



Aryeh Kontorovich · 23h
how many logicians showed up to this wedding?

252-0027

Einführung in die Programmierung Übungen

Woche 5: Arrays, Methoden, Debugger

Timo Baumberger
Departement Informatik
ETH Zürich



7

6

89

3.5K

Organisatorisches

- Mein Name: Timo Baumberger
- Bei Fragen: tbaumberger@student.ethz.ch
(Discord: troxhi)
 - Mails bitte mit «[EProg25]» im Betreff
- Meine Website: timobaumberger.com
- Neue Aufgaben: **Dienstag Abend** (im Normalfall)
- Abgabe der Übungen bis **Dienstag Abend (23:59)** Folgewoche
 - Abgabe immer via Git
 - Lösungen in separatem Projekt auf Git



Programm

- **Arrays**
- **Strings**
- **Methoden**
- **Java Debugger**
- **Nachbesprechung**
- **Vorbesprechung**
- **Kahoot**

1D Arrays

Eindimensionale Arrays

- Arrays belegen eine feste Grösse n im Speicher, haben daher auch eine feste Länge
- für jeden Eintrag kann für den deklarierten Datentyp ein Wert gespeichert werden
- auf ein spezifisches Element i zugreifen können wir mit **arr[i]**
- Indizierung startet bei **0**, geht also bis **n-1** (wie bei Strings)

```
● ● ●  
1 public class Main {  
2     public static void main(String[] args){  
3         int[] arr = {3,5,6,1,2,8};  
4         for(int i = 0; i < arr.length; i++){  
5             arr[i] *= 2;  
6         }  
7     }  
8 }
```

Beispiel 1D Array

2D Arrays

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

[[1,2,3],[4,5,6],[7,8,9]]

Zweidimensionale Arrays

- eine Matrix könnte als Zweidimensionales Array dargestellt werden
- `int[][] arr = new int[n][n];`
definiert ein Array, welches **n** integer-Arrays der Länge **n** speichert (**nxn-Matrix**)
- eine gesamte Zeile erhalten wir also mit `arr[i]` für **0≤i< n**
- einen Eintrag erhalten wir mit `arr[i][j]` für **0≤i,j< n**
- **Vorsicht:** Die Längen der Arrays könnten unterschiedlich sein!



```
1 public class Main {  
2     public static void main(String[] args) {  
3         int[][] arr = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};  
4         for (int i = 0; i < arr.length; i++) {  
5             for (int j = 0; j < arr[i].length; j++) {  
6                 arr[i][j] *= 2;  
7             }  
8         }  
9     }  
10 }
```

2D Arrays

array = {arrOne, arrTwo, arrThree}

i	array[i]	i	0	1	2	3	4	5
0		arrOne[i]	4	501	-1	200	42	0
1		arrTwo[i]	11	21	-170			
2		arrThree[i]	100	1	-321	3		

```
int[] arOneD = {1,2,3,4,5,6};
```

A diagram illustrating a one-dimensional array. At the top, the declaration `int[] arOneD = {1,2,3,4,5,6};` is shown. A curved arrow points from the variable name `arOneD` down to a table below. The table has two rows: a header row with columns labeled `i`, `0`, `1`, `2`, `3`, `4`, and `5`; and a data row with columns labeled `arOneD[i]`, `1`, `2`, `3`, `4`, `5`, and `6`. The entire table is enclosed in a red border.

<code>i</code>	<code>0</code>	<code>1</code>	<code>2</code>	<code>3</code>	<code>4</code>	<code>5</code>
<code>arOneD[i]</code>	<code>1</code>	<code>2</code>	<code>3</code>	<code>4</code>	<code>5</code>	<code>6</code>

```
int[][] arTwoD = {  
    {1,2,3},  
    {4,5,6},  
    {7,8,9}  
};
```

A diagram illustrating a two-dimensional array. On the left, the declaration `int[][] arTwoD = {{1,2,3}, {4,5,6}, {7,8,9}};` is shown. A curved arrow points from the variable name `arTwoD` down to a table below. The table has two rows: a header row with columns labeled `i`, `0`, `1`, and `2`; and a data row with columns labeled `arTwoD[i]`, which is empty. The entire table is enclosed in a red border.

<code>i</code>	<code>0</code>	<code>1</code>	<code>2</code>
<code>arTwoD[i]</code>			

A diagram showing the first row of the two-dimensional array. It consists of two tables. The top table has columns labeled `i`, `0`, `1`, and `2`. The bottom table has columns labeled `[i]`, `1`, `2`, and `3`. Arrows point from the `i` column of the top table to the `i` and `[i]` columns, and from the `1` through `3` columns of the top table to the `1` through `3` columns of the bottom table.

<code>i</code>	<code>0</code>	<code>1</code>	<code>2</code>
<code>[i]</code>	<code>1</code>	<code>2</code>	<code>3</code>

A diagram showing the second row of the two-dimensional array. It consists of two tables. The top table has columns labeled `i`, `0`, `1`, and `2`. The bottom table has columns labeled `[i]`, `4`, `5`, and `6`. Arrows point from the `i` column of the top table to the `i` and `[i]` columns, and from the `4` through `6` columns of the top table to the `4` through `6` columns of the bottom table.

<code>i</code>	<code>0</code>	<code>1</code>	<code>2</code>
<code>[i]</code>	<code>4</code>	<code>5</code>	<code>6</code>

A diagram showing the third row of the two-dimensional array. It consists of two tables. The top table has columns labeled `i`, `0`, `1`, and `2`. The bottom table has columns labeled `[i]`, `7`, `8`, and `9`. Arrows point from the `i` column of the top table to the `i` and `[i]` columns, and from the `7` through `9` columns of the top table to the `7` through `9` columns of the bottom table.

<code>i</code>	<code>0</code>	<code>1</code>	<code>2</code>
<code>[i]</code>	<code>7</code>	<code>8</code>	<code>9</code>

```
int[][] arTwoD = {  
    {1,2,3},  
    {4,5,6},  
    {7,8,9}  
};
```

arTwo[i] =

i	
0	{1,2,3}
1	{4,5,6}
2	{7,8,9}

arTwo[i][j] =

i\j	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9

Strings: Wiederholung

- Strings sind Referenzen (gespeichert im Heap)
- Strings sind aber immutable / unveränderbar
- Wirklich? Ja! 😊 `private final byte[] value;`
- Arrays sind veränderbar
- Bei 0 indexiert
- $a_0, \dots, a_{length()-1}$ $a.substring(i, j) = a_i, \dots, a_{j-1}$

Strings Internal (nice to know)

- Literal String sind immer identisch
- Strings werden in String Constant Pool gespeichert
- Konstante String Expressions werden im Constant Pool gespeichert (Auswertung vor Ausführung des Codes möglich)
- Sequenz von Chars (eigentlich von Bytes)

```
String hello = "Hello";
String lo = "lo";
System.out.println(hello == ("Hel" + lo)); // keine konstante String expression
System.out.println(hello == ("Hel" + "lo")); // konstante String expression
System.out.println(hello == new String(hello)); // erstellt neuen String
System.out.println(hello == new String(hello).intern()); // verwendet String aus Constant Pool
```

Strings Internal (extremely nice to know)

```
9      // Use Latin1 / compact strings (instead of UTF-16)
10     String string1 = "sdf";
11     String string2 = "sdf";
12     System.out.println("===== Old values =====");
13     System.out.println("Value of string1: " + string1);
14     System.out.println("Value of string2: " + string2);
15
16     mutateString(string1);
17
18     System.out.println("===== New values =====");
19     System.out.println("Value of string1: " + string1);
20     System.out.println("Value of string2: " + string2);
21 }
22
23 public static void mutateString(String immutable) throws Exception {
24     Field f = String.class.getDeclaredField("value");
25     f.setAccessible(true);
26
27     byte[] value = (byte[]) f.get(immutable);
28     value[0] = 'a';
29 }
```

Output

```
===== Old values =====
Value of string1: sdf
Value of string2: sdf
===== New values =====
Value of string1: adf
Value of string2: adf
```

String Performance (Lösung für Immutability)

```
StringBuilder builder = new StringBuilder("a");
for (int i = 0; i < 100_000; i++) {
    builder.append("a");
}
```

Elapsed time: 3

```
String s = "a";
for (int i = 0; i < 100_000; i++) {
    s = s + "a";
}
```

Elapsed time: 284 Lösung

String Beispiel

```
public class MyClass {  
  
    public static void main(String[] args) {  
        String str = "Kegelvolumen";  
        boolean result = str.startsWith("lvolumen", 4);  
        System.out.println(result);  
    }  
}
```

Output: true

String Beispiel

```
public class MyClass {  
  
    public static void main(String[] args) {  
        String str = "Kegelvolumen";  
        boolean result = str.startsWith("men", str.length() - 2);  
        System.out.println(result);  
    }  
}
```

Output: false

String Beispiele



```
1 public class MyClass{
2     public static void main(String[] args){
3         String str = "Einfuehrung";
4         String result = str.substring(2, 6).replace('f', 'X').concat(str.substring(6, 9));
5     }
6 }
```

Output: nXuehru

String Beispiele

```
● ● ●  
1 public class MyClass{  
2     public static void main(String[] args){  
3         String str = "Programmierung";  
4         String result = str.charAt(0) + str.substring(2, 5).toLowerCase() + str.charAt(str.length() - 1);  
5     }  
6 }
```

Output: Pogr̄g

Methoden

```
public static int add(int a, int b){ ...
```

```
public static int add(int a, int b){ ...
```



The diagram illustrates the structure of a Java method signature. It consists of several colored brackets pointing to specific parts of the code: a red bracket under "public" labeled "Sichtbarkeit" (Visibility); a green bracket under "int" labeled "Rückgabetyp" (Return Type); two yellow brackets under "add" and its parameters "a" and "b" labeled "Name" (Name); and a yellow bracket under the parameter list labeled "Parameter" (Parameters).

The diagram illustrates the components of a Java method signature. At the top, the word "Signatur" is written above a bracket that spans from the start of the method name "add" to the closing brace "...". Below this, the code "public static int add(int a, int b){ ...}" is shown. A red bracket under "public" is labeled "Sichtbarkeit" (Visibility). A green bracket under "int" is labeled "Rückgabetyp" (Return Type). Three yellow brackets under "add" are labeled "Name" (Name), "Parameter" (Parameters), and "Parameter" (Parameters), indicating that the second and third brackets both apply to the parameter list. The code uses color coding: red for visibility, green for return type, black for the method name, and yellow for parameters.

```
public static int add(int a, int b){ ...}
```

Java: Methoden Signatur

- Kombination aus Name der Methode, Anzahl der Parameter, Typen und Reihenfolge der Parameter
- Reihefolge wird nur beachtet wenn die Parameter Typen unterschiedlich sind
- Signatur einer Methode identifiziert eine Methode in Java eindeutig

Method-Overloading

- Eine Methode kann überladen werden, in dem die Methodenparameter verändert werden
- trotz gleichem Namen hat sie dann eine andere Signatur
- **add** kann durch Überladung mit unterschiedlichen Parametern aufgerufen werden, hat nun unterschiedliche Rückgabetypen

```
● ● ●  
1 public class Main {  
2     // Erste Version: Addiert zwei int-Werte  
3     public static int add(int a, int b) {  
4         return a + b;  
5     }  
6     // Überladene Version: Addiert drei int-Werte  
7     public static int add(int a, int b, int c) {  
8         return a + b + c;  
9     }  
10    // Überladene Version: Addiert zwei double-Werte  
11    public static double add(double a, double b) {  
12        return a + b;  
13    }  
14 }
```

Value vs Reference

Objekte 101 – (Deep Dive in einer späteren Übung)

- Klassen sind „Baupläne“, die definieren, wie Daten gespeichert werden
- Objekte sind konkrete Realisierungen der Klasse
- Die Variable des Objektes speichert einen Verweis, wo die Daten im Speicher liegen (Referenz)



```
1 public class Main {  
2     public static void main(String[] args){  
3         Coordinate coord = new Coordinate(2,3);  
4         System.out.println(coord);  
5     }  
6 }  
7 public class Coordinate {  
8     int x;  
9     int y;  
10    public Coordinate(int xcoord, int ycoord){  
11        x = xcoord;  
12        y = ycoord;  
13    }  
14 }
```

Primitives

- speichern tatsächlichen Wert
- sind von der Grösse begrenzt
- Standardwerte sind festgelegt
- Können niemals **NULL** sein

Objekte

- speichern Referenz auf Speicherort
- Grösse ist variabel
- Standardwert ist **NULL** (keine Referenz in den Speicher)
- Strings sind auch Objekte

Weitergabe von Referenzen

- **numbers** speichert die Referenz, wo die tatsächlichen Werte liegen
- gib Referenz an **modifyArray** weiter
- **modifyArray** sucht Werte im Speicher, verdoppelt sie, terminiert
- **main** sucht mit derselben im Speicher und findet verdoppelte Werte **ohne**, dass wir sie zurückgeben mussten



```
1 public class ReferenceArrayDemo {  
2     public static void main(String[] args) {  
3         // Erstelle ein Array  
4         int[] numbers = {1, 2, 3, 4, 5};  
5         System.out.println("Vor der Änderung:");  
6         System.out.println(Arrays.toString(numbers));  
7         modifyArray(numbers); // Übergabe des Arrays  
8         System.out.println("Nach der Änderung:");  
9         System.out.println(Arrays.toString(numbers));  
10    }  
11    public static void modifyArray(int[] arr) {  
12        for (int i = 0; i < arr.length; i++) {  
13            arr[i] *= 2; // Verdopple jeden Wert im Array  
14        }  
15    }  
}
```

Rekursion

Rekursion - Beispiel

1. Implementieren Sie die Methode `Calculations.checksum(int x)`, das heisst die Methode `checksum` in der Klasse `Calculations`. Die Methode nimmt einen Integer `x` als Argument, welcher einen nicht-negativen Wert hat. Die Methode soll die Quersumme von `x` zurückgeben. Sie sollen für diese Aufgabe **keine** Schleife verwenden.

Beispiele

- `checksum(258)` gibt 15 zurück.
- `checksum(49)` gibt 13 zurück.
- `checksum(12)` gibt 3 zurück.

Hinweis: Für einen Integer `a` ist `a % 10` die letzte Ziffer und `a / 10` entfernt die letzte Ziffer. Zum Beispiel `258 % 10` ist 8 und `258 / 10` ist 25.

Rekursion - Potenzieren

```
public static int pow(int n, int k) {  
    if (k == 0) {  
        return 1;  
    }  
    int r = pow(n, k/2);  
    if (k % 2 == 0) {  
        return r*r;  
    } else {  
        return r*r*n;  
    }  
}
```

Java Debugger

Was ist der Debugger und was tut er?

- Ein Tool (in Java), das beim Debuggen hilft.
- Mit dem Java-Debugger kann man Schritt für Schritt durch das Programm gehen und genau beobachten, wie es ausgeführt wird.
- Die Änderungen der Variablen und ihrer Werte in jeder Zeile werden in einer Tabelle dargestellt.

The screenshot shows the IntelliJ IDEA interface. The top navigation bar includes tabs for 'Current File' and various icons. On the left, the 'Project' tool window displays a file structure with 'DebuggerExample' as the root, containing 'idea', 'out', 'src' (with 'DebuggerPart1' selected), '.gitignore', 'DebuggerExample.iml', 'External Libraries', and 'Scratches and Consoles'. The main code editor window shows 'DebuggerPart1.java' with the following code:

```
public class DebuggerPart1 {
    public static void main(String[] args){
        int zahl1 = 10;
        int zahl2 = 20;
        int zahl3 = 30;
        int summe = zahl1+zahl2+zahl3;

        int anzahl = 3;
        int average = findAverage(summe, anzahl);
        System.out.println("average:"+average);
    }

    public static int findAverage(int summe, int anzahl){ 1usage
        int result = summe * anzahl; //Es sollte summe/anzahl sein
        return result;
    }
}
```

The bottom 'Run' tool window shows the output of the run command:

```
/Library/Java/JavaVirtualMachines/jdk-21.jdk/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA CE.app/Contents/lib/idea_rt.jar=53293 -Dfile.encoding=UTF-8
average:180
Process finished with exit code 0
```

Das Programm macht nicht, was du erwartest, und du kannst den Fehler nicht finden?

Benutze den Debugger!

Breakpoints

- Bis zum Breakpoint wird alles automatisch ausgeführt, und sobald die Zeile mit dem Breakpoint erreicht wird, stoppt die automatische Ausführung.
- Anschliessend kann der Benutzer Zeile für Zeile das Programm selbst ausführen.
- **Breakpoint auswählen:** An einer Stelle, an der man weiss, dass alles bis dahin wie erwartet funktioniert.
- **Breakpoint setzen:** Links von der Spalte mit den Zeilenummern klicken, um den Breakpoint zu setzen



The screenshot shows the IntelliJ IDEA interface. In the top navigation bar, there are icons for file operations and tabs for 'Current File' and 'Version control'. Below the navigation bar is the project structure sidebar, which includes sections for 'Project', 'External Libraries', and 'Scratches and Consoles'. The main area displays the code for 'DebuggerPart1.java'. The code contains a main method that calculates the sum of three integers (zahl1, zahl2, zahl3) and then calls a static method 'findAverage' to calculate the average. The 'findAverage' method takes the sum and the number of elements (anzahl) as parameters and returns the result. A red dot at the start of line 10 indicates a breakpoint has been set. The run console at the bottom shows the command used to run the application and the output 'average:180'. The status bar at the bottom right shows the file path 'DebuggerExample > src > DebuggerPart1', line count '23:1', encoding 'LF', and character count '4 spaces'.

```
public class DebuggerPart1 {  
    public static void main(String[] args){  
        int zahl1 = 10;  
        int zahl2 = 20;  
        int zahl3 = 30;  
        int summe = zahl1+zahl2+zahl3;  
  
        int anzahl = 3;  
        int average = findAverage(summe, anzahl);  
        System.out.println("average:"+average);  
    }  
  
    public static int findAverage(int summe, int anzahl){ 1 usage  
        int result = summe * anzahl; //Es sollte summe/anzahl sein  
        return result;  
    }  
}
```

- Da wir sicher sind, dass **summe** und **anzahl** stimmen, wollen wir nun die Methode **findAverage** testen.
- Dazu setzen wir einen Breakpoint in der Zeile, in der diese Methode aufgerufen wird.

The screenshot shows the IntelliJ IDEA interface with a Java project named "DebuggerExample". The code editor displays "DebuggerPart1.java" containing the following code:

```
public class DebuggerPart1 {  
    public static void main(String[] args){  
        int zahl1 = 10;  
        int zahl2 = 20;  
        int zahl3 = 30;  
        int summe = zahl1+zahl2+zahl3;  
  
        int anzahl = 3;  
        int average = findAverage(summe, anzahl);  
        System.out.println("average:"+average);  
    }  
  
    public static int findAverage(int summe, int anzahl){ 1 usage  
        int result = summe * anzahl; //Es sollte summe/anzahl sein  
        return result;  
    }  
}
```

An orange arrow points to the line `int average = findAverage(summe, anzahl);`. The run tool window at the bottom shows the output of the program: `/Library/Java/JavaVirtualMachines/jdk-21.jdk/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA CE.app/Contents/lib/idea_rt.jar=53445 -Dfile.encoding=UTF-8` and `average:180`.

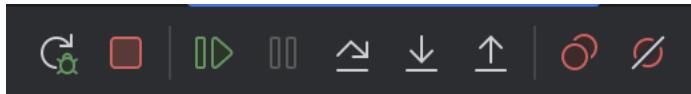
The screenshot shows a Java development environment with the following details:

Project View: The project is named "DebuggerExample". It contains a ".idea" folder, an "out" folder, a "src" folder which includes a package named "DebuggerPart1" containing a file "DebuggerPart1.java", a ".gitignore" file, and an "External Libraries" section. There is also a "Scratches and Consoles" section.

Code Editor: The file "DebuggerPart1.java" is open. The code defines a class "DebuggerPart1" with a main method that calculates the average of three integers (zahil1, zahil2, zahil3) and prints it. A breakpoint is set on line 10, where the variable "average" is assigned. The code editor shows variable values in the status bar: zahil1: 10, zahil2: 20, zahil3: 30, summe: 60, zahl2: 20, zahl3: 30, zahl1: 10, and anzahl: 3.

Debug View: The "Debug" view is active, showing a stack trace for the "main" method of "DebuggerPart1". The current frame is "main:11, DebuggerPart1". The "Threads & Variables" tab is selected. The variables pane shows the local variables: args (String[0]@823), zahil1 (10), zahil2 (20), zahil3 (30), summe (60), and anzahl (3). The status bar at the bottom indicates the current file is "DebuggerExample > src > DebuggerPart1".

- **Blaue Zeile:** alles **bis und ohne** diese Zeile wurde ausgeführt
- **Debugger Symbole:**



Termination
- Um den Debugger zu stoppen

Step over
- Um zur nächsten Zeile zu gehen.

Step into
- Um in eine Methode zu springen.

View breakpoints
- Alle breakpoints anzeigen

Step out
- Um aus einer Methode raus zu treten

Von dieser Zeile aus weitergehen...

The screenshot shows the IntelliJ IDEA IDE interface. The top bar includes the title 'DebuggerExample', version control dropdown, current file dropdown, and various tool icons. The left sidebar displays the project structure with 'DebuggerExample' selected, showing '.idea', 'out', and 'src' directories, including 'DebuggerPart1.java'. The main editor window shows the Java code for 'DebuggerPart1'. The code defines a 'main' method that calculates the average of three integers (zahll1, zahll2, zahll3) and prints it. It also defines a 'findAverage' static method. The line 'int average = findAverage(summe, anzahl);' is highlighted with a blue selection bar. The bottom panel features the debugger interface with tabs for 'Threads & Variables' and 'Console'. The 'Threads & Variables' tab is active, showing variable values: args = [String@823], zahl1 = 10, zahl2 = 20, zahl3 = 30, summe = 60, anzahl = 3. The status bar at the bottom shows file paths 'DebuggerExample > src > DebuggerPart1', encoding 'UTF-8', and a character count of '4 spaces'.

```
public class DebuggerPart1 {  
    public static void main(String[] args){    args: []  
        int zahl1 = 10;    zahl1: 10  
        int zahl2 = 20;    zahl2: 20  
        int zahl3 = 30;    zahl3: 30  
        int summe = zahl1+zahl2+zahl3;    summe: 60    zahl2: 20    zahl3: 30    zahl1: 10  
        int anzahl = 3;    anzahl: 3  
        int average = findAverage(summe, anzahl);    summe: 60    anzahl: 3  
        System.out.println("average:"+average);  
    }  
  
    public static int findAverage(int summe, int anzahl){    1 usage  
        int result = summe * anzahl; //Es sollte summe/anzahl sein  
        return result;  
    }  
}
```

Threads & Variables

Evaluate expression (E) or add a watch (W)

args = [String@823]
zahl1 = 10
zahl2 = 20
zahl3 = 30
summe = 60
anzahl = 3

44

Mit step into:

The screenshot shows the IntelliJ IDEA IDE interface with the following details:

- Project View:** Shows the project structure with a file named "DebuggerPart1.java" selected.
- Code Editor:** Displays the Java code for "DebuggerPart1". A line of code is highlighted in blue: `int result = summe * anzahl; //Es sollte summe/anzahl sein`. This indicates the current line of execution.
- Debug Bar:** At the bottom, the "Debug" tab is active, showing the current thread: "main" @ 1...n: RUNNING.
- Variables View:** Shows local variables: `summe = 60` and `anzahl = 3`.
- Status Bar:** Shows the file path: "DebuggerExample > src > DebuggerPart1".

Wir sind in die aufgerufene Methode **findAverage** hineingegangen und können nun diese Methode Schritt für Schritt ausführen.

Mit step over:

The screenshot shows the IntelliJ IDEA IDE with the Java code for 'DebuggerPart1.java' open. The code defines a main method that calculates the average of three integers (zahl1, zahl2, zahl3) and prints it. A breakpoint is set at the line 'int average = findAverage(summe, anzahl);'. The 'Threads & Variables' tool window shows local variables: args, zahl1, zahl2, zahl3, summe, anzahl, and average. The value of average is shown as 180. The 'Evaluate expression' field contains the expression '@args = (String[]@823) []'. The 'Step Over' button in the toolbar is highlighted.

```
public class DebuggerPart1 {
    public static void main(String[] args){    args: []
        int zahl1 = 10;    zahl1: 10
        int zahl2 = 20;    zahl2: 20
        int zahl3 = 30;    zahl3: 30
        int summe = zahl1+zahl2+zahl3;    summe: 60    zahl1: 10    zahl2: 20    zahl3: 30
        int anzahl = 3;    anzahl: 3
        int average = findAverage(summe, anzahl);    average: 180    anzahl: 3    summe: 60
        System.out.println("average:"+average);    average: 180
    }

    public static int findAverage(int summe, int anzahl){ 1 usage
        int result = summe * anzahl; //Es sollte summe/anzahl sein
        return result;
    }
}
```

Threads & Variables

Variable	Type	Value
args	(String[]@823)	[]
zahl1	int	10
zahl2	int	20
zahl3	int	30
summe	int	60
anzahl	int	3
average	int	180

Evaluate expression (♂) or add a watch (⌚)

Switch frames from anywhere in the IDE with ⌘ F4

DebuggerExample > src > DebuggerPart1

- Der Methodenaufruf von **findAverage** wurde in einem Schritt ausgeführt und der Rückgabewert an **average** zugewiesen.
- Wir befinden uns jetzt in der nächsten Zeile.

Step over vs. Step into: Wann welches benutzen?

- **Step over:** 
 - Zur nächsten Zeile springen.
 - Wenn in der Zeile eine Methode aufgerufen wird, wird sie ausgeführt. Man geht davon aus, dass sie wie erwartet funktioniert und dass der Rückgabewert korrekt ist.
- **Step into:** 
 - In die Methode hineingehen, um sie schrittweise zu durchlaufen.
 - Dies verwendet man, wenn man unsicher ist, ob die Methode wie erwartet funktioniert, und man den Ablauf genauer überprüfen möchte.

The screenshot shows the Eclipse IDE's Debug perspective. The title bar says "Debug" and "DebuggerPart1". The toolbar includes icons for start, stop, step over, step into, and step out. The "Threads & Variables" tab is selected. A status bar at the bottom shows the path: "DebuggerExample > src > DebuggerPart1".

The main area displays the current variable values:

- args = {String[0]@823} []
- zahl1 = 10
- zahl2 = 20
- zahl3 = 30
- summe = 60
- anzahl = 3

A tooltip at the bottom left says "Switch frames from anywhere in the IDE wi...".

- Zeigt alle Variablen und ihre Werte an, die im Scope der blauen Zeile liegen.
- Nach der Ausführung jeder Zeile kann man überprüfen, ob sich die Werte wie erwartet geändert haben.

```
1 public class DebuggerPart1 {  
2  
3     public static void main(String[] args){  args: []  
4  
5         int zahl1 = 10;  zahl1: 10  
6         int zahl2 = 20;  zahl2: 20  
7         int zahl3 = 30;  zahl3: 30  
8         int summe = zahl1+zahl2+zahl3;  summe: 60    zahl2: 20    zahl3: 30    zahl1: 10  
9  
10        int anzahl = 3;  anzahl: 3  
11        int average = findAverage(summe, anzahl);  anzahl: 3    summe: 60  
12        System.out.println("average:"+average);  
13    }  
14  
15    public static int findAverage(int summe, int anzahl){  1 usage  
16  
17        int result = summe * anzahl; //Es sollte summe/anzahl sein  
18        return result;  
19    }  
20  
21 }  
22 }
```

- Die Werte der Variablen werden auch am Ende jeder Zeile in Grau angezeigt

```
public class DebuggerPart1 {

    public static void main(String[] args){

        int zahl1 = 10;
        int zahl2 = 20;
        int zahl3 = 30;
        int summe = zahl1+zahl2+zahl3;

        int anzahl = 3;
        int average = findAverage(summe, anzahl);
        System.out.println("average:"+average);

    }

    public static int findAverage(int summe, int anzahl){ 1 usage           summe: 60      anzahl: 3

        int result = summe * anzahl; //Es sollte summe/anzahl sein   result: 180      summe: 60      anzahl: 3
        return result;  result: 180
    }

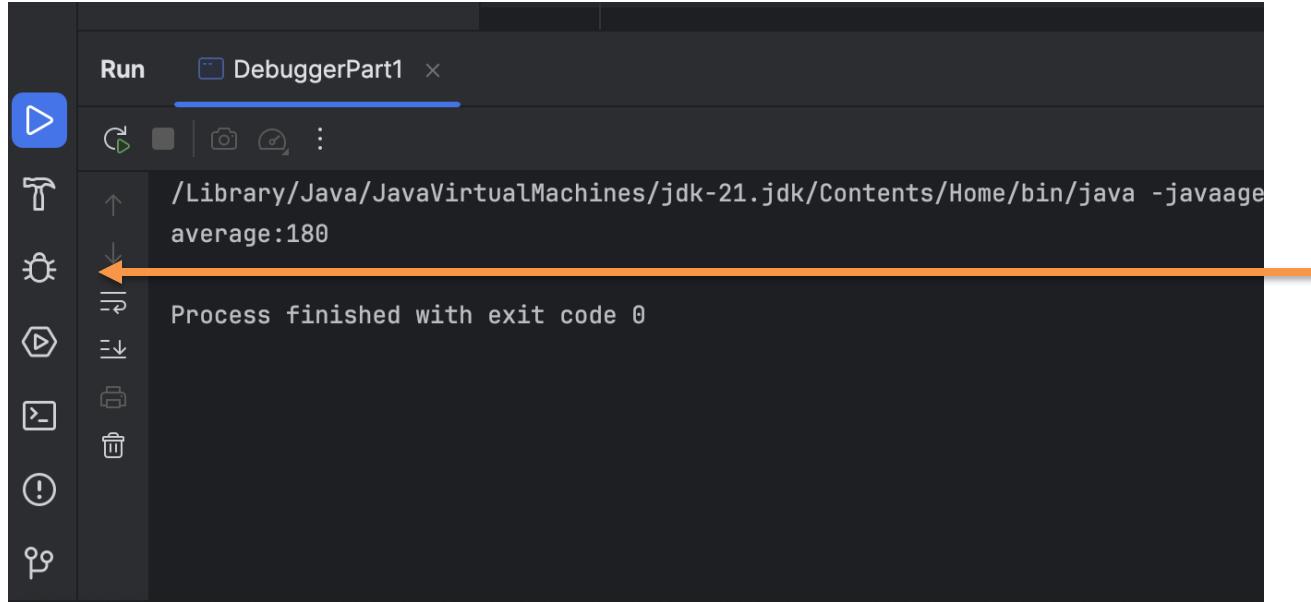
}

Threads & Variables  Console
Evaluate expression (⌚) or add a watch (⌚⌚⌚)
⌚ result = 180
⌚ summe = 60
⌚ anzahl = 3
⌚ result = 180
```

Hier kann man sehen,
welche Werte **summe**,
anzahl und **average**
enthalten.

Unter „Threads & Variables“
erkennt man, dass **average**
durch **(summe * anzahl)**
berechnet wird, anstatt
durch **(summe / anzahl)**.

Falls nicht automatisch Threads & Variables gezeigt wird:



The screenshot shows the IntelliJ IDEA Run tool window. The title bar says "Run" and "DebuggerPart1". Below the title bar are several icons: a play button (highlighted with an orange arrow), a green arrow, a camera, and a settings gear. The main area displays the command used to run the application: "/Library/Java/JavaVirtualMachines/jdk-21.jdk/Contents/Home/bin/java -javaagent:average:180". Below this, the message "Process finished with exit code 0" is shown. To the left of the main area is a vertical toolbar with various icons: a play button, a green arrow, a camera, a settings gear, a double minus sign, a double plus sign, a printer, a trash can, and an exclamation mark.

Debug auswählen

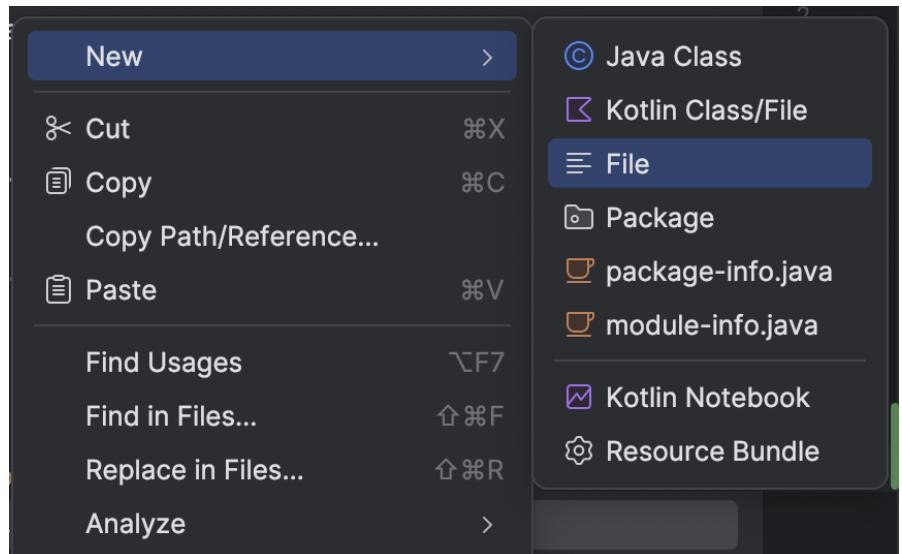
Feedback

Feedback

- Ihr sagt den TAs ab u05 wo ihr Feedback haben möchten.
- Dazu erstellt ihr eine requestfeedback.txt Datei (in den uXX Ordner – nicht src / test / resources).
- In die Datei schreibt ihr die Aufgaben, für welche ihr Feedback haben wollt. (**Bonus wird separat gegraded, für die Bonusaufgabe kriegt ihr immer ein Feedback**)

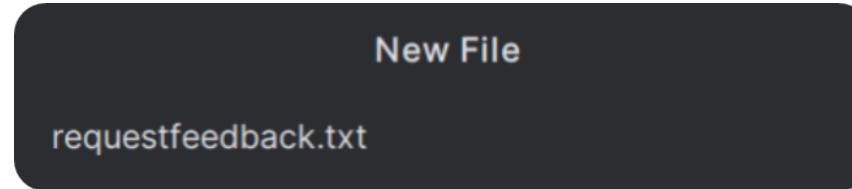
Feedback Datei erstellen

1. Projekt Ordner uXX rechts-klicken.
2. New -> File.



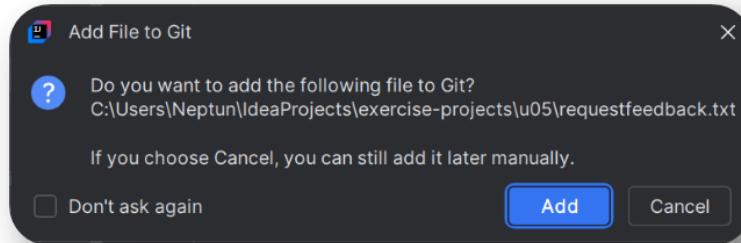
Feedback Datei erstellen

- 1. Projekt Ordner uXX rechts-klicken.**
- 2. New -> File.**
- 3. Benenne die Datei als requestfeedback.txt im erscheinenden Fenster.**



Feedback Datei erstellen

- 1. Projekt Ordner uXX rechts-klicken.**
- 2. New -> File.**
- 3. Benenne die Datei als requestfeedback.txt im erscheinenden Fenster.**
- 4. Wähle Add im auftauchenden Fenster «Add File to Git»**



Nachbesprechung

Aufgabe 1: Binärdarstellung

```
/*
 * Bestimmt exp so, dass  $2^{\text{exp}} \leq \text{number}$  und  $2^{(\text{exp} + 1)} > \text{number}$  gilt.
 */
public static int largestExponent(int number) { 1 usage & zelleraa

    int largestPowerOfTwo = 1; //  $2^0 = 1$ 
    int exp = 0;

    if(number <= 0) {
        System.out.println("Keine positive ganze Zahl!");
    } else {
        while(largestPowerOfTwo <= number) { //Erhöhe exp
            exp = exp + 1;
            largestPowerOfTwo = largestPowerOfTwo * 2; // $2^{\text{exp}}$ 
        }
        // largestPowerOfTwo > number gilt hier aber es gilt  $2^{\text{exp}} \leq \text{number}$ 
        // deshalb gehen wir einen Schritt zurück
        largestPowerOfTwo = largestPowerOfTwo / 2;
        exp = exp - 1;
    }

    return exp;
}
```

Schritt 1: Finde exp so, dass $2^{\text{exp}} \leq \text{number}$ und $2^{(\text{exp} + 1)} > \text{number}$.

Aufgabe 1: Binärdarstellung

```
/*
 * Gibt die Binaerdarstellung von number aus gegeben dem grössten exp,
 * wo  $2^{\text{exp}} \leq \text{number}$  und  $2^{(\text{exp} + 1)} > \text{number}$  gilt und
 */
public static String binaerDarstellung(int number, int exp, int largestPowerOfTwo) { 1 usage  2 zelleraa
    // Dummy variables to make code more readable
    int currentPowerOfTwo = largestPowerOfTwo;
    int remainingNumber = number;
    int currentExp = exp;

    String bDarstellung = "";

    while(currentExp >= 0) {
        // Prüfe ob verbleibende Zahl grösser (1) oder kleiner (0) als die currentPowerOfTwo ist
        int digit = remainingNumber / currentPowerOfTwo;

        // Füge digit an binärdarstellung an - nutzt int cast zu string bei + mit string
        bDarstellung = bDarstellung + digit;

        // Gehe zur nächstkleineren Zweierpotenz
        remainingNumber = remainingNumber - digit * currentPowerOfTwo;
        currentExp = currentExp - 1;
        currentPowerOfTwo = currentPowerOfTwo / 2;
    }

    return bDarstellung;
}
```

Schritt 2: Wenn $2^{\text{currentExp}} \geq \text{number}$ dann ist das nächste Bit 1 sonst 0.

Aufgabe 2: Grösster gemeinsamer Teiler

Schreiben Sie ein Programm "GGT.java", das den grössten gemeinsamen Teiler (ggT) zweier ganzer Zahlen mithilfe des Euklidischen Algorithmus berechnet. Hierbei handelt es sich um eine iterative Berechnung, die auf folgender Beobachtung basiert:

1. Wenn x grösser als y ist, dann ist — sofern sich x durch y teilen lässt — der ggT von x und y gleich y ;
2. andernfalls ist der ggT von x und y der gleiche wie der ggT von y und $x \% y$.

```
while (x <= y || x % y != 0) { //solange x kleiner als y oder x sich nicht durch y teilen lässt:  
    //GGT noch nicht gefunden  
    //update x und y für nächste Iteration  
    int altY = y; // Zwischenspeichern von y  
  
    y = x % y;  
    x = altY;  
  
}  
  
//GGT gefunden (x >= y && x % y == 0)  
System.out.println(y);
```

Negation der Bedingung in 1.

Aufgabe 3: Zahlerkennung

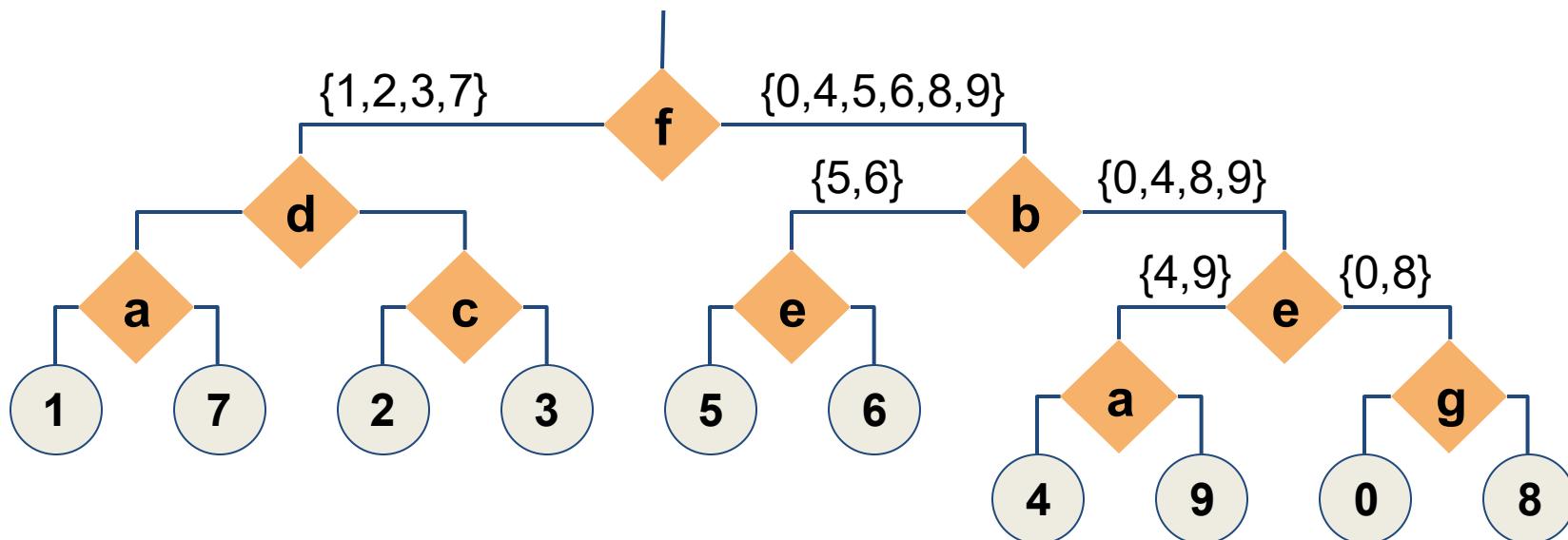
Für diese Aufgabe verwenden wir einen String um die erleuchtenden Segmente einer [Siebensegmentanzeige](#) zu kodieren. Die Segmente sind, wie im Bild gezeigt, von a bis g nummeriert. Die Kodierung einer möglichen Anzeige ist ein String, in welchem der Buchstabe 'x' genau dann vorkommt, wenn das 'x'te Segment der Anzeige erleuchtet ist. Zum Beispiel wird die Zahl 2 kodiert durch 'abged'. Zur Einfachheit darf angenommen werden, dass kein Buchstabe mehr als einmal in der Kodierung vorkommt und dass nur die Zahlen 0 bis 9 kodiert werden.

Schreiben Sie ein Programm "Zahlen.java", das einen String, der eine Anzeige kodiert, einliest und die kodierte Zahl als Integer ausgibt. Überlegen Sie wie viele IF Blöcke benötigt werden um jede Zahl zu erkennen.

Tipp: Sie können `str.contains("a")` verwenden, um zu überprüfen, ob ein String `str` den Buchstaben 'a' enthält.



Aufgabe 3: Zahlenerkenn



Aufgabe 4: Berechnungen

2. Implementieren sie die Methode Calculations.magic7(int a, int b). Die Methode gibt einen Boolean zurück. Die Methode soll true zurückgeben, wenn einer der Parameter 7 ist oder wenn die Summe oder Differenz der Parameter 7 ist. Ansonsten soll die Methode false zurückgeben.

Beispiele

- `magic7(2,5)` gibt true zurück.
- `magic7(7,9)` gibt true zurück.
- `magic7(5,6)` gibt false zurück.

Hinweis: Mit der Funktion `Math.abs(num)` können Sie den absoluten Wert einer Zahl num erhalten.

Aufgabe 4: Berechnungen

3. Implementieren Sie die Methode Calculations.fast12(int z). Das Argument z ist nicht negativ. Die Methode gibt einen Boolean zurück. Die Methode soll true zurückgeben, wenn z nahe an einem Vielfachen von 12 ist. Eine Zahl x ist nahe an einer Zahl y, wenn eine der Zahlen um maximal 2 grösser oder kleiner ist als die andere Zahl. Ansonsten soll die Method false zurückgeben.

- fast12(12) gibt true zurück.
- fast12(14) gibt true zurück.
- fast12(10) gibt true zurück.
- fast12(15) gibt false zurück.

2. Implementieren Sie die Methode Calculations.magic7(int a, int b). Die Methode gibt einen Boolean zurück. Die Methode soll true zurückgeben, wenn einer der Parameter 7 ist oder wenn die Summe oder Differenz der Parameter 7 ist. Ansonsten soll die Methode false zurückgeben.

Beispiele

- magic7(2,5) gibt true zurück.
- magic7(7,9) gibt true zurück.
- magic7(5,6) gibt false zurück.

```
public static boolean magic7(int a, int b) { 6 usages  ↗ zelleraa
    int sum = a + b;
    int diff1 = a - b;
    int diff2 = b - a;

    return a == 7 || b == 7 || sum == 7 || diff1 == 7 || diff2 == 7;
}
```

Hinweis: Mit der Funktion Math.abs(num) können Sie den absoluten Wert einer Zahl num erhalten.

3. Implementieren Sie die Methode Calculations.fast12(int z). Das Argument z ist nicht negativ. Die Methode gibt einen Boolean zurück. Die Methode soll true zurückgeben, wenn z nahe einem Vielfachen von 12 ist. Eine Zahl x ist nahe an einer Zahl y, wenn eine der Zahlen um maximal 2 grösser oder kleiner ist als die andere Zahl. Ansonsten soll die Methode false zurückgeben.

Beispiele

- fast12(12) gibt true zurück.
- fast12(14) gibt true zurück.
- fast12(10) gibt true zurück.
- fast12(15) gibt false zurück.

```
public static boolean fast12(int z) { 5 usages  ↗ zelleraa *
    return Math.abs(z - 12) <= 2;
}
```

Aufgabe 5: Scrabble

In dieser Aufgabe sollen Sie Scrabble-Steine legen, mittels ASCII-Art auf der Konsole. Vervollständigen Sie die Methode `drawNameSquare` in der Klasse `Scrabble`. Diese Methode nimmt einen Namen als String-Parameter und soll den Namen als in einem Quadrat angeordnete Scrabble-Steine auf der Konsole (`System.out`) ausgeben. Wenn z.B. der String `Alfred` übergeben wird, sollte folgendes Bild ausgegeben werden:

```
+---+---+---+---+---+  
| A | L | F | R | E | D |  
+---+---+---+---+---+  
| L |           | E |  
+---+           +---+  
| F |           | R |  
+---+           +---+  
| R |           | F |  
+---+           +---+  
| E |           | L |  
+---+---+---+---+---+  
| D | E | R | F | L | A |  
+---+---+---+---+---+
```

Vorbesprechung

Aufgabe 1: Sieb des Eratosthenes

Schreiben Sie ein Programm "Sieb.java", das eine Zahl $limit$ einliest und die Anzahl der Primzahlen, die grösser als 1 und kleiner oder gleich dem $limit$ sind, ausgibt. Dazu ermitteln Sie in einem ersten Schritt alle Primzahlen, die kleiner oder gleich $limit$ sind. Dieses Teilproblem können Sie mit dem [Sieb des Eratosthenes](#) lösen. Das Sieb des Eratosthenes findet Primzahlen bis n . Man betrachtet alle Zahlen von 2 bis n und streicht zuerst alle Vielfachen der ersten Zahl (2). Dann geht man zur nächsten ungestrichenen Zahl (3) und wiederholt das Streichen ihrer Vielfachen. Das macht man, bis man dies für alle Zahlen gemacht hat. Sie können ein Boolean-Array verwenden, um zu speichern, welche Zahlen Primzahlen sind und welche nicht. Übrig bleiben die Primzahlen. Danach können Sie die Anzahl der gefundenen Primzahlen anhand dieses Arrays bestimmen.

Beispiel: Für $limit = 13$ sollte Ihr Programm 6 ausgeben (Primzahlen: 2, 3, 5, 7, 11, 13).

Hinweis: Es ist nicht zwingend nötig von 2 bis n zu gehen. Von 2 bis \sqrt{n} zu gehen reicht bereits aus, da eine Zahl $\leq n$ nicht einen Teiler grösser als \sqrt{n} ausser sich selbst haben kann.

Aufgabe 2: Arrays

1. Implementieren Sie die Methode `ArrayUtil.zeroInsert(int[] x)` in der Datei "ArrayUtil.java". Die Methode nimmt einen Array `x` als Argument und gibt einen Array zurück. Der zurückgegebene Array soll die gleichen Werte wie `x` haben, ausser: Wenn eine positive Zahl direkt auf eine negative Zahl folgt oder wenn eine negative Zahl direkt auf eine positive Zahl folgt, dann wird dazwischen eine 0 eingefügt.

Beispiele:

- Wenn `x` gleich `[3, 4, 5]` ist, dann wird `[3, 4, 5]` zurückgegeben.
- Wenn `x` gleich `[3, 0, -5]` ist, dann wird `[3, 0, -5]` zurückgegeben.
- Wenn `x` gleich `[-3, 4, 6, 9, -8]` ist, dann wird `[-3, 0, 4, 6, 9, 0, -8]` zurückgegeben.

Versuchen Sie, die Methode rekursiv zu implementieren.

Aufgabe 2: Arrays

2. Implementieren Sie die Methode `ArrayUtil.tenFollows(int[] x, int index)`. Die Methode gibt einen Boolean zurück. Die Methode soll `true` zurückgeben, wenn im Array `x` ab Index `index` der zehnfache Wert einer Zahl `n` direkt der Zahl `n` folgt. Dies muss nur für das erste Auftreten der Zahl `n` ab Index `index` im Array `x` geprüft werden. Ansonsten soll die Methode `false` zurückgeben.

Beispiele:

- `tenFollows([1, 2, 20], 0)` gibt `true` zurück.
- `tenFollows([1, 2, 7, 20], 0)` gibt `false` zurück.
- `tenFollows([3, 30], 0)` gibt `true` zurück.
- `tenFollows([3], 0)` gibt `false` zurück.
- `tenFollows([1, 2, 20, 5], 1)` gibt `true` zurück.
- `tenFollows([1, 2, 20, 5], 2)` gibt `false` zurück.

Die `main` Methode in `ArrayUtil` gibt die oben genannten Beispielaufrufe sowie das entsprechende Ergebnis der jeweiligen Methode aus. Hiermit können Sie überprüfen, ob Ihre Implementierungen die richtigen Ergebnisse zurückliefern. In "ArrayUtilTest.java" im Ordner "test" in der Übungsvorlage finden Sie zusätzlich einige Unit-Tests für beide Methoden (für eine detaillierte Beschreibung zu automatisiertem Testen und der Ausführung solcher Tests siehe Aufgabe 3). Sie können die `main` Methode und die Tests beliebig ändern und/oder mit Ihren eigenen Inputs erweitern.

Aufgabe 3: 2D Arrays

Gegeben einer Matrix M , prüfen Sie zuerst ob diese eine $n \times n$ Matrix ist, deren Elemente positive ganze Zahlen sind. Danach prüfen Sie ob zusätzlich alle Zahlen kleiner gleich n^2 sind. Somit gilt nun $0 < m_{i,j} \leq n^2$. Prüfen Sie ebenfalls, ob die Elemente der Matrix jeweils genau einmal vorkommen, sprich ob $m_{x,y} = m_{p,q} \Rightarrow (x = p) \wedge (y = q)$ gilt. Wir sagen, dass die Matrix M *perfekt* ist, wenn zusätzlich alle Zeilensummen und Spaltensummen gleich sind (also $\sum_{k=0}^{k=n-1} m_{i,k} = \sum_{k=0}^{k=n-1} m_{j,k}$ für alle i, j und $\sum_{k=0}^{k=n-1} m_{k,i} = \sum_{k=0}^{k=n-1} m_{k,j}$ für alle i, j mit $0 \leq i, j < n$).

Vervollständigen Sie die Methode `boolean checkMatrix(int[][] m)` von der Klasse `Matrix`, so dass diese Methode `true` zurückgibt wenn die Input Matrix *perfekt* ist, und `false` sonst. Sie können davon ausgehen, dass der Parameter `m` nicht `null` ist. Alle anderen Eigenschaften müssen Sie selber testen. Eine Matrix ist nur perfekt, wenn alle genannten Eigenschaften gelten.

Testen Sie Ihr Programm ausgiebig - am besten mit JUnit - und pushen Sie die Lösung vor dem Abgabetermin. Wir haben Ihnen einen JUnit Test in der Klasse `MatrixTest` bereits erstellt.

Aufgabe 4: Testen mit JUnit

Zweck des Programms:

- Wochentag eines Datums (nach 01.01.1900) ausgeben
Beispiel: 13.10.2017 → Friday
Gibt fälschlicherweise aber “*The 13.10.2017 is a Sunday*” aus.
- Berücksichtigt Schaltjahre (“Leap year”)

Funktionsweise:

1. Überprüft, ob Datum OK ist
2. Zählt die Tage ab 1.1.1900 bis zum eingegebenen Datum
3. Wochentag = Tage % 7

Aufgabe 4: Testen mit JUnit

Tests in *PerpetualCalendarTest.java*

- Einzelne Tests prüfen Rückgabewerte von einzelnen Methoden des Programms *PerpetualCalendar.java*
- Tests sollten *interessante* Parameter für die Methoden testen
- Beispiel `testCountDaysInYear():`

```
assertEquals(366, PerpetualCalendar.countDaysInYear(1904));
```

1904 ist ein Schaltjahr, also sollte `countDaysInYear()` 366 Tage zurückgeben

DEMO

Aufgabe 5: Matching Numbers

Implementieren Sie die Methode `Match.matchNumber(long A, int M)`. Die Methode soll für eine Zahl A und eine nicht-negative drei-stellige Zahl M die Position von M in A zurückgeben. Sei M eine Zahl mit den Ziffern $M_2M_1M_0$ (das heisst, es gilt $M = M_0 + 10 \cdot M_1 + 100 \cdot M_2$), wobei jede Ziffer 0 sein kann. Zusätzlich sei A eine Zahl, sodass A_i die i -te Ziffer von A ist (das heisst, es gilt $|A| = \sum_i 10^i \cdot A_i$), wobei A unendlich viele führende Nullen hat. Die Position von M in A ist die kleinste Zahl j , sodass $A_j = M_0$ und $A_{j+1} = M_1$ und $A_{j+2} = M_2$ gilt. Die Methode soll -1 zurückgeben, falls es kein solches j gibt.

Beispiele:

`matchNumber(32857890, 789)` soll 1 zurückgeben.

`matchNumber(37897890, 789)` soll 1 zurückgeben.

`matchNumber(1800765, 7)` soll 2 zurückgeben.

`matchNumber(1800765, 8)` soll -1 zurückgeben (die drei Ziffern von 8 sind 008).

`matchNumber(75, 7)` soll 1 zurückgeben (da 007 and Position 1 von 0075 ist).

Aufgabe 5: Matching Numbers

Implementieren Sie die Berechnung in der Methode `int matchNumber(long A, int M)`, welche sich in der Klasse Match befindet. Die Deklaration der Methode ist bereits vorgegeben. Sie können davon ausgehen, dass $0 \leq M < 1000$ gilt.

In der main Methode der Klasse Match finden Sie die oberen Beispiele als kleine Tests, welche Beispiel-Aufrufe zur `matchNumber`-Methode machen und welche Sie als Grundlage für weitere Tests verwenden können. In der Datei `MatchTest.java` geben wir die gleichen Tests zusätzlich auch als JUnit Test zur Verfügung. Sie können diese ebenfalls nach belieben ändern. Es wird *nicht* erwartet, dass Sie für diese Aufgabe den JUnit Test verwenden.

Tipp: Die Methode `Integer.toString(int i)` wandelt einen Integer in einen String um.

Aufgabe 6: Substring-Counter (Bonus!)

- Diese Aufgabe gibt Bonuspunkte!
- Regeln findet ihr hier:

[https://lec.inf.ethz.ch/infk/eprog/2025/exercises/additionals/
bonusaufgaben_regeln.pdf](https://lec.inf.ethz.ch/infk/eprog/2025/exercises/additionals/bonusaufgaben_regeln.pdf)

Regeln Zusammenfassung (Llama-3-8B)



Introduction to Programming (252-0027-00)

Rules for Bonus Tasks

Last updated: September 12, 2025 These rules apply unless a task explicitly states otherwise.

1. **Java version** Submissions are tested with **Java 21**. If your solution requires a different Java version, you receive **0 points**.
2. **Automatic testing** Bonus tasks are automatically tested and evaluated. **Comments are ignored**. Program output must match the specification **exactly**. If the program does not compile, it will not be evaluated (**0 points**).
3. **Commit and push** You must **commit and push** your submission to **GitLab** before the deadline. The **last commit** before the deadline is evaluated.
4. **Solution location** Place the **entire solution** in the `src` directory of the respective bonus project. **No extra files** are allowed in that directory.
5. **Case sensitivity** Java is case-sensitive. Use the **exact capitalization and spelling** for method and constructor names.
6. **Method and constructor declarations** Do not change the provided signatures: name, return/parameter types, modifiers (public, static, protected, private), and declared exceptions.
7. **Adding new methods and classes** You may add new methods, constructors, attributes, classes, and interfaces.
8. **No additional output** Your program must not print anything beyond what the task specifies.
9. **No static attributes** Do not use **static attributes**, as the program may be run multiple times without reinitializing them.
10. **No unnecessary imports** Import only the classes you actually use. (Tip: IntelliJ can **optimize imports**.)
11. **Original solution** The submission must be **your own work**. **No copying**, and do not **share** your solution with others.
12. **Collaboration with others** You may **discuss** tasks with other students, but **do not take notes or recordings**. Wait at least one hour after discussions before developing your solution or writing notes.
13. **Using generative AI** If you use generative AI (e.g., ChatGPT, Copilot), the **prompts must be written by you**, and **no code** from the AI may be copied into your submission.
14. **Finding the push time** To find the push time in GitLab, check:
 - **Activity view** (shows push and other events), or
 - **Commit history** and locate the **last commit** pushed before the deadline.

Kahoot

<https://create.kahoot.it/details/e70bd2da-9dda-4697-a00f-b35f43b5ed60>