

252-0027

**Einführung in die Programmierung
Übungen**

Woche 5: Arrays, Methoden, Debugger

Timo Baumberger

Departement Informatik

ETH Zürich

Programm

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

Organisatorisches

- Mein Name: Timo Baumberger
- Bei Fragen: tbaumberger@student.ethz.ch
 - Mails bitte mit «[EProg24]» 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

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 $0 \leq i < n$
- 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}  
};
```

i	0	1	2
arTwoD[i]			

i	0	1	2
[i]	1	2	3

i	0	1	2
[i]	4	5	6

i	0	1	2
[i]	7	8	9

```
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
```

String Performance

```
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: Pogrg

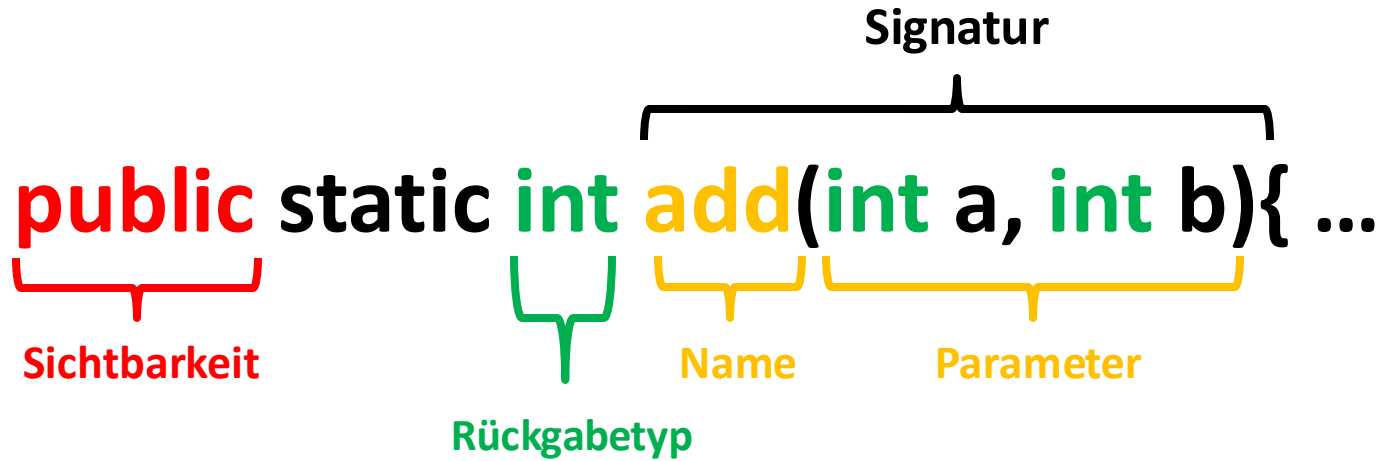
Methoden

```
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 parts of the signature, with labels below them:

- A red bracket under `public` is labeled **Sichtbarkeit** (Visibility).
- A green bracket under `int` 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).



Java: Methoden Signatur

- **Kombination aus Name der Methode, Anzahl der Parameter, Typen der Datentypen und Reihenfolge der Parameter**
- **Reihenfolge 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 }
```

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 Analysieren von Fehlern 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 an IDE with the following components:

- Package Explorer:** Shows a project structure with a package named 'src' containing several Java files, including 'DebuggerBeispiel.java'.
- Code Editor:** Displays the source code of 'DebuggerBeispiel.java'. The code is as follows:

```
1 public class DebuggerBeispiel {
2
3
4
5     public static void main(String[] args) {
6
7         int num1 = 10;
8         int num2 = 20;
9         int num3 = 30;
10
11        // Calculate the average (intentional bug here)
12        int sum = num1 + num2 + num3;
13        int anzahl = 3;
14        int average = findAverage(sum, anzahl);
15        System.out.println("The average is: " + average);
16
17    }
18
19    public static int findAverage(int sum, int anzahl) {
20
21        int average = sum * anzahl; // This should be sum / 3 to get the correct average
22        return average;
23    }
24
25 }
26
27
28
29 }
30
```
- Task List:** Empty.
- Outline:** Shows the class structure with 'main(String[]): void' and 'findAverage(int, int): int'.
- Console:** Shows the output: 'The average is: 180'.
- Problems:** Shows a message: '<terminated> DebuggerBeispiel [Java Application] |Library|Java|JavaVirtualMachines|temurin-21|re|Contents|Home|bin|java (10 Oct 2024, 15:04:44 - 15:04:44) [pid: 10724]'.

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 doppelt klicken, um den Breakpoint zu setzen (ein blaues Kreis-Icon sollte in der Spalte erscheinen).

The screenshot shows an IDE window with the following components:

- Package Explorer:** Shows a project structure with folders like 'JUnitExample', 'Uebungsstunde02', 'Uebungsstunde03', 'JUnit 5', 'JRE System Library', and 'arc'. The 'arc' folder contains 'DebuggerBeispiel.java' and 'DebuggerExample.java'.
- Code Editor:** Displays the source code of 'DebuggerBeispiel.java'. A breakpoint is set on line 14. The code includes a `main` method and a `findAverage` method. The `main` method calculates the sum of three numbers (10, 20, 30) and prints the average. The `findAverage` method calculates the average based on the sum and the number of elements.
- Task List:** Shows a search bar and 'All' / 'Activate...' buttons.
- Outline:** Shows the class structure with `main(String[]): void` and `findAverage(int, int): int`.
- Console:** Shows the output of the program: 'The average is: 30'.

```
1 public class DebuggerBeispiel {
2
3
4
5     public static void main(String[] args) {
6
7         int num1 = 10;
8         int num2 = 20;
9         int num3 = 30;
10
11        // Calculate the average (intentional bug here)
12        int sum = num1 + num2 + num3;
13        int anzahl = 3;
14        System.out.println("The average is: " + average);
15
16    }
17
18
19
20    public static int findAverage(int sum, int anzahl) {
21
22        int average = sum * anzahl; // This should be sum / 3 to get the correct average
23        return average;
24
25    }
26
27
28
29 }
30
```

Console Output: The average is: 30

- Da wir sicher sind, dass **sum** 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 an IDE with the following components:

- Package Explorer:** Shows a project structure with a package named 'src' containing several Java files, including 'DebuggerBeispiel.java' which is selected.
- Code Editor:** Displays the source code of 'DebuggerBeispiel.java'. The code is as follows:

```
1
2 public class DebuggerBeispiel {
3
4
5     public static void main(String[] args) {
6
7         int num1 = 10;
8         int num2 = 20;
9         int num3 = 30;
10
11         // Calculate the average (intentional bug here)
12         int sum = num1 + num2 + num3;
13         int anzahl = 3;
14         int average = findAverage(sum, anzahl);
15         System.out.println("The average is: " + average);
16
17     }
18
19
20     public static int findAverage(int sum, int anzahl) {
21
22         int average = sum * anzahl; // This should be sum / 3 to get the correct average
23         return average;
24
25     }
26
27
28 }
29
30
```

Line 14 is highlighted, and a debugger breakpoint is set at this line. The 'Debugger' icon in the top toolbar is also highlighted with an arrow.
- Task List:** Shows a search bar and a list of tasks.
- Outline:** Shows the class structure with 'main(String[]): void' and 'findAverage(int, int): int' listed.
- Console:** Shows the output of the program: 'The average is: 30'.

The screenshot shows an IDE window with the following components:

- Package Explorer:** Shows a project structure with folders like 'JUnitExample', 'Uebungsstunde02', 'Uebungsstunde03', 'JRE System Library', 'JUnit 5', and 'src'. The 'src' folder contains files like 'DebuggerBeispiel.java', 'DebuggerExample.java', 'Examples.java', 'Potenzieren.java', 'Reihe.java', 'Reihe2.java', and 'Strings.java'.
- Code Editor:** Displays the source code of 'DebuggerBeispiel.java'. The code includes a 'main' method and a 'findAverage' method. Line 14 is highlighted: `int average = findAverage(sum, anzahl);`.

```
1 public class DebuggerBeispiel {
2
3
4
5 public static void main(String[] args) {
6
7     int num1 = 10;
8     int num2 = 20;
9     int num3 = 30;
10
11     // Calculate the average (intentional bug here)
12     int sum = num1 + num2 + num3;
13     int anzahl = 3;
14     int average = findAverage(sum, anzahl);
15     System.out.println("The average is: " + average);
16
17 }
18
19
20 public static int findAverage(int sum, int anzahl) {
21
22     int average = sum * anzahl; // This should be sum / 3 to get the correct average
23     return average;
24
25 }
26
27
28 }
29
30 }
```
- Dialog Box:** A 'Confirm Perspective Switch' dialog is open. It contains the following text:
 - 'This kind of launch is configured to open the Debug perspective when it suspends.'
 - 'This Debug perspective supports application debugging by providing views for displaying the debug stack, variables and breakpoints.'
 - 'Switch to this perspective?'
 - An unchecked checkbox labeled 'Remember my decision'.
 - Two buttons: 'No' and 'Switch'.An arrow points to the 'Switch' button.
- Task List:** Shows a search bar and 'All' and 'Activate...' buttons.
- Outline:** Shows a tree view of the code structure:
 - DebuggerBeispiel
 - main(String[]): void
 - findAverage(int, int): int
- Bottom Panel:** Shows 'Problems @ Javadoc Declaration Console X' and a status bar with 'DebuggerBeispiel [Java Application] /Library/Java/JavaVirtualMachines/temurin-21.jre/Contents/Home/bin/java (10 Oct 2024, 15:07:55) [pid: 10778]'.

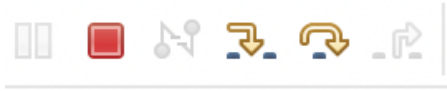
The screenshot shows an IDE with the following components:

- Project Explorer:** Shows the project structure with 'DebuggerBeispiel' and its main method.
- Code Editor:** Displays the source code of 'DebuggerBeispiel.java'. The code is as follows:

```
1 public class DebuggerBeispiel {
2
3
4
5     public static void main(String[] args) {
6
7         int num1 = 10;
8         int num2 = 20;
9         int num3 = 30;
10
11         // Calculate the average (intentional bug here)
12         int sum = num1 + num2 + num3;
13         int anzahl = 3;
14         int average = findAverage(sum, anzahl);
15         System.out.println("The average is: " + average);
16
17     }
18
19
20     public static int findAverage(int sum, int anzahl) {
21
22         int average = sum * anzahl; // This should be sum / 3 to get the correct average
23         return average;
24
25     }
26
27 }
28
29 }
30
```
- Variables Window:** Shows the current state of variables:

Name	Value
args	String[] (id=19)
num1	10
num2	20
num3	30
sum	60
anzahl	3
- Console:** Shows the output of the program: "DebuggerBeispiel [Java Application] /Library/Java/JavaVirtualMachines/temurin-21.jre/Contents/Home/bin/java (10 Oct 2024, 14:11:35) [pid: 9837]"

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



Termination
- Um den Debugger
zu stoppen

Step into (F5)
- Um in eine
Methode zu
springen.

Step over (F6)
- Um zur nächsten
Zeile zu gehen.

Von dieser Zeile aus weitergehen...

The screenshot shows an IDE with a Java application being debugged. The main method is highlighted at line 14, and the Variables window shows the current state of the program.

```
1 public class DebuggerBeispiel {
2
3
4
5 public static void main(String[] args) {
6
7     int num1 = 10;
8     int num2 = 20;
9     int num3 = 30;
10
11     // Calculate the average (intentional bug here)
12     int sum = num1 + num2 + num3;
13     int anzahl = 3;
14     int average = findAverage(sum, anzahl);
15     System.out.println("The average is: " + average);
16
17
18 }
19
20 public static int findAverage(int sum, int anzahl) {
21
22     int average = sum * anzahl; // This should be sum / 3 to get the correct average
23     return average;
24
25 }
26
27 }
28
29 }
30
```

Name	Value
no method return value	
args	String[] (id=19)
num1	10
num2	20
num3	30
sum	60
anzahl	3

Console: DebuggerBeispiel [Java Application] /Library/Java/JavaVirtualMachines/temurin-21.jre/Contents/Home/bin/java (10 Oct 2024, 14:11:35) [pid: 9837]

Mit step into:

```
DebuggerBeispiel.java X
1
2 public class DebuggerBeispiel {
3
4
5 public static void main(String[] args) {
6
7     int num1 = 10;
8     int num2 = 20;
9     int num3 = 30;
10
11     // Calculate the average (intentional bug here)
12     int sum = num1 + num2 + num3;
13     int anzahl = 3;
14     int average = findAverage(sum, anzahl);
15     System.out.println("The average is: " + average);
16
17 }
18
19
20 public static int findAverage(int sum, int anzahl) {
21
22     int average = sum * anzahl; // This should be sum / 3 to get the correct average
23     return average;
24
25 }
26
27
28
29 }
30
```



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

Mit step over:

```
DebuggerBeispiel.java X
1
2 public class DebuggerBeispiel {
3
4
5 public static void main(String[] args) {
6
7     int num1 = 10;
8     int num2 = 20;
9     int num3 = 30;
10
11     // Calculate the average (intentional bug here)
12     int sum = num1 + num2 + num3;
13     int anzahl = 3;
14     int average = findAverage(sum, anzahl);
15     System.out.println("The average is: " + average);
16
17 }
18
19
20 public static int findAverage(int sum, int anzahl) {
21
22     int average = sum * anzahl; // This should be sum / 3 to get the correct average
23     return average;
24
25 }
26
27
28
29 }
30
```

- 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 'Variables' window in an IDE. The window has a title bar with 'Variables' and a close button. Below the title bar are tabs for 'Breakpoints' and 'Expressions'. The main area is a table with two columns: 'Name' and 'Value'. The table contains the following data:

Name	Value
no method return value	
args	String[0] (id=19)
num1	10
num2	20
num3	30
sum	60
anzahl	3

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

The screenshot shows an IDE window titled "DebuggerBeispiel.java". The code editor contains the following Java code:

```
1 public class DebuggerBeispiel {
2
3
4
5 public static void main(String[] args) {
6
7     int num1 = 10;
8     int num2 = 20;
9     int num3 = 30;
10
11     // Calculate the average (intentional bug here)
12     int sum = num1 + num2 + num3;
13     int anzahl = 3;
14     int average = findAverage(sum, anzahl);
15     System.out.println("The average is: " + average);
16
17 }
18
19
20 public static int findAverage(int sum, int anzahl) {
21
22     int average = sum * anzahl; // This should be sum / anzahl to get the correct average
23     return average;
24
25 }
26
27
28
29 }
30
```

The variables window on the right shows the following data:

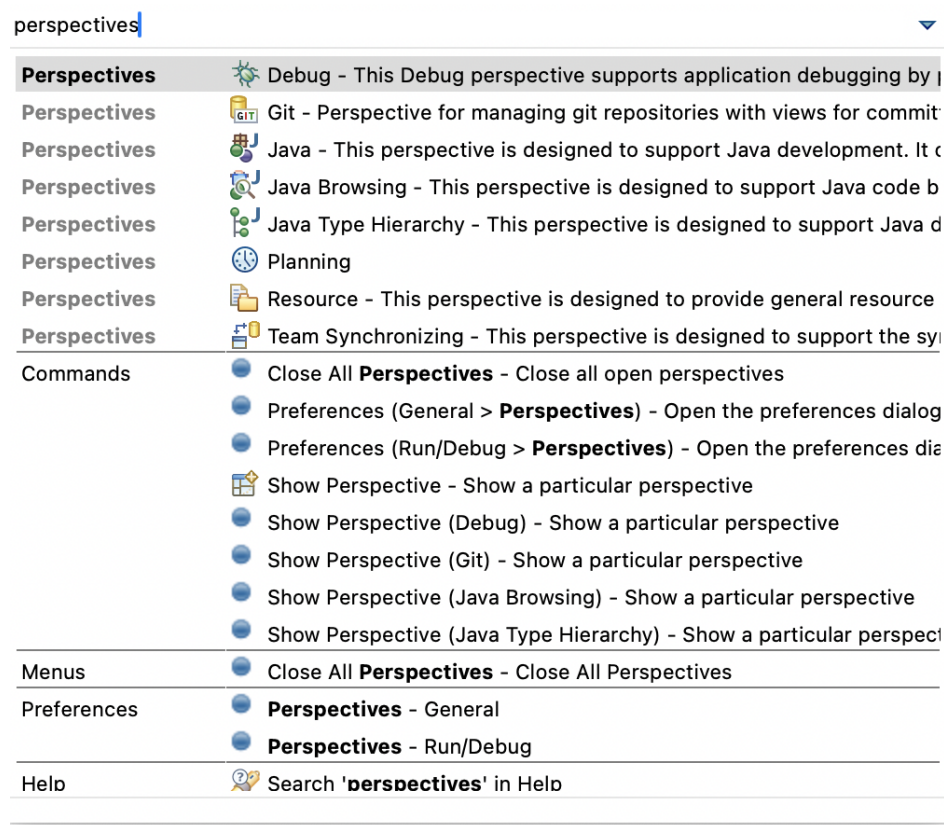
Name	Value
no method return value	
sum	60
anzahl	3
average	180

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

Anhand der Spalte "Value" erkennt man, dass **average** durch **(sum * anzahl)** berechnet wird, anstatt durch **(sum / anzahl)**.

Falls nicht automatisch zur Debugger Perspective gewechselt wird:

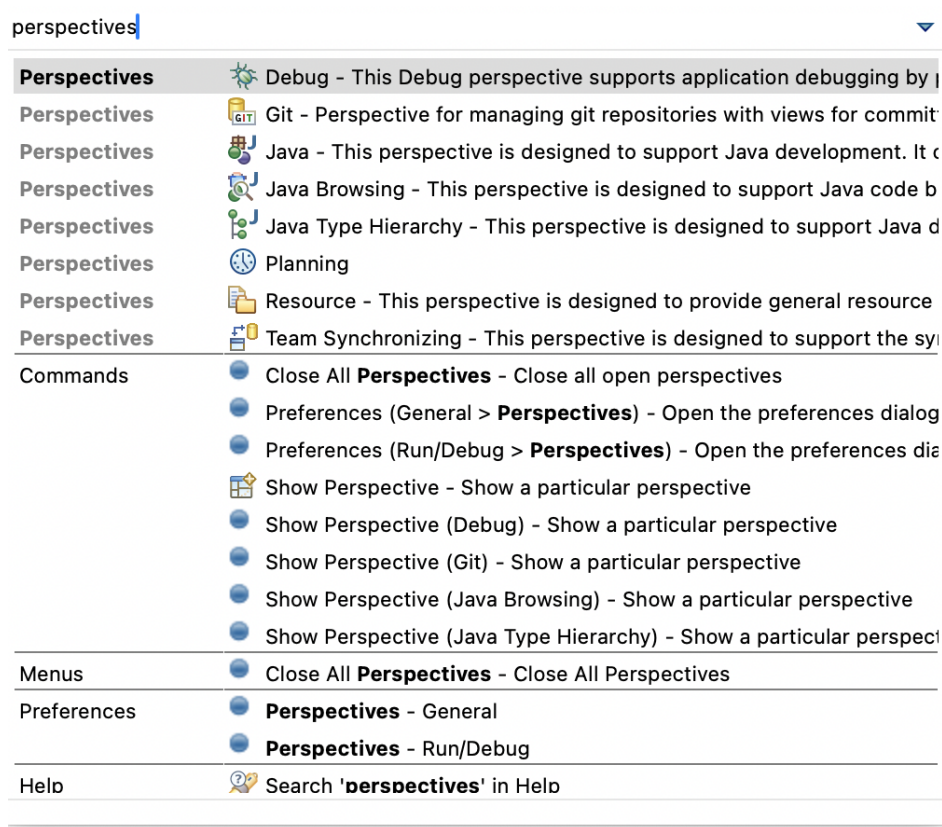
Rechts oben in der Searchbox “Perspectives” suchen:



Debug
Perspective
wählen

Zurück zur Java Perspective wechseln:

Rechts oben in der Searchbox “Perspectives” suchen:



The screenshot shows the Eclipse IDE search interface with the search term 'perspectives' entered. The search results are categorized into Perspectives, Commands, Menus, Preferences, and Help. The 'Java' perspective is highlighted with a blue arrow pointing from the text 'Java Perspective auswählen' on the right.


Category	Item
Perspectives	Debug - This Debug perspective supports application debugging by i
Perspectives	Git - Perspective for managing git repositories with views for commit
Perspectives	Java - This perspective is designed to support Java development. It c
Perspectives	Java Browsing - This perspective is designed to support Java code b
Perspectives	Java Type Hierarchy - This perspective is designed to support Java d
Perspectives	Planning
Perspectives	Resource - This perspective is designed to provide general resource
Perspectives	Team Synchronizing - This perspective is designed to support the sy
Commands	Close All Perspectives - Close all open perspectives
Commands	Preferences (General > Perspectives) - Open the preferences dialog
Commands	Preferences (Run/Debug > Perspectives) - Open the preferences dia
Commands	Show Perspective - Show a particular perspective
Commands	Show Perspective (Debug) - Show a particular perspective
Commands	Show Perspective (Git) - Show a particular perspective
Commands	Show Perspective (Java Browsing) - Show a particular perspective
Commands	Show Perspective (Java Type Hierarchy) - Show a particular perspect
Menus	Close All Perspectives - Close All Perspectives
Preferences	Perspectives - General
Preferences	Perspectives - Run/Debug
Help	Search ' perspectives ' in Help

Java
Perspective
auswählen

Git Merge Conflicts

Pushed to aroesti-test - origin



 master → master [rejected - non-fast-forward]

[rejected - non-fast-forward]

Message Details

Repository

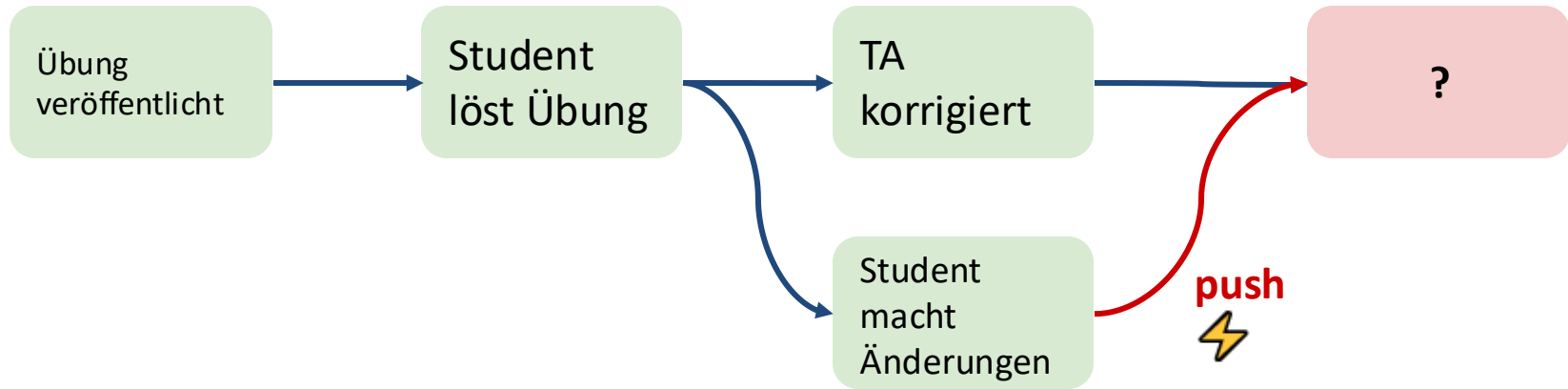
<https://gitlab.inf.ethz.ch/COURSE-EPROG2018/playground/aroesti-test.git>

Configure...

Close

Git - Merge Conflicts

Student löst Übung gleichzeitig während TA korrigiert
→ Beim Versuch zu pushen, Konflikt!



Einfache Lösung

1. Im Fenster Git Repositories, Rechtsklick auf das Repository, dann “git pull --rebase”.
2. Zwei mögliche Ergebnisse:
 - a. Result: “**Merged**”. Das Problem ist gelöst. Mit Rechtsklick auf Repo, dann “Push to Upstream” können die eigenen Änderungen hochgeladen werden.
 - b. Result: “**Conflict**”. TA und Student haben dieselbe(n) Datei(en) bearbeitet. Die rot markierten Dateien müssen manuell durchgegangen werden. Konflikt-Zeilen sind mit “<<<<”, “====” und “>>>>” markiert. Nach dem Bearbeiten erneut committen. Dann ist ein Push möglich.

Pull Result for aroesti-test

Fetch Result

- ▼  master : origin/master [5661974..55e203a] (1)
 - ▶  5661974c: Korrektur (Andre Roesti on 2018-10-13 20:00:53)



Update Result

Result Merged

New HEAD Merge branch 'master' of https://gitlab.inf.ethz.ch/COURSE-EPROG2018/playground/aroesti-test.git [8d16702]

Merge input

-  ef91996e: Übung gelöst (Andre Roesti on 2018-10-13 20:01:29)
-  5661974c: Korrektur (Andre Roesti on 2018-10-13 20:00:53)

Close

```
Repository 'aro'
1 <<<<<< HEAD
2 Ich bearbeite dieselbe Datei wie gerade auch der TA, ob das wohl
3 =====
4 Beide bearbeiten dieselbe Datei.
5 >>>>>> branch 'master' of https://gitlab.inf.ethz.ch/COURSE-EPP
6
```

```
Repository 'aro'
1 <<<<<< HEAD
2 Ich bearbeite dieselbe Datei
3 =====
4 Beide bearbeiten dieselbe Datei
5 >>>>>> branch 'master' of
6
```

- aroesti-test [aroesti-test]
- etwas_anderes.txt
- etwas.txt
- Übungsstunde-3

Update Result

Result Conflicting

Merge input

- f25452a0: Übung lösen (Andre Roesti on 2018-10-13)
- e78660cb: Korrektur (Andre Roesti on 2018-10-13)

Git Repositorien

- aroesti-test
 - Branches
 - Local
 - master f25452a Übung lösen
 - Remote Tracking
 - origin/master e78660c Korrektur
 - Tags
 - References
 - Remotes
 - Working Tree - /Users/androesti/git
 - .git
 - .project
 - etwas.txt

Update Result

Result Conflicting

Merge input

- f25452a0: Übung lösen
- e78660cb: Korrektur (Ar)

Repository state: Conflicts

Author:

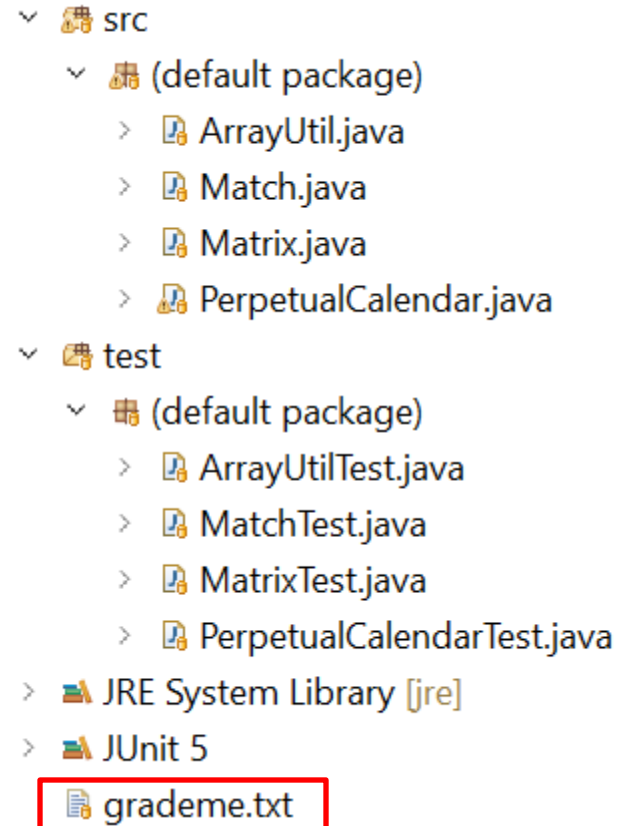
Committer:

Commit and Push... Commit

Feedback

Feedback

- Ihr sagt den TAs ab u04 wo ihr Feedback haben möchtet.
- Dazu schreibt ihr die Aufgaben in das grademe.txt file und pushed dieses mit eurer Abgabe.



Value vs Reference

- Variablen enthalten entweder einen **Wert** (Primitive Type) oder eine **Referenz** (Object oder Object[]).
- Variablen enthalten **nie** Objekte.
- Referenzen zeigen auf das Objekt im Speicher.
- Referenzen befolgen auch Value Semantics.
- Nur die **Objekte** selbst ermöglichen Reference Semantics.

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) {

    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) {
    // 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) {  
    // Zwischenspeichern von y  
    int altY = y;  
    y = x % y;  
    x = altY;  
}  
System.out.println(y);
```

Negation der Bedingung in 1.

Aufgabe 3: Zahlenerkennung

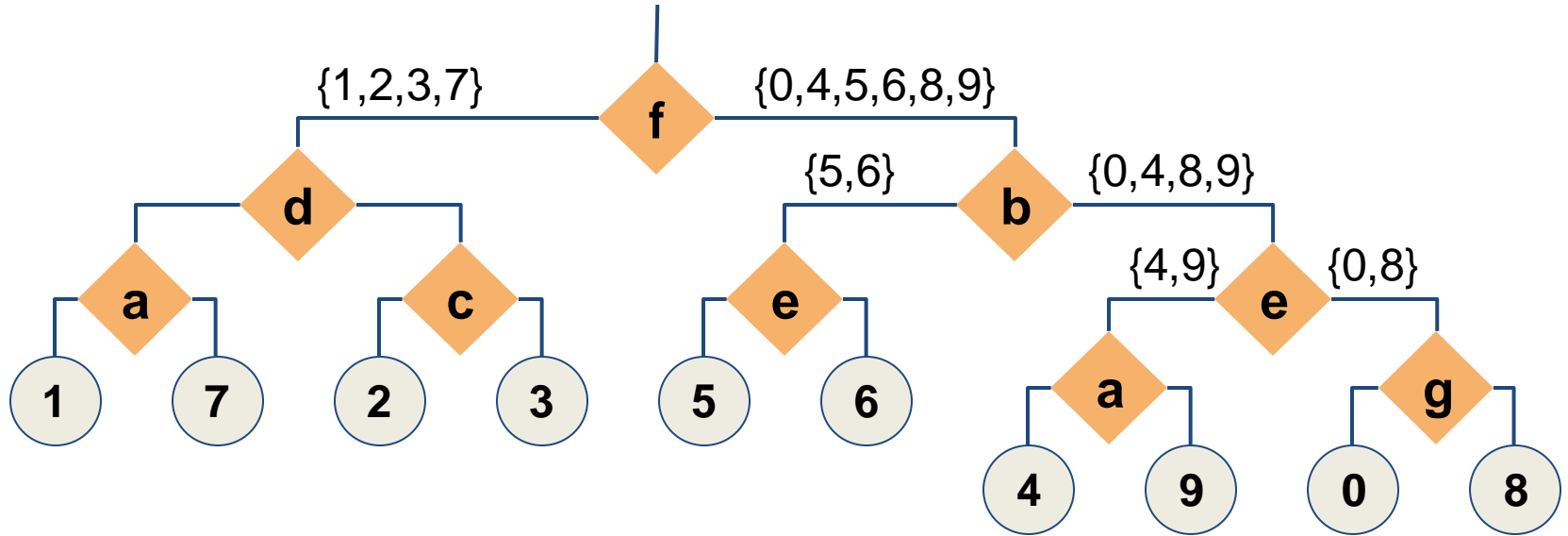
Für diese Aufgabe verwenden wir einen String um die erleuchtenden Segmente einer **Sieben-segmentanzeige** zu kodieren. Die Segmente sind, wie im Bild gezeigt, von a bis g nummeriert. Die Kodierung einer möglichen Anzeige ist ein String, in welchem der Buchstabe 'x' genau dann vorkommt, wenn das 'x'te 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) {  
    return a == 7 || b == 7 || a+b == 7 || a-b == 7 || b-a == 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) {  
    return (z + 2)%12 <= 4;  
}
```


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 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/2024/exercises/additional/
bonusaufgaben_regeln.pdf](https://lec.inf.ethz.ch/infk/eprog/2024/exercises/additional/bonusaufgaben_regeln.pdf)

Kahoot

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