Einführung in die Programmierung

by André Karge

Übung - Probeklausur

letzte Woche

- Vererbung
- Intefaces

diese Woche

- Besprechung Übungsblatt 8
- Besprechung Probeklausur
- Wiederholung Arrays
- Wiederholung Polymorphie

Übungsblatt 8

Übungsblatt 8

Aufgaben 1-3

Code Beispiel

Probeklausur

Aufgabe 1 - Modellierung

a) **static**

- 1 Punkt: Erklärung, wo **static** verwendet werden kann
- 1 Punkt: Wann **static** verwendet werden soll

Aufgabe 1 - Modellierung

a) **static**

- 1 Punkt: Erklärung, wo **static** verwendet werden kann
- 1 Punkt: Wann **static** verwendet werden soll

b) Handball

- 5 Punkte: auf die Implementation der Handball Liga
- Setzt sich zusammen aus:
  - Enum Position
  - Spiel-, Spieler-, Mannschaft-, Statistikklasse
  - Sichtbarkeit der Attribute und Methoden + static
  - Konstruktoren
  - Bester Spieler Funktion

Aufgabe 2 - Arrays

a) Heap vs. Stack

- 1 Punkt auf korrekte Antwort
- 1 Punkt auf korrekte Begründung

Aufgabe 2 - Arrays

a) Heap vs. Stack

- 1 Punkt auf korrekte Antwort
- 1 Punkt auf korrekte Begründung

b) call-by-reference vs. call-by-value

- 1 Punkt auf korrekte Antwort
- 1 Punkt auf korrekte Begründung

Aufgabe 2 - Arrays

c) Funktion

- 1 Punkt: Signatur
- 2 Punkte: check auf quadratisches Array + **null**-Ausgabe
- 2 Punkte: erfolgreiches Ermitteln des größten Wertes in i-ter **Zeile** an i-ter Stelle
- 2 Punkte: erfolgreiches Ermitteln des größten Wertes in i-ter **Spalte** an i-ter Stelle
- 1 Punkt für korrekte Ausgabe (Summe + Ausgabearray)

Aufgabe 3 - Polymorphie

a) Arten der Polymorphie

- 2 Beispiele für Arten von Polymorphie
- = 6 Punkte (Benennung, Funktion, Beispiel)

Aufgabe 3 - Polymorphie

a) Arten der Polymorphie

- 2 Beispiele für Arten von Polymorphie
- = 6 Punkte (Benennung, Funktion, Beispiel)

b) generischer Stack

- 8 Punkte setzen sich zusammen aus:
  - 1 Punkt: Klasse Stack mit generischem Typ und Konstruktor
  - 1 Punkt: Klasse Node mit generischem Typ, Konstruktor und next/prev Node
  - 2 Punkte: print Methode (Ausgabe + Schleife oder Rekursion)
  - 2 Punkte: push Methode (Funktion + checks)
  - 2 Punkte: pop Methode (Funktion + checks)

Aufgabe 4 - Algorithmus

Quersumme

- 2 Punkte: Korrektheit
- 1 Punkt: Schleife (For oder While)
- 1 Punkt: check < 0
- 1 Punkt: Signatur $+$ return

Arrays Wiederholung

Arrays

1-dimensionale Arrays

- zum Beispiel Vektoren

```
// Initialisierung
int[] myArray = new int[5]; // definierte Länge ohne Befüllung [0, 0, 0, 0, 0]
int[] myArray2 = {1,2,3,4}; // definierte Länge mit Befüllung  [1, 2, 3, 4]

// Befüllung (Indices starten bei 0)
myArray[0] = 42; // [42, 0, 0, 0, 0]
myArray2[myArray2.length - 1] = 666; // [1, 2, 3, 666]

// Iterieren über die Elemente eines Arrays
// mit for-loop
for(int i = 0; i < myArray.length; i++) { // i = ein Index (immer vom Typ int)
  myArray[i] = i + 1;
}
// mit iterator-for-loop
for(int i : myArray) { // i = ein Element aus dem Array, nicht! der Index (i = vom Typ des Arrayinhaltes)
  System.out.println(i);
}
```

Arrays

2-dimensionale Arrays

- zum Beispiel Matrizen

```
// Initialisierung
float[][] matrix = new float[3][3]; // 3x3 Matrix ohne Füllung
float[][] matrix2 = {{1,2,3},{4,5,6},{7,8,9}}; // 3x3 Matrix mit Füllung
/*
Spalte    0 1 2
          | | |
Zeile0 - [1,2,3]
Zeile1 - [4,5,6]
Zeile2 - [7,8,9]
*/
// Befüllung
matrix2[0][1] = 42; // 0te Zeile, 1te Spalte
matrix2[2][0] = 66; // 2te Zeile, 0te Spalte
/*
[1 ,42, 3]
[4 , 5, 6]
[66, 8, 9]
*/
```

Arrays

2-dimensionale Arrays

```
float[][] matrix2 = {{1,2,3},{4,5,6},{7,8,9}}; // 3x3 Matrix mit Füllung
// Iterieren über Elemente des 2-d Arrays
for(int i = 0; i < matrix2.length; i++) {  // loop über die Zeilen (Index für Zeilen)
  for(int j = 0; j < matrix2[i].length; j++) { // loop über die jeweiligen Spalten (Index für Spalten)
    System.out.println(matrix2[i][j]);
  }
}

// iterator-for-loop
for(float[] zeile : matrix2) { // Beachte: Schleifenelement ist ein 1d-Array vom Typ float (kein Index!)
  for(float spalte : zeile) { // Beachte: Schleifenelement ist ein float (kein Index!)
    System.out.println(spalte);
  }
}
```

Arrays

2-dimensionale Arrays

```
float[][] matrix2 = {{1,2,3},{4,5,6},{7,8,9}}; // 3x3 Matrix mit Füllung
// Iterieren über Elemente des 2-d Arrays
for(int i = 0; i < matrix2.length; i++) {  // loop über die Zeilen (Index für Zeilen)
  for(int j = 0; j < matrix2[i].length; j++) { // loop über die jeweiligen Spalten (Index für Spalten)
    System.out.println(matrix2[i][j]);
  }
}

// iterator-for-loop
for(float[] zeile : matrix2) { // Beachte: Schleifenelement ist ein 1d-Array vom Typ float (kein Index!)
  for(float spalte : zeile) { // Beachte: Schleifenelement ist ein float (kein Index!)
    System.out.println(spalte);
  }
}
```

Beachte: die Länge eines Arrays ist ein Attribut und kann einfach als solches aufgerufen werden (ohne Klammern):

```
int laengeDim1 = matrix.length;
int laengeDim2 = matrix[0].length;
```

call-by-value vs. call-by-reference

```java
public static void main(String[] args) {
  int a = 1;
  int[] b = {1,2,3,4,5};
  doSomething(a);
}

public void doSomething(int zahl, int[] arr) {
  zahl = 42;
  // zahl call by value oder call by reference?
  arr[0] = 42;
  // arr call by value oder call by reference?
}
```

call-by-value vs. call-by-reference

```
public static void main(String[] args) {
  int a = 1;
  int[] b = {1,2,3,4,5};
  doSomething(a);
}

public void doSomething(int zahl, int[] arr) {
  zahl = 42;
  // zahl call by value oder call by reference?
  arr[0] = 42;
  // arr call by value oder call by reference?
}
```

- zahl = Primitive → call-by-value (a in main wird nicht geändert)
- arr = Komplex → call-by-reference (b in main wird geändert!)

call-by-value vs. call-by-reference

```java
public static void main(String[] args) {
  int a = 1;
  int[] b = {1,2,3,4,5};
  doSomething(a);
}

public void doSomething(int zahl, int[] arr) {
  zahl = 42;
  // zahl call by value oder call by reference?
  arr[0] = 42;
  // arr call by value oder call by reference?
}
```

- zahl = Primitive → call-by-value (a in main wird nicht geändert)
- arr = Komplex → call-by-reference (b in main wird geändert!)

Merke: Primitive sind immer *call-by-value* und Komplexe sind immer *call-by-reference* (Auch Arrays)

Polymorphie Wiederholung

Polymorphie

Arten

- **Ad-Hoc** Polymorphie (bsp.: Überladen von Funktionen)
- Universelle Polymorphie:
  - **Inklusionspolymorphie** (bsp.: Inheritance - also Vererbung, Interfaces)
  - **Parametrische** Polymorphie (bsp.: Generics)

Überladen

```
public class Rechner {
  private int wert;
  public Rechner() { // Standardkonstruktor
    this.wert = 0;
  }
  public Rechner(int wert) { // Überladener Konstruktor
    this.wert = wert;
  }
  public int add(int a) { // erste add Funktion mit einem Wert
    return this.wert + a;
  }
  public int add(int a, int b) { // zweite add Funktion mit zweitem wert zum Abziehen
    return this.wert + a - b;
  }
  public int add(int[] a) { // dritte add Funktion mit Array
    int result = this.wert;
    for(int i : a) {
      result += i;
    }
    return result;
  }
}
```

Inheritance

```
public class Lebewesen {
  protected int alter;
  public Lebewesen() { // Konstruktor
    this.alter = 0;
  }
  public int getAlter() { // Getter für Alter
    return this.alter;
  }
  public void geburtstag() { // Setter für Alter (nur über den Geburtstag)
    this.alter++;
  }
}
public class Mensch extends Lebewesen { // Mensch = Lebewesen
  int anzahlAugen; // ein Mensch hat Augen
  public Mensch() {
    super(); // Aufruf des parent-Konstruktors
    this.anzahlAugen = 2; // genau Zwei Augen hat ein Mensch (Im Gegensatz zu einer Fliege)
  }
  public void feierGeburtstag() {
    geburtstag(); // vererbte Funktion
    System.out.println("Herzlichen Glückwunsch");
  }
```

Interfaces

- Interfaces beschreiben einen **Vertrag**, den alle Klassen zu erfüllen haben, die das Interface implementieren
- wenn das interface 2 spezifische Methoden vorraussetzt, müssen diese auch von den jeweiligen Klassen beschrieben werden

```
// Vertrag für Fahrräder
public interface Fahrrad {
  void gangWechsel(int gang);
  void bremse(int staerke);
}
```

Interfaces

```
public class BMX implements Fahrrad {
  private int anzahlRaeder;
  private int gaenge;
  private int aktuellerGang;
  private int geschwindigkeit;
  public BMX(int gaenge) { // Konstruktor
    this.anzahlRaeder = 2;
    this.gaenge = gaenge;
    this.aktuellerGang = 1;
    this.geschwindigkeit = 0;
  }
  public void gangWechsel(int gang) { // Erfüllung des Vertrags für Methode
    this.aktuellerGang = gang;
  }
  public void bremse(int staerke) { // Erfüllung des Vertrags für Methode
    if(this.geschwindigkeit - staerke >= 0) {
      this.geschwindigkeit -= staerke;
    }
  }
}
```

Interfaces

```
public class Dreirrad implements Fahrrad {
  private int anzahlRaeder;
  private int gaenge;
  private int aktuellerGang;
  private int geschwindigkeit;
  public Dreirrad(int gaenge) { // Konstruktor
    this.anzahlRaeder = 3; // Dreirrad hat aber 3 Räder
    this.gaenge = gaenge;
    this.aktuellerGang = 1;
    this.geschwindigkeit = 0;
  }
  public void gangWechsel(int gang) { // Erfüllung des Vertrags für Methode
    this.aktuellerGang = gang;
  }
  public void bremse(int staerke) { // Erfüllung des Vertrags für Methode
    if(this.geschwindigkeit - staerke >= 0) {
      this.geschwindigkeit -= staerke;
    }
    schlusslicht();
  }
}
```

Interfaces

Vorteil, man weiß jetzt, dass alle Klassen, die das Interface Fahrrad implementieren, die Funktionen *gangWechsel* und *bremse* haben und kann diese einfach aufrufen.

```
import java.util.List;
import java.util.ArrayList;
public class Test {
  public static void main(String[] args) {
    Fahrrad meinBMX = new BMX(12); // beachte: typ ist Fahrrad, Konstruktor ist BMX
    Fahrrad meinDreirrad = new Dreirrad(5); // typ ist Fahrrad, Konstruktor ist Dreirrad

    // man kann auch alle Fahrräder in einer Liste zusammenfassen:
    List<Fahrrad> meineFahrräder = new ArrayList<Fahrrad>();
    meineFahrräder.add(meinBMX);
    meineFahrräder.add(meinDreirrad);

    // nun können wir auf alle Fahrräder die Funktion "bremse" aufrufen,
    // da BMX und Dreirrad den Vertrag erfüllen
    for(Fahrrad f : meineFahrräder) {
      f.bremse(1000);
    }
  }
}
```

Generics

- Generics sind grob gesagt *variable Typen*
- werden benötigt, wenn man Funktionen / Klassen für beliebige Typen erstellen möchte
- bestes Beispiel sind hier Listen / Stacks / Queues
- Wir wollen nicht für jeden erdenklichen Typen eine neue Version einer Liste erstellen, die in der Lage ist diesen Typen zu speichern
  - in der Übung haben wir eine Liste programmiert, die integer Werte speichern konnte
  - Wenn wir jetzt eine Liste haben wollen, die float Werte speichern kann, müssten wir den kompletten Code der integer Liste kopieren und alle integer Typen mit float ersetzen
- Hier kommen Generics ins spiel, die uns diese Aufgabe zur Ausführzeit abnehmen

Generics

```
public class Node <meinVariablerTyp> { // <...> gibt an, dass die Klasse einen generic verwendet
  private meinVariablerTyp data; // Inhalt von unserem noch nicht definiertem Typ
  private Node<meinVariablerTyp> next; // Referenz auf das nächste Element in der Liste
  private Node<meinVariablerTyp> prev; // Referenz auf das vorherige Element in der Liste

  public Node(meinVariablerTyp data) { // Konstruktor
    this.data = data;
    this.next = null;
    this.prev = null;
  }
  public void setNext(Node<meinVariablerTyp> next) { // setter Next Node
    this.next = next;
  }
  public void setPrev(Node<meinVariablerTyp> prev) { // setter Prev Node
    this.prev = prev;
  }
  public Node<meinVariablerTyp> getNext() {...} // getter Next Node
  public Node<meinVariablerTyp> getPrev() {...} // getter Prev Node
  public meinVariablerTyp getData() {...} // getter data
  public void setData(meinVariablerTyp data) {...} // setter data
}
```

Generics

- Nun haben wir eine Klasse *Node*, die überall in ihrem Code einen Generic Typen verwendet
- Wenn wir jetzt einen generischen Liste haben wollen, müssen wir ebenfalls eine Klasse *Liste* definieren, die einen generischen Tpyen verwendet und diesen nutzt, um sich Nodes mit eben jenem Typen zu erstellen (Typ Kaskade)

```
public class Liste <T> {
  Node<T> first;
  Node<T> last;
  public List() {
    this.first = null;
    this.last = null;
  }
  public void push(Node<T> node) {...}
  public Node<T> pop() {...}
}
```

Damit geben wir an, dass Node<T> mit dem typen T initialisiert wird

(ein Stack mit T=int hat dementsprechend auch nodes mit meinVariablerTyp=int)

Lernt für die Prüfung!

Nutzt dafür das Vorlesungsscript, das Übungsscript, programmiert die Übungen erneut, wühlt euch durch *Java ist auch eine Insel*

Lernt für die Prüfung!

Nutzt dafür das Vorlesungsscript, das Übungsscript, programmiert die Übungen erneut, wühlt euch durch *Java ist auch eine Insel*

Fragen?