FITPED

Java fundamentals

Ján Skalka
Ľubomír Benko
Jaromír Landa
Mariusz Boryczka
Juan Carlos Rodríguez-del-Pino
José Daniel González-Domínguez

www.fitped.eu

2021

Co-funded by the Erasmus+ Programme of the European Union

Work-Based Learning in Future IT Professionals Education
(Grant. no. 2018-1-SK01-KA203-046382)

Java Fundamentals

Published on

November 2021

Authors

Ján Skalka | Constantine the Philosopher University in Nitra, Slovakia

Ľubomír Benko | Constantine the Philosopher University in Nitra, Slovakia

Jaromír Landa | Mendel University in Brno, Czech Republic

Mariusz Boryczka | University of Silesia in Katowice, Poland

Juan Carlos Rodríguez-del-Pino | University of Las Palmas de Gran Canaria, Spain

José Daniel González-Domínguez | University of Las Palmas de Gran Canaria, Spain

Reviewers

Anna Stolińska | Pedagogical University of Cracow, Poland

Peter Švec | Teacher.sk, Slovakia

Eugenia Smyrnova-Trybulska | University of Silesia in Katowice, Poland

Piet Kommers | Helix5, Netherland

Graphics

Ľubomír Benko | Constantine the Philosopher University in Nitra, Slovakia

David Sabol | Constantine the Philosopher University in Nitra, Slovakia

Erasmus+ FITPED

Work-Based Learning in Future IT Professionals Education

Project 2018-1-SK01-KA203-046382

Co-funded by the
Erasmus+ Programme
of the European Union

The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Licence (licence type: Attribution-Non-commercial-No Derivative Works) and may be used by third parties as long as licensing conditions are observed. Any materials published under the terms of a CC Licence are clearly identified as such.

All trademarks and brand names mentioned in this publication and all trademarks and brand names mentioned that may be the intellectual property of third parties are unconditionally subject to the provisions contained within the relevant law governing trademarks and other related signs. The mere mention of a trademark or brand name does not imply that such a trademark or brand name is not protected by the rights of third parties.

© 2021 Constantine the Philosopher University in Nitra

ISBN 978-80-558-1639-5

Table of Contents

1 The Java Language 6
1.1 Programming languages 7
1.2 Java 11
2 Output Commands 15
2.1 Outputs 16
2.2 Outputs (programs) 21
3 Variables 23
3.1 Variables 24
3.2 Variables operations 30
4 Data Input 32
4.1 Inputs 33
4.2 Inputs (programs) 38
5 Conditions 42
5.1 Command if 43
5.2 Comparison 47
5.3 If (programs) 52
6 Loops 54
6.1 Basic commands 55
6.2 More about Loops 58
6.3 For cycle (programs) 62
6.4 Loops with conditions 66
6.5 While loops (programs) 69
7 Numeric Data Types 71
7.1 Integer 72
7.2 Incremental and decremental operator 73
7.3 Number types 78
7.4 Real numbers 80
7.5 Number types (programs I.) 84
7.6 Number types (programs II.) 87
8 Other Data Types 90
8.1 Logical type and logical expression 91
8.2 Compound conditions 95

8.3 Char 99
8.4 Other data types (programs) 103
9 String I. 105
9.1 About String 106
9.2 Data type selection 112
9.3 Functions to work with string 114
9.4 Basic strings (programs) 119
9.5 String and number 120
10 String II. 123
10.1 Working with strings 124
10.2 More functions 129
10.3 Numbers in strings (programs) 133
10.4 Working with text (programs) 135
10.5 Advanced operations with text (programs) 138
11 Nested Loops and Effectivity 141
11.1 Nested loops 142
11.2 Simple problems (programs) 145
11.3 Advanced problems (programs) 149
11.4 Repair programs (programs) 151
12 Multiple Conditionals 159
12.1 Command switch 160
12.2 Switch (programs) 166
13 Exceptions 170
13.1 Exceptions and the treatment 171
13.2 Exceptions (programs) 176
14 Arrays 179
14.1 Basic terms 180
14.2 Reading data into array 184
14.3 Constants and random numbers 188
14.4 Random numbers (programs) 191
14.5 Simple arrays (programs) 192
14.6 Arrayless List (programs) 195
15 Array Processing 196
15.1 Array operations 197
15.2 Arrays operations (programs) 204
15.3 Arrays under scrutiny (programs) 209

15.4 Array sort (programs) 213
16 2D Arrays 216
16.1 Matrix 217
16.2 Working with matrix 223
16.3 Matrix (programs) 230
16.4 Table (programs) 235
17 Files 238
17.1 Streams 239
17.2 Text File 242
17.3 Working with files 250
17.4 Files processing (programs) 257
18 Exercises 264
18.1 Advanced exercises (programs) 265
18.2 List of tasks 274

The Java Language

Chapter 1

1.1 Programming languages

🕮 1.1.1

When writing an algorithm, we can also use the common language, but we encounter several problems:

- the common language has about 110 thousand words in the case of Slovak, and in the case of English even about 800 thousand words, which is very much for the interpretation of commands
- the speech is commonly used in various phrases (broad lace), homonymous (head - cabbage, screws, human, head as a leader, etc.), synonyms etc. the computer can not understand
- it is also natural to get new ones and to eject old words

For that reason, when creating an algorithmic language, much of the common speech is deleted, and only commands that are meaningful for a given situation or given a use (e.g. write, read, add, subtract, etc.)

1.1.2

Is it true? We can use the common language used for communication between people as an algorithmic language for non-thinking devices (computers).

- True
- False

🕮 1.1.3

Commands that are intended for a non-thinking device (computer) are performed by a processor. At present, only a few types of processors (used in a number of different devices) are produced, but each has its own language to translate algorithm commands. We designate this language as a machine code, and it is very far from the usual algorithmic language command. We call it a lower-level language.

It can only perform duplicate data from/to the memory location, add the value from the memory location to the processed content, and so on. All commands are additionally represented by numbers.

```
Lister - [c:\game\haxm_check.exe]
Súbor  Upraviť  Možnosti  Kódovanie  Pomocník                                  1 %
00000000: 4D 5A 90 00 03 00 00 00|04 00 00 00 FF FF 00 00  | MZ..........¨¨..
00000010: B8 00 00 00 00 00 00 00|40 00 00 00 00 00 00 00  | ¸.......@.......
00000020: 00 00 00 00 00 00 00 00|00 00 00 00 00 00 00 00  | ................
00000030: 00 00 00 00 00 00 00 00|00 00 00 00 08 01 00 00  | ................
00000040: 0E 1F BA 0E 00 B4 09 CD|21 B8 01 4C CD 21 54 68  | ..º..´.Í!¸.LÍ!Th
00000050: 69 73 20 70 72 6F 67 72|61 6D 20 63 61 6E 6E 6F  | is program canno
00000060: 74 20 62 65 20 72 75 6E|20 69 6E 20 44 4F 53 20  | t be run in DOS
00000070: 6D 6F 64 65 2E 0D 0D 0A|24 00 00 00 00 00 00 00  | mode....$.......
00000080: 9D 8F 54 33 D9 EE 3A 60|D9 EE 3A 60 D9 EE 3A 60  | ťŹT3Ůî:`Ůî:`Ůî:`
00000090: 6D 72 CB 60 D0 EE 3A 60|6D 72 C9 60 AE EE 3A 60  | mrË`Ðî:`mrÉ`®î:`
000000A0: 6D 72 C8 60 C1 EE 3A 60|E2 B0 39 61 C8 EE 3A 60  | mrČ`Áî:`â°9aČî:`
000000B0: E2 B0 3F 61 C4 EE 3A 60|E2 B0 3E 61 C8 EE 3A 60  | â°?aÄî:`â°>aČî:`
000000C0: 04 11 F1 60 DA EE 3A 60|D9 EE 3B 60 8D EE 3A 60  | ..ń`Úî:`Ůî;`Ťî:`
000000D0: 4E B0 33 61 D8 EE 3A 60|4B B0 C5 60 D8 EE 3A 60  | N°3aŘî:`K°Ĺ`Řî:`
000000E0: D9 EE AD 60 D8 EE 3A 60|4E B0 38 61 D8 EE 3A 60  | Ůî-`Řî:`N°8aŘî:`
000000F0: 52 69 63 68 D9 EE 3A 60|00 00 00 00 00 00 00 00  | RichŮî:`........
00000100: 00 00 00 00 00 00 00 00|50 45 00 00 4C 01 06 00  | ........PE..L...
00000110: 91 56 3B 5A 00 00 00 00|00 00 00 00 E0 00 02 01  | ‘V;Z........ŕ...
00000120: 0B 01 0E 00 00 FA 00 00|00 90 00 00 00 00 00 00  | .....ú..........
00000130: B7 13 00 00 00 10 00 00|00 10 01 00 00 00 40 00  | ·.............@.
00000140: 00 10 00 00 00 02 00 00|06 00 00 00 00 00 00 00  | ................
00000150: 06 00 00 00 00 00 00 00|00 D0 01 00 00 04 00 00  | .........Ð......
00000160: DE 44 02 00 03 00 40 81|00 00 10 00 00 10 00 00  | ÞD....@.........
00000170: 00 00 10 00 00 10 00 00|00 00 00 00 10 00 00 00  | ................
00000180: 00 00 00 00 00 00 00 00|7C 6C 01 00 28 00 00 00  | ........|l..(...
00000190: 00 B0 01 00 10 06 00 00|00 00 00 00 00 00 00 00  | .°..............
000001A0: 00 84 01 00 B8 26 00 00|00 C0 01 00 C8 0E 00 00  | .„..¸&...Ŕ..Č...
000001B0: C0 64 01 00 70 00 00 00|00 00 00 00 00 00 00 00  | Ŕd..p...........
000001C0: 00 00 00 00 00 00 00 00|00 00 00 00 00 00 00 00  | ................
000001D0: 30 65 01 00 40 00 00 00|00 00 00 00 00 00 00 00  | 0e..@...........
000001E0: 00 10 01 00 08 01 00 00|00 00 00 00 00 00 00 00  | ................
000001F0: 00 00 00 00 00 00 00 00|00 00 00 00 00 00 00 00  | ................
00000200: 2E 74 65 78 74 00 00 00|8B F9 00 00 00 10 00 00  | .text...<ů......
```

Rewriting our commands into this form would be very lengthy, impractical and, in many, mentally, too demanding.

1.1.4

How is the native language of the processor called? It is the language in which every command that is intended to be made by a foolish device must be translated.

1.1.5

A compromise between the comprehensibility of the common language and the speed of the machine language is the higher programming languages. **The programming language** is a selection of common language (algorithmic) commands that are used in the exact prescribed form,

```
print("Hello") - writes Hello
```

or

```
input.nextInt() – returns next number from input
```

The programming language is a **translator** that, by following the set rules, can translate these commands into a machine code that is already easy to process because it understands it.

1.1.6

The programming language is:

- the intermediate language between the human language and the machine code
- a language that is translated into a machine code executed by the processor using a translator
- selection common language commands that are used in a specified form
- a language designed to accurately express the processor - processor language
- a lower-level language that is better understable by people than a computer

1.1.7

There are currently hundreds of programming languages, each of which has its own translator. There are two ways to translate commands:

- Before running - after writing the program, but before it runs. Commands in the programming language translate into a separate file (application, exe-file) and the original source code is no longer needed. If there were any bugs in the program, they had to be repaired before the transfer. This translator is referred to as a **compiler**.
- Continuously - the commands are translated into the source code sequentially during the execution of the program. In order for this program to run, both the program and the translator must be available in the system. The program can also run if there are any errors - if the translator encounters them, they notify the user. We designate such an interpreter as an **interpreter**.

The best-known compiled languages are C, C++, Pascal, Fortran.

The most well-known interpreted languages are Python, PHP, JavaScript.

Some languages are first compiled into an intermediate language and the intermediate language is then interpreted, for example, Java and C#.

1.1.8

Which statements are true?

- Compiler is a translator that translates the program only once to create a bootable application.
- An interpreter is a translator that only translates the program once and produces an executable application.
- An interpreter is a translator that translates the program every time a command is executed after a command.
- The compiler is a translator that translates the program every time a command is executed after a command.

1.1.9

The advantage of compiled programs is the **speed** - the program is once and permanently translated into machine code, and after execution, it only executes commands in the native language of the processor.

The advantage of interpreted languages is **security**. While the compiled program will always be the same after interpreting, interpreters are constantly updated - if a feature later found to be disruptive to the security of the device may be repaired, replaced, or simply blocked in the new version of the interpreter. The new version of the interpreter is usually updated automatically on the computer to prevent security risks.

1.1.10

Which statements are true?

- A program written in a compiled language runs faster than the same program written in the interpreted language.
- A program written in a interpreted language runs faster than the same program written in the interpreted language.
- The advantage of interpreted languages is the security achieved for the same code by updating the translators they need to perform.
- The advantage of compiled languages is the security achieved for the same code by automatic compilation when an error is detected (e.g., in an operating system).

1.2 Java

🕮 1.2.1

Java is a high-level programming language created by Sun Microsystems company in the year 1995 (later it was bought by Oracle). Nowadays it is licensed with the GNU/GPL license and can be used to develop any type of application.

It is a fast, secure and reliable programming language that is currently implemented in several billion different devices. Java applications are developed for mobile devices, web servers, personal computers or notebooks, game consoles and many other devices.

The application can be created using the following:

- text editor for writing the code
- compiler to check and generate the intermediate code that can be understood by the interpreter
- an interpreter that ensures the implementation of the code

All three parts are usually part of the development environment and are interconnected. The most common development environments are NetBeans, Eclipse, IntelliJ IDEA and BlueJ.

We do not need a development environment to run the compiled code, but a Java Runtime Environment **JRE** (Java Runtime Environment) is needed that is available for all systems and devices.

1.2.2

The Java programming language is deployed on more than a billion different devices.

- True
- False

1.2.3

Java is an object-oriented programming language. That means that the language supplies features to create objects as the basic elements of the program's operation.

The way of working and the rules for implementing the code are described by classes (class). Each class has to have its name and commands that are to run first and have to be listed in the part (method) that is called main.

Example:

```
class Dog {        // definition that of the class called Dog
  public static void main(String[ ] args) { // description of
the main method has usually this form
    System.out.println("Wof, Wof!");        // command that
will write the text in the quotes
  }
}
```

To define a class, you must use the keyword **class** followed by the name of the class. The individual parts of the code are enclosed in brackets {} and commands are separated by a semicolon (;).

1.2.4

What is the keyword used to define a class?

1.2.5

We used "//" in the code and tried to explain the code behind them.

```
class Dog { // definition of the class with the name Dog
  public static void main(String[ ] args) { // description of
the main method has usually this form
    System.out.println("Wof, Wof!"); // command that will
write the text in the quotes
  }
}
```

The characters that are placed after "//" are **till the end of the row** ignored when running the code and are used as comments that can help to describe the code to another programmer or the author when he/she later returns to the code.

1.2.6

Fill in the text in the code that makes a line comment from the part to the right of the embedded answer.

```
class Cat {  _____ this is a class representing a cat
  public static void main(String[ ] args) {
    System.out.println("Miau, miau!");
  }
}
```

1.2.7

Simple comments are sometimes inadequate, and you need to make sure that the code is not executed (detailed comment, clear description, descriptive text about the program).

In this case, the pair of "/*" and the end "*/" are used as the beginning of the comment.

```
/* demo example
Author: Jan Skalka
Date: 18.7.2019
Description: The programm writes the text Wof, wof! and it
ends
*/
class Dog {
  public static void main(String[ ] args) {
    System.out.println("Wof, wof!");
  }
}
```

All text between these tags is ignored by the compiler.

1.2.8

Add text to the code that will create a text-based commentary between the text.

```
class Cat {
_____ this is a class representing a cat
   it will create code
```

```
   that will do the miau after execution _____
  public static void main(String[ ] args) {
    System.out.println("Miau, miau!");
  }
}
```

1.2.9

Characters {} limit the logical parts of the program.

```
class Dog {
  public static void main(String[ ] args) {
    System.out.println("Wof, wof!");
  }
}
```

In this case, they define the body of the class and the body of the main method. We will meet with them in many places.

1.2.10

Add text to the code that borders the blocks defined in the program.

```
class Cat _____
  public static void main(String[ ] args) _____
    System.out.println("Miau, miau!");
  _____
_____
```

Output Commands

Chapter 2

2.1 Outputs

📖 2.1.1

The Dog program contains only one command that executes a certain operation:

```
class Dog {
  public static void main(String[ ] args) {
    System.out.println("Wof, wof!");
  }
}
```

The command

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

serves to write the text that is placed between the quotes. Although at first glance it looks complicated, it only copies the structure of Java:

- *The system* is a library/class containing the basic commands
- *out* is a channel intended to output the data from the program
- *println* will ensure that the content listed in brackets is printed

The output command allows you to write virtually any text or number on the console/screen.

📝 2.1.2

Complete the command to output the text:

```
class Cat {
  public static void main(String[ ] args) {
    System.out._____("Miau!");      // output Miau!
  }
}
```

📖 2.1.3

If we want, we can also use more of the same commands in a row. It is used to write each command to a new line, but it is not an obligation. In order for the

system to determine where one command ends and the other begins has used a semicolon (;).

```
class Dog {
  public static void main(String[ ] args) {
    System.out.println("Wof!");     // outputs Wof!
    System.out.println("Grrr!");    // outputs Grrr!
  }
}
```

If the translator does not find the semicolon in the right place, the program can not be started.

Remember: Commands are separated by a semicolon.

2.1.4

Fill in the commands to output the text:

```
class Cat {
  public static void main(String[ ] args) {
    System.out.println("Miau!")_____    // output Miau!
    System.out.println("Krrss!")_____   // output Krrss!
  }
}
```

2.1.5

The command

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

prints the text in quotes and "insert a line feed" - ensures that the next text is placed in a new line.

If we want to put the text into a row with multiple commands, we use the command

```
System.out.print("text");
```

which writes the text in brackets and does not insert a line feed - the next text will continue to the current line at the next character position.

Program:

```
class OneRow{
```

```
  public static void main(String[ ] args) {
    System.out.print("Hello!");  // outputs Hello!
    System.out.print("I am a computer."); // outputs Hello!I
am a computer.
  }
}
```

will write the text only in one row.

Required texts are written immediately behind each other - **it does not leave a gap between them**.

2.1.6

Complete the code that way to print out the following text:

```
10 + 5 = 15
15 + 6 = 21
```

```
class Calculation {
  public static void main(String[ ] args) {
    System.out._____("10 + ");
    System.out._____("5 ");
    System.out._____("= 15");
    System.out._____("15 + ");
    System.out._____("6 ");
    System.out.print("= 21");
  }
}
```

- println
- print
- out
- println
- print
- println
- print
- print
- print
- out
- print

2.1.7

The following code:

```
class OneRow {
  public static void main(String[ ] args) {
    System.out.print("Hi,");
    System.out.print("I");
    System.out.print("am");
    System.out.print("John.");
  }
}
```

outputs the text:

```
Hi,IamJohn.
```

If we want to have gaps between the words, we need to put them in the quotes too - the system does not understand the language or can not estimate our intentions. We can insert a space at the beginning or at the end of the text in quotes - but usually, the spaces are placed at the end of the text.

The program:

```
class OneRow {
  public static void main(String[ ] args) {
    System.out.print("Hi, ");
    System.out.print("I");
    System.out.print(" am "); // we can put the space to the
beginning, but we usually do not
    System.out.print("John.");
  }
}
```

outputs the text:

```
Hi, I am John.
```

with spaces in the expected places.

2.1.8

Fill in the characters so it looks like the following:

```
Hello, I am not programmer.
```

Use the underscore instead of space - "_".

```
class OneRow {
  public static void main(String[ ] args) {
    System.out.print("Hel_____");
    System.out.print("I_____");
    System.out.print("a_____");
    System.out.print("not");
    System.out.print("_____rogrammer");
  }
}
```

2.1.9

If we compile the output using multiple print commands, we can optionally combine the *print* and *println* where we have to consider that the output after the *println* starts in a new row.

The command

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

without parameters moves the cursor to the new line.

The following program inserts a blank line between the two lines of text written with the *println* command

```
class MoreRows {
  public static void main(String[ ] args) {
    System.out.println("Hello");  // writes the text and moves
to a new line
    System.out.println(); // moves the cursor to another new
line
    System.out.println("before this line is one line
omitted"); //  writes the text and moves to a new line
  }
}
```

and the result is following:

```
Hello

before this line is one line omitted
```

2.1.10

Fill in the commands to get the following output:

```
WARNING!
Winter at the polar circle is...

... long

... and cold.
```

Instead of space use the underscore - "_".

```
class MoreRows {
  public static void main(String[ ] args) {
    System.out._____("W_____");
    System.out._____("W_____");
    System.out._____("at_the_polar_circle");
    System.out._____("_____...");
    System.out._____();
    System.out._____("...long");
    System.out._____();
    System.out._____("...and_cold.");
  }
}
```

2.2 Outputs (programs)

2.2.1 Output of data

Print the following:

```
Joseph
Cucumber
```

2.2.2 Hello World!

Print on the screen "Hello World!".

Output:

```
Hello World!
```

2.2.3 Greetings

Print the following: Hello, Good day, Hi

Output:

```
Hello
Good day
Hi
```

2.2.4 Print in a row

Print out using the three commands System.out.print the following words:

```
"I "
"am "
"learning."
```

Make sure there are spaces between the words. You need to put them between the quotes.

Variables

Chapter 3

3.1 Variables

3.1.1

A command is used to write the text

```
System.out.print("mytext");
```

that is placed between the quotas and a command

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

that will write the text and move the cursor to a new line.

The command *print* can be used not only to write text but also to do some calculation, for example:

```
System.out.println(15+3);
```

this will do the calculation at first and then write the result.

The calculation is written without the quotes based on which the system knows that it should work with the content of the brackets as with numbers and that we do not want to write the content of the brackets in the same manner as it is.

The notation

```
System.out.println("15+3");
```

would result in the same result as the text in the quotas.

```
15+3
```

3.1.2

What will be the output of the program?

```
class Riddle {
  public static void main(String[ ] args) {
    System.out.println(15+3+10);
  }
}
```

🕮 3.1.3

The programming language is not limited to writing simple texts, but it can also make calculations. In order to store intermediate results or input values, **variables** are used.

A Variable is a memory location that serves to store and remember the values. We can change it during the program.

Each variable has:

- **data type**, which determines whether there is a text or number stored in it (for now are these two types of values enough),
- **title** (name), according to which we refer to the variable.

If we want to use the variable in the program, we have to state it in the code as follows.

```
class Variables {
  public static void main(String[ ] args) {
    int number;
    number = 10;
  }
}
```

In the first row, we define that we will use the variable *number* into which we will enter integer values.

In the second row (number = 10), we assign the value 10 to this variable.

The command that determines that the variable is to be assigned a value is "=". The "=" character is an **assignment** character and we usually do not say that we put the value into the variable, but we **assign** it to it.

Both steps can be fused to one and write as follows:

```
class Variables {
  public static void main(String[ ] args) {
    int number = 10;
  }
}
```

3.1.4

Fill in the code so that it is possible to assign the integer values to the variable *num*.

```
class Task {
  public static void main(String[ ] args) {
    ______ num;
    num  = 1000;
  }
}
```

3.1.5

Variables are usually used in calculations (expressions).

```
class Sum {
  public static void main(String[ ] args) {
    int a = 10;
    int b, sum;
    b = 20;
    sum = a + b;
  }
}
```

We **assign** the value or result of the calculation on the right side to the variable to the left of the assignment symbol (=), so its value **changes**.

The variables listed to the right of the assignment symbol **only give** their value for the calculation - their content **does not change** with this use.

So the result of calculation a + b, which is actually 10 + 20, is assigned into the variable *sum*. First, the entire calculation is performed at the right of the "=" and the result is assigned to the variable after its completion.

3.1.6

What will the variable *c* contain after the last command of the program?

```
class Riddle {
  public static void main(String[ ] args) {
    int a = 74;
    int b = 33;
    int c = a - b;
```

```
  }
}
```

3.1.7

We can write the content of the variable same as the text or expression:

```
class Output {
  public static void main(String[ ] args) {
    int s = 20;
    System.out.println(s);
  }
}
```

it will write **20** that is the content of the variable *s*.

Just as we could calculate the value with the output, we can do it with variables:

```
class Output {
  public static void main(String[ ] args) {
    int a = 5, b = 10;
    System.out.println(a + b);
  }
}
```

Note the declaration of variables *a* and *b* in the first line of the program - such an entry is allowed in the declaration.

In the output, the calculation is performed first - instead of the variables, the values they contain are put in - and the result obtained is written.

3.1.8

What will be the output of the following code?

```
class Riddle {
  public static void main(String[ ] args) {
    int a = 3, b = 5;
    System.out.println(a + b - 4);
  }
}
```

🕮 3.1.9

The data type defines besides the type of values we can insert into variables also the operations we can perform with them.

For numbers that are the operations of:

- addition (+)
- subtraction (-)
- multiplication (*)
- division (/)

If more than one operation is used in the calculation (commonly referred to as the **expression**), the standard policy applies: multiplication and division take precedence over addition and subtraction. If they are in brackets, the expression in them is evaluated first. If they have the same priority, they move from left to right.

For example:

```
class Calculation {
  public static void main(String[ ] args) {
    int a = 5, b = 7, c = 3;
    int result1 = a + b * c;
    int result2 = (a + b) * c;
    int result3 = 2*(a + 5) - c;
  }
}
```

In the first case is calculated *b * c* that means *7 * 3 = 21* and after that is added 5 – the result will be 26.

In the second case will be added *a + b* which means *5 + 7 = 12* and after that, it will be multiplied by 3 – the result will be 36.

In the third case will be variables and numbers combined which is often used. Firstly, is calculated *a + 5* which means *5 + 5 = 10*, then it will be multiplied by two *2 * 10 = 20* and subtracted by *c* which means 3. The result will be 17.

3.1.10

What will be the output of the following code?

```
class Riddle {
  public static void main(String[ ] args) {
    int a = 3, b = 5, c = 2;
    System.out.println(a + b * c - 3 + 2 * a);
  }
}
```

3.1.11

The variable may have virtually any name, but we have to follow the following rules:

- the name of the variable must begin with a letter, or "_" (or $, but it is not used)
- other characters may be letters, numbers, or underscores
- no spaces, special characters may be used in the name (for example +, - ,*, =, etc.),
- the name of the variable must not be either commands or keywords of the language (for example class, for etc.)

Variable names are *case-sensitive*, meaning that *Mom and mom* are two different variables because they differ in the size of the first character. Also, *ContentRectangle* and *contentRectangle* are different variables.

Variable names are currently used in the following convention - the first letter in the name is small and if the name of the variable consists of several words, each additional word starts with a capital letter.

For example:

```
contentRectangle, shortButLongVariable
```

If we need to use more words in the name, such writing is easier to read and better to decode the programmer.

3.1.12

Which of the following can be used as the variable name:

- winter
- summer3
- _father
- t__a
- _aa_
- IHAVE_It
- look-1
- woof-woof
- 2_test
- c=
- c?11
- C8 c1
- for

3.2 Variables operations

3.2.1

Evaluate which of the following statements contain the correct assignment of a variable:

- c = a * b
- _field = a / b
- a_b = c
- f = 4 + _b
- b + c = d – 4
- c = n _ c
- c d = 4 + a

3.2.2

What will be contained in the variable *c* after the expression is calculated:

*c = 7 + 3 * (8 - 2 +(6 * 9)) + 21 / 3*

3.2.3

What will be contained in the variable *c* after the calculation of the expression:

*c = 6 * a + b - 7 * 11 + (6-b) + a * 3*, where *a = 3, b = 5*

3.2.4

What will be contained in the variable *c* after the calculation of the expression:

*c = b * a + 6 - (a * 3) + a * (b - 7)*, where *a = 2, b = 4*

3.2.5 Calculation - numbers

Write a code that will write the result of the following math equation *5 + 48 + 3 * 11 - 96* using the print command.

3.2.6 Calculation - variables

Write a code that will:

- declare the variable *a*
- assign the variable *a* the value 10
- declare the variable *b*
- assign the variable *b* the value 17
- print the sum of these two variables

3.2.7 Calculation - into a variable

Write a code where you will:

- declare the variable *a* and assign the value 5
- declare the variable *b* and assign the value 4
- declare the variable *product* and calculate the product of variables *a* and *b*
- print the variable *product*.

Data Input

Chapter 4

4.1 Inputs

4.1.1

From programs, we usually expect to be able to solve the problem for different values.

If we have a program that can only sum values of 230 and 180, instead of writing, it is enough to use a calculator or just the knowledge of the elementary school.

The purpose of the program is to be able to perform the same operation or sequence of arbitrary operations. These must somehow get into the program, so we do not have to write them directly into the code. We designate them as input values and need to get them from the user and store them in variables to work with them.

Operations that ensure that values are retrieved are referred to as **input operations**. Initially, you enter the required values from the keyboard and read them through the program.

4.1.2

What are the operations that ensure that user values are loaded into the program?

- input
- output
- ongoing

4.1.3

Several tools are available for Java to read the input data from users. Most often, a *Scanner* is used that can read the input values separated by a space or placed in separate rows.

To use the *Scanner*, we need to do the following:

- import the library *java.util.Scanner*
- create a new scanner for standardized input channel (is marked as *System.in*)

Although the order of these commands looks complex, we usually write it in the same form, and it is enough for us to learn to remember it.

After creating the scanner, we can use it to load integer values into integer variables using the *nextInt()* command.

```
import java.util.Scanner; // import of scanner

public class Application {
  public static void main(String[ ] args) {
    // creation of the scanner with the name input over a
standard input channel
    Scanner input = new Scanner(System.in);
    int a; // declaration of variable a
    // using the created scanner with the name input we can
read the number value and assign it to the variable a
    a = input.nextInt();
    // we can write out the read variable
    System.out.println(a);
  }
}
```

4.1.4

Fill in the source code the commands so that it is possible to read the data from the input

```
_____ java.util.Scanner;
public class Application {
  public static void main(String[ ] args) {
    // creation of scanner with the name input over the
standardized input channel
    Scanner input = new Scanner(System._____);
    ... // reading the data
  }
}
```

4.1.5

Fill in the source code commands so that it is possible to read the input data:

```
_____ java.util._____;
public class Application {
  public static void main(String[] args) {
```

```
    // creation of scanner with the name input over the
standardized input channel
    _____ input = new _____(_____._____);
    ... // reading the data
  }
}
```

- Scanner
- in
- System
- Scanner
- import
- System
- Scanner

4.1.6

Fill in the source code the commands so that it will print the twice of the read value:

```
_____ java.util._____;
public class Application {
  public static void main(String[ ] args) {
    // creation of scanner with the name input over the
standardized input channel
    _____ input = new _____(System._____);
    // declaration of integer variable
    _____ a;
    a = input._____(); // reading the integer value from the
input
    System.out.println(a_____)_____// writing the two times of
the read value
  }
}
```

- Scanner
- int
- :
- import
- in
- Scanner
- ;
- nextInt
- .
- *2

- Scanner

4.1.7

Usually, one value is not enough in the program so we need more.

Using the *Scanner* we can read any number of inputs. One integer value can be assigned to an integer variable using the command

```
input.nextInt();
```

where the input is the *Scanner*.

If we want to load more values, we can use the command more times. User-entered values are assigned into variables in the order they are entered at the input.

The following program will read two values and print their sum:

```
import java.util.Scanner;
public class Application {
  public static void main(String[ ] args) {
    // creation of scanner with the name input over the
standardized input channel
    Scanner input = new Scanner(System.in);
    int a, b; // declaration of integer variables a, b
    // using the created scanner with the name input we can
read the input integer value and assign it to the variable a
    a = input.nextInt();
    // using the same scanner we read anothe integer value and
assign it to the variable b
    b = input.nextInt();
    // we print the sum of the values
    System.out.println(a+b);
  }
}
```

The input values can be put in one row delimited by a space (after the last one you have to press *Enter*)

or we can press *Enter* after each value

4.1.8

Fill in the commands so that the program will read the two input values into variables *a* and *b* and print out their product:

```
import java.util._____;
public class Application {
  public static void main(String[ ] args) {
    Scanner input = new Scanner(_____);
    _____ a,b; // declaration of two integer variables a, b
    a = input._____;
    b = _____nextInt();
    System.out.println(a_____); // we will print the product
of values
  }
}
```

- System.out
- input.
- int
- Scanner
- next()
- Scanner
- System.in
- nextInt()
- *a
- *b

4.1.9

Arrange the program rows to load the two integer values *a* and *b*, calculate the difference in the variable *c* and list it. First, you should input the value a then b.

- Scanner input = new Scanner(System.in);

- }
- public static void main(String[ ] args) {
- b = input.nextInt();
- System.out.println(c);
- a = input.nextInt();
- int c = a - b;
- public class Application {
- }
- import java.util.Scanner;
- int a, b;

4.2 Inputs (programs)

4.2.1 Loading the values

Read the given integer value, multiply it by 2 and print.

For example:

```
Input : 3
Output: 6
```

```
Input : 5
Output: 10
```

JavaApp.java

```
import java.util.Scanner;

public class JavaApp {

    public static void main(String[] args) {
        // we create scanner to read the input values
        Scanner input = new Scanner(System.in);

        // input - read the values

        // output - print in the required form


    }

}
```

4.2.2 Area of a square

Write a code that will for the given integer value calculate the area of a square.

```
Input : 3
Output: 9
```

```
Input : 8
Output: 64
```

4.2.3 Family allowances

Write a code that will for the given number of children calculate and print the sum of family allowances if for one child you get 30 EUR.

```
Input : 3
Output: 90
```

```
Input : 8
Output: 240
```

4.2.4 Cubic value

Write a code that will return the cubic value of the given integer value.

```
Input : 3
Output: 27
```

```
Input : 8
Output: 512
```

4.2.5 The sum of two numbers

Calculate and print the sum of two integer numbers from the input that is in one row and divided by a space.

```
Input : 5 7
Output: 12
```

```
Input : -1 5
Output: 4
```

4.2.6 The product of two numbers

Calculate and print the product of two input integer numbers that are given in one row delimited by a space.

```
Input : 5 7
Output: 35
```

```
Input : 1 5
Output: 5
```

4.2.7 The area and perimeter of a rectangle

The two given integer values (entered at one line input and separated by space) represent the sides of the rectangle. Calculate the area and perimeter of the rectangle. Write the result in the following form: area space perimeter.

```
Input : 1 2
Output: 2 6
```

```
Input : 30 5
Output: 150 70
```

4.2.8 The surface area and volume of the block

Calculate the surface area and volume of the block (where the sides are on the input in one row separated by spaces). Print the result in the following form: surface area space volume.

```
Input : 2 2 2
Output: 24 8
```

```
Input : 3 2 4
Output: 52 24
```

4.2.9 Aircraft range

Write a code that will return the range of the aircraft from given velocity (km per hour) and flight time in hours. The input contains the velocity and flight time. Print the calculated flight length in km.

```
Input : 987 5
Output: 4935
```

```
Input : 230 4
Output: 920
```

Conditions

Chapter 5

5.1 Command if

📖 5.1.1

A sequence of commands that is executed in the order in which it is written in the program is referred to as a sequence.

In this case, the non-minded device proceeds the individual orders, and when the command executes, proceeds to the next.

All the programs we have met so far have worked in the same way, for example:

```
import java.util.Scanner;
class Calculation {
  public static void main(String[ ] args) {
    Scanner input = new Scanner(System.in);
    int a, b;
    a = input.nextInt();
    b = input.nextInt();
    int result1 = a + b;
    int result2 = 2 * (a + b);
    System.out.println(result1);
    System.out.println(result2);
  }
}
```

📝 5.1.2

What is the sequence of commands that are executed in the order they are entered?

📖 5.1.3

However, most programs do not only contain simple sequences, but very often they need to decide how to proceed on the basis of the data processed. For example, the program will behave differently when entering staff age for a person under 18, over 18 or over 70 years.

The ability to decide and execute other commands by meeting or failing a condition is referred to as **branching**. It consists of a condition and orders to be executed if the condition is met and not met.

A conditional statement allows us to, for example, inform the user that he entered incorrect values, find out which number is bigger and so on.

5.1.4

How do you name the sequence of commands used to make the execution of the order conditional upon fulfilment of the condition, or can we ensure that one order is executed when the condition is met and otherwise not executed?

- sequence
- branching
- loop

5.1.5

The condition statements (or statement of branching) has the following form:

```
if (condition)
 statement1;
else
 statement2;
```

The condition is always written in brackets.

If the program encounters a condition while executing the commands, it evaluates its truth and chooses which commands it will execute, depending on the result.

If the condition is fulfilled, **command1** is executed, and if not, **command2** is executed. The part that is being executed when the condition is met is called the positive branch and the part that is executed if the condition is not met - the negative branch. A negative branch is given after the **else** statement.

After the execution of the commands in the condition, it continues sequentially by executing additional commands.

5.1.6

How are named the parts of conditional command that contain commands that are executed when a condition is met or not?

- branches
- conditions
- command brackets

5.1.7

A typical example of using a condition is to compare two numbers. In the task of writing a larger number, we compare the values stored in the variables and output the larger one.

```
class Example {
  public static void main(String[ ] args) {
    int a = 10, b = 15;
    if (a > b)
      System.out.println(a);
    else
     System.out.println(b);
  }
}
```

The condition contains a comparison of two values by the larger one (**a > b**).

If the condition is met (**a** is greater than **b**), the positive branch statement is executed - the value **a** is printed.

If the condition is not (**a** is not greater, but less than or equal to **b**), the statement specified in the **else** branch is executed - the value of **b** is printed.

5.1.8

Fill in the code to decide whether the number is "positive" or "negative".

```
import java.util.Scanner;

class Calculation {
  public static void main(String[ ] args) {
    Scanner input = new Scanner(System.in);
    int a = input.nextInt();
    if _____a > 0_____
      System.out.println("positive");
    _____
      System.out.println("negative");
  }
}
```

5.1.9

Using one command in the positive and one in the negative branch is rather exceptional, we usually need to use more commands. At that time, we write a list of commands between {} in the form of:

```
if (condition) {
  statement1;
  statement2;
} else {
  statement3;
  statement4;
}
```

For example:

```
class Age {
  public static void main(String[] args) {
    int age = 16;
    if (age < 18) {
      System.out.println("The person is less than 18 years
old");
      System.out.println(age + " year old cannot be
employed");
    } else {
      System.out.println("The person is more than 18 years
old");
      System.out.println(age + " year old can be employed");
    }
  }
}
```

5.1.10

Fill in the missing code:

```
import java.util.Scanner;

class Example {
  public static void main(String[ ] args) {
   Scanner input = new Scanner(System.in);
   int price = input.nextInt();
   _____ (price < 20) _____
     System.out.println("Price of the purchase is " + price);
```

```
      System.out.println("The goods are cheap.");
    _____ else _____
      System.out.println("Price of the purchase is " + price);
      System.out.println("The goods are expensive.");
    _____
  }
}
```

5.2 Comparison

5.2.1

So far, we have only used a larger or smaller character in the condition. However, we can also compare using other characters:

- **==** compares whether the values are equal, **a == b**
- **<=** compares whether the value on the left side is lower or equal than the value on the right side, **c <= 10**
- **>=** compares whether the value on the left side is higher or equal than the value on the right side, **c >= 10**
- **!=** compares whether the values are not equal, **a != b** – condition is met if the values are different

In the case of usage of symbols **<=** and **>=** has to be the order followed. The use of **=<** will result in an error.

5.2.2

Fill in the right comparison operators to the following code:

```
import java.util.Scanner;

class Example {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    int a = input.nextInt();
    if (a _____ 0) {
      System.out.println("A zero value was given.");
    } else {
      System.out.println("A non-zero value was given.");
   }
```

```
  }
}
```

5.2.3

A branch in which both the positive and negative commands are defined is referred to as **complete** but often we may encounter a situation where orders are only listed if the condition is met or not.

Such branching is referred to as **incomplete** but does not mean that it is inferior - very often it is not necessary to execute some orders when the condition is not met.

In the case of incomplete branching (i.e. if no command is to be executed in case of non-compliance), we simply omit the **else** branch.

5.2.4

Is it possible to omit the **else** command with all of its branches?

- Yes
- No

5.2.5

Fill in the right comparison operators to the following code:

```
import java.util.Scanner;

class Example {
  public static void main(String[ ] args) {
    Scanner input = new Scanner(System.in);
    int a = input.nextInt();
    if (a _____ 0)  System.out.println("zero value");
    if (a _____ 0)  System.out.println("positive number");
    if (a _____ 0)  System.out.println("negative number");
  }
}
```

- ==
- >
- >=
- <|=

- <

5.2.6

Fill in the code to find the division of two numbers, where if the second number is 0, will inform the user about division by zero.

```
import java.util.Scanner;

class Division {
  public static void main(String[ ] args) {
  Scanner input = new Scanner(System.in);
   int a = input.nextInt();
   int b = input.nextInt();
   if (_____)
     System.out.println("Division by zero")
   _____ { // otherwise calculate and print the division
     int divis = a _____ b;
     System.out.println(divis);
   }
}
```

5.2.7

Exchange of values between variables.

There are two numbers at the input that we load into variables **a** and **b**. Write a program to ensure that when it ends, the values will be exchanged with each other.

If we want to exchange the values store in variables **a** and **b,** the following statements:

```
int a = 10, b = 15;
a = b;
b = a;
```

would cause that the value of both variables would be 15.

How is this possible?

In the code, each operation has its place and order that is the reason why in the first step the values are 10 and 15 but after the assignment

```
a = b;
```

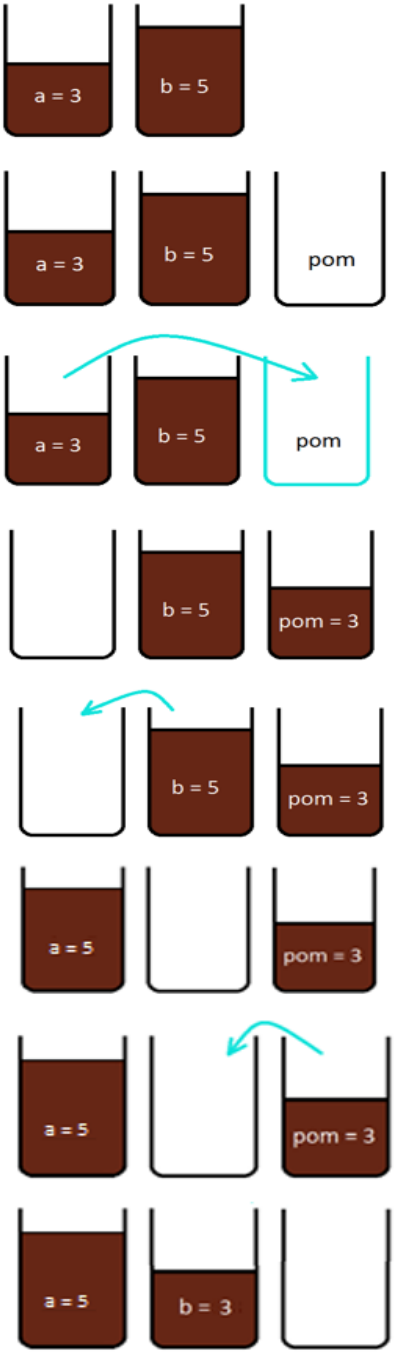
is in both variables the value 15 is in bot variables the value 15 and we lost the content of the variable **a** - it was overwritten by the content of variable **b**.

The value of variable **a,** where we want to save the new value, should be "saved" somewhere else.

We use a third (auxiliary) variable.

The algorithm will be then the following:

```
int a = 3, b = 5;
int pom;
pom = a;
a = b;
b = pom;
```



5.2.8

What values will be saved in variables **a** and **b** after the following commands?

```
int a = 10, b = 5;
a = a + b;
b = a + b;
a = b - a;
```

- a = 5, b = 20
- a = 20, b = 5
- a = 0, b = 15
- a = 15, b = 0
- a = 20, b = 15

5.3 If (programs)

5.3.1 Absolute value of a number

Write a code that prints the absolute value for the given integer.

```
Input : 0
Output: 0
```

```
Input : 3
Output: 3
```

```
Input :-8
Output: 8
```

5.3.2 Comparison of two numbers

Write a code that prints the larger number of the two given numbers. In the case of equal numbers print "Numbers are the same".

```
Input : 3 2
Output: 3
```

```
Input : 2 8
Output: 8
```

```
Input : 2 2
Output: numbers are the same
```

5.3.3 Maximum of three numbers

Write the code that prints the largest of the three entered numbers. If all three numbers are the same, it prints "Numbers are the same".

```
Input : 2 4 6
Output: 6
```

```
Input : 2 2 2
Output: Numbers are the same
```

Loops

Chapter 6

6.1 Basic commands

6.1.1

Most of the time we need to repeat a part of the algorithm. The commands that make it possible are named **loops**. With each repeat, it is important to know **what** (body of the loop) needs to repeat and **how many times** (condition of the loop) it needs to repeat.

Loops make it possible to repeat parts of code a given number of times or till the condition is met, for example:

```
how many times:     repeat 10 times
what:                 do a squat
```

```
how many times:     till you have money on bank account
what:                 buy gifts
```

```
how many times:     while you are not at the end of text
what:                 replace word "five" with "5"
```

6.1.2

How is named the statement in code that allows you to repeat commands?

- loop
- condition
- complex command

6.1.3

We often know that when designing the program some actions are needed to repeat. Loop execution is guarded by an integer control variable that is set at the start of the loop to a specific value. This value is changed in each iteration of the loop.

The loop is executed till the condition of the loop is met.

```
int i; // declaration of the control variable
for(i = 1; i <= 10; i = i + 1) { // control part of loop
```

```
  command; // loop body, list of commands that are to be
executed
}
```

Each part of the loop represents the following:

- **for(…)** – the loop command definition says it's a loop with a known count of iterations
- **i** – control variable
- **i = 1** – set of the start value for the variable
- **i <= 10** – condition for the loop; if the condition is met then the commands in loop body will be executed; if it is not met then the loop will terminate
- **i = i + 1** – loop step, after executing the commands in the loop body, the value will change each step based on this command: it will increase by 1

6.1.4

Fill in the code so that it prints 5-times the text "Hello".

```
class Hello {
  public static void main(String[] args) {
    int i;
    _____(i = 1; i <= _____; i = i + 1) {
      System.out.println("Hello");
    }
  }
}
```

6.1.5

Variable **i** is used only inside the loop. It is enough to declare it in its head.

We save one line of code and the variable stops existing after the termination of the loop.

It will not happen that we use it later in the code with no specified value.

```
for(int i = 1; i <= 10; i = i + 1) {
  command;
}
// variable i is no longer available
```

6.1.6

Fill in the code so that the loop will be executed 10 times.

```
for(_____ i = 1; i _____ 10; i = i _____ 1) {
 System.out.println("Hello")_____
}
```

6.1.7

The loop condition can have different shapes. The loop iteration 10 times can be written like this:

```
for(int i = 1; i<= 10; i = i + 1) { ... }
```

as well as:

```
for(int i = 1; i < 11; i = i + 1) { ... }
```

Both notations will ensure that the loop will be executed last time when the variable **i** will have the value 10.

6.1.8

How many times will be executed the following loop?

```
for(int i = 1; i < 7; i = i + 1) { ... }
```

6.1.9

On the inside the loop works following:

```
for(int i = 1; i <= 5; i = i + 1) { ... }
```

1. initialization - the control variable is set with a value
2. it is verified that the loop condition is met - if yes, the commands of the loop are executed; otherwise, the loop terminates
3. the commands in the loop body are executed
4. the loop step is executed - the value of the control variable is changed, and it continues with step number 2.

The notation

```
for(int i = 1; i <= 1; i = i + 1) { ... }
```

will execute the loop just one time.

The notation

```
for(int i = 1; i <= 0; i = i + 1) { ... }
```

will execute the loop not even once.

6.1.10

How many times will be executed the following loop?

```
for(int i = 0; i < 5; i = i + 1) { ... }
```

6.1.11

The loop condition can have different shapes. The loop iteration 10 times can be written like this:

```
for(int i = 1; i<= 10; i = i + 1) { ... }
```

as well as:

```
for(int i = 1; i < 11; i = i + 1) { ... }
```

Both of the notations will ensure that the loop will be executed last time when the variable **i** will have the value 10.

6.2 More about Loops

6.2.1

The loop is mainly used so that we use the control variable in the commands. The content of the control variable will be output from 1 to 10.

```
for(int i = 1; i <= 10; i = i + 1) {
  System.out.println(i);
}
```

The command will write in order the values 1, 2, 3 ... 10.

6.2.2

What will be the output after the loop ends?

```
for(int i = 1; i < 4; i = i + 1) {
  System.out.print(i);
}
```

6.2.3

Fill in the code so that the values from 3 to 7 appear below each other:

```
for(_____ i = _____; i < _____; i = i + 1) {
  System.out._____(i);
}
```

6.2.4

Except for the loop where the control variable is changed from a lower value to a higher value, we can use a notation where the control variable is changed from a higher value to a lower value:

- by initialization is the starting value higher
- in the loop step is the value of **i** decreasing by 1: **i = i – 1**

```
for(int i = 10; i > 5; i = i - 1) {
  System.out.println(i);
}
```

Similar rules are used as in the previous case:

- the control variable is set with an initialization value
- the loop condition is verified
- the loop body is executed
- the value of the control variable is changed based on the rule (in this case **i = i – 1**)

6.2.5

Fill in the code so that the result shall be in line values 987654:

```
for(int i = _____; i > _____; i = i _____ 1) {
  System.out._____(i);
}
```

6.2.6

The loops with a known count of iterations are used so, that after a few iterations they terminate but sometimes occur situations when the loop does not terminate, for example:

```
for (int i = 10 ; i >= 5; i = i + 1) {
  System.out.println(i);
}
```

The control variable is set to the value 10 and in each step, it is increased.

But the condition is set for **i >= 5** and it is met for each next step.

The value of **i** increases to the infinite or to the maximum integer value.

6.2.7

Make sure that the loop terminates:

```
for (int i = 10 ; i >= 5; i = i _____ 1) {
  System.out.println(i);
}
```

6.2.8

How many times will be the loop executed?

```
for (int i = 10 ; i >= 5; i = i - 1) {
  System.out.println(i);
}
```

6.2.9

Calculate the sum of the first 100 positive numbers.

Our task is to add the values *1 + 2 + 3 + 4 + 5 + 6 .... + 99 + 100*.

The numbers will be added gradually - in a loop that will be repeated from 1 to 100 and add each next value.

To save the temporary result we need space where the numbers will be added to - to a variable.

The variable **sum** will be increased in each step by the value of the variable **i**.

```
  int sum = 0;
  // from 1 to 100 in each step is i increased by 1
  for(int i=1; i <= 100; i = i + 1) {
    sum = sum + i; // i is added
  }
 System.out.println(sum);
}
```

To show how the loop is executed and how the values of each variable are changed is used a watch table, which contains values of each variable in each loop step.

i	Note	sum
	before the loop execution	0
1	sum = sum+1; //sum=0+1	1
2	sum = sum+2; //sum=1+2	3
3	sum = sum+3; //sum=3+3	6
4	sum = sum+4; //sum=6+4	10
...	...	...
99	sum = sum+99; //sum=4851+99	4950
100	sum = sum+100; //sum=4950+100	5050

6.2.10

Order the rows of the program that will find the sum of numbers between two values for which **a < b**.

- }
- }
- int sum = 0;
- for(int i = a; i <= b; i++) {
- public static void main(String[ ] args) {
- sum = sum + i;
- int a = input.nextInt();
- System.out.println(sum);
- Scanner input = new Scanner(System.in);
- public class Application {
- }
- import java.util.Scanner;
- int b = input.nextInt();

6.3 For cycle (programs)

6.3.1 Repeat print

Write an algorithm that prints "Hello" 10 times to the console. Each word is in a separate line.

```
Output:
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
```

6.3.2 Numbered print

Write an algorithm that prints 10 times word Hello with the number to the console - in the form "1Hello" and in the next line "2Hello" ... "10Hello".

```
Output:
1Hello
2Hello
3Hello
4Hello
```

```
5Hello
6Hello
7Hello
8Hello
9Hello
10Hello
```

6.3.3 The sum of n numbers

Write the code to get the sum of the first *n* integer numbers given at the input. Print the intermediate results.

```
Input: 5
Output:
1
3
6
10
15
```

```
Input: 4
Output:
1
3
6
10
```

6.3.4 Factorial

Write a code that calculates the factorial for the given number *n* (*n! = n. (n-1). ... .3.2.1*). Print the intermediate results.

```
Input : 3
Output:
1
2
6
```

```
Input : 4
Output:
1
2
6
```

```
24
```

6.3.5 The product of positive numbers without multiplication

Write an algorithm that calculates the product for two positive integers without using the multiplication operation.

```
Input : 5 3
Output: 15
```

```
Input : 5 5
Output: 25
```

```
Input : 2 5
Output: 10
```

6.3.6 The product of numbers in the interval

Write a code that calculates the product of all integers between the two given values. Ensure that the program displays the values of the variables in each cycle step during the run.

```
Input : 5 7
Output:
1 - 5
2 - 30
3 - 210
210
```

```
Input : 2 5
Output:
1 - 2
2 - 6
3 - 24
4 - 120
120
```

6.3.7 Multiplication table

Write an algorithm that prints a small multiplication table for the given integer.

```
Input : -5
Output:
```

```
1 * -5 = -5
2 * -5 = -10
3 * -5 = -15
4 * -5 = -20
5 * -5 = -25
6 * -5 = -30
7 * -5 = -35
8 * -5 = -40
9 * -5 = -45
10 * -5 = -50
```

```
Input : 5
Output:
1 * 5 = 5
2 * 5 = 10
3 * 5 = 15
4 * 5 = 20
5 * 5 = 25
6 * 5 = 30
7 * 5 = 35
8 * 5 = 40
9 * 5 = 45
10 * 5 = 50
```

6.3.8 The product of numbers without multiplication II.

Write an algorithm that detects the product for two integers (even **negative ones**) without using a multiplication operation.

```
Input : -5 3
Output: -15
```

```
Input : -5 -5
Output: 25
```

```
Input : 2 5
Output: 10
```

6.4 Loops with conditions

6.4.1

Sometimes when we use loops we do not know how many times it will be repeated but we can specify a condition till which the loop should repeat. For example: while you are hungry, eat a cake.

Execution of a loop can be done using the command **while** and by the condition that will specify the execution of commands in the loop body.

The notation of loops with the condition at the beginning is following:

```
while (condition) {
  command;
}
```

The condition has to be written in the brackets.

6.4.2

What keyword (command) is defined for the loop with the condition at the beginning?

6.4.3

Write 10-times "Hello" below each other.

The task is almost similar to in the case of the for a loop. Each task that needs to repeat some commands can be done using any type of loop and it is only on our choice which loop we will choose.

This time has the programmer has a task to specify all the operations that the for loop structure contains:

- set the start value of the control variable
- a condition that terminates the loop
- execution of commands in the loop
- increasing the value of the control variable

```
int i = 1; // initialization of control variable
while (i <= 10) {  // when the condition is met, do
  System.out.println("Hello"); // command execution
  i = i + 1; // increasing the value of the variable
}
```

6.4.4

Fill in the code so 5 dots are printed:

```
int i = 4;
_____ (i <= _____) {
  System.out.print(".");
  i = i + 1;
}
```

6.4.5

Write even numbers from 8 to 24 below each other.

We will write the content of the variable that will be increased in each step by 2.

The activity will be done until the value is not 24.

```
int num = 8;
while (num <= 24) {
 System.out.println(num);
 num = num + 2; // we increase the value by 2
}
```

The task can be rewritten also to a for loop using the following code:

```
for(int num = 8; num <= 24; num = num + 2)
 System.out.println(num);
```

6.4.6

Fill in the code so it prints all the numbers divided by 10 without the division operation that are less than the given number.

```
class Example {
  public static void main(String[ ] args) {
   Scanner input = new Scanner(System.in);
```

```
    int max = input.nextInt();
    int num = 0;
    _____ (num < _____) _____
      System.out.println(num);
      num = num + _____;
    _____
}
```

6.4.7

Except for the loop with the condition on beginning, we can use also a loop with the condition on end. This type of loop executes command till the condition is met- but only in order, first, it executes and then it evaluates the condition.

The notation is following:

```
do {
  command1;
   ...
  commandn;
} while (condition);
```

For example:

```
int i = 1;
do {
  System.out.println(i);
  i = i + 1;
} while (i<10);
```

The main difference with the other loops is that the commands in the loop body are executed at least once. After the first run is evaluated whether it shall continue with the repeat.

6.4.8

Which statements are true?

- loop with condition on end will be executed at least once
- loop with condition on beginning does not have to be run any times
- loop with condition on end does not have to be run any times
- loop with condition on beginning will be executed at least once
- loop with known number of repeats will be executed at least once

6.5 While loops (programs)

6.5.1 Division remainder without division

Write a code that will print the division remainder after the division of the first number by the second one without the use of division or modulo. In the case of zero division write "Zero cannot be divided".

```
Input : 15 5
Output: 0
```

```
Input : 5 0
Output: Cannot divide by zero
```

```
Input : 10 3
Output: 1
```

6.5.2 The greatest common divisor

Write a code that uses the Euclidean algorithm (using modulo operator %) to find out what is the greatest common divisor of two integers given at the input and separated by a space.

```
Input : 15 5
Output: 5
```

```
Input : 28 12
Output: 4
```

```
Input : 10 3
Output: 1
```

6.5.3 Euclidean algorithm

Write a code that will compute the greatest common divisor of two integers using the subtraction-based version of the Euclid-s algorithm (which was Euclid-s original version). In addition, the code should compute the smallest common multiple of these two integers (using the divisor computed within the first step). The input contains two integer numbers. Print the greatest common divisor and the smallest common multiple.

```
Input : 25 40
```

```
Output: 5 200
```

```
Input : 33 196
Output: 1 6468
```

6.5.4 Minimum and maximum

Write a code that will compute the min and the max values of the given series of integers. Do not use an array of integers. The input contains the series of numbers ended by the number 999999 (not a part of the series). Print the min and the max.

```
Input : 8 4 -5 33 22 56 45 -32 0 23 999999
Output: -32 56
```

```
Input : 3 -3 0 -5 -33 999999
Output: -33 3
```

Numeric Data Types

Chapter 7

7.1 Integer

📖 7.1.1

Integer variables are capable to save values of integral type and do following operations:

- + (addition) *a + b*, e.g.: *10 + 3 = 13*
- - (difference) *a – b*, e.g.: *10 - 3 = 7*
- * (multiplication) *a * b*, e.g.: *10 * 3 = 30*
- / (integral division) *a / b*, e.g.: *10 / 3 = 3*, where the decimal part is neglected
- % (remainder after division) *a % b*, e.g.: *10 % 3 = 1*

📝 7.1.2

What will be the output of the following code?

```
class Example {
  public static void main(String[ ] args) {
    int a = 17, b = 5;
    int c = a / b;
    System.out.println(c);
  }
}
```

📖 7.1.3

Integral numbers offer a special operation that returns the remainder after division. For its calculation is used the operator **%**.

E.g.:

- *10 % 3 = 1*
- *10 % 2 = 0*
- *15 % 7 = 1*
- *20 % 7 = 6*
- *10 % 0* – division by zero = error

📝 7.1.4

What will be the output of the following code?

```
class Example {
  public static void main(String[ ] args) {
    int a = 17, b = 5;
    int c = a % b;
    System.out.println(c);
  }
}
```

7.1.5

Working with integral numbers means also working with negative numbers that represent the other half of all of the integral numbers. A negative number is written using the symbol **-** placed before the numerical value.

E.g.:

```
int c = -1;
int d = 15 + -5;
```

If we want to stay loyal to math notation, we can enclose the negative value into brackets, e.g.:

```
int e = 15 / (-5);
```

7.1.6

What will be saved in variable **p** after executing the following commands?

```
int a = -3;
int b = 15 / -5;
int p = a - b;
```

7.2 Incremental and decremental operator

7.2.1

Change of the value of the variable by 1 is done using the incremental and decremental operator that replaces the "long" notation that serves to increase or decrease the value of the variable by 1.

Instead of:

```
i = i + 1;
```

we can use the incremental operator **++**:

```
i++;
```

Instead of:

```
i = i - 1;
```

we can use the decremental operator **--**:

```
i--;
```

7.2.2

What will be saved in the variable **a** after the execution of the following commands?

```
int a = 10;
a++;
a = a - 5;
a++;
a--;
```

7.2.3

Operators **++** and **--** can be placed also before the variable. The usage has in both cases similar effect:

```
int a = 1;
a++;
System.out.println(a);
```

```
int a = 1;
++a;
System.out.println(a);
```

However, if we use it in expressions, it behaves different:

++a will firstly increase the value of the variable and then it is used in the expression

```
int i = 10;
int j = 3;
int k = 0;
k = ++j + i; // result: k = 4 + 10 = 14 a j = 4
```

a++ will firstly use the value of the variable in the expression and after the calculation, it will be increased

```
int i = 10;
int j = 3;
int k = 0;
k = j++ + i; // result: k = 3 + 10 = 13, ale j = 4
```

7.2.4

What will be the output of the following code?

```
class Example {
  public static void main(String[ ] args) {
   int a = 5, b = 10;
   int c = a++ * --b;
  System.out.println(c);
 }
}
```

7.2.5

Assign the variables correct values:

```
   int a = 0;
   int b = a++; // after the operation contains a the value
_____, b contains the value _____
   int c = ++a + 5; // after the operation contains a the
value _____, c contains the value _____
   b = b++ + ++a;   // after the operation contains a the
value _____, b contains the value _____
```

7.2.6

Incremental and decremental operators are used to shorten the notation. The change of value can be done e.g. inside the condition where this notation saves us one row of code.

We can write:

```
int i = 0;
do {
  System.out.println(i);
```

```
} while (i++ < 10);
```

instead of:

```
int i = 0;
do {
  System.out.println(i);
  i++;
} while (i < 11);
```

7.2.7

How many values will print the following loop?

```
int i = 0;
while (++i < 5)
  System.out.println(i);
```

7.2.8

In addition to the incremental and decremental notation we can use also other shortening notation of other math operations, e.g., assigning:

```
i = i + 5;
```

can be shortened to

```
i += 5;
```

which means that the variable on the left side will be assigned the original value added by 5.

For

```
n *= 3;
```

will be the value of variable **n** multiplied by 3 and saved to the variable **n**.

7.2.9

What value will be printed after executing the following commands?

```
class Example {
```

```
  public static void main(String[ ] args) {
      int a = 0;
      a += 7;
      a *= 2;
      int b = 20 % a;
      b--;
      b += a;
      System.out.println(b);
  }
}
```

7.2.10

Many times, there are tasks where we have to decide whether the given value is even or odd.

When searching for a solution, we can use the fact that even numbers divided by 2 give the remainder after division 0 and odd numbers give 1.

E.g.:

- *10 % 2 = 0* – is even
- *11 % 2 = 1* – is odd

7.2.11

Fill in the code so that it decides whether the given integer value is even or odd:

```
import java.util.Scanner;

class App {
  public static void main(String[ ] args) {
    Scanner input = new Scanner(System.in);
    int a = input._____();
    if (a _____ 2 == _____)
      System.out.println("Value is even");
    else
      System.out.println("Value is odd");
  }
}
```

7.3 Number types

7.3.1

One of the characteristics of data types of variables is the range of values it can save, i.e., minimum and maximum value.

In the case of data type **int** that are values from -2147483648 to 2147483647.

Except of this type, there are other data types to save integral values:

- **byte**: -128..127
- **short**: -32 768 .. 32 767
- **int**: -2 147 483 648 .. 2 147 483 647
- **long**: -9 223 372 036 854 775 808 .. 9 223 372 036 854 775 807

These data types are used when we require a less or higher range for saving values.

7.3.2

Order the data types by the range of values from the lowest to the highest.

- int
- long
- short
- byte

7.3.3

Many tasks can be solved using integral operations but there are tasks where we need to use decimal numbers. To save decimal (real) numbers is used the data type **double**.

Declaration

```
double c;
```

makes it possible to the variable **c** save values from range -9 223 372 036 854 775 808 to 9 223 372 036 854 775 807 where the values can be decimal too.

The real numbers are written in standard format:

```
3.1415296536, 583.45
```

or in the scientific format:

```
5.8345e2
```

which means $5.8345 * 10^2 = 583.45$.

7.3.4

Fill into the declaration the correct data type:

```
_____ a; // integer numbers from -128 to 127
_____ b; // integer numbers with the highest range
_____ c; // decimal numbers
```

7.3.5

The decimal part is separated in code by a decimal point, e.g.

```
5.0
```

```
3.145
```

```
0.0001
```

etc.

7.3.6

Which values represent the real numbers in the Java programming language?

- 0.5
- 5.0
- 0,5
- 5,0

7.4 Real numbers

7.4.1

When you combine the integer and real type, the result is a real number (number with decimal point).

The output of the following code:

```
double a = 10; // real number
int b = 5; // integer number
System.out.println(a - b); // real number
```

is a real number. This is represented by the following notation of the result

```
5.0
```

Despite the result is an integer number (5), to obtain it was used a **double** type variable, so the result has to be written in this data type

7.4.2

What is the result of the following code?

```
double a = 2, b = 4.5;
double c = a * b;
System.out.println(c);
```

- 9.0
- 9
- 9.00
- 9,0

7.4.3

Often it is necessary to transform the decimal number to integer one. With a simple assignment, it will be not working.

The most used math function for this use is rounding. This function (**round**) is available in the library **Math**, which is written following:

```
double x = 10.51;
long a = Math.round(x);
```

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

The result of the operation is an integer value not of the **int** type but a **long** type that can save bigger values.

7.4.4

Fill in the following function to round the content of the variable *a*:

```
double a = 9.991;
long b = _____._____(a);
```

7.4.5

If we need to transfer the **long** type variable into an **int** type variable, we need to **retype** it. This will change the value from the original type so that it will be possible to save it into a new type and its value will be the same.

The notation

```
long b = 15;
int c = (int) b;
```

will update the content of the **b** variable so that it can be assigned to the **c** variable and copied to it.

Warning:

> This operation is not flawless. If you try to input into variable with bigger range into a variable with smaller range, then the it will be executed but the value will not be correct.
>
> The programmer has to think about this kind of situation and the secure the code before this mistake.

7.4.6

Make the retype of the variable **d** which is of a **long** type into an int variable **c**:

```
int c = _____d;
```

7.4.7

Retype can be realised also between real and integral values. By the reptype of real variables will the decimal part be removed.

After executing the following code:

```
double c = 5.8;
int d = (int) c;
```

will the variable **d** contain the value **5**.

7.4.8

Fill in the values that will be in the variables after executing the following commands:

```
double c = 10.51;
long d = Math.round(c); // d will contain _____
int e = (int) c; // e will contain _____
```

7.4.9

To round the number to specific decimal points is used the following approach.

E.g. to round to two decimal points:

```
double pi = 3.14159;
double pi2 = Math.round(pi*100)/100;
```

Multiplying the value of the variable by 100 will move the decimal point by 2 places to 314.159

This value will be rounded using the function **round** to integral number, i.e. 314

and finally, it will be divided by 100 and the decimal point will be moved by 2 places to the left.

The result will be 3.14

7.4.10

Fill in the values so that variable **a** will be rounded to one decimal point and variable **b** will be rounded to three decimal points:

```
double a = 6.845;
double b = 8.55478;
double new_a = Math.round(a*_____)/_____;
double new_b = Math.round(b*_____)/_____;
```

7.4.11

What will be the result of the following code?

```
double a = 100;
double b = Math.round(a/3*100)/100;
double c = a - b;
int d = (int)c;
System.out.println(d);
```

7.4.12

Loading a real number from the input to the program is done using the command **nextDouble()**.

The following code will read two decimal numbers from input and calculate its division.

```
class App {
    public static void main(String[] args) {
      Scanner vstup = new Scanner(System.in);
      double a = vstup.nextDouble();
      double b = vstup.nextDouble();
      double division = a / b;
      System.out.println(division);
    }
}
```

7.4.13

Fill in the code to calculate the content of a square:

```
class App {
    public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      double a = input._____;
      double cont = a _____ a;
      System.out.println(cont);
    }
}
```

7.5 Number types (programs I.)

7.5.1 Division and remainder

Write a code that divides two integers to determine the proportion and remainder (use integer division operations). Treat the division by zero at the beginning of the program and if this happens write "Zero cannot be divided". Otherwise, list the division result and the remainder separated by a space.

```
Input : 4 5
Output: 0 4
```

```
Input : 9 0
Output: cannot divide by zero
```

```
Input : 2 2
Output: 1 0
```

7.5.2 Even numbers 1 - 20

Write an algorithm that prints even numbers from 1 to 20.

```
Output:
2
4
6
8
10
12
14
16
18
20
```

7.5.3 Triangle

Write a code that will check whether the given three numbers may be the sides of the triangle. The input contains three double numbers. If these values may describe triangle sides, then print the area of that triangle. Otherwise, print -1.

```
Input : 3 4 5
Output: 6.0
```

```
Input : 1 2 3
Output: -1
```

7.5.4 Sum of real numbers

Write a code which for two real numbers (entered in a single line and separated by spaces), calculate their sum, round it and write it out.

```
Input : 8.2 2.5
Output: 11
```

```
Input : 1.5 3.3
Output: 5
```

7.5.5 Area and perimeter of a circle

Write a program that calculates the area and perimeter of the circle for the specified radius (**double** decimal). Let *pi* be 3.14. Round the results (using the **Math.round()** command) and separate the results with a space.

```
Input : 5
Output: 79 31
```

```
Input : 4.5
Output: 64 28
```

7.5.6 Perfect number

In number theory, a perfect number is a positive integer that is equal to the sum of its proper positive divisors, that is, the sum of its positive divisors excluding the number itself. Write a code that will check if the given number is perfect. The input contains the positive integer number. Print the value true if the given number is perfect or false if not.

```
Input : 28
Output: true
```

```
Input : 999
Output: false
```

7.5.7 Count of divisors

Write a code that detects and prints the number of divisors for the given number.

```
Input : 7
Output: 2
```

```
Input : 12
Output: 6
```

```
Input : 100
Output: 9
```

7.5.8 Count bits of 1

Write the code that will compute a number of bits set to 1 within the binary representation of the given number. Input the integer number. Print the number of ones.

```
Input : 7
Output: 3
```

```
Input : 0
Output: 0
```

```
Input : 1
Output: 1
```

```
Input : 1234567890
Output: 12
```

7.5.9 Coins

Write a code that will print a set of EURO coins that make up the given amount. Available coins are: 1c, 2c, 5c, 10c, 20c, 50c, 1€ and 2€. Input the amount of money is double type. Print the coins in descending order, separating the values with a space.

```
Input : 5.25
Output: 2€ 2€ 1€ 20c 5c
```

```
Input : 0
Output:
```

7.5.10 BMI index

Write a code that calculates the BMI index and print whether you are overweight or not. BMI (body mass index) is calculated as the ratio of the weight in kilograms and the square of the height in meters. *BMI<18.5* underweight, *18.5<= BMI<25* normal weight, *25<= BMI<30* overweight, *BMI>30* obesity.

```
Input : 45 1.70
Output: underweight
```

```
Input : 90 1.65
Output: obesity
```

```
Input : 80 1.80
Output: normal weight
```

7.6 Number types (programs II.)

7.6.1 Sorting numbers

Write a code that for two integers given at the input ensures that the larger of them is stored in variable *a*, the smaller in variable *b*. If the numbers are equal print to the console: "Numbers are equal", otherwise print first the bigger and then the smaller value.

```
Input : 3 2
Output: 3 2
```

```
Input : 2 8
Output: 8 2
```

```
Input : 2 2
Output: Numbers are equal
```

7.6.2 Prime number

Write the code that will check whether the given number is a prime number. Input is an integer number greater than 0. Print **true** if the number is prime and **false** otherwise.

```
Input : 5
Output: true
```

```
Input : 9
Output: false
```

```
Input : 1
Output: false
```

7.6.3 Value from interval

Write a code to see if the given number is in a specified interval ("yes" or "no" answer). At the beginning of the algorithm check that the interval you entered is correctly sorted (e.g., not 5,2 but 2,5), if not, correct it.

As input is a triple of integer values representing the two-interval boundaries and the given value.

```
Input : 5 10 7
Output: yes
```

```
Input : 10 20 10
Output: yes
```

```
Input : 30 4 85
Output: no
```

7.6.4 Day or night

Write the code that determines whether it is day or night, respectively light or dark, based on the specified hour (1-12) and the time period (0 = morning, 1 = afternoon). Suppose the sun rises at 6 am and sets at 6 pm.

```
Input : 10 0
Output: day
```

```
Input : 8 1
Output: night
```

```
Input : 12 0
Output: day
```

Other Data Types

Chapter 8

8.1 Logical type and logical expression

8.1.1

We are working often also with logic values that can have the value **true** or **false**.

Data type **boolean** is used to save this kind of value.

Often it is the result of comparison or evaluation of a condition.

E.g.:

the condition whether **a > b** can be evaluated by the following notation using the **if** structure:

```
if (a > b)
```

but the result of the evaluation of the expression can be saved into a variable

```
boolean res;
int a = 10, b = 5;
res = a > b;
System.out.println(res);
```

If the value **a** higher than **b** then the variable **res** is the result as **true**. Otherwise (less or equal) will the variable **res** be of value **false**.

8.1.2

Declare the variable **t** as a variable to save true/false values and assign it the result of the comparison of variable **a** and the value **5** for equality.

```
_____ t;
int a = 7;
t = a _____ 5;
```

8.1.3

The logical type is linked with comparison operators so we can say again:

- **>** - is bigger, e.g. **a > b**
- **>=** - is bigger or equal, e.g. **a >= b**
- **<** - is less, e.g. **a < b**
- **<=** - is less or equal, e.g. **a <= b**

- **==** - is equal, e.g. **a == b**
- **!=** - is not equal, e.g. **a != b**

Usage of symbols in the wrong order will raise an error (e.g.: **=>**, or **<>**).

8.1.4

Which comparison operators are correct?

- >=
- <|=
- ==
- !=
- <|>
- =>
- =<|

8.1.5

The result of the comparison can be used also in conditions so that we will get the result of the expression and then use it in condition, e.g.

```
int a = 10, b = 5;
boolean res = a == b
if (res == true)
  System.out.println("Values are equal");
else
  System.out.println("Values are different");
```

Notation

```
if (res == true)
```

can be usually written following

```
if (res)
```

because the result of the condition **res == true** is dependent on the value of the variable **res**.

If it is true,

```
if (res == true)
```

we ask if truth is truth (**true == true**) – the result is **true**.

If the variable contains the value representing untruth:

```
if (res == true)
```

we ask if the untruth is truth (**false == true**) – the result is **false**.

8.1.6

Fill in the code so that it prints whether the number is negative or positive.

```
import java.util.Scanner;

class App {
    public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      double a = input._____();
      boolean negative = a _____ 0;
      if (negative)
        System.out.println("negative");
       _____
        System.out.println("positive");
    }
}
```

8.1.7

Despite the verification of whether the condition is true, it is right to also use the notation: if it is not true then, e.g.:

```
int a = 5, b = 0, division;
boolean zeroDivider = b == 0;
if (!zeroDivider)
  division = a / b;
....
```

The notation beginning with **!** will **negate** the result of the expression or content of the variable after the exclamation mark- from value **true** will be made **false** and vice versa.

In this case, contains the variable **zeroDivider** the value **false** and the notation in the condition means the following:

- if it is not true that **zeroDivider** then calculate the division,
- or if **zeroDivider** contains the **false** value then execute,
- or if the negated content of the variable **zeroDivider** is true, the execute.

8.1.8

What kind of symbol is used to negate the content of a logical value/variable?

- !
- ?
- -
- **

8.1.9

Execution of the actions until the condition is not met or until the content of the logical variable is **false** is usually used in loops- until it is not true, execute commands.

```
boolean end = false;
int i = 1;
while (!end) { // until the variable end is false will be the
loop executed
 …
 i++;
 if (i > 10) end = true;
}
```

8.1.10

Fill in the code so that the program will read the values from the input till the input value is not zero. It will also write whether the number is even or odd.

```
import java.util.Scanner;

class App {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        _____ isZero = false;
        do {
         int a = input.nextInt();
```

```
        isZero = a _____ 0;
        if (a _____ 2 == 0)
           System.out.println("even");
        else
           System.out.println("odd");
       } while (_____isZero);
       System.out.println("end");
    }
}
```

8.2 Compound conditions

📖 8.2.1

Often you combine in codes many conditions that can be in different relations. Mostly we are in the following situations:

- all of the conditions have to be met at the same time,
- it is enough that only one of the conditions is met.

Based on the given age of the employee decide whether he/she is in productive age- between 18 and 70 years old.

The task can be solved following:

```
int age = input.nextInt();
if (age >= 18) // first condition is met
  // we verify whether the age is also less then the upper
boundary
  if (age <= 70)  // both of the conditions are met
    System.out.println("he/she is in productive age");
```

A simple notation makes it possible to write both notations into one complex condition. We use a logical connector AND (we use **&** in Java) to secure that both conditions have to be met at the same time.

So, the notation of two conditional commands is shortened by its combination into one complex condition:

```
int age = input.nextInt();
if ((age >= 18) & (age <= 70))
```

We put into the brackets each condition as well as the whole expression.

8.2.2

Fill in the expression so that it is true if both conditions are met at the same time:

```
...
if _____ (month >=3) _____month <= 6) _____
   System.out.println("spring");
```

8.2.3

Except the **&** operator can be used the alternative **&&**. Between the operators **&&** and **&** is the difference that **&&** will end the evaluation of the logical expression at the moment it finds out that the condition is not true and the following evaluation does not have an effect on the result, where **&** evaluates till the end.

Thinking about their differences has meaning only when the evaluation consists also of change of variables that are in the evaluation, e.g.:

```
int i = 0, j = 10;
boolean test;
test = (i > 10) && (j++ > 9);   // will end by &&
      // test = false, j = 10
test = (i > 10) & (j++ > 9);    // will go through all
executions
      // test = false, j = 11
```

8.2.4

What will be the output of the following code?

```
int a = 5, b = 7;
System.out.println(!(a > b)&& (b > a));
```

8.2.5

In some cases, it is necessary that only one condition needs to be met. In that case, is used logical connector **OR** written using the symbol |.

```
if ((a>0) | (b<0))
```

Evaluation of the expression is true if at least one of the conditions is met, i.e. it is enough if **a > 0** or **b < 0**.

If both conditions are met, the expression is also true.

Except the | operator can be used the alternative operator ||. Between the | and || operators is the difference that || will end the evaluation of the logical expression at the moment it finds out that the result of the condition is true and the following evaluation does not have an effect on the result, where | evaluates till the end.

8.2.6

Fill in the code so that it prints if the number is acceptable if it is positive or even. Evaluate the condition most effectively.

```
int num = input.nextInt()
if ((num > _____) _____ (num _____ 2 _____ 0))
  System.out.println("accept");
```

8.2.7

The combination of logical expressions and logical variables does not have to be restricted only to two elements. The evaluation is then done so, that first are evaluated the expressions in brackets, then the negation and then it goes from left to right.

E.g.

```
boolean h1 = false;
int a = 5, b = 7;
boolean res = (!(a > b) || (b - 5 < a) && h1 || !h1)
```

will be evaluated as:

```
(!false || true && false || !false)
(true   || true && false || true)
(      true     && false || true)
(             false      || true)
(                       true)
```

8.2.8

Make sure that the following fragment of code is outputting the text "do sport" in case that the division of height and weight is less than 2 or when the weight is higher than 150 kg. Evaluate the conditions most effectively.

```
if ((height/wieght _____ 2) _____ (weight > _____))

  System.out.println("do sport");
```

8.2.9

Make sure that the following code writes the following text with an effective evaluation of conditions:

- if the average is 1,5 or better – "excellent"
- if the average is bigger than 1,5 and less or equal than 4 – "good"
- if the average is bigger than 4 – "bad"

```
 if (average <= 1.5) System.out.println("excellent");
 if ((average > 1.5) _____ (average _____ 4))
System.out.println("good");
 if (average > 4) System.out.println("bad");
```

8.2.10

What will be the output of the following code as a result of the expression?

```
int a = 5, b = 7;
System.out.println((a < b) && !(b < a));
```

- True
- False

8.2.11

What is the result of the following expression:

```
int a = 5, b = 5, c = 7;
boolean res = (!(a == b) && (++b - 1 == a) && (c + 3 > a) ||
(b - c != 0));
```

- True
- False

8.2.12

What is the result of the following expression:

```
int a = 5, b = 5, c = 7;
boolean res = ((a >= b) && (b - 1 == a) && (c + 3 > a) && (b -
c != 0));
```

- False
- True

8.2.13

What is the result of the following expression:

```
int a = 5, b = 5, c = 7;
boolean res = !((a > b) & (b - 1 == a) || (b - c != 0));
```

- False
- True

8.3 Char

8.3.1

A narrowly specialized data type is a **char** type that allows a single character to be stored in a variable of this type.

The declaration is following:

```
char x;
```

The assigned value is enclosed in apostrophes

```
x = 'A';
```

The evaluation of the content of the variable is done using a standard comparison:

```
if (x == 'a')
```

Usage of this data type is very limited, but it can fasten up some tasks.

8.3.2

Declare a variable that can save one character:

```
_____ p = 'a';
```

8.3.3

Values saved in **char** type variable can be compared also based on the alphabet order *'a' < 'b' < 'c' … < 'z'*.

where all of the uppercase letters are less than all of the lowercase letters

'A' < 'B' … < 'Z' < 'a' < … < 'z'

A hint can be for us a simplified coding **ASCII table** that contains 255 base symbols (despite that nowadays are alphabets coded using Unicode/UTF8).

The first 32 symbols are controlling but the others are used often.

Since the **char** type is based on a coding table, the characters 'a' and 'A' are not the same (are not equal).

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
32	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
48		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
64	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
80	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
96	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
112	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
128	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

8.3.4

Which of the statements are true?

- 'Z' <| 'a'
- 'c' <| 'f'
- '1' <| 'q'
- 'a' <| 'A'
- 'a' <| 'Z'
- '9' <| '&'

8.3.5

How do we know whether the given character is a lowercase or uppercase letter?

If the given character is placed:

- between the first uppercase and last uppercase character, then it's an uppercase letter,
- between the first lowercase and last lowercase character, then it's a lowercase letter.

```
char c = '1';
if ((c >= 'a') && (c <= 'z'))
     System.out.println("lowercase letter");
   else if ((c >= 'A') && (c <= 'Z'))
            System.out.println("uppercase letter");
         else
            System.out.println("not a letter");
```

8.3.6

Fill in the following code where you can find out whether the character in the variable **ch** is a digit:

```
if ((ch >= '_____') && (ch <= '_____'))
       System.out.println("is a digit");
 _____
       System.out.println("is not a digit");
```

8.3.7

Except the usually used symbols are in outputs used also escape sequences that can be used to print some special characters:

- **\'** – inputs into text apostrophe
- **\"** – inputs into text quotation marks
- **** – inputs into text backslash

or manipulation with the cursor by the output:

- **\t** – inputs into the text a tabulator
- **\b** – inputs into the text a backspace (will delete the character before \b)
- **\n** – inputs into the text a new row (text after the symbol will begin in a new row)

E.g.:

```
System.out.println("t: text \t with \t tabulators");
System.out.println("b: let\bter");
System.out.println("n: one \n two \n three");
```

Output:

```
t: text      with      tabulators
b: leter
n: one
 two
 three
```

8.3.8

Fill in the appropriate escape sequence so that the output is following:

```
She said:
"Come tommorrow."
```

```
System.out.println("She said: _____ _____Come
tommorrow._____");
```

8.4 Other data types (programs)

8.4.1 Compare three numbers (one condition)

Write the code that uses one compound condition to determine if the three integer values entered on the input are identical. If so, it will print "Are identical" otherwise print "Are not identical".

```
Input : 4 4 4
Output: Are identical
```

```
Input : 9 0 9
Output: Are not identical
```

8.4.2 Hex value

Write the code that will translate hexadecimal digits (A - F, accept lower and uppercase) to their decimal values.

The input contains a character. If it is the hexadecimal digit print its decimal value else print -1.

```
Input : A
Output: 10
```

```
Input : x
Output: -1
```

```
Input : b
Output: 11
```

8.4.3 Letter or digit?

Write the code that detects for input character that it is a number, letter, or other characters.

On the console, it prints: for the digit "digit", for the letter "letter", for the other "other character".

```
Input : 9
Output: digit
```

```
Input : a
Output: letter
```

```
Input : !
Output: other character
```

8.4.4 Triangle type

Write a code that for three numbers will check what kind of a triangle they form (equilateral, isosceles, right-angled). Input three numbers of double type. Print three boolean values (false or true) that correspond to each kind of triangle. If the numbers do not define a triangle, then print -1.

```
Input : 5 12 13
Output: false false true
```

```
Input : 5 5 5
Output: true true false
```

```
Input : 1 2 1
Output: -1
```

```
Input : 3 4 5
Output: false false true
```

String I.

Chapter 9

9.1 About String

9.1.1

In addition to variables storing primitive types, we need often also to use more extensive data. To save longer text (till the range of 2 GB) we use the **String** data type.

The variable declaration is following

```
String data;
```

The content put into the String variable type is enclosed in quotes:

```
data = "Sun is shinning";
```

The content can be to the variable saved also at the declaration:

```
String data = "Sun is shinning";
```

9.1.2

Declare the variable **a** so that it is possible to save strings to it and save there the text **data**.

```
_____ a = _____data_____;
```

9.1.3

The **string** is not a simple data type but it goes about a **class** that contains special methods that allow manipulation with the saved content. More about the classes will be said in the next chapters but for now, it's enough to know that the **String** variable type will have to ability to browse, count the characters, etc.

The simplest operation is getting the number of characters of the saved content. We get it using the **length()** method.

The method is separated from the name using the dot "." and ends with brackets:

```
String data = "Mama";
int len = data.length();
System.out.println(len);
```

Into the variable **len** is saved the number of characters that are contained in the variable **data**, i.e., it's **4**.

9.1.4

Fill in the code so that it returns the count of characters saved in the variable **a**.

```
_____ a = "Winter in forrest";
int l = a_____length_____;
System.out.println(l);
```

- string
- ->
- String
- ()
- .
- []
- ,

9.1.5

Strings can be connected very simple - the addition operator is used or the **concat()** method.

```
String a = "Steven";
String b = "Spielberg";
String c = a + b; // variable c contains the text
StevenSpielberg without space
String d = a + " "; // variable d contains the text "Steven "
with space at the end
d = d.concat(b);  // content of the variable d is changed so
that the content of variable b is added - the result in the
variable d will be "Steven Spielberg"
// Note that "d.concat(b)" returns a new string but does not
change d
```

9.1.6

Make sure that in the variable **c** is the content of the variables in order **b** and **a**.

```
String a = "200", b = "100";
String c = _____ _____ _____; // the value will be 100200
```

9.1.7

If we want to express the empty content of the integral variable, we often input the value 0.

If we want to express the empty content of the **String** type variable, we use the assignment:

```
String s = "";
```

sometimes it is used also

```
String s = null;
```

The first entry will create the variable containing the empty string, the second the variable containing nothing.

The method to approach the empty content depends on the programmer where the reasons for this double approach are related to the concept of classes (that we will talk about later).

By manipulating with the variable containing **null** is this value in some cases taken as four-character text so it's needed to verify if the given variable is not empty:

```
if (a == null)
 ...
```

9.1.8

Make sure that in the variable **c** is the content of the variables in order **b** and **a**.

```
String a = "200", b = "100", c = _____;
c.concat(_____);
c._____(_____);
System.out.println(c);
```

9.1.9

Since we are unable to predict the given string number of characters, we have to read the whole row using the **nextLine()** method

The code to read the string using the **Scanner** is following:

```
import java.util.Scanner;
```

```
public class App {
   public static void main(String[] args) {
       Scanner input = new Scanner(System.in);
       String a = input.nextLine();
        ....
       System.out.println(a);
    }
}
```

9.1.10

Fill in the code so that it reads the string and returns the number of its characters.

```
import java.util.Scanner;

public class App {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        _____ a = input._____();
        int l = a._____();
        System.out.println(l);
    }
}
```

- string
- nextInt
- next
- String
- length
- nextLine

9.1.11

The result of the connection of any variable with a string is a string. If we add (using the + operator) content of the variable to any string, then this content will be transformed (converted) to string and the result is then the connection of two and more strings.

E.g.:

```
int a = 10;
String b = "the content of variable a is: " + a;  // converts
int
```

```
double c = a / 3;
String d = "the content of variable c is: " + c; // converts
double
System.out.println(b + "\n" + d); // converts char
```

Using this we can also output text and numbers:

```
int res = 100 + 15 / 3;
System.out.println("Result is: " + res);
```

9.1.12

Fill in the part of the code so that the final output was in form of the addition of two variables (e.g. *2 + 3 = 5*, not *2+3=5*).

```
int a = 2, b = _____;
int c = a _____ b;
System.out.println(a + " _____" + _____ + " _____" + c);
```

9.1.13

Data types are divided to:

- **primitive**
- **reference**

Primitive data types

- during the declaration is in memory reserved as much space as the data types needs (e.g. int needs 4 bytes, long - 8 bytes, char - 2 bytes, double - 8 bytes, etc.)
- size of the reserved space is not changed during the existence of the variable
- the content of the variable is saved always to the place where is the reserved space and the variable does not change the place

  All the previously mentioned types were primitive.

Reference data type

- it is not possible to decide how much space is needed for them to reserve because two variables of the same type can have different demands on the memory
- while the variable exists, its memory requirements may change

- the typical example of the data type is **String** that can save any long text (string), e.g. "mama" or also "mama has little Ema at home"
- in this case, the declaration reserved in memory only space to save the reference to the memory
- when you input a new value then is in the memory **always** looked for a coherent memory block with the needed size. The new value is saved into this block and the reference is updated to the starting position of the memory block.

```
int num = 10;
String name = "Hello";
```

Memory address	Variable name	Data
1000	num	10
...		
1500	name	address(2000)
...		
2000		"Hello"
...		

9.1.14

Choose the primitive data types:

- int
- double
- char
- boolean
- long
- String

9.2 Data type selection

📖 9.2.1

By now we worked only with variables that saved numbers and characters. Before the use of the variable, we had to **declare** it. Each declaration contains a data type and the name of the variable

```
int n;
int number, result;
```

The data type defines the type of values that can the variable obtain. It can be, e.g.:

- integral numbers
- decimal numbers
- character (letter)
- the text string, etc.

Except for that, it defines also:

- the amount of memory that will be reserved for the variable
- a set of values that can be stored in a variable

📝 9.2.2

Declare a variable *number* that will be used to save integer values:

```
_____ number;
```

📖 9.2.3

The reason for the existence of data types is the acceleration of operations and their limitation to selected data types.

E.g.

by adding integral values **5 + 10** we get the value **15**,

but in case we add text values then the result is the connection of the strings:

"5" + "10" = "510".

Function **abs** (absolute value) is defined for numbers but not for text strings, etc.

9.2.4

What defines a data type?

- the amount of memory that will be reserved for the variable
- operations and functions that can be applied to values of that type
- data that can be stored in the variable
- how to list the variable content

9.2.5

Base data types in Java are:

- numerical
- we already worked with integral numbers (**int** – e.g.: 10)
- decimal numbers (**double** – e.g.: 1.3)
- text
- character – type able to save one character (**char** – e.g. ‘m’)
- string – can save text – sequence of characters (**String** – e.g. "my name is Ema")
- logical
- saves the truth value (**boolean** – only values **true** or **false**)

9.2.6

Choose the correct data type to save the given value:

```
______ p1 = 1034;
______ p2 = "Warning ";
______ p3 = 'A';
______ p4 = true;
______ p5 = 1.5;
```

- double
- double
- boolean
- String
- int
- char
- int
- char
- boolean
- String

9.3 Functions to work with string

9.3.1

The String consists of characters. Each character has its place in the string that is defined by the index. Java counts elements in any list so, that it starts from zero.

The first character in the string is on position 0, the second is on position 1, etc. The last character is placed on the position decreased by one from the whole count of characters in the string.

E.g. for:

```
String data = "Madonna";
```

are characters placed on each position following:

index/position	0	1	2	3	4	5	6
character	M	a	d	o	n	n	a

The count of characters in the string is 7 where the last character is on position 6.

9.3.2

What character is placed on position 3 of the string?

```
String data = "Indiana Jones";
```

9.3.3

To save the characters we use the data type **char**. If we want to read and save a specific character, we use the **char** variable and the ability of the **String** variable to return a character of a given position:

```
char m = myString.charAt(position);
String data = "Indiana Jones";
char begin = data.charAt(0); // returns the first character of
string - I
char c = data.charAt(5);     // returns the character on the
position 5 (sixth) - n
```

```
int l = data.length();        // returns the count of all
characters in string
char end = data.charAt(l-1); // returns the last character
```

The length of the string is 13 in this case (including the space) and the character on the last position has an index of 12.

9.3.4

Fill in the code so that the last character of the string is printed.

```
String data = "Amadeus";
int ln = data._____();
System.out.println(data._____(ln-1));
```

9.3.5

Fill in the code that finds out whether the given string begins with the character "a" or "A".

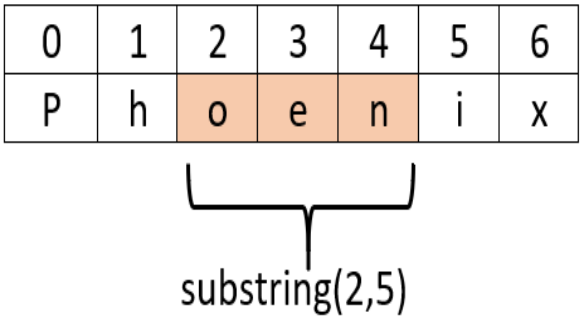
```
import java.util.Scanner;

public class App {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        String a = input._____();
        char first = a._____(_____);
        if ((first == _____a_____) || (first == _____A_____))
               System.out.println("it does contain");
         else
               System.out.println("it does not contain");
  }
}
```

9.3.6

Often, we need to get from the string not only one character but a substring. To obtain the part of the string is used a **substring** method. Its basic form defines the beginning position and the ending position of the substring.

The character chosen at the ending position is not counted to the substring. The method considers the characters from the beginning position to the character before the ending position:

```
String data = "Phoenix";
System.out.println(data.substring(2,5));  // prints characters
on the position 2-4 (so it does not take 5 into account) "oen"
```



9.3.7

Make the output to print the string "maged".

```
String data = "Armagedon";
System.out.println(data._____(2,_____);
```

9.3.8

The **substring** method has also a second form. In the case when we input only one parameter it will return a substring from the given position till the end of the string.

```
String data = "Armagedon";
String subS = data.substring(4); // will contain substring
beginning on the position 4, i.e. "gedon"
```

9.3.9

Fill in the code so that the final string does not contain the first character of the original one:

```
String data = "Winter";
```

```
String subS = data._____(_____);  // will contain the
substring "inter"
```

9.3.10

While the data from primitive types are compared using **==**, in the case of reference variables is this not possible because its content represents a reference (link) to a place in the memory.

Comparison of the string is done using the method **equals** that is used through the following notation:

```
str1.equals(str2)
```

where the result of the method is **true** when the variables **str1** and **str2** contain a similar string, e.g.:

```
String a = "mama";
String b = "papa";
if (a.equals(b))
  System.out.println("simmilar");
else
  System.out.println("different");
```

The following notation is possible

```
"mama".equals("papa")
```

or any other alternative ones.

9.3.11

Fill in the code comparing two strings from the input:

```
import java.util.Scanner;

public class App {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        String a = input._____();
        String b = input._____();
        if (a._____(_____))
              System.out.println("similar");
```

```
        else
                System.out.println("not similar");
   }
}
```

- nextInt
- nextInt
- nextLine
- a
- b
- equals
- nextLine

9.3.12

Two strings that differ only by the lowercase or uppercase are not equal, e.g. „mama“ and „Mama“.

But because many people take this kind of strings as similar, we can use a method that ignores the lowercase or uppercase characters and take the strings as similar if they differ only by the uppercase or lowercase.

This alternative is possible through method **equalsIgnoreCase()** used in the following manner:

```
String a = "mama";
String b = "MamA";
if (a.equalsIgnoreCase(b))
 ...
```

that returns in this case **true**.

9.3.13

Fill in the code with the correct methods and variables:

```
import java.util.Scanner;

public class App {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        String a = input._____();
        String b = input._____();
```

```
        if (a._____(_____))  System.out.println("similar
content");
        if (a._____(_____))  System.out.println("similar also
the lowercase and uppercase letters");
  }
}
```

9.4 Basic strings (programs)

9.4.1 Greeting

Write the code that for the name entered at the input, of *String* type, print a greeting in the form of "Hello" space name.

```
Input : Peter
Output: Hello Peter
```

```
Input : Anna
Output: Hello Anna
```

9.4.2 Greeting II.

Write the code that after entering a salutation, first name and last name in a row below it, prints a sentence that begins with a greeting, adds a comma, and addresses the user in the first name, last name.

```
Input : Good day
Peter
Carrot
Output: Good day, Peter Carrot
```

```
Input : Hello
Anna
Soul
Output: Hello, Anna Soul
```

9.4.3 Number of characters

Write a code that prints the number of characters it contains for the specified string.

```
Input : MAMA
Output: 4
```

9.4.4 Palindrome

Write a code that will reverse the string of characters (String). It should take the following chars from the first string and put each of them to the beginning of the second one. The input contains the string and the output the original string, the reversed string, and the value **true** if they are identical or **false** if not.

```
Input : ABBA
Output: ABBA ABBA true
```

```
Input : program
Output: program margorp false
```

9.5 String and number

9.5.1

So far, we have used numerical and text (string) data types. The conversion of values between the data types is called conversion or retype.

Retype was done by noting the type to which we want the value to convert to, in brackets, e.g.:

```
double c = 10.5;
int d = (int) c;
```

This notation can be used in the case of numerical types but in the case of string conversion to number, it is not possible.

Conversion of numbers to strings is very easy, we use it by printing. On a basis, it is enough to add the symbol "+" to any value of the **String** type. If the text string is empty and we add to it a value, then the result is the original value converted to a string.

```
int a = 7;
String a_conv = "" + a;
```

The result is the **string** "7".

9.5.2

What is the result of the following code statements?

```
String a = "a";
int b = 10;
String x = a + b;
System.out.println(x);
```

9.5.3

Conversion of string to integral number is more complicated. A method **parseInt()** from the **Integer** package is used. The input is text, and the result is the corresponding numerical value:

```
String text = "15";
int a = Integer.parseInt(text); // conversion to number
```

In the case the used string does not contain a numerical value, the execution ends with the error that can stop the program.

9.5.4

Fill in the code so that it prints the math addition of numbers:

```
import java.util.Scanner;

public class App {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        String a = input.nextLine();
        String b = input.nextLine();
        int x = _____._____(a);
        int y = _____._____(b);
       System.out.println(x + _____);
   }
}
```

9.5.5

The previous tasks showed that we can combine **String** and **int** type variables (or other primitive data types). Evaluation of expression created by the combination of **String** type variable and other variables is done based on math principles. First is evaluated the expression in brackets, then *, / and at last +, -. In the case of operation of similar priority, it is taken from left to right.

E.g.

```
String a = "data" + 10 + 5 * 7;
```

will the variable **a** contain value data1035.

The reason is evaluation of product and then it is taken from left to right:

- in the first step is string "data" connected with the numerical value - so the result is the string "data10"
- then is added the numerical value: "data10" + 35, where the result is the connection of the original string with a new value changed to the string: "data1035"

Another approach is used in the case of the following code:

```
String a = "data" + (10 + 5 * 7);
```

where is firstly dealt with the product (35), then with the expression in brackets (45) and finally follows the connection of string and numerical value ("data45").

9.5.6

What will be the output of the following code?

```
String a = "data" + 10 + ("data" + 5 * 7);
```

String II.

Chapter 10

10.1 Working with strings

10.1.1

Often we need to browse the given text.

Find out how many times is the digit 3 repeated in the given string. E.g. in string "353253593", it is 4 times.

The approach of browsing through strings is based on browsing separate characters of the string and their processing.

In this case, it means that:

- first, we have to find out the number of characters of the given string (number)
- then we will go through the string using the loop from the first to the last character
- each character will be read, e.g. using the **charAt()** method
- we compare it with the searched character and if it is equal, we increase the number of its occurrences
- at last, we print the result

Character 3 is placed in apostrophes (it is a character, not a string).

```
Scanner input = new Scanner(System.in);
String text = input.nextLine();
int len = text.length();
int count = 0;                    // counts the occurences of
3, at start its 0
for(int i = 0;i < len; i++) {   // we browse through
characters
    if (text.charAt(i) == '3') {  // we take the character on
the i-th position and compare it with 3
        count++;
    }
}
System.out.println(count);
```

10.1.2

Fill in the code that finds out if the string contains the character **B**.

```
Scanner input = new Scanner(System.in);
String text = input.nextLine();
```

```
int len = text._____();
boolean is_there = false;
for(int i = 0;i < len; i++) {
    if (text._____(i) == '_____') {
       is_there= _____;
       break;
    }
}
if (is_there)
   System.out.println("it's there");
else
   System.out.println("it's not there");
```

- false
- b
- size
- char
- length
- B
- substring
- true
- charAt

📖 10.1.3

We can solve the tasks also using the **substring()** method.

In this case, we read the character on the i-th position to a **String** type variable and compare it with the searched character using the **equals()** method.

Because we work with the value 3 as with a string, we place it into quotes:

```
Scanner input = new Scanner(System.in);
String text = input.nextLine();
int len = text.length();
int count = 0;                    // counts the occurence of 3,
at start it is 0
for(int i = 0; i < len; i++) {    // we browse through
characters
    String digit = text.substring(i,i+1); // it takes the
string on the i-th position
    if (digit.equals("3")) {              // it compares the
variable digit with the string "3"
        count++;
```

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

Both solutions are correct and differ only in the usage of tools.

10.1.4

Fill in the code that finds out how many times the string contains the character **D**.

```
Scanner input = new Scanner(System.in);
String text = input.nextLine();
int len = text.length();
int count = 0;
for(int i = 0; i < len; i++) {
    String character = text.substring(_____,_____);
    if (character._____(_____D_____))
      count++;
}
System.out.println(count);
```

10.1.5

Find in the given number the maximum digit. E.g. for 784541, it will be 8.

Because of that, the number of digits is in case of using (and reading) the **int** data type very restricted, we will use the **String** type to read long numbers.

The approach will be simple:

- as the maximum digit will be chosen the smallest possible - 0,
- we will be reading the values at each position of the string (from beginning to end) and compare them with the actual maximum value
- string (or digit) will be always converted to a number and so we can find out if it is bigger than the actual - if yes we remember it in the variable **mx**

```
Scanner input = new Scanner(System.in);
String text = input .nextLine();
String character;
int len = 0, mx = 0;
int digit;
```

```
len = text.length();
for(int i = 0;i < len; i++) {
 character = text.substring(i, i + 1); // we read the string -
character on the i-th position
 digit = Integer.parseInt(character);  // we convert the text
to number
 if (mx < digit) {                    // if the actual digit is
bigger than the original one
     mx = digit;                      // we save it
 }
}
System.out.println(mx);
```

10.1.6

Fill in the code to find the biggest digit in the number.

```
Scanner input = new Scanner(System.in);
String text = input.nextLine();
int x = _____;
for(int i = 0; i < text._____(); i++) {
  String character = text.substring(_____, _____);
  int digit = Integer._____(character);
  if (digit _____ x)
    x = digit;
 }

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

10.1.7

Find out the digit sum of the given long number, e.g. for 4532187 will be the digit sum: *4+5+3+2+1+8+7 = 30*.

Once again we have to:

- browse each digit of the number - we use loop to go from first to last string position
- in the loop we read the character on the i-th position (from *i* to *i+1*)
- convert it to a number
- and add it to the variable **sum**

```
Scanner input = new Scanner(System.in);
String text = input.nextLine();
int sum = 0;
for(int i = 0; i < text.length(); i++) {
  int digit = Integer.parseInt(text.substring(i,i+1));
  sum = sum + digit;
}
System.out.println(sum);
```

10.1.8

Fill in the code that finds out the digit product of the given long number, e.g. for 4532187 will be the digit product: *4*5*3*2*1*8*7 = 6720*.

```
Scanner input = new Scanner(System.in);
String text = input.nextLine();
int product = _____;
for(int i = 0; i < text._____(); i++) {
 _____ digit = _____.parseInt(text.substring(i,i+1));
 product = product _____ digit;
}
System.out.println(product);
```

- len
- String
- *
- 0
- length
- Int
- 1
- Integer
- int

10.1.9

Write a code that will create a mirror image of the given text, e.g.:

```
Mama -> amaM
winter -> retniw
```

etc.

There are more solutions but for our need, we create a variable where we input the characters so that the next character will be put before the existing one, e.g. for the word Aladin we will do the following:

- first, we read **A** and save it to result (res = "A")
- then read **l** and save it before the result (res = "l" + res, i.e. "lA")
- read **a** and save it before the result (res = "a" + res, i.e. "alA") etc.

The code will be following:

```
Scanner input = new Scanner(System.in);
String text = input.nextLine();
String res = "";
for(int i = 0; i < text.length(); i++) {
     String character = text.substring(i,i+1);
     res = character + res; // the character is put before the
created string
}
System.out.print(res);
```

10.1.10

Fill in the code that will print the text string backwards (in one row):

```
Scanner input = new Scanner(System.in);
String text = input.nextLine();
for(int i = text.length()-1; i _____ 0; i_____) {
     char character = text._____(i);
     System.out._____(character);
}
```

10.2 More functions

10.2.1

Working with digits of integral numbers is many times solved using maths where we obtain digits based on integral division:

E.g. for **a = 251** is valid:

if we want to obtain the last digit using maths, we have to:

```
digit = a % 10; // remainder after the division by 10 is 1
```

if we want to obtain the number without the last digit then we have to divide it by 10

```
digit = a / 10; // result after division by 10 is 25
```

This solution is questionable and case-specific - if you want to obtain specific digits of number it is faster and more understandable to use the **String** type.

10.2.2

Fill in the code so that the given number is printed backwards

```
Scanner input = new Scanner(System.in);
int num = input._____(); // read the integral number
while (num _____ 0) {           // until the number contains
any digit
  int digit = num _____ 10;     // get the last digit
  System.out.print(digit);            // print it
  num = num _____ 10;         // remove the last digit from the
number
}
```

10.2.3

The identity of strings can be obtained using the **equals()** method. This method does not say anything about that what string is alphabetically bigger or smaller.

To determine the lexicographic (alphabetical) comparison, the **compareTo()** method is used, which uses a Unicode character table and for two variables **s1** and **s2** it is following:

```
s1.compareTo(s2);
```

and returns the following results:

- if **s1>s2**, then it returns a positive number
- if **s1<s2**, then it returns a negative number
- if **s1==s2**, then it returns 0

The number value represents the distance of the characters on the first position that the string differs in Unicode table, e.g.:

```
String s1 = "Aladin", s2 = "Jasmina";
System.out.println(s1.compareTo(s2));
```

returns the value -9 that represents that **s1** is in the alphabet before the **s2** and the position of the first different characters ("A" and "J") are in the Unicode table between each other by 9 positions.

```
String s1 = "Aladin", s2 = "Amadeus";
System.out.println(s1.compareTo(s2));
```

returns the value -1 that represents that **s1** is in the alphabet before the **s2** and the position of the first different characters ("l" and "m") are in the Unicode table between each other by 1 position.

> Simillary as the **equals()** method is also for the **compareTo()** method possible to ignore the lowercase and uppercase letters: **compareToIgnoreCase()**.

10.2.4

What is the result of the following code?

```
String a = "Dingo", b = "Bingo";
System.out.println(a.compareTo(b));
```

10.2.5

The occurrence of the substring in the existing string is verified by the **indexOf()** method and returns the position where the substring is placed.

```
String text = "Wolfgang Amadeus Mozart";
int pos = text.indexOf("ga");
```

The variable **pos** will contain the value 4 because the 4th position was first found at the beginning of the searched substring.

In case that the searched substring is not found in the string, it returns the value -1. This can be used to notify the user.

```
String text = "Wolfgang Amadeus Mozart";
int pos = text.indexOf("ba");
if (pos == -1)
  System.out.println("Substring was not found.");
else
  System.out.println("Substring beginns at position " + pos +
".");
```

10.2.6

What is the result of the following code?

```
String a = "Dingo", b = "ing";
System.out.println(b.indexOf(a));
```

10.2.7

The string can be browsed also from the end using the method **lastIndexOf()** that returns the last occurrence of the substring:

```
String text = "Wolfgang Amadeus Mozart";
int pos = text.lastIndexOf("a");
```

It returns the position of the last occurrence of character "a" that is in this case 20.

10.2.8

What will be the output of the following code:

```
String text = "Victor Igor Hugo";
int pos = text.lastIndexOf("go");
System.out.println(pos);
```

10.2.9

We can also browse the string from a given position using the variation of **indexOf()** with two parameters where the second one defines the position from where we have to start to browse.

```
String text = "Wolfgang Amadeus Mozart";
int poz = text.indexOf("a", 10);
```

it returns 11 that is the first position of "a" from position 10.

10.2.10

What will be the output of the following code:

```
String text = "Wolfgang Amadeus Mozart";
```

```
int pos = text.indexOf("g", 5);
System.out.println(pos);
```

10.3 Numbers in strings (programs)

10.3.1 Number of digit occurrences

Write a program to find out how many times the number 3 repeats in the given string.

```
Input : 57,33
Output: 2
```

```
Input : OlfeK,.3fe8
Output: 1
```

10.3.2 Occurrences of zero

Write the code that will find out if there is a zero in the specified string, if so, it will write "Zero is here", otherwise "Zero is not here".

```
Input : 976a
Output: Zero is not here
```

```
Input : 8Ddí970d8
Output: Zero is here
```

```
Input : Afé0
Output: Zero is here
```

10.3.3 Digit sum

Write the code that returns the sum of the digits of the number you entered.

```
Input : 123
Output: 6
```

```
Input : 0124
Output: 7
```

```
Input : 0
Output: 0
```

10.3.4 Maximum digit

Write the code that finds the maximum digit of the entered number.

```
Input : 5787
Output: 8
```

```
Input : 311
Output: 3
```

10.3.5 Even digits in string

Write the code to find out how many even digits are in the specified string and whether there is a zero. Print "Number of even:" count of even digits, on the console. In a new line, if there is 0 in the string, then "Zero is here" otherwise "Zero is not here".

```
Input : 98
Output:
Number of even: 1
Zero is not here
```

```
Input : 09a
Output:
Number of even: 1
Zero is here
```

10.3.6 Number correction

Write the code that changes all non-numeric characters to 1 in the specified string and prints the changed string to the console.

```
Input : 57ada87
Output: 5711187
```

```
Input : 3.,úôéáá23Â§ô!3
Output: 311111112311113
```

10.3.7 Sorting without an array

Write a code that will sort the data placed in five variables (without using an array). The input contains five integer numbers. Print these numbers in ascending order. If there are not 5 numbers on the input, write -1.

```
Input : 2 1 4 3 6
Output: 1 2 3 4 6
```

```
Input : 5 5 4
Output: -1
```

10.4 Working with text (programs)

10.4.1 String comparison

Write the code to see if two strings specified on separate lines are identical. If they are identical, write "yes" otherwise write "no".

```
Input: Mom
mom
Output: no
```

```
Input: daddy
daddy
Output: yes
```

10.4.2 List of vowels

Write the code that will print all the vowels (a, e, i, o, u, y) according to Slovak grammar.

```
Input : ahoj
Output: ao
```

```
Input : mama isla do mesta
Output: aaiaoea
```

10.4.3 Uppercase

Type a program that prints all characters of the specified string to uppercase in the console.

```
Input : car
Output: CAR
```

```
Input : HeLlo
Output: HELLO
```

10.4.4 Char in the string

Write the code that reads the string and the char character at the input to determine whether the char is in the string. The result will be "Yes" or "No".

```
Input :
Hello
h
Output: No
```

```
Input :
Hello
H
Output: Yes
```

```
Input :
John
o
Output: Yes
```

10.4.5 Mirror

Write a code that will mirror the given string.

```
Input : john
Output: nhoj
```

```
Input : 124
Output: 421
```

```
Input : a
Output: a
```

10.4.6 Occurrence and replace

Write the code to see if "y" is in the given word. If so find out how many times it is there and replace it with "i". Print the number of occurrences on the console and print the modified string in a new row.

```
Input : Byeli
Output: 1
Bieli
```

```
Input : Tree
Output: 0
Tree
```

10.4.7 Initials

Write the code that writes the initials for the given name and surname, for example for Joseph Carrot print "JC".

```
Input :
Joseph
Carrot
Output: JC
```

```
Input :
Anna
Soul
Output: AS
```

10.4.8 Delete digits

Write a code to replace the digits with dashes in the given string. Letters remain unchanged.

```
Input : Hello123
Output: Hello---
```

```
Input : 123
Output: ---
```

```
Input : hello 0john
Output: hello -john
```

10.4.9 Remove spaces

Write the code to remove the spaces from the given string.

```
Input : Hello Peter
Output: HelloPeter
```

```
Input : bye   bye   bye
Output: byebyebye
```

10.5 Advanced operations with text (programs)

10.5.1 Change character with counting of changes

Write the code that changes all semicolons to commas in the given string, lists how many times the change was made and prints the changed string. The statement will be following:

```
Input : abc;5325;543;55
Output: 3 abc,5325,543,55
```

```
Input : Hi;Hello;Bye
Output: 2 Hi,Hello,Bye
```

10.5.2 The number of occurrences of a substring

Write the code to find out how many times the specified string is in another specified string. The searched string (substring) is given first, the text to search in is given in a new line. The output is the number of occurrences of the substring.

```
Input :
car
carpool tree car
Output: 2
```

```
Input :
Hello
Anička Hello How are you, hello
Output: 1
```

10.5.3 Remove words

Write the code that removes all the words "hello" from the input string. The output is a changed string.

```
Input : HihelloPeter
Output: HiPeter
```

```
Input : Hellohello
Output: Hello
```

10.5.4 Average Word Length

Write a code that will compute the average length of words separated by any number of spaces read from the input. Input is the line of text. Print the average length of the words.

```
Input : The  average   length  of words is 4.0
Output: 4.0
```

```
Input : Java
Output: 4.0
```

```
Input :
Output: 0.0
```

10.5.5 Ones and Zeros

Write the code that will check if a given series of integer numbers contains the same number of zeroes and ones. Input integer numbers. If the read number is not equal to 0 or 1 stop reading and print *true* if the number of zeroes is equal to the number of ones and *false* otherwise.

```
Input : 0 0 1 1 0 1 0 0 0 0 0 1 1 1 1 1 99
Output: true
```

```
Input : 0 1 1 8
Output: false
```

10.5.6 ZerosAndOnes II.

Write a program that will check if a given string of characters contains the same number of zeroes and ones. The input is a string of characters.

Print "*true*" if the number of zeroes is equal to the number of ones and "*false*" otherwise.

If the string contains any other character than 0 or 1 print "*error*" (skip the space characters - neither count them nor treat them as wrong characters).

```
Input : 00110  10000011111
Output: true
```

```
Input : 011
Output: false
```

10.5.7 Decompress

Write the code that will decompress a string of characters. The compressed version of the string consists of pairs <counter><character> separated by the comma (e.g. 5a,10b). Input the compressed string. Print the string after decompression.

```
Input : 5a,10b
Output: aaaaabbbbbbbbbb
```

```
Input : 1a,2 ,33
Output: a  333
```

Nested Loops and Effectivity

Chapter 11

11.1 Nested loops

11.1.1

Many tasks can be solved using one loop, but it is not extraordinary if the solution needs to use a loop in the body of another loop. The inside loop is called a **nested loop**.

It has the following form:

```
for(int i = 1; i < 10; i++ ) {
   for(int j = 1; j < 5; j++ ) {
      command;
    }
}
```

The combination of multiple loops with a known number of iterations has a small issue where you have to take into account that the control variables have to have **different names**.

11.1.2

How is named the loop placed inside another loop?

- nested
- internal
- subloop
- hybrid

11.1.3

Write a code that will print to the first row one character 1, to the second row two characters 2, etc. till 9.

```
1
22
333
4444
55555
666666
7777777
88888888
```

```
999999999
```

The solution of the task needs two different loops:

- in the first loop, we change the digit that will be printed.
- in the second loop, we take this digit and print it and the number of outputs is similar to the value we print.

This leads to the following loop:

```
for(int i = 1; i <= 9; i++) {   // goes from 1 to 9
  for(int j = 1; j <= i; j++) { // this row makes it possible
to repeat the output i-times
     System.out.print(i);       // this prints the value
specified in the first loop
  }
  System.out.println();         // after printing all of the
digits we move to a new row
}
```

11.1.4

Fill in the code so that for the given **n** will write a rectangle of stars.

```
Scanner input = new Scanner(System.in);
int n = input.nextInt();
for(int i = 1; i <= _____; i++) {
  for(int j = 1; j<= _____; j++) {
     System.out._____("*");
  }
  System.out._____();
}
```

11.1.5

Write a code that for the given integral values **m** and **n** shows **m** rows below each other and, in each row, will be **n** circles (o).

```
Scanner input = new Scanner(System.in);
int m = input.nextInt();
int n = input.nextInt();
for (int i = 1; i <= m; i++) {
  for (int j = 1; j <= n; j++) {
     System.out.print("o");
```

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

The solution gives the required result but if we look at it in more detail, we see that in the nested loop we do always the same action - we always print the same time's character "o".

This operation can be simplified by preparing the whole row (putting it into a text variable) and its printing - in one step we would print the whole row.

The modified code would be following:

```
Scanner input = new Scanner(System.in);
int m = input.nextInt();
int n = input.nextInt();
String row = "";
// we fill in the variable row with n characters
for (int i = 1; i <= n; i++) row = row + "o";
// m times we output the whole row
for (int i = 1; i <= m; i++) System.out.println(row);
```

The loops are independent, and we can use (don't have to) similar control variables.

In the first case we do the operation in loop **m** x **n** times, in the second case **n**-times we repeat the assignment to the variable and **m**-times the output - the resulting number of operations is **m+n**.

11.1.6

Fill in the code that it is the most effective to create triangles from characters "x" for the given n.

```
x
xx
xxx
xxxx
xxxxx
```

```
Scanner input = new Scanner(System.in);
int n = input.nextInt();
String row = _____;
for (int i = _____; i <= n; i++)  {
```

```
  row = _____ + "x";
  System.out._____(row);
}
```

11.1.7

What will be saved in the variable **sum** after the following code?

```
int sum = 0;
for(int i = 5; i > 2; i--) {
  for (int j = 1; j <= 3; j++ ) {
    sum = sum + i + j;
  }
}
System.out.println(sum);
```

11.1.8

What will be saved in the variable **row** after the following code?

```
String row = "";
for(int i = 1; i < 5; i++) {
  row = "" + i;
  for (int j = 1; j <= 3; j++ ) {
    row = row + j;
  }
}
System.out.println(row);
```

11.2 Simple problems (programs)

11.2.1 Digits sequence

Type the code that writes the number 1 to the first line once, two times number 2 to the second line, and so on up to 9, 9 nine times in the 9th row.

```
Output:
1
22
333
4444
55555
```

```
666666
7777777
88888888
999999999
```

11.2.2 Rectangle of stars

Write the code that displays *m* rows for the specified integer values *m* and *n*, with *n* stars (x) in each row.

```
Input : 2 2
Output:
xx
xx
```

```
Input : 2 5
Output:
xxxxx
xxxxx
```

11.2.3 Triangle of stars

Write the code that reads *n* from the input and displays 1 star in the first line, 2 stars in the second line, 3 stars in the third line ..., *n* stars in the *n*-th line.

```
Input : 6
Output:
x
xx
xxx
xxxx
xxxxx
xxxxxx
```

```
Input : 3
Output:
x
xx
xxx
```

11.2.4 Geometric sequence of stars

Write the code that reads *n* from the input and displays 1 star in the first line, 2 stars in the second line, 4 stars in the third line and in each additional double of the previous star rating.

```
Input : 5
Output:
x
xx
xxxx
xxxxxxxx
xxxxxxxxxxxxxxxx
```

11.2.5 Rectangle frame from stars

Write the code that displays *m* lines with *n* characters to create a rectangle from asterisks represented by the letter x. The inside of the rectangle will be empty, the asterisks will only be on the perimeter.

At the beginning of the output do a line feed. Leave one space at the beginning of the line and between the stars.

```
Input :  5 5
Output:
 x x x x x
 x       x
 x       x
 x       x
 x x x x x
```

11.2.6 Opposite triangle from stars

Write the code that displays for *n* from the input: *n-1* spaces and 1 star in the first row, *n-2* spaces and 2 stars in the second, *i*-th *n-i* spaces and *i*-stars ..., *n*-th row 0 spaces and *n* stars. At the beginning of the output do a line feed.

```
Input : 5
Output:
    x
   xx
  xxx
 xxxx
xxxxx
```

11.2.7 Print to square

Write the code that reads the number *n* from the input and prints numbers from 1 to *n* * *n* so that there are *n* numbers in each row and in each column that together square the square.

Allocate four spaces to list the integer variable.

```
Input : 5
Output:
   1   2   3   4   5
   6   7   8   9  10
  11  12  13  14  15
  16  17  18  19  20
  21  22  23  24  25
```

11.2.8 Median of words

Write the code that will compute the median of words separated by any number of spaces read from the input.

The median should correspond to the lexical order of words. Input the line of text. Print the median.

```
Input : Write a code that will compute the median
Output: median
```

```
Input : 1 3 2 5 4 7 6
Output: 4
```

11.2.9 Median of word length

Write the code that will compute the median of words separated by any number of spaces read from the input.

The median should correspond to the length of words and all the words should be of different lengths or if not then print *error*. Input the line of text. Print the median.

```
Input : A code computes the correct median
Output: code
```

```
Input : 1 3 2 5 4 7 6
Output: error
```

11.2.10 Compression

Type the code that compresses the specified character string. Specifies the character first and then the number of occurrences of consecutive characters.

Print the list of pairs: character and the number representing the length of the sequence of its occurrences, separated by the colon.

```
Input : 122333444455555444
Output: 1:1 2:2 3:3 4:4 5:5 4:3
```

```
Input : aaaaabbbbbbb  ooo
Output: a:5 b:7  :2 o:3
```

11.3 Advanced problems (programs)

11.3.1 Small multiplication table

Write the code that writes a small multiplication table (from 1x1 to 10x10), allocating 4 spaces for each number.

```
Output:
   1   2   3   4   5   6   7   8   9  10
   2   4   6   8  10  12  14  16  18  20
   3   6   9  12  15  18  21  24  27  30
   4   8  12  16  20  24  28  32  36  40
   5  10  15  20  25  30  35  40  45  50
   6  12  18  24  30  36  42  48  54  60
   7  14  21  28  35  42  49  56  63  70
   8  16  24  32  40  48  56  64  72  80
   9  18  27  36  45  54  63  72  81  90
  10  20  30  40  50  60  70  80  90 100
```

11.3.2 The character with the highest occurrence

Write the code that finds the character that is most common in the specified lowercase string. If the number of occurrences is the same for all characters, print the first small character in alphabet to the console.

```
Input : jajaj
Output: j
```

```
Input : Hello
Output: l
```

```
Input : hello
Output: l
```

11.3.3 Sum of numbers in the string

Write the code that calculates the sum of integers occurring in the string.

```
Input : We have 12 hens at home, 54 geese and 3 ducks.
Output: 69
```

```
Input : 12.3,8 9
Output: 32
```

11.3.4 Reduction of fraction

Write the code for the fraction entered by two values in the order of the *numerator*, *denominator*, write its reduction form *n* / *d*.

```
Input : 10 8
Output: 5/4
```

```
Input : 4 12
Output: 1/3
```

11.3.5 Prime numbers from 2 to n

Write a program that lists all prime numbers from 1 to *n* for the specified number *n*.

```
Input : 10
Output:
2
3
5
7
```

11.4 Repair programs (programs)

11.4.1 Power

Repair the code to return a^a for the specified value *a*, greater than zero.

```
Input : 3
Output: 27
```

```
Input : 1
Output: 1
```

JavaApp.java

```
import java.util.Scanner;

public class JavaApp {

    public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      int a, power;

      a = input.nextInt();
      power = 1;
      do {
         power = power * a;
         a = a - 1;
      } while (a > 0);

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

11.4.2 Stars

Repair the code to write *n* stars to the console.

```
Input : 6
Output: xxxxxx
```

```
Input : 1
Output: x
```

JavaApp.java

```
import java.util.Scanner;
```

```
public class JavaApp {

    public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      int n = input.nextInt();
      String stars = "x";
      for(int i=1; i<|=n; i++);
         stars +="x";
      System.out.println(stars);
    }
}
```

11.4.3 Divisors

Repair the code to find all the divisors and their number for the specified number and write them to the console. Print the divisors from the largest to the smallest, line feed and list their number.

```
Input : 2
Output:
21
2
```

```
Input : 6
Output:
6321
4
```

JavaApp.java

```
import java.util.Scanner;

public class JavaApp {

    public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      int a = input.nextInt();
      int count = 0;
      int i = a;
      while (i > 0) {
        i = i - 1;
        if ((a % i) == 0) {
         count = count + 1;
```

```
          System.out.print(i);
        }
      }
      System.out.println();
      System.out.println(count);
    }

}
```

11.4.4 Factorial

Repair the code to calculate the factorial for the given number *n* (for n <17).

```
Input : 5
Output: 120
```

```
Input : 4
Output: 24
```

JavaApp.java

```
import java.util.Scanner;

public class JavaApp {

    public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      int n = input.nextInt();
      int factorial = 1;
      for(int i = n; i>=0; i--)
        factorial = factorial * i;
      System.out.println(factorial);
    }
}
```

11.4.5 Power II

Repair the code to calculate the power of ab for the positive integers a, b. Use the variable type with the largest range.

```
Input : 3 3
Output: 27
```

```
Input : 2 5
```

```
Output: 32
```

JavaApp.java

```
import java.util.Scanner;

public class JavaApp {

    public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      int a = input.nextInt();
      int b = input.nextInt();
      int power = 0;

      for(int i = 1; i <| b; i++)
        power = power * a;

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

11.4.6 Sum of digits

Repair the code to print the digits of the integer input. The program works correctly eg. for number 2 586 but does not work for 3 108. Find the reason and secure the remedy.

```
Input : 12
Output: 3
```

```
Input : 2586
Output: 21
```

JavaApp.java

```
import java.util.Scanner;

public class JavaApp {

    public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      int c, n = input.nextInt();
      int sum = 0;
      do {
        c = n % 10;
        sum = sum + c;
```

```
        n = n / 10;
      } while (c > 0);
      System.out.println(sum);
    }
}
```

11.4.7 Remove spaces

Repair the code to remove spaces in the specified string.

```
Input : Hi Peter
Output: HiPeter
```

```
Input : bye   bye   bye
Output: byebyebye
```

JavaApp.java

```
import java.util.Scanner;

public class JavaApp {

    public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      String str = input.nextLine();
      int position;

      while (str.contains(" ")) {
        position = str.indexOf(" ");  // return space position
        str = str.substring(0, position)
                + str.substring(position);
      }
      System.out.println(str);
    }
}
```

11.4.8 Palindrome

Repair the code to see if the palindrome string is specified. If so, write "It is a palindrome" on the console, otherwise, it will write "It is not a palindrome".

```
Input : kayak
Output: It is a palindrome
```

```
Input : hello
Output: It is not a palindrome
```

JavaApp.java

```
import java.util.Scanner;
public class JavaApp {

    public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      String str = input.nextLine();

      str = str.toLowerCase();     // shift to lowercase

      while (str.contains(" ")) {        // if there is a
space
        int position = str.indexOf(" "); // return space

      position
        str = str.substring(0,position)  // remove space on

      position
                  + str.substring(position+1);
      }

      String endstr = "";
      for(int i=str.length(); i<|=1; i--) // go trouhgt string

      from end
         endstr = endstr + str.substring(i,i+1);  // save

      if(str.equals(endstr))
          System.out.println("It is a palindrome");
        else
          System.out.println("It is not a palindrome");
    }
}
```

11.4.9 Triangle

Repair the code to display *n-1* spaces and 1 star in the first row, *n-2* spaces and 2 stars in the second row, *n* spaces and *i* stars in the *i*-row, 0 spaces in the *n*-row and stars.

```
Input : 5
Output:
    x
   xx
  xxx
 xxxx
xxxxx
```

JavaApp.java

```
import java.util.Scanner;

public class JavaApp {

    public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      int n = input.nextInt();

      String row = "";
      for(int i = 1; i <|= n; i++) {  // the cycle provides
                                                          //a
print of n rows
       for(int j = 1; j <|= n; j++)   // the cycle fills the
                                                          //row
with stars or spaces
        if(j <|= (n-i))
          row = row + "x";
        else
          row = row + " ";
        System.out.println(row);      // insert created row
      }
    }
}
```

11.4.10 Mirror

Repair the code so that it will mirror the specified string.

```
Input : john
Output: nhoj
```

```
Input : 124
Output: 421
```

```
Input : a
Output: a
```

JavaApp.java

```
import java.util.Scanner;

public class JavaApp {

    public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      String str = input.nextLine();
      String mirror = " ";        // insert an empty string =
                                                    nothing
into the variable
      for(int i = 1; i<|= str.length()-1; i++)
       mirror = mirror + str.charAt(i);  // i-th character
read

            before

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

Multiple Conditionals

Chapter 12

12.1 Command switch

12.1.1

Usually is for conditionals used the **if** structure that by one usage allows us to difference maximum 2 situations (if-else).

But in practice is sometimes used structure that allows multiple conditionals (branches) and allows to define different behavior of the program for the list of values.

It has the following structure:

```
switch (variable) {

        case 1: command1;
                command2;
                break;
        case 2: command1;
                command2;
                break;...
       default: command1;
                command2;
                break;...
}
```

For each possibility is using the **case** command defined a label that is searched by the control structure based on the content of the variable defined in the **switch** part.

Special status has the **default** label that contains commands that are executed when none of the above cases occurs.

Switch controls the value of the variable and compares it with the values placed after the **case**. If the value after **case** and the value of the variable are equal then it starts to execute the commands. The **case** commands do not add another block or command but only help to find a place where the switch has to start working based on the content of the variable.

12.1.2

What statements are used to represent the label?

- case
- default

- switch
- break
- else

12.1.3

Write the structure of the multiple conditionals that evaluates the test based on the achieved points. Print for

- 10 points – „excellent“
- 9 points – „good job“
- 8 points – „study more“
- fewer points – „weak“.

```
Scanner input = new Scanner(System.in);
String points = input.nextInt();
switch(points) {
      case 10: System.out.println("Excellent!");
               break;
      case 9 : System.out.println("Good job!“);
               break;
      case 8 : System.out.println("Study more!" );
               break;
      default: System.out.println("Weak...");
}
```

In the case that the variable **points** have another value as the named (10, 9, 8) then are executed the commands in the **default** part.

The **break** command has the task to ensure that it jumps out of the **switch** structure and continue with the code placed after the **switch** block.

Without the **break** commands would the execution continue through another command till the end of the **switch**.

```
switch(points) {
      case 10: System.out.println("Excelent!");
      case 9 : System.out.println("Good job!“);
      case 8 : System.out.println("Study more!" );
      default: System.out.println("Weak...");
}
```

In our case it would mean e.g. for the 9 points student:

- setting on case 9
- printing „Good job!“
- printing "Study more!"
- printing "Weak..."

12.1.4

What is the output of the following code?

```
  int op = 3;
  int res  = 5;
  switch (op)   {
        case 1: res++;
                break;
        case 2: res--;
        case 3: res = res + 2;
        case 4: res = res + 3;
                break;
        case 5: res = res - 2;
        case 6: res = res - 3;
                break;
        default: res = res + 4;
    }
    System.out.println(res);
```

12.1.5

For the month number write the number of its days. Do not take into account the leap year.

To do the same commands for more values of the variable we structure the **switch** following:

```
switch (variable) {
         case value1:
         case value2:
         case value3:
         case value4: command1;
                        break;
         case value5:
         case value6: command2;
                        break;
         default:       command3;
```

```
                        break;
}
```

The last **break** command is not necessary because the **switch** will end either way at that point. However, we recommend using the **break** statement to prevent an error that can be created by following the code edition.

The solution of the assignment will be following:

```
int month = 2;
switch (month) {
         case 1:
         case 3:
         case 5:
         case 7:
         case 8:
         case 10:
         case 12: System.out.println("31");
                  break;
         case 4:
         case 6:
         case 9:
         case 11: System.out.println("30");
                  break;
         case 2:  System.out.println("28");
                  break;
        default:  System.out.println("Wrong input month.");
                  break;
}
```

12.1.6

Fill in the code so that it does the corresponding operation with the variables **a** and **b** based on the given operator.

```
Scanner input = new Scanner(System.in);
String operator = input.nextLine();
int a = 10, b = 3;
_____ (operator) {
   _____ "+" : System.out.println(a + b);
           break;
   _____ "-" : System.out.println(a - b);
           break;
   _____ "*" : System.out.println(a * b);
```

```
            break;
   _____ "%" : System.out.println(a % b);
            break;
   _____ : System.out.println("undefined operation");
            break;
}
```

📖 12.1.7

The **break** statement will stop the execution of commands inside of the following structures:

- switch
- for
- while
- do-while

After its execution is continued with the command placed after the stopped structure.

We do not recommend substituting the loop condition and "force" the loop to end this way, however, it is a good option in case of reading foreign source codes.

E.g.:

Does the text string contain a given character?

In this task, it's enough to find the first occurrence of the character and end with the search. If the character is found we set the **boolean** variable to **true** and the command **break** will interrupt the **for** loop and skip after the loop end to print the output at the end of the program.

If the searched character is not found the loop will come to an end and the variable will stay with the value **false**.

```
char search = 'a';
String txt = "Mama has Ema";
boolean found = false;
int i;
for (i = 0; i < txt.length(); i++) {
     if (txt.charAt(i) == search) { // if the searched
character is on the given position
         found = true;
         break;
     }
}
```

```
if (found) {
  System.out.println("It is on position " + i);
} else {
  System.out.println("Not found");
}
```

12.1.8

What is the output of the following code?

```
for(int i = 1; i < 10; i++) {
  if (i == 4)
     break;
   else
     System.out.print(i);
}
```

12.1.9

While the **break** command interrupts the loop execution, the **continue** statement ends the loop execution only in the one-step - the loop continues (evaluates the loop condition and increases the value of the control variable if it is the loop with a known number of iterations).

E.g.

```
for(int i = 1; i <= 5; i++) {
  if (i == 4)
     continue;
   else
      System.out.print(i);
}
```

The part of the program will print values 1, 2, 3.

In case the variable **i** has value 4, the command **continue** will move to the next iteration of the loop, increase the value to 5 and continue.

The result will be 1235.

12.1.10

What will be the output of the following code?

```
int i = 0;
while (i < 10) {
  i++;
  if (i % 2 == 0) continue;
  System.out.print(i);
}
```

12.2 Switch (programs)

12.2.1 Number of even digits

Write the code that prints the number of occurrences of each even number (0-9) for the given number.

```
Input : 123
Output: 0-0 2-1 4-0 6-0 8-0
```

```
Input : 12312346
Output: 0-0 2-2 4-1 6-1 8-0
```

12.2.2 Number of days per month (numeric input)

Write the code that prints the count of days representing the given month's numeric designation (do not assume a leap year).

For a wrong input, greater than 12 or less than 1, the program will display "Invalid month".

```
Input : 1
Output: 31
```

```
Input : 0
Output: Invalid month
```

12.2.3 Count of days in month (verbal input)

Write the code that prints the number of days in a month. The month is given in lowercase letters.

```
Input : january
Output: 31
```

```
Input : february
Output: 28
```

12.2.4 Calculator

Write the code that finds the sum, difference, product and division of the two given numbers based on the given mathematical operation (+, -, *, /).

Use the integer division command for quotient and enter the mathematical operation character in the order and the next line of the two numbers.

```
Input :
+
1 2
Output: 3
```

```
Input :
/
6 3
Output: 2
```

12.2.5 Seasons

Write the code that retrieves the number of the given month and prints which season the month belongs to:

- 3 - 5 months: "SPRING"
- 6 - 8 months: "SUMMER"
- 9 - 11 months: "FALL"
- 12, 1, 2 months: "WINTER"

If the input number is less than 1 or greater than 12, "Invalid month" is displayed on the console.

```
Input : 2
Output: WINTER
```

```
Input : 7
Output: SUMMER
```

12.2.6 The number of days left in a month

Write a program that reads two integers representing the *day* and *month* at the input and prints the number of days left in that month.

```
Input : 2 2
Output: 26
```

```
Input : 1 1
Output: 30
```

12.2.7 Age categories

Write the code that reads a number representing the *age* of a person at input and prints whether it is the age for a child (0-11 years), a teenager (12-18 years), a young adult (19-35 years), an adult (36 - 60 years) or a senior (61-99 years).

```
Input : 9
Output: child
```

```
Input : 89
Output: senior
```

12.2.8 Hex Digits

Write the code that will translate hexadecimal digits (lower and uppercase) entered on input to their decimal values.

The input contains a character. If it is the hexadecimal digit print its decimal value else print "-1".

```
Input : A
Output: 10
```

```
Input : x
Output: -1
```

```
Input : b
Output: 11
```

12.2.9 Print Vowels

Write the code that prints only the vowels (a, e, i, y, o, u, small and large) of the given input word using the case structure.

```
Input : Hello Oto
Output: eoOo
```

```
Input : Bye johnny
Output: yeoy
```

12.2.10 Prints without vowels

Write the code that removes the vowels (a, e, i, y, o, u, uppercase and lowercase) from the given word.

```
Input : hello
Output: hll
```

```
Input : mAmA
Output: mm
```

```
Input : mother go to the restaurant
Output: mthrgtthrstrnt
```

12.2.11 Lowercase and uppercase

Write the code to find out how many times there are lowercase letters, how many uppercase letters and how many digits are in the input string.

```
Input : hello123A
Output: lowercase-5 uppercase-1 digit-3
```

```
Input : Hello Peter
Output: lowercase-8 uppercase-2 digit-0
```

Exceptions

Chapter 13

13.1 Exceptions and the treatment

13.1.1

In the case of error will the execution of the program be put into an **exceptional status (exception)** and the program execution will be interrupted.

Because Java is a language that is focused on security, it forces the user to treat all situations where an error can occur that would lead to the program crash. Based on the severity of the errors, the programmer:

- **can** treat some exceptional states (division by zero, conversion of string to number)
- **has to** treat the other (input-output)

13.1.2

The errors in the program are in Java language denoted as:

- exceptions
- errors
- mistakes

13.1.3

The errors can be many times predicted:

```
int a = 1, b = 0;
if (b == 0)
   System.out.println("no division");
else
   System.out.println("division: " + a / b);
```

We will check if the **b** variable contains a value that would lead to the program crash, if yes, we do not allow the dangerous operation.

13.1.4

Fill in the correct value where in this case it is necessary to prevent the conversion of text to number if it contains other characters than 0-9.

```
Scanner input = new Scanner(System.in);
String txt  = input.nextLine();
boolean isNum = _____;
int num;
for(int i = 0; i < txt.length(); i++) {
   if ((txt.charAt(i) < _____) || (txt.charAt(i) > _____)) {
      isNum = false;
      break;
   }
}
if (_____) {
   num = Integer.parseInt(txt);
} else {
   System.out.println("not integral number")
   num = 0;
}
```

13.1.5

The alternative to check the dangerous inputs into methods is to catch the error because if it happens = a treatment of the program crash.

This exceptional situation (exception) is treated by the block **try** – **catch** – **finally**

- **try** – begins the commands block where can the error happen
- **catch** (catch the error) – ends the commands block and also notes what has to be done if the error occurs (e.g. output info)
- **finally** – code that should be executed if the exception occurs and also if not (e.g. releasing the memory), this code block is optional

```
try {
  // commands that can cause an error
}
  // it can happen that the treated code can generate various
  // exceptions on different places, so to various errors we
can react different
catch (exceptionType1 variable1) {
 // in the case the exception of type 1 is generated,
 // you put here the commands that have to be executed
}
catch (exceptionType2 variable2) {
 // the commands executed for the exception of type 2
}
```

```
finally {
 // commands that will be executed always
}
```

E.g. catching the exception of division by zero will be following:

```
int a = 1, b = 0;
try {
   int c = a / b;
   System.out.println(c);
} catch (ArithmeticException e) {
   System.out.println("division by zero");
}
```

In the case, that by the division occurs an error,

```
  int c = a / b;
```

will be the program interrupted and continue in the **catch** part that is defined for the treatment of the division by zero (i.e. the output of the **c** variable will not be done).

13.1.6

Fill in the code to catch the error by the conversion of text to number:

```
int i;
Scanner input = new Scanner(System.in);
_____ s  = input.nextLine();
_____ {
  i = _____.parseInt(s);
}  _____ (NumberFormatException e)  {
      System.out.println("String is not a number!");
}
```

- try
- Integer
- String
- catch

13.1.7

The information about the error occurrence in the **catch** part will be saved to the **e** variable and we can print it out then:

```
int a = 1, b = 0;
try {
  int c = a / b;
  System.out.println(c);
} catch (ArithmeticException e) {
    System.out.println(e);
}
```

In this case, contains the **e** variable the text

```
java.lang.ArithmeticException: / by zero
```

that can be enough for the user or not. It is on the programmer if he/she uses the system message or writes his/her own.

The alternative to print all of the content of the variable is showing the message that contains the text without the notation of the error:

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

prints:

```
/ by zero
```

13.1.8

Fill in the output of the error text that catches the following exception treatment:

```
_____ {
  c = a / b;
}
_____ (ArithmeticException _____) {
  System.out.println(exc);
}
```

13.1.9

One **try** block is used to treat more types of errors by its enumeration into separate **catch** blocks following way:

```
try {
   ...
   i = Integer.parseInt(s1);
   j = Integer.parseInt(s2);
```

```
   division = i / j;
 }
 catch(NumberFormatException e) {
    System.out.println("String is not a number!");
 }
 catch(ArithmeticException e) {
    System.out.println("Division by zero!");
 }
```

If the error occurs during the conversion of text to number it continues in the **NumberFormatException** part, if an error is raised by the division by zero it continues in the **ArithmeticException** part.

Exception types that can occur during the program execution can be found in the **Exceptions** class specification.

13.1.10

Fill in the correct error types:

```
try {
   i = Integer.parseInt(s1);
   j = Integer.parseInt(s2);
   division = i / j;
 }
 catch(_____ e) {
    System.out.println("String is not a number!");
 }
 catch(_____ e) {
    System.out.println("Division by zero!");
 }
```

- ArithmeticException
- NumberAsStringException
- NumberFormatException
- ZeroDivisionException

13.1.11

If we are not sure what errors can occur, it is enough to use a similar code to treat all errors, so we can use the **Exception** type to catch all errors following:

```
int a = 1, b = 0, c = 0;
String str = "102a";
try {
  b = Integer.parseInt(str);
  c = a / b;
  System.out.println(c);
}
catch (Exception e) {
  System.out.println(e.getMessage());
}
```

The **catch** block in this case catches all types of errors and reacts to all of them similar - prints the error text.

13.1.12

Fill in the code to catch any type of error by the input of the data:

```
Scanner input = new Scanner(System.in);
_____ {
   int i  = input.nextInt();
   int j  = input.nextInt();
   System.out.println("division is: " + i / j);
}  _____ (_____ e)  {
        System.out.println("An error occured: " +
e.get_____());
}
```

13.2 Exceptions (programs)

13.2.1 Enter numeric value correctly

Write the code that detects if an integer value was entered correctly by catching an exception. In the case of the correct value, the text "OK" is displayed on the console, in the case of incorrect value the text "Exception" is displayed.

```
Input : -268
Output: OK
```

```
Input : i am 5
Output: Exception
```

13.2.2 Zero division error handling

Write the code that is resistant to zero by dividing two integers entered at the input.

Error catch with a *try* - *catch* block when dividing.

If it is not a division by zero, it will print the result as a decimal number on the console otherwise it will write "Division by zero"

```
Input : 1 2
Output: 0.5
```

```
Input : 105 0
Output: Division by zero
```

```
Input : 5 1
Output: 5.0
```

13.2.3 Error resistant addition

Write the code that is resistant to incorrect values. At the input are given two integers to be added together. Each number is given in a separate line.

If the entry is incorrect print to the console whether the first or second number is incorrect, in the form: "1st number is incorrect" / "2nd number is incorrect".

```
Input : 1
2
Output: 3
```

```
Input : j
5
Output: 1st number is incorrect
```

```
Input : cislo
cislo
Output: 1st number is incorrect
```

```
Input : 5
cislo
Output: 2nd number is incorrect
```

13.2.4 Printing positions with error

Write the code to see if the given integer is spelt correctly. In the case of incorrect input, write to the console the position of the error. If the number is entered correctly, it will display "OK".

```
Input : -1
Output: OK
```

```
Input : -1j00
Output: error at position 2
```

```
Input : 568-1
Output: error at position 3
```

```
Input : 5045
Output: OK
```

Arrays

Chapter 14

14.1 Basic terms

14.1.1

Working with data is often not only a simple calculation. More than 90 % of applications do not work with simple data but with lists. The example of lists are people, invoices, websites, measured values, etc.

We request that we can do the following operations with lists, adding and deleting data, various calculations, sorting, etc.

The simplest list that we have already worked with is **String** - it contained the list of characters ordered into a string that allows reading, adding, deleting, etc.

The access to specific characters of the list was secured through the index:

index/position	0	1	2	3	4	5	6
character	M	a	d	o	n	n	a

14.1.2

Fill in the command to get the 4. character of the string:

```
String str = "Joseph Balsamo";
char res = str._____(_____);
```

14.1.3

To create lists of data of the same type is used the data type **array**.

The access to each element is done using an index where the first value is saved at position 0.

If we want to use the array in the program, we need to declare it:

```
int[] arr1;
```

or

```
int arr2[];
```

Both notations are similar, important is to use **[]**, which defines that it is a list of values defined at the beginning of the notation - in this case, it is a list of integral numbers.

Alternatively, we can define the list of decimal numbers:

```
double arr3[];
```

or strings:

```
String[] strs;
```

14.1.4

Declare a boolean type array:

```
boolean_____ arr;
```

or

```
boolean arr_____;
```

14.1.5

Declaration:

```
int[] arr;
```

defines the reference to the array but we have not reserved any memory for it yet.

The following operation creates the space in the memory for 100 elements:

```
arr = new int[100];
```

The memory is reserved using the command **new**.

The number of elements that we want to reserve the memory for, is defined in square brackets.

The capacity that is reserved corresponds to the number of elements and the data type of the array.

To simplify and increase the code clarity we can create and reserve the space for the array in one row using the following command:

```
int arr[] = new int[50];
```

14.1.6

Fill in the code so that you declare and reserve space for an array of 25 real numbers:

```
double_____ arr_double = _____ double[_____]
```

14.1.7

The array begins from the zero indexes and the last element has the index **numberOfElements - 1**:

index	0	1	2	3	4
int	15	20	16	100	33

The number of elements in the array can not be after the reservation changed anymore.

The information about the count of the elements is get using the command:

```
numberOfElements = myArray.length;
```

where **length** is the characteristics of the array, not a method, that is the reason why we use it in the array without the brackets.

14.1.8

What is the index of the first element of the array?

14.1.9

Fill in the command that returns the count of elements of the array:

```
...
int size = arr._____;
System.out.println("The number of elements of the array is: "
+ size);
```

14.1.10

After the creation is in the **int** type array set all elements to the value 0 (zero).

During the execution of the program can be the value of the element changed following way:

```
myArray[3] = 7;
```

Reading the value from the array is like reading the values of variables, e.g.:

```
sum = sum + myArray[3];
if (myArray[0] == 4)
```

14.1.11

Fill in the following code so that you declare an array of 10 integral numbers and print the difference of the first and last element.

```
_____ arr = new int[_____];
int c = arr[_____] - arr[arr._____ - _____];
System.out.println(c);
```

- length
- int()
- 0
- lenght()
- int[10]
- 1
- 11
- int[]
- int
- 9
- 10

14.1.12

The field values can be filled right at the creation (and this way specify how many elements will the array have):

```
int[] arr = {2, 8, 15, 22, 34};  // array with 5 elements
```

The size of the field is defined by the number of initialized values and this way more information is not needed for memory reservation.

If the initial values are put, then the values of all elements must be mentioned.

To set values of only some elements it is necessary to use the assignment commands in code.

14.1.13

Fill in the code so that in the array **arr** are saved values 2, 4, 6, 8, 10.

```
_____ arr =_____2, 4, 6, 22, 10_____;
arr[_____] = 8;
```

14.2 Reading data into array

14.2.1

Operations that we do over the array usually require processing each element. The transition is done using a loop from the first element to the last contained at the position **arr.length-1**.

The output can be done following:

```
int[] arr = new int[10];
...
for(int i = 0; i < arr.length; i++) {
   System.out.println(arr[i]);
}
```

14.2.2

Fill in the code so that the value of each element is increased by 2:

```
int[] arr = new _____[10];
...
for(int i = 0; i < arr._____; i++) {
   arr[_____] = arr[_____] + 2;
}
```

14.2.3

Fill in the code so that the value of the element is the same as its index in the array:

```
int[] arr = new _____[10];
for(int i = _____; i < arr._____; i++) {
   arr[i] = _____;
}
```

14.2.4

The array elements are usually not available at the program creation but we need to obtain them from the user. In that case, we create the array and input the elements in a loop:

```
Scanner input = new Scanner(System.in);
int[] arr = new int[10];
for(int i = 0; i < 5; i++) {
   arr[i] = input.nextInt();
}
...
```

14.2.5

Fill in the source code that declares a **String** type array of 5 elements that are read from the input.

```
Scanner input = new Scanner(System.{1:SA:=in});
_____[] arr = new _____[5];
for(int i = 0; i < arr._____; i++) {
   arr[i] = input._____();
}
```

- length
- nextLine
- int
- nextInt
- int
- String
- length()
- String

14.2.6

The number of elements we have to work in the program does not have to be always defined in the program.

We can obtain it from the user input, reserve the necessary space and each element read similarly as in the previous case.

```
Scanner input = new Scanner(System.in);
int count = input.nextInt();
int[] arr = new int[count]; // reserve the space
for(int i = 0; i < count; i++) {
   arr[i] = input.nextInt();
}
...
```

14.2.7

Fill in the code that will create an array of a given number of elements and read each element into the array and finds out the sum of the given numbers:

```
Scanner input = new Scanner(System.in);
// find out the number of elements
int n = input._____();
int arr_____ = new int[_____];
// read elements
for(int i = 0; i < n; i++) {
   arr[_____] = input.nextInt();
}
// find out the sum
int sum = 0;
for(int i = 0; i < n; i++) {
   sum = _____ + arr[_____];
}
System.out.println("Sum of the numbers is: " + sum);
```

14.2.8

In the situation where we do not know the number of array elements even after the program execution it is necessary to create an array with a large number of elements and remember how many of them have a value.

E.g.:

Write a program that will read the elements till the input will not be the value 0. After that output the elements in reverse order. Assume that the maximum number of elements on the input is 100.

```
Scanner input = new Scanner(System.in);
int[] arr = new int[100]; // reserve the space for maximum
number of elements
int count = 0; // counter of elements
// in loop we will read the elements till there will not be 0
or we do not reach 100 elements, that is the maximum size of
the array
while (count<100) {
  int num = input.nextInt();
  if (num == 0)  break; // if the input is 0, we terminate the
read
  arr[count] = num;     // assign the given value to the array
  count++;                // and increase the number of
elements in the array
}

// loops ends if the input was 0 or if the count was more than
the maximum elements in the array
for(int i = count - 1; i >= 0; i--)
  System.out.println(arr[i]);
```

14.2.9

Fill in the code that will read the values into the array and add them till on the input is not -1.

```
Scanner input = new Scanner(System.in);
int[] arr = new int[100];
int count = 0, sum = 0;
do {
  int num = _____.nextInt();
  if (num == _____) _____;
  arr[_____] = num;
  count_____;
  sum _____= num;
} while (count<100);
System.out.println(sum);
```

14.3 Constants and random numbers

14.3.1

Value to reserve the size of the array was in previous tasks used always on different places (by array definition, by evaluation if the given count was not exceeded, etc.)

If we want to change this value in the future, we will have to change it in all places what is in the case of long programs complicated.

For this reason, would be great to remember this value in a separate variable and instead of an integral value using a variable. If we would need to change the number of processed elements, we would need to change the value only in one place. Everywhere else would be used already the updated value.

In addition, if we want to prevent accidental rewrite of the variable, we can define it as a **constant** or **final variable**.

This kind of variable can obtain during the program only one value - **constant**, **unchangeable**.

If we once initialize the variable to some value and note it as constant through the keyword **final**, then the value of the variable is unchangeable:

```
final int count = 10;
```

Constant can be at first only declared and then assign it a value - however **only once**.

14.3.2

What keyword is used to define the variable as a constant?

14.3.3

The random number is a useful means to test a program or implementing an element of randomness into programs or games.

To get a random number you can use the method

```
Math.random()
```

that returns real (decimal) value from interval <0,1) – the range contains the value 0.0 but not the value 1.0, in other words:

```
0.0 <= Math.random() < 1.0.
```

If we want to obtain bigger values it is necessary to multiply the obtained value with the maximum value of our request, e.g.

```
Math.random() * 10
```

returns the value from the range

```
0.0 – 9.99999999999
```

14.3.4

Fill in the code so that you generate a value from 0 - 20 (except 20) into the variable **a**.

```
double a = Math._____() * _____;
```

14.3.5

If we do not want to generate decimal but integral numbers, we need to retype.

Notation:

```
int num = (int)(Math.random() * 9);
```

will input into integral variable the value from 0-8

14.3.6

Fill in the code so that you put an integral value from 0 - 15 (included) into a variable.

```
int num = _____(_____.random() * _____);
```

- Math
- (int)
- 16
- 15
- System

- round

14.3.7

Fill in the code so that you put a value from range 50-150 (included) into an integral variable:

```
int num = _____(_____ + _____.random() * _____)));
```

14.3.8

Even in the case, the range is in negative numbers we choose the same approach:

- identify the minimal required value
- we add the random value multiplied by the range interval (eventually we add 1 for integral numbers)

E.g. generate random value from range -20 to 30.

- minimal value is -20 and the interval range is 50

The notation will be following:

```
int c = (int)(-20 + Math.random() * 51);
```

14.3.9

Fill in the code so that you put a value from range -20 to 20 (included) into an integral variable.

```
int num = _____(_____ + _____.random() * _____)));
```

14.3.10

The random number generator is useful by inputting the array of random values.

Make sure that the 10 element array was filled with random integral values from the range of -50 to 50.

We use the knowledge of using constants and define the range of the array using a constant. Then we create the notation for the generation of random numbers

(minimum value is -50, the interval range is 100, i.e., 101) and then we print the array.

```
final int count = 10;
int[] arr = new int[count];
for(int i = 0; i < count; i++)
  arr[i] = (int) (-50 + Math.random() * 101);
for(int i = 0; i < count; i++)
  System.out.println(arr[i]);
```

14.3.11

Fill in the code so that it saves into an array 20 random values from the range -50 to 35 included.

```
_____ int count = 20;                // constant
int[] arr = new int[_____];
for(int i = 0; i < count; i++) _____
  arr[i] = (int) (_____ + _____.random() * _____);
  System.out.println(arr[i]);
_____
```

14.4 Random numbers (programs)

14.4.1 Random number from 0 to 100

Write the code that will generate and print a random integer from the <0,100> interval to the console. For example:

```
Output: 42
```

14.4.2 Random number from -50 to 50

Generate and print a random number from interval <-50,50>.

```
Output (Eg.): -5
```

14.4.3 Random number from given interval

Generate and print a random number for a given interval of two integer numbers. The interval numbers do not have to be given in order smaller, higher.

```
Input : 20 80
Output (Eg.): 61
```

```
Input : 22 -68
Output (Eg.): -3
```

14.5 Simple arrays (programs)

14.5.1 The largest element in the array

Write the code that prints the largest value of the given array. At the input is the first given the number of array elements (space) each array element separated by a space.

```
Input : 5 4 8 12 21 7
Output: 21
```

14.5.2 Smallest array element index

Write the code that prints the first index of the smallest array element of the integer array given on the input. At the input, first is given the number of array elements (space), each array element is separated by a space.

```
Input : 5 4 -8 12 21 7
Output: 1
```

14.5.3 The number of occurrences in the array

Write a program that prints the number of occurrences of a given value in the given integer array. At the input, first is given the number of array elements (space) individual array elements separated by a space, (space) the searched value. Eg.:

```
Input : 6 4 -8 12 21 7 4 4
Output: 2
```

14.5.4 The number of positive and negative values

Write the code that prints the number of positive and negative values in the array for a given integer array specified at the input (zeroes are not counted). At the input, first is given the number of array elements (space) each array element separated by a space. The console will display the number of positive "positive:" and the number of negative "negative:" numbers in separate lines. Eg.

```
Input : 6 4 -8 0 12 -21 7
Output:
positive: 3
negative: 2
```

14.5.5 Divisible numbers

Write the code that prints all the array elements divisible by a given value for the given integer array. At the input, first is given the number of array elements (space), individual array elements separated by a space (space) divisor.

Comma-separated elements are printed to the console, followed by a dot after the last value. If there are no divisible numbers in the array, the console displays the text: "No element divisible by the specified value".

```
Input : 6 24 -8 -12 21 7 4 4
Output: 24,-8,-12,4.
```

```
Input : 6 24 -8 -12 21 7 4 11
Output: No element divisible by the specified value
```

14.5.6 Difference between largest and smallest element

Write the code that prints the difference between the largest and smallest array element for a given integer array given at the input. At the input, first is given the number of array elements (space), each array element separated by a space.

```
Input : 5 4 8 12 21 7
Output: 17
```

14.5.7 The first and second largest number

Write the code that prints the value of the largest and second-largest element for the given integer array from the input. At the input, first is given the number of array elements (space), each array element separated by a space. The console will display the largest and second-largest value of the array, separated by a comma.

```
Input : 5 4 8 12 21 7
Output: 21,12
```

14.5.8 Number of above and below average elements

Write the code that prints the number of above-average and below-average elements in the array for the integer array specified in the input. The average value does not count. At the input, first is given the number of array elements (space), each array element separated by a space. The number of above-average "above:" and below-average "below:" elements, in separate rows, are displayed on the console.

```
Input : 6 4 -8 0 12 -21 7
Output:
above: 4
below: 2
```

14.5.9 Occurrence of divisible numbers

Write the code that finds the number of array elements that are divisible by 8 in a given integer array of 10 elements specified at the input. The array elements are given one at a time, always in a new line.

```
Input :
10
24
21
41
40
31
77
80
4
3
Output: 3
```

```
Input :
1
2
1
1
4
1
1
1
1
6
Output: 0
```

14.6 Arrayless List (programs)

14.6.1 MinMax

Write the code that calculates the minimum and maximum values of a series of integers. Do not use the array of integers. Enter the number of elements to enter, separated by a space. Output the minimum and maximum values.

```
Input : 10 8 4 -5 33 22 56 45 -32 0 23
Output: -32 56
```

```
Input : 5 3 -3 0 -5 -33
Output: -33 3
```

14.6.2 Mean

Write the code that calculates the arithmetic and geometric mean values of a given series of positive integers. Do not use an integer array. At the input are each element separated by a space and the last digit is 999999 that should be omitted. At the output, write the mean values separated by a space and rounded to integers.

```
Input : 1 1 1 1 1 999999
Output: 1 1
```

```
Input : 1 3 5 7 9 11 13 15 999999
Output: 8 6
```

Array Processing

Chapter 15

15.1 Array operations

15.1.1

The simplest operation above the array is its browsing and finding out if it contains some value or how many times it is contained in the array.

Generate random values to a 10 element array from the range -10 to 10 and find out how many times it contains the value 1.

```
final int count = 10;
int[] arr = new int[count];
for(int i = 0; i < count; i++)
  arr[i] = (int) (-10 + Math.random() * 21);
int occurences = 0;
for(int i = 0; i < count; i++)
    if (arr[i] == 1) occurences++;       // if the i-th
element contains 1 increase the occurences
```

The counting can be done also inside the loop that generates the random values.

15.1.2

Find out if the array defined by the user using the element naming is included in a given name.

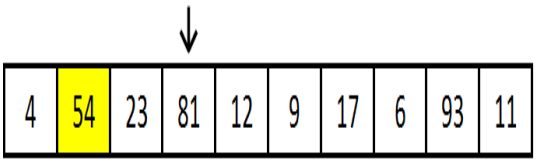
```
Scanner input = new Scanner(System.in);
String str = input.nextLine();
String[] arr = _____"Ewa","Anna","Jan","Eva","Jan", "Jose",
"George"_____;
int i = 0;
boolean contains = _____;
while (i < arr._____) {
   if (arr[i]._____(str)) {
        contains = _____;
        _____;
   }
   i++;
}
if (contains)
  System.out.println(str + "is contained in the list.");
else
  System.out.println(str + "is not contained in the list.");
```

-)
- length()
- (
- false
- length
- }
- true
- compare
- equals
- {
- break
- exit
- continue
- true

📖 15.1.3

Find the maximum in an array of 20 random integral numbers that are generated from range 0 to 100.

Generating the array is for us already routine. Finding the maximum value was solved already in browsing a string. We will solve the task in arrays the same way where we will browse the array of integral numbers from the 0. position till the last and if we find the value that is bigger than the actual maximum, we assign it as a new maximum.




```
// at the beginning can be the first value taken as the
maximum
int max = arr[0];
// we will browse the list from first (the following element)
till the last element
for(int i = 1; i < arr.length; i++) {
  // if the value of the i-th element is bigger than max
  if (arr[i] > max)
    max = arr[i];          // then is arr[i] the new maximum
}
System.out.println(max); // output
```

15.1.4

Fill in the code that reads a given number of elements into the array and finds the maximal value.

```
Scanner input = new Scanner(System.in);
_____ int count = input.nextInt(); // read as a constant
int arr_____ = new _____[_____];
arr[0] = input.nextInt(); // read the first value of the array
int max = _____; // remember the first value of the array
for(int i = _____; i < count; _____) {
  arr[i] = input.nextInt();
  if (max _____ arr[_____])
      max = arr[_____];
}
System.out.println("Maximum is "+ _____);
```

15.1.5

Find out the average of the read integral values in the array ended with 0. Do not count the zero into the average.

The list that is read does not have to be always saved into an array. To process the data can be used one read and we never again need to return to them.

The average is calculated as the sum of all given elements divided by their count. E.g. for 1, 3, 5, 11, it will be

```
(1 + 3 + 5 + 11) / 4 = 20 / 4 = 5
```

In this case, it's enough to read each value once and add it to a common sum and then divide it with the count of elements.

```
Scanner input = new Scanner(System.in);
int sum = 0;
int count = 0;
do {
  int a = input.nextInt();
  if (a == 0) break;  // if the read value is 0, we jump out
of the loop
  sum += a;
  count++;
} while (true);       // because we jump out of the loop using
another way, we can let it run till infinity
double avg = sum / count;
```

```
System.out.println("Average is "+ avg);
```

15.1.6

Find the maximum in a list of integral numbers which count is given as the first value on the input.

```
Scanner input = new Scanner(System.in);
int count = input.nextInt();
_____ max = input.nextInt();
for(int i = _____; i < count; i++) {
  int a = input.nextInt();
  if ( _____ > _____ )
    max = a;
}
System.out.println("Maximum is "+ max);
```

15.1.7

For the given number put as a string find out the number of occurrences of each digit and print it out.

Let's have e.g. number 1419104

We need to obtain the information about the number of the repeat of digits 0, 1, 2 ... 9. The browsing can be done so that we browse the number and find out the count of zero occurrences and print them, then we find out the occurrences of 1, etc.

More effective will be to remember the number of occurrences of each digit and by browsing only increasing the corresponding digit.

This solution takes us to the use of an array where on the 0 positions will be the information about zero's occurrences, on the 1. position about the one's occurrences, etc. We use an integer array with 10 elements (indexes 0-9).

0	1	2	3	4	5	6	7	8	9
1	3	0	0	2	0	0	0	0	1

By stepping over the read number we identify the digit and increate the corresponding position in the array. If we find the value 3, we increase the content of the array **arr[3]** by 1, if we find the value 0, we increase the content of the array **arr[0]** by 1, etc.

```
// we declare the array of 10 elements that have the value set
on 0
int[] arr = new int[10];
// auxiliary variable that is used to read the digit
int digit;
// we read the number we want to examine
String str = input.nextLine();
// we browse its digits
for(int i = 0; i < str.length(); i++) {
  // we get the actual digit and convert it to number...
  digit = Integer.parseInt(str.substring(i, i+1));
  // ...so we can increase the value at the specific index by
1
  arr[digit]++;
}
for(int i = 0; i < 10; i++)  // and at the end we output the
digit and number of occurences
  System.out.println(i + "-"+arr[i]);
```

15.1.8

Fill in the code that finds out how many single-digits, double-digit till 20-digit numbers are in the input.

The reading is ended by the number 0.

Write out only the non-zero values.

```
Scanner input = new Scanner(System.in);
int[] arr = new int[_____];
do {
  String a = input.nextLine();
  int count = a._____();
  arr[count]++;
} while (!a._____("0"));
for(int i = _____; i < 21; i++)
   if (arr[i] != _____)
     System.out.println(i + " - " + arr[i]);
```

- 20
- 0
- compare
- 21
- length
- equals

- 1
- length()
- 0
- 1
- size()

15.1.9

Very often we need to order the saved data during the solving of tasks.

The criteria for ordering can be the following

- numerical (0,1,2,10,11,20...)
- text (0,1,10,11,111,2,20...)

The ordering is mostly named **sorting**. The sorting algorithms do not have to be created as new because there are a lot of proven and functional algorithms that differ in code complexity or requirements on memory or computer performance.

Sorting can be:

- ascending - from the smallest to the largest
- descending - from the largest to the smallest

15.1.10

Which of the following sequences are ordered?

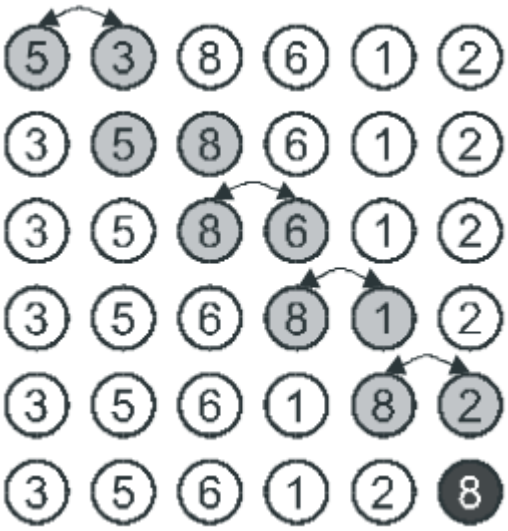
- 1, 11, 110, 112, 2, 21
- abc, bab, bad, element
- 100, 80, 33, 12, 7
- 1, 2, 121, 14, 20, 205, 30
- lur, rul, url, rlu, lru
- list, disp, au, ag, al

15.1.11

The simplest sort is bubble-sort (sorting based on comparison).

The algorithm is based on the comparison of neighbouring elements. By ascending, sort is compared to the neighbouring elements and if the following element is smaller than the previous, then they are exchanged.

By the first iteration through the array will the maximum element get to its (last) position, where others don't. By the second iteration, we don't need to compare all of the pairs, so the last comparison is for the penultimate pair - we save one comparison, etc.



By each iteration through the array is always correctly placed the next element at the end of the array, after the second iteration is the correct one at the penultimate position, etc. Gradually, all the elements "bubble" into the right place.

The number of all iterations through elements will be n-1 because:

- by the first iteration is on its place the 1. element
- by second iteration the 2. element
- etc. till by the n-1 iteration n-1. element and that way the last one

By each iteration through the array will be added at the end one correctly ordered element and by each other iteration, it's enough to go till the already ordered element.

The code is following:

```
for(int i = 0; i < arr.length-1; i++) { // number of
iterations
   for(int j = 0; j < arr.length-i-1; j++) { // moving till
last, penultimate, etc., (n-i) element
      if (arr[j]>arr[j+1]) {  // exchange of elements
            int pom = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = pom;
      }
   }
 }
```

15.1.12

Fill in the code for bubble sort:

```
n = arr.length;
for(int i = 0; i < n - _____; i++) {
   for(int j = 0; j < n-i-1; j++) {
      if (arr[j] > arr[_____]) {
            int pom = arr[_____];
            arr[_____] = arr[_____];
            arr[_____] = pom;
      }
   }
 }
```

15.2 Arrays operations (programs)

15.2.1 Division of the array into even and odd elements

Write the code that divides the integer array from the input into two arrays: the first array will have even and the second array will have odd values. On the output, print "even:" in the new line and "odd:" in the next line. Print the numbers in the same order as they were in the original array. The number of array elements (space) individual array elements separated by spaces is given at the input.

```
Input : 7 4 -8 0 12 -21 7 2
```

```
Output:
even: 4 -8 0 12 2
odd : -21 7
```

15.2.2 An array of even values

Write the code that will create a new array from an integer array with 10 elements given at the input, containing only the even elements from the original. The even elements, separated by a space, are printed to the console.

```
Input :
1
2
1
1
1
1
1
1
1
1
Output: 2
```

```
Input :
1
2
1
1
4
1
1
1
1
6
Output: 2 4 6
```

15.2.3 List positions for the specified value

Write the code that prints the positions of the searched value of the given integer array from the input. The number of array elements (space), each array element separated by spaces, (space), the searched value, are given at the input. At the output print individual positions in separate lines.

```
Input : 5 4 -8 0 4 -21 4
```

```
Output:
0
3
```

15.2.4 Replace an element in an array

Write the code for the specified array of five strings, change that element from the array to the specified string, and prints the modified array.

The string array, the index of the element to change, and the string to replace, are given at the input. The input values are always on a new line. Print the modified array on the console, also cut it off.

```
Input :
shopping
swiming
running
learning
cooking
3
working
Output:
shopping
swiming
working
learning
cooking
```

```
Input :
1
2
3
4
5
3
10
Output:
1
2
10
4
5
```

15.2.5 Remove array element

Write a code that will remove the array element based on the given element index and create a new array without this element.

The input and output values are on separate rows.

```
Input :
1
2
3
4
5
3
Output:
1
2
4
5
```

```
Input :
shopping
swimming
running
learning
cooking
3
Output:
shopping
swimming
learning
cooking
```

15.2.6 Mirror

Write the code that reads 5 given integer values into the array and then mirror them into the second array and prints them. Enter numbers separated by a space at the input. At the output, it prints to the console a mirrored order of numbers, the numbers are separated by space again.

```
Input : 1 2 3 4 5
Output: 5 4 3 2 1
```

```
Input : 9 5 1 4 7
Output: 7 4 1 5 9
```

15.2.7 Sequence

Write the code that calculates the value of all other elements using the first and second elements of the array. Declare the array to work with larger values. The first number is given in the first line and the second number in the second line. The value of the next element is calculated using their sum. The other array values are calculated using the sum of the two previous elements. Print a series of 20 elements separated by a space on the console.

```
Input :
1
2
Output: 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584
4181 6765 10946
```

```
Input :
2
3
Output: 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584
4181 6765 10946 17711
```

15.2.8 List of names

Write the code that will print all the names from a given array that begin with a given letter from the input. At the input, the number of elements of the array is given, the individual elements of the array and the initial letter are all values placed into separate rows in the input. On the output, print out the elements - names that begin with the given letter, separated by a space.

```
Input : 4
john
george
james
charles
j
Output: john james
```

```
Input: 7
charles
gabriel
adam
george
peter
andrew
leonard
```

```
b
Output:
```

15.2.9 The second largest number

Write the code that detects its second-largest element for an integer array (with 10 elements) given at the input. The given values are separated by spaces.

```
Input : 1 2 3 4 5 6 7 8 9 10
Output: 9
```

```
Input : 10 20 30 40 50 60 70 80 90 100
Output: 90
```

15.2.10 How Many Chars

Write the code that counts the occurrences of the character in the given string. At the input, is given a character string consisting of uppercase and lowercase letters and numbers (no spaces). Output each string character and the number of occurrences (separated by a colon) in the order corresponding to the character codes. If the string contains an invalid character, type "Error".

```
Input : Java
Output: J:1 a:2 v:1
```

```
Input : The string for character counting
Output: Error
```

15.3 Arrays under scrutiny (programs)

15.3.1 Frequency of numbers

Write the code that prints the number of occurrences of each unique value from the array for the given integer array. At the input, is given the number of array elements (space), each array element separated by spaces.

Print to the console, in the order in which they follow in the array, each value and the number of occurrences. The values are in separate rows marked with a serial number. For example:

```
Input : 6 4 -8 0 4 0 7
Output:
1. 4: 2
```

```
2. -8: 1
3. 0: 2
4. 7: 1
```

15.3.2 Occurrence of digits

Write the code that detects the number of occurrences of each digit for the number you enter. An integer value is given at the input. Print on the console, each digit and the number of its occurrences in the form of digit - number. Individual values are in separate rows.

```
Input : 1233
Output:
0-0
1-1
2-1
3-2
4-0
5-0
6-0
7-0
8-0
9-0
```

```
Input : 100999233
Output:
0-2
1-1
2-1
3-2
4-0
5-0
6-0
7-0
8-0
9-3
```

15.3.3 Insert control and minimum

Write the code that finds a minimum of 5 integers in the input array into which incorrect values could also be inserted. For correct array values, the minimum value is printed to the console, otherwise, the text "List contains an incorrect value, we do not consider it" and a minimum from the correct values.

```
Input :
1
2
3
4
5
Output: 1
```

```
Input :
2
1
a
0
5
Output: List contains an incorrect value, we do not consider
it
0
```

15.3.4 The occurrences of letters

Write the code that finds the number of occurrences of individual characters (a - z) for the given text and prints them in the form of character - count. Output individual values in alphabetical order and in separate lines. Skip zero occurrences.

```
Input : hello
Output:
e - 1
h - 1
l - 2
o - 1
```

```
Input : agriculture
Output:
a - 1
c - 1
e - 1
g - 1
i - 1
l - 1
r - 2
t - 1
u - 2
```

15.3.5 Average Word Length

Write the code that calculates the average length of words placed in an array, prints the shortest and longest words. At the input, are given the elements of the array - a list of words separated by a space. The shortest word (space), the longest word (space), the average length (rounded to 2 decimal numbers) are printed to the console.

```
Input : Write a program that will compute the average length
of words
Output: a program 4.64
```

```
Input : x x
Output: x x 1.00
```

```
Input : 1 12 123 1234 12345 123456
Output: 1 123456 3.50
```

15.3.6 Remove prime numbers from the array

Write the code that removes prime numbers from the integer array of positive numbers read from the input. At the input, is given the number of array elements (space), each array element separated by spaces. Output prints a new array without prime numbers.

```
Input : 7 4 8 2 12 21 7 47
Output: 4 8 12 21
```

15.3.7 Median

Write a code that will compute the median of integer numbers read from the input to the 1-dimensional array. The numbers should be of different values – if not then print “error”. Input the number of array’s elements and then these elements (integer numbers). Print the median.

```
Input : 6 2 5 33 7 1 -1
Output: 2
```

```
Input : 7 11 66 55 44 33 22 11
Output: Error
```

```
Input : 1 1
Output: 1
```

15.4 Array sort (programs)

15.4.1 Sort the array (numbers)

Write the code that sorts the values from the smallest to the largest in the given integer array. At the input, is given the number of array elements (space), each array element separated by spaces. Sort and print array elements. Display an ordered array on the console, separated by commas, followed by a dot after the last element.

```
Input : 6 -33 63 -29 2 32 6
Output: -33,-29,2,6,32,63.
```

15.4.2 Division and sort

Write the code that divides the given integer array into two separate arrays, one for the positive and the other for the negative numbers. Array with positive elements is sorted in ascending order, the array with negative elements in descending. At the input, is given the number of array elements (space), each array element separated by spaces. On the console, print positive numbers in one line and negative numbers in the other line, separated by spaces.

```
Input : 6 -33 63 -29 2 32 6
Output:
2 6 32 63
-29 -33
```

15.4.3 Sort the array (text)

Write the code that retrieves a list of words from the input and sorts them alphabetically. At the input, is given the number of words in the array, and each word in a separate line. Print the ordered words, again in separate lines.

```
Input : 4
mom
dad
bro
sis
Output:
bro
dad
mom
sis
```

15.4.4 Descending string order

Write the code that sorts the array of strings in descending order (z-a, Z-A). At the input, are given strings separated by a space. Output the ordered strings.

```
Input : Write a program that will sort an array
Output: will that sort program array an a Write
```

```
Input : Bangkok London Paris Dubai Singapore New York Kuala
Lumpur Tokyo
Output: York Tokyo Singapore Paris New Lumpur London Kuala
Dubai Bangkok
```

```
Input : 1 2 3 11 123 1567
Output: 3 2 1567 123 11 1
```

15.4.5 Is the array sorted?

Write the code that prints for the given integer array whether its elements are sorted from smallest to largest. At the input, is given the number of elements (space), each array element separated by spaces. If the array is sorted correctly, "Yes" is displayed on the console, otherwise "No".

```
Input : 6 -33 63 -29 2 32 6
Output: No
```

```
Input : 5 1 2 38 74 115
Output: Yes
```

15.4.6 Sort the array by length and alphabet

Write the code to sort the given array of strings so that the ordered array will contain groups of strings of the same length sorted in ascending order, the strings of each group will be sorted alphabetically A-Z. At the input, given strings are separated by a space. The output should be ordered by array elements.

```
Input : Write a program that will sort an array
Output: a an sort that will Write array program
```

```
Input : Bangkok London Paris Dubai Singapore New York Kuala
Lumpur Tokyo
Output: New York Dubai Kuala Paris Tokyo London Lumpur Bangkok
Singapore
```

```
Input : 1 2 3 11 123 1567
Output: 1 2 3 11 123 1567
```

2D Arrays

Chapter 16

16.1 Matrix

🕮 16.1.1

Write a code that will read the list of student names and their height. The number of students is given on input. Find the highest student and print his/her name and height.

To save the data we will use two arrays where on the same position will be data (name, height) of the same student. So, the student in the second position is called Ivan and his height is in the second array also on position 2 - 133 cm.

0	1	2	3	4	5	6
Jozef	Wolf	Ivan	Sara	Juan	Martin	Lua

0	1	2	3	4	5	6
140	158	133	141	145	138	137

When we browse through the array, we will not save the value of the height but its position (index) in the array. Based on the position we can then find its name. The data reading will be done following:

```
...
Scanner input = new Scanner(System.in);
String data = input.nextLine(); // we read the count of
students
final int count = Integer.parseInt(data); // convert the input
into number constant
// we define arrays with the length of the students count
String names[] = new String[count];
int heights[] = new int[count];
// loop to read the data
for(int i = 0; i < count; i++) {
  names[i] = input.nextLine();
  heights[i] = Integer.parseInt(input.nextLine());   // read
and convert at the same time
}
```

We continue with finding the highest value - we will save the array position:

```
// lets assume that the first one is the biggest
int index = 0;
// browsing loop
for(int i = 1; i < count; i++) {
```

```
  // if the height on the browsed position is higher than the
actual saved, then the index becomes the new maximum index
  if (heights[i] > heights[index]) index = i;
}
// output is trivial
System.out.println("The most height is: " + names[index]+": "
+ heights[index]);
```

16.1.2

Fill in the code that will find the pupils with the given name in the list and will list the age and position in which it is in the array for each of them.

```
String arr[] = {"Michael", "Ivana", "Leo", "Juan", "Anna",
"Quassimodo", "Helena", "Marty"}; // list of pupils
int age[]  = {154, 124, 181, 125, 138, 142, 114, 125}; // list
of ages
String name = input.nextLine();  // loading the search name
for(int i = _____; i < arr._____; i++) {
 if (arr[i]_____(name))
   System.out.println("position: " + _____ + ", age: " +
_____);
}
```

- age[i]
- size
- i
- .equals
- 1
- age(i)
- 0
- length
- .equal
- i+1

16.1.3

Using more arrays to save data about objects is not useful and is hard to code.

In practice is for this using a **matrix** (two-dimensional array) that represents the repository for table data without heading:

name	age	height
Ivan	20	170
Jan	22	184
Juanita	19	162

The matrix represents a data table of the **same type**

- integer table
- string table

We can declare it and reserve the memory space for its elements the same way as for an array:

```
int[][] matrixOfNumbers = new int[10][10];
String[][] matrix = new String[20][30];
```

The first parameter is often taken as several rows and the second as the number of columns (but it's up to the programmer how he/she deals with the values).

The fact that we declare a two-dimensional array is determined by two pairs of brackets.

16.1.4

Complete the matrix declaration for integers with 5 rows and 8 columns

```
int_____ numbers = new int[_____][_____];
```

16.1.5

Matrix has its rows and its columns. Its count is set during the memory reservation:

```
int[][] data = new int[6][6];
```

The intersection of row and column is called **cell** and is the variable equivalent - in the case of matrix declared for integer values, it's an **int** type variable. We access it using the row and column value following way:

```
data[row][column]
```

	0	1	2	3	4	5
0	34	6	3	48	11	35
1	37	32	15	18	9	9
2	11	11	30	24	12	18
3	37	25	1	35	20	4
4	8	48	13	45	16	24
5	7	46	3	15	41	33

E.g.

```
System.out.println(data[1][3]);
```

will print the content of the cell in another row (has index 1 because the numbering starts from 0) and in the fourth column (index 3).

16.1.6

Complete the code to list the contents of the selected cell.

34	6	3	48	11	35
37	32	15	18	9	9
11	11	30	24	12	18
37	25	1	35	20	4
8	48	13	45	16	24
7	46	3	15	41	33

```
System.out.println(data[_____][_____])
```

16.1.7

The change of the cell value will be done using a simple assignment of the value to the cell

```
data[1][2] = 76
```

The check or comparison of the cell value is similar as for other variables, e.g.:

for integer numbers:

```
if (data[2][7] == 9)...
```

for strings:

```
if (data[2][7].equals("John")) . . .
```

The integer matrix has after declaration set all cells to the value 0.

16.1.8

Ensure the content in the tagged cells is set up as shown:

0	6	3	1
37	32	15	18
2	11	30	3

```
data[_____][_____] = 0;
data[_____][_____] = 1;
data[_____][_____] = 2;
data[_____][_____] = 3;
```

16.1.9

The matrix content can be filled already by declaration. The values are listed by rows, where the number of rows nor columns is not declared, and the space is reserved based on the count of put values:

```
int[][] data = {
                  { 0, 6, 3, 1},
                  { 37, 32, 15, 18},
                  { 2, 11, 30, 3}
                };
```

or

```
String[][] data = {
                    {"Ivan", "Jan", "Sara", "Barbora"},
                    {"181", "178", "164", "177"}
                   };
```

In the case of a string matrix, we put the integer values as *String*.

When listing the values, it is possible to have different counts of elements in rows, e.g.:

```
String[][] data2 = {{"John", "Ferdinand", "Michael"},
                    {"Fizgerald", "Habsburg"},
                    {"31", "27", "40", "38", "11", "7"},
                   };
```

With its reading and interpretation is needed to be dealt with in code.

16.1.10

Fill a code that fills an integer matrix with two rows and four columns in the declaration:

```
_____[][] data = _____ _____0, 6, 3, 1_____,
                       _____37, 32, 15, 18_____ _____;
```

16.1.11

If we want to print the content of the matrix, we need to access each cell, i.e. we need to browse all columns in all rows.

The count of **rows** of matrix declared as

```
matrix[m][n]
```

can get using its length:

```
int rows = matrix.length
```

In this case, we use the property **length** without brackets - the same way as for an array.

Because the matrix definition in Java allows using a different count of elements in each row, the information about the count of row elements can get the following way:

```
int columns = matrix[i].length
```

where *i* represents the i-th row of the matrix.

The output of all elements of a matrix can be done following way:

```
for(int i = 0; i < matrix.length; i++) {
  for(int j = 0; j < matrix[i].length; j++) {
     System.out.print(matrix[i][j]+"; ");  // element output
```

```
  }
  System.out.println(); // new line
}
```

16.1.12

Fill in the code to output the content of the matrix:

```
for(int i = 0; i < matrix._____; i++) {
  for(int j = 0; j < matrix[_____]._____; j++) {
     System.out.print(matrix[_____][_____]+"; ");
  }
  System.out.println();
}
```

16.2 Working with matrix

16.2.1

So far, we have worked with the matrix that was entered into the program. If we want to get data from the user, we have to retrieve the values one at a time or read them by line and then divide them into elements.

Read a matrix of *m* rows and *n* columns that contains only the values 0 or 1. Write a code that will evaluate the matrix and print out if it contains more ones or zeros.

We can choose from two approaches:

- find out the number of rows and columns of the matrix and repeat the reading of the value *m x n times*
- find out the number of rows and columns of the matrix, read the matrix by rows and each row divide into columns

The number of rows and columns is given by the user and then we can create the space in memory to save the elements:

```
Scanner input = new Scanner(System.in);
final int m = input.nextInt(); // constants can be used to
ensure that the dimensions of the matrix do not change
final int n = input.nextInt();
 int[][] matrix = new int[m][n];
 for(int i = 0; i < m; i++)
```

```
    for(int j = 0; j < n; j++) {
        matrix[i][j] = input.nextInt();
    }
}
```

In the loop, we read the values from input that are divided by a space, e.g. for the matrix of 3 x 4 it can be the following way

```
1 0 0 1
0 0 1 1
1 1 1 1
```

16.2.2

Fill the code so that constants are used to retain the dimensions of the matrix:

```
Scanner input = new Scanner(System.in);
_____ int m = input.nextInt();
_____ int n = input.nextInt();
int[][] matrix = new int[_____][n];
for(int i = 0; i < m; i++) {
    for(int j = 0; j < _____; j++) {
        matrix[_____][_____] = input._____();
    }
}
```

16.2.3

The second option is to read the whole rows and then dividing them into columns, where:

- the input will be the same way as in the previous example
- we will read the whole row at a time
- we will use the command **split** that can divide the content of the string into an array

```
1 0 0 1
0 0 1 1
1 1 1 1
```

The *split* command is used the following way:

```
String array[] = text.split(" ");
```

Let's have input, e.g.:

```
100 20 50 Anna Casablanca
```

The *split* parameter (in this case space) will serve as a delimiter of the elements. The number of the elements in the array is not known and will be known after the division of the string based on the space occurrence. The number of elements is one greater than the number of separator occurrences in the text. In this case 5 (space is there 4 times)

The result of the division is a string array - we have to assume that the array does not contain only numbers.

```
array = {"100", "20", "50", "Anna", "Casablanca"};
```

Except for space, we can use as a delimiter any character or string. The values are often delimited by following characters: ",", ";", "|" etc.

16.2.4

How many elements will the array get from the following listing?

```
String myText ="Anna;Dana;Lama";
String array[] = myText.split(";");
```

16.2.5

How many elements will the array get from the following listing?

```
String myText ="Anna;Dana;Lama";
String array[] = myText.split(" ");
```

16.2.6

Let's return to our task:

Read the matrix of *m* rows and *n* columns that contains only values 0 or 1.

and let's read each row using the string:

```
 Scanner input = new Scanner(System.in);
 int m = Integer.parseInt(input.nextLine());  // when reading
the rows its appropriate to read all inputs the same way
```

```
 int n = Integer.parseInt(input.nextLine());
 String[][] matrix = new String[m][n];
 String arr[];
 String row;
 for(int i=0; i < m; i++) {
   row = input.nextLine(); // reads the whole row
   arr = row.split(" ");  // divides the row content based on
the space
   matrix[i] = arr; // puts the elements of array into the
reserved space in matrix
}
```

The result will be "array of arrays" in the following way:

	1	2	3	4
matrix[0]				
matrix[1]				
matrix[2]				

During the input can happen that the user will input less or more elements than is reserved for the matrix. In this case, we should inform the user about this.

If we input fewer elements, then the row of the matrix will not contain enough elements. If we input more elements than is possible, then we will input only the maximum allowed.

16.2.7

Fill the code to list whether the matrix contains more 0 or 1 values.

```
Scanner input = new Scanner(System.in);
int m = input.nextInt();
int n = input.nextInt();
count_0 = 0;
count_1 = 0;
int[][] matrix = new int[m][n];
for(int i = 0; i < m; i++) {
   for(int j = 0; j < n; j++) {
        matrix[_____][_____] = input._____();
   }
}
for(int i = 0; i < matrix._____; i++) {
```

```
   for(int j = 0; j < matrix[_____]._____; j++) {
     if (matrix_____ == 0) _____++;
     if (matrix_____ == 1)_____++;
   }
}
if (count_0 _____ count_1) System.out.println("equal count");
if (count_0 _____ count_1) System.out.println("more 0
values");
if (count_0 _____ count_1) System.out.println("more 1
values");
```

16.2.8

Java can print out the list of elements of an array or matrix also using a special loop that does not contain the number of elements. We can just say: go through all elements of the array.

This loop is in some languages called also as *foreach* - for each element.

```
int[] array = new int[20];
// variable value will contain the content of each element of
the array in sequence
for(int value : array)
  System.out.print(value + " ");
```

The loop will do the transition through each element of the array without the control variable.

At each step of the cycle, the array element is inserted into a value variable, which is used to list the contents.

16.2.9

Fill the code so that the sum of all elements in the array is displayed:

```
...
// in array arr are the integer values
int sum = _____;
_____(int x _____ _____)
   sum = sum + _____;
System.out.println(_____);
```

16.2.10

Same as the array elements you can write using this loop also the elements of a matrix:

```
for(String[] row : matrix) { // the element of matrix is the
whole array
   for(String cell : row) {     // in the array we will browse
its elements
       System.out.print(cell + " ");  // the output value
   }
   System.out.println(); // new line
}
```

The first loop goes through the elements of matrix - *matrix* is an array of arrays, so the first element is an array (the variable *row* is an array).

The second loop goes through elements of the row (elements of an array), i.e. one element of the array is *String*.

16.2.11

What type of *row* variable is in the following program?

```
for(int[] row : matrix) {
   for(int cell : row) {
      System.out.print(cell + " ");
   }
   System.out.println();
}
```

- int[]
- array of integers
- array arrays
- matrix of integers
- String
- String[]

16.2.12

Matrix is often taken as a table.

E.g. table:

name	average	age
Jan	1,3	21
Anna	2,8	18
Helen	3,1	16
Francesco	2,5	18

can be saved as a matrix with the following content:

```
String[][] table = {{"Jan", "1,3", "21"},
                    {"Anna", "2,8", "18"},
                    {"Helen", "3,1", "16"},
                    {"Francesco", "2,5", "18"}};
```

We cannot name the columns and cannot use different data types for the columns, but we can work with the saved data.

16.2.13

What does the following program list?

```
String[][] matrix = {{"Jan", "1.3", "21"},
                     {"Anna", "2.8", "18"},
                     {"Helen", "3.1", "16"},
                     {"Francesco", "2,5", "18"}};
System.out.println(matrix[1][2]);
```

16.2.14

Find in the table of names, averages, and ages all 18 years old students. Print out all the information about them.

name	average	age
Jan	1,3	21
Anna	2,8	18
Helen	3,1	16
Francesco	2,5	18

```
String[][] table = {{"Jan", "1.3", "21"},
                    {"Anna", "2.8", "18"},
```

```
                        {"Helen", "3.1", "16"},
                        {"Francesco", "2,5", "18"}};
```

The used matrix will have 3 columns where the age will be in the third one (index 2).

Transition through matrix will be done using a loop. The searched data will be text (table is a string matrix) so we will compare the text ("18") or we will convert it to a number and then compare it with 18.

In the case of similarity, we will print all data from the row.

```
for(int i = 0; i < table.length;i++) { // we read the data by
rows
 if (Integer.parseInt(table[i][2]) == 18)
   System.out.println(    table[i][0] +
                        ", average:" + table[i][1]+
                        ", age: " + table[i][2]+".");
}
```

16.2.15

Fill in the code to find out the number of registered students under 18.

```
String_____ data = {{"Jan", "1.3", "21"},
                     {"Anna", "2.8", "18"},
                     {"Helen", "3.1", "16"},
                     {"Francesco", "2.5", "18"}}
int x = 0;
for(int i = 0; i < _____; i++)
  if (Integer._____(data[i][_____]) < 18) x++;
System.out.println(_____);
```

16.3 Matrix (programs)

16.3.1 Even and odd values

Write the code to find out how many even and odd numbers are included for the given integer matrix (2x4 size). At the input, are given values, each in a separate line. On the output, print the text "even" (space) number, if there are more even numbers in the matrix. Otherwise, print "odd" (space) count on the console. If the number of even and odd numbers is the same, print the text "equal".

```
Input :
1
```

```
2
3
4
5
6
7
8
Output: equal
```

```
Input :
2
2
2
3
3
2
3
2
Output: even 5
```

16.3.2 Reset values below the main diagonal

Write the code that creates a matrix (3x3 size) from the integer values obtained at the input and resets all elements below the main diagonal. The given 9 numbers are separated by a space at the input. Print the modified matrix on the console. Allocate 4 spaces for each value for the matrix.

```
Input : 44 -2 45 -29 35 14 0 50 -34
Output:
  44  -2  45
   0  35  14
   0   0 -34
```

16.3.3 Square

Write the code that prints the numbers *1..n * n* (max. *n* is 10) in a two-dimensional array of *n x n* dimensions. Integer *n* is given at the input. Print the table as seen on the example:

```
Input : 5
Output:
   1   2   3   4   5
   6   7   8   9  10
  11  12  13  14  15
```

```
  16   17   18   19   20
  21   22   23   24   25
```

16.3.4 Sum of diagonals

Write the code that calculates two amounts for a quadratic matrix. At the input, is given the size of the matrix (number of rows/columns) and the individual integer values separated by a space. Output two values: the sum of the elements placed on the diagonal (top left - bottom right) and the sum of the elements on the opposite diagonal (top right - bottom left).

```
Input : 2 1 2 2 1
Output: 2 4
```

```
Input : 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4
Output: 10 10
```

16.3.5 Searching in matrix of integers

Write the code that will check if a given element can be found in the 2D matrix containing integer values. The input contains first the size of the matrix: the number of rows and the number of columns, and then elements counting from left to right and from top to down.

The last input value is the element to search for. Print "found at x y", where x and y are row number and column number respectively of the first occurrence of searched element (by searching the matrix from left to right and from top to down) or "not found".

```
Input : 3 3 1 2 3 4 5 6 7 8 9 5
Output: found at 1 1
```

```
Input : 2 1 1 1 1
Output: found at 0 0
```

```
Input : 3 0 1 0 2 0 2 0 3 0 99
Output: not found
```

16.3.6 Symetric matrix

Write the code that will check if a given square array is symmetric to the matrix diagonal. The input contains the size of an array: the number of rows (and this will be also the number of columns), and then elements, counting from left to right and from top to down. Print "true" if the array is symmetric and "false" otherwise.

```
Input : 3 1 2 3 4 5 6 7 8 9
Output: false
```

```
Input : 4 1 5 6 7 5 1 8 9 6 8 1 10 7 9 10 1
Output: true
```

16.3.7 Mirror matrix

Write the code that prints a mirror image flipped along a vertical axis for a matrix of size *n* x *n* containing 0 and 1. At the input, is given *n*, each array element separated by a space. Print a mirror image of the matrix on the console.

```
Input : 4 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
Output:
1 1 0 0
1 1 0 0
1 1 0 0
1 1 0 0
```

16.3.8 Looking for relations

Write the code that inserts numbers in the two-dimensional array of integers of size *m* x *n* (maximum 11x11) as shown below (identify dependencies). The values *m* and *n* are given at the input. Print the matrix on the console as shown.

```
Input :
8
4
Output:
   0   0   0   0   0
   0   1   2   4   7
   0   2   4   8  14
   0   3   6  12  21
   0   4   8  16  28
   0   5  10  20  35
   0   6  12  24  42
   0   7  14  28  49
```

16.3.9 Sum row vs. elements

Write the code that will compute the number of elements of the 2D matrix (integer values) which values are equal to the sum of the row number and the column number of cells in which this element is placed.

The input contains first the size of the matrix: the number of rows and the number of columns, and then elements counting from left to right and from top to down. Print the number of elements fulfilling the above condition.

```
Input : 3 3 1 2 2 4 5 6 7 8 4
Output: 2
```

```
Input : 2 1 1 0
Output: 0
```

16.3.10 Sorted matrix with integers

Write a program that will check if a given 2D matrix of integer numbers is sorted in ascending order. Check the array by rows (up-to-down) and left-to-right by columns.

The input contains first the size of the matrix: the number of rows and the number of columns, and then elements counting from left to right and from top to down. Print "sorted" or "unsorted".

```
Input : 3 3 1 2 3 4 5 6 7 9 8
Output: unsorted
```

```
Input : 2 3 1 2 3 4 5 6
Output: sorted
```

16.3.11 Sorted matrix with strings

Write the code that will check if a given 2D matrix of strings of characters is sorted in descending order. Check the array by rows (up-to-down) and left-to-right by columns.

The input contains first the size of the matrix: the number of rows and the number of columns, and then elements counting from left to right and from top to down. Print "sorted" or "unsorted".

```
Input : 2 4 Write a program that will search an array
Output: unsorted
```

```
Input : 1 10 j i h g f e d c b a
Output: sorted
```

16.4 Table (programs)

16.4.1 Search in matrix of strings

Write a program that will check if a given element can be found in the 2D matrix containing the strings of characters. The input contains first the size of the matrix: the number of rows and the number of columns, and then elements counting from left to right and from top to down. In the end, input the string to search for. Print "found at x y", where x and y are row number and column number respectively of the first occurrence of searched element (by searching the matrix from left to right and from top to down), or "not found".

```
Input : 2 4 Write a program that will search an array array
Output: found at 1 3
```

```
Input : 1 10 Bangkok London Paris Dubai Singapore New York
Kuala Lumpur Tokyo Warsaw
Output: not found
```

16.4.2 Search in table

Write the code that adds 3 records to the data matrix and find out if the given entry is in the matrix. The input contains the name, surname, and year of birth, each in a separate line. The last input value represents the search string. In the case of a match, all the data belonging to the search string will be printed to the console. If the search string is not in the matrix, it prints "No match".

```
Input :
Adam
Mally
1996
Matthew
Great
1987
Joseph
Carrot
1998
Adam
Output Adam Mally 1996
```

```
Input :
Adam
Mally
1996
Matthew
```

```
Great
1987
Joseph
Carrot
1998
John
Output:
No match
```

16.4.3 Delete a line

Write the code that adds 3 records to the data matrix to see if the given entry is in the matrix. The input contains the name, surname, and year of birth, each in a separate line. The last input value represents the row number (not its index!) that we want to delete. A new matrix is printed to the console without the row name (space) surname (space) year removed.

```
Input :
Adam
Mally
1996
Matthew
Great
1987
Joseph
Carrot
1998
2
Output:
Adam Mally 1996
Joseph Carrot 1998
```

16.4.4 Character search

Write a program that will compute the number of occurrences of the given character within the 2D matrix containing the strings of characters. The input contains first the size of the matrix: the number of rows and the number of columns, and then elements counting from left to right and from top to down. The last input contains the character to count to. Print the number of occurrences of the character. Ignore case differences.

```
Input : 2 5 Write a program that will compute the number of
occurrences R
Output: 6
```

```
Input : 1 10 Bangkok London Paris Dubai Singapore New York
Kuala Lumpur Tokyo x
Output: 0
```

16.4.5 Character sort

Write a program that will compute the number of occurrence characters in a given string. The input contains the string of characters.

Print each character of the string and the number of its occurrences (separated by a space) in the descending order of these numbers of occurrences.

```
Input : Java
Output: a:2 J:1 v:1
```

```
Input : The string for character counting
Output:  :4 r:4 n:3 c:3 t:3 h:2 i:2 a:2 o:2 e:2 g:2 f:1 s:1
T:1 u:1
```

Files

Chapter 17

17.1 Streams

17.1.1

We communicated with the program as a user already from the first lesson of programming:

The *Scanner* made possible the input of the data through channel *System.in*:

```
Scanner input = new Scanner(System.in);
```

the console made it possible to output the text results through the channel *System.out*:

```
System.out.print("Hello World!");
```

To transfer data (reading or writing) is dealt with a communication channel that has to be created or accessed. This channel is called a **stream**.

The channels can be divided into:

- input - input of data into the program
- output - output of data from the program

Because of that is the work with stream always the same no matter what kind of stream it is, we can use the same procedures and commands to access:

- file,
- user input/output,
- memory,
- IP network etc.

17.1.2

What do we refer to as a communication channel for data transmission?

17.1.3

The life cycle of the stream is pretty simple:

creation and opening

- before we start working with a stream, we need to identify it by creating or opening it
- the stream is often open already by creation but if it was not opened it is necessary to do this separately (to reserve the needed system resources)

own work with stream

- you do the needed operations (writing, reading)

closing the stream

- if we finish the work with the stream, we have to close it
- so that the data from cache are written,
- so that the exclusive opened stream is accessible for other objects/processes/users.

17.1.4

Order each step of the life cycle of a stream:

- closing a stream
- own work with a stream
- creation and opening

17.1.5

Working with streams means a potential source of errors. The most common errors can be:

- an attempt to read an empty stream
- an attempt to access a nonexistent stream
- an attempt to write to a closed stream

Because Java is a safe language, and places in the code where errors can be handled, it is difficult to work with files at first glance.

Ultimately, however, it is sufficient to remember that all streaming operations need to be wrapped in a try-catch pair, and in any case ensure that the stream is closed when work is terminated:

```
try {
  // open or create stream
```

```
  // work with stream
} catch (IOException e) {
  System.out.println("Error");
} finally {
 // close stream
}
```

17.1.6

Order each program activity into a logical order:

- } finally {
- } catch (IOException e) {
- // closing the stream
- // working with the stream
- // opening or creating of the stream
- try {
- // treatment / listing the errors
- }

17.1.7

In the program, we need to first determine the type of stream based on what we want to do. There are two groups of streams:

- input - read data from the stream,
- output - write data into the stream.

When using stream, we have to define the source or aim (e.g. keyboard, file, network, etc.).

Based on this information we can choose a suitable stream type (class) that is dedicated for our purposes.

A comprehensive class hierarchy for working with streams is contained in the *java.io* library.

17.1.8

What groups do we divide streams into?

- input

- output
- input-output
- floating
- valid
- fluent

📖 17.1.9

Each type of stream accesses data processing in different ways, and there is always a separate type for reading and writing data:

- binary (work with bytes) - include *FileInputStream*, *FileOutputStream*,
- character (translates bytes into characters: 1 character = 1 or 2 bytes) – *FileReader*, *FileWriter*,
- text (work at the same time with the whole row) – *BufferedReader*, *BufferedWriter*.

In most cases, streams that work with text files and read whole rows are enough.

📝 17.1.10

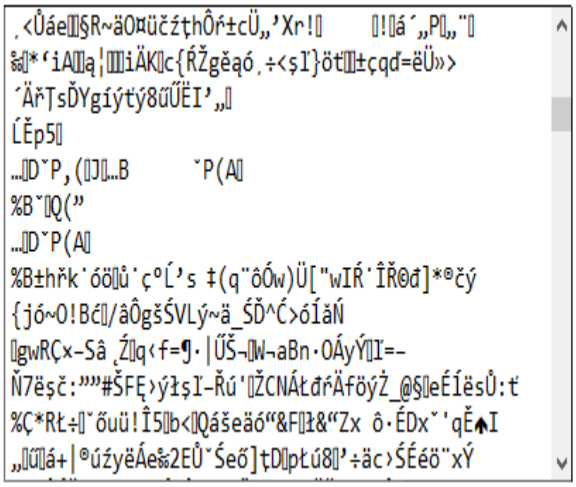
What types of streams are designed to work with rows in a text file?

- BufferedReader
- BufferedWriter
- FileInputStream
- FileOutputStream
- FileReader
- FileWriter

17.2 Text File

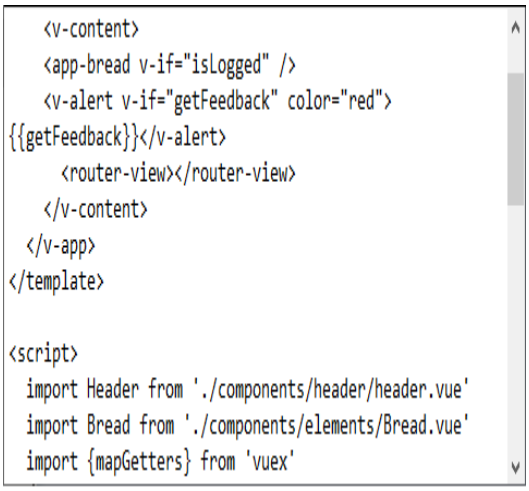
📖 17.2.1

The first form for storing data were binary files in which data was written so that they could be processed as quickly as possible. Whole arrays or even more complex data structures were written. The characters typed were encoded numbers or text as they were stored in the computer's memory.



The disadvantage was that the reading had to always be in the same data structure and if the file was damaged or a character was accidentally overwritten, the file was unusable.

With increasing computing power and the spread of IT to all industries, text files (more precisely text files that are user-friendly) have become the most widespread storage standard.



```
    <v-content>
    <app-bread v-if="isLogged" />
    <v-alert v-if="getFeedback" color="red">
{{getFeedback}}</v-alert>
      <router-view></router-view>
    </v-content>
  </v-app>
</template>

<script>
  import Header from './components/header/header.vue'
  import Bread from './components/elements/Bread.vue'
  import {mapGetters} from 'vuex'
```

Data in text files are stored in a “human-readable” form, often structured by special tags (XML, HTML, etc.). Overwriting or deleting a random character usually has very little impact and the damage can be easily repaired.

Working with text files is usually not programmed by character, but by row.

17.2.2

Which statements are true?

- Data in text files are stored in human readable form.

- Deleting a character in a binary often destroys all of its contents.
- Data in binary files is stored in human readable form.
- Deleting a character in a text file often destroys its entire contents.

17.2.3

Work with files is done using so-called **buffer** classes that can write more characters at the same time or read the whole row. They contain a buffer (cache memory) that can allow you to work with more characters.

Reading/writing is dealt with by *FileReader* /*FileWriter* that allow the data to be saved in a specified place e.g. hard disk and manipulate them as with characters (not bytes).

File creation is done using *FileWriter* that creates and makes accessible a file and offers tools to read characters.

```
FileWriter fW = new FileWriter("file.txt");
```

The following connection is offered to *BufferedWriter* that will extend the existing *FileWriter* to offer writing a whole sequence of characters.

```
BufferedWriter bW = new BufferedWriter(fW);
```

The whole notation can be done also in one step:

```
BufferedWriter bW = new BufferedWriter(new
FileWriter("file.txt"));
```

BufferedWriter is created that will use the just created *FileWriter* with the reference to a file with name *file.txt* as a parameter.

17.2.4

Which statements are true?

- FileWriter ensures character-by-character access to file data.
- FileWriter ensures a link to a file based on the file name.
- BufferedWriter ensures the entire sequence of characters is written in one step.
- BufferedWriter need FileWriter for activities.
- FileWriter ensures that the entire character sequence is written in one step.
- FileWriter ensures file byte access to file data.
- FileWriter ensures access to entire lines in a file at once.

17.2.5

Use the command to write the string to the file

```
String txt = "Any text to write";
bW.write(txt);
```

for row feed

```
bW.newLine();
```

The stream needs to be closed after the operations are completed in order to store cached data and release file access. We use the following command:

```
bW.close();
```

Closing *BufferedReader* also closes its *FileReader*.

17.2.6

Fill in the commands

```
bW._____(data);  // write data
bW._____();    // create a new row
```

- newLine
- println
- write
- addLine
- writeln
- print

17.2.7

Write to a file *name.txt* your name and surname into separate rows.

Working with files is a potential source of errors so we need to catch possible exceptions. The first issue can be wrong given filename, so we start with the *try - catch* block already before file opening:

```
String name = "Jozef";
```

```
String surname = "Bryndza";

try {
   FileWriter fw = new FileWriter("name.txt");   // create
FileWriter to make the file available
   BufferedWriter bW = new BufferedWriter(fw);   // create a
buffered class over it
   bW.write(name);                               // write name
to the file
   bW.newLine();                                 // line feed
   bW.write(surname);                            // write
surname to the file
   bW.close();                                   // close
working with the file
} catch (IOException e) {
   System.out.println(e.getMessage());
}
```

In this case, we did not use the recommended schema *try-catch-finally* because of simplifying the code.

17.2.8

Fill the code that writes to the *user.txt* file the names of the three users stored in variables *u1-u3* into separate rows.

```
String u1 = "One";
String u2 = "Two";
String u3 = "Three";

_____ {
   FileWriter fw = new FileWriter("_____");
   _____ bW = new _____(fw);
   bW._____(u1);
   bW._____();                 // new row
   bW._____(u2);
   bW._____();                 // new row
   bW._____(u3);
   bW._____();                   // close file
} catch (IOException e) {
   System.out.println(e.getMessage());
}
```

- newLine
- close
- try
- user.txt
- write
- BufferedWriter
- BufferedWriter
- finish
- OutputFileWriter
- write
- attempt
- newLine
- write
- open
- writeln
- writeln
- write
- OutputFileWriter

17.2.9

To load data using a class *FileReader* a *BufferedReader*.

In the first step, we need to create access to the file *FileReader*.

```
FileReader fR = new FileReader("file.txt");
```

In the second step, ensure the ability to read data row by row using *BufferedReader*.

```
BufferedReader bR = new BufferedReader(fR);
```

The row reading itself is done by:

```
String s = bR.readLine();
```

17.2.10

Assign to each activity an appropriate command or class:

class to make the file available for reading: _____

class to make the file available for writing: _____

writing a sequence of characters using *bW* in variable *data*: _____

reading the whole row using *bR*: _____

writing a new row using *bW*: _____

closing of the file represented by *bW*: _____

- FileWriter
- bR.readLine()
- bW.close()
- bW.newLine()
- FileReader
- bW.write(data)

📖 17.2.11

Read from file *user.txt* created in the previous task names of three users and print them out.

Using the *try-catch-finally* combination we can secure that the file gets closed even there will be an error during the data reading.

If we want to close the file, we cannot declare the variable inside the *try-catch* but before it. Of course, access to the file should be done inside the block:

```
FileReader fR;                          // declare variables
BufferedReader bR;

try {
   fR = new FileReader("user.txt");     // create FileReader to
access the file
   bR = new BufferedReader (fR);        // create it above the
buffered class
   String u1 = bR.readLine();           // read 1. row
   String u2 = bR.readLine();           // read 2. row
   String u3 = bR.readLine();           // read 3. row
   System.out.println(u1 + ", " + u2 + ", " + u3);
} catch (IOException e) {
   System.out.println(e.getMessage()); // if there is an
error, it will be displayed
} finally {
   bR.close();                          // regardless of
whether an error occurred or not, the file closes
}
```

17.2.12

Sort the correct code commands to retrieve two lines from the *data.txt* file.

- BufferedReader bR;
- try {
- System.out.println(q + ", " + a);
- String q = bR.readLine();
- bR = new BufferedReader (new FileReader("data.txt"));
- } finally {
- System.out.println(e.getMessage());
- String a = bR.readLine();
- bR.close();
- } catch (IOException e) {
- }

17.2.13

If you do not specify a path to identify the file (in its name), it is stored and searched in the application folder.

If you want to specify its absolute location in the filesystem, you must use a double slash when defining the path:

- one occurrence says that it is a special character,
- two that we code „\“.

The path will be defined e.g.:

```
String myFile = "C:\\folder\\data.txt";
```

17.2.14

Which name or file paths are true?

- data
- data.txt
- C:\\folder\\data.txt
- E:\\folder\\data
- C:\folder\data.txt
- D:\folder\\data.txt
- D:\\folder\data.txt

17.3 Working with files

📖 17.3.1

Write a program that generates the specified number from random integers <-500,500> and saves them in a text file.

The task is quite simple - generate a random number and write it to a file in text form:

```
int number, count = 10;
String txt;
BufferedWriter bW;
try {
  bW = new BufferedWriter (new FileWriter("data.txt"));
  for(int i = 0; i < count; i++)   {
    number = (int) (-500 + Math.random()*1001);          //
generate random number
    txt = ""+number;                                     //
convert it to a String
    bW.write(txt);                                       //
write text to the file
  }
} catch (IOException e) { System.out.println(e.getMessage());
} finally { bW.close(); }                                //
close file
```

In this form, we can generate content for the file that does not clearly identify where the number begins and ends.

```
10-200-13548674-4505-5403872114-4540-76544-5421270854
```

Although we could separate the numbers with a comma or semicolon, the standard is to write them in a new row. We add a new line with the command and the result will be a file containing random numbers placed one below the other.

```
int number, count = 10;
String txt;
BufferedWriter bW;
try {
  bW = new BufferedWriter (new FileWriter("data.txt"));
  for(int i = 0; i < count; i++)   {
    number = (int) (-500 + Math.random()*1001);          //
generate random number
```

```
    txt = ""+number;                                          //
convert it to a String
    bW.write(txt);                                            //
write text to the file
    bW.newLine();                                             //
new row
  }
} catch (IOException e) { System.out.println(e.getMessage());
} finally { bW.close(); }                                     //
close file
```

17.3.2

Fill a code that generates 20 random numbers of the range <-20, 50> and writes them into the rows to the *data.txt* file.

```
int number, count = _____;
String txt;
BufferedWriter bW;
try {
  bW = new BufferedWriter(new _____("data.txt"));
  for(int i = 0; i < count; i++)    {
    number = (int) (_____ + Math._____()*_____);
    txt = ""+number;
    bW._____(txt);
    _____._____();
  }
} catch (IOException e) {
   System.out.println(e.getMessage());
} finally {
   bW._____();
}
```

17.3.3

Write a code that can read the given text file by rows and print them out.

Regardless of whether we are reading data from a numbered file or another file, we cannot rely on knowing the number of lines in a file in advance.

It is therefore necessary to check during the reading whether we have reached the end of the file, which is expressed by reading the value *null*.

We will load the data into a string variable row, if it contains text, we will write it out and repeat the reading. If it contains a *null* value, we will not do the listing and stop loading.

Given that we must definitely retrieve data from the file at least once, the optimum structure will finally be a loop with a condition at the end.

```
String row;
BufferedReader bR;
try {
  bR= new BufferedReader (new FileReader("data.txt")); //
prepare access to the file
  do {                                                  //
repeat the following steps
     row = bR.readLine();                               // read
the row
                            // if it was not the end of the
file (null), we write it
     if (row != null ) System.out.println(row);
  } while (row != null);                                //
repeat as long as the content of the row variable is other
than null
} catch (IOException e) {
   System.out.println(e.getMessage());
} finally {
   bR.close();
}
```

17.3.4

Fill in a code to determine the number of rows in the file.

```
String row;
int count = 0;
BufferedReader bR;
try {
  bR = new BufferedReader (new FileReader("data.txt"));
  _____ {
    row = bR._____();
    if (row != _____) count++;
  } while (row != _____);
} catch (_____ e) {
   System.out.println(e.getMessage());
} _____ {
```

```
   bR.close();
}
System.out.println(count);
```

17.3.5

In practice, reading through a loop with a condition at the beginning is more often used, where we read the contents of a row in one step and also verify that it is non-*null*.

```
String row;
BufferedReader bR;
try  {
  bR = new BufferedReader (new FileReader("data.txt"));
                                          // read the line
directly in the condition
                                          // and compare the
read value with null
  while ((row = bR.readLine()) != null )  {
    System.out.println(row);              // if it was not
null, write it
  }
} catch (IOException e) {
   System.out.println(e.getMessage());
} finally {
   bR.close();
}
```

In the loop condition, the contents of the next line in the file are read into the row variable, and if the *null* value is not read, the loop body continues. If the read value is *null*, the cycle ends (or does not run).

17.3.6

Fill in a code to find out how many common characters the file contains.

```
String row;
int count = 0
BufferedReader bR;
try {
  bR = new _____(new _____("data.txt"));
  while ((row _____ bR.readLine()) _____ null )
    count = count + row_____;
  }
```

```
} catch (IOException e) {
    System.out.println(e.getMessage());
} finally {
 _____.close();
}
System.out.println(count);
```

📖 17.3.7

Write a code that will return the count of digits in the file data.txt.

Perhaps the easiest thing to do is to read the row from the file, browse through it by character, and count the numbers.

Loading is provided by a cycle with a condition at the beginning, comparison of characters will be done through for example the method *charAt*.

```
String row;
int count = 0;
BufferedReader bR;
try  {
  bR = new BufferedReader (new FileReader("data.txt"));
  while ((row = bR.readLine()) != null ) { // load until the
end of the file is read
    for(int i = 0; i < row.length(); i++) { // we will browse
through the characters
                                            // of the loaded
row
                                            // if the
character is in the range 0-9
                                                  // it is a
number
      if (row.charAt(i) >= '0' && row.charAt(i) <= '9')
count++
    }
  }
} catch (IOException e) {
   System.out.println(e.getMessage());
} finally {
   bR.close();
}
System.out.println(count);                  // write count of
number
```

17.3.8

Fill a code that finds the number of occurrences a user-entered word is in the *data.txt* file.

```
String row;
int count = 0;
_____ bR;
Scanner input = new Scanner(System.in);
String data = input._____();
 try  {
   bR = new _____(new FileReader("data.txt"));
   while ((row _____ bR._____()) != null )  {
    while (row._____(data) > -1)  {
      count++;
      row = row.substring(row._____(data) + 1);
    }
  }
} catch (_____ e) {
   System.out.println(e.getMessage());
} finally {
   bR.close();
}
System.out.println(count);
```

17.3.9

The file input.txt contains numerical values (each row contains only one number). Create a new file output.txt that will contain absolute values of even numbers from the first file.

E.g. for:

```
-5
4
-8
9
2
-22
```

will be the result:

```
4
8
22
```

In this program we will work with two files - one will read the data, the other will write data.

```
String row;
int number;
BufferedReader bR;
BufferedReader bW;
try  {
  bR = new BufferedReader (new FileReader("input.txt"));  //
prepare for reading
  bW = new BufferedWriter (new FileWriter("output.txt"));  //
prepare for writing
  while ((row = bR.readLine()) != null ) {                //
loading until the end of the file is loaded
    number = Integer.parseInt(row);                       //
convert the contents of the line
                                                          //
to a number
    if (number % 2 == 0) {                                //
see if it's even
       row = "" + Math.abs(number);                       //
we use the row variable to store
                                                          //
the absolute value
       bW.write(row);                                     //
write it
       bW.newLine();                                      //
line feed
    }
  }
} catch (IOException e) {
   System.out.println(e.getMessage());
} finally {
   bR.close();
   bW.close();
}
```

17.3.10

Fill a code that will create a new file *output.txt* that will contain digit sums of odd numbers from the file *input.txt* that contains numerical values (each row contains one number).

```
String row;
int number;
BufferedReader bR;
BufferedWriter bW;
 try  {
   bR = new _____(new FileReader("input.txt"));
   bW = new BufferedWriter(new _____("output.txt"));
   while ((row = bR._____) != _____) {
     number = Integer.parseInt(row);
     if (number _____ 2 == _____) {
       int sum = 0;
       for(int i = 0; i < row.length(); i++) {
         sum = sum + Integer.parseInt(row.substring(_____,
_____));
         bW._____(_____ + sum);
         bW._____();
        }
    }
  }
} catch (IOException e) {
   System.out.println(e.getMessage());
} finally {
   bR.close();
   bW.close();
}
```

17.4 Files processing (programs)

17.4.1 Read from file

Write the code that reads numeric data from the specified text file into a 10-element array. At the input, read the name of the file from which to read. Output the array elements.

```
Input : numbers.txt
Output:
1
2
3
4
5
6
7
8
```

```
9
10
```

Preview of file numbers.txt:

```
1
2
3
4
5
6
7
8
9
10
```

17.4.2 First and last pupil

Write the code that reads the names of the pupils from the specified text file and prints the names of the first and last pupil of the list. Load the text file at the input.

```
Input : list.txt
Output:
Peter R.
Mira M.
```

```
Input : list2.txt
Output:
Miro V.
Brona A.
```

Preview of text file list.txt:

```
Peter R.
Miro V.
George L.
Beata G.
Andrea I.
Tom T.
Alena A.
Brona A.
Mira M.
```

17.4.3 Names on B

Write the code that reads the names of the pupils from the specified text file and prints the names beginning with the letter B. Read the text file at the input.

```
Input : zoznam.txt
Output:
Beata G.
Brona A.
```

```
Input : zoznam2.txt
Output:
Bibiana A.
Bohus A.
```

Preview of text file zoznam.txt:

```
Peter R.
Miro V.
Juro L.
Beata G.
Andrea I.
Tester T.
Alena A.
Brona A.
Mira M.
```

17.4.4 The best students

Write the code that lists students with an average grade of less than 1.5. At the input, read two text files that contain the students' names and their average grades. The average is separated by a semicolon in the file, in some numbers, a dot is used as a decimal separator, in some a comma and some are written as an integer. Print the names of all honoured students on the console (average <= 1.5). Print only names, not averages.

```
Input :
3A.txt
3B.txt
Output:
Peter R.
Miro V.
Andrea I.
Mira M.
Lolo L.
```

```
Miso K.
```

```
Input :
3A2.txt
3B2.txt
Output:
Miro V.
Andrea I.
```

Preview of text file 3A.txt:

```
Peter R.;1.2
Miro V.;1.3
Juro L.;3,3
Andrea I.;1,2
Tester T.;3.0
Alena A.;2.2
Mira M.;1,5
```

Preview text file 3B.txt:

```
Lolo L.;1.2
Miso K.;1,3
Juro J.;3.3
```

17.4.5 Number of characters, lines, sentences and words

Write the code that detects how many characters, rows, sentences, and words are contained in the specified text file. The name of the text file is given at the input. Suppose words do not divide at the end of a line, and no sentence ends with three dots. Print the following information to the console: "characters: 67 rows: 2 sentences: 9 words: 14".

```
Input : book.txt
Output: characters: 70 rows: 3 sentences: 5 words: 16
```

```
Input : book2.txt
Output: characters: 34 rows: 2 sentences: 2 words: 7
```

Preview of text file book.txt:

```
Ahoj ako sa mas? Mam sa dobre. A ty?
Tento test je test.
Testuje sa sam!
```

17.4.6 Maximum absence

Write the code that will find out in the given text file the name of the student with the most absence. At the input, is entered the file name that contains the student's name in each line and a colon-separated number of absence hours. Read the data into an array that has 30 elements in size for up to 30 pupils. Print only the name of the student with the most absence on the console.

```
Input : data1.txt
Output: Anna
```

```
Input : data3.txt
Output: Lavonda
```

Preview of text file data1.txt:

```
Anna:55
Galen:10
Gustavo:20
Bethann:25
Rochel:0
Larhonda:15
```

17.4.7 Calculation of absence

Write the code that finds the average number of absences in the specified text file. At the input, is given the file name that contains the student's name in each row and a colon-separated number of absence hours. Print the number of registered pupils, the total and the average number of absences on the console. Round the number to one decimal place.

```
Input : data1.txt
Output:
10
122
12.2
```

Preview of text file data1.txt:

```
Anna:12
Jano:10
Peter:20
Adam:30
Mato:5
```

```
Jozo:15
Fero:16
Miro:4
Jana:7
Dana:3
```

17.4.8 First and last alphabetically

Write the code that searches in the specified file and prints the names of the first and last pupils in alphabetical order. At the input, is given the name of the file containing the list of students (one name is given in each row). Use an array of max. size 10. Print the name of the first and last pupil in alphabetical order on the console.

```
Input :
list1.txt
Output:
Adam
Zuzana
```

Preview of text file list1.txt:

```
Jano
Peter
Anna
Adam
Mato
Jozo
Zuzana
Miro
Jana
Dana
```

17.4.9 The longest name

Type the code that looks for the longest name in the specified file. At the input, is given the name of the file containing the list of students (one name is given in each row). Use an array of max. size 10. Print the longest name found on the console.

```
Input : list1.txt
Output: Kvetoslava
```

Preview of text file list1.txt:

```
Jano
Peter
Anna
Adam
Mato
Kvetoslava
Zuzana
Miro
Jana
Dana
```

17.4.10 Mirror

Write the code that loads numbers from the given file and mirrors them. At the input, is given the name of the file that contains the data in the form of numbers and saves in a number array of 10 elements. Print numbers from last to first on the console.

```
Input : myData1.txt
Output:
10
9
8
7
6
5
4
3
2
1
```

Preview of text file myData1.txt:

```
1
2
3
4
5
6
7
8
9
10
```

Exercises

Chapter 18

18.1 Advanced exercises (programs)

18.1.1 Complex numbers (bad assignment)

Write a code that will calculate the sum, difference, and product of given complex numbers. Input four numbers of type double (*a, b, c, d*) that form two complex numbers (*a+bi* and *c+di*). Print the sum, difference, and product as 6 numbers (three pairs - first for the sum, second for the difference and the third for the product of given complex numbers).

```
Input : 1 2 3 4
Output: 4.0 6.0 -2.0 -2.0 -5.0 10.0
```

```
Input : 10 0 20 0
Output: 30.0 0.0 -10.0 0.0 200.0 0.0
```

```
Input : 0 10 0 20
Output: 0.0 30.0 0.0 -10.0 -200.0 0.0
```

18.1.2 Ones

Write a code that will compute a sum of numbers encoded in "one-s" (one-digit) system (a system where there is only one digit: 1). The input contains two numbers in "one-s" system. Print the sum of these values both in "one-s" system and decimal system.

LIMITATION: The final sum cannot be greater than 19 digits.If the number is entered differently from the "one-s" system, print -1.

```
Input : 1 11
Output: 111 3
```

```
Input : 12 11
Output: -1
```

```
Input : 11 111
Output: 11111 5
```

```
Input : 1 0
Output: -1
```

18.1.3 Linear Equations

Write a code that will compute solution to a system of linear equations of two variables.The input values are of the double type, (*a1, b1, c1, a2, b2, c2*) are the values of equations - the values for which:

a1x + b1y = c1

a2x + b2y = c2

Print solutions of type double:

- if the system has no solution: 0
- if the system has one solution: three numbers: 1 and values of x and y
- if the system has an infinite number of solutions: Infinity

```
Input : 1 1 1 1 1 2
Output: 0
```

```
Input : 2 3 6 4 9 15
Output: 1 1.5 1.0
```

```
Input : 1 1 1 1 1 1
Output: Infinity
```

18.1.4 RPN2Infix

Write the code converting an arithmetic expression given in the Reversed Polish (postfix) Notation to infix notation. Use only binary operations "+, -, * , / ", 1-digit positive integer values and 1-letter (lower case) variables.

Input a string of characters containing the expression. Print the expression in infix notation or word "error" if the expression is incorrect (to many operators or arguments, wrong operator or wrong order of elements).

```
Input : 12+53-*
Output: ((1+2)*(5-3))
```

```
Input : 1a3+-*
Output: error
```

```
Input : ab/
Output: (a/b)
```

```
Input : ab-cde-*+
Output: ((a-b)+(c*(d-e)))
```

18.1.5 RPN2PN

Write a code converting an arithmetic expression given in the Reversed Polish (postfix) Notation to Polish (prefix) Notation. Use only binary operations "+, -, * , / ", 1-digit positive integer values and 1-letter (lower case) variables. Input a string of characters containing the expression. Print the expression in prefix notation or word "error" if the expression is incorrect (to many operators or arguments, wrong operator, or wrong order of elements).

```
Input : 12+53-*
Output: *+12-53
```

```
Input : 1a3+-*
Output: error
```

```
Input : ab/
Output: /ab
```

```
Input : ab-cde-*+
Output: +-ab*c-de
```

18.1.6 FloatToBits

Write the code that will convert a number of type *float* to its binary representation (sign-exponent-mantissa). Include conversion of: +0.0, -0.0, +Infinity, -Infinity and NaN. Input the *float* number. Print the 32-bit binary string with bits of sign, exponent and mantissa separated by the "-" character.

```
Input : 1.5
Output: 0-01111111-10000000000000000000000
```

```
Input : NaN
Output: 0-11111111-10000000000000000000000
```

```
Input : 0.0
Output: 0-00000000-00000000000000000000000
```

18.1.7 FactorialInf

Write the code that will compute the factorial which is the smaller factorial less then a given number. Input a non-negative integer number (of any size). Print both the -n- and factorial of -n- which meets the requirements.

```
Input : 1000000000000000000000000000000
Output: 29 884176199373970195454361600000 0
```

```
Input : 0
Output: 1 1
```

```
Input : 1
Output: 2 2
```

18.1.8 Fibonacci

Write the code that will compute the *n*-th Fibonacci number according to the iterative algorithm. Input the integer number *n*, greater-or-equal 0 and less-or-equal 90. Print the nth Fibonacci number.

```
Input : 1
Output: 1
```

```
Input : 0
Output: 0
```

```
Input : 10
Output: 55
```

```
Input : 100
Output: 3736710778780434371
```

18.1.9 Cesars cipher

Write the code that will convert a given string of characters according to the Caesar cipher. Input first the key and then the string of characters. Print the converted string. The string should contain letters (in the following order: lowercase and uppercase) and space character, without leading or trailing spaces. The key should be an integer number from the [-100, 100]. If the string contains a character outside acceptable set or the key has an incorrect value then print the word "error".

```
Input : 1 Java
Output: Kbwb
```

```
Input : 200 text to encode
Output: error
```

18.1.10 Coding

Write the code that encodes the text by shifting each letter of the alphabet by 3 positions.

```
Input : Hello
Output: Khoor
```

```
Input : john
Output: mrkq
```

18.1.11 Trimming

Write the code that corrects multiple spaces in a string by replacing them only once.

```
Input : John  has      cold at home.
Output: John has cold at home.
```

18.1.12 Numeric input verification

Write the code to see if the string is a number. An integer or decimal number is given at the input. If it is a number, the text "It is a number" is written to the console otherwise it is "It is not a number".

```
Input : ahoj
Output: It is not a number
```

```
Input : 12.547
Output: It is a number
```

18.1.13 Conversion

Write the code which will convert an integer number from radix 10 to the given radix from range <2, 36>. Input contains the positive number of type *long* and the *radix*. Print the value of given number converted to the given radix.

```
Input : 9223372036854775807 2
Output:
1111111111111111111111111111111111111111111111111111111111111111
1
```

```
Input : 65536 16
```

```
Output: 10000
```

```
Input : 1000 32
Output: V8
```

18.1.14 SumOfPowers

Write the code that will compute the greatest exponent k for the sum *n+n^2+n^3+...+n^k* for given *n*. Power *k* should be computed for all integer types. Input contains integer number from range <2, 127>. Print the values of power *k* for type *byte*, *short*, *integer* and *long* respectively.

```
Input : 2
Output: 6 14 30 62
```

```
Input : 127
Output: 1 2 4 9
```

18.1.15 FibonacciInf

Write the code that will compute the *n*th Fibonacci number (*n* may be of any non-negative value of type *int*) according to the iterative algorithm. Input contains the integer number *n*, greater or equal 0. Print the *n*th Fibonacci number.

```
Input : 200
Output: 280571172992510140037611932413038677189525
```

```
Input : 0
Output: 0
```

```
Input : 1
Output: 1
```

18.1.16 BracketsStr

Write the code that will check if a given string of characters (mimicking the arithmetic expression) contains a correct bracket arrangement: {[(..)]}. The string may contain any kind of characters but only a bracket arrangement should be checked. Input contains a string of characters. Print "correct" if a bracket arrangement is O.K. and "incorrect" otherwise.

```
Input : z[a(bc)d]/e
Output: correct
```

```
Input : [a(*(b+c)-d]/e)
Output: incorrect
```

```
Input : (lkl{jnjn})
Output: incorrect
```

```
Input : {nn[nn(jj)   (mm)mm]mm}
Output: correct
```

18.1.17 RealNumber

Write the code converting string of bits (representing the value of 32-bit float type in its internal form according to the IEEE 754 Standard for Floating-Point). Include +0.0, -0.0, +INFINITY, -INFINITY and NaN values. The input contains the string of 32 bits. Print the converted float value.

```
Input : 01000010110010000000000000000000
Output: 100.0
```

```
Input : 00000000000000000000000000000000
Output: +0.0
```

JavaApp.java

```
public class JavaApp {

    static int bin2int( String str ){
        // Conversion between str and int
    }

    public static void main(String[] args) {
         // write your code here
    }
}
```

18.1.18 Hash

Write the code that will compute a hash for the given string. The hash is computed as the sum of the ASCII code of the following character multiplied by its position number in the string (counting from 0 from right to left). Finally, the computed hash should be brought to range <0, *n*). Input contains the integer number n, greater than 1 and the string of characters. Print the hash of the string.

```
Input : 10 ABC
```

```
Output: 4
```

```
Input : 5 Java
Output: 0
```

18.1.19 Shift

Write a code that will read the *integer* number which bits of given range will set to a given value. Bits are numbered from the left side starting from 0. Input contains the *integer* number, then two values describing the range of bits: from and to, and finally the new value (integer number) of bits to set. Print the result in the binary form of length 32 binary digits.

```
Input : 1 5 7 0
Output: 11111000111111111111111111111111
```

```
Input : 0 1 2 3
Output: 01100000000000000000000000000000
```

18.1.20 PN2Infix

Write the code converting an arithmetic expression given in the Polish (prefix) Notation to infix notation. Use only binary operations "+, -, * , / ", 1-digit positive *integer* values and 1-letter (lower case) variables. Input contains a *string* of characters containing the expression. Print the expression in infix notation or word "error" if the expression is incorrect (to many operators or arguments, wrong operator or wrong order of elements).

```
Input : *+12-53
Output: ((1+2)*(5-3))
```

```
Input : *-+1a3
Output: error
```

```
Input : /ab
Output: (a/b)
```

```
Input : +-ab*c-de
Output: ((a-b)+(c*(d-e)))
```

18.1.21 Prefix notation

Write a code evaluating an arithmetic expression given in the Polish (prefix) Notation. Use only binary operations " +, - , * " and 1-digit positive *integer* values. Input contains a *string* of characters containing the expression. Print the expression-s value or text "error" if the expression is incorrect (to many operators or arguments, wrong operator or wrong order of elements).

```
Input : *+12-53
Output: 6
```

```
Input : *-+123
Output: ERROR
```

```
Input : /ab
Output: ERROR
```

```
Input : +-28*3-25
Output: -15
```

18.1.22 Reverse Polish Notation

Write a code evaluating an arithmetic expression given in the Reversed Polish (postfix) Notation. Use only binary operations " + , - , * " and 1-digit positive *integer* values. Input contains a *string* of characters containing the expression. Print the expression-s value or word "error" if the expression is incorrect (to many operators or arguments, wrong operator or wrong order of elements).

```
Input : 12+53-*
Output: 6
```

```
Input : 123+-*
Output: error
```

```
Input : 12/
Output: error
```

```
Input : +28-3*25-+
Output: -15
```

18.2 List of tasks

18.2.1 Triangle type

Write the code that for three pairs of numbers of type double will check what kind of a triangle they form (isosceles, right-angled). Input three pairs of numbers of type double (x1, y1, x2, y2, x3, y3). Print two boolean values (false or true) that correspond to each kind of triangle. If the points described by the pairs of numbers do not define a triangle, then print "error".

```
Input : 1 10 3 10 2 13
Output: true false
```

```
Input : 10 3 16 3 10 6
Output: false true
```

18.2.2 Occurs at the beginning or end

Write the code to see if the given *string* is in another given *string* at the beginning or end. If it is at the end "Match at end" is displayed on the console, if it is at the beginning, print "Match at the beginning". If the *string* is in the second string but not at the end or at the beginning, it prints "Match is not at the beginning or at the end". If the string is not found at all, it prints "No match".

```
Input :
mama ma maslo
ma
Output: Match at the beginning
```

```
Input :
mama ma maslo
maslo
Output: Match at end
```

```
Input :
mama ma maslo
nema
Output: No match
```

```
Input :
otec kosi travu
kosi
Output: Match is not at the beginning or at the end
```

18.2.3 Palindrome?

Write the code to see if the given string is palindrome. If it is, the console displays "It is palindrome" otherwise it says, "It is not palindrome".

```
Input : kayak
Output: It is palindrome
```

```
Input : ahoj
Output: It is not palindrome
```

18.2.4 Spelling correction

Write the code that change all "i" in the input string for "y".

```
Input : Mi home
Output: My home
```

```
Input : Miro
Output: Myro
```

18.2.5 Swap part of a string

Write the code that finds and replaces one substring with another in the given string. At the input is given the string in which the swap is performed, the original substring to replace is given in the new line and a new substring is given in the last line. The output prints the changed string to the console.

```
Input :
jano
ja
la
Output: lano
```

18.2.6 NonRepDigits

Write the code that will compute several 3-digit numbers with unique (non-repeated) digits encoded in system of given radix. The input contains the radix. Print the number of combinations with unique digits.

```
Input : 10
Output: 648
```

```
Input : 8
Output: 294
```

```
Input : 2
Output: 0
```

```
Input : 16
Output: 3150
```

18.2.7 Months

Write the code that will translate names of the months into their numbers. The method should be case-insensitive. The input contains a name of the month. Print its number and if the input string is not the name of the month, then print 0.

```
Input : January
Output: 1
```

```
Input : january
Output: 1
```

```
Input : JANUARY
Output: 1
```

```
Input : month
Output: 0
```

18.2.8 NoOfDays

Write the code that calculates the number of days in that month for the month and year numbers you enter. At the input, are given 2 *integers*, the first number between 1 - 12 (month number), a space, the second number from 1900-2200 (year). Print the number of days of the month on your console. 31 days for 1st, 3rd, 5th, 7th, 8th, 10th, 12th month; 30 days for 4th, 6th, 9th, 11th month and 28/29 (leap year) days for 2nd month. If the numbers are out of range, print the error "-1".

```
Input : 2 1900
Output: 28
```

```
Input : 1 2000
Output: 31
```

```
Input : 2 2000
Output: 29
```

```
Input : 1 2201
Output: -1
```

18.2.9 Error resistant subtraction

Write the code that subtracts two numbers from each other and is resistant to entering incorrect values. There are given two *integers* at the input. The result of the subtraction is displayed on the console. In case of incorrect input print down if the first number "Error number 1" or the second "Error number 2" is wrong.

```
Input : 1
2
Output: -1
```

```
Input: j
5
Output: Error number 1
```

```
Input : cislo
cislo
Output: Error number 1
```

```
Input : 5
cislo
Output: Error number 2
```

18.2.10 Equation with complex

Write a code for the solution of the quadratic equation of the form: $ax^2 + bx + c = 0$.

Code should:

- read three real numbers
- print complex roots as variable *u, w, v, z* (they represent complex numbers *u+wi* and *v+zi*)
- if a == 0, print -1

```
Input : 1 1 2.5
Output: 2 -0.5 -1.5 -0.5 1.5
```

```
Input : 0 1 2
Output: -1
```

```
Input : 1 -5 6
Output: 2 2.0 3.0
```

```
Input : 0 1 2
Output: -1
```

18.2.11 SortArray

Write the code to sort the array of *integers* in ascending order. At the input, is given the number of array elements (space), each array element separated by a space. Print ordered array elements separated by a space on the console.

```
Input : 5 2 5 33 7 1
Output: 1 2 5 7 33
```

```
Input : 7 77 66 55 44 33 22 11
Output: 11 22 33 44 55 66 77
```

18.2.12 ReversArray

Write the code that prints the *integer* array given at the input in reverse order. At the input, is given the number of array elements (space), each array element separated by a space. Print the array elements in reverse order separated by a space on the console.

```
Input : 5 2 5 33 7 1
Output: 1 7 33 5 2
```

```
Input : 7 77 66 55 44 33 22 11
Output: 11 22 33 44 55 66 77
```

PRISCILLA

priscilla.fitped.eu