software
development
academy

Introduction to Java

Handbook, exercises and assignments

Software Development Academy

April 2019

Contents

Contents

Introduction

Welcome to the fascinating world of Java. This Handbook focuses on several basic and essential aspects so that after having familiarized with it you can successfully write simple programs and express simple logics in the Java code. This does not mean that the material is only addressed superficially. On the contrary, each issue is explained in detail and exemplified. In addition, each chapter is concluded with exercises, and the final part of the Handbook includes some assignments for you to be done. However, due to the volume of the Handbook, it does not exhaust the subject.

I encourage you to read "from beginning to end". While reading, it is advisable to run code snippets on your own. Each chapter is concluded with a *Summary*, that is, a revision and reminder of information in a nutshell as well as additional conclusions. *Exercises* are meant for independent work, experimenting, verification and testing the codes. *Assignments* in many cases involve the knowledge from several chapters and they should be executed at the end, after reading the entire Handbook.

In many places additional markings have been applied.

ℹ	This icon indicates additional information (usually to increase the knowledge) or a reference to a subject beyond the scope of this Handbook.
💡	This icon indicates advice, explanation or a good practice.
❗	This icon is meant to highlight and emphasize very important information that should be memorized.

The individual chapters of the Handbook provide a brief introduction to the most important issues of Java language.

Introduction

In this chapter you will learn a short history of the language as well as its basic design goals and concepts. You will get to know how to set up your work environment and then you will write your first program in Java.

Types of data

This chapter fairly thoroughly discusses the available data types. You will learn about both primitive and object-oriented types and the conversion mechanisms between them.

Operators

After mastering the data types, it is time to perform operations on them. In this chapter you will learn about mathematical and logic operations and comparisons.

Conditional statements

Computer programs are a series of statements, but very often performing a given operation depends on a certain condition.

Loops

Just like in your everyday life, also in programming certain operations must be performed many times.

You will learn how to use various types of loops and how to write an infinite loop.

Arrays

When you have many things of the same or similar kind, you often want to keep them together, close one to another. In programming, you can use arrays for this purpose, which allow you to store multiple data of the same type in one place.

Object-oriented programming

Java is an object-oriented language. This chapter describes classes and objects. You will learn to define classes, create objects of these classes and perform operations on them.

Enjoy your reading!

A brief history of Java

James Gosling

The history of Java begins in 1991, when a group of engineers from the **Sun** company under the leadership of **James Gosling** and **Patrick Naughton** started to work on the project that got a code name "**Green**". The team aimed at developing a possibly "light" programming language that would be independent of the hardware platform. Initially, the language was to be applied in cable TV tuners.

The engineers derived from Unix circles, therefore they based their new language on C++ and Smalltalk. Thanks to this, they created an object-oriented, and not a procedural language.

At the initial stage of the project, the language was called Oak. Reportedly, the name was chosen by James Gosling because of the view of the oak tree outside the window of his office. It turned out that a language with this name already existed and a decision was made to change the name to Java (which is a coffee bush variety), which turned out to be a hit.

The first version of Java was launched in early 1996. In the next version (1.1) many features were added and improved, but the language still had many restrictions.

Then from the version to the version new functionalities were introduced to Java, yet retaining its backward compatibility.

Since 2010, Java has been developed by **Oracle**.

Language assumptions

In May 1996, James Gosling and Henry McGilton published “A White Paper” where they described, among other things, the assumptions and goals of the language. The basic assumptions were as follows:

Simplicity

Unlike other programming languages (e.g. C++), Java has been deprived of many rarely used or difficult and erroneous structures and functions. Also, it is relatively easy to learn this language.

Object-orientation

Java is an object-oriented language where data, or objects, play the most important role. They have *identity*, *status* and *behavior*.

Reliability

A Java compiler can detect many errors that in other languages would not occur until the application was launched.

Networking

Java provides a broad library of functionalities that enable communication through many popular network protocols.

Architecture Neutral and Portable

A compiled Java code can be run on many processors, regardless of their architecture. Thank to (in simple terms) the *Java Virtual Machine,* it is translated into a machine code on the fly. In addition, it is platform-independent, which means you can focus on programming without worrying about the specific mechanisms of a particular operating system.

High Performance

Despite the need to compile and then interpret a byte code, programs written in Java can be very efficient. Compilers feature optimizing mechanisms, e.g. they can optimize the most frequently executed code snippets in terms of speed or eliminate the number of function calls *(inlining*).

Multithreading Capability

Concurrent and parallel programming is relatively difficult. Java provides an accessible abstraction that covers complex constructions, yet allows you to write a code to be run on various processors (cores).

Basic concepts

While learning Java, you will encounter many names and abbreviations. Below you fill find the ones that you should know from the very beginning of your adventure with the language.

JRE

Java **R**untime **E**nvironment allows you to run applications written in Java.

JDK

Java **D**evelopment **K**it thanks to which applications can be developed, debugged and monitored (and also run, because it contains the JRE).

Java SE

Java **S**tandard **E**dition, a platform to be used on standard computers and in simple server applications.

Java EE

Java **E**nterprise **E**dition, a platform designed for complex and expanded server applications.

Java ME

Java **M**icro **E**dition, a platform for applications on mobile phones, mobile devices and small devices.

IDE

Integraed **D**evelopment **E**nvironment, integrated development environment; it is used to develop, test, build and run programs.

JVM

Java **V**irtual **M**achine, executes a Java byte code by interpreting/compiling into machine code.

Source code

A record of a computer program using a specific programming language that describes the operations to be performed by the computer on collected or received data. The source code is the result of a programmer's work and makes it possible to express the structure and operation of a computer program in a human-readable form. In Java, the source code is included in text files with a .java extension.

Compiler

A program that converts a **source code** into a code written in other language. In addition, a compiler is to find lexical and semantic errors and optimize the code. In Java, a compiler processes the code stored in files with a .java extension and places the result of the operation in files with a .class extension.

Bytecode

The outcome of compiling a program written in Java. This code is understandable for the Java runtime environment (JVM).

Working Environment

To execute exercises and assignments, you will need the following:

☑**JDK** (*Java Development Kit*), i.e. tools necessary for programming in Java

☑**IDE** (*Integrated Development Environment*), i.e. the environment in which writing a code in the Java language is possible.

If for any reasons you do not want to or cannot install the above applications or during the installation unexpected problems occurred, you can use one of the tools available online, such as **OnlineGDB**. It is sufficient for the performance of exercises and assignments from this Handbook.

JDK Installation

JDK installation for Windows and Mac OS is very similar. For Linux (Ubuntu) we recommend using the *APT* package management system.

For the purpose of this Handbook, we use Java version 8. All examples, exercises and assignments will work on newer versions as well.

Installation for Windows Environment

- Install the JDK
  - go to https://www.oracle.com/java/
  - click *Download Java for Developers*
  - click *Java SE 8u201* (the version may vary)
  - select *Accept License Agreement* and click *jdk-8u201-windows-x64.exe* (for a 64-bit processor) or *jdk-8u201-windows-i586.exe* (for a 32-bit processor)

    You do not know how to check what version of the Windows OS you have? You can find the answer here.

  - download the file, run it and follow the instructions
- Check the correctness of Java installation
  - right-click the *Start menu* icon and select *Run*
  - write cmd.exe in the open window; the command line window will appear
  - enter the following command

```
java -version
```

  -

```
java version "1.8.0_181"
Java(TM) SE Runtime Environment (build 1.8.0_181-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.181.-b13, mixed mode)
```

- Set the `JAVA_HOME` environment variable to the directory where Java is installed
  - click the magnifying glass button on the taskbar and start entering the following name: *Edit system environment variables*

- right-click the found position
- left-click the *Environment variables* button…
- in the *System variables* section, choose *New…*
- in the *New system variable* window, enter:
  - in *Variable name* field: `JAVA_HOME`
  - in *Variable value* field: `C:\Program Files\Java\jdk1.8.0_181`
- confirm by clicking *OK*
- to verify the correctness, enter the following command in the command line:

```
echo %JAVA_HOME%
```

- the value you have entered above should appear

More information about installation in the Windows environment can be found at JDK Installation for Microsoft Windows

Installation for Mac OS environment

- Install the JDK
  - go to https://www.oracle.com/java/
  - click *Download Java for Developers*
  - click *Java SE 8u201* (the version may vary)
  - select *Accept License Agreement* and click *jdk-8u201-macosx-x64.dmg*
  - download the file, run it and follow the instructions
- Check the correctness of Java installation
  - run the *Terminal* application (Command+Space and enter *Terminal*)
  - enter the following command

```
java -version
```

- the result similar to the one below should appear

```
java version "1.8.0_181"
Java(TM) SE Runtime Environment (build 1.8.0_181-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.181.-b13, mixed mode)
```

- Set the JAVA_HOME environment variable to the directory where Java is installed
  - enter the following command

```
vim ~/.bash_profile
```

- in the open text editor window, add the following line at the end of the file

```
export JAVA_HOME="$(/usr/libexec/java_home -v 1.8)"
```

- save changes to the file and leave the editor
- enter the following command

```
echo $JAVA_HOME
```

- the value you have entered above should appear

More information about installation in the Mac OS environment can be found at JDK Installation for OS X

Installation for Linux (Ubuntu) environment

- Install the JDK
  - run the *Terminal* application (click *Show Applications* button and select *Terminal*)
  - enter the following command

```
sudo add-apt-repository ppa:webupd8team/java
```

  - and then the command

```
sudo apt-get update
```

  - and finally the command to install Java

```
sudo apt-get install oracle-java8-installer oracle-java8-set-default
```

- Check the correctness of Java installation
  - enter the following command

```
java -version
```

  - The result similar to the one below should appear

```
java version "1.8.0_181"
Java(TM) SE Runtime Environment (build 1.8.0_181-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.181.-b13, mixed mode)
```

- Set the JAVA_HOME environment variable to the directory where Java is installed
    - enter the following command

```
sudo gedit /etc/environment
```

    - in the open text editor window, add the following line at the end of the file

```
JAVA_HOME="/usr/lib/jvm/java-8-oracle"
```

    - save changes to the file and leave the editor
    - enter the following command

```
echo $JAVA_HOME
```

    - the value you have entered above should appear

More information about installation in the Linux environment can be found at JDK Installation for Linux Platforms

IDE installation

There are many programming environments designed for Java programmers. The most popular of them are as follows:

- **IntelliJ IDEA**
- Eclipse IDE
- NetBeans

For the purposes of this Handbook, we will install IntelliJ IDEA.

IntelliJ IDEA is available in two versions:

- **Community Edtion**—free version sufficient for our needs
- Ultimate Edition—paid version that offers more options

IntelliJ IDEA installation for Windows and Mac OS is very similar. For Linux (Ubuntu), just download, unpack and run the application.

During installation or start-up, select:

- program's color pattern: for everyday work, we recommend the dark one—**Darcula**
- map of keyboard shortcuts: we recommend the *I used IDEA before* option

Leave the remaining options as default.

Installation for Windows environment

- go to https://www.jetbrains.com/idea/
- click *Download*
- make sure the proper version for your OS (*Windows*) is selected
- click again the *Download* link in the **Community** section
- download the file, run it and follow the instructions
- run the IntelliJ IDEA application by selecting it from the programs menu

Installation for Mac OS environment

- go to https://www.jetbrains.com/idea/
- click *Download*
- make sure the proper version for your OS (*Mac OS*) is selected
- click again the *Download* link in the **Community** section
- download the file, run it and follow the instructions
- run the IntelliJ IDEA application by selecting it from the programs menu

Installation for Linux (Ubuntu) environment

- go to https://www.jetbrains.com/idea/
- click *Download*
- make sure the proper version for your OS (*Linux*) is selected
- click again the *Download* link in the **Community** section
- download the file
- unpack the file to the selected directory using the following command

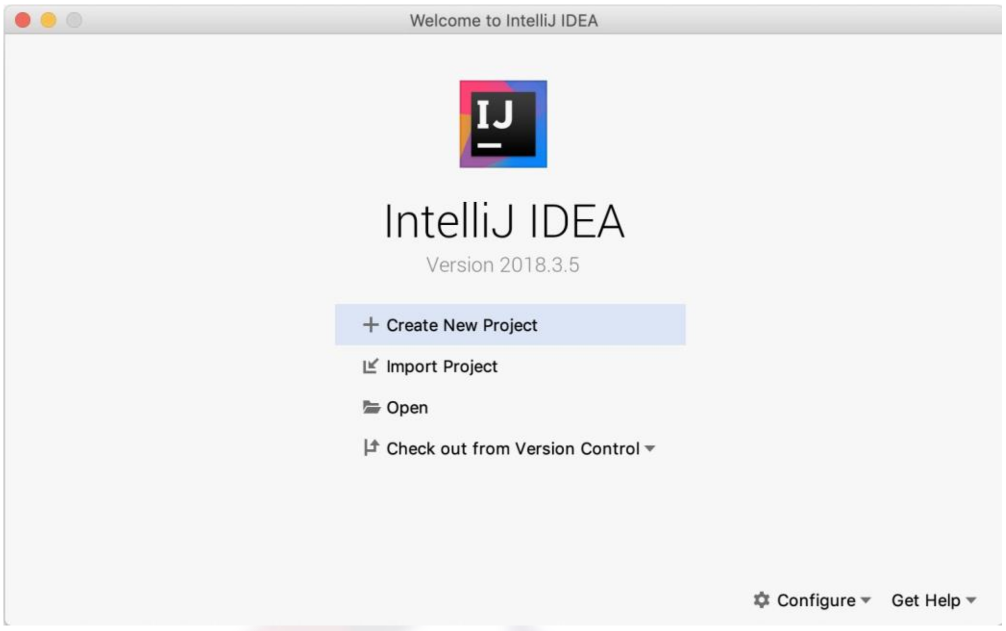
```
tar -xzf idea-2018.3.5.tar.gz
```

- enter the `bin` directory and run the program with command

```
./idea.sh
```

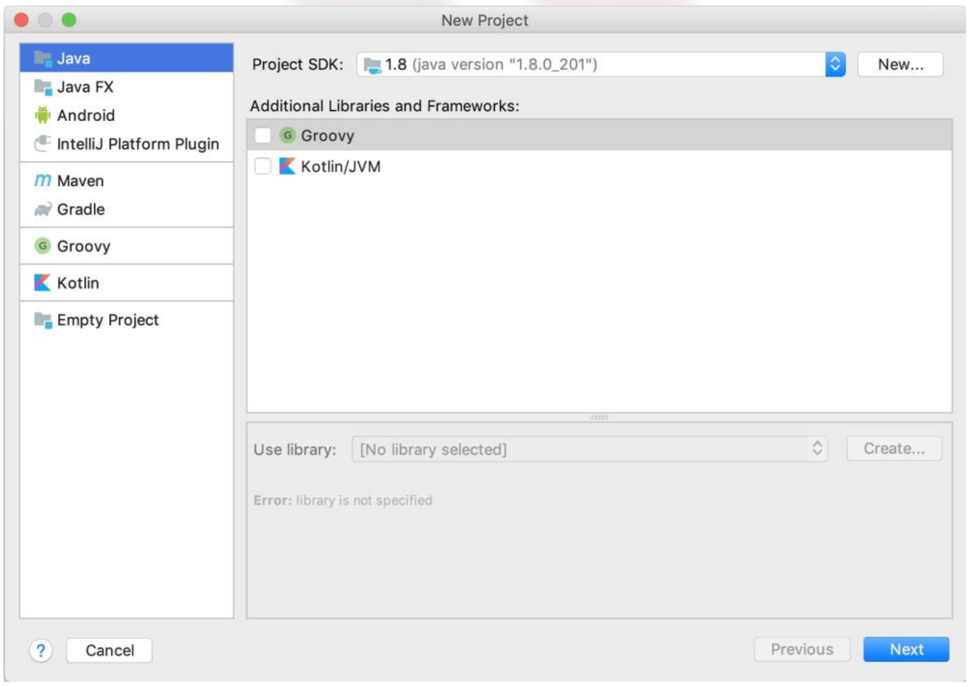
Creating a project

When you have installed the JDK and IntelliJ IDEA, it is time to create a project in which you will locate Java programs. After you run IntelliJ IDEA, the *Welcome to IntelliJ IDEA* window will appear with several options to choose from.



Welcome to IntelliJ IDEA window

- select *Create New Project* (if the window shown above does not appear on your screen, select from the menu **File › New › Project…**



View of the project type choice

- on the left, select *Java, because you* are creating a Java project
- in the middle part, in the *Project SDK* section at the top, you need to indicate the directory where you have installed the JDK. This will be, respectively (the version may vary):

for Windows

C:\Program Files\Java\jdk1.8.0_181

for Mac OS

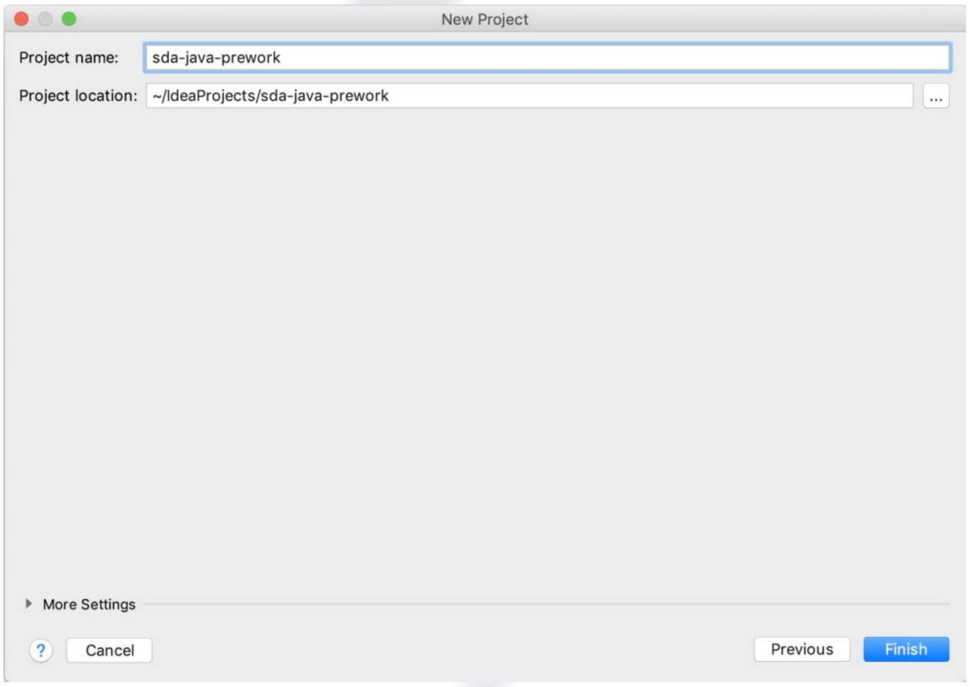
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home

for Linux (Ubuntu)

/usr/lib/jvm/java-8-oracle

The *Project SDK* value that you have entered may be located in the JAVA_HOME environment variable if such has been set. To get the value of this variable in Windows, use command echo %JAVA_HOME% in the terminal (cmd.exe), and in Linux and Mac OS use command echo $JAVA_HOME. The JAVA_HOME variable supports other applications and tools applied in the programming in Java.

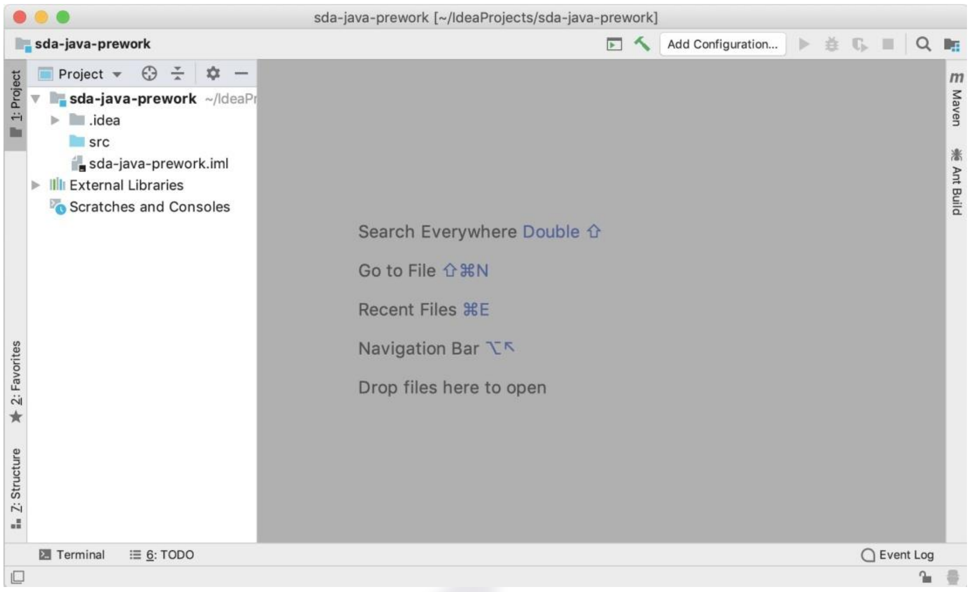
- click *Next* and once again *Next*



Project name: sda-java-prework
Project location: ~/IdeaProjects/sda-java-prework

View of entering the project name

- in the *Project name* field*,* enter sda-java-prework and click *Finish*
- you have just created your first Java project. The result is shown below



View after creating the project

Your first program

Traditionally, the exemplary text "Hello, world!" is presented as the first program. Its task is to display the following text on the screen:

```
Hello, World!
```

Below, the (source) code of this program saved in Java is presented.

MyFirstJavaProgram.java

```
public class MyFirstJavaProgram {          ①

  public static void main(String[] args) {   ②
    System.out.println("Hello, World!");     ③
  }
}
```

Does it look complicated? Below I will discuss step by step what is happening in the individual lines.

① MyFirstJavaProgram is the name of our program. Besides, it is also the name of the file where the program, i.e. MyFirstJavaProgram.java (a file with .java extension), is located. At the moment we will skip the explanation of the words public and class; I will only mention that MyFirstJavaProgram is also the name of a class. Classes are basic entities in Java, and each program consists of one or more classes (later on, you will learn more on what the class is).

② public static void main(String[] args)—this fragment has a special "power". It indicates the declaration of the main method that is responsible for running the entire program; everything starts here. For now, treat this line as a mantra. In our future programs, in this very method we will be placing the Java code that we are going to run.

③ System.out.println() is responsible for writing the text on the screen, which in this case is: **Hello, World!**.

CamelCase notation—if a name consists of several words, we type it without space, and each word begins with a capital letter, for instance MyFirstJavaProgram, AccountService or BankAccount.

It has also been adopted that all elements such as classes, methods or variables are named in English.

Let us go back to our example again.

MyFirstJavaProgram.java

```
public class MyFirstJavaProgram {                ①

  public static void main(String[] args) {       ②
    System.out.println("Hello, World!");         ③
  }                                              ④
}                                                ⑤
```

① We already know that MyFirstJavaProgram is the name of the file and class that we have created. Please notice the opening brace ({) that defines the beginning of the **class body.** The end of the class body is defined by the final brace (}).

② Notice the opening brace at the end of the line—it defines the beginning of the **method body**.

③ System.out.println() is instruction. Is means invoking the println() method on the out object (more information on this topic can be found in further part of this Handbook). The println() method displays the content included between the quotes. Please also note the semicolon at the end of the instruction.

④ The brace ending the main method body

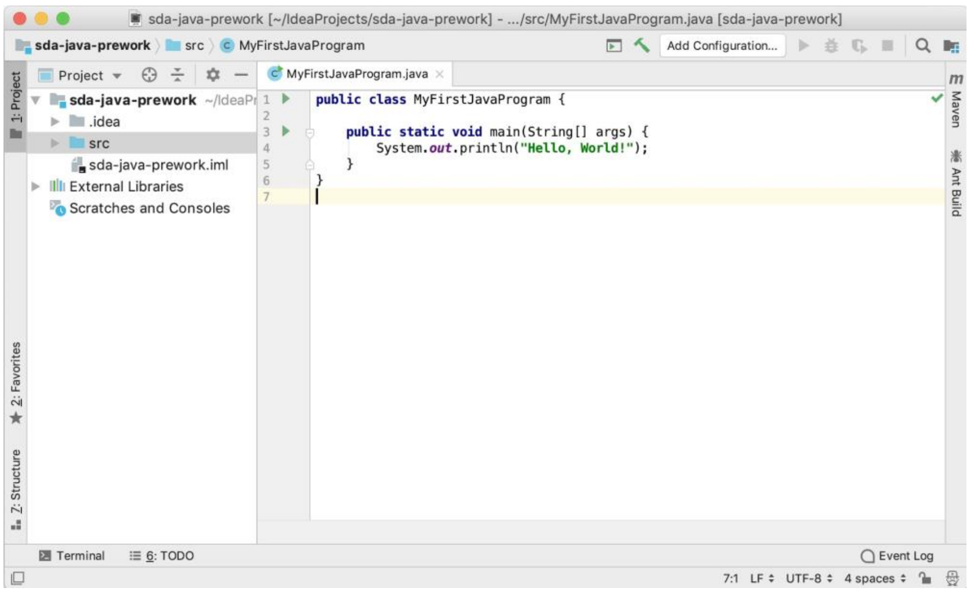
⑤ The brace ending the MyFirstJavaProgram class body

Please notice the way of formatting the code and indentation that are used for fast identification of blocks and analysis of control flow. They have no meaning for the computer, but they make the program more legible for the programmer.

Working with IntelliJ IDEA

Now let us see how to save and run this program in a Java environment. We will use the IntelliJ IDEA program.

- click the src directory and then right-click to select **New › Java Class** from the menu
- in the *Name* field, enter MyFirstJavaProgram (without .java) and click *OK*
- rewrite the above code snippet



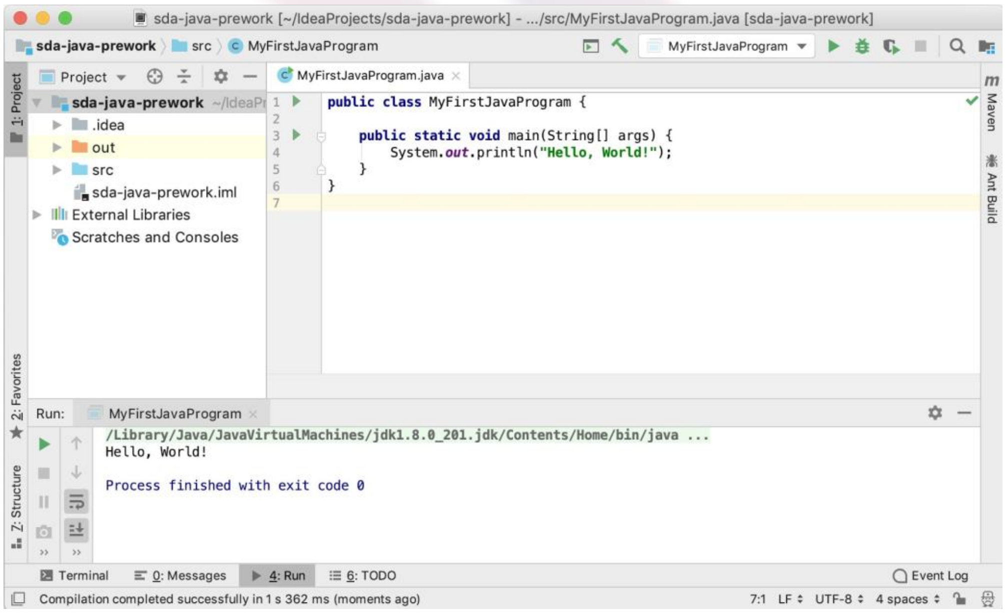
```
public class MyFirstJavaProgram {

    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

The code in IntelliJ IDEA program

Then run the program using one of the following methods:

- Select **Run › Run...** from the top menu and select the MyFirstJavaProgram. You can also use a keyboard shortcut: Alt+Shift+F10
- Click the green triangle (▶) located on the left at the height of public class MyFirstJavaProgram or public static void main(String[] args) and select *Run MyFirstJavaProgram.main()*

In the window below the code, you should see the text Hello, World!



```
public class MyFirstJavaProgram {

    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_201.jdk/Contents/Home/bin/java ...
Hello, World!

Process finished with exit code 0
```

Compilation completed successfully in 1 s 362 ms (moments ago)

The result of running the program in IntelliJ IDEA

In order to improve the efficiency of your work, it is worth mastering at least basic keyboard shortcuts. I encourage you to download the full list of keyboard shortcuts from IntelliJ IDEA site.

You can also use *Key Promoter X* plugin to learn and master keyboard shortcuts. More information can be found here.

Working on the console

This Handbook focuses on writing and running programs in the IDE—IntelliJ IDEA. However, many books, studies and websites provide examples of Java programs operating from the terminal level. Therefore I decided to discuss this way here as well.

All examples presented in this Handbook can successfully run on the console.

If you want to run programs from the console level, you can write them using one of the following editors:

- Sublime Text
- Visual Studio Code
- jEdit

Creating a class

Create a directory on the hard drive to store your Java programs. Using the selected editor create the MyFirstJavaProgram.java file in this directory, with the following content:

```
public class MyFirstJavaProgram {

  public static void main(String[] args) {
    System.out.println("Hello, World!");
  }
}
```

Compilation

Activate the terminal and navigate to the directory where MyFirstJavaProgram.java file is located. Using the javac command, compile the class.

```
javac MyFirstJavaProgram.java
```

As a result of this operation, the MyFirstJavaProgram.class file is created. There should be two files in the directory now:

- MyFirstJavaProgram.java—the source code
- MyFirstJavaProgram.class—the compiled class with the byte code (*bytecode*)

Activation

Now it is time to run the compiled code using the java command.

```
java MyFirstJavaProgram
```

The following text should appear in the terminal window:

```
Hello, World!
```

The javac command is used to **compile**. To do this, enter a filename, e.g. javac App.java.
The java command is used to **run** a program, e.g. java App.

Developing your first program

Now we will try to display something else on the screen so that you can see how Java's instructions are invoked. Let us add text Hello, Java! to the program.

MyFirstJavaProgram.java

```java
public class MyFirstJavaProgram {

  public static void main(String[] args) {
    System.out.println("Hello, World!");
    System.out.println("Hello, Java!");
  }
}
```

The result of program execution should be as below:

```
Hello, World!
Hello, Java!
```

You have learned two things now. The first one is that Java executes instructions from top to bottom: the first message displayed was Hello, World!, followed by Hello, Java!. Besides, now you know more about the println() method operation. This method displays the text passed to it and "goes to a new line", i.e., adds a new line sign to the displayed text. Is there a method that can only display the text without going to a new line? Of course there is. Let us try to remove ln from the print**ln**() name to invoke the print() method.

MyFirstJavaProgram.java

```
public class MyFirstJavaProgram {

  public static void main(String[] args) {
    System.out.print("Hello, World!");
    System.out.print("Hello, Java!");
  }
}
```

When the program is run, the displayed result should be as below:

```
Hello, World!Hello, Java!
```

The println() and print() methods are very useful and offer more possibilities. Later you will learn that more than a text can be passed to them.

And what if you would like to separate these two strings with an empty line? For example, you can invoke the println() method without passing any text to it. Then the method will have no text to display, but it will go to a new line anyway.

MyFirstJavaProgram.java

```
pulic class MyFirstJavaProgram {

  public static void main(String[] args) {
    System.out.println("Hello, World!");
    System.out.println();                ①
  System.out.println("Hello, Java!");
  }
}
```

① invoking the println() method without passing a text to display; the method does not display anything but only goes to a new line.

The result of this program will be as below:

```
Hello, World!

Hello, Java!
```

Summary

What have you learned in this chapter?

What is the history of Java?

The history of Java began in 1991. Engineers from the Unix circle created a “light” programming language independent of the hardware platform. Initially, the language was named Oak, but then it was changed to Java. Since 2010, Java has been developed by Oracle.

What are the basic assumptions of the language?

The basic assumptions are: *simplicity, object-orientation, reliability, networking, independence of architecture, portability, high efficiency and multithreading.*

What is a source code?

A source code is the result of the programmer's work. These are the operations to be executed by the computer. In Java, you place a source code in text files with .java extension.

What is a compiler?

A compiler is a program that converts (and at the same time, optimizes and detects errors) the *source code* into a code written in other language. In Java, a compiler processes files with .java extension and places the result in .class files.

What is a bytecode?

It is the result of compiling a program written in Java.

What do I need to run programs written in Java?

You need **JRE**, that is, **J**ava **R**untime **E**nvironment.

What do I need to write and compile (build) programs in Java?

You need **JDK** (**J**ava **D**evelopment **K**it), that is, an environment with tools for writing and building Java codes.

What is the difference between JRE and JDK and which one should I choose?

If you *only* want to run ready applications written in Java on your computer, all you need is **JRE**. If you want to program in Java, you should install **JDK**. **JRE** serves only to run applications, while **JDK** offers more options, for instance writing and developing applications. **JDK** contains **JRE**.

What is an IDE and which ones are now popular?

IDE means an **I**ntegrated **D**evelopment **E**nvironment. It is used to create, test, build and run programs. It offers many capabilities that can be expanded with a plugin mechanism. Three environments are now popular: ***IntelliJ IDEA***, *Eclipse* and *NetBeans*.

Do I need an IDE to write a program in Java?

No, you can use your favorite text editor. You can build an application using the javac command and run it using java.

How can I check which Java version has been installed on my computer?

You can do this, for example, by using the console and entering the java -version command.

What is the name of the method that runs the program in Java?

This method is public static void main(String[] args).

What is the operation of println() and print() methods?

Both methods display the text passed to them on the screen. However, the println() method goes to a new line after the text is displayed.

How can I display an empty line?

There are several ways to do this. You have learned one of them: you can invoke the println() method without passing a text to it.

What are the basic rules for naming in Java?

It is assumed to use English to name classes, fields and variables in Java. A class name starts with a capital letter, while other elements start with a lowercase letter. If the name consists of several words, we use the so-called *CamelCase* notation: each subsequent word of the name starts with a capital letter.

Exercises

- Write a program (create a new class using the main method) to display Hello, <Your Name>! instead of Hello, World! on the screen.
- Write a program that displays your business card like the one below:

```
##############################
#                            #
#       John Smith           #
#  Sesame Street 345/7b      #
#      New York 10019        #
#                            #
##############################
```

  - use different characters, for instance:
    - - for horizontal lines
    - | for vertical lines
    - / and \ (or +) for corners
- Test the compiling and running the class from the console
  - create a new directory
  - create a PrintHello class (in the file named PrintHello.java) in the new directory:

```
public class PrintHello {

     public static void main(String[] args) {
        System.out.print("Hello, World!");
     }
}
```

  - compile the class using javac command:

```
javac PrintHello.java
```

  - run the PrintHello class by using the java command:

```
java PrintHello
```

  - Check whether the text like below is displayed:

```
Hello, World!
```

- Remove one of the quotes around the Hello, World! text:

```
public class PrintHello {

    public static void main(String[] args) {
        System.out.print("Hello, World!);
    }
}
```

- try to compile the program…
- check whether the following error appeared:

```
PrintHello.java:4: error: unclosed string literal
        System.out.println("Hello, World);
                           ^
1 error
```

- correct the quote and try to “damage” the program in some other way (each time try to compile and pay attention to the errors returned):
  - remove or add brace { or }
  - remove ;
  - remove or add a letter to one of the methods
  - change the class name (without changing the file name)

Types of data

In Java, as in other languages, we distinguish many (built-in) data types that can store numbers, characters, strings (texts), logical type, and many sophisticated, complex types that can perform operations in addition to storing values.

Data types can be divided into two types:

- simple (primitive)
- complex (object-oriented/reference)

The names of primitive types are written in lowercase, while object-oriented types are capitalized.

Variables

Variables can be compared to “containers” or “boxes” for data. The data that can be stored in a specific variable are defined by their type. Each variable has:

- a type
- a name
- a value

Pattern of a variable declaration

```
type variable-name;
```

While declaring the variable, you **must** specify its name. The value can be assigned later.
Pattern of declaration and initialization of a variable

```
type variable-name = value;
```

Before a variable can be used, a value must be assigned to it. The code below will result in an error!

Example of using a variable without initialization

```
int size;
System.out.println("Size = " + size);
```

The entry "Size = " + size means *concatenation* of two values. Concatenation is performed by the + operator which in this case combines the Size = string with the value of the size variable. More information can be found in chapter Arithmetic and concatenation operators.

Names of variables should be descriptive and self-documenting.

The division of data types determines the types of variables:

variables of simple types
they store values only

variables of complex types
they store only references (an address of a memory cell) to the data that they “point to”

To better understand the difference between the variables of simple and complex (object-oriented) types, I will use some examples.

Coins and banknotes can be compared to simple type variables, because the value indicated by the denomination is stored in them. However, a bank card or a credit card is in this case responsible for a variable of the complex type (references); the card only *indicates* the account where the money (value) is physically located.

A remote control that remotely executes operations on a receiver, e.g. a TV, an amplifier, etc., is another example of analogy to the reference (object-oriented) types. Unlike variables of simple types (that store the value), variables of complex types only store the reference to the object, therefore it is possible to have more than one variable that has the same reference to the object; it is like having more than one remote control for a TV or a stereo set.

Java features **strict type control**. Each variable must be of a specific type that has been once declared—it cannot be changed anymore!

Primitive types

There are eight basic types in Java. These are types that have a certain size in memory; they can store numeric values (integers and floating point values), characters and logical values.
All numerical types are **signed**, i.e., they have a sign.

Variables (values) of basic types are compared using sign ==.

In a simple type variable, only the value is stored!

Integer types

Integer types are numbers with no fractional part. They can represent both positive and negative values.

- byte: from -128 to 127 (2^8 numbers)
- short: from -32,768 to 32,767 (2^{16} numbers)
- int: from -2,147,483,648 to 2,147,483,647 (2^{32} numbers)

- long: from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (2^{64} numbers); suffix l or L is applied (L is recommended)

The most frequently used types are byte, int and long.

byte

In programming, we often use *bits,* and *1 byte* is *8 bits*; byte is often used when working with files, networks and large arrays

int

is the **default integer data type** in Java. When you enter a value, for instance 5, Java treats this value as int by default. Operations on numbers (integers) are also performed on int types by default. This range is sufficient for most operations

long

sometimes during operations on the int type we can exceed the range for this type and we need a larger one, e.g. when adding very large numbers of the int type we get a number out of range—then it is worth switching to the long type. The long values are marked with letter l or L, e.g. 5L, 120000000L, -556677889900L

Examples of variables of integer types

```
byte  b  =  (byte)  56;  ①
short  s  =  (short)  89;  ②
int i = 256;
long l = 5500L;
```

① (byte) indicates *casting* on the byte type

② (short) indicates *casting* on the short type

In this case, casting is not necessary (Java can perform it automatically), but this mechanism is worth knowing (more information about it can be found in chapter CONVERSION OF TYPES)

Comment on the int type

By default, integer operations in Java are performed using the int type. Please look at the code below:

```
long a = 24 * 60 * 60 * 1000 * 1000;
long b = 24 * 60 * 60 * 1000;
System.out.println("a / b = " + a / b);
```

According to what we remember from our maths lessons, we can reduce the a / b quotient by pulling value 24 * 60 * 60 * 1000 in front of the bracket and expect a result equal to **1000**. Unfortunately, in the current form, the value displayed on the screen will be as below:

```
5
```

What is more, if both the a and b values are multiplied by 1000 first, the result will be even more surprising:

```
long a = 24 * 60 * 60 * 1000 * 1000 * 1000;
long b = 24 * 60 * 60 * 1000 * 1000;
System.out.println("a / b = " + a / b);
```

Result:

```
-3
```

Why is this happening? Why when we multiply and divide positive numbers, we get a negative number as a result?

As I have mentioned earlier, Java performs calculations on the int type by default. When multiplying int type values, the range is exceeded and the result becomes incorrect (it is still the int type). Then it is assigned to the long type.

To avoid such a situation, it is enough to add letter L to one of the values, which will allow Java to perform the operation on the long type.

```
long a = 24 * 60 * 60L * 1000 * 1000;
long b = 24 * 60 * 60 * 1000;
System.out.println("a / b = " + a / b);
```

Floating point types

Floating point types are types with a fractional part, which can be both positive and negative values.

float
: a single precision format on 32 bits; suffix f or F is applied (F is recommended)

double
: a double precision format on 64 bits; **the default type for floating point values**; to increase readability, you can use suffix d or D (D is recommended)

Examples of variable of floating point types

```
float f = 3.141592F;
double d = 2.718281828D;
```

Pay attention to float and double types

Types float and double should not be used to store financial values, because in some cases the string of these numbers leads to loss of information (precision). This is due to the representation of floating point numbers in a binary system in which there is no accurate representation of fraction 1/10, just like in the decimal system there is no exact representation of fraction 1/3.

Example of a problem with precision

```
System.out.println(2.00 - 1.10);
System.out.println(2.00 - 1.50);
System.out.println(2.00 - 1.80);
```

The result of the operation will be as below:

```
0.8999999999999999
0.5
0.19999999999999996
```

For storing financial values, use the BigDecimal class.

Character types

char
: is used to store one character. These characters are entered between apostrophes, e.g. 'a', 'T', '+'. They are character codes (non-negative integers from 0 to 65,556). Each code indicates a character in *Unicode* standard. We can express them in hexadecimal notation (in the position system based on number 16). Their values range from '\u0000' to '\uFFFF'. \u is a substitute symbol for a character in the *Unicode*. char is the type **without a sign** (*unsigned*)—it stores non-negative values.

Examples of character type variables

```
char a = 'a'; char
bigA = 'A';
char newLine = '\n'; char
plus = '+';
```

Sample *Unicode* character codes

- 'A' is '\u0041'
- 'B' is '\u0042'
- 'C' is '\u0043'
- 'a' is '\u0061'
- 'b' is '\u0062'
- 'c' is '\u0063'
- '0' is '\u0030'
- '1' is '\u0031'
- '9' is '\u0039'

In addition to the \u symbol, there are several other important and useful substitute symbols.

Substitute symbols for special characters

substitute symbol	name
\b	Backspace
\t	Horizontal tab
\n	New line
\r	Carriage return
\"	Double quote
\'	Single quote
\\	Backslash

They can be used, for instance, with strings of characters.

Example of substitution symbols applied

```
System.out.print("Hello, World!\n");
System.out.print("\tHello, World!\n");
System.out.print("\t\t\"Hello, World!\"\n");
```

The result will be as below:

```
Hello, World!
        Hello, World!
                "Hello, World!"
```

Logical type

boolean
: a logical type that can store one of two values: false or true. It is used to verify logic conditions

Examples of variables of a logical type

```
boolean isAdult = true;
boolean active = false;
```

Primitive types—summary

type	size	description	default value
byte	1 byte	-128 … 127	(byte) 0
short	2 bytes	-32,768 … 32,767	(short) 0
int	4 bytes	-2,147,483,648 … 2,147,483,647	0
long	8 bytes	-9,223,372,036,854,775,808 … 9,223,372,036,854,775,807	0L
float	4 bytes	~ -3,40282347E+38 … ~ 3,40282347E+38	0.0F
double	8 bytes	~ -1,79769313486231570E+308 … ~ 1,79769313486231570E+308	0.0
char	2 bytes	0 … 65 556 ('\u0000' … '\uFFFF')	'\u0000'
boolean	1 bit*	false, true	false

*) boolean type stores *1 bit* of information, but according to the documentation its size is not precisely specified

Object-oriented types

In addition to the above-mentioned simple types (which only store the value), Java provides thousands of object-oriented types that can consist of both primitive and object-oriented types. In addition to storing values, they also provide operations (methods).

In addition to the types found in the Java distribution, there are hundreds of thousands types created and shared (in the form of libraries) by Java programmers around the world.

What is more, each of us—by programming and defining new classes—creates new types of data every day.

Variables of object-oriented types are compared using the equals() method.

Instances of classes (objects) of object-oriented types are created using a key word: new.

The default value for object-oriented variables is null.

String

Perhaps the most popular object-oriented type in Java is String—we were using it in the program displaying text Hello, World!. A String is called a *chain type* since it consists of a chain (string) of characters, that is, individual characters (chars) surrounded by quotes " ". For instance, “Alice has a cat.", "Hello, World!", "New York 10019" or "" (indicating an empty string of characters). Character strings are often called *inscriptions* or *literals*.

Examples of variables of the String *type*

```
String helloWorld = "Hello, World!";
String title = "Thinking in Java";
String text;                        ①
```

① the text variable has a null value; performing any operation on this variable will result in a runtime error!

String is a unique class

The String class is unique in many respects.

Primitive types have only values that we assign to variables using the = sign.

To create an object-oriented type (create an object of a given class; more information on this topic can be found in chapter "Object-oriented programming"), you should use the keyword new before invoking the constructor.

Examples of creating String type objects

```
String helloWorld = new String("Hello, World!");          ①
String title = new String("Thinking in Java");
```

① entry new String("text") indicates invoking the constructor of the String class and transferring the "text" argument to it; it creates a new object of the String class.

The String type is so popular that we do not need to use the keyword new and the constructor in order to create a new instance; you simply need to assign a string of characters to the variable.

Examples of creating objects of the String type—short version

```
String helloWorld = "Hello, World!";
String title = "Thinking in Java";
```

❗ There is a subtle difference between these two entries!

In addition to storing entries, you can perform operations on the String type, that is, you can invoke methods.

An example of invoking the length() method that returns the length of a string.

```
System.out.println("Alice has a cat, and a cat has Alice!".length());          ①
```

① to invoke the method, we use a dot character (.) followed by the method name, in this case length(); in round brackets we pass arguments to the method—in this case we invoked the argumentless (non-parametric) method. The following text will be displayed on the screen:

```
37
```

Example of invoking substring() method that returns the string length

```
String text = "Alice has a cat, and a cat has Alice!";
System.out.println(text.substring(13, 25));          ①
```

① the substring(int beginIndex, int endIndex) method returns the substring (fragment of the text) according to the passed argument values. beginIndex indicates the beginning of the fragment (including this character), while endIndex indicates the end of the fragment (without this character); the values are provided starting **from zero**

The screen will display:

```
Alice has a cat, and a cat has Alice!
01234567891111111111222222
          0123456789012345
and a cat has Alice
              ^           ^

              |           |
         (inclusive) (exclusive)
```

Some useful methods of the String class

Pay attention to null

If we try to invoke an operation, i.e., refer after the dot (.) on an uninitialized variable, a runtime error will occur (an exception NullPointerException will be returned).

name of the method	operation
charAt(int index)	returns the character located at the index position
endsWith(String suffix)	checks whether the text ends with a suffix string
equalsIgnoreCase(String otherString)	compares the text and ignores the case of the letters
indexOf(String str)	returns the position of the beginning of the str string in the text
isEmpty()	checks whether the text is empty ("")
lastIndexOf(String str)	returns the last position of the beginning of the str string in the text
replace(char oldChar, char newChar)	returns the text, replacing oldChar with newChar
startsWith(String prefix)	checks whether the text begins with the prefix string
toLowerCase()	returns the text, changing uppercase letters to lowercase ones
toUpperCase()	returns the text, replacing lowercase letters with uppercase ones
trim()	returns text, removing white characters from the beginning and end

String is *immutable*

The String class is *immutable* (unmodifiable)! Please note that the methods of this class do not modify the original value of the string and only return the modified copy.

```
String hello = "Hello, World!"
hello.toUpperCase();          ①
System.out.println(hello);
```

① The toUpperCase()was invoked, which does not modify the original text but only returns the modified string; in this case, we did not assign the result to the new variable, so we have lost the effect of its operation.

```
Hello, World!
```

Operations on classes and objects

I would like to point to the difference in the invoking of the methods that we have used so far. Please look at the code below:

```
String.valueOf(23);
String text = new String("Alice has a cat");   ①
text.endsWith("cat");                          ②
```

① is the invoking of the valueOf() method on the **String** class. We did not invoke this method on an (object-oriented) variable, i.e. we did not invoke this method on a particular value of this variable, but only (*in a context-free manner*) on the String class. The valueOf() method is a *static method,* that is, it can be run without having (creating) a variable of the String type.

Classes, methods, blocks and static variables are not within the scope of this Handbook.

② this is invoking the endsWith() method on a **specific** variable. Depending on the value stored in the text variable, the action (result of the invocation) of this method will vary. The endsWith() method is an *instance method*, that is, we invoke it on a String *class object*. You will learn more about it in chapter Object-oriented programming.

Classes responsible for date and time

Now I will show you several classes that store the date and time and offer the opportunity to work on this data.

- LocalDate
- LocalTime

- LocalDateTime

Example of receiving current date and time

```
System.out.println(LocalDate.now());
System.out.println(LocalTime.now());
System.out.println(LocalDateTime.now());
```

To use these classes, we need to import them (indicate to Java in which *package* (directory) they are located. At the beginning of the file, before the declaration of the class, add the following text:

```
import java.time.*;
```

The result will vary depending on the start time, but it will have the format as below.

```
2019-04-08
20:21:16.726
2019-04-08T20:21:16.726
```

Based on the output, we see that:

- LocalDate represents date
- LocalTime represents time
- LocalDateTime represents date and time

You will often encounter also Date and Calendar classes where date and/or time are processed.

As an exercise, test the methods of these classes.

Class for simple mathematical operations

For simple mathematical operations, we can use the Math class.

Example of some basic mathematical operations

```
System.out.println(Math.max(5, 10));
System.out.println(Math.min(55, 100));
System.out.println(Math.abs(-77));
System.out.println(Math.ceil(3.55D));
System.out.println(Math.floor(3.55D));
System.out.println(Math.pow(2, 10));
System.out.println(Math.random());
System.out.println(Math.round(9.99D));
System.out.println(Math.sqrt(81));
System.out.println(Math.PI);
System.out.println(Math.E);
```

Math.PI indicates a reference to a *constant*. A constant is similar to a variable, but it has a value assigned that cannot be changed. In addition, the names of constants are written in capital letters.

The result of running the above code:

```
10
55
77
4.0
3.0
1024.0
0.20464094804059307 ①
10
9.0
3.141592653589793
2.718281828459045
```

① this value differ for each run; the random() function returns a pseudo-random value within the range of <0, 1).

Your exercise now it to test the above for other values and using other methods.

Wrappers of simple types

Apart from the simple types mentioned before, everything in Java is an object. For each simple type, its equivalent, that is, a wrapper class, was created in Java.

Wrappers of simple types

simple type	complex type
byte	Byte
short	Short
int	Integer

simple type	complex type
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Example of creation and application

```
byte byteValue = 56;
Byte byteObject = new Byte(byteValue);
short shortValue = 89;
Short shortObject = new Short(shortValue);
int intValue = 256;
Integer integerObject = new Integer(intValue);
long longValue = 5500L;
Long longObject = new Long(longValue);
float floatValue = 3.141592F;
Float floatObject = new Float(floatValue);
double doubleValue = 2.718281828D;
Double doubleObject = new Double(doubleValue);
char charValue = 'a';
Character characterObject = new Character(charValue);
boolean booleanValue = true;
Boolean booleanObject = new Boolean(booleanValue);

System.out.println("byteValue = " + byteValue);
System.out.println("byteObject = " + byteObject);
System.out.println("doubleValue = " + doubleValue);
System.out.println("doubleObject = " + doubleObject);
System.out.println("charValue = " + charValue);
System.out.println("characterObject = " + characterObject);
System.out.println("booleanValue = " + booleanValue);
System.out.println("booleanObject = " + booleanObject);
```

The result is:

```
byteValue = 56
byteObject = 56
doubleValue = 2.718281828
doubleObject = 2.718281828
charValue = a
characterObject = a
booleanValue = true
booleanObject = true
```

Variables xxxObject are object-oriented, we can execute methods on them, while we cannot do it on xxxValue variables because they are variables of primitive types, that is, they only have a value.

Example of invocation of a few methods

```
double doubleValueFromByte = byteObject.doubleValue();
float floatValueFromInteger = integerObject.floatValue();
char charValueFromCharacter = characterObject.charValue();
boolean booleanValueFromBoolean = booleanObject.booleanValue();
```

Autoboxing and unboxing

Thanks to the built-in *autoboxing* and *unboxing* mechanisms, we can shorten the record and conversion between primitive types and the corresponding object-oriented types (wrappers).

autoboxing

automatic conversion of simple types into complex ones. Where, for example, an Integer type is expected, we can provide e.g. value 5 or a variable of the int type, and it will be automatically converted into the Integer type. The same applies to other types.

Example of autoboxing

```
Byte byteObject = 56;
Short shortObject = 89;
Integer integerObject = 256;
Long longObject = 5500L;
Float floatObject = 3.141592F;
Double doubleObject = 2.718281828D;
Character characterObject = 'a';
Boolean booleanObject = true;
```

unboxing

automatic conversion of object-oriented types to primitive ones. Where, for example, a double type is expected, we can provide a variable of the Double type. It will be automatically converted into the double type. The same applies to other types.

Example of unboxing

```
byte byteValue = new Byte((byte) 56);            ①
short  shortValue  =  new  Short((short)  89);   ②
int intValue = new Integer(256);
long longValue = new Long(5500L);
float floatValue = new Float(3.141592F);
double doubleValue = new Double(2.718281828D);
char charValue = new Character('a');
oolean booleanValue = new Boolean(true);
```

① constructor Byte(byte value) requires passing a byte type value—it is required to cast the int type values to byte: (byte)

② constructor Short(short value) requires passing the short type value—it is also required to cast the int type value to short: (short)

Pay attention to null

The ranges of values for basic types and their wrappers are the same. The difference lies in the default values: for object-oriented types it is null, which has no equivalent (there is no such value) in primitive types.

When converting object-oriented types to primitive ones (*unboxing*), pay attention to whether **the variable value is** null. If this is the case, an error will appear.

An example of incorrect unboxing

```
Integer integerObject = null;
int intValue = integerObject;
```

Result:

```
Exception in thread "main" java.lang.NullPointerException
```

```
Integer a = 0;
Integer b = null;
```

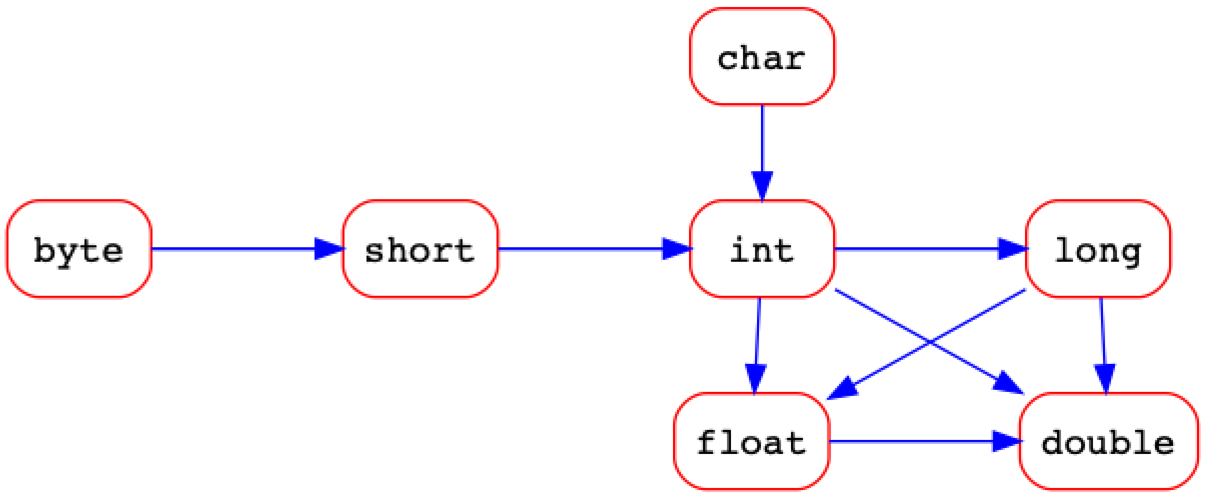
Variables a and b are not the same, 0 and null are completely different values. Variable a indicates the object of the Integer class that stores value 0. Variable b does not indicate any object!

You can compare a variable b to a remote control without a receiver so it has nothing to control.

Conversion of types

Java copes quite well with adjusting the types. Only in some cases we must use additional mechanisms to help it.

Diagram of conversion of types



Let us look at types: byte, short, int and long. Their ranges increase: byte is included in short, short in int, and int in long. Any value from the byte range will fit in the short range, etc. However, the result for the other direction is not so obvious. For example, if you store a small value in an int type variable (that fits in short or byte), you can *cast* an int type variable to a variable with a smaller range.

Conversions from a smaller range to a larger one happen automatically (this is indicated by the direction of the arrows).
Conversions from a larger range to a smaller one must be done consciously using the projection operator: (<typ>).

Example of automatic conversions to extend the type

```
byte b = (byte) 56;
short s = (short) 89;
int i = b;               ①
long l = s;              ②
float f = i;             ③
```

① byte type is extended automatically to the int type
② short type is extended automatically to the long type
③ int type is extended automatically to the float type

Example of an explicit narrowing conversion

```
byte a = 127;            ①
byte b = (byte) 130;     ②
float c = 3.14F;

int d = 5;
byte e = (byte) d;       ③
long f = (long) c;       ④

System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("f = " + f);
```

① there is no need to cast, because value (literal) 127 is in the range of the byte type

② casting is necessary because value 130 is out of range; pay attention to the result!

③ casting is necessary, because Java can only “see” the int type of variable c (it “cannot see” the value) and requires explicit casting

④ casting is necessary, because the range is exceeded; besides, the fractional part is lost.

The result is as below:

```
a = 127
b = -126
f = 3
```

Conversion from complex types to simple types

Objects of simple-type wrapper classes provide a number of methods for converting from complex types to simple types.

```
Byte, Short, Integer, Long, Float, Double
      • byteValue()
      • shortValue()
      • intValue()
      • floatValue()
      • doubleValue()
Character
      • charValue()
Boolean
      • booleanValue()
```

Conversion from simple to complex types

Wrapper classes offer methods that enable creating objects based on values or simple-type variables.

Byte.valueOf(byte b)
: creates a Byte class object based on the b value

Short.valueOf(short s)
: creates a Short class object based on the s value

Integer.valueOf(int i)
: creates an Integer class object based on the i value

Long.valueOf(long l)
: creates a Long class object based on the l value

Float.valueOf(float f)
: creates a Float class object based on the f value

Double.valueOf(double d)
: creates a Double class object based on the d value

Boolean.valueOf(boolean b)
: creates a Boolean class object based on the b value

Conversion of values stored in String

Wrapper classes (except Character) provide additional methods for converting values stored in a String type to a primitive type. These are methods in the parseXXX(String s) form.

Example of converting values stored in String

```
System.out.println(Boolean.parseBoolean("true"));
System.out.println(Boolean.parseBoolean("TRUE"));
System.out.println(Boolean.parseBoolean("1"));
System.out.println(Boolean.parseBoolean("0"));
System.out.println(Integer.parseInt("123"));
System.out.println(Double.parseDouble("3.1415"));
```

Result:

```
true
true
false
false
123
3.1415
```

Summary

What have you learned in this chapter?

How do we divide data types in Java?

We divide them into simple (primitive) and complex (object-oriented, reference) types.

What are the differences between simple and complex types?

Simple types have only a value. Complex types, in addition to storing value, provide methods for operations on them.

What is a variable?

A variable is like a "place", or a "container", for data. A variable always has its own type and only data of this type can be stored in it.

How do we compare variables (values) of primitive types?

We use the == sign, e.g., a == 5 or a == c.

How do we compare variables of object-oriented types?

We use the equals() method.

How can I check whether the object variable is null?

By comparing the variable to the null value, e.g. a == null.

What types of integers are offered by Java?

There are four integer types in Java: byte, short, int and long. All of them are *signed* (they have a sign). The long type is marked by adding letter L to the value.

Is any of the integer types default?

Yes, it is the int type.

What floating point types are offered by Java?

There are two floating point types: float and double; float is of single precision, while double is of double precision. The float type is marked by adding letter F to the value, while the double type is marked by adding letter D.

What is the character type?

The character (char) type enables storing exactly one character in the variable. These are characters in the *Unicode* standard, e.g. 'a', 'Z' or ' '. The char type is a type without a sign (*unsigned*), which means that the cores are non-negative numbers in the range from 0 to 65556.

What is the logical type?

The logical type (boolean) can have only two values: false or true. It is used to check logic conditions.

What are object-oriented types?

These are types that in addition to storing values also provide operations (methods). Java provides several thousand (over 4,000) types. Developers around the world provide more than ten hundred thousand types. When creating a class, you also create a new type.

What is the String class for?

The String class is used for storing strings and provides methods for operations on them. It features a short form of declaration and initialization:

```
String text = "I love Java!";
```

What does it mean that the String is *immutable*?

This means that the class does not change its status, but only creates a new instance with the changed value. If you perform an operation on the String class that modifies data (changes the string), then the original value is not actually changed; the method returns only the changed string.

What classes can be used to support the date and time?

These can be LocalDate, LocalTime or LocalDateTime classes. Also *older* classes can be often encountered: Date and Calendar.

What class can you use for simple mathematical operations?

It is the Math class.

What happens when I try to invoke a method on a variable that is null?

When attempting to invoke a method on a null reference (variable), a program runtime error will occur. In detail, the NullPointerException will appear.

What are simple type wrappers?

These are curtain classes for all simple types available in Java. They expand their capabilities by providing methods; they are a bridge between simple and object types.

What is *autoboxing* and *unboxing*?

Autoboxing is a mechanism for automatic conversion of simple types into complex ones.
Unboxing is a mechanism that works in the opposite direction. **Note!** The mechanism may return an error if you try to convert a variable that stores the null value to a simple type variable. In simple types there is no equivalent of the null value!

What is conversion of types?

Conversion of types is the assignment of one type to another. If you convert a *narrower* type into a *wider* one (e.g. short to int), then such conversion is safe. It is called *casting up*. If, on the other hand, you convert a *wider* type into a *narrower* one, then such conversion can be dangerous. In such cases you should use the casting operator: (<type on which you cast>).

Exercises

- Create (declare and initialize) several variables of primitive types based on things that surround you or can be found in your industry. Think over which type to use. Display the values.
- Try to assign a value out of range for the variable. See the result displayed. Try using casing operator ( ).
- Use wrapper classes and try to create object-oriented equivalents for all variables (of primitive types) from the previous exercise.
- Test the chosen methods (xxValue(), valueOf() and parseXXX()) from wrappers of simple types.
- Write a program (create a new class with the main method) that displays Hello, <Your name>! on the screen. Make your name stored in a variable of the String type.
- Write a program that displays your business card like the one below:

```
##############################
#                            #
#        John Smith          #
#  Sesame Street 345/7b      #
#      New York 10019        #
#                            #
##############################
```

  - use different signs, for instance:
    - - for horizontal lines
    - | for vertical lines
    - / and \ (or +) for corners
  - make the horizontal line be stored in a variable; you will be able to declare its value **only once**, but use it twice
  - make the characters for vertical lines and corners also be stored in variables
  - create further variables for the *name, surname, street, building number, flat number, zip code*
  - add a variable (and its display) that stores the phone number
  - suggest further variables, e.g. *area code, address* (concatenation of *street*, *building number* and *flat number*)
  - add boolean type variable specifying whether to display the frame; if it has a false value, then do not display it
  - create any variable and try to use it without initialization; observe the error
- Create a few String variables and test the presented methods that operate on strings.
- Create a few variables of the LocalDate, LocalTime and LocalDateTime types and test invoking a few methods on them. Display the results.

  A few sample methods for LocalDate*—find similar ones for other types*

    - `LocalDate.now()`
    - `LocalDate.of(2019, 04, 10)`
    - `LocalDate.now().plusDays(12)`
    - `LocalDate.now().isLeapYear()`
    - `LocalDate.parse("2019-04-10")`
    - `LocalDate.of(2019, 04, 10).isAfter(LocalDate.now())`
    - `LocalDate.now().getDayOfMonth()`
    - `LocalDate.now().getDayOfYear()`
    - `LocalDate.now().getYear()`
    - `LocalDate.now().lengthOfMonth()`
    - `LocalDate.now().lengthOfYear`
- Test the methods of the `Math` class.

Operators

You can perform various actions and operations on variables and literals.

Arithmetic and concatenation operators

Arithmetic operations are used to perform mathematical operations. In the case of operations for different types, the result is the wider of types.

Mathematical operators

+
: addition, e.g. 5 + 4, x + y

-
: subtraction, e.g. 9 - 7, x - y

*
: multiplication, e.g. 3 * 6, x * y

/
: division, e.g. 15 / 3, x / y

%
: remainder (*modulo*), e.g. 13 % 5, x % y

++
: increase by 1 (*increment*), e.g. x++ (*post-increment*), ++x (*pre-increment*)

--
: decrease by 1 (*decrement*), e.g. x-- (*post-decrement*), --x (*pre-decrement*)

Assignment operators

=
: x = 10; (assigning value 10 to variable x; unlike in mathematics where it means equality)

+=
: x += 10; it is a shortened record of x = x + 10;

-=
: x -= 10; it is a shortened record of x = x - 10;

*=
: x *= 10; it is a shortened record of x = x * 10;

/=
: x /= 10; it is a shortened record of x = x / 10;

%=
: x %= 10; it is a shortened record of x = x % 10;

Example of pre- and post-increment and decrement

```
int x = 5;
System.out.print(x++);   ①
System.out.print(++x);   ②
System.out.print(--x);
System.out.print(x--);
```

① first, value x will be assigned to the print()function, then displayed, and then increased by 1
② value x will be increased by 1, then transferred to the function and then displayed.

Result:

```
5766
```

Operator of String concatenation

Operator + for the String types has a different action. It connects (*concatenates*) strings together.

Examples of String concatenation

```
String text = "Alice" + " " + "has" + " " + " "a" + "cat";                    ①
String anotherText = text + " and a cat has Alice"                            ②
int count = 5;
System.out.println("Alice has " + count + " cats");                           ③
```

① combining "Alice" with " ", and then with "has" etc.
②combining a String text type variable with the text " and a cat has Alice!"
③ combining two texts with the int variable, it has been automatically converted into String

Comparison operators

Operators of comparison (relation) are used for comparing values. The result of such operation is a logical value true or false.

==

    equal to, e.g. 5 == 6, x == y

!=

    not equal to, e.g. 5 != 6, x != y

<

    less than, e.g. 5 < 6, x < y

<=

    less than or equal to, e.g. 5 <= 6, x <= y

>
: greater than, e.g. 5 > 6, x > y

>=
: greater than or equal to, e.g. 5 >= 6, x >= y

Examples of the use of comparison operators

```
int i = 8,  j = 8;
System.out.println(i == j);
System.out.println(i != j);
System.out.println(i < 24);
System.out.println(j != 5);
System.out.println(i >= 8);
```

Example of comparing simple types with object-oriented types

```
int fiveValue = 5;
Integer fiveObject = 5;
Integer fiveObjectByNew = new Integer(5);
Integer anotherFiveObjectByNew = new Integer(5);

System.out.println(fiveValue == fiveObject);
System.out.println(fiveValue == fiveObjectByNew);
System.out.println(fiveObject == fiveObjectByNew);
System.out.println(fiveObjectByNew == anotherFiveObjectByNew);

System.out.println(fiveObject.equals(fiveObjectByNew));
System.out.println(fiveObject.equals(anotherFiveObjectByNew));
System.out.println(fiveObjectByNew.equals(anotherFiveObjectByNew));

int twoHundredValue = 200;
Integer twoHundredObject = 300;
System.out.println(twoHundredValue == twoHundredObject);
```

Result:

```
true
true
false
false
true
true
true
false
```

The results may be a bit surprising. Particularly it may come as a surprise that for value 5 it is true, while for value 200 it is false. Explanation of this issue goes beyond the scope of this Handbook .

Remember!

- Primitive types are compared using the == sign
- Object-oriented types are compared using the equals() method
- When comparing a primitive type with an object-oriented type, we also use the equals() method to invoke it on an object variable, and pass a primitive type variable as an argument

Operator = means assignment.
Operator == means comparison.

```java
int i = 8,  j = 8;
Integer k = new Integer(8);
Integer l = new Integer(8);
System.out.println(i == j);
System.out.println(k.equals(l));
System.out.println(l.equals(j));
```

Logical operators

Logical operators are used to combine logical conditions. The result is also a logical value `true` or `false`.

&&

logical conjunction, AND, e.g. x && y

The truth table for the && operation

x	y	x && y
true	true	true
true	false	false
false	true	false
false	false	false

||

logical alternative, OR, e.g. x || y

Truth table for the || operation

x	y	x || y
true	true	true
true	false	true
false	true	true
false	false	false

!

logical negation, NOT, e.g. !x

Truth table for the `!` *operation*

x	`!` **x**
`true`	`false`
`false`	`true`

Summary

What have you learned in this chapter?

What types of operators are there in Java?

Operators are divided into:

- arithmetic
- comparisons
- logical

What arithmetic operators do we use?

Arithmetic operators are as follows:

- + — addition
- - — subtraction
- * — multiplication
- / — division
- % — remainder (*modulo*)
- ++ — increase by 1
- -- — decrease by 1

What is the *modulo* (%) operator?

The *modulo* operator (%) returns the remainder of the division. For instance:

- 7 % 5 = 2 (digit 5 can be included in digit 7 once, and the "left over" is 2)
- 7 % 9 = 7 (digit 9 can be included in digit 9 zero times and the "left over" is 7)

What are the ++ and -- operators and how do they operate?

These are operators that (respectively) *increase by 1* and *decrease by 1.*

- ++ — increase by 1 (*increment*); it can be x++ (*post-increment*) or ++x (*pre-increment*)
- -- — decrease by 1 (*decrement*); it can be x-- (*post-decrement*) or --x (*pre-decrement*)

What are abbreviated assignment operators?

These are operators that enable performing an action and changing the value of the variable at the same time.

- += — addition with assignment
- -= — subtraction with assignment
- *= — multiplication with assignment
- /= — division with assignment

- %= *modulo* with assignment

How does the “Strings” concatenation operator work?

The operator of Strings concatenation (+) combines two texts together to form one new text.

What are the comparison operators?

The comparison (relation) operators are as follows:

- == — equal to
- != — not equal to
- < —less than
- <= —less than or equal to
- > — greater than
- >= — greater than or equal to

What are the logical operators?

The logical operators are as follows:

- && — logical AND (*conjunction*)
- || — logical OR (*alternative*)
- ! — NOT, logical false (*negation*)

Exercises

- Declare a few variables of various types and test invoking the following operations on them: +, -, *, /, %.
- Declare an int type variable and test the ++ and -- operations both in their pre- and post- variations.
- Declare two variables corresponding to the sides of a rectangle and count its field and circumference.
- Declare two variables corresponding to the sides of a right-angled triangle and calculate its field.
- Declare the variable corresponding to the diameter of a circle and count its field and circumference.
- Declare three variables corresponding to the sides of a triangle and check whether it is right-angled.
- Declare a variable corresponding to age; use a ternary operator to display: *adult* or *underage*.

Conditional statements

Until now, the code we wrote was executed sequentially from top to bottom; unconditionally. However, in programming the logics of our application very often depends on some conditions. For example, if the amount to be paid exceeds 1000, we would like to charge a discount. Otherwise, the amount will remain unchanged. To accomplish this, we can use *conditional statements*.

The if statement

The if conditional statement checks the logical condition contained between brackets (and). If the condition **is met** (is true), the statement executes the code contained between brackets { and }.

Structure of the if *statement*

```
if (<logical-condition>) {
    // operations
}
```

Example

```
if (totalPrice > 1000) {
    totalPrice = totalPrice - discount;
}
```

Please note the indentation of the code in the line in the if body. Similarly as in the code in the main function, also in conditional statements indentation is applied.

Logical condition

A logical condition is a statement which after calculation returns value `true` or `false`. It is placed in round brackets.
Examples of correct logical conditions are as follows:

- age > 5
- price == 7.45
- dayNo != 7
- !active (where active is of the boolean type)
- true
- false
-

Examples of incorrect logical conditions:

- age + 1
- price = 7.45
- dayNo % 2
- 1
- “Alice has a cat”

A logical condition can be complex and consist of several expressions, but the result must still be a logical value.

Example of a complex logic condition

```
if ((age > 18 && dayNo == 6) || amount > 1000) {

}
```

The if-else statement

The code after the else statement (included in brackets { and }) is invoked when the logical condition of the if statement **is not met** (is false).

Structure of the if-else statement

```
if (<logical-condition >) {
    // operations
} else {
    // operations
}
```

Example of the if-else statement

```
if (age >= 18) {
System.out.println("You are adult");
} else {
    System.out.println("You are underage");
}
```

The if-else if-else statement

The else if structure is used to combine conditional statements and works just like the connection of if and else.

Structure of the if-else if-else statement

```
if (<logical-condition-1>) {
    // operations
} else if (<logical-condition-2>) {
    // operations

  ...

} else {
    // operations
}
```

Example of the if-else if-else statement

```
if (WeekDayNumber == 7) {
    System.out.println("Today is Sunday—holiday!");
} else if (WeekDayNumber == 6) {
    System.out.println("Today is Saturday—free!");
} else {
    System.out.println("Today is a business day—work!");
}
```

Pay attention to brackets

Braces { and } are not required, but it is ADVISABLE to ALWAYS use them, even when the if is is followed by one statement only. A code with one statement after if will be working properly.

```
if (totalPrice > 1000)
    totalPrice = totalPrice - discount;
```

But if we would like to add, for example, a discount from the seller in the next line, unfortunately the code will not work as we would expect—the discount from the seller will always be counted unconditionally. In addition, the indentation in the second line of the if statement can be confusing.

```
if (totalPrice > 1000)
    totalPrice = totalPrice - discount;
    totalPrice = totalPrice - dealersDiscount;
```

To make the code work as intended, we must enclose it in brackets.

```
if (totalPrice > 1000) {
    totalPrice = totalPrice - discount;
    totalPrice = totalPrice - dealersDiscount;
}
```

Ternary operator

A ternary operator is also called *a conditional operator.* It is similar to the if-else statement, but it returns the value. In addition, it can be seen as an abbreviated record of this statement.

Structure of a ternary operator

```
<logical-expression> ? <value-1> : <value-2>
```

If the value of the *logical expression* is true, the *value-1* expression is calculated and the operator returns its result. If the *logical expression* is false, then the *value-2* expression is calculated and the operator returns its result.

Example of a ternary operator

```
String text = age >= 18 ? "adult" : "underage";
System.out.println("You are " + text);
```

The above logic can also be expressed with the use of the if-else statement.

Example of the if-else statement corresponding to the ternary operator

```
String text;
if (age >= 18) {
    text = "adult";
} else {
    text = "underage";
}
System.out.println("You are" + text);
```

Note

```
<logical-expression> ? <value-1> : <value-2>
```

Usually *value-1* and *value-2* are of the same type, e.g. int or String as in the previous example. However, in the example below:

```
System.out.println(true ? 1 : 3.1415);
```

although the *logical expression* is true (is unconditionally true), the screen will display value 1.0 and not 1 as initially expected.
The ternary operator works in such a way that the returned type is the common (narrowest) type, in this case it is double. If we would like to assign the result of this operation, we could write, for example:

```
double result = true ? 1 : 3.1415;
```

The switch statement

Another type of conditional statement is the switch statement. It is useful in situations where there are many options to choose from; then the if-else statement may be ineffective.

Structure of the switch statement

```
switch (<variable>) {
    case <value-1>:
        //
        operations
        break;
    case <value-2>:
        //
        operations
        break;

    ...

    default:
        // operations
}
```

Example of rolling the dicw

```
int valueOnTheDice = 5;
switch (valueOnTheDice) {
    case 1:
        System.out.println( “Outcome is 1”);
        break;
    case 2:
        System.out.println( “Outcome is 2”);
        break;
    case 3:
        System.out.println( “Outcome is 3”);
        break;
    case 4:
        System.out.println( “Outcome is 4”);
        break;
    case 5:
        System.out.println( “Outcome is 5”);
        break;
    case 6:
        System.out.println( “Outcome is 6”);
        break;
    default:
        System.out.println(”Error or the dice has landed on the edge!”);
}
```

For the given value of the valueOnTheDice = 5 variable, the following result will be displayed on the screen:

```
Outcome is 5
```

If we change the value of the valueOnTheDice variable for instance to 100, the following text will be displayed:

```
Error or the dice has landed on the edge!
```

The case *variable and values* can be as follows:

- constant expressions of byte, short, char and int types
- relevant wrapper classes: Byte, Short, Character and Integer
- enumeration constants (Enum)
- character Strings

The default section is optional. It is executed when none of the values in the case can be adjusted.

The break statements are also optional, but they are very significant in this structure. I will present it using an example with dice. Let us remove the break statement from the case statement:

```
int valueOnTheDice = 5;
switch (valueOnTheDice) {
    case 1:
        System.out.println("Outcome is 1");
    case 2:
        System.out.println("Outcome is 2");
    case 3:
        System.out.println("Outcome is 3");
    case 4:
        System.out.println("Outcome is 4");
    case 5:
        System.out.println("Outcome is 5");
    case 6:
        System.out.println("Outcome is 6"); default:
        System.out.println("Error or the die has landed on the edge!");
}
```

For the provided value of the valueOnTheDice = 5 variable, the following text will be displayed on the screen now:

```
Outcome is 5
Outcome is 6
Error or the dice has landed on the edge!
```

Why has this happened? The switch statement works so that as soon as it matches the case value, it executes all statements from all the case sections (including default), until it encounters a possible break statement. It may look a bit unintuitive, but remember about this and use the break statement properly.
When can such an operation be useful? For example, when for the value of pips on the dice we want to display the information whether the result was even. Then we can apply two break statements and group the case values together.

```
int valueOnTheDice = 5;
switch (valueOnTheDice) {
    case 1:
    case 3:
    case 5:
        System.out.println("Odd number of pips");
        break;
    case 2:
    case 4:
    case 6:
        System.out.println("Even number of pips");
        break;
    default:
        System.out.println("Error or the dice has landed on the edge!");
}
```

For values 1, 3 and 5, the following text will be displayed:

```
An odd number of pips has been obtained
```

For values 2, 4 and 6, the displayed text will be as follows:

```
An even number of pips has been obtained
```

Summary

What have you learned in this chapter?

What are the conditional statements in Java?

These are the if and switch statements. The if statement has a few forms: if, if-else and if, else if-else.

What is a *logical condition*?

A logical condition is an expression that (after a possible calculation) returns the true or false value.

What types can be used in the case value in the switch statement?

These can be as follows:

- byte, short, char and int types
- relevant wrapper classes: Byte, Short, Character and Integer
- enumeration constants (Enum)
- character strings (String)

What is a *ternary operator*?

This is an abbreviated form of the if-else statement, yet it returns the value. It is expressed as:

```
<logical -expression> ? <value-1> : <value-2>
```

When is it advisable to use the switch statement instead of the if statement?

When we have many options to choose from or there is one option for multiple values.

Exercises

Loops

When writing a program, we often need to perform a certain operation multiple times, e.g. display a **Java** string three times or perform an operation until a condition is met, e.g. to withdraw coins from the wallet until we collect the required amount. To repeat the operations in Java, we can use a loop. There are several of them.

The for loop

The for loop is used to iterate (that is, to repeat statements or a group of statements).

Structure of the for loop

```
for (<initialization>; <logic-condition>; <step>) {
    // operations
}
```

Example of a loop that displays text Java four times

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

The result of running the program

```
Java
Java
Java
Java
```

The loop is executed as long as the ***logical condition*** is met; in this case, until i is less than or equal to 4.

The ***initialization*** section is performed only once: at the beginning, before the loop is executed.

The ***step*** section is performed at the end of each turn of the loop; after each ***operations*** section.

To better illustrate this, I will write the previous code in a different form.

An example of writing the same loop in other way

```
int i=1;                              ①
for ( ; i<=4; ) {
    System.out.println("Java");       ②
    i++
}
```

① the assignment (initialization) of the value to the variable takes place at the beginning, before the loop is started

② after each operation (in this case, after the text is displayed), the *step* is performed.

The above example works in the same way as the previous one.

For loops, it is accepted to use short variable names (e.g. i, `j', `k' etc.) as *loop counters*. In other cases, it is recommended to use more descriptive variable names.

The for loops are used primarily when we know how many times the operations are to be performed. If necessary, we can also save the for loop so that it can be repeated endlessly.

Example of an infinite loop

```
for ( ; ; ) {
    System.out.println("Java");
}
```

Now, the word **Java** will be displayed infinitely on the screen. To end the operation of this program, press the red square on the right of (▶).

Note

The for loop may not be executed even once if the first logical condition is false!

```
for (int i=5; i<3; i++) {
    System.out.println(i);
}
```

The advantage of the for loop is that during iteration we have access to the current value of the variable (the so-called *loop counter*), in our case it is variable i. For example, we can quickly enter all values from 1 to 10.

Example of a loop that writes numbers from 1 to 10

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

The result displayed on the screen will be as follows:

```
1
2
3
4
5
6
7
8
9
10
```

We can use the print() function (instead of println()) to display the values in one line.

Example of using the print() function

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

We will then see on the screen:

```
1 2 3 4 5 6 7 8 9 10
```

These values can also be displayed from the end, that is, from 10 to 1.

Example of a loop with decrement

```
for (int i=10; i>=1; i--) {
    System.out.print(i + " ");
}
```

Result:

```
10 9 8 7 6 5 4 3 2 1
```

You can also perform some calculations for the i variable. For instance, you can display individual numbers and their squares.

Example of using a loop counter for additional calculations

```
for (int i=1; i<=10; i++) {
    System.out.println("i=" + i + "i^2=" + (i*i));
}
```

The result will be as follows:

```
i=1, i^2=1
i=2, i^2=4
i=3, i^2=9
i=4, i^2=16
i=5, i^2=25
i=6, i^2=36
i=7, i^2=49
i=8, i^2=64
i=9, i^2=81
i=10, i^2=100
```

With the loop, you can easily sum all numbers from 1 to 10

Example of the operation of adding numbers in a loop

```
int sum = 0;
for (int i=1; i<=10; i++) {
    sum += i;
}
System.out.println("Sum = " + sum);
```

Result:

```
Sum = 55
```

The use of conditional statements with loops opens up further possibilities. For example, you can add only even numbers from 1 to 10

Example of the operation of adding numbers in a loop using a conditional statement

```
int sum = 0;
for (int i=1; i<=10; i++) {
    if (i%2 == 0) {   ①
        sum += i;
    }
}
System.out.println("Sum = " + sum);
```

① you can define any restrictions as needed by changing this condition.

Result:

```
Sum = 30
```

A step does not need to change by 1. You can, for example, write all multiples of digit 5 in the range from 5 to 100.

Example of a loop with a step incremented by 5

```
for (int i=5; i<=100; i=i+5) {
    System.out.print(i + " ");
}
```

The result will be as follows:

```
5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
```

The break and continue commands

For loops, in addition to conditional statements, we can use statements that change the control flow.

To stop the iteration at a given moment, you can use the break statement that definitively terminates the loops. This can be useful, for example, in a situation where you sum up values and you want to stop the iteration after a certain value is exceed. You can additionally display the number of the summed values and the value itself.

Example of a loop using the break statement

```
int sum=0;
int count=0;
for (int i=1; i<=200; i++) {
    sum += i;            ①
    count++;             ②
    if (sum >= 50) {     ③
        break;
    }
}
System.out.println("Sum = " + sum);
System.out.println("Number of summed values = " + count);
```

① summing up the values
② increment of the number of summed values
③ if a sum greater than or equal to 50 has been achieved, the entire loop is stopped

If, on the other hand, you would like to ignore several turns of the loop for the set values, then you can use the continue statement. Let us assume you want to display numbers from 1 to 20, but without the values from 13 to 15. You can achieve this in the following way:

Example of a loop using the continue statement

```
for (int i=1; i<=20; i++) {
    if (i>=13 && i<=15) {     ①
        continue;              ②
    }
    System.out.print(i + " ");
}
```

① if the i value is between 13 and 15

② it interrupts the **current** turn (iteration) of the loop and goes to the next one

The (enhanced) for loop

In Java, there is one more kind of for loop, sometimes also called an "improved" or "enhanced" loop, or a for each loop. It allows you to browse (iterate) arrays and collections without using the index (loop counter).

Structure of the for each loop

```
for (<variable> : <collection/array>) {
    // operations
}
```

During each loop rotation, consecutive value from the *collection* or *array* is passed to the *variable* and you can access it in the individual iteration.

Example of iterating over a 10-element array

```
int[] digits = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
for (int i : digits) {
    System.out.print(i + " ");
}
```

This loop type is very convenient when you want to view the entire collection or array and you do not care about controlling how to do it, i.e. on defining the scope and order of the elements to which you want to have access.

When you use the for each loop but you would like to have access to a counter, you need to program it by yourself.

Example of the for each loop with its own counter

```
int[] digits = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int oddCount = 0;
int allCount = 0;
for (int i : digits) {
    allCount++;
    if (i%2 != 0) {
        oddCount++;
        System.out.println(i + " is odd");
    }
}
System.out.println("Count: all=" + allCount + ", odd=" + oddCount);
```

The result will be as follows:

```
1 is odd
3 is odd
5 is odd
7 is odd
9 is odd
Count: all=10, odd=5
```

The while loop

The while loop allows you to execute specific statements for as long as the *logical condition* for the loop is met (its value is true).

Structure of the while loop

```
while (<logical-condition>) {
    // operations
}
```

Example that lists numbers from 1 to 10

```
int max=10;
int i=1;
while (i<=max) {
    System.out.print(i + " ");
    i++;
}
```

The while loop statements can be never executed if the *logical condition* for the loop is false at the beginning.

The while loop is used most often when the number of repetitions is not known in advance, but you know the condition that must be met for the loop.

As for the for loop, the while loop also can be performed endlessly.

Example of an infinite while loop

```
while (true) {
    System.out.println("Java");
}
```

You can also apply conditional statements such as break and continue.

The do while loop

The do while loop (similarly as the while loop) allows you to execute specific instructions as long as the *logical condition* for the loop is met (its value is true).

Structure of the do while loop

```
do {
    // operations
} while (<logic-condition>);
```

Unlike the while loop, the do while loop **will always be executed at least once**, even if the *logical condition* is not met at the beginning.

```
do {
    System.out.println("Java is great!!!");
} while (false);
```

Example of the do while loop

The result will be as follows:

```
Java is great!!!
```

As in the case of the for and while loops, the do while loop also can be performed endlessly, and you can apply conditional statements such as break and continue.

Nested loops

It is possible to enclose a loop in another loop's body.

Example of nested loops

```
for (int i=1; i<=10; i++) {                ①
    for (int j=1; j<=20; j++) {            ②
        System.out.print("*");             ③
    }
    System.out.println();                  ④
}
```

① **outer loop**—it loops 10 times; it is responsible for generating individual lines

② **internal loop**—it loops 20 times; it is responsible for generating individual characters in the line

③ the print() method prints the * character as many times as the inner loop is executed

④ in this place, after the execution of all rotations of the inner loop, a new line mark is printed for one rotation of the outer loop.

Result:

```
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
```

Analyze the operation of these two nested loops carefully to understand the code above. Change the values of the i and j variables. Replace the print() function with println() (and opposite) and see how the operation of the program will change. Add conditional statements or the break and continue statements.

Thanks to the nested loops you can, for example, draw a "checkerboard".

```
for (int i=1; i<=10; i++) {
    for (int j=1; j<=10; j++) {
        if ((i % 2 == 0 && j % 2 == 1) || (i % 2 == 1 && j % 2 == 0)) {      ①
            System.out.print("#");
        } else  {
            System.out.print(" ");
        }
    }
    System.out.println();
}
```

① the # sign is to be printed alternately; if you are in an *even* row and an *odd* column or in an *odd* row and an *even* column. In other cases, a space is to be printed.

The result will be as follows:

```
 # # # # #
# # # # #
 # # # # #
# # # # #
 # # # # #
# # # # #
 # # # # #
# # # # #
 # # # # #
# # # # #
```

In programming, you will often encounter situations where the loop counter starts from 0. In IT, the 0 value is encountered much more often than in mathematics :)

Summary

What have you learned in this chapter?

What loops are there in Java?

There are several types of loops available:

- for
- (enhanced) for
- while
- do while

When should you use the for loop and when the while/do while loop?

We use the for loop most often when we know the number of elements or the number of repetitions of a given statement. In addition, we use it when we want to have an (easy) access to the loop counter. We use the while/do while loop when we do not know in advance how many times we will iterate since it depends on some factor.

What is the difference between the for loop and the (enhanced) for loop?

The for loops allows you to declare the iteration method by controlling the *loop count*. If we want only to go through a given set (without probing into the iteration method), we choose the for (enhanced loop that is also called the for eachloop.

What is the difference between the while loop and do while loop?

The do while loop will always be executed at least once, even if the logical condition is false at the beginning; the while loop will not be executed in this case.

Exercises

Arrays

An array is a data structure that allows you to store multiple elements of the same type. Instead of creating several, a dozen or several dozen variables that are somehow related to each other, you can group them to be stored in one array. You can access each element of an array through its *index* (position in the array), which is a number of the int type.

When creating an array, you must specify its size. The size of an array **cannot be modified**. If it turns out that you need a larger array, you must create a new (larger) one and copy the data from the smaller array to it.

Indexes of an array are **numbered from 0.** Thus, the first element is the element with *index* 0, and the last one is the element with *index* n-1, where n is the size of the array.

Elements of the array can be both primitive and object-oriented types. An array (of whatever type) **is an object**! Arrays can be single- or multidimensional.

Creating arrays

Pattern of array declaration

```
type[] variable;
```

Example of array declaration

```
int[] numbers;
boolean[] switches;
String[] names;
Double[] values;
```

Sometimes you can encounter alternative declarations of array variables. Equivalent declarations of array variables

```
int[] a;
int []b;
int c[];
```

[big] The recommended record is int[] a

Pattern of array declaration and initialization

```
type[] variable = new type[N];                                  ①
type[] variable = new type[]{value1, value2, ..., valueN};      ②
type[] variable = {value1, value2, ..., valueN};                ③
```

① creating an N-element array without filling it with data; each element now has a default value relevant for the array type

② creating an array and filling it in with data; based on their quantity, Java knows the size of the array
③ shortened (recommended) entry of the above declaration

Examples of array declaration and initialization

```
int[] numbers = new int[10];                                    ①
String[] labels = new String[2];                                ②
String[] names = new String[]{"Sandra", "Tom", "Kate", "Bart"}; ③
Double[] values = {3.1415D, 2.7182D};                           ④
```

① we have created a 10-element array of the int type; we have not filled it with values, therefore it has been filled with default values for the int type, that is 0

② we have created a 2-element array of the String type; we have not filled it with values, therefore it has been filled with default values for the String type, that is, null

③ we have created a 4-element array of the String type and provided 4 elements

④ we have created a 2-element array of the Double type and provided 2 values

What method of array creation should you choose?

- If at the time of creating the array you **know how many elements you want to store** (you know the array size, but you **do not know the values** yet), then use the following pattern:

  ```
  type[] variable = new type[N];
  ```

  e.g.

  ```
  int[] numbersFromLottery = new int[7];
  ```

- However, if at the time of creating the array you **know all the values** to be stored (their number is the size of the array), then use the following record:

  ```
  type[] variable = {value1, value2, ..., valueN};
  ```

  e.g.

  ```
  String[] seasons = {"Spring", "Summer", "Fall", "Winter"};
  ```

Operations on arrays

Let us try to first display the content of the created (declared and initialized) arrays.

Example of displaying the array contents

```
int[] numbers = new int[5];
int[] luckyNumbers = {1, 3, 7, 17, 21};
String[] names = new String[10];
String[] seasons = {"Spring", "Summer", "Fall", "Winter"};

System.out.println(Arrays.toString(numbers));
System.out.println(Arrays.toString(luckyNumbers));
System.out.println(Arrays.toString(names));
System.out.println(Arrays.toString(seasons));

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

Result:

```
[0, 0, 0, 0, 0]
[1, 3, 7, 17, 21]
[null, null, null, null, null, null, null, null, null, null]
[Spring, Summer, Fall, Winter]
[I@610455d6
[Ljava.lang.String;@511d50c0
```

To display the contents of the entire array, we have used the Arrays.toString() method. If this method is underlined in red on your screens, add the following line above the class declaration:

```
import java.util.Arrays;
```

It allows you to apply methods from the Arrays class in our code.

If you try to display the arrays with the System.out.println() method, you will get an poorly legible result.

You can also download and modify individual elements. Access to an element is obtained through its *index*.

Pattern of access to the element of the array

```
variable[index]
```

Example of operations on individual elements of arrays

```
int[] numbers = new int[5];
String[] seasons = {"Spring", "Summer", "Fall", "Winter"};

System.out.println(Arrays.toString(numbers));
System.out.println(Arrays.toString(seasons));

numbers[0] = 25;
System.out.println(Arrays.toString(numbers));
System.out.println(numbers[0]);
numbers[4] = -200;
System.out.println(Arrays.toString(numbers));

seasons[1] = "";
seasons[2] = null;
System.out.println(Arrays.toString(seasons));
System.out.println(seasons[3]);
```

Result:

```
[0, 0, 0, 0, 0]
[Spring, Summer, Fall, Winter]
[25, 0, 0, 0, 0]
25
[25, 0, 0, 0, -200]
[Spring, , null, Winter]
Winter
```

It is useful to know the size of the array (if, for example, you do not remember its size or you were only given an array variable). The size of the array is obtained by referring to a field called length.

Example of access to the array size

```
int[] numbers = new int[5];
String[] seasons = {"Spring", "Summer", "Fall", "Winter"};

System.out.println(numbers.length);
System.out.println(seasons.length);
```

Result:

```
5
4
```

Thus, the first element of the array is the element with *index* 0, and the last element is the one with *index* variable.**length** - 1

If you try to refer to an *index* outside the range (0, variable.length-1), the program will end its operation and an error will appear.

Examples of incorrect references

```
String[] seasons = {"Spring", "Summer", "Fall", "Winter"};
System.out.println(seasons[-1]);
System.out.println(seasons[5]);
```

Summary

What have you learned in this chapter?

What are arrays?

Arrays are data structures that allow you to store several elements of the same type. You can access individual elements of the array through the *index*.

What is an *index* in an array?

Index is the position of an element in an array. *Indexes* are numbered from 0. This means that the first element of the array is the element with *index* 0. The last element of the n-element array is the element with *index* n-1.

Is it possible to change the size of the array?

No, an array once declared has a fixed size that cannot be changed. If you need a larger (or smaller) array, you must create a new array and move elements to it.

How should I display the array contents?

To display the contents of the array, you can use the Arrays.toString(<arraya>) function, e.g.

```
String[] seasons = {"Spring", "Summer", "Fall", "Winter"};
System.out.println(Arrays.toString(seasons));
```

Remember about the import declaration:

```
import java.util.Arrays;
```

How should I get information about the size of the array?

Every array contains a field called length that defines its size.

Exercises

1. You have a 10-element array of the int type:

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

   Using one of the loops, write a code snippet that will display:

   a. all digits
   b. the first 6 digits
   c. the last 6 digits
   d. all even digits
   e. all digits with odd indexes
   f. all digits from backwards
   g. all digits except digit 5
   h. all digits to digit 7 inclusive
   i. all digits divisible by 3
   j. the sum of all digits
   k. the sum of digits greater than or equal to 4
   l. the smallest and largest digits

2.

Object-oriented programming

Java is an object-oriented language.

Basic concepts:

Class

A rule for an object. It is a template (pattern) according to which you can create various objects. It defines a new data type.

Object

An instance, an item, a specific example of the class. A programming way of presenting an entity.

Field

A feature of a given object describing a given property of an object in terms of its type.

Method

An operation performed on the class object.

The body of the class is enclosed in braces { i }. These are fields (status, data) and methods (operations, behavior). Java does not allow you to create fields and methods outside of the class! The class field can have a default value. If you do not specify a default value, it will be set by a compiler. Class names are capitalized; field and method names are lowercase.

```java
public class Car {

    private String model;
    private int productionYear;
    private String color;
    private boolean used = false;

    public Car() {

    }

    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }
}
```

Classes and objects

dsafa

Creating objects—constructors

Fields and methods

Getters and setters

The toString() method

The equals() method

Summary

What have you learned in this chapter?

How do we divide data types in Java?

Simple and complex types

How do we compare variables (values) of primitive types?

We use sign: ==

How do we compare variables of object-oriented types?

We use the equals() method.

Exercises

Conclusion

Object-oriented programming was the last topic covered by this Handbook. At this point, you should be able to use basic Java components and build simple programs from them. You already know that Java provides many types of data that you can use in your programs. If necessary, you can define your own data type and provide operations in it. These operations can use conditional statements, operators or loops.

Now you are ready for the next part of information about Java that have not been included in this Handbook:

- interfaces, abstract classes and enumeration types
- file management
- inheritance, composition and polymorphism
- collections
- cohesive and parallel programming
- exception mechanism
- functional programming

Literature

- James Gosling, Henry McGilton. The Java Language Environment. A White Paper. 1996.
- Code Conventions for the Java™ Programming Language. 1999.
- Cay S. Horstman. Java. Podstawy. Edition X. Helion. 2016
- Cay S. Horstman. Java. Techniki zaawansowane. Edition X. Helion. 2017.

Assignments

1. Write a program that will display a multiplication table to one hundred.

```
   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
```

For formatting, you can use the String.format() function:

```
String.format("%4s", <value-to-display>) ①
```

① %4s displays the value on 4 characters aligning to the right.

You can find more information about the format() function here

2. Extend the multiplication table with horizontal and vertical lines and borders composed of characters |, - and +. For the table to 9 it could look like below.

```
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 2 | 4 | 6 |
+---+---+---+
| 3 | 6 | 9 |
+---+---+---+
```

3. Write a program that would draw the following patterns
   a. Triangle 1

```
#
# #
# # #
# # # #
# # # # #
# # # # # #
# # # # # # #
# # # # # # # #
```

b. Triangle 2

```
# # # # # # # # 
# # # # # # # 
# # # # # # 
# # # # # 
# # # # 
# # # 
# # 
#
```

c. Triangle 3

```
              #
            # #
          # # #
        # # # #
      # # # # #
    # # # # # #
  # # # # # # #
# # # # # # # #
```

d. Triangle 4

```
# # # # # # # #
  # # # # # # #
    # # # # # #
      # # # # #
        # # # #
          # # #
            # #
              #
```

e. Square

```
# # # # # # #
#           #
#           #
#           #
#           #
#           #
# # # # # # #
```

f. Letter S

```
# # # # # # #
  #
    #
      #
        #
          #
# # # # # # #
```

g. Letter *Z*

```
# # # # # # #
          #
        #
      #
    #
  #
# # # # # # #
```

h. Hourglass

```
# # # # # # #
  #       #
    #   #
      #
    #   #
  #       #
# # # # # # #
```

i. Square with diagonals

```
# # # # # # #
# #       # #
#   #   #   #
#     #     #
#   #   #   #
# #       # #
# # # # # # #
```

j. Based on the above figures, propose your own ones; try also to fill in the selected parts of the figures.

4. Write a program multiplies the digits for a given number (as String) until the result is a one-digit number.

Example for "123"

```
123
123 → 1x2x3 = 6
```

Example for "277777788888899"

```
277777788888899
277777788888899 → 2x7x7x7x7x7x7x8x8x8x8x8x8x9x9 = 4996238671872
4996238671872 → 4x9x9x6x2x3x8x6x7x1x8x7x2 = 438939648
438939648 → 4x3x8x9x3x9x6x4x8 = 4478976
4478976 → 4x4x7x8x9x7x6 = 338688
338688 → 3x3x8x6x8x8 = 27648
27648 → 2x7x6x4x8 = 2688
2688 → 2x6x8x8 = 768
768 → 7x6x8 = 336
336 → 3x3x6 = 54
54 → 5x4 = 20
20 → 2x0 = 0
```

Use the code below to implement the reduce(String number) method

```
public class Reducer {

    public static void main(String[] args) {
        Reducer reducer = new Reducer();
        String numberToReduce = "<enter value here>";
        int reducedValue = reducer.reduce(numberToReduce);
        System.out.println("Number to reduce: " + numberToReduce);
        System.out.println("After reduce: " + reducedValue);
    }

    public int reduce(String number) {
        return 0;
    }
}
```

   a. Consider how to optimize this program. Hint: *digit zero.*

5. Write a program that checks who won a “Tic-Tac-Toe” game.

   a. The status of the board is presented as a String, e.g.

   ```
   OX_OOXO_X
   ```

   where:

   - X means *a cross*

        - 0 means *a circle*
        - _ means an empty field

   b. Replace the above entry with a two-dimensional array:

```
0X_
00X
0_X
```

   c. Determine who won or whether that was a tie.
   d. Sample input data:

```
X0XXX000X
X0XX00XX0
X0XX000X0
0X_00X0_X
X00X0X0X0
0X0X00XXX
```

   e. Adjust the program to handle uppercase and lowercase letters (x, o, X, O) and 0 (zero) as a *circle*.

6. Write a program that shortens the contents of text messages. The program should:
    - remove unnecessary blank characters at the beginning and end of the contents
    - remove empty characters between words and start each word with a capital letter

      Example of program operation

```
    Alice has a cat, and a cat has Alice!
AliceHasACat,AndACatHasAlice!

Hey, I will be back later tonight. Do not wait with dinner for me.
Hey,IWillBebackLaterTonight.DoNotWaitWithDinnerForMe.
```

   In addition, make the program display:

    - the number of characters of the original message
    - the number of characters after compression
    - the price for sending a text message, assuming that **1 text message** is **160 characters**, and each message costs 25 cents.

7. Write a program that validates the correctness of a given personal ID number. The personal ID number has the following form:

```
YYMMDDXXXXC
```

   where:

◦ YY—year
◦ MM—month
◦ DD—day
◦ XXXX—ordinal number
◦ C—control digit

Assign the values to variables, e.g.

```
YYMMDDXXXXC
abcdefghijk

          result = 1xa + 3xb + 7xc + 9xd + 1xe + 3xf + 7xg + 9xh + 1xi + 3xj + 1xk
```

If result % 10 = 0, the personal ID number is correct!

a. Suggest a signature for the method of validation of the personal ID number correctness

8. A Car class is provided

```
public class Car {
    private String model;
    private int productionYear;
    private String color;
    private boolean used = false;
}
```

a. Generate *getter* and *setter* methods for all fields; use key shortcuts Alt+Ins
b. Test the Car class using the CarApplication class

```
public class CarApplication {
    public static void main(String[] args) {
        Car audi = new Car();
        audi.setModel("A8");
        audi.setColor("czerwony");
        audi.setProductionYear(2018);
        audi.setUsed(true);
        System.out.println(audi.getModel());
        System.out.println(audi.getColor());
        System.out.println(audi.getProductionYear());
        System.out.println(audi.isUsed());
    }
}
```

c. Add a field to store the mileage (int mileage) to the Car program; set the default value to 0

d. Add (generate) a *getter* type method for this field (getMileage())
e. Add the drive(int mileage) method that increases the mileage
f. Test the program operation
g. Modify the drive(int mileage) method to set the used field to the appropriate value; if the mileage is positive, the used field should have value true
h. Test the operation

9. The Calculator i CalculatorApplication classes are provided

```java
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}
```

```java
public class CalculatorApplication {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(5, 12));
    }
}
```

a. Based on the add(int a, int b) method, implement the subsequent methods:
    i. int subtract(int a, int b), subtraction: a – b
    ii. int multiply(int a, int b), multiplication: a * b
    iii. double divide(int a, int b), division: a / b
    iv. boolean isPositive(int a), checks whether the number is positive
    v. boolean isNegative(int a), checks whether the number is negative
    vi. boolean isOdd(int a), checks whether the number is odd
    vii. int min(int a, int b), returns the smaller of the numbers
    viii. int max(int a, int b), returns the greater of the numbers
    ix. double average(int a, int b), returns the average for the numbers
    x. int power(int a, int x), returns a^m (a to the power m)
b. Based on the solution from point *a*, implement 3-argument versions of these methods (try to use 2-argument versions of these methods):
    i. int add(int a, int b, int c)
    ii. int subtract(int a, int b, int c)
    iii. int multiply(int a, int b, int c)
    iv. double divide(int a, int b, int c)
    v. int min(int a, int b, int c)

vi. int max(int a, int b, int c)

vii. double average(int a, int b, int c)

10. Implement a stack to store Integer type numbers

```
public class Stack {
    public Stack(int count) {}
    public void push(Integer e) {}
    public Integer pop() {}
    public boolean isEmpty() {}
    public boolean isFull() {}
    public String toString() {}
}
```

a. the constructor creates a count element array for storing Integer type elements
b. the push() method throws the element onto the stack (to the first free position). If the stack is empty, a relevant message is displayed and nothing happens
c. the pop() method deletes (returns and removes from the stack) the first element from the "top" of the stack. If the stack is empty, the method displays an appropriate message and nothing is done (null to be returned)
d. the isEmpty() methods checks whether the stack is empty, i.e. whether there is no element in the array
e. the isFull() method checks whether the stack is full, i.e. for example at a 5-element stack, all 5 array elements have a value (other than null)
f. the toString() method displays the stack contents
g. hint: enter an additional variable that stores the index of the current or next item in the array and modify its value when implementing the push() and pop() methods

11. Account and AccountApplication classes are provided

```
public class Account {
    private String name;
    private int balance;
}
```

```
public class AccountApplication {
    public static void main(String[] args) {
        Account account = new Account();
        account.setName("Premium Account");
        System.out.println("Name: " + account.getName());
        System.out.println("Account balance: " +
        account.getBalance());
    }
`
```

a. Set a default value 0 for the balance field
b. Add (generate) *getter* and *setter* type methods for the name field

c. Add (generate) a *getter* type method for the balance field
d. Add a (private) debit field of the boolean type that determines whether the account balance is negative; set the default value to false
e. Add the possibility to deposit and withdraw money

```
public void deposit(int amount) {

}

public void withdraw(int amount) {

}
```

i. Implement the above methods
ii. The withdraw method is to set the debit field to true when the account balance is negative
iii. Check the operation of the method

f. Add validation of the amount parameter in the deposit and withdraw methods;
   i. the methods are to perform the logics only when the amount value is positive
   ii. otherwise they are to display the following message: “The deposit/withdrawal amount must be positive!"
g. For the withdraw method, add the message display: “Negative account balance" if the debit field value is true
h. For the deposit and withdraw methods, add a summary display like the one below (e.g. for deposit and withdraw, respectively)

```
"Account balance: 300 | Deposit: 250 | After operation: 550"
"Account balance: 200 | Withdrawal: 500 | After operation: -300"
```

i. Add a function to support the maximum debit, e.g. 1000. If the amount after the operation is lower, do not execute withdrawal but display the message: “You cannot perform an operation exceeding the debit”

j. Implement the transfer method for transfers from the current account to another one.

```
public void transfer(Account other, int amount) {

}
```

i. The amount is to be withdrawn from the current account
ii. The amount should be paid to the other account

k. Add the toString method

```
public String toString() {
    return " Account{name: " + name + ", balance: " + balance + "}";
}
```

and invoke it as below:

```
public class AccountApplication {
    public static void main(String[] args) {
        Account account = new Account();
        account.setName("Premium Account");
        System.out.println(account);
    }
}
```