Java Fundamentals – coursebook

Welcome to the **Java – Fundamentals** module!

During this block you will learn about the basic mechanisms of operation of programming languages on the example of Java. Some of the topics that will be clarified during this block are:

- Data types, variables, constants, operators, casts
- The String class
- Conditional statements
- Loops
- Tables
- OOP (class, object, state, behavior)
- Fields, methods, constructors, packages, package import

Get ready for class

Check the list below if you have any programs that we will work with during this class:

✔ **Java OpenJDK 11**

✔ **IntelliJ IDEA Community**

#Java basics – Introduction

What is programming

We refer to "computer programming" when we design,create,test and maintain source code for computer programs or microcomputers (microelectronics).

"Source code" is written in the programming language which has it's own set of rules, it can be a modification of an existing program or something entirely new. Programming requires a lot of knowledge and experience in many different fields such as app design,algorithms, data structures, programming languages and tools, compilers or how the components of a PC work.

In programming the task of implementing the program itself is only one of the stages of development.

Why Java?

Here are a few reasons why beginners decide to use Java:

- High level object oriented language
- Platform independent (Write Once, Run Anywhere)
- Automatic memory administration
- Ease of use
- Popularity
- Big community
- A lot of literature
- Big interest on the job market

History of the Java Language

We can pinpoint the birth date of Java to the year 1991. It was then when Sun together with Patrick Naughton and James Gosling decided to create a small and easy language, which could be run on many platforms with different settings. The project was titled Green.

The first version of Java 1.0. was released in 1996. Unfortunately, it was not a big success and the Sun engineers were well aware of that. Mistakes were soon fixed and new libraries were added including a GUI and the **reflection** mechanism was fixed. All of this was done with Java 1.1.

The following versions of Java, up until the current one, are mainly based on adding new functionalities and increasing the performance of the standard libraries. Most revisions were made with version 5.0. where generic clsses and static import were introduced.

You can find more information on Java history in our Workbook – **Introduction to Java**

#The first program

Implementation

Similar to all other languages it is widely known that the easiest program to create is "Hello World!". It will show only one text – exactly "Hello World" – and nothing more. You can see the example below:

```
package pl.sdacademy.example;

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

A few words of explanation on the code above:

- `public class HelloWorld` – this is a declaration of the public class `HelloWorld`– the topic of classes and object oriented programming will be discussed in further chapters.
- `public static void main(String[] args)` – the 'main' method is the starting point of every application – this is where it all begins.
- `System.out.println("Hello World!")` – this will display the text given as its argument aka ("Hello World") and then it will go to the next line – by the use of the ' '.

Compilation and execution

If we installed the OpenJDK java compiler and installer correctly we only need to run the following commands from the Windows command line:

```
javac HelloWorld.java
```

```
java HelloWorld
```

The first command will run the program as seen in the file 'HelloWorld.java' and it will create a ready-to-go compiled code in the file `HelloWorld.class`.

The second command will run the application. Modern IDE will automatically run the following commands for us in the background.

As a result of executing those commands the terminal will display the following text:

```
Hello World!
```

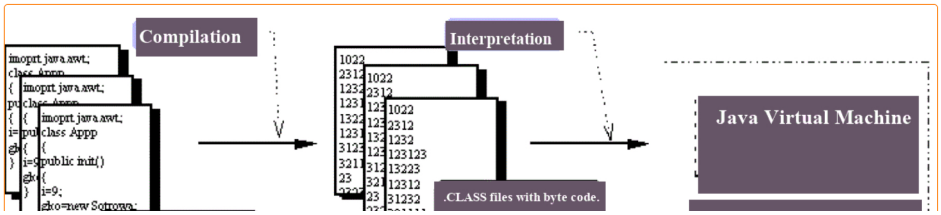
How does the compilation process work?

Compilation

The process of creating (implementation) and launching Java applications can be divided into several phases:

- Implementation – creating the source code of the application, i.e. typing down the computer program with the use of a specific programming language, describing the operations that the computer should perform on the data that is collected or received. The source code is the result of the programmer's work and allows to express in a human-readable form the structure and operation of the computer program.
- Compilation – our program written in Java language is run by a special tool called **compiler** on the so-called virtual Java machine (**JVM**). The compiler converts source code into machine code understood by JVM, so called **JVM. bytecode**. Besides that, the compiler is supposed to find lexical and semantic errors and optimize the code.
- Interpretation – this is the process of machine code analysis and its execution. These actions are performed by a special program called the **interpreter**.

The whole process is illustrated in the diagram below:



Installation of the Java Launcher

As programmers, we will write our Java code in `.java` files, and the compiler will translate it into bytecode and put it in `.class` files. The entire JVM environment and development tools can be downloaded and installed as **JDK (Java Develpment Kit)**. The JDK package includes both a bootable environment (**JRE – Java Runtime Environment**) and additional tools for developers, such as a compiler (`javac` – Java compiler).

Variables and types of data

What is a variable?

A variable is a programming construct which is composed of three basic attributes: symbolic name,storage location and value which, when you are browsing the source code, helps refer to it's actual source value or the physical storage location. The name is used to identify the variable which is why it is often called the **variable identification**. The variable is stored in the computer memory and it is identified by the address and data length. It's value corresponds to it's storage location.

Java identifies two phases of variable creation:

- Declaration – we create the variable name and set up it's type.
- Initialization – we assign a value to the variable.

Variable declaration

An example declaration of a variable can look like this:

```
private int number;
```

Here we have declared a whole number of a private type.

Similarly, the process of initializing a variable can be presented as follows:

```
number = 5;
```

The variable 'number' has been assigned the whole number of 5.

A more general declaration of a variable can be presented as such:

```
{access type} {type of variable} {variable identifier}
```

where:

- **access type** – these are called the access modifiers (optional)
- **type of variable** – describes the type,structure and value that the variable can take
- **variable identifier** – the name which helps identify the given variable

Based on access privileges we can distinguish:

- "Global" variables within a class – this class operates within the area of the class where it has been declared:

```
class ExampleClass {
    int myGlobal = 12;
    void someMethod() {
        // we can use the myGlobal variable
        System.out.print("My global variable: " + myGlobal);
    }
}
```

- "Local" variables with a method – the variable is accessible within the method where it was declared:

```
class ExampleClass {
    void someMethod() {
        int myLocalVariable = 5;
```

```
        System.out.print("My local variable: " + myLocalVariable);
    }
    int myGlobal = myLocalVariable; // Error - myLocalVariable is not visible outside of
the method where it was declared
}
```

- "Local variables" declared within [conditional statements] (conditional_statements.md) or as a counter in [loops] (loops.md), such as:

```
class MyExampleClass {
    void someExampleMethod() {
        if (someCondition) {
            int a = 1;
            // local variable declared within a conditional statement - it is visible
only within such an instruction
        }
        for (int i = 0; i < 10; i++) {
            // the variable 'i' is only visible within the loop
        }
    }
}
```

Final variables

A final variable cannot change it's value once it has been set. When we declare the variable we add the key word 'final' . An example is provided below:

```
final int finalVariable = 25;
```

The next code snippet will result in a compilation error since we will be trying to change the final variable value:

```
private void finaVariableSample() {
    final int finalVariable = 123;   // final variable declaration and Initialization
    final long anotherFinalVariable;
    finalVariable = 12;             // the attempt to change the final variable will
result in a compilation error
    anotherFinalVariable = 12345L;   // correct Initialization of a final variable

}
```

Final variables behave differently since they refer to an [object] (classes_objects.md) and not primitive types as seen above.

The key word 'final' means we are unable to change the value of it's reference but we can change the object state for instance by using such a definition of a class:

```
public class SimpleClass {
  private String name;

  public String getName() {
    return name;
  }

  public void setName(final String name) {
    this.name = name;
  }
}
```

We can execute this code:

```
public static void main(String[] args) {
  final SimpleClass simpleClass = new SimpleClass();
  simpleClass.setName("Michael");   //we are able to change the state of the object
  simpleClass = new SimpleClass();   // compilation error, we are unable to change the
reference
}
```

Types of data

What are types of data?

What are types of data?

Data types refer to the type, structure and the value range that a given literal,variable, final, argument, result of a function can take

Types of data in Java

We distinguish the given types of data in Java:

- numeric types
    - integer values (aka whole numbers)
    - floating points
- logic data types
    - true
    - false
- char data types
- string data types (aka text data types)

Java is a static based language as such:

- variable types are assigned during the program compilation.
- it is easy to detect errors during compilation.
- it is imperative to declare the types of variables before initializing them.

Given the rules provided above the application seen below will result in a compilation error:

```
String myVariable;
myVariable = 2;
```

Whole numbers

The types representing whole numbers are listed below:

- byte
- short
- int
- long

byte

The 'byte' type assigns 1 byte (8 bits) into memory and we can use it to write numbers ranging from -128 to 127 (2^8=256).

An example of `byte` declaration is given below:

```
byte myByteNumber = 125;
```

short

The 'short' assigns 2 bytes (16 bits) into memory and we can use it to write numbers ranging from -32768 to 32767.

An example of `short` declaration is given below:

```
short myShortNumber = -22556;
```

int

The 'int' type assigns 4 bytes into memory and we can use it to write numbers ranging from -2147483648 to 2147483647.

An example of `int` declaration is given below:

```
int myIntNumber = 1230000;
```

long

The 'long' type assigns 8 bytes (64 bits) into memory and we can use it to write numbers ranging from -2^63 to (2^63)-1 which is really a lot. In reality they are mostly used for identity occurrence in databases. When we declare logs we add the prefix 'L'. We can also use the small letter 'l' but for readability we should use the big 'L'.

An example of a `long` declaration is given below:

```
long myLongNumber = 254555455672L;
```

There are also the non-primitive data types or reference data types which correspond to primitive types. They enable the use of methods which help automate repetitive functions. Java does not have or use the 'Unsigned' type (with only positive values) and when we exceed the maximum value of a given type we move to it's negative range.

Floating point numbers

Floating point numbers consist of:

- float
- double

float

The 'float' type assigns 4 bytes into memory and we can assign numbers with accuracy of maximum 6 or 7 decimal spaces. When we declare the numbers we use the prefix 'F' or 'f'.

An example use of `float`:

```
float myFloatNumber = 12.0005f;
```

double

The 'double' type assigns 8 bytes into memory and we can assign numbers with accuracy of maximum 15 decimal spaces. When we declare the numbers we use the prefix 'D' or 'd'.

An example use of `double`:

```
double myDoubleNumber = 12.000000005d;
```

Important: We distinguish the whole from the floating-point by using a dot not a comma separator. We also need to remember that floating-point number types which represent values using mantissa and exponent (e.g. `float` or `double`) should not be used for financial calculations where accuracy is valued, because we are unable to calculate exact values of all numbers. In such case approximate values are used. One class that solves this issue is `BigDecimal`. There is another class `BigInteger` which is seen as a reference for whole numbers and can show all ranges of numbers.

Logic data types

Java has only one logical data type 'boolean' which takes two possible values:

- **true**, where something is true
- **false**, where something is false

We often use boolean values in [conditional statements] (conditional_statements.md) and [loops] (loops.md). We can therefore check if a given instruction can be executed or not.

An example use of a logical type:

```
boolean myFalseValue = false;
boolean myTrueValue = true;
boolean myBooleanValue = myFalseValue && myTrueValue; // myBooleanValue will have the
value 'false'
```

Char data type

The char data type is represented by 'char' in Java. We declare single characters by use of single quotes like this: 'a'. It is used to represent single characters using Unicode encoding.

A sample declaration of char can be seen below:

```
char signValue = 'y';
```

We can also represent special characters but they have to be preceeded by a " character such as:

- `\t` – tab
- – new line
- `\r` – carriage return

An example use of the '\t' char:

```
char tab = '\t';
```

String type

The 'String' type is used to represent text using Unicode. It is of **immutable** type and it's state can not be changed. It is also a type of object (you can read more in further chapters). It's value is represented by double quotes as seen in the example below:

```
String someText = "This is a simple text.";
```

Operators

In Java we distinguish between the following types of operators:

- Assignment Operators
- Arithmetic Operators
- Conditional Operators
- Equality and Relational Operators

Assignment Operators

With assignment operators we can assign (or give) a new value to a given variable. We distinguish:

1. The '=' operator gives a value to a given variable:

```
int intValue = 5; // from this point on the intValue variable has the value of 5
```

If following the initial assignment we wish to change it's value we can do so at any moment:

```
int someValue = 6; // we declare someValue at 6
someValue = 7;     // we change it's value to 7
someValue = 4;     // now someValue has the value of 4
```

2. The '+=' operator – this operator adds a value present at the right hand side to the variable and sums it up in the background automatically:

```
int a = 50;
a += 50;    // now the variable 'a' will have the value of 100
```

3. The '-=' operator is similar in use to '+='. It subtracts the variable by the value provided at the right hand side:

```
int a = 50;
a -= 40;    // after subtracting the variable a will have the value of 10
```

4. The '*=' operator works the same way as the other ones only the multiplication operator is used:

```
int a = 10;
a *= 10;    // after the operation the variable will have the value of 100
```

5. For the '/=' operator the logic is the same, only the operation in place is the division:

```
int a = 200;
a /= 100;    // after the division operator, the result will be 2
```

Arithmetic Operators

The arithmetic operators similar to their mathematical counterparts perform basic operations on provided numbers or variables.

1. The `+ - * / %` – represent the basic operation of:

+ Additive operator) **–** Subtraction operator ***** Multiplication operator / Division operator **%** Remainder operator

```
public class ArithmeticOperations {
    public static void main(String[] args) {
        int a = 5;
        int b = 10;
        int c = a + b;  // result: 15

        c = a - b;      // result: -5
        c = a * b;      // result: 50
        c = b / a;      // result: 2
        c = b % a;      // result: 0
    }
}
```

2. Incrementation operator – adds 1 (one) to a given number. It can be used in two ways: post and pre. It's the equivalent of 'i=i+1'

- **Post Incrementation** returns the value of a variable and then it modifies it:

```
int someVariable = 5;
System.out.print(someVariable++);   // At first the value of 5 will be printed, then it's
value will be changed to 6.
```

- **Pre Incrementation** at first the variable is modified then it returns it's value.

```
int someVariable = 5;
System.out.print(++someVariable);   // the value of 6 will be provided.
```

3. Decrementation – decreases the value by 1 (one) and it is used in two forms: post and pre. It is the equivalent of 'i = i-1':

```
int someVariable = 10;
System.out.print(someVariable--);   // the value of 10 will be printed out, after
execution the variable will have the value of 9.

someVariable = 15;
System.out.print(--someVariable);   // the value of 14 will be printed out.
```

Equality and Relational Operators

The equality and relational operators compare expressions and return it's logical values based on whether it's value is 'true' or not. The arguments for comparison can be numbers,strings,logical or object based. We most often use them within [conditional statements] (conditional statements.md). We can distinguish such operators:

- Equality '==' which checks whether two arguments are equal:

```
int a = 5;
int b = 6;
System.out.print(a == b);   // the result will be 'false' as 5 is not equal to 6.
```

- The '!=' operator check whether two arguments are not equal:

```
int a = 5;
int b = 6;
System.out.print(a != b);   //  the result will be 'true' as 5 is not equal to 6.
```

- The greater than '>' or greater or equal than '>=' checks whether the first argument is greater or greater or equal than the other one:

```
int a = 6;
int b = 6;
System.out.print(a > b);    // the result will be false
System.out.print(a >= b);   // the result will be  true
```

- The less than '<' operator or less or equal operator '<=' works this way:

```
int a = 5;
int b = 6;
System.out.print(a < b);     // this will return true
System.out.print(a <= b);    // this will also return true
```

Conditional Operators

They operate with arguments representing logical values which in turn also provide a logical result. We distinguish:

- && – **Conditional AND** this takes two `boolean` arguments and also return a `boolean` result. The arguments is true only when both logical arguments are true (**logical multiplication**). Here is an example:

```
boolean boolValue1 = true;
boolean boolValue2 = false;
System.out.print(boolValue1 && boolValue2);     // The result of false will be the
result.
```

- || – **Conditional OR**, also known as the logical sum. It's true when at least one of it's arguments is true. Here is an example:

```
boolean boolValue1 = true;
boolean boolValue2 = false;
System.out.print(boolValue1 || boolValue2);     // the result true will be provided
```

- ! – **Logical NOT**, also known as the negation. It can be interpreted as "not true that". An example is provided below:

```
boolean boolValue1 = true;
boolean boolValue2 = false;
System.out.print(!boolValue1);                   // the result false will be printed out
System.out.print(!(boolValue1 || boolValue2));     // the result false will also be
printed out
```

Conversion of types and projection

Classified conversion

Converting types is nothing more than replacing the type of a variable with another. If it is possible to do it in a safe way and without losing information (data). The compiler allows for the so called "type conversion". **classified conversion** in an automatic way. In practice, when writing code, we assign variables of one type to another type, e.g. all instructions shown below will be true:

Conversions of numerical types

Sometimes it is necessary to convert from one numerical type to another. The figure below shows the permitted conversion types:

are lossless, only those mentioned above may involve partial loss of data, e.g. the integer 123456789 consists of more digits than can fit into the `float` type, so in the example below we will lose some precision:

```
int n = 123456789;
float m = n;
System.out.println(m); // the value 1.23456792E8 is printed
```

When converting between **number types** we should remember the following rules:

- If one of the operands is of the `double` type, the other will also be converted to the `double` type.
- Otherwise, if an operand is of the `float` type, the other will also be converted to the `float` type.
- Otherwise, if an operand is of the `long` type, the other will also be converted to the `long` type.
- Otherwise both operands will be converted to the `int` type.

When converting types, we can generally accept the principle that a smaller type is converted to a larger bit capacity.

Casting

Sometimes we have to convert a numerical type with a higher bit capacity to a smaller one, e.g. the `double` type to the `int` type. This is of course possible, but it may involve the loss of some information. Conversions where information can be lost are called **casting**.

To perform a casting, place the target type name in round brackets before the name of the variable being cast. The general rule is as follows:

```
(type_converted) variable_smaller_capacity;
```

Example:

```
double n = 99.9989;
int m = (int) n;
System.out.println(m); // value 99 will be displayed
```

```
double n = 99.9989;
int m = (int) Math.round(n);
System.out.println(m); // value 100 will be displayed
```

A mistake will be shown in the construction below:

```
double n = 99.9989;
int m = n; // Error!
```

The above example will end with a compilation error, because we tried to automatically convert from floating-point type to integer `int`, which can't happen because of the risk of data loss (decimal places) – the programmer has to openly **shut down** types in this situation.

The following example shows the use of both conversion and projection:

```
int valA = 3;
int valB = 4;
double avgIncorrect = (valA + valB) / 2; // no projection, result is 3.0
double avgCorrect = ((double)valA + valB) / 2; // result is the expected value 3.5
```

In the last line of the above example, we project the value in the `valA` variable into the double type. Then we add the variables `double` and `int` to each other. A conversion from `int to` double`follows, which also results in`double`. In the next step, the resulting sum (which is of type` double`) is divided by 2. Another conversion (`int to `double`) takes place and the result is `3.5`.

Conditional Statements

What's the conditional statement?

A conditional statement is an instruction defined in the syntax of a specific programming language, allowing you to determine and change the order of execution of instructions contained in the source code. In Java there are several designs for conditional statements.

The IF statement

This is the simplest form of conditional statements. It checks the logical condition and if it is true, the instructions in its body (inside the block) are executed, otherwise they are omitted. The structure is as follows:

```
if (condition) {
    // Follow the instructions inside the block if the condition is true
}
```

A block diagram is shown below:

Example:

```
float temperature = 38.5f;
```

```
if (temperature > 36.6f) {
    System.out.print("You have a fever/heat state!");
}
```

In the example above, the output will be displayed: `You have a fever/heat state!`

IF ELSE

In the `if else` instruction, the code in the `else` block is executed if and only if the logical condition declared in parentheses `if(...)` is not met. The diagram of this construction can be written as follows:

```
if (condition) {
    // {\pos(192,220)}take the code for the condition that you've met
} else {
    // ...execute a code for an unfulfilled condition...
}
```

A block diagram is shown below:

Example:

```
float temperature = 36.6f;
if (temperature > 36.6f) {
    System.out.print("You have a fever/heat state!");
} else{
    System.out.print("You're healthy/healthy!");
}
```

It is also possible to build a mixed construction of `if... else if... else` with any number of blocks `else if`, which gives you the possibility to branch the expression to multiple conditions. The diagram of this construction can be written as follows:

```
if (condition1) {
    // execute the code for condition condition1 fulfilled.
} else if (condition2) {
    // execute the code for condition 1 not fulfilled and condition 2 fulfilled
}
... // " another block else if
} else {
    // execute the code if none of the preceding conditions are met
}
```

Example:

```
float temperature = 36.4f;
if (temperature >= 37.0f) {
    System.out.print("You have a fever/heat state!");
} else if (temperature >= 36.6f && temperature < 37.0f) {
    System.out.print("You're healthy/healthy!");
} else {
    System.out.print("You are weakened!");
}
```

In the example above, the output will be displayed: `You are weakened!`.

SWITCH instructions

The last schema we discussed can be replaced by a more convenient method, which is at the same time more readable, the `switch` instruction. It is characterized by the fact that:

- It covers multiple expressions of `if-else`
- It consists of many conditions
- It has a default `default` block if the other conditions are not met
- From Java SE7 it is possible to use the variable type `String`.

The design is as follows:

```
switch(variable) {
```

```
    case number one:
        // execute the code in case the variable = value1
        break;
    case number two:
        // execute the code in case the variable = value2
        break;
        // other cases...
    default:
        // ...perform if and only if none of the above conditions are met...
}
```

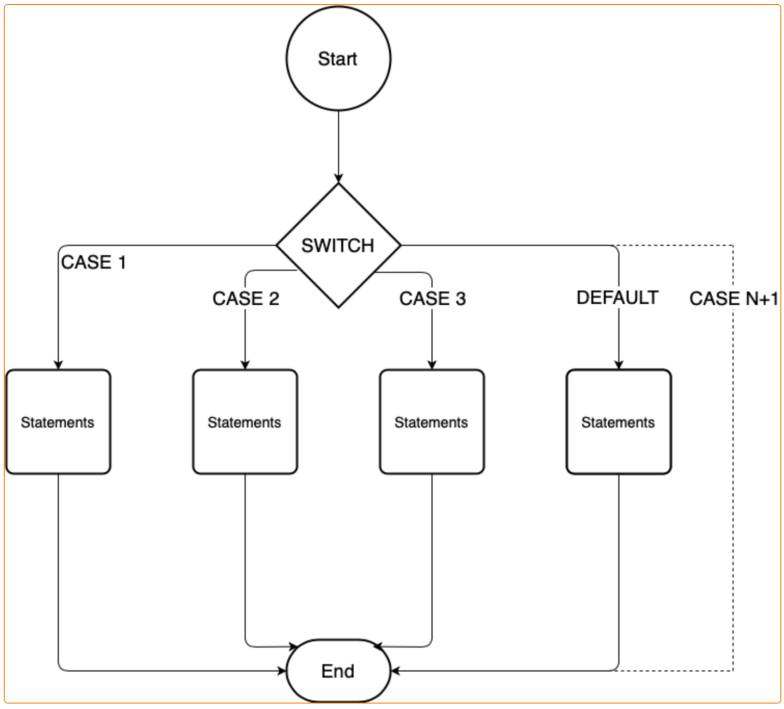
Pay attention to the keyword `break`:

- if the condition is met, the code block will be executed up to the `break` directive,
- if it's missing, a code block will be executed from the next `case`.

If none of the cases of the `case` is fulfilled, the code block contained in the `default` will be executed. The limitation of the `switch` instruction is the type on which we can execute conditional instructions. At present, simple types and their equivalents representing objects can be used, i.e., the "witch`:

- int, Integer
- Byte, Byte
- Char, Char
- short, short
- String
- any enum

A block diagram is shown below:



Note also that the keyword `break` in some cases is specifically omitted and the section `default` is not mandatory, but should be implemented in most cases, e.g:

```
switch (month) { // month is of enum java.time.Month
  case APRIL: // no break instructions
  JUNE case:
    System.out.println("Month has 30 days");
    break;
  JANUARY case: // no instructions break
  MARCH case:
    System.out.println("Month has 31 days");
    break;
   FEBRUARY case:
     System.out.println("Month has always less than 30 days");
}
```

Java 14 also introduces a new way of saving `switch` instructions. The character `:` for the condition `case` can be replaced by `->`. The use of `->` instead of `:` also relieves you of the need to write the `break` instruction because you always execute the code only at a particular block of the `case`. If you want to execute the same code for multiple values, we should separate them by commas. The examples below show a new alternative syntax for the `switch` expression

them by commas. The examples below show a new, alternative syntax for the `switch` expression.

"java switch (number) { case 1 -> System.out.println("I'm one"); case 2 -> System.out.println("I'm two"); case 3 -> System.out.println("I'm three"); default -> System.out.println("I am a number other than 1, 2 and 3"); }

```
``java
switch (number) {
case 1, 3, 5, 7, 9 -> System.out.println("I am the odd positive number");
case 2, 4, 6, 8 -> System.out.println("I am an even positive number");
default -> System.out.println("I'm not a positive odd number");
}
```

Loops

A loop helps with repeating the same instruction (on each element) a given number of times until a certain condition is met or not (then it goes to infinity).

We can specify a few different types of loops:

- They execute the provided block of code, until certain conditions are met.
- The code is executed a finite number of times (loops).
- Loops that have no end point are called infinite loops.

In Java we specify the following types of loops:

- `for`
- `while`
- `do while`

for

In general, a "for" loop looks like this:

```
for(initial statement; statement end point; statement incrementation) {
    // instruction
}
```

We can therefore distill the "loop" code into three phases:

- **initial statement** – they are executed once, just before the loop is started. They are often used to create the **initial condition** based on which the loop functions.
- **statement end point** – this checks if the loop is still valid or whether it has reached it's end point. If the condition is still true the code within the loop will execute. If it is not, the code will go outside the loop.
- **statement incrementation** – this block of code is executed after **each** loop run (in common terms: the code has finished executing, at least, once). It is often used to modify (increment/decrement) the variable provided in the initial statement.

The simplest use of "for" can be seen below:

```
for (int i = 0; i < 10; i++) {
    System.out.println("Hello World!");
}
```

The use of this loop is fairly easy: You set up the steering variable 'i' at 0 initially **(i=0)** . This helps to print *"Hello World!"* once. With the statement incrementation of **i++** you add 1 to the initial value of i. Until the statement end point of **(i<10)** is reached the code block of printing "Hello World" will repeat 10 times.

The 'for' loop is also great for working with collections:

```
String[] array = {"We", "have", "a cat"};
for (String element: array) {
    System.out.println(element + " ");
}
```

This is referred to in arrays in further chapters.

while

The 'while' loop is used a bit differently. Imagine a situation, when you don't know how many times you wish to repeat a loop

but you know which condition to use so that the code block is executed. The general schema is as follows:

```
while (condition) {
    // instruction
}
```

The 'while' loop can be described in this way: keep repeating the instruction provided within the code block until the condition within the loop is not not fulfilled. When the condition is met the loop and it's code is repeated. If the condition is not met the loop might not even initialize (in other words: it might not even start).

A simple example is provided below (the behavior will be the same as in the 'for' example above):

```
int i = 0;
while (i < 10) {
    System.out.println("Hello World!");
    ++i;
}
```

As a result of running the loop the "Hello World!" string will be displayed 10 times.

do while

There is a linguistic difference in using this loop as opposed to 'while'. With "do while" all the instructions in the code block will run **at least once** – as the name implies. The 'while' condition is checked at the end of the loop (after the whole code block has been executed once).

The general schema can be showcased as seen below:

```
do {
    // instructions
}
while (condition);
```

A similar example to the previous ones would look like this:

```
int i = 0;
do {
    System.out.println("Hello World!");
    ++i;
} while (i < 10);
```

At first glance the text is printed the same number of times as in previous examples. There is a slight difference though: if the variable 'i' had an initial value of 12 and not 0 the text would be printed at least once. In a 'while' loop it would not show at all.

break and continue

There are two keywords strongly linked to loops. They are 'break' and 'continue'.

The word 'break' helps end the loop which is ongoing. The next iteration of that loop will not start:

```
for (int i = 0; i < 10; i++) {
    System.out.println("Hello World!");
    if (i == 1) {
        break;
    }
}
```

In the above example the text **Hello World!** will be shown two times only. With the use of break at (i==1) the loop will print the text twice – when i is equal to 0 and 1. Then it will "break" and go outside the loop.

On the other hand by using the 'continue' keyword the condition will enable the loop to go on:

```
for (int i = 0; i < 10; i++) {
    if (i == 8) {
        continue;
    }
    System.out.println("Hello World!");
}
```

By using 'continue' when the variable i reaches the number 8 the code will skip over printing "Hello World!" and as result the text will be printed only 9 times. The use of 'continue' has no influence over the final condition in the 'for' loop (it will always be executed).

The String class

The class called `String` is a built into Java. It is used to represent character strings. The text in the form of `String` can be declared in two ways:

- Using a literal, e.g.:

```
String myText = "This is a simple text."
```

- Using the keyword `new`:

```
String myText = new String("This is a simple text.")
```

The difference between the two approaches is that when creating a string using a literal, we allow Java to manage the created object, and when using the constructor we have control over the created instance ourselves. In practice, we should use the literal approach most often.

Immutability of String objects

The `immutability` of String objects is that once created, an instance of the `String` object will no longer change its value. The `String` class does not have any method (e.g., setter) that can modify its value, so this construction is possible:

```
String text = "This is. "
text += "my text";
```

In the above example, we have created a new object with the value `This is my text` and set the reference variable `text` to it.

What gives us the immutability of `String` objects?

- security – it is easier to protect against code modifications from outside, and the `String` class is secure in multi-threaded applications.
- performance – the `String` class is often used as a key in data structures such as 'HashMap'. This means that once the value of `hashCode` is calculated it will not be recalculated, which significantly affects performance in an application.
- String pool – multiple references can point to the same object thanks to the pool – a Java optimization mechanism that stores only one unique String value. This optimization allows for a significant reduction of the amount of text that needs to be stored.

The use of a String from the pool can be done by calling the `intern` method on an instance of the `String` type object, e.g:

```
String text1 = "This is a test";
String text2 = "This is a test";

String val1 = text1.intern(); // the value 'This is a test' is taken
String val2 = text2.intern(); // the value 'This is a test' is taken

System.out.print(val1.equals(val2))     // the true value will be returned
```

In the example above, the variables `text1` and `text2` will point to the same value in the String pool. The `equals` method will return the value of `true` because the values of both variables are the same – this method will be discussed later.

Text Concatenation

`The concatenation of` Strings is nothing more than the process of connecting them together. This can be easily done using the + operator, as in the following example:

```
String text1 = "My name is ";
String text2 = "John Doe";
String finalText = text1 + text2;
```

The linking operation is done from left to right, so the value of the `finalText` variable will be: `My name is John Doe`. The concatenation can also be done by calling the `concat` method on an instance of a String type object, e.g. the `finalText`

concatenation can also be done by calling the `concat` method on an instance of a String type object, e.g., the `finalText`:

```
String text1 = "My name is ";
String text2 = "John Doe";
String finalText = text1.concat(text2);
```

Comparing two or more texts

To check if the two texts are identical, use the `equals` method. Example:

```
String text1 = "Text to compare";
String text2 = "Text to compare";
System.out.print(text1.equals(text2)); // the value true will be displayed because both
strings are identical
```

Let's not compare chains using the operator ==. This is illustrated by the example below:

```
String text1 = "Text to compare";
String text2 = new String("Text to compare");
System.out.print(text1 == text2); // false !
```

The operator === can only check if both strings are stored in the same memory location, and in the above example they are not (strings are not shared).

String class methods

The `String` class has several useful methods. Below are the most commonly used ones:

length()

Returns the chain length, e.g:

```
System.out.print("This is test value".length()); // a value of 18 is displayed
```

As you can see, white characters (in this case spaces) are also counted as chain characters.

toUpperCase() and toLowerCase()

Both of these methods change the string into a character string consisting of capital letters themselves and, in the second case, lower case letters. Example:

```
String testValue = "This is test value";
System.out.print(testValue.toUpperCase()); // the text 'THIS IS TEST VALUE' will be
displayed
System.out.print(testValue.toLowerCase()); // the text 'this is test value' is displayed
```

indexOf()

The `indexOf()` method returns the position ** of the first** occurrence of the specified text in a string of characters (the counting is performed together with white characters, e.g. a space) – the position is counted from the value 0 (the first item). An example follows:

```
String testValue = "This is test value";
System.out.print(testValue.indexOf("is")); // value 2 will be displayed
```

replaceAll()

The `replaceAll()` method allows to replace all occurrences of a given substring with another, e.g:

```
String text = "Hahahah! Funny joke!"
System.out.print(text.replaceAll("a", "o")); // the string 'Hohohohoh! Funny joke!'
```

substring()

The `substring()` method allows to retrieve a substring, based on the initial (or final) index, e.g:

```
String text = "Example test";
text.substring(3); // will be returned String "mple test"
text.substring(2, 6); // will be returned String ampl
```

contains()

The `contains()` method returns information whether a given String contains a specific string, e.g:

```
String text = "Example test";
System.out.println(text.contains("xam")); // true
System.out.println(text.contains("Test")); // false
```

trim(), isBlank() and isEmpty()

The `trim()` method returns a String, which is the result of removing all white characters (such as spaces or tabs) from the beginning and end of a character string. The `isEmpty()` method tells us if the String is empty. A similar information is obtained by calling the `isBlank()` method, but unlike the `isEmpty()` method, all white-space characters at the beginning and end of the string will be removed before this information is obtained, e.g., the `isEmpty()` method.

```
System.out.println("\tsda"); // \t is a tab character, it will display " sda"
System.out.println("\tsda".trim()); // "sda"
System.out.println(" ".isEmpty()); // false
System.out.println(" \"T". isBlank(); // true
```

Standard input/output

Scanner – basic entrance

The `Scanner` class is ideal for retrieving user input as well as the basic reading of the file. To read the data, an instance of an object of the type `Scanner` must be created and the input stream `System.in` must be specified as parameter. In the following example, we try to read the text line entered by the user:

```
Scanner scan = new Scanner(System.in);
String textLine = scan.nextLine(); // we loaded the line of text entered by the user here
                                   // and we saved it in the textLine variable
```

The class `Scanner` is included in the `java.util` package, so it should be openly imported in our application. As an argument to the constructor, we have given the `System.in`, i.e., the standard input stream (console) – another stream type can also be given here. The `nextLine()` method stops the program from running until the user enters something in the console and confirms it with the enter key. It also returns the entire line typed by the user as a string of `String` type characters.

In addition to the `nextLine()` method, the `Scanner` class offers a number of additional methods that allow you to read data of other types, such as

- `nextInt()` – reads another integer
- `nextDouble()` – reads another floating-point number
- `nextBoolean()` – reads another logical value

Standard output

Java has several basic methods for displaying data. The class `System` is used for this. We have a few basic variants:

- `print()` – prints on the screen a given string or a numeric variable that it converts previously to the `String` type, e.g., `System`:

```
int myNumber = 125;
System.out.print("This is a simple text."); // the following will be displayed: 'This is
a simple text.
System.out.print(myNumber); // The number 125 will be displayed.
```

- `println()` – works the same way as `print()` by adding a new line character at the end
- `printf()` – a method that can format the data in addition to writing it. Special conversion operators are used for this purpose. Here is a list of basic ones:

- d – decimal integer
- e – floating point number in exponential notation
- f – floating-point number
- x – integer in hexadecimal system
- o – integer in the octal (octal) number system
- s – character string
- c – the sign
- b – logical value

Examples of how the `printf()` method works:

```
System.out.printf("100.0 / 3.0 = %5.2f", 100.0 / 3.0); // the result will be a floating
point number consisting of 5 characters and 2 digits after the decimal point
System.out.println();
System.out.printf("100 / 3 = %4d", 100 / 3); // the result will be an integer occupying 4
characters – the division will be rounded
```

Regular expressions

Regular expressions (also known as **regex**) are patterns that enable us to verify whether a given string of text (for instance some user input) abide a pre-selected format (like time and date). With regex we can confirm whether the user input data follows the correct formula. For instance this regex helps check whether the user has provided the correct name:

```
"[A-Z][a-z]+"
```

Regular expressions are structured out of an atom sequence. The basic atom is a single character,number or a special character. We can group such 'literals' with brackets. We can also use 'quantifiers' which count the number of atom occurrences and an alternative character. The simplest regular expression can take this form:

```
abcde
```

Quantifiers

Quantifiers specify the number of occurrences to match against in a given character string. Let us see an example below:

```
a+bcde
```

This instance is quite more interesting and complex. This expression will return true for the following strings: "abcde", "aabcde", "aaabcde". By using the '+' quantifier it will match one or more occurrence of the pattern that is present before it. Below you can find the table with popular regex quantifiers:

Quantifier	Meaning	Example	Regex matches
'*'	zero or more occurrences	a*b	ab,b,aab,aaab,aaab (and similar)
'+'	one or more occurrences	a+b	ab,aab,aaaab,aaaab (and similar)
'?'	zero or one occurrence	a?b	ab,b
'{n,m}'	minimum of n and maximum of m occurrences	a{1,4}b	ab,aab,aaab,aaaab
'{n,}'	at least n occurrences	a{3,}b	aab,aaab,aaaab,aaaaab (and similar)
'{,n}'	a maximum of n occurrences	a{,3}b	b,ab,aab,aaab
'{n}'	exactly n occurrences	a{3}b	aaab

Scope and groups

When talking about scope in regular expressions we usually wish to say: "here you will find one of those characters". Such a scope of characters is also understood as an atom. We define it in square brackets. We can do this in two ways: by naming all possible characters (one next to another, no commas used) or introduce a group. We can also combine them. We identify a group by specifying the first and final item separated by a hyphen. This refers to numbers (like 1–3 ; 0–9; 1–5) or characters (a-z ; A-Z).

Below you will find a table of regex groups and scope usage:

Expression	Description
[abcde]	one of the characters: a,b,c,d,e
[a-zA--Z]	Any small or upper case letter from the range a-z (or A-Z)
[a-c3-5]	Any letter from a to c or number from 3 to 5
[a-c14-7]	Any letter from a to c, number 1, numbers from 4 to 7
[abc\[\]]	Any letter from a,b to c or a square bracket (more on that below)
[.]	Any letter (read below)

Implementing regular expressions in Java

In Java we accomplish most tasks related to regex by using the 'Pattern' and 'Matcher' classes.

Pattern

Pattern

The 'Pattern' class represents a compiled regular expression. In other words it's an expression calculated (or 'manufactured') by the computer which makes it's execution more efficient. We get an Object with representation of our expression by using the static method of 'compile(regexAsString)':

```
Pattern pattern = Pattern.compile("a+bcd");
```

Matcher

We get an instance of the 'Pattern' class which includes a 'matcher()' method which in the end returns an instance of the 'Matcher' class

```
Matcher matcher = pattern.matcher("aaaaabcd aaaaaaabbcd");
```

The 'Matcher' object also has a method called 'matches()' which informs us whether the string of characters used to create an instance of the 'Matcher' class fits into our regular expression:

```
matcher.matches();  // returns true or false
```

The 'Matcher' class also has a 'find()' method which returns 'true' if there is something that matches to the regex expression:

```
matcher.find();     // this will return true to our examples provided above.
```

Arrays

A Java array is a special type of object variable that serves as a container to hold structured data of one type. We can refer to its individual elements with **index**.

For example, if we wanted to store 1000 names in our application, we wouldn't have to declare 1000 variables of the `String` type, but only declare one 1000-element array variable that stores strings of the `String` type.

To declare an array, you need to specify the type of data stored in the array and specify its size. The size of the array should be constant, so when declaring an array, the size must be specified or the elements must be explicitly declared so that the compiler can calculate the length of the array itself.

There are two basic types of arrays:

- one-dimensional
- multidimensional

We will discuss the differences between them below.

One-dimensional arrays

The schema of the array type declaration is as follows:

```
type[] table name = new type[number of elements];
```

How should we read this declaration? We have created an array variable (or more precisely a one-dimensional array) named `table_name and size` number of elements`, which will store elements of` type`. Below is an example:

```
String[] myArray = new String[10];
```

In the example above, we have declared a 10-element array that will hold elements of the `String` type.

It is also possible to declare an array without specifying its size, but the individual elements of the array should be openly initiated inside the curly brackets separating the individual elements with a comma sign, e.g., "`String`:

```
String[] array = new String[] {"Hello", "World", "!" };
```

Here are some examples of table declarations of different types:

```
int[] myNumbers = new int[5];
```

```
int[] myNumbers2 = new int[]{1, 2, 3};
String[] myStrings = new String[2];
long[] myLongs = new long[3];
```

Table initialization and default values

When declaring an array and not specifying what data to be completed, it will be completed with default values for the selected type, e.g. for numbers it will be 0, and for reference variables: `null`. This is well illustrated by the example:

```
String[] forenames = new String[4]; // we have declared an empty 4-element array storing
character strings
```

The indexes of tables

Values stored in arrays can be written and read using the indexes under which they are located. Indices are natural numbers. Index numbers begin with **`0`**, i.e., the first element will be located in the array under index number **`0`**. The last element will be located under an index with a value of `size_table -1`. Remember not to exceed the array index numbering range, because the compiler will report an exception `java.lang.ArrayIndexOutOfBoundsException`.

Let's use the initialized array `name` from the previous example and modify the corresponding fields:

```
String[] names = new String[4];
names[0] = "John";
forenames[3] = "romance";

System.out.println("Element number 1: " + forenames[0]); // Element number 1: Jan
System.out.println("Item number 2: " + names[1]); // Item number 2: null
System.out.println("Item number 3: " + names[2]); // Item number 3: null
System.out.println("Element number 4: " + names[3]); // Element number 4: rom
```

Graphical status of the array after the operations of modification of its relevant cells is now as follows:

Going over the array

To print all values of the array `name` from the previous example, we called the printing method `System.out.println()` four times, each time changing the cell index. We can say that this is not quite a "nice" and automatic method. We are helped by the `for` loop instruction already known to us. How do we use this method for arrays? Quite easy: just go through all the array elements in the `for` loop (make the so-called **iteration**) using the control variable that will refer to the corresponding array index. An example is shown below:

```
int tabLength = 4;
String[] names = new String[tabLength];
forenames[0] = "John";
forenames[3] = "roman";

for (int i = 0; i < tabLength; i++) {
    System.out.println("Element number " + (and + 1) + ": " + names[i]);
}
```

The effect is identical to the previous example, and the code is more universal and automated – we do not repeat each item's print call code separately – this is provided by the `for` loop. It should also be noted that, as previously written, we iterate from an index numbered **`0`** to an index `tabLength - 1`, where `tabLength` is the size of an array.

Downloading an array size

It is possible to download the array length. The `length` attribute can be used for this, as the array is an object type and therefore has built-in methods and attributes. Here is an example:

```
String[] myArray = new String[10];
```

```
System.out.println(myArray.length); // 10
```

Multidimensional arrays

As we mentioned earlier, arrays can also store any object, not just simple types. There is therefore nothing to prevent an array from storing other arrays as well, as the array is also an object. Therefore, you can create two-, three- or multidimensional arrays. A two-dimensional array is nothing more than a structure containing rows and columns that store data.

The basic declaration and initialization of a two-dimensional array looks like this:

```
type [][][] name of the table; // table declaration
table name = new type[number of rows][number of columns]; // table initialization
type[][][] table name = new type[number of rows][number of columns]; // declaration and
initialisation in one
```

The number of columns and rows need not be identical, so you can create any rectangular arrays. An example of creating and initializing an array that stores character strings:

```
String[][] myArray = new String[2][];
myArray[0] = new String[]{"Alice", "has", "cat"}; // create the first line, i.e. with
index number 0
myArray[1] = new String[]{"Cat", "has", "Alice"}; // create a second line, i.e. with
index number 1
```

As you can see in the example above, you can initialize the rows of a two-dimensional array with one-dimensional tables. And how to refer to individual elements of a two-dimensional array? We present this below using a previously created array called `myArray`:

```
System.out.println(myArray[0][0]); // Alice
System.out.println(myArray[0][2]); // cat
System.out.println(myArray[1][1]); // has
System.out.println(myArray[1][3]); // Error! The java.lang.ArrayIndexOutOfBoundsException
will be thrown
```

As you can clearly see in the example, the first coordinate represents a row of the array, the second – a column, the standard numbering starts with index number 0. Remember not to exceed the range of declared array sizes!

Going through all the elements of a two-dimensional array is analogous to iterating after a one-dimensional one. The only difference is that we additionally add a second nested `for` loop: the first loop will iterate after rows, while the second loop will iterate after columns. Here is an example:

```
for (int i = 0; i < myArray.length; i++) {
    for (int j = 0; j < myArray[i].length; j++) {
        System.out.print(myArray[i][j] + " ");
    }
    System.out.println();
}
```

Let's clarify a few things:

- The expression `myArray.length` will return the number of lines of the table
- The expression `myArray[i].length` will return the number of columns in a row
- The table cell is referenced by `myArray[i][j]`, where `i` and `j` are the current row and column indices.

Methods

The method is nothing more than a set of instructions. We group the code in this way for several reasons:

- If a piece of code is to be executed in many places, it is definitely better to create a method and run (call) it, than to copy the same piece of code repeatedly. This is important, because in case of an error, you need to correct it in one place, not in several.
- Programs are large, without proper division, mastering the whole structure is very time consuming. A sensible division into smaller parts allows you to understand the code faster.

Definition

The syntax for defining the method is as follows:

```
<access modifiers> <type returned> <name of method>(<optional list of arguments>) {
    <body of the method>
}
```

Let's create a simple example of a method definition:

```
void printName(String name) {
    System.out.println("My name is: " + name);
}
```

This method accepts the `name` argument of the `String` type and displays its value easily.

In the above example, no access modifiers are used. These will be discussed in detail in the other section.

The part of the method declaration where we give the access modifiers, the returned type, the name of the method and an optional list of arguments is called the **name of the method**.

The method signature from the previous example is presented below:

```
void printName(String name)
```

Arguments

Methods may or may not accept any number of arguments. Just because it is possible does not mean that it should be done. In most cases, methods with a large number of arguments are a sign of poor quality code (aka. code smell).

If a method contains several arguments, they are separated by commas. Below is a method that calculates the difference between two integers:

```
int diff(int arg1, int arg2) {
    return arg1 – arg2;
}
```

The returned value

Methods can return values. The value returned by the method is preceded by the keyword `return`. In the method signature we additionally declare the returned type. This is illustrated by a simple example:

```
int returnedNumberExample() {
    return 5;
}
```

It is also possible that the method will not return any value. In this case, the declared type that our method returns is a special type of `void`, e.g., the 'background' type:

```
void print() {
    System.out.println("Hello World!");
}
```

The value returned can be any simple variable or an [object] type (class_objects.md). We can also prematurely interrupt a method that does not return any values and exit it, using the keyword `return` without providing a value, e.g., [object_classes.md]:

```
void returnExample(int number) {
    if (number % 2 == 0) {
        return;
    }
    System.out.println(number);
}
```

The body of the method

The body of the method is the entire code contained between the curly brackets '{}'. For example, the body of the `diff` method defined in one of the above examples is:

```
return arg1 – arg2;
```

```
return arg1 - arg2;
```

The body of the method above consists only of the value returned by the method, which was calculated on the basis of the arguments provided.

```
System.out.println("Hello World!");
```

As in the example above, the body of the method can only be a simple call of the printing method and the method does not return anything. Of course, method bodies can be more complicated and consist of many lines of code.

Calling a method

A clear distinction must be made between the definition of a method and its call. The definition, as we described earlier, is a declaration of the method together with the name of the method, the type returned, the arguments and its body. A method call is a reference to the method name with the appropriate parameters (or without parameters in the case of a non-argument method). An example of a method definition and its call is presented below:

```
// declaration of the method
int multiple(int arg1, int arg2, arg3) {
    return arg1 * arg2 * arg3;
}
```

```
// calling the method
int multipleValue = multiple(23, 2, 5); // 230
```

The method named `multiple` as defined in the input has three arguments with the following names: `arg1 arg2 arg3`, and as a result, returns the result of a multiplication of these arguments. When calling the method, refer to its name (in our case, `multiple`) with the corresponding values of the input parameters (remember the correct types, as in the definition).

Since our method returns a numeric value of `int` type, it is possible to assign the returned result to a variable of `int` type (according to the method definition).

Method naming

In Java it has been assumed that the names of the methods are written in the so-called **camelCase** standard, i.e. the following words are written together, starting each subsequent one with a capital letter (except the first). Examples are provided below:

- sumElements
- startProcessingData

The same principle applies to variable name declarations.

Classes and Objects

Classes

Classes in object-oriented programming are used to describe surrounding objects, events, activities, states and relations between the described objects. The classes defined in our applications can then be used to create variables of these types. Such types defined by classes are called **complex** or **reference types**.

Classes have two basic components:

- **fields** – it's nothing else than the variables we've learned before. It is also a feature that describes the object of the class.
- **methods** – by method we mean the operation that our class makes available to the outside world and it's use.

To sum up: a class is a schema which consists of fields and methods defined in it.

Suppose that we want to describe the movie as a class. So, what features (fields) can a class consist of to describe our movie? We all know this: the title, the year of production, the description, the actors, etc. All these features are nothing more than the fields in the film class. There are still activities (methods), e.g: "play the movie", etc. Below is an example of the definition of the `Movie` class:

```
public class Movie { // (1)
    private String title; // (2)
    private String description; // (3)
    private int productionYear; // (4)
```

```
    public void play() { // (5)
        // the body of the method
    }
}
```

In the example above we have defined a public class called `Movie` – this is indicated by the *(1)* line declaration: `public class Movie`. We have used the keyword `class` and the `public` modifier that defines the class's visibility range (this will be discussed in the following chapters). The lines *(2)* – *(4)* are field declarations representing the film title, description and year of production. They are also preceded by `private` modifiers, which define the range of these fields. The line *(5)* is a declaration of the `void play()` method, which will allow us to start playing our film. Thus, we see that our class includes fields that represent features and methods, i.e. operations that our class provides.

Let's look at a slightly more complicated example of a class:

```
public class Car {
    private String color;
    private int maxSpeed;
    private String brand;

    public void setColor(String color) {
        this.color = color;
    }

    public void setMaxSpeed(int maxSpeed) {
        this.maxSpeed = maxSpeed;
    }

    public void setBrand(String brand) {
        this.brand = brand;
    }

    public void printCarParameters() {
        System.out.println(String.format("Car color is: %s, max speed is: %d, car brand
is: %s", color, maxSpeed, brand));
    }
}
```

In addition to field declarations, the class also has several methods: `setColor`, `setMaxSpeed` and `setBrand` – these are the so-called "field declarations". These are the so-called `setters`, i.e. single-argument methods that set the values of our fields (this will be mentioned in the following paragraphs).

In general, the class definition can be written as follows:

```
[access modifier] class nameClasses {
    // class field definitions

    // method definitions
}
```

Objects

What are objects? You can create copies (**instances**) of classes from the template (classes). These instances will contain fields defined within the class and you can call on them the methods that were defined in the class. Instances of the class that were created are called **objects**. Therefore, it is important to remember what is the main difference between the object and the class: if we use the example of the `Car` class, the class is nothing more than a set of features representing a car and actions that can be performed (methods). The object is a specific copy of this class, for example, a Mercedes car, which is white and reaches a maximum speed of 250km/h. We then move on to the process of creating an instance of a given class.

Creating class instances

Let's try to use the `Car` class defined earlier and create two objects of this class:

```
Car car1 = new Car();
Car car2 = new Car();
```

In the above example, we have created two instances of the `Car` class with names: `car1` and `car2`. To create objects, we have used the keyword `new`, which invokes the so-called "car1andcar2. **constructor**, which we will talk about later. This example is very simple and just creating a class instance without setting the value of fields does not give us much. For this purpose, we can use methods to set the values of these fields, the so called **setters**, which we defined in the class `Car`, so we can develop our example a little more:

```
Car car1 = new Car(); // (1)
car1.setBrand("Mercedes"); // (2)
car1.setColor("white"); // (3)
car1.setMaxSpeed(250); // (4)
car1.printCarParameters(); // (5)

Car2 = new Car();
car2.setBrand("Fiat");
car2.setColor("red");
car2.setMaxSpeed(210);
car2.printCarParameters();
```

In the example above we created two instances with names: `car1` and `car2` representing two copies of the `Car` class. The first of these is a Mercedes car. In this instance, we have set the values of the relevant fields giving them specific characteristics:

- *(2)* – in this line, we have set the make (field `brand`) to the value of `Mercedes` calling the method `setBrand`.
- *(3)* – we have similarly set the color (`color` field) to `white`.
- *(4)* – here we have defined the maximum speed of the car as the numerical value of `250`
- *(5)* – this line is nothing more than a call to the printing method of the standard output (console) of all data (features) of the `car1` instance.

As you can see in the above example, to call a method on an object, you have to refer to the name of this method on the object reference variables using the dotted notation, e.g. `car1.setBrand("Mercedes")`. In the same way, we refer to the instance fields to retrieve their value, e.g. `car1.brand`.

The following lines are a similar creation of an instance of `car2`, which represents a Fiat brand of a car. In the above example, each assignment of a value to a field was associated with calling the appropriate method (setter), which was used only to assign values. We can initialize our fields with the appropriate values using one instruction, namely **constructors**.

Access Modifiers

Access modifiers are used to determine the scope and visibility of methods, class fields, and are also used to set the visibility of the classes themselves. They allow us to determine how the classes will be used and who will be able to use the fields and methods defined inside them and in what situation. There are four basic modifiers:

- `public`
- `protected`
- `default`
- `private`

Let's discuss two of them first: public and private. When we define a field, method, or class with a `public` modifier, we give them access from anywhere in our application – anyone can refer to or call such a method. We then say that our field, method or class is **public**.

It may also happen that we don't want to give access to some fields or methods outside the definition of our class – they are supposed to be **private** for our class. We can manage such fields and methods (i.e. download and change field values or call methods) only from the level of that defined class. In this case, we use the access modifier `private` – we say that our fields and methods are then **private**.

In summary: access modifiers regulate access to fields and methods of classes when we refer to them from the level of other classes. Let's illustrate this with an example. Let's define the class describing the book:

```
public class book {
    public String title;
    public String author;
    private int numberOfPages;

    public void setNumberOfPages(int numberOfPages) {
        if (isNumberOfPagesCorrect(numberOfPages)) {
            this.numberOfPages = numberOfPages;
        } else {
            System.out.println("The provided number of pages is incorrect: " +
numberOfPages);
        }
    }

    private boolean isNumberOfPagesCorrect(int numberOfPages) {
        return numberOfPages > 0;
    }
}
```

We defined two public fields using the `public` modifier: `title` and `author` and one private field `numberOfPages`. We have also defined two methods: the public method `setNumberOfPages`, which sets the value of the private field `numberOfPages` calling the private method `isNumberOfPagesIsCorrect`, which checks if the new value is greater

field `numberOfPages` calling the private method `isNumberOfPagesCorrect`, which checks if the new value is greater than zero. What does this give us? First, it makes no sense for the `isNumberOfPagesCorrect` method to be public, since it is only invoked inside the `Book` class. Secondly, we wanted the method setting the value of the `numberOfPages` field to be available outside the defined class, so we made it public.

Now let's try to see how method calling and referring to fields outside the `Book` class works. So we will define a BookTest class that will test this behavior:

```
public class BookTest {
    public static void main(String[] args) {
        Book testBook = new Book();
        testBook.author = "Charles Dickens";
        testBook.title = "A Chrismas Carol";
        testBook.setNumberOfPages(250);

        System.out.println("Book title: " + testBook.title); // (1)
        System.out.println("Book author: " + testBook.author); // (2)

        System.out.println("Number of pages: " + testBook.numberOfPages); // Compilation
error!
    }
}
```

In the `BookTest` test class, we created the `testBookTest` object, then referred to the public fields: `author` and `title` by dotted notation and set their values. We also checked the range of the `setNumberOfPages` method, which sets the value of the private `numberOfPages` field.

The next part of the exercise is an attempt to write down field values. Unfortunately, an attempt to refer to a private `numberOfPages` field outside of the `Book` class definition will result in the following compilation error:

```
Error:(14, 55) java: numberOfPages has private access in pl.sdacademy.Book
```

The *(1)* and *(2)* lines correctly refer to the public `title` and `author` fields.

To sum up, when should we use private modifiers, and when public? We should know that private modifiers hide from the user the inside, that is, the implementation of the class, that is, the way it does what it does. We should also apply the following few rules:

- All class fields should be private if possible – the class should usually provide public methods for modifying and retrieving the values of the class fields (so-called **getters** and **setters**).
- All methods that are used internally by the class should be private.
- Only methods to be used by users of the class should be public.

We will talk about the default modifiers once `protected` in the chapter on packages.

Getters and setters

The concept of **getter** and **setter** is closely related to the concept of **encapsulation**. **Encapsulation** is a way of defining classes, where from the outside world (i.e. other classes and the rest of the application) we hide the internal implementation of classes. Instead, we define a set of methods to be used by users. To ensure this, the fields of our classes (as in the previous chapter we mentioned) should always be **private**, so we need to define a set of methods to modify and retrieve the values of the class fields. We call these methods **accessors**. We distinguish two groups of accordions because of the tasks they perform:

For the definition of the accessories we adopt a conventional naming convention: the getter methods take the prefix **get** in the name, and the setters** in the setter: **set**. The second part of the name should be the name of the field being set or downloaded. For example, if we define the accessors for a field called `author`, we could generate these method names:

- `getAuthor` – for the getter
- `setAuthor` – for setter

From now on, we don't define public class fields, but only the appropriate accessories, of course if we need them. So let's try to rebuild our `Book` class in such a way that it doesn't expose public fields to the outside world, but only create public accessories:

```
public class book {
    private String title;
    private String author;

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
```

```
    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }
}
```

In the example above, we see that the `title` and `author` fields have become private – you cannot easily refer to them by their name outside the classroom. We have defined the accessories for this purpose:

- `getTitle()`
- `setTitle()`
- `getAuthor()`
- `setAuthor()`

While the definitions of the getters are not complicated, the setters are a bit more complex because of the `this` construction. **this** indicates the current object, i.e. is the instance to which the accessor method was called. We use the `this` connotation so that the parameter of the method with the same name as the one given to the field that is set will not obscure each other.

The following example shows calling an accessor outside the class definition:

```
public class BookTest {
    public static void main(String[] args) {
        Book testBook = new Book();
        testBook.setAuthor("Charles Dickens");
        testBook.setTitle("A Chrismas Carol");

        System.out.println("Book title: " + testBook.getTitle());
        System.out.println("Book title: " + testBook.getAuthor());
    }
}
```

Note that attempts to refer to private fields using: `testBook.title` and `testBook.author`, will end up with compilation errors this time. The result of the above program is as follows:

```
Book title: A Chrismas Carol
Author: Charles Dickens
```

Packages

Given that many developers can create and use classes with the same names, sooner or later we will come across a situation where we have two classes with the same names in our application. We need a mechanism to do manage this. In Java we can specify in which package (**package**) our class is located. The package becomes an integral name of our class, so we minimize the potential problem of repeating the same class name.

Defining

We declare packages using the keyword `package`, for example

```
package pl.sdacademy.example;
```

The packages reflect the name of the reversed Internet domain. Let us now assume that we would like to have two classes with the same name `Book`. This is of course possible, but you should put them in two separate packages, e.g. `Book`:

```
package pl.sdacademy.myfirstbookexample;

public class Book {
    // definitions
}
```

```
package pl.sdacademy.mysecondbookexample;

public class Book {
    // definitions
}
```

However, when creating packages and classes in these packages, we must remember a few rules:

- The class belonging to a package (keyword `package`) must be at the beginning of the file – the only thing that can precede the use of `package` is the comments,
- It is best that package names contain only letters and numbers. In addition, the package names are separated by dots and do not use the *camelCase* notation – we always use lower case letters,
- The name of the package should correspond to the directory structure on the disk, so for example, when creating a package called `en.sdacademy.myfirstbookexample`, we should have this directory structure on disk:

```
en
|
--sdacademy
|
--myfirstbookexample
```

Importing classes

To use the classes defined in other packages, we must import them using the keyword `import`. It is also possible to import all classes from a given package – for this we use the `*` character instead of the name of the imported class. Suppose we have a Book class defined in the `en.sdacademy.bookexample` package, and we would like to use the `Book Test` in the `BookTest` class definition in the `en.sdacademy` package. We can see that both classes are in different packages, so it was necessary to import the `Book` class in the `BookTest` class:

```
package pl.sdacademy;

import pl.sdacademy.bookexample.Book;

public class BookTest {
    public static void main(String[] args) {
        Book testBook = new Book();
        // other instructions
    }
}
```

How about importing two other classes with the same name into one class?

There is, of course, such a possibility. Let's assume that we have two definitions of classes named `Book`: in the package `en.sdacademy.bookexample.Book` and `en.sdacademy.secondbookexample.Book`. In this case one of them should be openly imported and the other one should be referred to in the code after the full package name. Example below:

```
package pl.sdacademy;

import pl.sdacademy.bookexample.Book;

public class BookTest {
    public static void main(String[] args) {
        Book testBook = new Book(); // (1)
        pl.sdacademy.secondbookexample.Book secondTestBook = new
pl.sdacademy.secondbookexample.Book(); // (2)
    }
}
```

The *(1)* line refers to the `Book` class defined in the `en.sdacademy.bookexample` package, while the *(2)* line refers to the `Book` class giving the full package path: `en.sdacademy.secondbookexample.Book`.

Static import

Sometimes in our applications it is necessary to use the same static (or fixed) method defined in class from another package many times. Unfortunately, we then have to refer to this method by the name of the class in which it is defined, e.g. "static":

```
System.out.println(Math.PI);
System.out.println(Math.round(Math.PI));
```

In order to make it easier for us to use such constants and static methods, we are helped by the so-called "static" methods. **static import**. All we need to do is to import our methods/constants together with the keyword `static`, e.g. **static:

```
import static java.lang.Math.PI;
```

```
public class StaticImportExample {
    public static void main(String[] args) {
        System.out.println(PI);
        System.out.println(Math.round(PI));
    }
}
```

In the example above, we made a static import of the fixed `PI`: `import static java.lang.Math.PI`. This allows us to refer to our constant now only by its name `PI`, and not by the name together with the class name.

Available by default

In the previous chapter, we learned several different access modifiers, namely: `private`, default, `protected` and `public`. We discussed two of these, `private` and `public`.

The default access is also called **package-private. It is unique in that if we don't use any access modifier, our field, method, or class will just get **default** access. It is almost as restrictive as the `private` modifier, but it also allows access to classes that are in the same package, i.e. have the same package defined by the keyword `package`.

An example of a class definition with a method with a default modifier:

```
package pl.sdacademy.myfirstbookexample;

public class Book {
    private String title;
    private String author;

    String getAuthorAndTitle() {
        return title + " " + author;
    }
}
```

We see in the above example that the `getAuthorAndTitle()` method has a defined default range, so it will only be available in classes from the `en.sdacademy.myfirstbookexample` package.

The protected modifier

The `protected` modifier will be thoroughly discussed on the occasion of the inheritance, but it should now be mentioned that it shares a feature with the default modifier, i.e. it allows access to fields and methods from classes in the same package. In addition, it should also be mentioned that elements of the class are made available to the class itself and its subclasses (the inheritance will be discussed in other chapters).

Constructors

Definition

Until now, we have created our objects using the keyword `new`, then to set the value of a given field in the object, we have defined **setters**, which we called on the instance as usual methods. This is a rather cumbersome way, especially when our class consists of multiple fields. The **constructors** are one solution to this. Java constructor is a special type of method that is used to initialize the state of an object, i.e. it sets the values of object fields. Constructors have two distinguishing features compared to ordinary methods:

- The name of the constructor is identical to the name of the class in which we define it, e.g. inside the `Car` class we define a constructor also called `Car`.
- When defining the constructor, we omit that part of the method signature in which we define the returned type – it is always "empty".

It should also be mentioned that constructors' methods, like normal methods, can take any number of input arguments. Let's check this with the example of the previously discussed class `Car` – let's redefine it using the constructor:

```
public class Car {
    private String color;
    private int maxSpeed;
    private String brand;

    public Car(String color, int maxSpeed, String brand) { // (1)
        this.color = color; // (2)
        this.maxSpeed = maxSpeed; // (3)
        this.brand = brand; // (4)
    }

    public void printCarParameters() {
        System.out.println(String.format("Car color is: %s, max speed is: %d, car brand
is: %s", color, maxSpeed, brand));
```

```
    }
}
```

Compared to the previous definition, this one has the constructor method and there is no definition of the methods **setters**. By using the constructor, in this case they become redundant. Let's move on to the discussion of the constructor method: in the line *(1)* there is a signature declaration of the constructor method (without the returned type), having the same name as the class in which it is defined. This constructor takes three parameters which represent the types of individual fields of the `Car` class. The lines *(2) – (4)* are the assignments of parameter input values to the class fields. In the constructor we again use the keyword `this` which, as before, is a reference to the current instance of the class for which the method was called. The following assignment will set us the value of the `color` field:

```
 this.color = color;
```

The above line of code is nothing more than to assign a value from the `color` parameter given in the constructor to the `color` field of the current instance (reference by `this.color`).

The definition of the constructor we are analyzing is the definition of the so called "color". ** constructor with parameters** – as the name suggests, the method takes the parameters. There is one more version of the constructor – **default constructor**.

Default constructor

In the first definition of the `Car` class, we did not define the constructor, but by creating an instance of the `Car` class, we called the constructor using the instruction:

```
Car car1 = new Car();
```

This is nothing more than creating an instance of the `Car` class using the valueless constructor or otherwise a default one. You can see that the constructor method does not take any parameters. How is it possible that we could use the constructor without defining it in the class? This is natural, because in Java, the non-parameter constructor is the default one – you don't need to define it, because you inherit (we will talk about the inheritance later) it from the `Object` class. The constructor is called the default constructor and is automatically generated for us by the Java compiler in one particular case: if we don't provide the constructor for the class ourselves (no definition of any constructor).

Let's analyze another example:

```
public class Car {
    private String color;
    private String brand;

    public Car(String color, String brand) {
        this.color = color;
        this.brand = brand;
    }
}
```

Now let's try to create a `Car` class instance using a valuless builder:

```
Car car1 = new Car();
```

We will get the following compilation error:

```
Error:(7, 20) java: constructor Car in class pl.sdacademy.Car cannot be applied to given
types;
required: java.lang.String,java.lang.String
found: no arguments
reason: actual and formal argument lists differ in length
```

This is because the default constructor has been replaced by the definition of the constructor with parameters`Car(String color, String brand)`. For the compilation to be successful, a definition of the non-parameter constructor must be explicitly created. The `Car` class will then have a form:

```
private String color;
private String brand;

public Car() {
}

public Car(String color, String brand) {
    this.color = color;
```

```
    this.brand = brand;
}
```

Then a call to `Car car1 = new Car();` will correctly create an instance of the `Car` class.

Overloading the constructors

Each class can have multiple constructors. Because constructors are a special kind of methods, in order to have more than one constructor, each of them must differ in number, type or order of arguments. Let's illustrate this with the example of the `Car` class – let's expand it with more constructors:

```
public class Car {
    private String color;
    private int maxSpeed = 180;
    private String brand = "Fiat";

    public Car() {
        this.color = "white";
        this.maxSpeed = 180;
        this.brand = "Fiat";
    }

    public Car(int maxSpeed, String brand) {
        this();
        this.maxSpeed = maxSpeed;
        this.brand = brand;
    }

    public Car(String color, int maxSpeed, String brand) {
        this(maxSpeed, brand);
        this.color = color;
    }

    public void printCarParameters() {
        System.out.println(String.format("Car color is: %s, max speed is: %d, car brand
is: %s", color, maxSpeed, brand));
    }
}
```

A few sentences explaining the above example: the first constructor (with no arguments) sets the fields with default values. The next constructor (with two arguments) calls the valueless constructor calling the `this()` instruction, additionally it sets the values of the fields `maxSpeed` and `brand` with the values of parameters passed to the method. The last constructor (three-parameter one) calls the two-parameter constructor (the `this(maxSpeed, brand)` instruction) and additionally sets the value of the `color` field with the value passed on to the method.

Now let's try to call overloaded constructors in the example:

```
Car car1 = new Car();
car1.printCarParameters();

Car car2 = new Car(250, "Mercedes");
car2.printCarParameters();

Car car3 = new Car("Red", 320, "Ferrari");
car3.printCarParameters();
```

The result of the action is as follows:

```
Car color is: white, max speed is: 180, car brand is: Fiat
Car color is: white, max speed is: 250, car brand is: Mercedes
Car color is: Red, max speed is: 320, car brand is: Ferrari
```

To sum up, **overloading** of the constructors is the process of defining many of them in one class, bearing in mind that each of them must differ in number, type or order of arguments.

Overloading methods

In addition to overloading the constructors, we can also overload methods in a class. **The overload** of a method happens when you define in one class a method with a name that already exists in it. For such code to compile, each of the overloaded methods must differ in number, type or order of arguments. Below is an example of a class that correctly overloads the `add` method:

```
public class SimpleCalculator {

  public int add(int numA, int numB) {
    return numA + numB;
  }

  public int add(int numA, int numB, int numC) {
    return numA + numB + numC;
  }
}
```

It is worth noting that changing only the access modifier or returned type **not** is a correct overload of the method, e.g.

```
public class SimpleCalculator {

  public int add(int numA, int numB) {
    return numA + numB;
  }

  private long add(int numA, int numB) { // incorrect overload, difference is only in
access modifier and type returned
    return numA + numB;
  }
}
```

Primitives and classes

Java offers several primitives types like `int`, `double` or `float`. In some cases using primitive types is impossible. For example trying to define a list of integers in a following way is **impossible**:

```
List<int> ints = new ArrayList(); // błąd kompilacji
```

For such cases java offers several **wrapper classes** for primitive types:

primitive type	wrapper class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

Thanks to wrapper classes, in each place where an **object type** is required, and we would like to use a primitive type, we should use wrapper class, e.g.:

```
List<Integer> ints = List.of(1, 2, 3, 4);
Set<Long> longs = Set.of(2L, 3L , 4L);
```

Despite the fact wrapper classes are objects we do **not** have to use constructors to create a new instance of such an object. For wrapper references we can assign values as if those were primitive types, e.g.:

```
byte b = 3;
Byte bObj = 3;

int intNumber = 7;
Integer intObject = 19;

long longNumber = 19L;
Long longObject = 120L;

float floatNumber = 1.1F;
Float floatObject = 2.2F;
```

```
double doubleNumber = 3.0D;
Double doubleObject = 3.1D;
```

Using constructors is possible but **not** recommended:

```
new Integer(3); // niezalecane
new Long(2L);
new Double(3.1);
```

However, there are some minor differences when using primitives types and wrapper classes. Such an exception is e.g. declaration of `long`. In case a number that fits an `int` is assigned, then we do **not** have to add `L` letter to the declaration, but if we want to assign a value to a `Long` then we always have to do it (even for small numbers), e.g.:

```
long a = 2_000_000_000; // ok
long b = 30000000000; // za duża liczba - błąd kompilacji
long c = 40000000000L; // ok

Long x = 2000000000; // błąd kompilacji
Long y = 2000000000L; // ok
```

Methods

Wrapper classes have access to such methods as `toString()`, `hashCode()` or `equals()` which are described in the later chapter. They do offer also some methods (some of them *static*) which allow conversion to and from primitive type. There also exists an *overload* of `valueOf` method which allows getting a value from a `String`, e.g.:

```
int a = 3;
Integer intA = Integer.valueOf(3);
Integer intB = Integer.valueOf("4");
int primitiveInt = intA.intValue();
System.out.println(intA.toString());

long b = 3;
Long longB = Long.valueOf(b);
Long longC = Long.valueOf("17L");
long primitiveLong = longB.longValue();
System.out.println(longB.toString());
```

Static methods and classes

Static methods

A static method is a method that does not require an object to be created. Most often static methods are used if we want to execute the same algorithm in many places within the project.

Declaration

To declare a static method before the returned type , simply add the keyword `static`, such as:

```
public class MyPrinter {
    static void printNumber(int number) {
        System.out.println("The number is: " + number);
    }
}
```

Calling

To call the static method, we don't need to create an instance of the object, but simply refer directly to the class, e.g:

```
public class MyPrinterExample {
    public static void main(String[] args) {
        MyPrinter.printNumber(10); // The number is: 10
    }
}
```

It should be remembered that the abuse of static methods is a **bad idea** that will result in negative consequences very quickly. Firstly, we don't write objective code at this point, which is in denial of the logic of using object language. Secondly, this way of writing code doesn't allow us to use the potential of so-called Object Oriented Programming (OOP).

Static internal classes

What's an internal class?

The easiest way to illustrate it is with an example:

```
public class MyOuterClass {
    private int outerNumber = 5;

    public class MyInnerClass {
        public void printNumber() {
            System.out.println(outerNumber);
        }
    }

    public MyInnerClass init() {
        return new MyInnerClass(); // = this.new MyInnerClass()
    }
}
```

In the above example, we are dealing with an external class called `MyOuterClass` and an internal class called `MyInnerClass` defined in `MyOuterClass`. Access modifiers used before the definition of an internal class work identically to attributes, methods or constructors. It should also be added that an internal class has access to all fields and methods of the external class (also private) in which it was defined.

Creating an internal class instance

To create an internal class instance we need an external class instance. This is illustrated by an example:

```
public static void main(String[] args) {
    MyOuterClass myOuterClass = new MyOuterClass();
    MyOuterClass.MyInnerClass myInnerClass1 = myOuterClass.init();
    myInnerClass1.printNumber(); // 5
    MyOuterClass.MyInnerClass myInnerClass2 = myOuterClass.new MyInnerClass();
    myInnerClass2.printNumber(); // 5
}
```

To refer to the `MyOuterClass.MyInnerClass` type is nothing more than to refer to the public internal type. In the example above, we have created instances of an internal class in two ways. The `myInnerClass1` instance is created by calling the `init()` method on an instance of an external class `myOuterClass` – this method in turn calls the constructor of an internal class `new in MyInnerClass()`. We also create the `myInnerClass2` instance using the `myOuterClass` instance, but here we openly call the internal class constructor, that is: `myOuterClass.new MyInnerClass()`.

What is a static internal class?

Let's start with a simple example:

```
public class MyOuterClass {
    public static class MyInnerClass {
        // declarations
    }

    public MyInnerClass init() {
        return new MyInnerClass();
    }
}
```

As can be seen in the example above, the only difference from a normal internal class is that the internal class declaration is preceded by a `static` modifier. It tells us that we are dealing with a declaration of **static** internal class. By default, all internal interfaces and enumeration types are static, the static modifier is redundant. What is important is the difference in creating an internal class static instance.

Creating an internal class static instance

Unlike the internal class, which is not static, you do not need an external class instance to create a static internal class

instance. This is illustrated by an example:

```
public static void main(String[] args) {
    MyOuterClass myOuterClass = new MyOuterClass();
    MyOuterClass.MyInnerClass myInnerClass = new MyOuterClass.MyInnerClass();
    MyOuterClass.MyInnerClass myInnerClass1 = myOuterClass.init();
}
```

We can see that it is enough to call the constructor using the static internal class type `new in MyOuterClass.MyInnerClass()` to create its instance.

When to use static internal classes?

If there is a reason why we want to have an internal class and simultaneously instance regardless of its external class, then it must be a static class.

Static fields

Declaration

Static fields are a class attribute, not an object. This means that it can be referenced without creating an object instance. To declare a static field, simply add a `static` modifier before declaring the type:

```
public class StaticFieldExample {
    public int myNumber = 10; // normal public field, requires a class instance
    public static int myStaticNumber = 15; // class static field, NOT requiring a class
instance
}
```

The `myStaticNumber` field will exist even if we do not create any instance.

Using static fields

Reference to a normal and static field is shown in the example below:

```
System.out.println(StaticFieldExample.myStaticNumber); // (1)
System.out.println(StaticFieldExample.myNumber); // Compilation error - attempt to refer
to a non-static field!
StaticFieldExample staticFieldExample = new StaticFieldExample();
System.out.println(staticFieldExample.myNumber); // (2)
```

In the example above, in the line *(1)* we referred to the `myStaticNumber` variable based on the class type, not the instance of the object of that class. Attempting to refer to a non-static field `myNumber` without an object instance will result in a compilation error. The *(2)* example is a valid reference to an instance of the `myNumber` field.

Date and time

One of the new features introduced in Java 8 is the new date and time API, also known as JSR-310, which is easy to understand, logical and largely similar to the Joda library (available even before Java 8). The main classes for date/time handling starting from local time will be discussed below.

Local time support (without TimeZone)

The main classes supporting local date and time (without time zones) are:

- **java.time.LocalTime **
- **java.time.LocalDate **
- **java.time.LocalDateTime **
- **java.time.Instant **

LocalTime

The class `LocalTime` represents time, without linking it to a specific time zone or even a date. This is the time without the time zone in the calendar system `ISO-8601`. It has some very useful methods:

- now() – a static method that returns the current time. The default format is `HH:mm:ss.nnn` (hour:minute:second.millisecond). Example:

is HH:mm:ss.nnnnnnnnn (hours:minutes:seconds:milliseconds). Example:

```
LocalTime localTime = LocalTime.now();
System.out.println("Current time: " + localTime); // Current time: 22:34:27.106
```

- **withHour(), withMinute(), withSecond(), withNano()**–set the time, minutes, seconds and nanoseconds in the current `LocalTime` object, e.g:

```
LocalTime localTime = LocalTime.now()
        .withSecond(0) // set the seconds to 0
        .withNano(0); // set nanoseconds to 0
System.out.println("Current time: " + localTime); // Current time: 22:41
```

- **plusNanos(x), plusSeconds(x), plusMinutes(x), plusHours(x), minusNanos(x), minusSeconds(x), minusMinutes(x), minusHours(x)** – adding (subtraction) nanoseconds, seconds, minutes, hours to (from) the set time, e.g:

```
LocalTime now = LocalTime.now();
System.out.println("Current time: " + now); // Current time: 22:49:01.241
now = now.plusMinutes(10).plusHours(1);
System.out.println("Current time after addition: " + now); // Current time after
addition: 23:59:01.241
```

- **getHour()** – returns the time that the object represents
- **getMinute()** – returns the minute of time that the object represents
- **getSecond()** – returns the second time that the object represents

The example below shows how to format the hour, minutes and seconds from the current time in any way you want:

```
LocalTime now = LocalTime.now();
String formattedTime = now.getHour() + ":" + now.getMinute() + ":" + now.getSecond();
System.out.println(formattedTime); // 22:55:26
```

LocalDate

The essence of the existence of the `LocalDate` class is to represent the date (year, month, day). This object does not take into account and does not store the time (e.g. current time) or time zone. The date is stored in the format `ISO-8601` by default. The main methods operating on local date objects are presented below:

- **now()** – a static method that returns the current date. The default format is `YYYY-mm-dd`, for example:

```
LocalDate now = LocalDate.now();
System.out.println(now); // 2020-03-27
```

- **of(year, month, dayOfMonth)** – creates an object representing a date (year, month, day). The month can be represented by the enum `java.time.Month` or the month index, e.g:

```
LocalDate localDate = LocalDate.of(2020, Month.MARCH, 28);
System.out.println(localDate); // 2020-03-28
```

- **getYear()** – returns the `int` representing the year
- **getMonth()** – returns a month using the `java.time.Month` object.
- **getDayOfYear()** – returns the `int` informing which day of the year
- **getDayOfMonth()** – returns the `int` representing the day of the month
- **getDayOfWeek()** – returns the day of the week using the enum `java.time.DayOfWeek`

LocalDateTime

The essence of the existence of the class `LocalDateTime` is the representation of date (year, month, day) and time (hour, minute, second, millisecond). This is a format without the time zone in the `ISO-8601` calendar system. Here are some of the methods from this class that are most commonly used:

- **now()** – a static method that returns the current date and time. The default format is `YYYY-MM-ddThh:mm:ss.mmm`, e.g.

```
```

```
LocalDateTime localDateTime = LocalDateTime.now();
System.out.println(localDateTime); // 2020-03-28T20:25:16.124
```

- **of(year, month, dayOfMonth, hour, minutes, seconds, milliseconds)** – a static method that returns a local date and time according to the set parameters (year, month, day of the month, hour, minutes, seconds, milliseconds). There are also overloaded equivalents of this method with a variable number of parameters. Example below:

```
LocalDateTime localDateTime = LocalDateTime.of(2020, Month.MARCH, 28, 20, 0, 10, 0);
System.out.println(localDateTime); // 2020-03-28T20:00:10
```

In the value printed let's note that the sign T is a conventional separator separating the date value from the time.

Instant

This class stands out from the others in that it represents a specific and clearly defined point in time (with an accuracy of one second). The second important feature is that it is not related to the concept of days or years, but only to the universal time, the so-called "time of the year". UTC. In short, it stores internally the number of seconds (to the nearest nanosecond) from a certain fixed point in time (1 January 1970 – the so-called **Epoch time**). It is best suited to represent time in a way that will be processed by the system and not displayed to end users. A class instance of the `Instant` can be used to create instances of classes such as `LocalTime`, `LocalDate` or `LocalDateTime`, e.g.

```
    LocalDateTime localDateTime = LocalDateTime.ofInstant(instant,
ZoneId.systemDefault());
    System.out.println(localDateTime); // 2020-04-19T18:33:29.116691800

    LocalTime localTime = LocalTime.ofInstant(instant, ZoneId.of("CET"));
    System.out.println(localTime); // 18:33:29.116691800

    LocalDate localDate = LocalDate.ofInstant(instant, ZoneId.ofOffset("UTC",
ZoneOffset.ofHours(2));
    System.out.println(localDate); // 2020-04-19
```

The classes that represent the intervals

`JSR-310` (**JSR -Java Specification Request**) introduces the concept of interval as the time that elapsed between moments A and B. There are two classes for this:

- **java.time.Duration**
- **java.time.Period **

They differ only in units (they allow to represent respectively: time units (Duration) and e.g. months or years (Period)). Examples follow:

```
System.out.println(Duration.ofHours(10).toMinutes()); // 10 hours expressed in minutes:
600

// In the example below, the time difference in minutes between the current time and the
time 2 days later was calculated

System.out.println(Duration.between(LocalDateTime.now(),
LocalDateTime.now().plusDays(2)).toMinutes()); // 2880

// The number of months between the two dates is calculated below.

System.out.println(Period.between(LocalDate.now(),
LocalDate.now().plusDays(100)).getMonths()); // 3
```

The distinction between `Duration` and `Period` is based on a simple fact – all lengths expressed by `Duration` are represented in basic units of time (i.e., e.g., in seconds), while those expressed by `Period` (month, year, age, millennium) may have different real lengths (by different number of days in months, leap years, etc.).

Date display format

For formatting type objects: The `LocalDate LocalTime LocalDateTime` is used by the `format(formatter)` method. An example for local time is given below:

```
LocalTime localTime= LocalTime.now();
String formattedLocalTime = localTime.format(DateTimeFormatter.ISO_LOCAL_TIME);
System.out.println(formattedLocalTime); // 21:11:00.024
```

Here is a list of predefined formats:

Formatter	Description	Example
ofLocalizedDate(dateStyle)	Formatter with date style from the locale	'2011-12-03'
ofLocalizedTime(timeStyle)	Formatter with time style from the locale	'10:15:30'
ofLocalizedDateTime(dateTimeStyle)	Formatter with a style for date and time from the locale	'3 Jun 2008 11:05:30'
ofLocalizedDateTime(dateStyle,timeStyle)	Formatter with date and time styles from the locale	'3 Jun 2008 11:05'
BASIC_ISO_DATE	Basic ISO date	'20111203'
ISO_LOCAL_DATE	ISO Local Date	'2011-12-03'
ISO_OFFSET_DATE	ISO Date with offset	'2011-12-03+01:00'
ISO_DATE	ISO Date with or without offset	'2011-12-03+01:00'; '2011-12-03'
ISO_LOCAL_TIME	Time without offset	'10:15:30'
ISO_OFFSET_TIME	Time with offset	'10:15:30+01:00'
ISO_TIME	Time with or without offset	'10:15:30+01:00'; '10:15:30'
ISO_LOCAL_DATE_TIME	ISO Local Date and Time	'2011-12-03T10:15:30'
ISO_OFFSET_DATE_TIME	Date Time with Offset	2011-12-03T10:15:30+01:00'
ISO_ZONED_DATE_TIME	Zoned Date Time	'2011-12-03T10:15:30+01:00[Europe/Paris]'
ISO_DATE_TIME	Date and time with ZoneId	'2011-12-03T10:15:30+01:00[Europe/Paris]'
ISO_ORDINAL_DATE	Year and day of year	'2012-337'
ISO_WEEK_DATE	Year and Week	2012-W48-6'
ISO_INSTANT	Date and Time of an Instant	'2011-12-03T10:15:30Z'
RFC_1123_DATE_TIME	RFC 1123 / RFC 822	'Tue, 3 Jun 2008 11:05:30 GMT'

Apart from ready-made formats, we can also prepare our own implementations. For this purpose, we must use special symbols that have a specific meaning and representation, such as

```
String date = LocalDate.now().format(DateTimeFormatter.ofPattern("MM:YYYY:dd"));
System.out.println(date); // 04:2020:19
```

A list and description of all available symbols for formatting the `LocalDate LocalTime LocalDateTime` objects can be found here.

Varargs

Java has introduced a method with a variable number of arguments called 'varargs'. This feature was incorporated with the fifth version of Java. When used, it enables the developer to use a variable-length of arguments without the need to resort to arrays.

In general, the method syntax can be written in this way:

```
[typ] methodName(type... groupOfArguments)
```

When you declare the group of arguments it it proceeded by it's type and it remains common for the whole group. It is then followed up by '...' and the given name of the argument 'groupOfArguments'.

Below you will find a simple example of 'varargs' usage, where for the main argument we provide a random number of integer values. We then print them out to the screen.

```
void printNumbers(int... numbers) {
    for (int i = 0; i < numbers.length; i++) {
        System.out.println(numbers[i]);
    }
}
```

Here is the example use of the method above:

```
printNumbers();         // nothing will be printed, correct usage
printNumbers(2);        // 2
printNumbers(3, 125);   // 3 125
printNumbers(1, 2, 3);  // 1 2 3
```

We also need to remember that the name of the group of arguments should its final element if we wish to use other arguments. It is worth mentioning, that when declaring those arguments we can use only one type of variables (only integers or longs and so on). Below you will find an **incorrect** use of 'varargs':

```
void incorrectVarargs(int... numbersGroupA, long... numbersGroupA) { // compilation error
  // main method
}
```

In the next example we will print out the given arguments with the distinction whether it's a constant argument or a member

of the variable arguments group:

```
void printArgs(int firstArg, int... numbers) {
    System.out.println("Permanent variable: " + firstArg);
    for (int i = 0; i < numbers.length; i++) {
        System.out.println("Variable argument: " + numbers[i]);
    }
}
```

Example usage:

```
printArgs();           // Compilation error!
printArgs(3);          // Constant argument: 3
printArgs(1, 2, 3);    // Constant argument: 1
                       // Variable argument: 2
                       // Variable argument: 3
```

As seen above, the constant argument is a mandatory element of the method execution – otherwise it will lead to compilation errors.

Complete Lesson