

Arrays und Datenkapselung

Programmieren Tutorium Nr.12

Aleksandr Zakharov | 9. Dezember 2025

Organisatorisches

Übungsblatt 2

Abgabe am 4.12. (um 6.00 Uhr)!

Übungsschein

Anmeldung: Bis zum 10.12. (nicht vergessen!)

Organisatorisches
●○

Wiederholung
○○○○○○○

Arrays
○○○○○○○

String
○○○○○

Referenzen
○○○○○

Geheimnisprinzip
○○○○○○○

Übungsaufgaben
○○○

Präsenzübung

Wie läuft das ab?

- 21.01.2026 17:30-19:00
- In Präsenz
- Einfache Programmieraufgaben auf Papier vorrechnen
- Lückentexte, "Lückencode", Programmieraufgaben auf Papier...

Wiederholung

Wahr oder Falsch?

```
1 // initialiser
2 while(condition) {
3   // body
4 }
5 // after
```

ist die Umformung von

```
1 for(initialiser ; condition ; after) {
2   // body
3 }
```

Falsch!

Das obere after gehört *in* die Schleife!

Wiederholung

Wahr oder Falsch?

Alle Schleifen können ineinander umgeformt werden.

Wahr!

Diese Aussage ist wahr.

Wiederholung

Wahr oder Falsch?

Ein if ist eine Schleife.

Falsch!

Ein if ist eine **VERZWEIGUNG**! Schleifen sind Konstrukte, die Quelltext mehrmals ausführen.

Wiederholung

Wahr oder Falsch?

In

```
1 for(int i : array)
```

nimmt `i` jeden *Wert* im Array an.

Wahr!

Der enhanced-for-loop gibt immer die Werte, nicht den Index zurück.

Formatierung

Einrückung

- Eine neue Zeile für jeden Befehl
- 4 Leerzeichen (keine Tabs!)
- Absetzen von Strukturen
 - Tiefere Einrückung kontrollierter Befehle (konsistent!)
 - Keine übertriebene Einrückung
 - Vor und hinter der Struktur (inkl. Kommentar) genau eine Leerzeile
- Strukturierung von Operatoren und Operanden durch Leerzeichen
- Leerzeilen zwischen Methoden (konsistent!)

Formatierung

Überlange Zeilen

- Passt ein Ausdruck nicht auf eine Zeile muss er umgebrochen werden
- Maximale Breite: 120 Zeichen, gewünscht maximal 100 Zeichen
- Hinter einem Komma, aber vor Operatoren
- Trenne die äußere Struktur
- Benutze Stringkonkatenation für überlange Zeichenketten
- Vermeide verwirrende Einrückungen

Formatierung

Beispiel

```
1 // Zu lang
2 int result = a.getA().getB().getC().getD().getE().getF();
3 // Nein!
4 int result = a.getA().getB().getC(
5     ).getD().getE().getF();
6 // Okay
7 int result = a.getA().getB().getC()
8     .getD().getE().getF();
9 // Okay
10 int result = a
11     .getA()
12     .getB()
13     .getC() // und so weiter
```

Arrays - Definition

Definition

- geordnete (nicht sortierte!) Sammlung von Elementen ...
- ... eines Typs: Alle Elemente haben den gleichen Typ
- Länge: Anzahl der Elemente (n)
- Nummerierung: Von 0 bis $n - 1$ (Indizierung started bei 0)

Beispiel

0.7	23.2	0.003	-12.7	1.1
0	1	2	3	4

"Maier"	"Mayr"	"Meier"	"Meyr"
0	1	2	3

Arrays - Deklaration

Deklaration

Es gibt zwei Varianten, Arrays zu deklarieren ...

- `Typ1[] arrayName = new Typ1[anzahlElemente];`
z.B. `float[] a = new float[8];`
- `Typ2[] arrayName = {element1, element2, ...};`
z.B. `int[] primeNumbers = {2, 3, 5, 7, 11};`

Häufiger Fehler

```
1 int[5] a;  
2 int[5] a = new int[5];  
3 int[5] a = {1, 2, 3};
```

Wieso?

Keine Zahl im Datentyp!

Organisatorisches
○○

Wiederholung
○○○○○○

Arrays
○●○○○○○

String
○○○○○

Referenzen
○○○○○

Geheimnisprinzip
○○○○○○○

Übungsaufgaben
○○○

Arrays - Selektion

Elemente selektieren

- `arrayName[elementIndex];`
- `primeNumbers[0]` liefert das erste Element
- *Was liefern: `primeNumbers[1]` und `primeNumbers[4]`?*
- *Was ist mit `primeNumbers[5]`?*

Arrays - Werte setzen

Wertzuweisung

- `arrayName[elementIndex] = wert;`
- z.B. `primeNumbers[0] = 13;`

Wie sieht jetzt die Belegung aus?

Arraylänge

Die Länge eines Array erhält man mit

- `arrayName.length;` // Achtung, Attribut, keine Methode!
z.B. `primeNumbers.length;` // liefert den Wert 5

Arrays - iterieren

Iterieren über Arrays

```
1 String[] array = {"Programmieren", "bereitet", "Freude"};
2
3 // for-loop
4 for (int i = 0; i < array.length; i++) {
5     System.out.println(array[i]);
6 }
7
8 // foreach-loop
9 for (String i : array) {
10     System.out.println(i);
11 }
```

Arrays - Objektlike

Achtung!

Arrays werden intern wie Objekte behandelt!
Also bitte beachten, wann Ihr eine Referenz und wann einen Wert erhaltet!

Organisatorisches
○○

Wiederholung
○○○○○○○

Arrays
○○○○○●○

String
○○○○○

Referenzen
○○○○○

Geheimnisprinzip
○○○○○○○

Übungsaufgaben
○○○

Aufgabe - Arrays

Arrays - einfach Array

Schreiben Sie eine Methode ...

- **public static int arraySum(int[] array),**
die die Summe der Zahlen des übergebenen Arrays als Rückgabewert hat.
- **public static double average(int[] array),**
die den durchschnittlichen Wert der Zahlen des übergebenen Arrays als Rückgabewert hat.

String (Wdh)

Was Strings so können ...

```
1 char charAt(int index);
2 int length();
3 String substring(int start, int end);
4 String toString();
5 String replace(char old, char newChar);
6 char [] toCharArray();
7 String valueOf(char []);
8 String toLowerCase();
9 String toUpperCase();
10 String trim();
11 String valueOf(int i);
12 String concat(string);
```

String (Wdh)

.toString()

- Jedes Objekt sollte sich durch die Methode `.toString()` mit einer Zeichenkette identifizieren
- Die Methode wird automatisch aufgerufen, wenn die Methode `print()` oder `println()` mit einer Objektreferenz aufgerufen werden

String (Wdh)

Beispiel .toString()

```
1 public class Player {  
2     private String name;  
3     private int    age;  
4  
5     @Override  
6     public String toString () {  
7         return "[name=" + name + ", age=" + age + "];"  
8     }  
9 }
```

StringBuilder / StringBuffer

Da gibt's auch noch ...

StringBuilder oft Mittel der Wahl (z.B. bei Stringkonkatenation in Schleifen) (*Link: String vs StringBuffer vs StringBuilder*). Auszugsweise (siehe auch zugehörige JavaDoc):

```
1 String append(String);
2 StringBuilder delete(int start, int end);
3 StringBuilder insert(int offset, any primitive or char []);
4 StringBuilder replace(int start, int end, String s);
5 StringBuilder reverse();
6 void setCharAt(int index, char ch);
7 String substring(int start);
8 String substring(int start, int end);
9 int indexOf(String searchKey);
10 int lastIndexOf(String searchKey);
```

Organisatorisches
○○

Wiederholung
○○○○○○○

Arrays
○○○○○○○

String
○○●○○

Referenzen
○○○○○○

Geheimnisprinzip
○○○○○○○○

Übungsaufgaben
○○○

Nicht vergessen!

Wichtig!

Strings immer mit `.equals` vergleichen!

Organisatorisches
○○

Wiederholung
○○○○○○○

Arrays
○○○○○○○

String
○○○○●

Referenzen
○○○○○

Geheimnisprinzip
○○○○○○○

Übungsaufgaben
○○○

Referenz (Wdh.)

Warum equals?

Warum eigentlich der Hinweis, dass man String nur mit equals vergleichen sollte?

Referenzen

- Werte von nicht-primitiven Typen sind Referenzen auf Objekte! (String ist nicht primitiv!)
- '==' prüft auf Wert-Gleichheit, bei Objekten also auf Identität.
- 'equals(Object o)' prüft auf inhaltliche Objektgleichheit und sollte für eigene Klassen immer implementiert werden, sofern Vergleiche nötig werden.
- Referenzdatentypen werden immer mit welchem Wert initialisiert? → 'null'

Referenz

Beispiel

```
1 public class Player {
2     private String name;
3     private int    age;
4
5     @Override
6     public boolean equals(final Object object) {
7         if (object != null && getClass().equals(object.getClass())) {
8             final Player player = (Player) object;
9             return name.equals(player.name) && age == player.age;
10        }
11        return false;
12    }
13 }
```


Referenz

Beispiel mit Objekten

```
1 Point p = new Point(5, 6);
2 Point q = new Point(5, 6);
3 Point r = p;
4 Terminal.println (p == q);
5 Terminal.println (p.equals(q));
6 Terminal.println (p == r);
7 Terminal.println (q == r);
```

Was wird ausgegeben und warum? (*Hinweis: `Point.equals == true` \iff beide Koordinaten identisch*)

Lösung ...

false, true, true, false

Referenz

.toString, .equals, .hashCode und .compareTo

- Nach Möglichkeit all diese implementieren
- Wird `.equals()` überschreiben, auf alle Fälle auch `.hashCode()`

.hashCode()

```
1 @Override
2 public int hashCode() {
3     // Vertrag zur equals beruecksichtigen und selben Attribute verwenden
4     return Objects.hash(color, id, name, etc);
5 }
```

Shallow Copy vs. Deep Copy

Betrachte folgende Methode:

```
1 public static int [] copy(int [] a) {  
2     //  
3     int [] b = new int[a.length];  
4     for (int i = 0; i < a.length; i++) {  
5         b[i] = a[i];  
6     }  
7     return b;  
8 }
```

Was macht diese Methode?

Organisatorisches
○○

Wiederholung
○○○○○○○

Arrays
○○○○○○○

String
○○○○○

Referenzen
○○○○●○

Geheimnisprinzip
○○○○○○○

Übungsaufgaben
○○○

Shallow Copy vs. Deep Copy

Betrachte nun folgende Methode:

```
1 public static Point[] copy(Point[] a) {  
2     //          ^^^^^  
3     Point[] b = new Point[a.length];  
4     for (int i = 0; i < a.length; i++) {  
5         b[i] = a[i];  
6     }  
7     return b;  
8 }
```

Was ist zu beachten, wenn die Arrayelemente nun vom Typ Point sind?

Aufgabe

Schreibe eine Methode **public static Point[] deepCopy(Point[] a)**, die eine Kopie des Arrays a als Rückgabewert hat und dabei auch alle im Array a enthaltenen Objekte kopiert.

Organisatorisches
○○

Wiederholung
○○○○○○○

Arrays
○○○○○○○

String
○○○○○

Referenzen
○○○○●

Geheimnisprinzip
○○○○○○○

Übungsaufgaben
○○○

Geheimnisprinzip

Allgemein

- Programmierschnittstelle (API): Programmteil, welcher von einem Softwaresystem anderen Programmen zur Anbindung an das System zur Verfügung gestellt wird
- Programmbibliothek (Library): Sammlung von Unterprogrammen und Routinen, die Lösungswege für thematisch zusammengehörende Problemstellungen anbieten
- Zur Bereitstellung solcher Schnittstellen gehört eine detaillierte Dokumentation aller Funktionen samt ihren Parametern

Geheimnisprinzip

Datenkapselung

- Attribute werden vor dem Zugriff von außen verborgen: Der direkte Zugriff auf die interne Datenstruktur wird unterbunden und erfolgt stattdessen über definierte Methoden
- Abschotten der internen Implementierung vor direktem externen Zugriff: Dieser darf nur über eine explizit definierte Schnittstelle erfolgen, um ihn unabhängig von den Implementierungsdetails zu machen
- Vorteile:
 - Die Implementierung von Klassen kann geändert werden, ohne die Zusammenarbeit mit anderen Klassen zu beeinträchtigen
 - Objekte können den internen Zustand anderer Objekte nicht in unerwarteter Weise lesen oder ändern

Geheimnisprinzip

Sichtbarkeit

- Mit Zugriffsmodifikatoren lassen sich die Sichtbarkeiten von Programmteilen regeln:
 - public: Element ist für alle Klassen sichtbar
 - private: Element ist nur innerhalb seiner Klasse sichtbar
 - protected: Element ist nur innerhalb seiner Klasse, deren Subklassen und allen Klassen im selben Paket sichtbar
 - default: Kein Modifikator bedeutet, dass das Element nur innerhalb seiner Klasse und der Klassen im selben Paket sichtbar ist

Geheimnisprinzip

Ab sofort

- Klassen sind in der Regel immer public
- Sämtliche Attribute einer Klasse sollten private sein!
- Bei Konstanten (static final) kann public sinnvoll sein
- Methoden sind in der Regel immer public
- Wenn Sie nur als lokale Hilfsmethoden gedacht sind, sollten sie private sein

Geheimnisprinzip

Sichtbarkeit & Methoden

- Schnittstellen-Methoden (public)
 - Schützen das Klassengeheimnis
 - Bieten Abstraktion über Implementierungsdetails
 - Haben eine Aufgabe, die mit dem Namen zusammenhängt
- Hilfsmethoden (private)
 - Sind funktionale und oder logische Einheiten in sich
 - Vermeiden Code-Redundanz

Geheimnisprinzip

Sichtbarkeit & Pakete

- Klassen kennen normalerweise nur die Klassen im eigenen Paket
- Falls eine Klasse ohne Paket-Angabe implementiert wird, befindet sie sich standardmäßig im unbenannten Paket
 - Eine im Paket befindliche Klasse kann jede andere sichtbare Klasse aus anderen Paketen importieren, aber keine Klassen aus dem unbenannten Paket
- Kein Zugriffsmodifikator bedeutet, dass ein Element nur innerhalb seiner Klasse und der Klassen im selben Paket sichtbar ist.
 - Verhält sich innerhalb eines Paketes wie: public
 - Verhält sich außerhalb eines Paketes wie: private

Geheimnisprinzip

Beispiel

```
1 package edu.kit.informatik;
2 public class Point {
3     private final int x;
4     private final int y;
5
6     public Point(int x, int y) {
7         this.x = x;
8         this.y = y;
9     }
10
11     public int getX() {
12         return this.x;
13     }
14 }
```

Organisatorisches
○○

Wiederholung
○○○○○○○

Arrays
○○○○○○○

String
○○○○○

Referenzen
○○○○○

Geheimnisprinzip
○○○○○○●○

Übungsaufgaben
○○○

Geheimnisprinzip

Beispiel

```
1  public int getY() {  
2      return this.y;  
3  }  
4  
5  @Override  
6  public boolean equals(final Object object) {  
7      if (object != null && getClass().equals(object.getClass())) {  
8          final Point point = (Point) object;  
9          return x == point.getX() && y == point.getY();  
10     }  
11     return false;  
12 }  
13 }
```

Modellierung einer To do Listen App

Aufgabentext

Um dein Studium zu organisieren möchtest du dir deine eigene To do Listen App schreiben (weil es ja nicht schon genug gibt). Die App soll mehrere Nutzer:innen verwalten können und diesen personalisierte E-Mails schreiben (mit Namen). Manche Nutzer:innen sind auch Admins.

Außerdem soll die App natürlich auch die Aufgaben selbst verwalten. Diese haben eine Wichtigkeit und eine Dringlichkeit $0 \leq w, d \leq 100$ und $w, d \in \mathbb{N}_0$ (mit $w = 0$ die höchste Wichtigkeit und $d = 100$ die am dringendsten zu erledigte Aufgabe bezeichnen), sowie ein:e Nutzer:in, der/die die Aufgabe erstellt hat. Außerdem hat jede Aufgabe einen Titel und eine Beschreibung. Die Aufgaben sollen nach Kategorien eingeteilt werden können. Jede Kategorie hat einen Namen und eine Farbe, die als Hexadezimalwert #000000 bis #FFFFFF gespeichert wird.

Aufgabe

1. Modelliere eine To do Listen App anhand des oben genannten Textes.

Organisatorisches
○○

Wiederholung
○○○○○○○

Arrays
○○○○○○○

String
○○○○○

Referenzen
○○○○○

Geheimnisprinzip
○○○○○○○

Übungsaufgaben
●○○

Aufgabe - Arrays (cont.)

Arrays - Vektorrechnung

- **public static double[] sum(double[] vectorA, double[] vectorB),**
die eine Vektoraddition auf den beiden übergebenen Vektoren durchführt und das Ergebnis als Rückgabewert hat. Achten Sie darauf, dass die beiden Vektoren hierzu dieselbe Länge haben müssen und dass Sie beim Berechnen der Summe weder vectorA noch vectorB verändern.
- **public static double[] scalarMult(double[] vectorA, double scalar),**
die den Vektor vectorA mit dem Skalar scalar multipliziert und das Ergebnis als Rückgabewert hat. Achten Sie auch hier darauf, dass vectorA unverändert bleibt.

Aufgabe - Arrays (cont.)

Arrays - Matrixrechnung

- **public static double[][] sum(double[][] matrixA, double[][] matrixB),**
die die Matrizen matrixA und matrixB addiert und das Ergebnis als Rückgabewert hat. Beachten Sie, dass hierzu die Dimensionen der Matrizen gleich sein müssen und dass auch hier weder matrixA und matrixB verändert werden sollen.
- **public static double[][] multiply(double[][] matrixA, double[][] matrixB),**
die die Matrizen matrixA und matrixB multipliziert und das Ergebnis als Rückgabewert hat. Beachten Sie auch hier die für die Multiplikation notwendigen Dimensionen der Matrizen und verändern Sie auch hier weder matrixA noch matrixB.