technische universität
dortmund

Fakultät für Informatik
Lehrstuhl 14
Lars Hildebrand, Alexander Lochmann,
Iman Kamehkhosh, Marcel Preuß, Dominic Siedhoff

Praktikum zu

Einführung in die Informatik für LogWiIngs und WiMas

Wintersemester 2016/17

Übungsblatt 0

Besprechung:
n.a.

Ergänzende Aufgaben

Blatt 1

Aufgabe 0.1: Einzelprobleme vs. Problemklassen cont.

Für jedes Einzelproblem bzw. jede Problemklasse aus Aufgabe 1.6: Geben Sie eine Problemklasse, in der das Einzelproblem enthalten ist, bzw. ein Einzelproblem aus der Problemklasse an.

Das Finden des kürzesten Weges zwischen zwei Städten

Das Berechnen der Primzahlzerlegung der Zahl 42

Das Finden aller direkten Wege zwischen zwei Städten

Das Berechnen des Volumens eines Würfels

Das Berechnen des Umfangs eines Kreises mit dem Radius 5

Aufgabe 0.2: Grundstruktur von Java-Programmen

Machen Sie sich über folgende Fragen Gedanken:

- Welche Gemeinsamkeiten haben die beiden Programme aus Aufgabe 4 und 5?

  public class Name , public static void main(...), System.out

- Welche grundlegende Struktur eines Java-Programmes können Sie entsprechend daraus ableiten?

```
public class ProgramName {
    public static void main(String[] args) {
        [...]
    }
}
```

Blatt 2

Aufgabe 0.3: Zahlensysteme extreme 1

Geben Sie folgende Dezimalzahl im Zahlensystem zur Basis 36 mit den Ziffernwerten 0–9 A–Z an:

677358_{10}

$677358/36 = 18815$	Rest I_{36}
$18815/36 = 522$	Rest N_{36}
$522/36 = 14$	Rest I_{36}
$14/36 = 0$	Rest E_{36}

$EINI_{36}$

Aufgabe 0.4: Zahlensysteme extreme 2

Rechnen Sie folgende Zahlen aus dem Hexadezimal- in das Oktalsystem um:

a) FEE_{16}

$FEE_{16}/8 = 1FD_{16}$	Rest 6_8
$1FD_{16}/8 = 3F_{16}$	Rest 5_8
$3F_{16}/8 = 7_{16}$	Rest 7_8
$7_{16}/8 = 0_{16}$	Rest 7_8

7756_8

b) 2015_{16}

$2015_{16}/8 = 402_{16}$	Rest 5_8
$402_{16}/8 = 80_{16}$	Rest 2_8
$80_{16}/8 = 10_{16}$	Rest 0_8
$10_{16}/8 = 2_{16}$	Rest 0_8
$2_{16}/8 = 0_{16}$	Rest 2_8

20025_8

Blatt 3

Aufgabe 0.5: Rückrichtung

Schreiben Sie ihren Code so um, dass Sie nun zuerst die Temperatur in °Fahrenheit im Quellcode angeben und die Temperatur in °Celsius umrechnen. Ändern Sie entsprechend auch die Ausgaben des Programmes.

```
1  package blatt03;
2
3  public class Thermometer {
4      public static void main(String[] args) {
5          double celsius;
6          double fahrenheit;
7          fahrenheit = 20.0;
8          System.out.print("fahrenheit = " + fahrenheit + "\n");
9          System.out.println("begin calculation:");
10         celsius = (fahrenheit - 32.0) * 5.0 / 9.0;
11         System.out.println("calculation finished");
12         System.out.println("celsius = " + celsius);
13     }
14 }
```

Blatt 4

Aufgabe 0.6: Zahlen-Ausschreibung

Betrachten Sie Ihren Code aus Aufgabe 5. Welche Änderungen sind nötig, um die Zahlen bis 99 richtig auszuschreiben?

Die Zahlen zwischen 0 und 20 müssen einzeln behandelt werden

Die Zahlen zwischen 20 und 99 bestehen immer aus dem selben Muster

Zehnerstelle + Einerstelle (z.B. twenty+four, twenty+five, etc.)

Blatt 5

Aufgabe 0.7: Verschachtelte Schleifen

In dieser Aufgabe sollen Sie sich mit der Verwendung zweier verschachtelter Schleifen vertraut machen.

Ein Tupel (a, b) ist eine Kombination zweier mathematischer Objekte zu einem einzelnen.

So ist z. B. das Tupel $(1, 2)$ das Tupel mit den Zahlen 1 und 2.

a) Schreiben Sie ein Programm namens **Tuples** im Paket **blatt05**, in dem Sie zwei Variablen n und m deklarieren und initialisieren und alle Tupel **unterschiedlicher** Zahlenpaare zwischen 1 und n für das erste Element des Tupels und zwischen 1 und m des zweiten Elementes ausgeben. Verwenden Sie dazu zwei ineinander verschachtelte Schleifen.

   Die Ausgabe *kann* für $n = 2$ und $m = 4$ z. B. folgendermaßen aussehen:

```
(1,2)
(1,3)
(1,4)
(2,1)
(2,3)
(2,4)
```

   Lösung:

```
1  package blatt05;
2
3  public class Tuples {
4      public static void main(String[] args) {
5          int n = 2;
6          int m = 4;
7          for(int i = 1; i <= n; i++) {
8              for(int j = 1; j <= m; j++) {
9                  if(i != j) {
10                     System.out.println("("+i+", "+j+")");
11                 }
12             }
13         }
14     }
15 }
```

b) Schreiben Sie ihr Programm so um, dass es nun ein Zahlenpaar nur genau einmal ausgibt, also anstatt z. B. die Tupel $(1, 2)$ und $(2, 1)$ auszugeben nur eines von beiden ausgibt.

   Hierfür können Sie annehmen, dass $m \geq n$ gilt.

   Die Ausgabe *kann* für $n = 2$ und $m = 4$ z. B. folgendermaßen aussehen:

```
(1,2)
(1,3)
(1,4)
(2,3)
(2,4)
```

Lösung:

```java
package blatt05;

public class Tuples {
    public static void main(String[] args) {
        int n = 2;
        int m = 4;
        for(int i = 1; i <= n; i++) {
            for(int j = i; j <= m; j++) {
                if(i != j) {
                    System.out.println("("+i+", "+j+")");
                }
            }
        }
    }
}
```

Blatt 6

Aufgabe 0.8: Programmstrukturierung

In dieser Aufgabe sollen Sie sich mit der Möglichkeit befassen, ein Programm durch Umstrukturierung in Subroutinen verständlicher zu machen.

Wir wollen ein Programm schreiben, das entscheidet, welcher von 3 Quadern das größte Volumen besitzt.

- Markieren Sie zuerst, in welchem Teil des Programmes sich komplexe, sich wiederholende Strukturen befinden.
- Lagern Sie diese Strukturen in Subroutinen aus, indem Sie das Programm umschreiben. Legen Sie dazu eine neue Klasse mit dem Namen **BiggestVolume** im Paket **blatt06** an.

```
1  package blatt06;
2
3  public class BiggestVolume {
4      public static void main(String[] args) {
5          double firstHeight = 16.5, firstWidth = 27.5, firstDepth = 38.0;
6          double secondHeight = 20.0, secondWidth = 20.0, secondDepth = 20.0;
7          double thirdHeight = 40.2, thirdWidth = 22.5, thirdDepth = 18.5;
8
9          if((firstHeight * firstWidth * firstDepth >=
10                 secondHeight * secondWidth * secondDepth) &&
11                 (firstHeight * firstWidth * firstDepth >=
12                 thirdHeight * thirdWidth * thirdDepth)) {
13             System.out.println("Volume number 1 is the biggest");
14         }
15         else if((secondHeight * secondWidth * secondDepth >=
16                 firstHeight * firstWidth * firstDepth) &&
17                 (secondHeight * secondWidth * secondDepth >=
18                 thirdHeight * thirdWidth * thirdDepth)) {
19             System.out.println("Volume number 2 is the biggest");
20         }
21         else {
22             System.out.println("Volume number 3 is the biggest");
23         }
24     }
25 }
```

Lösung:

- Es ist offensichtlich, dass die Volumenberechnung in eine Funktion ausgelagert werden kann.
- Genau so kann man die Berechnung des Maximums auslagern
- Das kann man natürlich auf verschiedene andere Arten machen, dies hier ist nur ein Vorschlag!

```java
package blatt06;

public class BiggestVolumeSolution {
    public static double volume(double height, double width, double depth) {
        return height * width * depth;
    }

    public static int maxOf3(double first, double second, double third) {
        if(first >= second && first >= third) {
            return 1;
        } else if(second >= first && second >= third) {
            return 2;
        } else {
            return 3;
        }
    }

    public static void main(String[] args) {
        double firstVolume = volume(16.5, 27.5, 38.0);
        double secondVolume = volume(20.0, 20.0, 20.0);
        double thirdVolume = volume(40.2, 22.5, 18.5);

        System.out.println("Volume number " +
            maxOf3(firstVolume, secondVolume, thirdVolume) + " is the biggest");
    }
}
```

Blatt 7

Aufgabe 0.9: Euklidscher Algorithmus

In dieser Aufgabe sollen Sie lernen, ein rekursives Programm zu analysieren und zu optimieren.

Der **größte gemeinsame Teiler** (ggT) zweier natürlicher Zahlen m und n ist die größte Zahl, durch die sowohl m als auch n teilbar ist.

Ein Algorithmus zur Berechnung des ggT ist der **euklidsche Algorithmus**. In diesem wird immer abwechselnd die kleinere Zahl von der größeren abgezogen, bis eine von beiden 0 ergibt. Die andere Zahl ist dann der ggT.

Eine rekursive Implementierung könnte so aussehen:

```
public static int euclid(int m, int n) {
    if(m == 0)
        return n;
    else if(n == 0)
        return m;
    else if(m > n)
        return euclid(m - n, n);
    else
        return euclid(m, n - m);
}
```

Diese Implementierung funktioniert, ist aber ineffizient. Im Laufe dieser Veranstaltung haben Sie eine Möglichkeit kennengelernt, die Rechenschritte dieses Algorithmus zu verkürzen.

Berechnen Sie per Hand `euclid(15, 42)`.

euclid(15, 42) = euclid(15, 27) = euclid(15, 12) = euclid(3, 12)

= euclid(3, 9) = euclid(3, 6) = euclid(3, 3) = euclid(3, 0) = 3

Was fällt Ihnen auf? Angenommen m sei nach der letzten Vertauschung größer als n, was haben Sie berechnet, wenn in diesem Rekursionsschritt m nicht mehr größer als n ist?

m **mod** n

Geben Sie eine rekursive Implementierung des euklidschen Algorithmus an, der sich diese Erkenntnis zu Nutze macht.

Lösung:

```
public static int euclid(int m, int n) {
    if(n == 0)
        return m;
    else
        return euclid(n, m % n);
}
```

Blatt 8

Aufgabe 0.10: Zeitmessung

In den vergangenen Aufgaben haben Sie die Fibonacci-Folge bereits kennengelernt. Hier wollen wir verschiedene Implementierungen vergleichen.

a) Überlegen Sie sich eine rekursive Definition der Fibonacci-Folge.

$$\textbf{fib}(x) = \begin{cases} 1 & \textbf{falls } x = 1 \vee x = 2 \\ \textbf{fib}(x-1) + \textbf{fib}(x-2) & \textbf{sonst} \end{cases}$$

b) Implementieren Sie in der Klasse **FibonacciComparison** eine Funktion **recFib**, die einen **int**-Wert **n** entgegen nimmt, die n-te Fibonacci-Zahl rekursiv berechnet und diese als **long** zurückgibt.

c) Machen Sie aus Ihrer iterativen Implementierung der Fibonacci-Folge aus Blatt 5 eine Funktion **itFib** mit den selben Eigenschaften.

d) Diese Implementierungen vergleichen wir nun miteinander:

- Die Funktion **System.nanoTime()** gibt einen Zeitstempel als **long**-Wert zurück. Die Differenz zweier solcher Zeitstempel ist die zwischen den Aufrufen vergangene Zeit in Nanosekunden. Diese können wir verwenden, um Laufzeiten von Programmabschnitten zu messen.
- Rufen Sie in der **main**-Funktion die rekursive und die iterative Berechnungsfunktion mit dem selben Wert auf.
- Messen Sie mit jeweils zwei Zeitstempeln die Laufzeit der jeweiligen Implementierung.
- Was fällt Ihnen auf? Was könnten Vor- und Nachteile der jeweiligen Implementierungen sein?

Bei größeren n ist die iterative Implementierung deutlich schneller.

Die rekursive Implementierung ist einfacher zu lesen

und näher an der mathematischen Definition.

Lösung:

```java
package blatt08;

public class FibonacciComparison {
    // b)
    public static long recFib(int n) {
        if(n <= 2) {
            return 1;
        } else {
            return recFib(n - 2) + recFib(n - 1);
        }
    }

    // c)
    public static long itFib(int n) {
        long fibLast = 0;
        long fibCurrent = 1;

        while(n > 1) {
            long tmp = fibLast;
            fibLast = fibCurrent;
            fibCurrent = tmp + fibCurrent;
            n--;
        }

        return fibCurrent;
    }

    // d)
    public static void main(String[] args) {
        int which = 40;
        long it, rec;
        long beforeIt, afterIt;
        long beforeRec, afterRec;

        beforeIt = System.nanoTime();
        it = itFib(which);
        afterIt = System.nanoTime();
        System.out.println("iterative: " + it +
                " in " + (afterIt - beforeIt) + " ns");

        beforeRec = System.nanoTime();
        rec = recFib(which);
        afterRec = System.nanoTime();
        System.out.println("recursive: " + rec +
                " in "  + (afterRec - beforeRec) + " ns");
    }
}
```

Blatt 9

Aufgabe 0.11: Vergleichende Methoden

In dieser Aufgabe sollen Sie Methoden programmieren, die die bisher geschriebenen Objekte miteinander vergleichen können.

Ergänzen Sie die beiden Klassen **Vehicle** und **Car** um jeweils eine öffentliche Methoden mit dem Namen **compare** und dem Rückgabetyp **int**. Die Methoden sollen jeweils ein Objekt des gleichen Typs entgegennehmen und das Objekt, auf dem die **compare**-Methode aufgerufen wurde mit dem übergebenen Objekt vergleichen:

- Das Vergleichskriterium für Vehikel ist ihre Reifenzahl.
- Das Vergleichskriterium für Autos ist ihre Motorleistung.
- Die Methoden sollen einen Wert größer 0 zurückgeben, wenn das aufrufende Objekt mehr Reifen bzw. Leistung hat, als das andere.
- Die Methoden sollen einen Wert kleiner 0 zurückgeben, wenn das aufrufende Objekt weniger Reifen bzw. Leistung hat als das andere.
- Die Methoden sollen 0 zurückgeben, wenn die Objekte gleich viele Reifen bzw. Leistung haben.

Lösung:

```
[...]
public int compare(Vehicle other) {
    if(this.wheels > other.wheels) {
        return 1;
    } else if(this.wheels < other.wheels) {
        return -1;
    } else {
        return 0;
    }
}
[...]
public int compare(Car other) {
    if(this.getPower() > other.getPower()) {
        return 1;
    } else if(getPower() < other.getPower()) {
        return -1;
    } else {
        return 0;
    }
}
[...]
```

Blatt 10

Aufgabe 0.12: Umsetzung

Setzen Sie die Änderungen der Aufgabe 10.8 in die Praxis um. Ändern Sie entsprechend auch Ihre Testfälle in der Testklasse.

Hinweise:

Das, was getan werden muss wurde in der Präsenzaufgabe vollständig diskutiert. Hier daher nur ein paar Hinweise:

```
public abstract class Vehicle {
    /* [...] */
    public abstract double getPower();
}
public class VehicleTest {
    public static void main(String[] args) {
        Vehicle aygo = new Car("Toyota", "Aygo", 45);
    }
}
```

Aufgabe 0.13: Fehlersuche

Betrachten Sie folgendes Programm, das auf die Klassen aus Aufgabe 10.**??** zurückgreift. Welche Fehler haben sich eingeschlichen?

```
1  package blatt10;
2
3  public class WrongCode {
4      public static void main(String[] args) {
5          Person people[] = new Person[3];
6
7          people[0] = new Employee("Robert", "Schneider");
8          System.out.println(people[0].toString());
9
10         people[1] = Person("Sabine", "Meier");
11         System.out.println(people[1].toString());
12
13         Student admin = new Employee("Anja", "Mueller", "Computer Science", 17
14         people[2] = admin;
15         System.out.println(people[2].toString());
16
17         Person theNewOne = new Student("Paul", "Thomas", 113862);
18         people[3] = admin;
19         System.out.println(people[3].toString());
20     }
21 }
```

Zeile 7: Der Konstruktor von Employee erwartet noch chair und salary

Zeile 10: das new-Keyword fehlt

Zeile 13: einer Student-Variable kann kein Employee-Objekt zugewiesen werden

Zeile 18/19: Das Array hat nur 3 Elemente

Blatt 11

Aufgabe 0.14: Liste von Objekten

Schreiben Sie eine alternative Implementierung der Liste, die in der Lage ist, Objekte wie z. B. **Vehicle** aus Blatt 10 zu verwalten.

Lösung:

Den Typen des **value**-Attributes auf z. B. **Vehicle** ändern. Entsprechend müssen alle Methoden, die dieses Attribut referenzieren auch ihre Parametertypen oder Rückgabetypen ändern, aber letztenendes ist es nur einfache Textersetzung an den passenden Stellen. In der Praxis würde man die Klassen mit sog. *Generics* versehen, um den verwendeten Typen dynamisch anzupassen.

```
public class Element {
    private Vehicle value;
    private Element next;

    public Element(Vehcile value) {
        this.value = value;
        this.next = null;
    }
    /* [...] */
    public Vehicle getValue() {
        return this.value;
    }
}
```

```
public class List {
    private Element first;
    private Element last;

    /* ... */

    public void add(Vehicle value) {
        /* ... */
    }

    public Vehicle get(int n) {
        /* ... */
    }

    public Vehicle remove(int n) {
        /* ... */
    }
}
```

Blatt 12

Ergänzende Aufgaben

Aufgabe 0.15: Höhe eines Baumes

Schreiben Sie eine Methode **height**, die die Höhe eines **BinarySearchTree**-Objektes zurückgibt.

Lösung:

```
public int height() {
    return this.height(root);
}

private int height(TreeNode node) {
    if(node == null) {
        return 0;
    }

    int leftHeight = height(node.getLeftChild());
    int rightHeight = height(node.getRightChild());
    if(leftHeight > rightHeight) {
        return leftHeight + 1;
    } else {
        return rightHeight + 1;
    }
}
```