

252-0027

Einführung in die Programmierung
Übungen

Woche 5: Arrays, Methoden, Debugger

Timo Stucki

Departement Informatik
ETH Zürich

Organisatorisches

- Mein Name: *Timo Stucki*
- Bei Fragen: *tistucki@student.ethz.ch*
 - Mails bitte mit «[EProg25]» im Betreff
- 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

Discord: timostucki

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**



```
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 }
```

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 $0 \leq i < n$
- einen Eintrag erhalten wir mit `arr[i][j]` für $0 \leq i, j < n$
- **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]
0	
1	
2	

i	0	1	2	3	4	5
arrOne[i]	4	501	-1	200	42	0

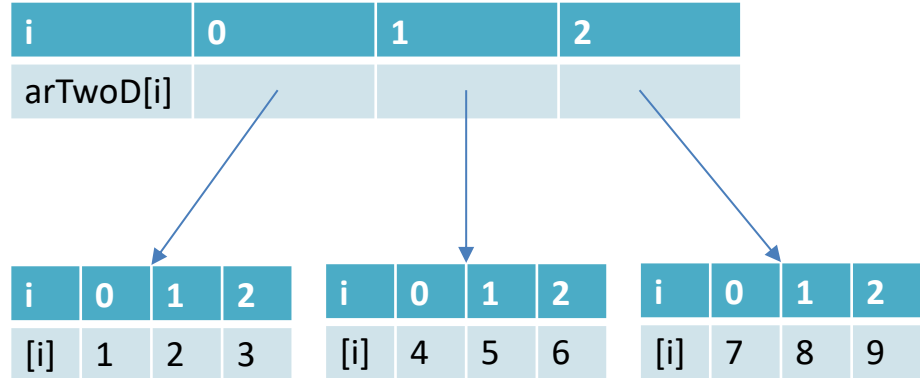
i	0	1	2
arrTwo[i]	11	21	-170

i	0	1	2	3
arrThree[i]	100	1	-321	3

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

i	0	1	2	3	4	5
arOneD[i]	1	2	3	4	5	6

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



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

arTwoD[i] =

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

arTwoD[i][j] =

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

Methoden

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

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

The diagram illustrates the components of the Java method signature `public static int add(int a, int b){ ...}`. Brackets are used to group the keywords and the return type, and labels are placed below them to identify their roles:

- A red bracket under `public` is labeled **Sichtbarkeit** (Visibility).
- A green bracket under `static` is labeled **Rückgabotyp** (Return type).
- A yellow bracket under `add` is labeled **Name** (Name).
- A yellow bracket under `(int a, int b)` is labeled **Parameter** (Parameter).

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

Sichtbarkeit

Rückgabotyp

Name

Parameter

Signatur

The diagram illustrates the components of a Java method signature. The code 'public static int add(int a, int b){ ...' is shown with color-coding: 'public' is red, 'static' is black, 'int' is green, 'add' is yellow, and the parameters '(int a, int b)' are green. Brackets and labels identify these parts: a red bracket under 'public' is labeled 'Sichtbarkeit'; a green bracket under 'int' is labeled 'Rückgabotyp'; a yellow bracket under 'add' is labeled 'Name'; a yellow bracket under '(int a, int b)' is labeled 'Parameter'; and a black bracket spanning from 'add' to the closing brace is labeled 'Signatur'.

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

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

Objekte

- Variable speichert Referenz auf Speicherort
- Grösse ist variabel
- Standardwert ist **NULL** (keine Referenz in den Speicher)

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

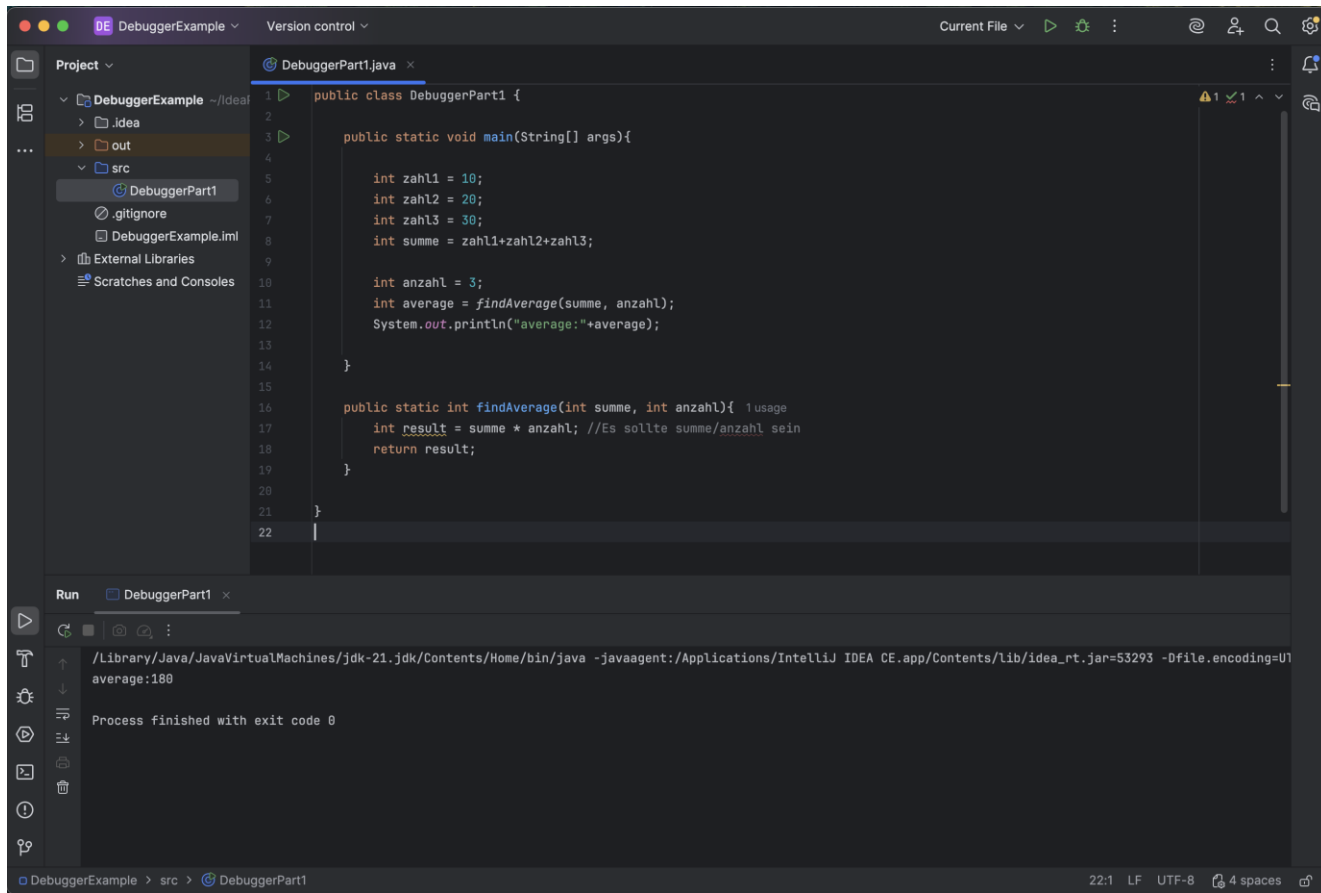
- `checksum(258)` gibt 15 zurück.
- `checksum(49)` gibt 13 zurück.
- `checksum(12)` gibt 3 zurück.

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.

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.



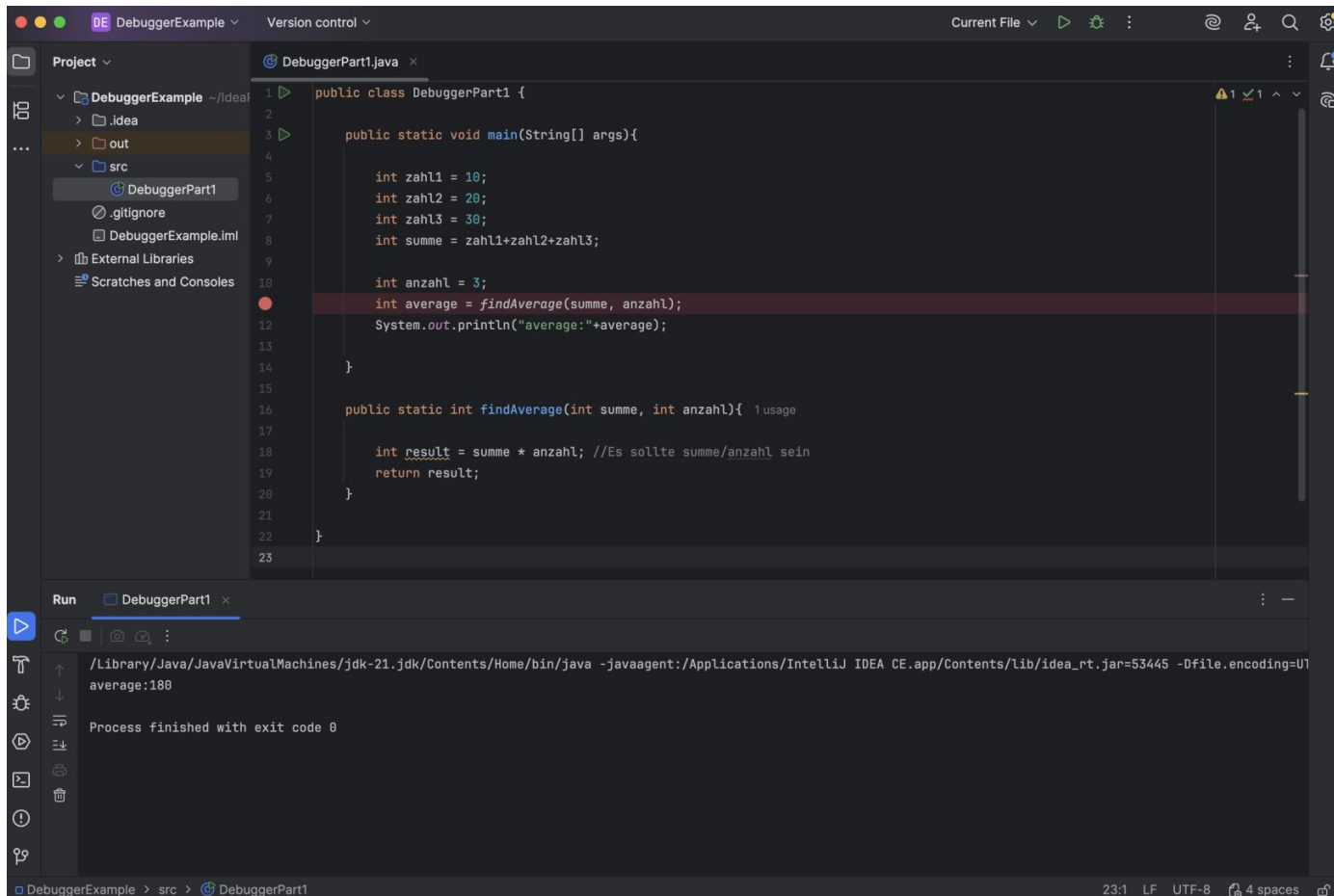
Das Programm macht nicht, was du erwartest, und du kannst den Fehler nicht finden?

Benutze den Debugger!

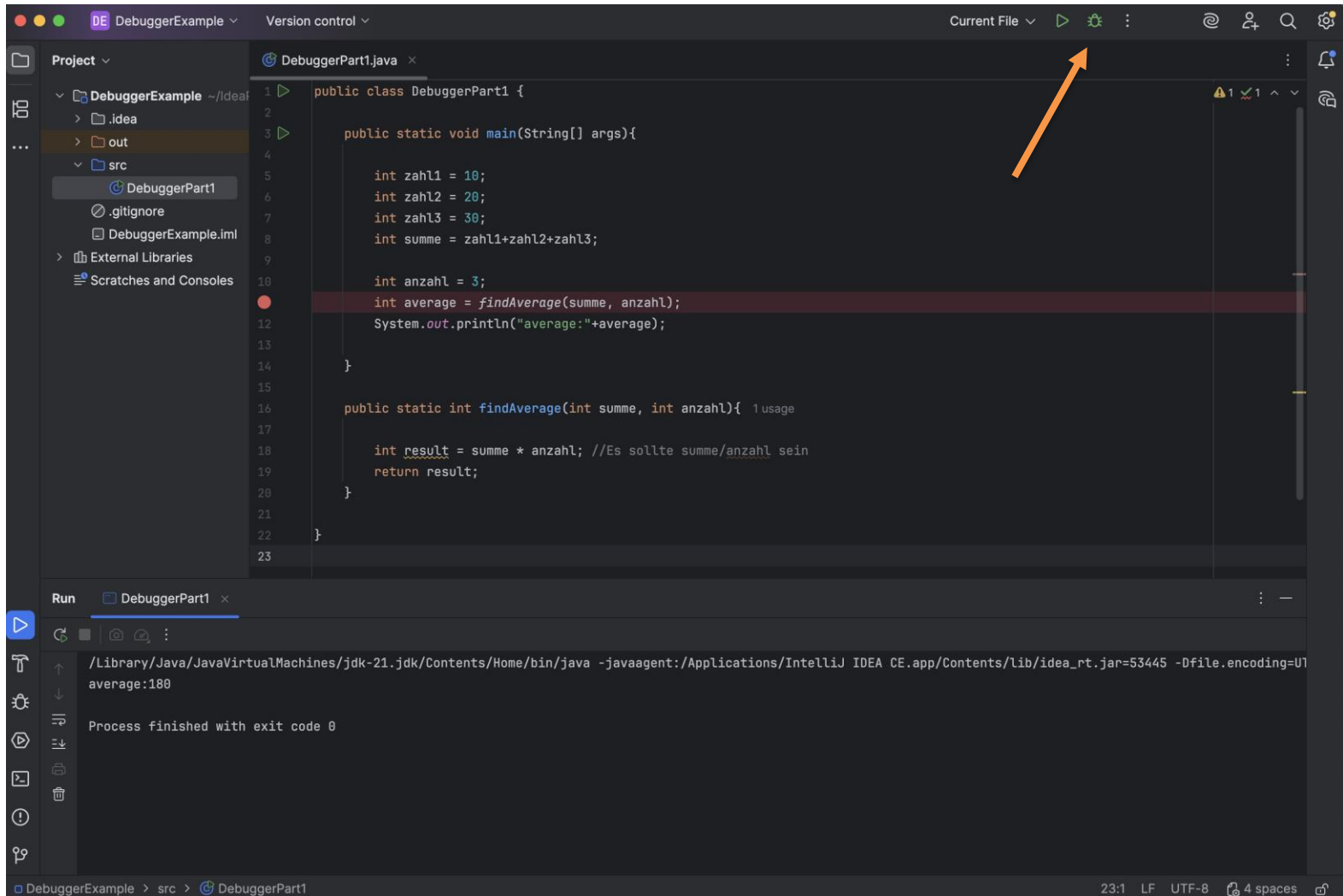
Breakpoints

- 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 Zeilennummern klicken, um den Breakpoint zu setzen





- 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.



DebuggerExample Version control Current File

Project

- DebuggerExample ~//Idea
- .idea
- out
- src
 - DebuggerPart1
 - .gitignore
 - DebuggerExample.iml
- External Libraries
- Scratches and Consoles

DebuggerPart1.java

```
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);    summe: 60    anzahl: 3
12        System.out.println("average:"+average);
13
14    }
15
16    public static int findAverage(int summe, int anzahl){    1 usage
17
18        int result = summe * anzahl; //Es sollte summe/anzahl sein
19        return result;
20    }
21
22 }
23
```

Debug DebuggerPart1

Threads & Variables Console

✓ "main" @ 11:11:11: RUNNING

main:11, DebuggerPart1

args = {String[0]@823} []

zahl1 = 10

zahl2 = 20

zahl3 = 30

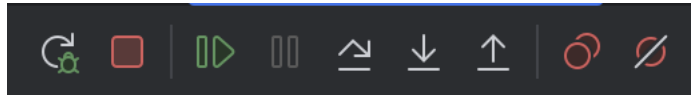
summe = 60

anzahl = 3

DebuggerExample > src > DebuggerPart1

11:1 LF UTF-8 4 spaces

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



Termination
- Um den Debugger zu stoppen

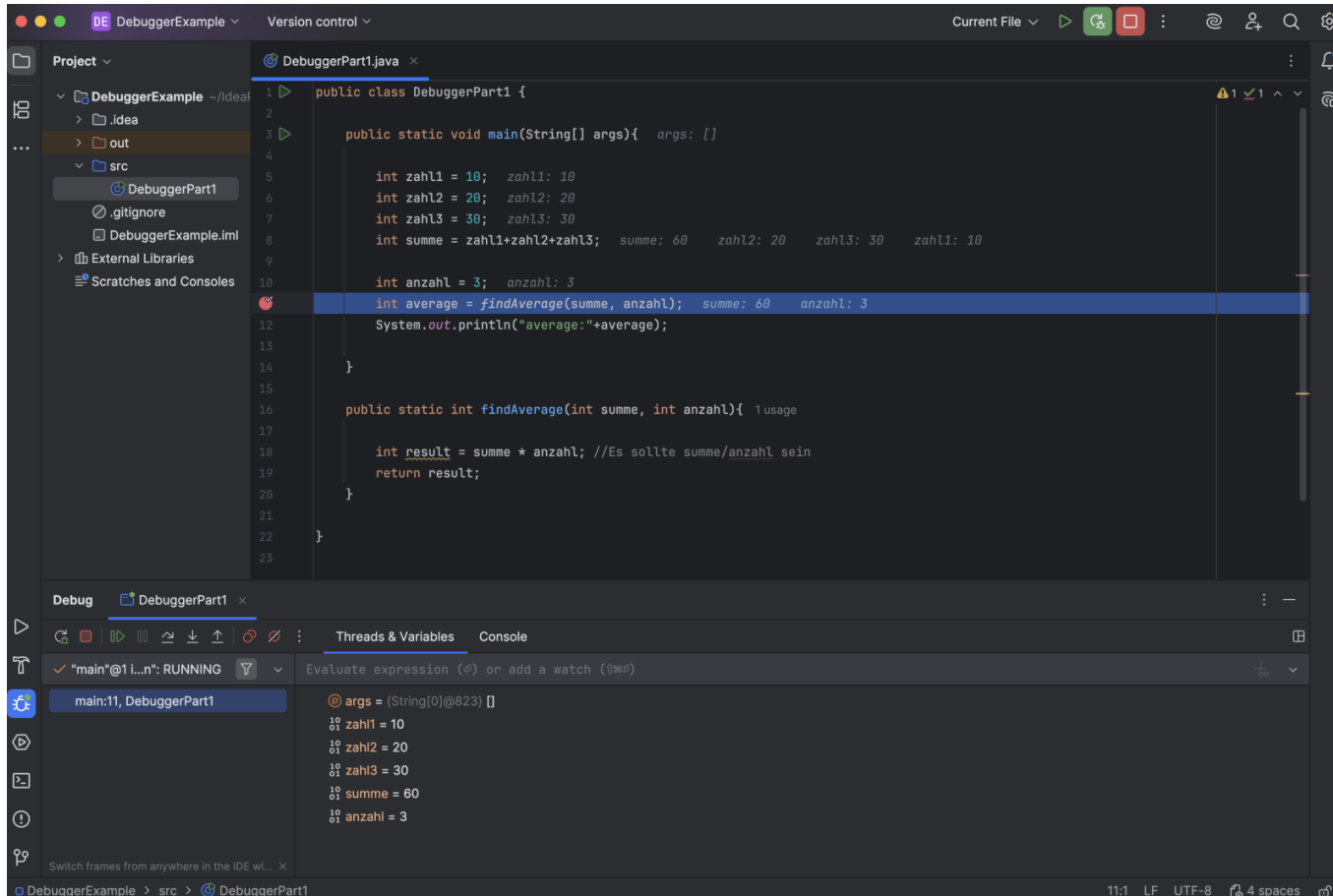
Step over
- Um zur nächsten Zeile zu gehen.

Step into
- Um in eine Methode zu springen.

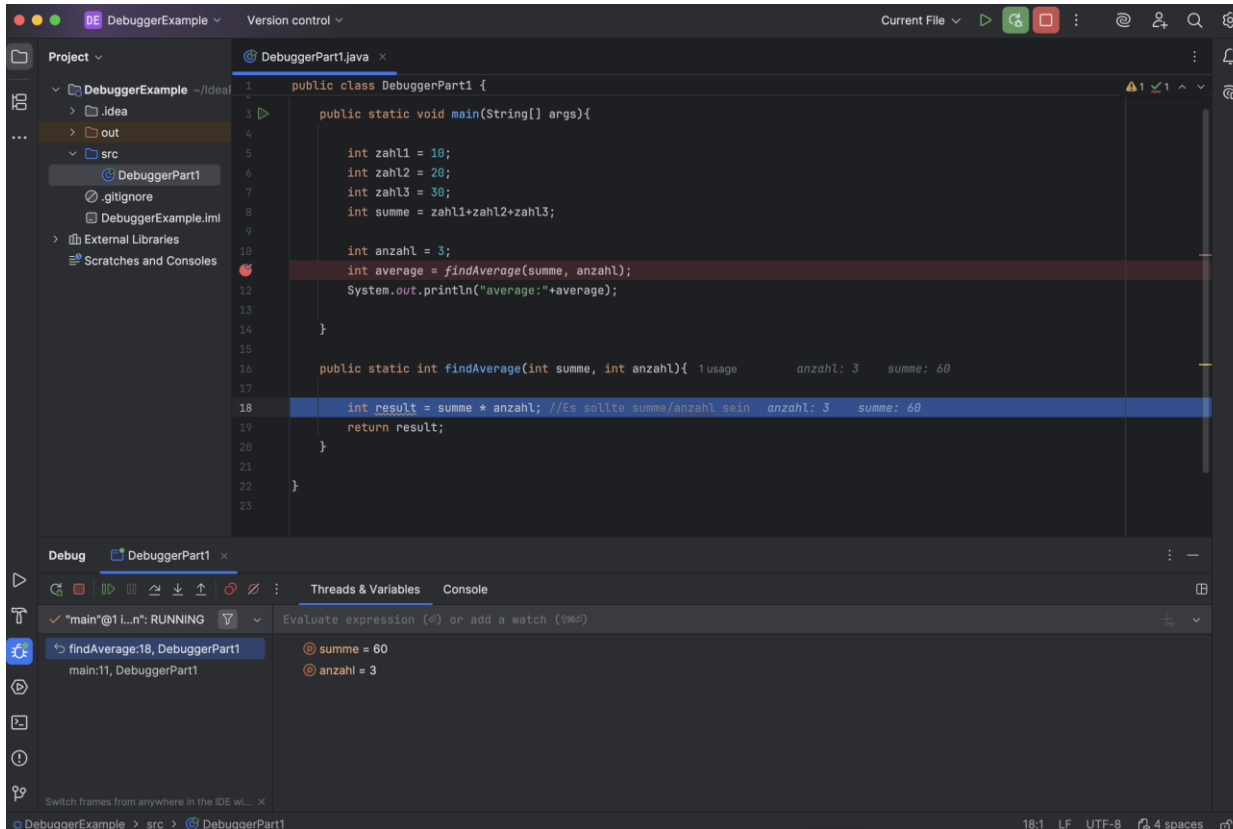
View breakpoints
- Alle breakpoints anzeigen

Step out
- Um aus eine Methode raus zu treten

Von dieser Zeile aus weitergehen...

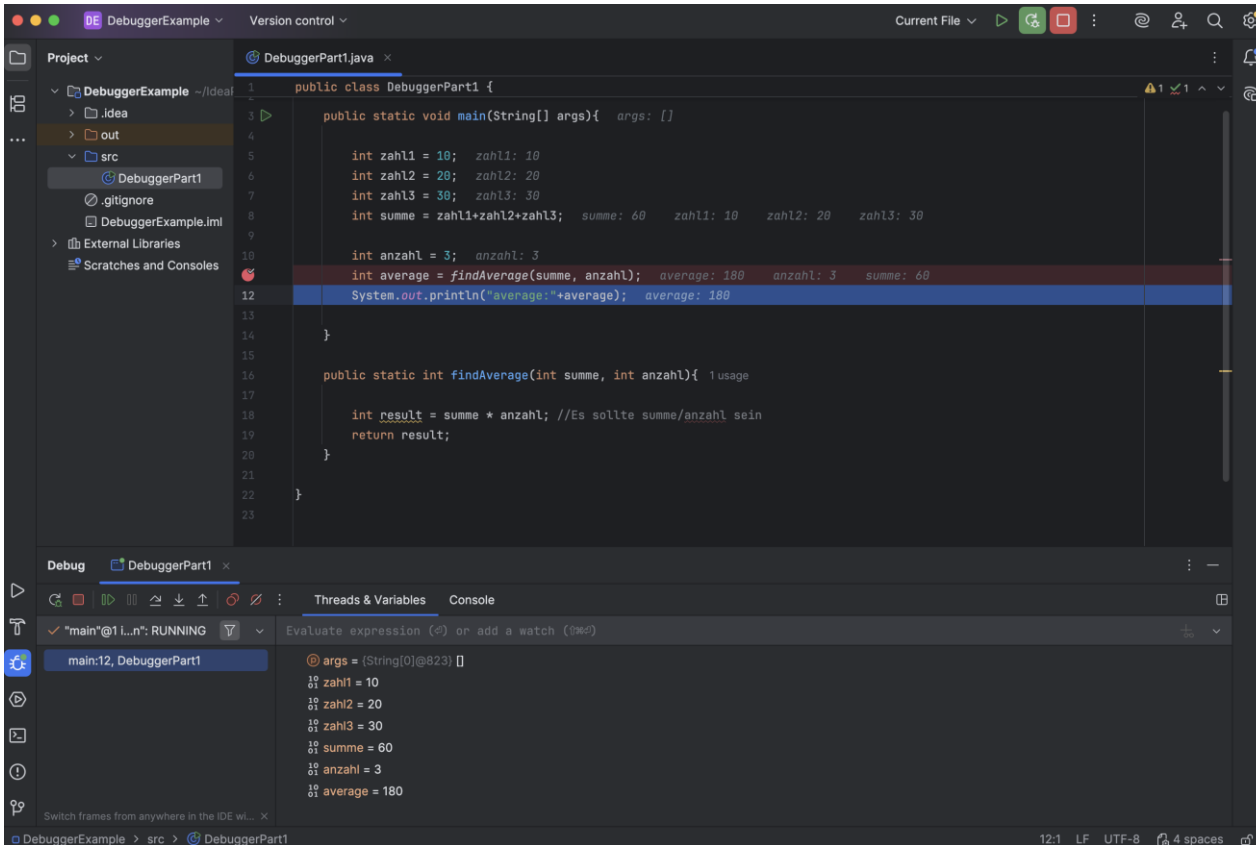


Mit step into:



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

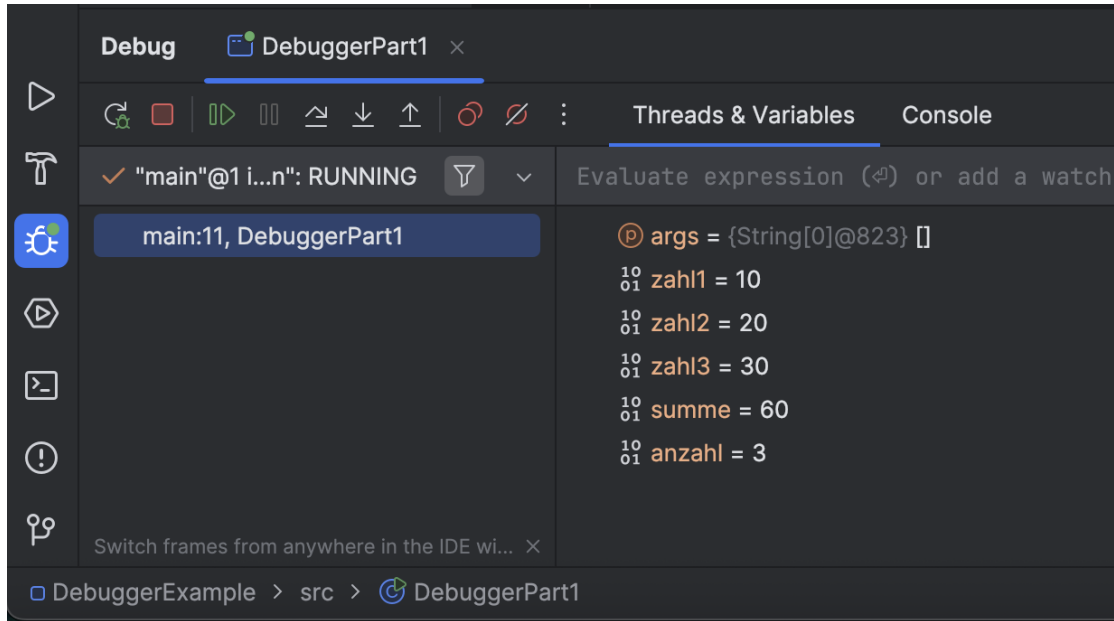
Mit step over:



- Der Methodenaufwurf 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.



- 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
16    public static int findAverage(int summe, int anzahl){  1 usage
17
18        int result = summe * anzahl; //Es sollte summe/anzahl sein
19        return result;
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

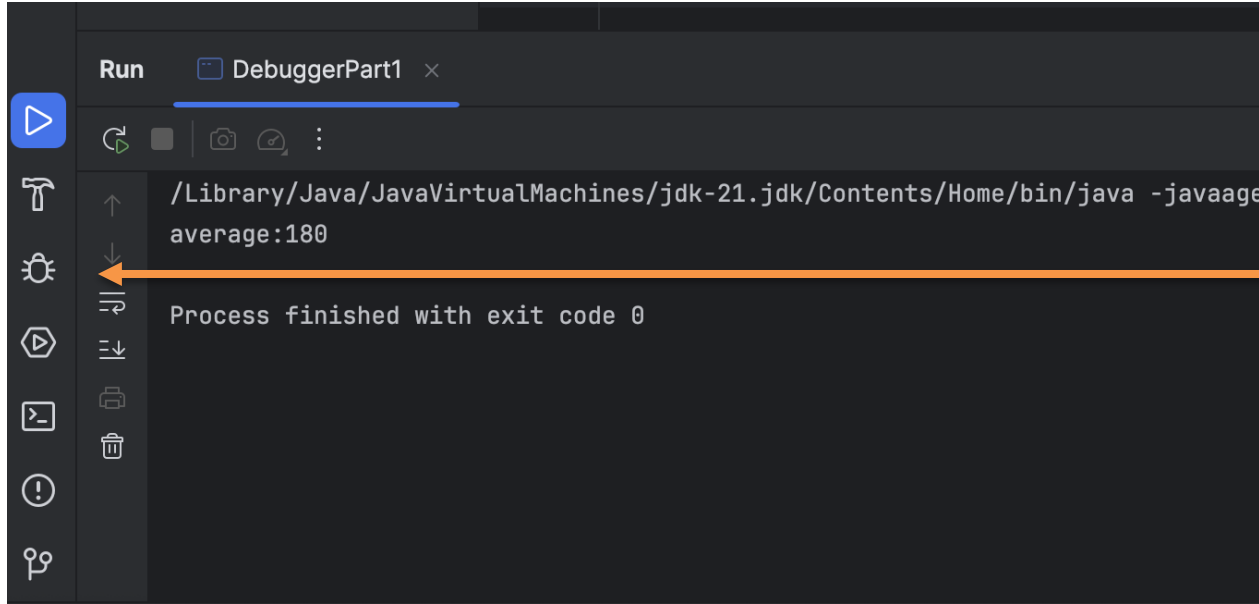
Evaluate expression (⌘) or add a watch (⌘%⌘)

🔍 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:



Debug
auswählen

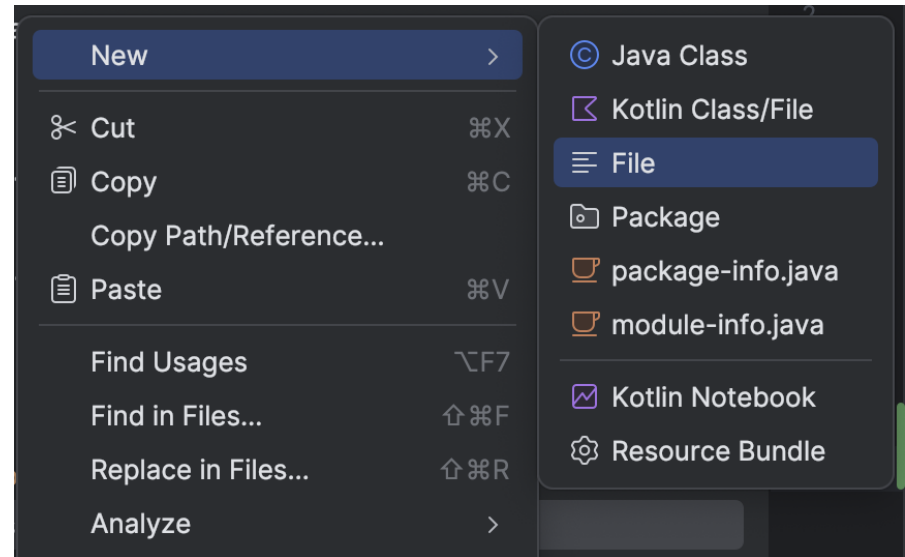
Feedback

Feedback

- Ihr sagt den TAs ab u05 wo ihr Feedback haben möchtet.
- 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 geggraded**)

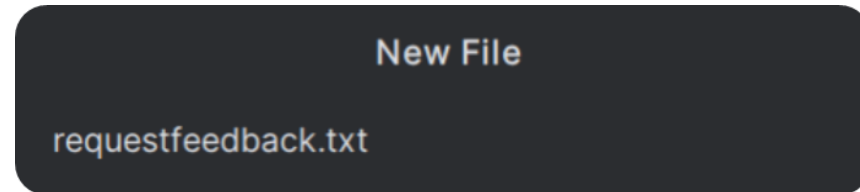
Feedback Datei erstellen

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



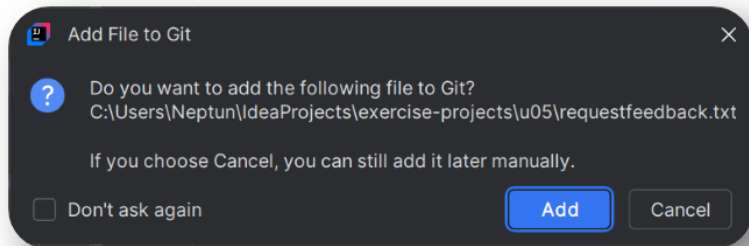
Feedback Datei erstellen

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 2 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  $2^{(\text{exp} - 1)} \leq \text{number}$ 
        // 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);
```

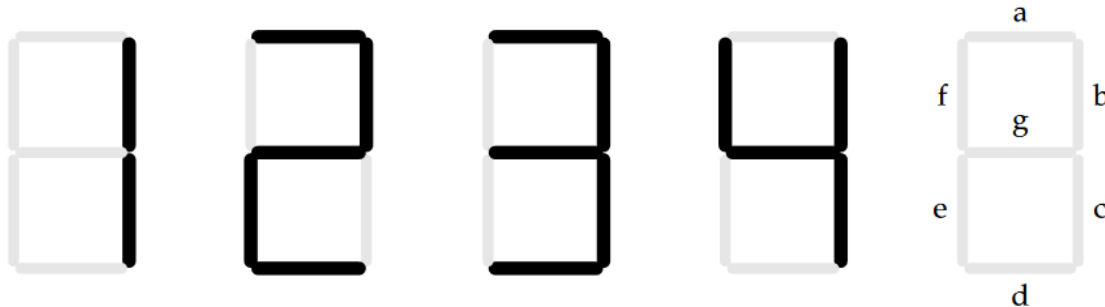
Negation der Bedingung in 1.

Aufgabe 3: Zahlenerkennung

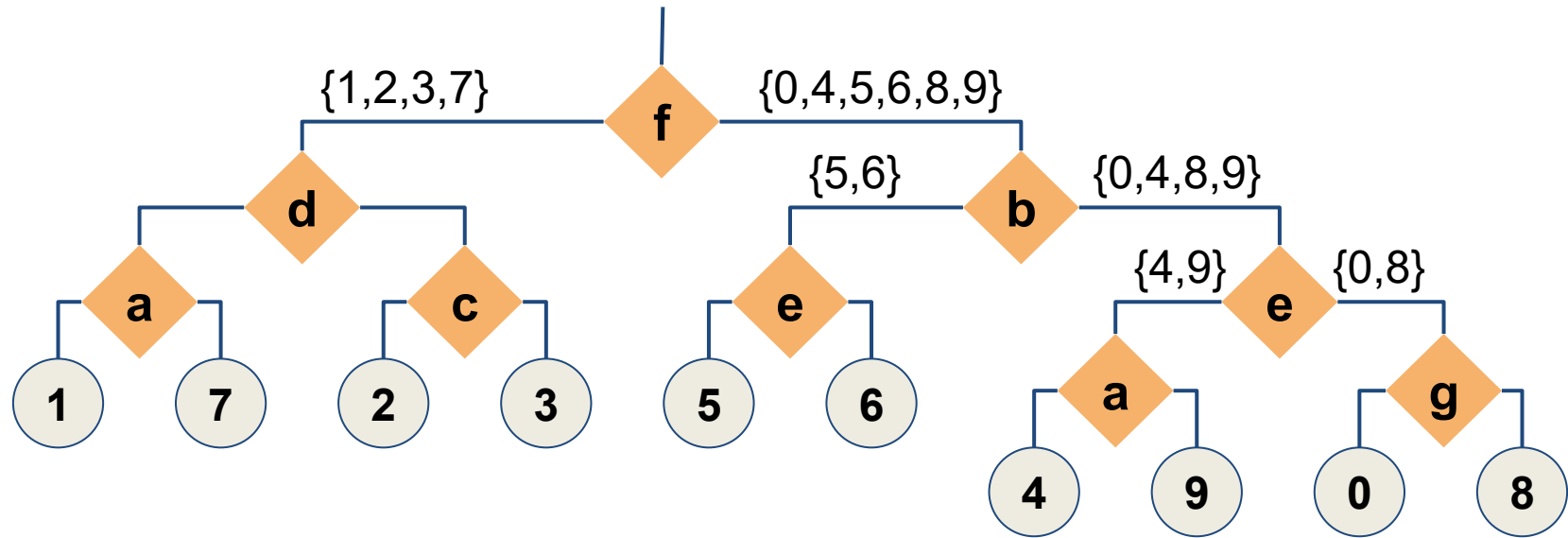
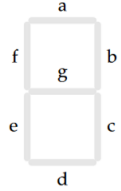
Für diese Aufgabe verwenden wir einen String um die erleuchtenden Segmente einer **Sieben-segmentanzeige** zu kodieren. Die Segmente sind, wie im Bild gezeigt, von a bis g nummeriert. Die Kodierung einer möglichen Anzeige ist ein String, in welchem der Buchstabe 'x' genau dann vorkommt, wenn das 'x'te 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 Methode `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

Meine Aufgaben Ratings:



- **Best Case: Ihr macht alle Aufgaben (, wenn die Zeit reicht)**
- **Falls nicht: Ich mache Ratings zur Wichtigkeit der einzelnen Aufgaben**
- **(Keine offizielle Empfehlung, meine subjektive Meinung auf Basis meiner eigenen Erfahrung als Student in diesem Kurs)**
- **! Sagen nichts über die Schwierigkeit der Aufgaben aus !**

**Wichtig für die Prüfung:
(Prüfungs- oder
prüfungsähnliche Aufgaben)**



**Wichtig für euer Verständnis:
(Aber nicht in Prüfungsform)**



**Wichtig für tiefes Verständnis/
Zusatzaufgaben:
(Bspw. viele vom gleichen Typ)**



Webseite



■ timostucki.com

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

