Einführung in die Programmierung

by André Karge

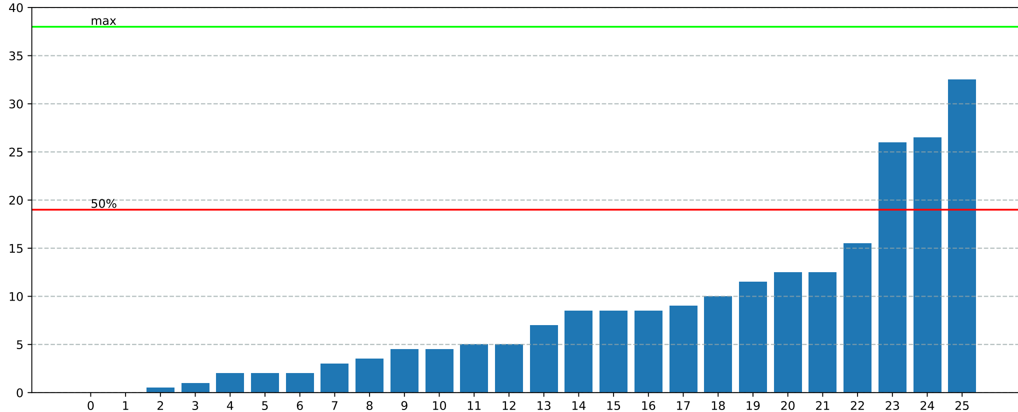
Übung - Probeklausur, Generics

Heute

- Probeklausur
- Polymorphie

Probeklausur

Probeklausur



Wiederholung MethodenHeader

[visibility] [static] [returnType] **methodenBezeichner** ([ParameterType] [parameterName], ...)

Wiederholung MethodenHeader

[visibility] [static] [returnType] **methodenBezeichner** ([ParameterType] [parameterName], ...)

- visibility:
  - public
  - private
  - protected
  - keine angegeben? → Default (Sichtbar im Package)

Wiederholung MethodenHeader

[visibility] [static] [returnType] **methodenBezeichner** ([ParameterType] [parameterName], ...)

- visibility:
  - public
  - private
  - protected
  - keine angegeben? → Default (Sichtbar im Package)
- static
  - methode static? → schreibe **static**
  - methode nicht static? → schreibe **kein** static

Wiederholung MethodenHeader

[visibility] [static] [returnType] **methodenBezeichner** ([ParameterType] [parameterName], ...)

- visibility:
  - public
  - private
  - protected
  - keine angegeben? → Default (Sichtbar im Package)
- static
  - methode static? → schreibe **static**
  - methode nicht static? → schreibe **kein** static
- returnType:
  - soll methode was zurückgeben? nein? → schreibe **void**
  - ja? primitive Type? → schreibe den jeweiligen primitive Type (**int, float, double, bool, ...**)
  - ja? komplex Type? → schreibe den jeweiligen komplex Type (**Fahrrad, Person, ...**)

Wiederholung MethodenHeader

[visibility] [static] [returnType] **methodenBezeichner** ([ParameterType] [parameterName], ...)

Wiederholung MethodenHeader

[visibility] [static] [returnType] **methodenBezeichner** ([ParameterType] [parameterName], ...)

- **methodenBezeichner**:
  - methoden beginnen **IMMER** mit einem Kleinbuchstaben
  - können im camelCase geschrieben werden
  - müssen die Methode beschreiben

Wiederholung MethodenHeader

[visibility] [static] [returnType] **methodenBezeichner** ([ParameterType] [parameterName], ...)

- **methodenBezeichner**:
  - methoden beginnen **IMMER** mit einem Kleinbuchstaben
  - können im camelCase geschrieben werden
  - müssen die Methode beschreiben
- Parameterliste
  - besteht aus beliebig vielen **Typ** + **bezeichner** Paaren = ein Parameter
  - bei mehr als einem Paar: Separation mittels Komma
  - sind die Platzhalten für Dinge, die in die Methode hinein gegeben werden
  - wenn die Methode keine Parameter braucht: muss man trotzdem eine leere Liste angeben: methodenBezeichner**()**

Wiederholung MethodenHeader

[visibility] [static] [returnType] **methodenBezeichner** ([ParameterType] [parameterName], ...)

Wiederholung MethodenHeader

[visibility] [static] [returnType] **methodenBezeichner** ([ParameterType] [parameterName], ...)

- wir brauchen **IMMER** einen returnType, einen **methodenBezeichner** und eine ParameterListe

Wiederholung MethodenHeader

[visibility] [static] [returnType] **methodenBezeichner** ([ParameterType] [parameterName], ...)

- wir brauchen **IMMER** einen returnType, einen **methodenBezeichner** und eine ParameterListe
- Bsp.:
  - void querSumme(){...}

Wiederholung MethodenHeader

[visibility] [static] [returnType] **methodenBezeichner** ([ParameterType] [parameterName], ...)

- wir brauchen **IMMER** einen returnType, einen **methodenBezeichner** und eine ParameterListe
- Bsp.:
  - void querSumme(){...}
  - public static int **calcSum** (int zahl1, int zahl2){...}

Wiederholung MethodenHeader

[visibility] [static] [returnType] **methodenBezeichner** ([ParameterType] [parameterName], ...)

- wir brauchen **IMMER** einen returnType, einen **methodenBezeichner** und eine ParameterListe
- Bsp.:
  - void querSumme(){...}
  - public static int **calcSum** (int zahl1, int zahl2){...}
  - private void **interneMethode** (Person person1){...}

Wiederholung MethodenHeader

[visibility] [static] [returnType] **methodenBezeichner** ([ParameterType] [parameterName], ...)

- wir brauchen **IMMER** einen returnType, einen **methodenBezeichner** und eine ParameterListe
- Bsp.:
  - void querSumme(){...}
  - public static int **calcSum** (int zahl1, int zahl2){...}
  - private void **interneMethode** (Person person1){...}
  - public static Car **copyCar** (Car car){...}

Wiederholung Konstruktoren

- Konstruktoren sind spezielle Methoden, die keinen Bezeichner haben und ein Objekt des Typen der Klasse, in der sie sich befinden zurück geben.
- Das ganze sieht so aus:

[visibility] [ClassType] ([ParameterType] [parameterName], ...)

Wiederholung Konstruktoren

- Konstruktoren sind spezielle Methoden, die keinen Bezeichner haben und ein Objekt des Typen der Klasse, in der sie sich befinden zurück geben.
- Das ganze sieht so aus:

[visibility] [ClassType] ([ParameterType] [parameterName], ...)

- Wenn wir also eine Klasse **Fahrrad** schreiben, dann hat sie folgenden Standardkonstruktor

public Fahrrad () {...}

Wiederholung Konstruktoren

- Konstruktoren sind spezielle Methoden, die keinen Bezeichner haben und ein Objekt des Typen der Klasse, in der sie sich befinden zurück geben.
- Das ganze sieht so aus:

[visibility] [ClassType] ([ParameterType] [parameterName], ...)

- Wenn wir also eine Klasse **Fahrrad** schreiben, dann hat sie folgenden Standardkonstruktor

public Fahrrad () {...}

- Der Aufruf eines Konstruktors erfolgt mittels **new Klassenname()**
- das erstellte Objekt muss jedoch in einer Variable abgelegt werden, was der linke Teil des Ausdrucks ist: Fahrrad meineVariable = **new Fahrrad();**

Wiederholung Konstruktoren

- Konstruktoren sind spezielle Methoden, die keinen Bezeichner haben und ein Objekt des Typen der Klasse, in der sie sich befinden zurück geben.
- Das ganze sieht so aus:

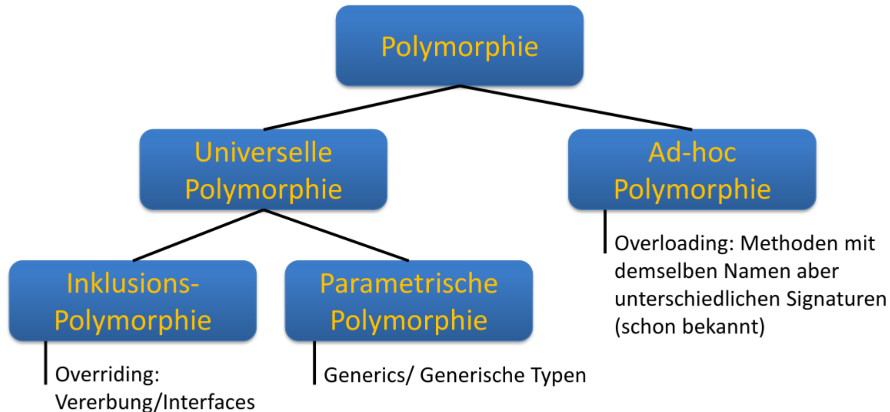
[visibility] [ClassType] ([ParameterType] [parameterName], ...)

- Wenn wir also eine Klasse **Fahrrad** schreiben, dann hat sie folgenden Standardkonstruktor

public Fahrrad () {...}

- Der Aufruf eines Konstruktors erfolgt mittels **new Klassenname()**
- das erstellte Objekt muss jedoch in einer Variable abgelegt werden, was der linke Teil des Ausdrucks ist: Fahrrad meineVariable = **new Fahrrad();**
- Konstruktoren können auch mittels *Ad-hoc Polymorphie* überladen werden.

Wiederholung Polymorphie



Quelle: Vorlesungsscript 9 Slide 9

Sonstiges

- Schreibt mit einem Kugelschreiber oder Füller - alles was mit Bleistift geschrieben ist, wird von uns ignoriert
- Wenn ihr mehr als eine Lösung für eine Aufgabe abgebt, markiert die, welche von uns bewertet werden soll. Wenn das nicht der Fall ist, suchen wir uns eine aus!
- Leserliches Schreiben ist von Vorteil - was wir nicht lesen können, können wir auch nicht bewerten.
- Lest die Aufgabenstellung genau!

Polymorphie part 2

Inklusionspolymorphie (Vererbung) Insel 5.8

Vererbung

- Grundprinzip der Objektorientierten Programmierung (OOP)
- Klassen *erben* Attribute und Methoden von übergeordneten Klassen
- Stichwort *extends*
- in der Regel spricht man von *Parent*-classes und *Child*-classes (siehe Vorlesung)

Vererbung - Insel 5.8

Sichtbarkeit von Attributen und Methoden

- Kennen wir schon aus früheren Übungen: *public*, *protected*, *private*
- **public** = Attribut / Methode einer Klasse mit diesem Keyword sind von allen anderen Klassen aus direkt sicht- und aufrufbar
- **protected** = Attribut / Methode einer Klasse mit diesem Keyword sind von allen anderen Klassen nicht sichtbar, außer von *child*-Klassen
- **private** = Attribut / Methode einer Klasse mit diesem Keyword sind von allen anderen Klassen nicht sichtbar, einschließlich *child*-Klassen

Vererbung - Insel 5.8

Overriding

- Methoden von Parent-classes können in Child-classes überschrieben werden
- Stichwort: *@Override*
- wird vor den Methodenheader geschrieben
- Wenn mehrere Klassen ein und die selbe Funktion durchführen können sollen, aber der Ablauf der Funktion unterschiedlich ist, sollte man in der übergeordneten Instanz angeben, dass es diese Funktion gibt und dann in den untergeordneten Instanzen konkret definieren
- Beispiel: alle 2D Körper haben eine Funktion *berechneFläche*, nur die Fläche eines Quadrates wird anders berechnet als die eines Kreises

Vererbung - Insel 5.8

Beispiel - Parent Class

```
public class Koerper {
  protected String name;
  public Koerper(String name) {
    this.name = name;
  }
  public float getArea() {
    return 0;
  }
  public void printName() {
    System.out.println(this.name);
  }
}
```

Vererbung - Insel 5.8

Beispiel - Child Classes

```
public class Quadrat extends Koerper { // abgeleitet von Koerper (heißt: Zugriff auf name,
  private float laenge;               // std. Konstr., getArea())
  public Quadrat(String name, float l) {
    super(name); // Aufruf des Konstruktors der parent-class
    this.laenge = l;
  }
  @Override // Überschreibung der vorhandenen Parent-Methode
  public float getArea() {
    return this.laenge * this.laenge;
  }
}
public class Circle extends Koerper {
  private float radius;
  public Circle(String name, float radius) {
    super(name);
    this.radius = radius;
  }
  @Override
  public float getArea() {
    return (float)(Math.PI * Math.pow(this.radius, 2));
  }
```

Vererbung - Insel 5.8

Beispiel Instanziierung

```
// Virtuelle Methoden
Quadrat q = new Quadrat("eins", 42); // neues Quadrat
Koerper k = new Quadrat("zwei", 5); // ein Quadrat ist automatisch auch ein Körper

Circle c = new Circle("kreis1", 12);
Koerper ck = new Circle("kreis2", 42);


q.printName();
System.out.println(q.getArea()); // Methode wurde in Koerper definiert
k.printName();
System.out.println(k.getArea());
c.printName();
System.out.println(c.getArea());
ck.printName();
System.out.println(ck.getArea());
```

Vererbung - Insel 5.8

Übungsaufgabe - 15min

1. Schreiben Sie ein Programm, das aus folgenden Klassen besteht: **Koerper2d, Kreis, Quadrat, Rechteck, Dreieck**
2. Nutzen Sie **Koerper2d** als Parent und den Rest als Child (möglicherweise gibt es auch eine Klasse die von einer anderen Klasse erben kann?)
3. Definieren sie die Funktionen für den Flächeninhalt und den Umfang der Koerper
4. Testen Sie ihr Programm in einer entsprechenden main-Methode

Interfaces

Interfaces - Insel 6.7

Beschreibung

- Setzt eine Bedingung für die Instanziierung von Objekten (Methoden im Interface **müssen** implementiert werden)
- Methoden können jedoch unterschiedlich realisiert sein
- Stichwort: *interface* und *implements*

```
public interface IKoerper {
  public float getArea(); // Was muss eine Klasse an Methoden Implementieren
  public void printName();
}
public class Circle implements IKoerper { // Implementation eines Interfaces
  public float getArea() { // konkrete Implementation des Interfaces
    return (float)(Math.PI * Math.pow(this.radius, 2));
  }
  public void printName() { // Ebenso wie hier
    System.out.println("Hallo! Ich bin " + this.name);
  }
}
```

Fragen?