abstrakte Instanzmethoden deklariert
  - Java-Schlüsselwörter `interface` und `implements`
  - Von Interfaces können mit `new` keine Instanzen erstellt werden. Es kann aber als „Variablen-/Referenzen-Typ“ verwendet werden
  - Sinn: Ersatz für in Java nicht mögliche *Mehrfachvererbung*

```
public interface Graphic {  
    public void draw(); // Methodenrumpf nicht erlaubt!  
}  
public class Line implements Graphic {  
    public void draw() {  
        System.out.println(".....");  
    }  
}
```



# Interfaces: Beispiel (1)

```
public interface LandVehicle {  
    public void drive(); // kein Methodenrumpf!  
}
```

```
public interface Watercraft {  
    public void swim(); // kein Methodenrumpf!  
}
```

```
public class Vehicle {  
    Motor m;  
    // ...  
}
```

```
public class Car extends Vehicle implements LandVehicle {  
    public void drive() {  
        // Implementierung der Methode  
    }  
}
```



# Interfaces: Beispiel (2)

```
public interface LandVehicle {  
    public void drive(); // kein Methodenrumpf!  
}
```

```
public interface Watercraft {  
    public void swim(); // kein Methodenrumpf!  
}
```

```
public class Vehicle {  
    Motor m;  
    // ...  
}
```

```
public class MotorBoat extends Vehicle implements Watercraft {  
    public void swim() {  
        // Implementierung der Methode  
    }  
}
```



# Interfaces: Beispiel (3)

```
public interface LandVehicle {
    public void drive(); // kein Methodenrumpf!
}
```

```
public interface Watercraft {
    public void swim(); // kein Methodenrumpf!
}
```

```
public class Vehicle {
    Motor m;
    // ...
}
```

```
public class AmphibiousVehicle
    extends Vehicle implements LandVehicle, Watercraft {

    public void drive() { /* Implementierung der Methode */ }

    public void swim() { /* Implementierung der Methode */ }

}
```



## Interfaces: Beispiel (4)

```
<type> x = new AmphibiousVehicle();
```

? Was darf hier für <type> stehen?



# Abstrakte Klassen und Interfaces: Vergleich

## ■ Abstrakte Klassen:

- abgeleiteten Klassen soll bereits ein bestimmtes Grundverhalten zur Verfügung gestellt werden (➔ Vererbung)
- (Einfach-)Polymorphie

## ■ Interfaces:

- ausschließliche Definition des *Protokolls* („Methodenköpfe“), keine Implementierung erlaubt
- (Mehrfach-)Polymorphie

# Erweiterungen, die zu Interfaces in Java 8 eingeführt wurden (1)

## ■ Statische Methoden in Interfaces

- Mit Java 8 ist es möglich, auch *statische Methoden* in Interfaces zu implementieren.

## ■ Default-Methoden

- Bisher mussten Methoden in Interfaces abstrakt sein. Eine Implementierung (Rumpf) konnten sie nicht haben.
- Nun kann in einem Interface mittels einer *Default-Methode* auch eine Methode *mit Implementierung* zur Verfügung gestellt werden.
- Java-Schlüsselwort: `default`
- Diese Methoden werden dann auch wie üblich vererbt.
- Dadurch ist auch eine Mehrfachvererbung von Funktionalität möglich.

# Erweiterungen, die zu Interfaces in Java 8 eingeführt wurden (2)

```
public interface InterfaceA {  
  
    default void methodA() {  
        System.out.println("Invoked <InterfaceA>.methodA()");  
    }  
}  
  
public class ClassA implements InterfaceA {  
  
    public static void main(String[] args) {  
        ClassA c1 = new ClassA();  
        c1.methodA();  
    }  
}
```



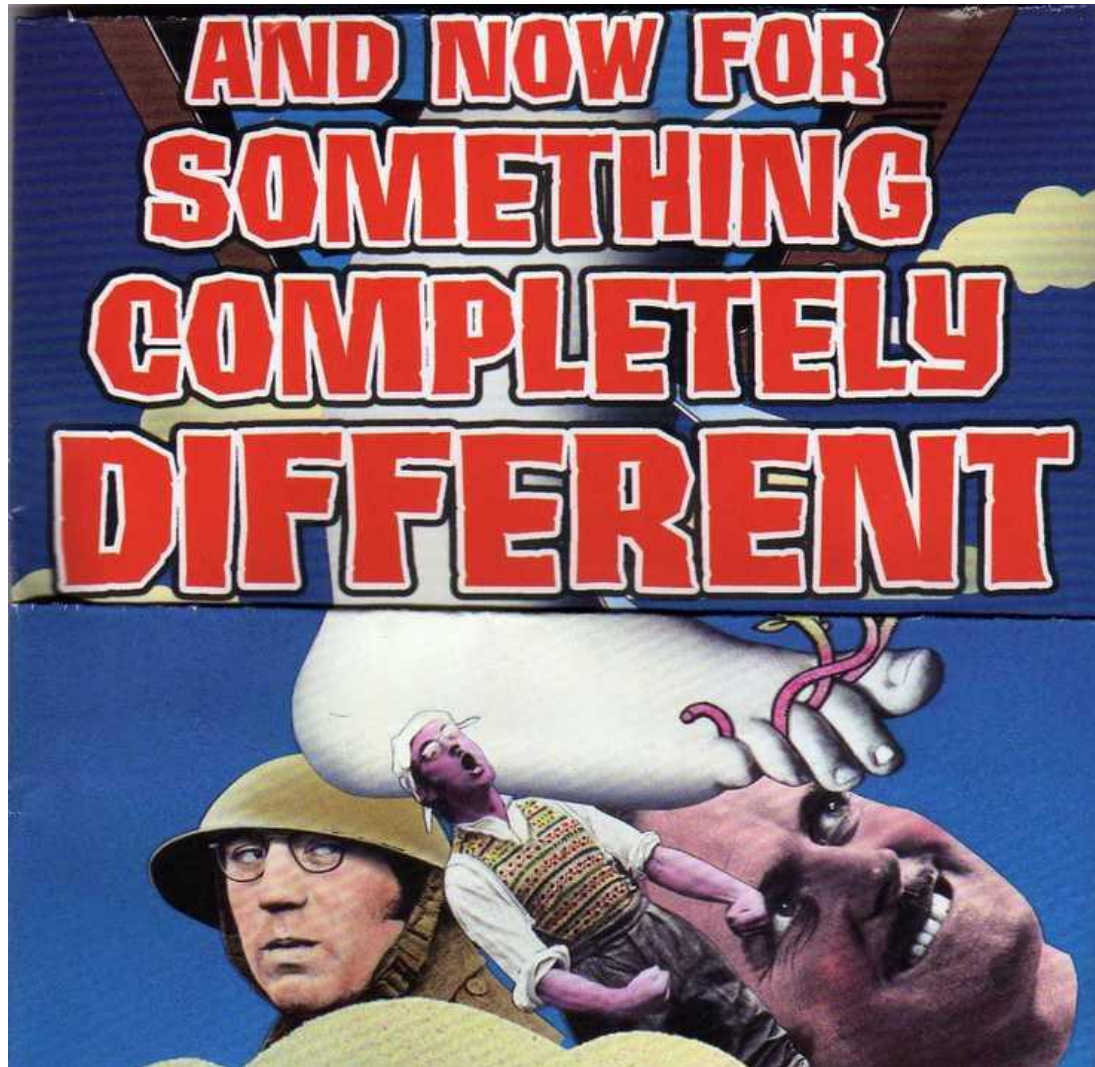
Ausgabe

```
Invoked <InterfaceA>.methodA()
```



**Ab Java 1.8**





© Monty Python's Flying Circus



## Definitionen

### ■ Subobjekt

- ein Attribut vom Typ einer Klasse

### ■ Komposition

- Zusammensetzung eines Objektes aus mehreren Subobjekten
- „part-of-Beziehung“ (z.B. Bestandteile eines Autos)
- (Vererbung: „is-a-Beziehung“)

### ■ Aggregation

- Loser Verbund von Subobjekten (z.B. Vögel in einem Schwarm)
- Subobjekte haben ihre eigene Identität

### ■ Delegation

- Prinzip der Implementierung einer Methode durch Weiterreichen eines Methodenaufrufs an ein Subobjekt

# Subobjekte/Komposition (1)

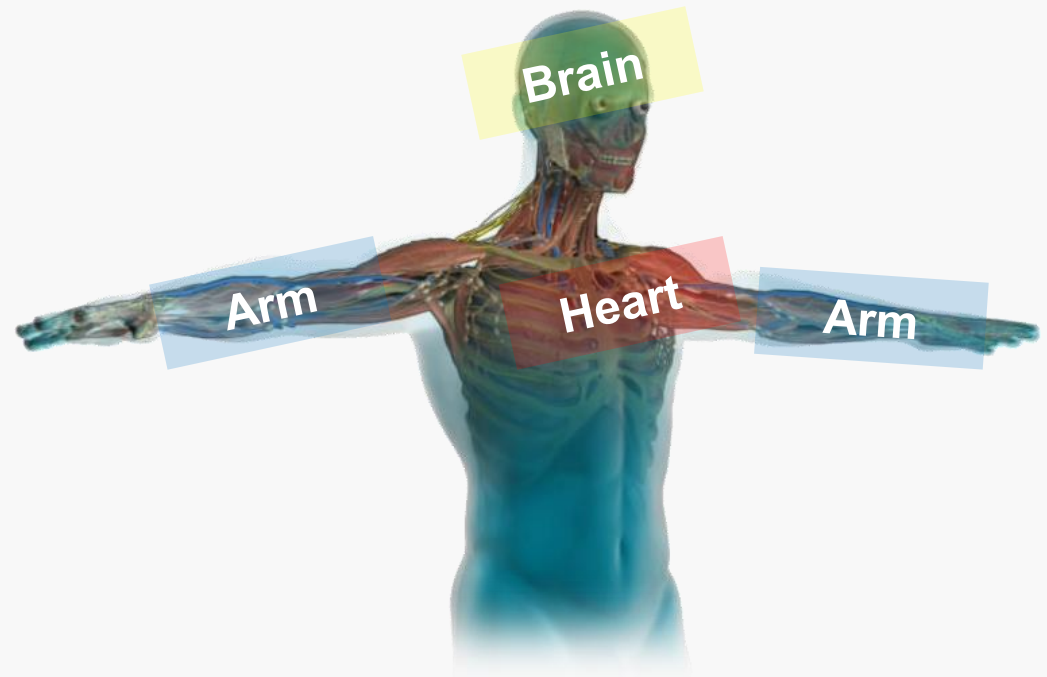
```
public class Brain {  
    public void remember(Object info) { /* ... */ }  
  
    public Object think() { /* ... */ }  
}
```

```
public class Heart {  
    public void beat(int count) { /* ... */ }  
}
```

```
public class Arm {  
    public void bend(double degrees) { /* ... */ }  
  
    public void lift(Object load) { /* ... */ }  
}
```

# Subobjekte/Komposition (2)

```
public class Human {  
    Brain brain; // Subobjekt  
    Heart heart; // Subobjekt  
    Arm[] arms; // mehrere Subobjekte (in Array)  
  
    public Human() {  
        this.brain = new Brain();  
        this.heart = new Heart();  
        this.arms = new Arm[2];  
        this.arms[0] = new Arm();  
        this.arms[1] = new Arm();  
    }  
  
    public Object think() {  
        return this.brain.think();  
    }  
  
    public void doSports() {  
        this.heart.beat(180);  
    }  
  
    public void sleep() {  
        this.heart.beat(50);  
    }  
}
```



# Subobjekte: Delegation

```
public class LandVehicle {  
    public void drive() {  
        // ...  
    }  
}
```

```
public class Watercraft {  
    public void swim() {  
        // ...  
    }  
}
```

```
public class AmphibiousVehicle {  
    LandVehicle land;           // Subobjekte  
    Watercraft water;  
    // ...  
  
    public void drive() {  
        this.land.drive(); // Delegation  
    }  
  
    public void swim() {  
        this.water.swim(); // Delegation  
    }  
}
```

 Hier gibt es keine Vererbung!