Zusammenfassung

1. JAVA: Aufbau eines Programms und Syntax

1.1 Grundlegender Aufbau eines JAVA-Programms

Ein JAVA-Programm besteht aus (mindestens) einer Klasse. Innerhalb dieser Klasse brauchen wir die `main()`-Methode. Diese wird beim Starten des Programms aufgerufen.

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

Listing 1: Aufbau einer JAVA-Klasse

1.2 Befehle

Jeder auszuführende Befehl muss in JAVA mit einem ; Semikolon abgeschlossen werden.
Dagegen werden bei Verzweigungen, Schleifen und Methoden mehrere Befehle in Blöcken, welche mit { } eingeschlossen werden, zusammengefasst.

1.3 import

JAVA ist grundsätzlich modular in Paketen bzw. *packages* aufgebaut. Wollen wir Befehle und Objekte nutzen, die nicht im „Standardpaket“ von JAVA sind, so müssen wir die entsprechenden packages mit `import` einbinden. (s. beispielsweise *Eingabe über die Konsole*)
Wichtig: diese `import`-Befehle müssen **vor** der Klasse (`class Name { [...] }`) geschrieben werden!

1.4 Kommentare

Um Code verständlicher zu machen können in JAVA *Kommentare* benutzt werden. Diese werden von JAVA komplett ignoriert und können vom Programmierer dazu genutzt werden, Methoden und Befehle zu beschreiben.
Es gibt zwei Arten von Kommentaren. Beschreibe kurz den Unterschied:

`// [...]` bis Zeilenende

`/* [...] */` kann über mehrere Zeilen gehen

2. Variablen

Eine Variable ist ein Speicherplatz für Daten, die im Programm verwendet werden können. Bei der *Deklaration* gibt man der Variable einen Typ und einen Namen. Der *Datentyp* gibt an, welche Art von Werten in der Variable gespeichert werden über den der Variablenwert später wieder abgerufen und verändert werden kann.

2.1 Variablentypen

Beschreibe zunächst die verschiedenen Variablentypen und gib – sofern bekannt – bei den Datentypen für Zahlen den Zahlbereich an den diese Typen aufnehmen können:

`int` ganze Zahlen -2^{31} bis $+2^{31}-1$

`float` Dezimalzahlen $-3{,}4 \cdot 10^{38}$ bis $+3{,}4 \cdot 10^{38}$

`double` Dezimalzahlen $-1{,}7 \cdot 10^{308}$ bis $+1{,}7 \cdot 10^{308}$

`boolean` Wahrheitswerte 1/0 bzw true/false

`char` einzelne Zeichen

`String` Zeichenketten

Hinweis: Neben diesen Standarddatentypen gibt es noch viele weitere und wir können uns auch selbst neue Datentypen erstellen.

2.2 Deklaration und Initialisierung

Erzeuge eine Variable `jahr`, die einen ganzzahligen Wert speichern kann und weise ihr den Wert 2018 zu:

```
int jahr;
jahr = 2018;
```

Listing 2: Deklaration und Initialisierung einer Variablen

3. Ausgabe auf der Konsole

Damit Ergebnisse auch angezeigt werden, müssen wir diese natürlich ausgeben lassen. Die einfachste Ausgabe ist auf der Konsole, hierzu gibt es den Befehl:

```
System.out.println( .... );
```

Listing 3: Ausgabe auf der Konsole

4. Eingabe über die Konsole

Die einfachste Möglichkeit, wie wir Benutzereingaben von der Konsole einlesen können, ist mit einem `Scanner`-Objekt.
Hierfür müssen wir zunächst das Paket java.util.Scanner – wie im Punkt 1.3 beschrieben – importieren.
Anschließend erzeugen wir uns ein Objekt vom Typ `Scanner` und können dann die Benutzereingabe in eine Variable einlesen:

```
// Scanner-Objekt erstellen
Scanner sc = new Scanner(System.in);
// Einlesen einer ganzen Zahl
int a = sc.nextInt();
```

Listing 4: Eingabe über die Konsole

Hinweis: neben ganzen Zahlen können auch andere Datentypen eingelesen werden. Hierfür gibt es jeweils eine entsprechende Methode des `Scanner`-Objektes.

5. Verzweigungen

Verzweigungen diesen dazu, bestimmte Befehle bzw. Programmteile nur unter bestimmten Voraussetzungen auszuführen.
Wir haben bisher die `if-else`-Verzweigungen kennengelernt. Diese stellt eine einfache wenn-dann-sonst-Beziehung her: **Wenn** eine Bedingung erfüllt ist **dann** wird der erste Block ausgeführt **sonst** wird der zweite Block ausgeführt.

5.1 Bedingungen

Bedingungen können hierbei z. B. einfache numerische Vergleiche sein:

`a == b` Gleichheit von a und b

`a < b` **bzw.** `a > b` kleiner als bzw. größer als

`a <= b` **bzw.** `a >= b` kleiner oder gleich bzw. größer oder gleich

`a != b` Ungleich

5.2 Beispiel

Ergänze das Beispiel:

```
// Wenn Variable "jahr" größer oder gleich 2018
if(jahr >= 2018) {
// dann Ausgabe "Zukunft"
  System.out.println("Zukunft");
}
// sonst Ausgabe "Vergangenheit"
else {
  System.out.println("Vergangenheit");
}
```

Listing 5: `if-else`-Verzweigung

6. Methoden

Methoden sind vergleichbar mit mathematischen Funktionen: beim Aufruf übergibt man ihnen bestimmte *Parameter* und die Methoden lösen dann definierte Teilaufgaben. Das Ergebnis dieser Teilaufgaben kann beispielsweise eine Bildschirmausgabe oder ein *Rückgabewert* sein.
Eine Methode wird dazu benutzt, um wiederkehrenden Code zu *kapseln* und den Programmaufbau damit logisch zu strukturieren. Die Algorithmen müssen so nur ein einziges Mal programmiert werden und können immer wieder verwendet werden.

6.1 Aufbau einer Methode

Eine Methode sieht im Allgemeinen wie folgt aus:

```
public static boolean istSchaltjahr(int jahr) {
  if(((jahr%4)==0)&&(((jahr%100)!=0)||((jahr%400)==0))) {
    return true;
  }
  else {
    return false;
  }
}
```

Listing 6: Aufbau einer Methode

Erkäre kurz die folgenden Teile davon:

`public`: die Methode ist von überall aufrufbar (genaueres später bei der Objektorientierung)

`static` die Methode ist ohne Objekt (genaueres später bei der Objektorientierung)

`boolean` Rückgabetyp

`istSchaltjahr` Methodenname

`(int jahr)` „Eingabewert", Parameter

`{ [...] }` *Methodenrumpf* Auszuführende Befehle

`return [...];` Abbruch der Methode mit Ergebnis

7. Schleifen

Beschreibe, wozu man Schleifen verwendet und welche 2 Schleifenarten wir benutzt haben:

mit einer Schleife kann eine Befehlsfolge mehrfach durchgeführt werden.

1. Schleifenart: for - Schleife / Zählschleife

```
for (int i=0; i<10; i++) {

}
```

Initialisierung vor 1. Durchlauf

Bedingung (vor jedem Durchlauf)

Befehl nach jedem Durchlauf

2. Schleifenart: while - Schleife

```
while (i<10) {

}
```

Bedingung

Aufgabe:

Programmiere ein Programm, das mehrere Dreiecke ausgibt. Das Programm soll als erstes fragen wie viele Dreiecke es ausgeben sollen.

Anschließend soll gefragt werden, wie hoch jedes Dreieck sein soll.

Nachfolgend ein Programm-Ablauf-Beispiel zum besseren Verständnis:

```
Wieviele Dreiecke wollen Sie ausgeben? 2
Wie hoch soll jedes Dreieck sein? 3
*
**
***
*
**
***
```

Programmiere die Ausgabe eines Dreiecks als Methode

```
public static void dreieck(int hoehe) {
    […]
}
```

Rekursion

„Wer Rekursion verstehen will, muss vorher Rekursion verstehen."

Einführung

Unter einer *rekursiven Methode* versteht man eine Methode, die sich selbst wieder aufruft.
Damit diese Aufrufe nicht *unendlich* weitergehen und das Programm zu einem Ergebnis kommt, brauchen wir eine *Abbruchbedingung*.

1. Fakultät

Eine wichtige mathematische Funktion ist die Fakultät:

$$n! = 1 \cdot 2 \cdot 3 \cdots (n-2) \cdot (n-1) \cdot n$$

$$5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$$

Diese kann mit einer rekursiven Methode `fak` berechnet werden.

a) Notiere dir zunächst die rekursiven Methodenaufrufe und ein konkretes Zahlenbeispiel (für $n = 5$) dafür:

$5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5$

$5! = 4! \cdot 5$

$3! \cdot 4$

$2! \cdot 3$

$1! \cdot 2$

$fak(5) = 5 \cdot fak(4)$

$fak(4) = 4 \cdot fak(3)$

$fak(3) = 3 \cdot fak(2)$

$fak(2) = 2 \cdot fak(1)$ 1

$fak(1) = 1$

b) Wann bricht die Rekursion ab, d. h. wann wird nicht mehr erneut die Methode `fak` aufgerufen?

fak(n) → bei n = 1

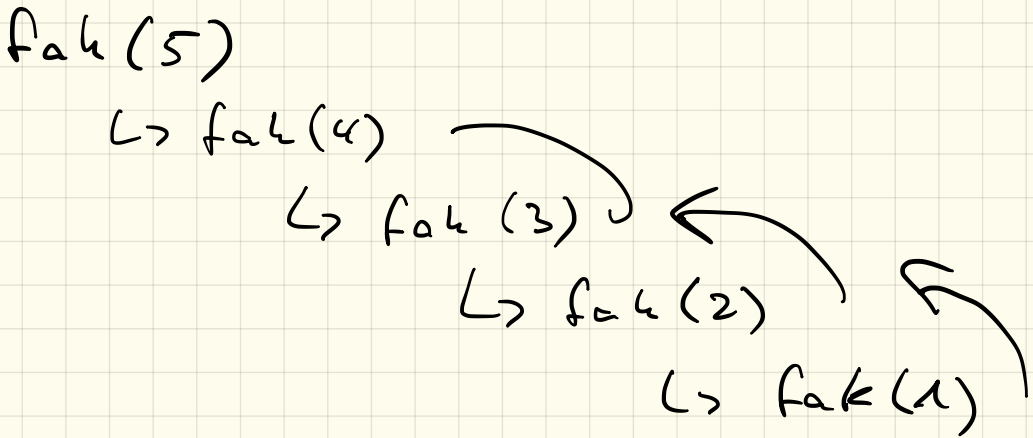
c) Programmiere die Methode `fak` in Java. (*Hinweis: diese soll wieder direkt programmiert werden, ohne ein Objekt anzulegen, also direkt unter die `main`-Methode.*)

d) Wie muss der Methodenkopf in Java aussehen?

Rückgabetyp, Parameter

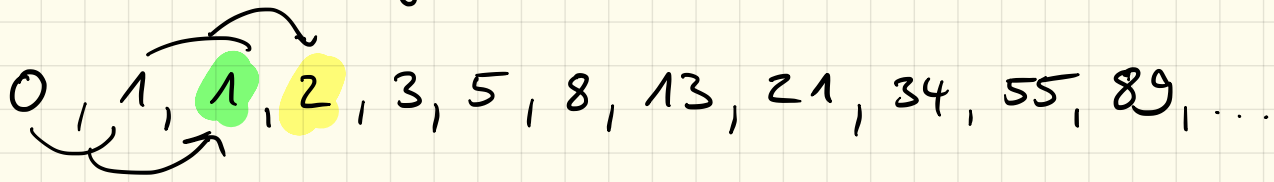
```
public static int fak(int n)
```

e) Rufe deine Methode `fak` aus der `main`-Methode auf mit folgenden Parametern: `fak(1);` (Ergebnis: 1), `fak(3);` (Ergebnis: 6), `fak(5);` (Ergebnis: 120), `fak(11);` (Ergebnis: 39 916 800)

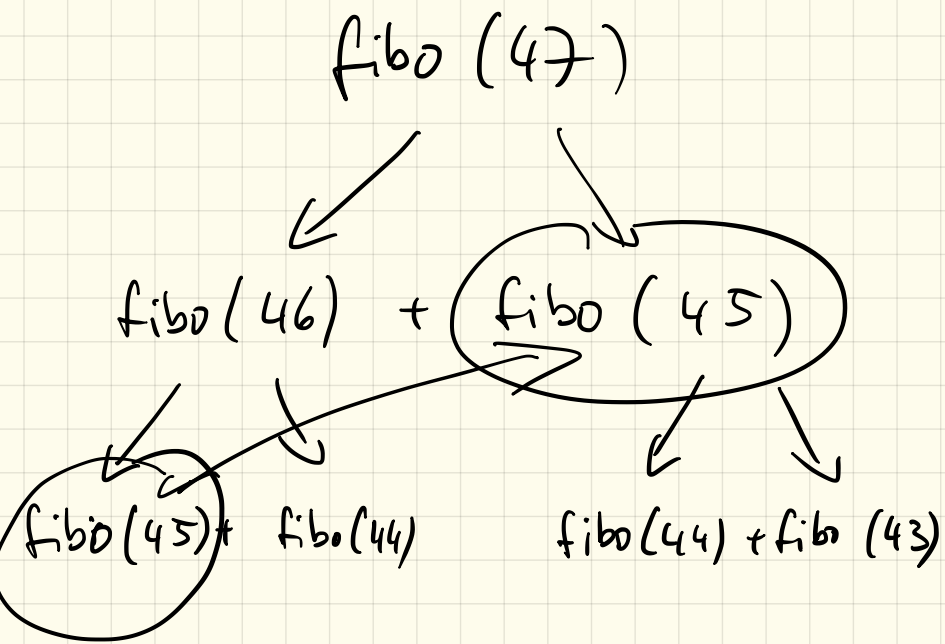

fak(5)
↳ fak(4)
↳ fak(3)
↳ fak(2)
↳ fak(1)

```
public static int fak(int n) {

    if(n == 1) {
        return 1;
    }
    else {
        return n * fak(n-1);
    }

}
```

Fibonacci-Folge




```
try {
    fibo(-5);
}
catch(Exception e){
    ||
}
public static int fibo(int n) {
    if(n<0) throw new Exception("Fehler");
}
```


fibo (47)
fibo(46) + fibo (45)
fibo(45) + fibo(44)
fibo(44) + fibo (43)

2. Fibonacci-Folge

Die Fibonacci-Folge

$$0,1,1,2,3,5,8,13,21,34,55,\ldots$$

ist folgendermaßen definiert:

- $F(0) = 0$
- $F(1) = 1$
- $F(2) = F(0) + F(1)$
- $F(3) = F(1) + F(2)$
- oder allgemein: $F(n) = F(n-2) + F(n-1)$

Diese soll nun mit einer Methode `fibo` umgesetzt werden.

a) Notiere dir zunächst die rekursiven Methodenaufrufe dafür und das Aufrufschema für $n = 5$:

b) Wann bricht die Rekursion ab, d. h. wann wird nicht mehr erneut die Methode `fibo` aufgerufen?

c) Wie muss der Methodenkopf in Java aussehen?

d) Programmiere die Methode `fibo` in Java. (*Hinweis: diese soll wieder direkt programmiert werden, ohne ein Objekt anzulegen, also direkt unter die `main`-Methode.*)

e) Rufe deine Methode `fibo` aus der `main`-Methode auf mit folgenden Parametern: `fibo(1);` (Ergebnis: 1), `fibo(3);` (Ergebnis: 2), `fibo(5);` (Ergebnis: 5), `fibo(11);` (Ergebnis: 89), `fibo(23);` (Ergebnis: 28 657)

3. Pascal'sches Dreieck

Das PASCALsche Dreieck sieht folgendermaßen aus:

$$
\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
1 & & 5 & & 10 & & 10 & & 5 & & 1 \\
 & 6 & & 15 & & 20 & & 15 & & 6 & \\
\ddots & & & & \vdots & & & & & & \ddots
\end{array}
$$

Die Zahlen darin sollen mit der Methode `binom(n,k);` berechnet werden, wobei n die Zeile und k die Spalte (innerhalb dieser Zeile) angibt. (Beispiele s. unten, *Hinweis: die Zeilen- und Spaltennummerierung beginnt bei 0!*)

a) Recherchiere im Internet, wie das PASCALsche Dreieck bzw. dessen Inhalte *rekursiv* berechnet werden können.

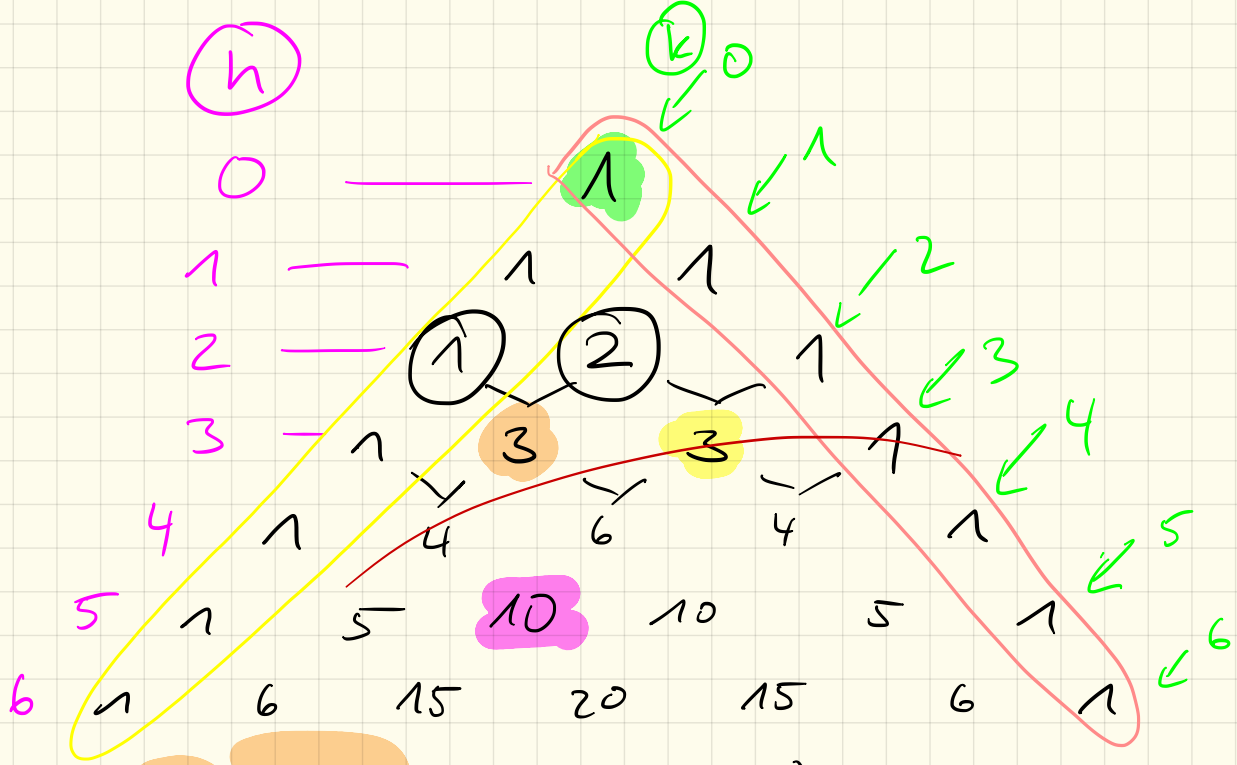
b) Notiere dir die rekursiven Methodenaufrufe und das Aufrufschema für $n = 3$ und $k = 2$:

c) Wann bricht die Rekursion ab, d. h. wann wird nicht mehr erneut die Methode `fibo` aufgerufen?

d) Wie muss der Methodenkopf in Java aussehen?

e) Programmiere die Methode `binom` in Java. (*Hinweis: diese soll wieder direkt programmiert werden, ohne ein Objekt anzulegen, also direkt unter die `main`-Methode.*)

f) Rufe deine Methode `binom` aus der `main`-Methode auf mit folgenden Parametern: `binom(0,0);` (Ergebnis: 1), `binom(2,1);` (Ergebnis: 2), `binom(5,3);` (Ergebnis: 10), `binom(9,4);` (Ergebnis: 126), `binom(20,7);` (Ergebnis: 77 520)



binom(3,1) = binom(2,0) + binom(2,1)

binom(n,k) = binom(n-1,k-1) + binom(n-1,k)

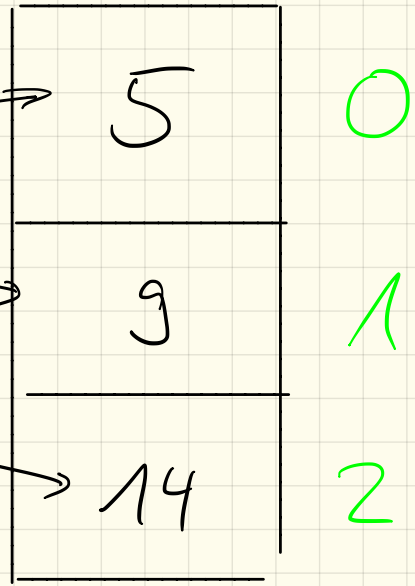
binom(n,0) = 1 / binom(n,n) = 1

19.4.18

Klausur: ~~3.5.~~ 5.7.

Array

```
int zahl1 = 5;
int zahl2 = 9;
int zahl3 = 14;
```



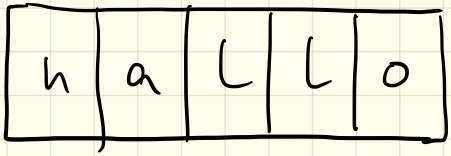

```
int[] zahlen = new int[3];
```

```
zahlen[0] = 5;
zahlen[1] = 9;
```

zahlen[3]

Index Out Of Bound Exception

String test = "hallo";



8. Arrays

Unter einem Array in Java versteht man einen „Container“, vergleichbar mit einem Schubladenschrank, der in der Lage ist, mehrere Objekte _____ aufzunehmen und zu verwalten.
Wir haben zwei unterschiedliche Möglichkeiten kennengelernt um Arrays zu erzeugen:

8.1 Deklaration und direkte Initialisierung

Vervollständige die Zeile um ein Array mit 5 Elementen zu erzeugen, das die Zahlen 1 bis 5 enthält:

```
int meinArray =
```

Listing 7: Arrays: direkte Initialisierung

Der Nachteil an dieser Möglichkeit ist, _____

_____

_____

8.2 Deklaration ohne Initialisierung

Um den oben genannten Nachteil zu umgehen können wir ein Array auch ohne Initialisierung deklarieren, beispielsweise mit Platz für 100 Werte:

```
int meinArray =
```

Listing 8: Arrays: Deklaration ohne Initialisierung

8.3 Arbeiten mit Arrays

Mit dem Ausdruck _____ können wir dann auf das dritte Element des Arrays zugreifen.
Achtung: der *Index* beginnt bei _____!
Die Länge eines Arrays `meinArray` können wir mit dem Ausdruck _____ bestimmen.

SelectionSort – Teil 1

1. sortierte Ausgabe

Erstelle in deinem Projekt `Sortierung` ein Paket `selectionsort`. Lege in diesem eine Klasse `Ausgabe` (inklusive `main`-Methode) an.
Implementiere ein Programm, welches:

- Ein Array mit 20 Zufallszahlen (zwischen 0 und 50) füllt.
- Dieses Array soll zunächst unsortiert ausgegeben werden.
- Mithilfe der Minimumsuche sollen wie Werte sortiert auf der Konsole ausgegeben werden.

Hinweis: du darfst natürlich den Code von letztem Mal nutzen und in die neue Klasse kopieren!

Beispielausgabe auf der Konsole:

```
Unsortiert:
20,6,30,34,5,11,0,34,28,12,4,26,11,15,44,28,40,7,20,7
Sortiert:
0
4
5
6
7
7
...
```

2. Abspeicherung

Erstelle im Paket `selectionsort` eine Klasse `OutOfPlace` mit einer `main`-Methode.
Programmiert werden soll ein Programm, welches wie oben ein zufällig befülltes Array generiert. Anstatt die Werte direkt auszugeben, sollen diese nun in einem *neuen* Array gespeichert werden, damit diese Werte später im Programm wieder weiterverwendet werden können.
Gib zur Kontrolle nach der Sortierung mithilfe einer Schleife das sortierte Array auf der Konsole aus.

Beispielausgabe auf der Konsole:

```
Unsortiert:
20,6,30,34,5,11,0,34,28,12,4,26,11,15,44,28,40,7,20,7
Sortiert:
0,4,5,6,7,7,11,11,12,15,20,20,26,28,28,30,34,34,40,44
```

3. Zusatzaufgabe

Informiere dich über die Begriffe *out-of-place* und *in-place*. Was bedeuten diese im Hinblick auf Sortieralgorithmen?
Hinweis: soll hier noch nicht schriftlich festgehalten werden!

7.6.2018

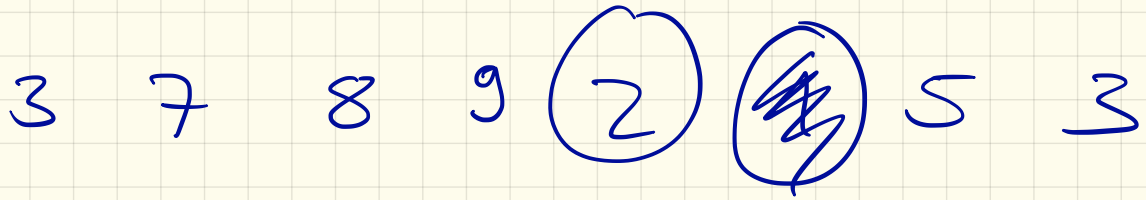
* ~~SelectionSort~~
* InsertionSort
* MergeSort
* BubbleSort
* SwapSort
* HeapSort
* QuickSort
* TimSort
* ...

- Verfahren vorstellen + zeigen
  Vor-/Nachteile
- Geschwindigkeit / Operationen im Vergleich mit anderen Verfahren
- Präsentation (~5 min)
- Handout (A4)

Recherche: 7.6.

Präsentation: 14.6.

Quellen



Anzahl: n

Durchgänge: n

1 2

Insgesamt $n \cdot n$ Operationen

$O(n^2)$

Klausur 5.7.

- Rekursion
- Sortierverfahren
  - SelectionSort
  - InsertionSort
  - Bubble Sort
  - Heap Sort
  - QuickSort
  - ~~Merge Sort~~

- Arrays
  - in-place / out-of-place
  - stabil

QuickSort 5 7 19 3 6 2

```
public static int[] quicksort(int[] unsortiert)
{   if (unsortiert.length <= 1) -> Abbruch
    int pivot = unsortiert[0];
    int[] links = new int[unsortiert.length-1];   int l = 0;
    int[] rechts = new int[unsortiert.length-1];  int r = 0;
    for(int i = 1 ; i<unsortiert.length; i++){
        if (unsortiert[i] < pivot){
            links[l] = unsortiert[i];
            l++;
        }
        else {
            rechts[r] = unsortiert[i];
            r++;
        }
    }
    // links + rechts kürzen
    links = quicksort(links);
    rechts = quicksort(rechts);
                                          // sortiert = links + pivot + rechts
                                      }
```

QuickSort

Das QuickSort-Verfahren basiert darauf, zuerst ein Pivot-Element auszuwählen, anschließend die Liste anhand diesem aufzuteilen und die Teillisten wieder per QuickSort rekursiv zu sortieren. Anschließend werden die Teillisten wieder zusammengesetzt.

```
public static int[] quicksort(int[] unsortiert) {
    // Abbruchbedingung wenn unsortiert.length<=1
    // Pivot-Element auswählen (erstes Element des Arrays)
    // linke und rechte Teilliste anlegen
    // Liste anhand des Pivot-Elements aufteilen
    // linke und rechte Teilliste "kürzen"
    // linke und rechte Teilliste rekursiv sortieren
    // Gesamtliste sortiert zusammenfügen
}
```

Listing 1: Überblick QuickSort

Liste „kürzen“ bzw. zusammenfügen

Ein Array in JAVA hat immer eine konstante Länge. Um ein kürzeres Array zu erhalten, müssen wir dieses neu anlegen und die Elemente kopieren. Hierzu gibt es die Methode

```
System.arraycopy(Quelle, Start-Index, Ziele, Start-Index, Länge);
```

Listing 2: Inhalt eines Arrays kopieren

1. Programmierung

Fülle zunächst oben stehenden Code aus. Programmiere anschließend die Methode.

2. Test

Lasse anschließend ein Array mit 10 (100, 1000,...) Einträgen zufällig befüllen und ausgeben. Rufe dann deine Methode auf und lasse die sortierte Liste wieder ausgeben.

3. Laufzeit

Um die Laufzeit des Verfahrens zu messen, können wir die Methode `System.currentTimeMillis();` verwenden. Diese gibt die vergangenen Millisekunden seit 1. Januar 1970 an. (Datentyp `long`)
Lassen wir diese vor und nach der Sortierung ausgeben können wir daraus die benötigte Zeit berechnen. Erzeuge (nacheinander) verschieden große Arrays und messe damit die Laufzeiten. Vergleiche die verschieden großen Arrays.