TU München - Fakultät für Informatik
Lehrstuhl IV - Software & Systems Engineering

Prof. Dr. Alexander Pretschner

Wintersemester 2021/22

Musterlösung:
Einführung in die Informatik
IN0001 – 21.02.2022

Informationen zum Ablauf

- Bearbeitungszeit: **120 Minuten**
- Es sind keine Hilfsmittel erlaubt. Dies gilt insbesondere für elektronische Geräte und die Hilfe Dritter. Einzig ein Wörterbuch Muttersprache <-> Deutsch darf verwendet werden.

1	2	3	4	5	6	7	8	
21	10	12	10	10	17	20	20	Σ 120

(leer)

Aufgabe 1: Kurz und Knapp

/21

1.1: Call-Me-Maybe [/2]

Gegeben sei folgender Code:

```
1  import java.util.Arrays;
2  public class Increment {
3
4      static void inc1(int[] array){
5          array = new int[]{2,3,4,5};
6      }
7
8      static void inc2(int[] array){
9          array[0] += 1;
10         array[1] += 1;
11         array[2] += 1;
12         array[3] += 1;
13     }
14
15     static void inc3(int number){
16         number = number + 1111;
17     }
18
19     public static void main(String[] args) {
20         int[] array1 = new int[]{1,2,3,4};
21         int[] array2 = new int[]{1,2,3,4};
22         int number = 1234;
23
24         inc1(array1);
25         inc2(array2);
26         inc3(number);
27
28         System.out.println(Arrays.toString(array1));
29         System.out.println(Arrays.toString(array2));
30         System.out.println(number);
31     }
32 }
```

Hinweis: Arrays.toString() in Zeile 28 und 29 wandelt die Inhalte des Arrays in einen String um, sodass dessen Einträge in eckigen Klammern stehen und durch ein Komma getrennt sind.

Was wird in den jeweiligen Zeilen auf der Kommandozeile ausgegeben?

- Zeile 28: [1,2,3,4] (0,5Pkt)
- Zeile 29: [2,3,4,5] (0,5Pkt)
- Zeile 30: 1234 (0,5Pkt)

Die entsprechenden Ausgaben lassen sich dadurch begründen, dass Java sowohl Referenzen als auch Basistypen immer nach dem Prinzip call-by-value an Methoden übergibt. (0,5Pkt)

0,5Pkt je vollständig richtig ausgefüllte Lücke, sonst 0Pkt.
Wenn Wert statt value geschrieben wird, dann gibt es auch die 0,5Pkt.

1.2: (Un-)Gleichheit [/3.5]

```
1  public class NumberCompare {
2      public static void main(String[] args) {
3          X x1 = new X(5);
4          X x2 = new X(5);
5          X x3 = x1;
6          X x4 = new X(x1.x);
7
8          double d1 = Math.sqrt(5.0) * Math.sqrt(5.0);
9          double d2 = 5.0;
10
11         System.out.println(x1 == x2);
12         System.out.println(x1.equals(x2));
13         System.out.println(x1 == x3);
14         System.out.println(x1 == x4);
15
16         System.out.println(d1 == d2);
17     }
18
19 }
20 class X{
21     int x;
22
23     public X (int x){
24         this.x = x;
25     }
26 }
```

Gegeben sei der obige Code.

- Wählen Sie für Zeile 11, 13, 14 aus, ob `true` oder `false` ausgegeben wird. (1.5 Pkt)
- Erklären Sie im Falle von Zeile 12 kurz, warum `false` ausgegeben wird. (1 Pkt)
- Erklären Sie im Falle von Zeile 16 kurz das Problem mit dem Vergleich. (1 Pkt)

Zeile	`true`	`false`	Erklärung
11	☐	☑	
12	☐	☑	Dadurch, dass die equals Methode nicht überschrieben wurde, wird auch hier auf Objektgleichheit überprüft.
13	☑	☐	
14	☐	☑	
16	??	??	Da die Nachkommastellen vieler Doubles nicht präzise dargestellt werden können, muss gerundet werden, was dazu führen kann, dass ein Vergleich ein falsches Ergebnis liefert.

- 0.5Pkt je richtigen Haken in Zeile 11, 13, 14
- 1Pkt darauf, dass geschrieben wird, dass die equals Methode nicht überschrieben wird. Nur 0.5Pkt, falls nur geschrieben wird, dass auch hier die Objektreferenz verglichen wird.
- 1Pkt darauf, dass entweder doubles nicht präzise dargestellt werden können oder gerundet werden müssen.

1.3: (Un-)checked Exceptions [/1]

Gegeben seien folgende, voneinander unabhängige Klassen in verschiedenen Dateien.

```
1 class FooClass{
2     public void foo(int div){
3         int result = div / 0;
4     }
5 }
```

```
1 class BarClass{
2     public void bar(String path){
3         FileWriter writer = new FileWriter(path, true);
4     }
5 }
```

Angenommen, alle notwendigen `Importe` sind jeweils vorhanden, warum kompiliert die Klasse `FooClass` und die Klasse `BarClass` **nicht**? (Es können mehrere Antworten richtig sein.)

☑ Alle Runtime Exceptions sind unchecked exceptions.
☐ Alle Runtime Exceptions sind checked exceptions.
☐ Alle Exceptions **außer** Runtime Exceptions sind unchecked exceptions.
☑ Alle Exceptions **außer** Runtime Exceptions sind checked exceptions.

Aussagen 1 und 2 werden unabhängig von Aussagen 3 und 4 betrachtet.

- 0.5Pkt darauf, dass das erste, aber nicht das zweite Kästchen angekreuzt wurde.
- 0.5Pkt darauf, dass das vierte, aber nicht das dritte Kästchen angekreuzt wurde.

1.4: Handling Exceptions [/3]

Ergänzen Sie den folgenden Code auf die zwei verschiedenen, unten genannten Arten (a und b) so, dass die Methode trotz der möglicherweise geworfenen `FileNotFoundException` kompiliert. Gehen Sie hier nach den gelernten Best-Practice für Exception Handling vor.

Angabe:

```
1 public void foobar(){
2             FileReader fr = new FileReader("C:\\PinguDirectory\\fish.txt");
3 }
```

a) Propagieren Sie hier die Exception an die aufrufende Methode weiter.

- 0.5Pkt für throws und 0.5Pkt für FileNotFoundException.
- Es gibt **keine** Punkte, falls **nur** in b) eine FilesNotFoundException geworfen wird.
- Klammern() hinter throws FileNotFoundException werden ignoriert.
- throws **new** FileNotFoundException gibt einen halben Punkt Abzug
- 0Pkt für die a) falls dort ein try-catch Block und nichts geworfen wird.
- -0.5 falls zusätzlich zu der throws Angabe eine neue FileNotFoundException geworfen wird

```
1  public void foobar()                                                    {
2
3
4              FileReader fr = new FileReader("C:\\PinguDirectory\\fish.txt");
5
6
7
8
9
10 }
```

b) Geben Sie hier "Ups!" aus, falls eine `FileNotFoundException` auftritt.

- 0.5Pkt auf den try-Block (Keyword mit Klammern)
- 0.5Pkt auf den catch-Block (Keyword mit Klammern)- 0.5Pkt auf das Argument des catch-Blocks - **keinen Punkt**, falls nur eine andere oder die allgemeine Klasse Exception gefangen wird.
- 0.5Pkt auf die richtige Ausgabe.
- Sollte hier zusätzlich noch throws FileNotFoundException angegeben worden sein, dann gibt es trotzdem volle Punktzahl.

```
1     public void foobar()                                                  {
2
3
4          FileReader fr = new FileReader("C:\\PinguDirectory\\fish.txt");
5
6
7
8
9
10    }
```

Lösung für a und b:

```java
public void foobar() throws FileNotFoundException {
    FileReader fr = new FileReader("C:\\PinguDirectory\\fish.txt");
}

public void foobar(){
    try{
        FileReader fr = new FileReader("C:\\PinguDirectory\\fish.txt");
    }catch(FileNotFoundException e){
        System.out.println("Ups!");
    }
}
```

1.5: Evaluierung [/4.5]

a) Gegeben sei folgender Code-Ausschnitt links. Welchen Wert haben die vier Variablen nach der Ausführung? Sie können davon ausgehen, dass der Code-Ausschnitt kompiliert. (2 Pkt)

```
1 int w = 0;
2 int x = 10;
3 int y = 20;
4
5 w = w--;
6 int z = ++x + y--;
```

w: 0

x: 11

y: 19

z: 31

b) Betrachten Sie folgende Statements unabhängig voneinander. Geben Sie **entweder** den Wert der Variable nach der Zuweisung an **oder** kreuzen Sie "kompiliert nicht" an. (2.5 Pkt)

`double a = 10 /4;`	2.0	☐ kompiliert nicht
2 ohne ein Komma gibt null Punkte		
`int b = 5.4;`	_____	☑ kompiliert nicht
`String c = "false"+ 8 + 3 + false;`	"false83false"	☐ kompiliert nicht
`int d = 0%2 + (-8)%3;`	-2	☐ kompiliert nicht
`boolean e = 2 != 6.0/2.0 ? false : true;`	false	☐ kompiliert nicht

0,5Pkt je Lücke bzw. Kreuz.

- Die Anführungszeichen bei dem String sind egal
- Falls eine Antwort gegeben wurde **und** kompiliert nicht angekreuzt wird, dann null Punkte.

1.6: Interfaces & abstrakte Klassen [/4]

Kreuzen Sie je Aussage genau die eine richtige Antwort an. Geben Sie eine kurze Begründung an, falls Sie "falsch, weil" angekreuzt haben sollten. 0,5 je richtiges Kreuz / richtige Erklärung

Eine abstrakte Klasse	☐ kann ☑ kann nicht	instanziiert werden.
Eine Interface	☐ kann ☑ kann nicht	instanziiert werden.
Eine Klasse kann von	☑ genau einer ☐ maximal zwei ☐ beliebig vielen	anderen Klassen direkt erben.
Eine Klasse kann	☐ genau ein ☐ maximal zwei ☑ beliebig viele	Interfaces direkt implementieren.
Dadurch, dass ein Interface default Methoden besitzen kann, lässt es sich vermeiden, dass eine Klasse, die ein Interface implementiert, alle im Interface definierten Methoden implementieren muss.	☑ wahr ☐ falsch, weil	
Wenn eine nicht abstrakte Klasse von einer abstrakten Klasse erbt, müssen alle Methoden der abstrakten Klasse implementiert werden.	☐ wahr ☑ falsch, weil	nicht alle Methoden mit `abstract` gekennzeichnet werden müssen. Hier kann vieles gelten, z.B.: "Es müssen nur die nicht implementierten Methoden / die abstrakten Methoden überschrieben werden."
Alle Attribute von Interfaces sind implizit static und final.	☑ wahr ☐ falsch	

1.7: show() [/1]

```
1  public class MyInteger {
2      int myInt;
3
4      MyInteger(int myInt){
5          this.myInt = myInt+10;
6      }
7
8      public String show(){
9          return ""+this;
10     }
11
12     public static void main(String[] args) {
13         MyInteger ten = new MyInteger(10);
14         System.out.println(ten.show());
15     }
16 }
```

Was ist das Ergebnis der Ausführung des Codes? Es ist genau eine Antwort richtig.

☐ Es wird `20` auf der Kommandozeile ausgegeben.

☐ Das Programm kompiliert nicht.

☐ Es wird `""+20` auf der Kommandozeile ausgegeben.

☑ Es wird die Objektreferenz des MyInteger-Objekts ausgegeben(z.B. MyInteger@3b28bdfa).

Entweder einen oder null Punkte

1.8: Hallo Welt? [/2]

```
1  public class Example {
2      static String shared = "Hallo Welt";
3
4      static class AppendThread extends Thread{
5          private String append;
6
7          AppendThread(String append){
8              this.append = append;
9          }
10
11         public void run(){
12             shared = shared + append;
13         }
14     }
15
16     public static void main(String[] args) throws InterruptedException {
17         Thread exclamationMark = new AppendThread("!");
18         Thread fullStop = new AppendThread(".");
19
20         exclamationMark.start();
21         fullStop.start();
22
23         exclamationMark.join();
24         fullStop.join();
25
26         System.out.println(shared);
27     }
28 }
```

Wenn das Programm wiederholt ausgeführt wird, welche Ausgaben können für `shared` in Zeile 26 auftreten? Es kann mehr als eine Antwort richtig sein. Sie können davon ausgehen, dass das Programm keine Exception wirft.

- ☐ Hello World
- ☑ Hello World!
- ☑ Hello World.
- ☑ Hello World.!
- ☑ Hello World!.
- ☐ Hello World!!
- ☐ Hello World..

Hier ist ein Fehler in der Angabe. Der String shared im Code ist auf Deutsch, während der Aussgabestring auf Englisch ist. Deshalb gilt:

- Volle zwei Punkte, falls nichts angekreuzt ist. Falls die Aufgabe (nicht die Kreuze) allerdings sichtbar durchgestrichen wurde, dann gibt es null Punkte.
- Genau die vier richitgen angekreuzt gibt volle Punkte.
- Sonst: 0,5Pkt pro richtiges Kreuz und -1Pkt pro falsches Kreuz. Minimal kann es auf die Aufgabe 0 Punkte geben.

(leer)

Aufgabe 2: Bytecode

/10

Übersetzen Sie folgendes MiniJava-Programm auf der linken Seite **direkt** in Bytecode **ohne das Programm hierbei zu vereinfachen oder den Programmfluss zu verändern**. Eine Übersicht an MiniJava-JVM Bytecode-Befehlen finden Sie rechts daneben. Sie dürfen ausschließlich die dort gelisteten Befehle benutzen.
Hinweis: Sie können für Ihre Orientierung Kommentare wie in Java mit // einfügen. Dies ist allerdings **nicht** notwendig, um die volle Punktzahl zu erreichen.

```
1  int x;
2  x = readInt();
3
4  if(x > 10){
5      x = x - 10;
6  }
7  else{
8      x = x % 3;
9  }
10
11 write(x);
```

Befehlssatz der Mini-Java JVM

- NEG, ADD, SUB, MUL, DIV, MOD
- NOT, AND, OR
- LESS, LEQ, EQ, NEQ
- CONST i, TRUE, FALSE
- HALT
- LOAD i, STORE i
- JUMP i, FJUMP i
- READ, WRITE
- ALLOC i

Hinweis: Das untere Element auf dem Stack entspricht immer dem linken Operanden und das obere Element immer dem rechten!

Der Anfang gibt 1Pkt. Anstatt der drei Zeilen ist auch nur ein READ zulässig, dies bring dennoch den vollen Punkt.
ALLOC 1
READ
STORE 0

2Pkt auf den richtigen Vergleich (0.5Pkt jeweils für CONST und LOAD und einen Punkt, dass am Ende der richtige boolean auf dem Stack liegt (hier bitte auf die Semantik des Programms achten.))
CONST 10
LOAD 0
LESS

2Pkt für die richtige Verknüpfung der labels und jumps - 0,5Pkt Abzug für jeden Fehler
FJUMP else

0,5 Punkte je richtige Zeile der Subtraktion (=2Pkt)
LOAD 0
CONST 10
SUB
STORE 0

JUMP end
else:

0,5 Punkte je richtige Zeile der Modulorechnung (=2Pkt)
LOAD 0
CONST 3
MOD
STORE 0

end:

1Pkt auf folgenden Block, -0.5Pkt, falls ein Statement davon fehlt oder falsch ist.
LOAD 0
WRITE
HALT

Aufgabe 3: Mut zur Lücke

/12

Bitte vervollständigen Sie nachfolgendes Programm, indem Sie die leeren Boxen ausfüllen. Die Methode `int convert(String)` nimmt eine als String repräsentierte römische Zahl entgegen und gibt einen entsprechenden Integer zurück. Im System der römischen Zahlen gibt es sieben verschiedene Zahlzeichen, wobei jedem Zahlzeichen ein Wert zugeordnet ist (siehe `RomanToInt.java` Zeile 15 bis 21). Die Zahlzeichen werden der Reihe nach von links nach rechts betrachtet. Für jedes einzelne Zeichen gibt es drei verschiedene Fälle zu beachten:

- Auf das Zahlzeichen z, welches wir aktuell betrachten, folgt ein Zahlzeichen, dessen Wert kleiner oder gleich groß ist wie der von z. Hier wird der Wert von z zum Endergebnis addiert.
- Auf das Zahlzeichen z, welches wir aktuell betrachten, folgt ein Zahlzeichen, dessen Wert größer ist als der von z. Hier wird der Wert von z vom Endergebnis abgezogen.
- Auf das Zahlzeichen z, welches wir aktuell betrachten, folgt kein weiteres Zahlzeichen. Hier wird der Wert von z zum Endergebnis addiert.

Im Folgenden einige Beispiele:

$$\begin{aligned} \text{XII} &\rightarrow 10+1+1 &&= 12 \\ \text{XLIX} &\rightarrow -10+50-1+10 &&= 49 \\ \text{MDCXLIV} &\rightarrow 1000+500+100-10+50-1+5 &&= 1644 \end{aligned}$$

Zusätzlich überprüft die Methode `int convert(String)` in den Zeilen 24 bis 29, ob der übergebene `String roman` nur erlaubte Zeichen enthält. Ist dem nicht so, wird die in `IllegalCharException.java` implementierte Exception geworfen. Abgesehen von dieser potentiellen Fehleingabe können Sie für Ihre Lösung davon ausgehen, dass nur valide römische Zahlen übergeben werden.

Hinweis: Die Größe der Boxen lässt nicht zwangsläufig auf die Länge der richtigen Antwort schließen.

IllegalCharException.java

```
1  import ... // alle notwendigen Imports vorhanden
2
3  public class IllegalCharException [a (0.5)] {
4     private final char used;
5
6     // Konstruktor
7     [b (1.0)] {
8        [c (0.5)] .used = used;
9     }
10
11    @Override
12    public String toString() {
13       return "IllegalCharacterException: the following character is not allowed
             " + used;
14    }
```

RomanToInt.java

```
1  import ... // alle notwendigen Imports vorhanden
2
3  public class RomanToInt {
4     public static void main(String[] args) {
5        try {
6           // Was gibt Zeile 7 bei korrekter Lösung der Aufgabe aus? [d (1.0)]
7           System.out.println(convert("CXIX"));
8        } catch (IllegalCharException e) {
9           System.out.println(e.toString());
10       }
11    }
12
13    public static int convert(String roman) [e (0.5)]  {
14       Map<[f (0.5)] , [g (0.5)] > mapping = new HashMap<>();
15       mapping.put('I', 1);
16       mapping.put('V', 5);
17       mapping.put('X', 10);
18       mapping.put('L', 50);
19       mapping.put('C', 100);
20       mapping.put('D', 500);
21       mapping.put('M', 1000);
22
23       // Überprüfe ob der übergebene String nur Characters enthält, die in
             mapping definiert sind.
24       char[] romanArray = roman.toCharArray();
25       for ([h (1.0)] ) {
26          if (mapping.containsKey(ch) == false) {
27             throw new IllegalCharException(ch);
28          }
29       }
30
31       // Berechne den entsprechenden integer-Wert.
32       int result = [i (0.5)] ;
33       for (int i = 0; [j (1.0)] ; i++) {
34          if (i + 1 == [k (1.0)]  || mapping.get(romanArray[i])[l (1.0)]
                mapping.get(romanArray[i+1])) {
35             result [m (0.5)]  mapping.get(romanArray[[n (0.5)] ]);
36          }
37          else {
38             result [o (0.5)]  mapping.get(romanArray[[p (0.5)] ]);
39          }
40       }
41       [q (1)] ;
42    }
43 }
```

Groß-/Kleinschreibung wird nicht bewertet.

a) (0.5 Punkte): `extends Exception`
   Alternativen: `Throwable, RuntimeException`
b) (1.0 Punkte): `public IllegalCharException(char used)`
   Access Modifier beliebig;
   0.5 Abzug für fehlende/falsche Parameter;
   0.5 Abzug für falschen Namen (außer Typos);
   0.5 Abzug für Rückgabetyp
c) (0.5 Punkte): `this`
d) (1.0 Punkte): `119`
e) (0.5 Punkte): `throws IllegalCharException`
f) (0.5 Punkte): `Character`
   keine Punkte für `char`
g) (0.5 Punkte): `Integer`
   keine Punkte für `int`
h) (1.0 Punkte): `char ch : romanArray`
   0.5 Abzug für fehlendes `char`
i) (0.5 Punkte): `0`
j) (1.0 Punkte): `i < romanArray.length`
   0.5 Abzug für Klammern nach `length`
k) (1.0 Punkte): `romanArray.length`
   0.5 Abzug für Klammern nach `length`
   keine Punkte für `romanArray.length - 1` und Ähnliches
l) (1.0 Punkte): `>=`
m) (0.5 Punkte): `+=`
n) (0.5 Punkte): `i`
o) (0.5 Punkte): `-=`
p) (0.5 Punkte): `i`
q) (1.0 Punkte): `return result`

Aufgabe 4: Testen von Vierecken

/10

Gegeben ist ein Programm, das vier Parameter in einem Integer Array erwartet, die den vier Seitenlängen eines Vierecks entsprechen. Ausgabe des Programms ist, ob es möglich ist, aus den vier Seiten ein Quadrat (`SQUARE`), ein Rechteck (`RECTANGLE`), ein 3-gleichschenkliges Trapez (`THREE_EQUAL_SIDE_TRAPEZOID`) oder ein normales Viereck (`QUADRILATERAL`) zu erzeugen. Dabei wird immer der möglichst spezifischste Wert zurückgegeben. (Zum Beispiel ist jedes Quadrat auch ein Rechteck, aber das Programm gibt `SQUARE` zurück, da dies der spezifischste Wert ist.) Falls es nicht möglich ist, ein gültiges Viereck aus den vier übergebenen Seitenlängen zu erzeugen, gibt das Programm `INVALID` zurück.

```
1  public class QuadrilateralClassifier {
2
3     enum Type {
4        SQUARE,
5        RECTANGLE,
6        THREE_EQUAL_SIDE_TRAPEZOID,
7        QUADRILATERAL,
8        INVALID
9     }
10
11    public Type checkQuadrilateral(int[] sideLengths) {
12       // Testen auf ungültige Eingabeparameter
13       if (sideLengths == null || sideLengths.length != 4) {
14          throw new InvalidParameterException("Diese Methode erwartet ein int
                Array der Länge 4.");
15       }
16
17       // Sortieren des Arrays, sodass in a der kleinste und in d der größte Wert
18       // steht
19       Arrays.sort(sideLengths);
20       int a = sideLengths[0];
21       int b = sideLengths[1];
22       int c = sideLengths[2];
23       int d = sideLengths[3];
24
25       if (a <= 0) {
26          System.out.println("Kein gültiges Viereck");
27          return Type.INVALID;
28       }
29
30       if (a == d) {
31          return Type.SQUARE;
32       }
33
34       if (a == b && c == d) {
35          return Type.RECTANGLE;
36       }
37
38
```

```
39        if (a + b + c > d) {
40           // Durch die Sortierung in Zeile 18 gilt automatisch bei b == d auch
41           // b == c == d
42           if (a != b && b == d) {
43              return Type.THREE_EQUAL_SIDE_TRAPEZOID;
44           // Durch die Sortierung in Zeile 18 gilt automatisch bei a == c auch
45           // a == b == c
46           } else if (c != d && a == c) {
47              return Type.THREE_EQUAL_SIDE_TRAPEZOID;
48           }
49        } else {
50           System.out.println("Kein gültiges Viereck");
51           return Type.INVALID;
52        }
53
54        return Type.QUADRILATERAL;
55     }
56  }
```

4.1: [/8]

Für dieses Programm ist die kleinste Anzahl an Testfällen, mit der jede Codezeile mindestens einmal ausgeführt wird, gleich acht. Ihre Aufgabe ist es genau die acht Testfälle als JUnit 5 Testfälle in das nachfolgende Codebeispiel zu schreiben, die nötig sind, um jede Codezeile mindestens einmal auszuführen.
Hinweis: Schreiben Sie genau acht Testfälle, die genau obige Anforderung erfüllen. Wenn Sie weniger oder mehr als acht Testfälle implementieren, können Sie nicht die volle Punktzahl erreichen.

```
1  class QuadrilateralClassifierSolution {
2
3     private QuadrilateralClassifier sut;
4     private QuadrilateralClassifier.Type out;
5
6     @BeforeEach // setup() wird vor jedem Testfall einmal ausgeführt
7     void setup() { sut = new QuadrilateralClassifier(); }
8
9     @AfterEach // reset() wird nach jedem Testfall einmal ausgeführt
10    void reset() { out = null; }
11
12    @Test // assertThrows() überprüft, ob eine bestimmte Exception-Klasse beim
          Ausführen der Executable im zweiten Argument geworfen wird
13    void test1() {
14       assertThrows(InvalidParameterException.class, () ->
             sut.checkQuadrilateral(null
              ));
15    }
16
17
18    @Test
```

```
19    void test2() {
20       out = sut.checkQuadrilateral(new int[]{ 0, 0, 0, 0 });
21       assertEquals(out, QuadrilateralClassifier.Type. INVALID ); }
22
23    @Test
24    void test3() {
25       out = sut.checkQuadrilateral(new int[]{ 1, 1, 1, 1 });
26       assertEquals(out, QuadrilateralClassifier.Type. SQUARE ); }
27
28    @Test
29    void test4() {
30       out = sut.checkQuadrilateral(new int[]{ 1, 2, 2, 1 });
31       assertEquals(out, QuadrilateralClassifier.Type. RECTANGLE ); }
32
33    @Test
34    void test5() {
35       out = sut.checkQuadrilateral(new int[]{ 1, 1, 1, 2 });
36       assertEquals(out, QuadrilateralClassifier.Type. THREE_EQUAL_SIDE_TRAPEZOID
             ); }
37
38    @Test
39    void test6() {
40       out = sut.checkQuadrilateral(new int[]{ 1, 2, 2, 2 });
41       assertEquals(out, QuadrilateralClassifier.Type. THREE_EQUAL_SIDE_TRAPEZOID
             ); }
42
43    @Test
44    void test7() {
45       out = sut.checkQuadrilateral(new int[]{ 1, 3, 1, 1 });
46       assertEquals(out, QuadrilateralClassifier.Type. INVALID ); }
47
48    @Test
49    void test8() {
50       out = sut.checkQuadrilateral(new int[]{ 5, 2, 7, 10 });
51       assertEquals(out, QuadrilateralClassifier.Type. QUADRILATERAL ); }}
```

Zusätzlicher Platz für Aufgabe 4.1, falls benötigt.

Jeder korrekte Testfall 1P
Pro Testfall zuviel -0.5P (9 Testfälle mit 100% statement coverage = 7.5P)
1P für jeweils: (Achtung Werte im Array können beliebig vertauscht werden!)

- null, new int[0], new int[1]{1}, new int[]{0, 0}, new int[]{0, 0,}, new int[3], new int[5] (Kein Punkt für nicht kompilierendes wie z.B. {1, 2, 3}, new int[] = 1,2,3, oder int[] sidelengths) oder arrays der Länge 4
- (a,a,a,a) → SQUARE
- (a,a,b,b) → RECTANGLE
- (n,n,n,d>n) → THREE_EQUAL_SIDE_TRAPEZOID
- (a<n,n,n,n) → THREE_EQUAL_SIDE_TRAPEZOID
- (a,b,c,d) → QUADRILATERAL (Achtung: (1,1,1,5) ist kein QUADRILATERAL und (1,1,1,2) ist auch kein QUADRILATERAL)
- (0,x,x,x) → INVALID (negative Zahl(en) gehen auch anstatt 0)
- (a,b,c,d>=a+b+c) → INVALID

Bei Enum sind: Rechtschreibfehler, typos, Groß/Kleinschreibung, unterstriche egal.
Bei THREE_EQUAL_SIDE_TRAPEZOID sind auch kurzantworten wie “Three” und “TRAPEZOID” zulässig
Bei leerem Enum 0P
Nicht compilierender Test Input: 0P
Extra Platz auf Seite 19 checken!

4.2: [/1]

Ist die Praktik, die Sie gerade angewendet haben (Optimierung der Testfälle nach "jede Codezeile mindestens einmal ausgeführt") sinnvoll in der Praxis? Begründen Sie Ihre Antwort in einem kurzen Satz.

Antworten mit “Ja, ...” 0P
Nein, diese Praktik ist nicht sinvoll. (0.5P für “Nein”)
Coverage ist notwendig aber nicht hinreichend.
Außerdem gibt es keine zugrundeligende Fehlerhypothese gibt.
(0.5P für “nicht hinreichend” oder “Fehlerhypothese” oder)
(0.5P für sinvolle erklärung/begründung wieso “nein, ...” oder “nicht umbedingt, ...”)
keine sinvolle begründung:

- wir sollen stattdessen edge cases testen
- wir brauchen zu viele testfälle/ ist zu aufwendig

4.3: [/1]

Wann sind Testfälle gut? Antworten Sie in einem kurzen Satz.

- Nur 0P oder 1P für diese Aufgabe. Keine Halben Punkte!
- Keywords die zu 1P führen: “Testfall Semantik”, “edge cases”, “requirements based”, “classification tree”, “equivalence partitioning”
- Keywords die zu 1P führen: “potenzielle Fehler aufdecken”, “unentdeckten Fehler aufdecken”, “detecting undiscovered error/failure/fault”, “kosten Effizienz”, Sonderfälle)
- ODER antwort ähnlich zu:
- Testfälle sind dann gut, wenn sie potenzielle Fehler im Programm aufdecken können. (Antwort Möglichkeit 1)
- Ein guter Testfall deckt einen potenziellen Fehler mit guter kosten Effizienz auf. (Antwort Möglichkeit 2)
- Ein Testfall ist dann gut, wenn er mit hoher Wahrscheinlichkeit einen unentdeckten Fehler aufdeckt. (Antwort Möglichkeit 3)
- A good testcase is one that has a high probability of detecting an as yet undiscovered error. (Antwort Möglichkeit 3 Englisch)

Kein Punkt für:

- use cases

Aufgabe 5: Dynamic Dispatching - Hi Fisch

/10

Welche Ausgaben erzeugen Zeile 22 bis 30 im untenstehenden Programm? Schreiben Sie die exakten Strings in das Lösungsfeld direkt darunter. Betrachten Sie dabei jede Zeile isoliert. Sollte eine Zeile zu einem Fehler führen geben Sie das mit Begründung an und nehmen Sie für alle anderen Zeilen an, dass sie trotzdem ausgeführt werden.

```
1  public class DynamicDispatch {
2     static class Fisch {
3        String name;
4        public Fisch(String name) {this.name = name; write("Ich bin " + name);}
5        static void gruss(Fisch f) {write("Hi "); write(f.name);}
6        static void gruss(Hering h) {write("Hi "); write(h.name);}
7        void bildeSchwarm(Fisch f) {Fisch.gruss(f); write(" mein Fisch Schwarm");}
8        void bildeSchwarm(Hering h) {Fisch.gruss(h); write(" mein Hering Schwarm");}
9     }
10
11    static class Hering extends Fisch {
12       public Hering(String name) {super(name); write(", ein stolzer Hering");}
13       static void gruss(Fisch f) {write("Servus "); write(f.name);}
14       static void gruss(Hering h) {write("Servus "); write(h.name);}
15       void bildeSchwarm(Fisch f) {Hering.gruss(f); write(" mein Fisch Schwarm");}
16       void bildeSchwarm(Hering h) {Hering.gruss(h); write(" mein Hering Schwarm");}
17    }
18
19    static void write(String str) {System.out.print(str);}
20
21    public static void main(String[] args){
22       Fisch f = new Fisch("Frank");
          Ich bin Frank (0.5)
23       Hering h = new Hering("Hannelore");
          Ich bin Hannelore, ein stolzer Hering (1.0)
24       Fisch g = h;
          "" (keine Ausgabe) (0.5)
25       g.bildeSchwarm(h);
          Servus Hannelore mein Hering Schwarm (1.5)
26       write(((Fisch)h).name);
          Hannelore (1.5)
27       Fisch.gruss(h);
          Hi Hannelore (1.0)
28       Hering.gruss(f);
          Servus Frank (1.0)
29       f.bildeSchwarm((Fisch)h);
          Hi Hannelore mein Fisch Schwarm (1.5)
30       g.bildeSchwarm((Hering)f);
          Kompiliert nicht (und gleichbedeutende Aussagen) (1.5)
31    }}
```

- Eine Leere Abgabe gibt 0P
- Punkte für eine Lücke werden entweder voll oder gar nicht gegeben
- Ein teilweises aufschreiben der korrekten Ausgabe (z.B. “Hannelore” anstatt “Hi Hannelore”) gibt 0P
- Ob mit oder ohne Anführungszeichen / zu viele oder zu wenige ist egal
- Rechtschreibung, Kommasetzung egal
- 3. Lücke: Muss ausgefüllt sein, bspw “”, leer ,
- ODER bei Lücken 1 und 2 müssen sie versucht haben eine Antwort hinzuschreiben → dann gibt eine leere Lücke 3 0.5P
- Aufgabe missverstanden: z.B. "Lücke 3 kopmiliert nicht wegen isolierter Betrachtung"gibt 0P
- Selbes gilt für alle anderen Lücken: 0P bei kompiliert nicht, variable nicht definiert, oder nullptr exception
- Zeile 30: Alles gibt volle Punktzahl, was zeigt, dass es nicht klappt. Bspw. auch wenn irgendeine konkrete Exception angegeben wird.

(leer)

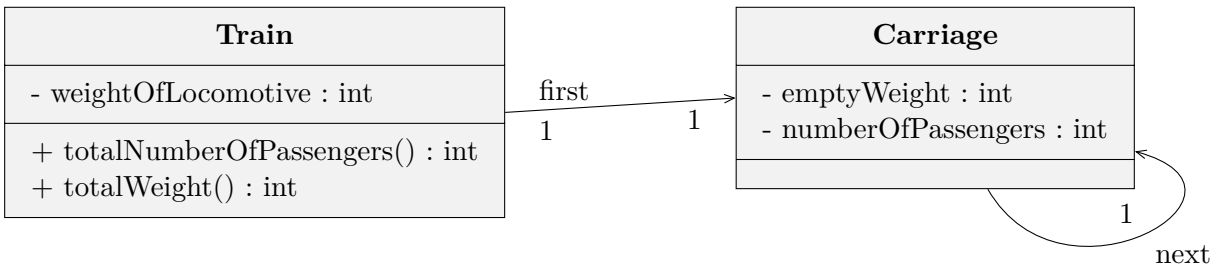
Aufgabe 6: Rekursion in vollen Zügen

/17

Die Firma *Antarctic Railways*™ bittet Sie, sie bei einem Software-Projekt zu unterstützen. Dabei sollen Züge und Personal verwaltet werden. Für einen Zug soll dabei eine rekursive Struktur verwendet werden, bei der der Zug seinen ersten Waggon kennt und dann jeder Waggon seinen nächsten. Im Falle des Personals soll ähnlich das Objekt, welches das Zugpersonal darstellt, nur das höchstrangige Mitglied kennen und dann jedes Mitglied des Personals nur seine direkten Untergebenen.
Ihre Aufgabe bei dem Ganzen ist es, insgesamt drei Methoden auf diesen beiden Strukturen zu implementieren. Dabei sollen (ebenfalls eigens implementierte) rekursive Hilfsmethoden verwendet werden. Füllen Sie jeweils die Lücken im Code aus!

6.1: Züge und Waggons [/11]

Ein Zug wird durch ein Objekt der Klasse `Train` dargestellt. Jedes `Train`-Objekt enthält dabei eine Referenz auf den ersten Waggon (`first`). Ein Waggon wird durch ein Objekt der Klasse `Carriage` repräsentiert, welches eine Referenz auf den jeweils nächsten Waggon (`next`) enthält. In einem Klassendiagramm sieht das Ganze folgendermaßen aus:



Gewichte werden stets in Kilogramm gemessen. Die Methoden in `Train` sollen Folgendes tun:

`totalNumberOfPassengers()`	Soll die Anzahl an Passagieren im Zug zurückgeben.	*(5 Pkt.)*
`totalWeight()`	Soll das Gesamtgewicht des Zuges berechnen. Dabei soll davon ausgegangen werden, dass eine Person stets 75 kg wiegt.	*(6 Pkt.)*

Implementieren sie für diese beiden Methoden in der Klasse `Carriage` je eine Hilfsmethode, welche die entsprechende Zahl für sie akkumuliert. Die **von `totalNumberOfPassengers()` und `totalWeight()` aufgerufenen Hilfsmethoden** sollen dabei **rekursiv** implementiert werden.
"Rekursiv" bedeutet hier im Speziellen Folgendes: Zu keinem Zeitpunkt darf innerhalb der Methoden `totalNumberOfPassengers()` und `totalWeight()` eine Referenz auf ein `Carriage`-Objekt außer `first` in einer Variable gespeichert oder überhaupt verwendet werden.
Ähnliches gilt auch für die von den beiden Methoden in `Train` aus aufgerufenen Hilfsmethoden in `Carriage`. Zu keinem Zeitpunkt darf in einer Methode in `Carriage` eine Referenz auf `Carriage`-Objekte außer `this` und `next` gespeichert oder verwendet werden.

Train.java

```
public class Train {
   // Attribute
   private int weightOfLocomotive;
   private Carriage first;

   // Methode, welche die Anzahl an Passagieren im gesamten Zug bestimmt
   // Hilfsmethode in Carriage muss rekursiv sein
    public int totalNumberOfPassengers() {
        // Der Fall 'first == null' darf, aber muss nicht behandelt werden.


        // (1 P: Korrekter Methodenaufruf auf 'first')
        return first.totalNumberOfPassengers();




    }

   // Methode, welche das Gesamtgewicht des Zuges bestimmt
   // Hilfsmethode in Carriage muss rekursiv sein
    public int totalWeight() {
        // Der Fall 'first == null' darf, aber muss nicht behandelt werden.


        // (1 P: Korrekter Methodenaufruf auf 'first')
        // (0,5P: 'weightOfLocomotive' addiert)
        return weightOfLocomotive + first.totalWeight();




    }
}
```

Carriage.java

```
public class Carriage {
   // Attribute
   private int emptyWeight;
   private int numberOfPassengers;
   private Carriage next;

   // Platz für Methoden
```

```
   // 1P: Rekursion wird unter richtigen Bedingungen abgebrochen
   //    und es wird die Anzahl an Passagieren zurückgegeben
   //     (Unabhängig davon, ob diese korrekt bestimmt wurde)
   // 1,5P: Der rekursive Aufruf ist korrekt
   // 1P: Die Anzahl an Passagieren wird korrekt berechnet
   // 0,5P: Aus der Rückgabe des rek. Aufrufs und der Anzahl
   //     an Passagieren wird der richtige Rückgabewert für
   //     den nicht rekursiven Fall berechnet
   public int totalNumberOfPassengers() {
      if(next == null) {
         return numberOfPassengers;
      }

      return numberOfPassengers + next.totalNumberOfPassengers();
   }

   // 1P: Rekursion wird unter richtigen Bedingungen abgebrochen
   //    und es wird das Gewicht dieses Waggons zurückgegeben
   // 1,5P: Der rekursive Aufruf ist korrekt
   // 1,5P: Das Gewicht wird korrekt berechnet
   // 0,5P: Aus der Rückgabe des rek. Aufrufs und dem Gewicht
   //     wird der richtige Rückgabewert für den nicht
   //     rekursiven Fall berechnet
   public int totalWeight(){
      int weight = emptyWeight + 75 * numberOfPassengers;
      if(next == null) {
         return weight;
      }

      return weight + next.totalWeight();
   }
```

```
}
```

In den Kommentaren über den jeweiligen Methoden ist die Punkteverteilung beschrieben. Desweiteren gelten noch folgende Regeln:

Regeln für `Train` und `Carriage`

Gewicht der Passenger vergessen	-1P
Gewicht der Lokomotive vergessen	-0,5P
Leergewicht vergessen	-0,5P

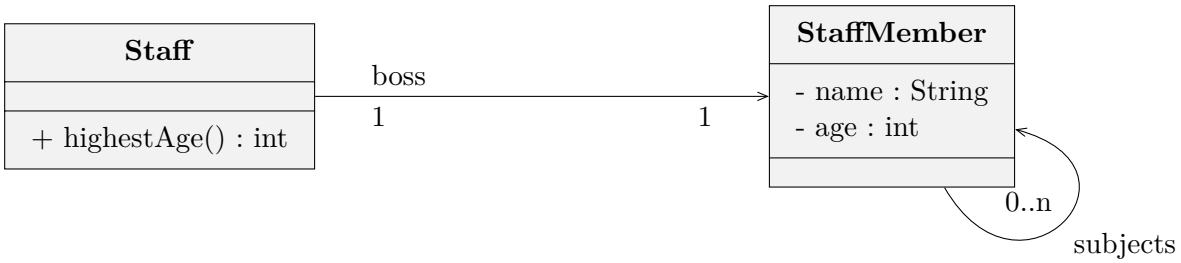
Regeln für beide Teilaufgaben

("global" heißt hier also "in beiden Teilaufgaben"!!)

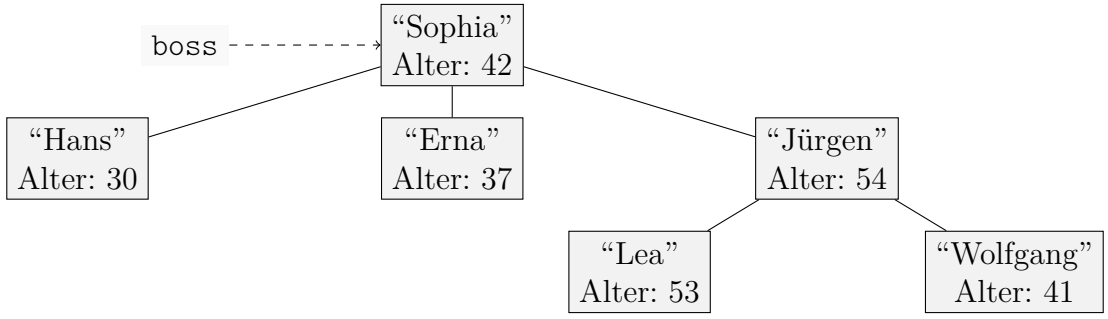
Fehler im Code wie = statt ==	-0.5P je Fehler, max. -2P global
Rekursiver Aufruf von der Form `recMethod(current.next)` statt `next.recMethod()`, also Durchlaufen der Liste über den Parameter der rekursiven Methode statt über den Caller, falls dadurch die Regel aus der Anganbe verletzt wird	einmal -2P
Abbruchbedingung fehlt	jedes Mal die Punkte dafür abziehen
Rein iterative Lösungen	0P auf die ganze Methode (inkl. Hilfsmethoden)

6.2: Personal und Mitarbeiter [/6]

Die Hierarchie des Personals soll durch Objekte der Klassen `Staff` und `StaffMember` abgebildet werden. Dabei dient `Staff` als Repräsentation der Gesamtheit der Hierarchie und unterhält einen Zeiger auf deren höchstgestelltes Mitglied (`boss`). Jedes Mitglied des Personals hat eine Liste mit Referenzen auf seine direkten Unterstellten (`subjects`). Das wird folgendermaßen in einem Klassendiagramm ausgedrückt:



Sie können davon ausgehen, dass `subjects` nie `null` ist. Alle anderen Referenzvariablen können den Wert `null` annehmen.
Man kann sich die Hierarchie der Mitglieder des Personals also wie einen Baum vorstellen - mit `boss` als Wurzel. Dann ist das `Staff`-Objekt der Baum und die `StaffMember`-Objekte sind die Knoten. Im Folgenden wird Ihnen eine Beispiel-Hierarchie visualisiert:



Sophia ist also `boss`. Ihre `subjects` sind Hans, Erna und Jürgen. Jürgen hat wiederum zwei weitere `subjects`, nämlich Lea und Wolfgang. Die Methode `highestAge()` in `Staff` nun soll Folgendes tun:

`highestAge()`	Soll das höchste Alter eines `StaffMembers` in der Hierarchie bestimmen. Wenn es kein `StaffMember` gibt, soll -1 zurückgegeben werden. (In obiger Beispiel-Hierarchie sollte also 54 zurückgegeben werden, da der Älteste - Jürgen - 54 Jahre alt ist.)	*(6 Pkt.)*

Auch hier soll die Methode `highestAge()` jeweils auf der Referenz `boss` auf das höchstrangige `StaffMember` eine rekursive Hilfsmethode aufrufen. Das heißt wieder, dass es verboten ist, eine Referenz auf ein `StaffMember`-Objekt außer `boss` in der Klasse `Staff` zu speichern oder zu benutzen. Außerdem dürfen in Methoden der Klasse `StaffMember` keine Referenzen auf `StaffMember`-Objekte außer `this` und die Objekte in der Liste `subjects` verwendet werden. Man darf also auf dem Knoten selbst und auf seinen Kindern Attribute referenzieren und Methoden aufrufen, nicht aber direkt auf den Kindeskindern, deren Kindern usw.

Nur zur Klarstellung: Schleifen sind generell erlaubt, solange sie nicht das eben Genannte verletzen. Man darf über Arrays, Listen etc. iterieren, nur eben nicht über die Kinder der Kinder, deren Kinder usw.

Staff.java

```
public class Staff {
   // Attribute
   private StaffMember boss;

   // Methode, die das höchste Alter eines Personalmitglieds berechnet.
   // Hilfsmethode in StaffMember muss rekursiv sein.
    public int highestAge() {
         // 1P: Korrekter Fall 'boss == null'
         if(boss == null) {
            return -1;
         }

         // 1P : Korrekter Methodenaufruf auf 'boss'
         return boss.highestAge();




    }
}
```

StaffMember.java

```
public class StaffMember extends Person {
   // Attribute
   private String name;
    private int age;
    private List<StaffMember> subjects;

    // Platz für Methoden
    // 0,5P: Rückgabewert kann 'this.age' sein
    // 1P: Schleife über 'subjects' korrekt
    // 1,5P: Ein rekursiver Aufruf pro Element in 'subjects'
    // 1P: Max. Alter der Teilbäume und Alter von this werden
    //    korrekt zusammengeführt
    public int highestAge() {
        int highestAge = age;
        for(StaffMember subject : subjects) {
            int highestAgeSubtree = subject.highestAge();
            if(highestAge < highestAgeSubtree) {
                highestAge = highestAgeSubtree;
            }
        }
        return highestAge;
    }

}
```

In den Kommentaren über den jeweiligen Methoden ist die Punkteverteilung beschrieben. Desweiteren gelten noch folgende Regeln:

Regeln für `Staff` und `StaffMember`

Test auf `getName() == null`	ignorieren

Aufgabe 7: Second-Hand Buchhandlung

/20

In dieser Aufgabe dürfen Sie ausschließlich Streams benutzen. Schleifen sind somit explizit verboten. Gegeben sei folgendes Code-Gerüst:

```
1  import ... // alle notwendigen Imports vorhanden
2
3  public class BookShop {
4
5      public static void main(String[] args) {
6          // Von Ihnen im Lückentext zu vervollständigen
7      }
8
9  }
10
11 static class Book{
12     String genre;
13     String title;
14     String author;
15     double resellPrice;
16
17     Book(String genre, String title, String author, double resellPrice){
18         this.genre = genre;
19         this.title = title;
20         this.author = author;
21         this.resellPrice = resellPrice;
22     }
23
24     //Getter und Setter sind vorhanden
25 }
```

Das Ziel der Aufgabe ist es eine, große Buchspende an einen Second-Hand Buchladen zu verwalten. Die `main`-Methode übernimmt als Parameter ein `String-Array` aus Beschreibungen der gespendeten Bücher. Jeder String, also jeder Eintrag, entspricht einem Buch. Die Buchattribute sind in diesem `String` durch ein Semikolon getrennt. Ein Buch-`String` im Array ist also nach folgendem Schema aufgebaut: `"genre; title; author; originalPrice"` Wenn die Buchspende zwei Bücher umfassen würde, dann würde `args` der `main`-Methode beispielsweise folgendermaßen aussehen:

```
1      String[] args = new String[]{
2          "Klassiker;Faust. Eine Tragödie;Johann Wolfgang von Goethe;10.00",
3          "Informatik;Weapons of Math Destruction;Cathy ONeil;18.00"};
```

a) Jeder Array-Eintrag soll zunächst in ein `Book`-Objekt umgewandelt werden. Alle so entstandenen Buchobjekte sollen gemeinsam in einer Liste gespeichert werden. Da die gespendeten Bücher gebraucht sind, soll der `resellPrice` hierbei 50% vom `originalPrice` betragen, falls das Buch vom `genre` "Klassiker" ist, ansonsten soll der `resellPrice` 70% vom `originalPrice` betragen.

b) Anschließend sollen die Bücher nach ihrem `genre` in eine `HashMap` eingefügt werden, damit sie einfacher in die Bücherregale (`shelves`) eingeordnet werden können.

c) Um die Münchner Informatikbildung in die Welt zu tragen, sollen die Titel (`title`) aller Bücher mit dem `genre` "Informatik", welche von "Alexander Pretschner" geschrieben wurden, alphabetisch sortiert (mit System.out.println()) ausgegeben werden, um als Buchempfehlung zu dienen.

Hinweise:

- Die Lücken werden unabhängig voneinander bewertet. Sie können also auf Zwischenergebnissen aufbauen, auch wenn Sie diese nicht oder falsch implementiert haben.
- Die vorgegebene Anzahl an Zeilen orientiert sich an unserer Lösung. Wenn Sie mehr oder weniger Zeilen brauchen, um mit Hilfe von Streams zu demselben Ergebnis zu kommen, ist dies in Ordnung.
- Am Ende des Codes befindet sich eine Übersicht an Funktionen, welche für Sie hilfreich sein können. Sie können auch andere Funktionen verwenden.
- Sie müssen **keine** Fehlerbehandlung implementieren und die Eingabe **nicht** auf Gültigkeit überprüfen.

Bei Klammern für Getter und auch für die stream() Methode darüber hinwegsehen, wenn die fehlen sollten.

```
1  public static void main(String[]args){
2      List<String> books = Arrays.asList(args);
3  // a) = 6 Pkt
4      //1 Pkt - 0.5 für die richtige Nutzung von books, 0.5 für stream()
5      List<Book> library=books.stream() .map(b->{
6          String[] atts =b.split(";");
7
8          /* 2Pkt: 0.5 falls double richtig geparsed wurde mit atts[3],
9          0.5 für die if-Bedingung mit atts[0], euquals und Klassiker
10            0.5 für jeweils den richtigen Endpreis (0,5 und 0,7)*/
11         double resellPrice=atts[0].equals(``Klassiker'')
12         ? 0.5*Double.parseDouble(atts[3])
13         : 0.7*Double.parseDouble(atts[3]);
14
15         /*2Pkt - 0.5 für new Book, 1.5 für Attribute, je Fehler 0.5 Abzug
16         Folgefehler der vorherigen Lücke werden berücksichtigt
17         Es gibt auch Punkte wenn das new Book in der vorherigen Lücke zurückgegeben wird.*/
18         return new Book(atts[0],atts[1],atts[2],resellPrice) ;
19         })
20         /*1 Pkt - 0.5 Pkt auf collect, 0.5Pkt auf das richtige Argument
21         - nur toList() gibt keine Punkte*/
22         .collect(Collectors.toList()) ;
23
24 // b) = 6.5 Pkt
25         HashMap<String, List<Book>>shelves=new HashMap<>();
26         List<String> distinctGenre=library
27             .stream()  // 0.5 Pkt
28             // 2Pkt - 1Pkt für map, 0.5 für Lambdaausdruck,  0.5 für getGenre()
29             .map(book->book.getGenre())
30
31             .distinct()  //1Pkt
32
33             //1 Pkt - 0.5 Pkt auf collect, 0.5Pkt auf das richtige Argument
34             .collect(Collectors.toList()) ;
35
36         distinctGenre.forEach(genre->shelves.put(genre,new ArrayList<>()));
37         // 2 Pkt - 0.5 je Lücke
38         library.stream().forEach (book ->shelves.get(book.getGenre() ).add(book ));
39
40
```

```
41 /*c) = 7.5 Pkt - 1Pkt Abzug, falls map und filter vertauscht wurden und man somit nicht
42     mehr auf den Autor zugreifen kann*/
43         shelves.get("Informatik")
44         .stream()  //0.5 Pkt
45
46         /*2.5 Pkt - 1Pkt für filter, 0.5 für Lambdaausdruck, 0.5 für getAuthor(), 0.5 für equals*/
47         .filter(book->book.getAuthor().equals(``Alexander Pretschner''))
48
49         //2 Pkt - 1Pkt für map, 0.5 für Lambdaausdruck,  0.5 für getTitle()
50         .map(book->book.getTitle())
51
52         .sorted()  //1Pkt - sort statt sorted gibt auch volle Punkte
53
54         /*1.5 Pkt - 1Pkt für forEach, 0.5 für Lambdaausdruck
55         Es gibt die Punkte auch für write statt System.out.println*/
56         .forEach(title->System.out.println(title)) ;
57 }
```

Aufgabe 8: Fakultätenpool

/20

Schreiben Sie einen Server, der über TCP eine Liste von zehn zufälligen Fakultäten zurück gibt. Gehen Sie davon aus, dass die Requests nur sporadisch ankommen und halten Sie deshalb einen Puffer an `BUFFER_SIZE` Fakultäten bereit. `THREAD_COUNT` Threads sollen den Puffer befüllen und schlafen, wenn mindestens `BUFFER_SIZE` Fakultäten im Puffer liegen. Der Server soll Fakultäten zwischen `1` und `MAX_FAC` berechnen und auf Port `PORT` auf Requests warten. Bei einem Request soll der Server Output wie beispielsweise diesen liefern:

```
Hello, your factorials are:
2!: 2
3!: 6
2!: 2
10!: 3628800
8!: 40320
1!: 1
4!: 24
3!: 6
9!: 362880
10!: 3628800
```

Hinweise:

- Schreibe Sie Ihre Lösungen zu den TODOs im Quelltext. Es ist genug Platz für die Lösung. Bitte bedenken Sie, dass weder der freie Platz noch die Punkte eins-zu-eins für eine Codezeile stehen!
- `Vector` ist im Prinzip ein dynamisches Array. Wichtige Methoden sind `add()`, um ein Element am Ende hinzuzufügen, `remove(0)`, um das erste Element aus dem Vector zu löschen und zurück zu geben, und `size()`, um zu erfahren wie viele Elemente aktuell im Vector sind.
- Vector ist threads-safe, das heißt, `add()`, `remove()` und `size()` müssen sie nicht für den parallelen Zugriff absichern. Andere Funktionen sind nicht notwendigerweise thread-safe und können deshalb ein Lock auf den Monitor benötigen.
- Verwenden Sie nur notwendige Locks, damit die Threads auch wirklich parallel arbeiten können.
- Wenn es keine Arbeit zu tun gibt, sollen Threads inaktiv sein.
- Denken Sie daran Exceptions abzufangen.

Gegeben sei folgende Klasse:

```
// imports ausgelassen
public class FacServer {
   private static ServerSocket serverSocket;
   private static Vector<Fac> facBuffer = new Vector<Fac>();
   public static final int BUFFER_SIZE = 100;
   public static final int MAX_FAC = 10000;
   public static final int PORT = 35000;
   public static final int THREAD_COUNT = 2;

   /* Berechne die Fakultät von n */
   public static BigInteger factorial(int n) {
      BigInteger result = BigInteger.ONE;
      for (int i = 2; i <= n; i++)
         result = result.multiply(BigInteger.valueOf(i));
      return result;
   }

   /* Datenstruktur um n und Ihre Fakultät zu speichern */
   private static class Fac {
      public int n;
      public BigInteger f;
      public Fac(int n, BigInteger f){
         this.n = n;
         this.f = f;
      }
   }

   private static class FacGenerator extends Thread
   { /* Implementierung in Aufgabe 8.1 */ }
   private static class FacServerHandler extends Thread
   { /* Implementierung in Aufgabe 8.2 */ }
   public static void main(String[] args) throws IOException
   { /* Implementierung in Aufgabe 8.3 */ }
}
```

8.1: Generator [/8]

Ergänzen Sie die Klasse `FacGenerator`, die in einem Thread den `facBuffer` befüllt und inaktiv wird, wenn der `facBuffer` voll ist.

```
// Die 'this.' sind alle optional!

// Fehlende Strichpunkte sollen keinen Abzug geben

// Klammern sind nur wichtig wenn sie offensichtilich falsch sind.

// Falls statt der Konstanzen die Zahl steht, gelten lassen.

// Falls sleep() statt wait: 0p

// Falls das try/catch über die ganze Funktion gezogen wird: volle Punkte geben

// wait() und notifyAll() müssen nicht auf den facBuffer laufen - gehen auch
// ohne. ABER dann muss der synchronized() block auch auf this laufen.
// also: synchronized(this){ wait()/notifyAll() }: 1p
// wenn einfach nur wait()/notify(), ohne etwas da steht: 0.5p.
// facBuffer.wait()/notifyAll(): 1p, auch wenn synchronized fehlt, weil der
// Monitor klar ist.


private static class FacGenerator extends Thread {
   private Vector<Fac> facBuffer;
   public FacGenerator( Vector<Fac> facBuffer){
      this.facBuffer = facBuffer;
   }

   public void run(){
      // endlos-event loop (1p)
      while(true){
         // erzeuge eine neue Zufallszahl
         int n = ThreadLocalRandom.current().nextInt(1, MAX_FAC + 1);
         // Fakultät berechnen
         Fac f = new Fac(n,factorial(n));

         // TODO prüfen ob wir mehr Zahlen in den Puffer füllen müssen (1p) und
         // wenn ja, füge f zum Puffer hinzu (1p) und wecke alle anderen
             Wartenden
         // Threads auf (2p)
         // Falls statt BUFFER_SIZE die Zahl 100 steht, gelten lassen
         if(this.facBuffer.size() < BUFFER_SIZE){ // if 1p
            this.facBuffer.add(f);     // add 1p
            // synchronized() kann auch um den ganzen block herum gelegt werden
            synchronized (this.facBuffer){ // synchronized: 1p
               this.facBuffer.notifyAll(); // notfiyall 1p
            }
         // TODO wenn wir keine neuen Zahlen brauchen, thread schlafen legen (3p)
```

```
        // falls das else fehlt, keine Punkte abziehen.
        }else{
           try{ // exception: 1p
              synchronized (this.facBuffer){ // synchronized 1p
                 this.facBuffer.wait();  // wait 1p.
              }
           }catch (InterruptedException e){
              System.out.println("Thread was interrupted");
           }
        }
     }
  }
}
```

8.2: Server [/8]

Schreiben Sie die Klasse `FacServerHandler`, die bei einem neuen TCP-Request als Thread gestartet wird (siehe auch die `main`-Methode unten). Sie gibt die zehn nächsten Fakultäten zurück. Falls es zu wenige Fakultäten gibt, wartet die Klasse, bis mindestens zehn im Puffer sind. Gibt es weniger als `BUFFR_SIZE` Fakultäten, weckt sie die `FacGenerator` Threads auf.

```
private static class FacServerHandler extends Thread {
   private Socket client;
   private PrintWriter out;
   private Vector<Fac> facBuffer;
   public FacServerHandler(Socket s, Vector<Fac> facBuffer){
      this.client = s;
      this.facBuffer = facBuffer;
   }
   public void run(){
      try {
         this.out = new PrintWriter(this.client.getOutputStream(), true);
         this.out.println("Hello, your factorials are: ");
         // Warte bis mindestens 10 Zahlen im Pool sind (4p)
         while(this.facBuffer.size() < 10){  // while: 1p
            try{                     // execption: 1p
         // wait() und notify() müssen sich auf den Zustand des
         // Prädikats, hier facBuffer, einig sein. Drum müssen
         // wait() und notify() jeweils in einen synchronized block
               synchronized (this.facBuffer){ // synchronized 1p
                  this.facBuffer.wait();  // wait: 1p
               }
            }catch (InterruptedException e){
               System.out.println("Thread was interrupted");
            }

         }
         // Nimmt 10 Zahlen aus dem Pool und schreibt sie in den output stream
         for(int i = 0;i < 10;i++){
            // Vector ist thread-safe, wir müssen den Zugriff nicht extra
            // absichern.
            Fac n = facBuffer.remove(0);
            // Zahl richtig ausgeben (1p)
            // println(n) o. ä. wo die formattierung nicht passt: 0p
            // Falls System.out.println, statt this.out: 0.5p abzug
            // Falls print() verwendet wird, schauen dass '\n' am Ende steht
            // Falls 'Hello, your factorials are:' ausgegeben wird, -0.5p
            // Falls write statt println: -0.5p
            this.out.println(n.n + "!: "+n.f);
         }
         // Falls weniger als 100 Zahlen im pool sind, wecke die
         // Generator threads auf (3p)
         if(this.facBuffer.size() < 100){  // if: 1p
            synchronized (this.facBuffer){ // synchronized 1p
               this.facBuffer.notifyAll(); // notify all: 1p
```

```
            }
         }
         // Stream und Socket beenden
         this.out.close();
         this.client.close();
      } catch (IOException e){
         System.out.println("Cannot write to socket. ");
      }
   }
}
```

8.3: main [/4]

Die main-methode soll die FacGenerator Threads starten und dann einen Socket öffnen und bei jedem TCP-Request auf einem ServerSocket einen FacServerHandler Thread starten.

```
public static void main(String[] args) throws IOException {
      // TODO: Generatorthreads ausführen (2p)
      // Falls Threads 'von Hand' und nicht in einer Schleife gestartet
      // werden: 0.5p abzug.
      for(int i = 0; i < THREAD_COUNT;i++)
         new FacGenerator(facBuffer).start();
      // ServerSocket öffnen
      serverSocket = new ServerSocket(PORT);
      // TODO: Neue Anfragen annehmen und in einem neuen FacServerHandler thread
          bearbeiten. (2p)
      // Hinweis: verwenden Sie serverSocket.accept() um eine
      // Anfrage anzunehmen
      while(true)
         // Wenn start() vergessen: -0.5p
         // Wenn parameter nicht passen: -0.5p
         // Die Zeile muss semantisch passen: wenn man es auf viele Zeilen
         // aufteilt, die aber falsch sind: 0p. Falls sie nur 'ein bisschen
         // falsch' sind: 0.5p
         new FacServerHandler(serverSocket.accept(), facBuffer).start();
}
```