Java fundamentals slides

Programming

What is programming?

`Computer Programming` is the process of designing, creating, testing and maintaining source code for computer programs or microprocessor devices (microcontrollers).

The `Source code` is written in the programming language, using specific rules, it can be a modification of an existing program or something completely new.

In software engineering, programming (**implementation**) is only one of the stages of program creation.

Why Java?

- High-level object oriented language
- Platform Independence (Write Once, Run Anywhere)
- Automatic memory management
- Simplicity
- Popularity
- A big community
- Huge amount of literature
- High demand on the labour market

History of the Java language

- 1991 – Beginning of the Java language – Green project
- 1996 – First version of Java
- The next editions of Java up to the current one are mainly adding new functionalities and working on the performance of standard libraries. The biggest breakthrough was in version 5.0 (adding generic types etc.).

Java – Basics: First program

The first program “Hello World!”

Implementation

The first program will only display the phrase “Hello World!” as the output – nothing more:

```
package com.sda.example;

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

Compilation and running

Compilation and running

If you have correctly installed the OpenJDK boot and development environment, just run the following commands from the command line (system console):

```
javac HelloWorld.java
```

```
java HelloWorld
```

As a result of both these commands, the system console will display the text:

```
Hello World!
```

Java – Basics: Compilation process

How does the compilation process work?

Compilation

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

- Implementation
- Compilation
- Interpretation

Implementation

Implementation – creation of the application source code, i.e. a record of the computer program with the use of a specific programming language, describing the operations that the computer should perform on the collected or received data. The source code is a result of the programmer's work and allows to express in a human-readable form the structure and operations of the computer program.

Compilation

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 the source code into machine code understandable for JVM, so called **bytecode**. Besides that, the compiler is supposed to find lexical and semantic errors and optimize the code.

Interpretation

Interpretation – this is the process of machine code analysis and its execution. These actions are performed by a special program called **Interpreter**.

Installation of Java Runtime Environment

As developers, 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).

Java – Basics: data types, variables and constants

Variables and data types

What is a variable?

A variable is a programming structure with three basic attributes: a symbolic name, a storage location and a value; allowing the source code to refer to a value or storage location by name. The name is used to identify the variable, so it is often called **identifier**.

Variable declaration

```
private int number;
```

In general, the Java variable declaration can be described as follows:

```
{accessor*} { variable type} { variable identifier}
```

Variable Initialization

Variable initialization example:

```
number = 5;
```

The variable with the identifier `number` is initialized with an integer value of 5.

‘Global’ variables within a class

The operating range of such a variable is within the class in which it is declared.

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

‘Local’ variables within the methods

Coverage only within the method in which it was declared.

```
class ExampleClass {
    void someMethod() {
        int myLocalVariable = 5;
        System.out.print("My local variable: " + myLocalVariable);
    }
    int myGlobal = myLocalVariable; // Error – the variable myLocalVariable is not
visible outside the method in which it was declared
}
```

‘Local’ variables declared inside flow control instructions

```
class MyExampleClass {
    void someExampleMethod() {
        if (someCondition) {
            int a = 1;  // local variable declared inside the flow control instruction –
visible only inside this instruction
        }
        for (int i = 0; i < 10; i++) {
            // variable 'i' is only visible inside this loop
        }
```

```
    }
}
```

Final variables

Such variables cannot change value once initialized. During the declaration we also have to use the keyword `final`.

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

An attempt to change the value of the final variable ends with a compilation error:

```
private void finaVariableSample() {
    final int finalVariable = 123;   // declaration and initialization of the final
variable
    final long anotherFinalVariable;
    finalVariable = 12;             // an attempt to change the value of the final
variable ends with a compilation error
    anotherFinalVariable = 12345L;  // correct – final variable initialization
}
```

What is the data type?

A data type is a description of the type, structure and range of values that a character, variable, constant, argument, function result or value can contain.

Data types in java

In Java we have the following data types:

- numeric type
    - integer
    - floating point
- boolean type
    - true
    - false
- characters
- strings

Java as a static typed language

Java is a statically typed programming language, therefore:

- the types of variables are given during the compilation of the program
- it's easy to detect errors during compilation
- the types of variables must be declared before their initialization

Integers

The types representing integers in Java are divided into:

- byte
- short
- int
- long

byte

The `byte` type allocates 1 byte (8 bits) to the memory, and thus numbers from -128 to 127 (2^8=256) can be stored with it.

```
byte myByteNumber = 125;
```

short

The `short` type allocates 2 bytes (16 bits) to the memory, and therefore numbers from -32768 to 32767 can be stored with it.

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

int

The `int` type allocates 4 bytes to the memory, so you can store numbers from -2147483648 to 2147483647.

```
int myIntNumber = 1230000;
```

long

The `long` type allocates 8 bytes (64 bits) to the memory, so you can use it to store numbers from -2^63 to (2^63)-1 – this is really a lot 🙂 In practice, it is used to store e.g. entity identifiers in the context of databases. When declaring, add the suffix `L`.

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

In addition, there are wrapping classes, which are object-oriented equivalents of primitive types. They provide methods thanks to which many routine activities are always at hand.

Floating Point Types

The types representing floating point numbers in Java are divided into:

- float
- double

float

The `float` type allocates 4 bytes to the memory, so you can use it to store numbers to the maximum 6-7 decimal places. When declaring a number, add the suffix `F` or `f`.

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

double

The `double` type allocates 8 bytes to the memory, and thus you can use it to store numbers to the maximum 15 decimal places. When declaring a number, add the suffix `D` or `d`.

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

Note: The integer part is separated from the fraction by a dot, not a comma. It should also be remembered that floating point numbers are not suitable for financial calculations where accuracy is important.

Boolean type

The logical type in Java is the `boolean` type. It has only two possible values:

- true
- false

We often use boolean values in control instructions and loops.

```
boolean myFalseValue = false;
boolean myTrueValue = true;
boolean myBooleanValue = myFalseValue && myTrueValue; // myBooleanValue will be 'false'
```

Characters

Characters

The character type in Java is represented by `char`. A value of this type is declared by a single character enclosed in `'` quotes. It is used to represent single characters in Unicode. There are also special characters (example):

- \t -tab
-
- a new line
- \r -carriage return

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

Strings

The `String` data type is used to store a sequence of characters (text). It is the so-called **immutable** type – it means its state cannot be changed and it is an object type (about it in the following chapters). Its values should be included in double quotation marks.

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

Java – Basics: Operators

Division of operators

In Java we distinguish the following groups of operators:

- Assignment
- Arithmetic
- Logical
- Relational

Assignment operators

They allow the operation of assigning, entering a new value into a specific variable.

The operator = – assigns a value to the specified variable.

```
int intValue = 5;
```

The operator += – adds the specified value to the value of an existing variable and automatically assigns the result to that variable.

```
int a = 50;
a += 50;
```

Assignment operators

The operator -= – the same operation as the operator +=, only instead of the addition operation we have a subtraction operation.

```
int a = 50;
a -= 40;
```

The operator *= – the same operation as above, only instead of the + and – operations a multiplication is performed.

```
int a = 10;
a *= 10;
```

Assignment operators

The operator `/=` – the same action as the operator `*=`, only division is performed instead of the `*` operation.

```
int a = 200;
a /= 100;
```

Basic arithmetic operators

They act on the given arguments representing numerical values, as a result of which they also return the determined numerical value. Simply put, they perform basic arithmetic operations.

`+ - * / %` – represent the following operations, respectively: addition, subtraction, multiplication, division and the remainder from division (the so-called modulo operation).

Basic arithmetic operators – Example

```
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
    }
}
```

Incrementation operators

Increases the value of the variable by one and occurs in two forms: post and pre. It is equivalent to the expression `i = i + 1`:

- **Post-incrementation** returns the value of the variable first, and then modifies it.

```
int someVariable = 5;
System.out.print(someVariable++);   // a value of 5 will be printed
```

- **Pre-incrementation** first modifies the variable value and then returns it.

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

Decrementation operators

It reduces the value of the variable by one and occurs in two forms: post and pre. It is equivalent to the expression `i = i - 1`.

```
int someVariable = 10;
System.out.print(someVariable--);   // a value of 10 will be printed

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

Relational operators

They compare the arguments of an expression and return a logical value, based on verifying that the value is true.

Equality == – checks that both arguments are equal.

```
int a = 5;
int b = 6;
System.out.print(a == b);   // false will be printed
```

Inequality `!=` – checks if the two arguments are different.

```
int a = 5;
int b = 6;
System.out.print(a != b);   // true will be printed
```

Relational operators

Greater than `>` and greater than or equal to `>=` – checks whether the first argument is greater than the second and greater or equal respectively.

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

Less than `<` and less than or equal to `<=` – the opposite of the above.

```
int a = 5;
int b = 6;
System.out.print(a < b);     // true will be printed
System.out.print(a <= b);    // true will be printed
```

Logical operators

They act on arguments representing logical values, as a result of which they also return logical value.

The `&&` – **conjunction** accepts two arguments of the `boolean` type and also returns the `boolean` type. This can be translated as a sentence that the conjunction is true if, and only if, both its arguments are true (**logical product**).

```
boolean boolValue1 = true;
boolean boolValue2 = false;
System.out.print(boolValue1 && boolValue2);     // false will be printed
```

Logical operators

`||` – **alternative**, which is a logical sum. It is true if at least one of its arguments is true.

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

`!` – **negation**, or contradiction. It can be translated as "it's not true that".

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

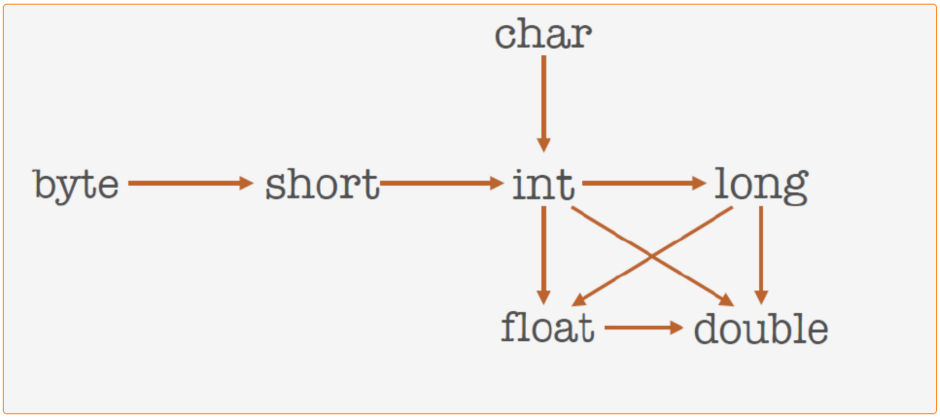
Java – Basics: Type conversion and casting

Automatic (widening) 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 **automatic (widening) conversion** in an automatic way.

```
int i = 25;
float j = i;    // Here, the automatic conversion to float type is done
```

Conversion of numerical types



Conversion of numerical types

All numerical conversions from the previous scheme except:

- int -> float
- long -> float
- long -> double

are lossless, only those mentioned above may involve partial loss of data.

Conversion of numerical types – basic 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 `float` type, the other will also be converted to `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 take the principle that a smaller type is converted to a type with larger bit capacity.

Casting – definition

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_with_smaller_capacity;
```

Casting – examples

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

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

The construction below will cause an error. (Let's think about why?)

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

Java – Basics: Conditional statements

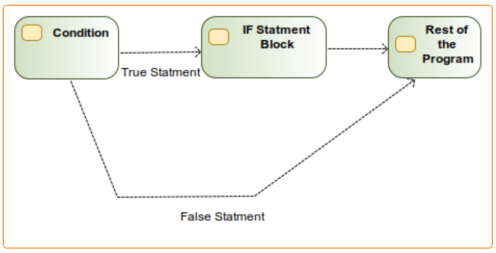
What is the conditional statement?

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

IF statement

It checks the logical condition and if it is true, the instructions in her body (inside the block) are executed, if not – they are omitted.

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



IF statement – example

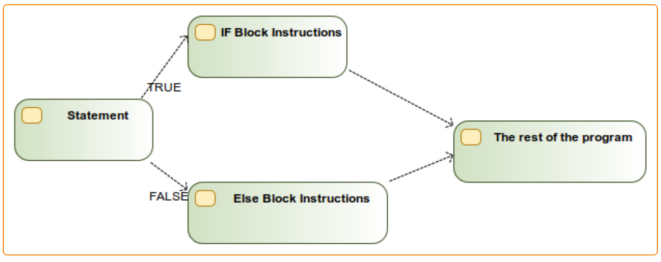
```
float temperature = 38.5f;
if (temperature > 36.6f) {
    System.out.print("You have a fever!");
}
```

IF ELSE

In the `if else` statement, the code in the `else` block is executed **if and only if** the logical condition declared in parentheses `if(...)` is not met.

```
if (condition) {
    // execute the code for the fulfilled condition
} else {
    // execute a code for an unfulfilled condition
}
```

IF ELSE – flowchart




```
float temperature = 36.6f;
```

```
if (temperature > 36.6f) {
    System.out.print("You have a fever!");
} else{
    System.out.print("You are healthy!");
}
```

if... else if... else

It is also possible to build a mixed `if... else if... else` with any number of blocks of `else if`, which gives the possibility to branch the expression to many conditions.

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

if... else if... else – example

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

SWITCH statement

The last discussed scheme can be replaced in a more convenient way, which is at the same time more readable, the `switch` statement.

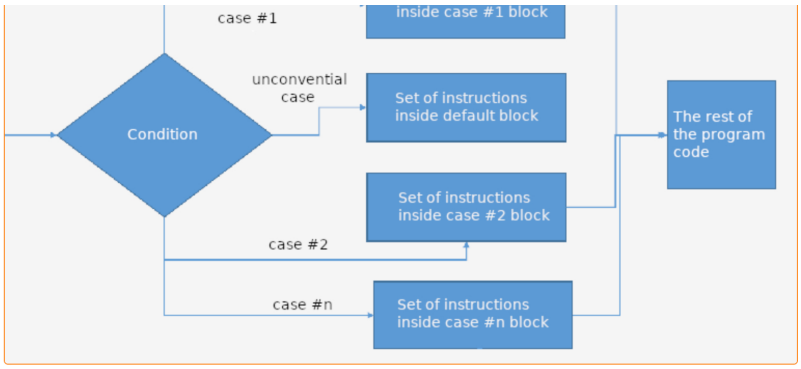
Characteristics:

- Covers multiple expressions `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`.

SWITCH statement – syntax

```
switch(variable) {
    case value1:
        // execute the code in case variable = value1
    break;
    case value2:
        // execute the code in case variable = value2
    break;
        // other cases
    default:
        // perform if and only if none of the above conditions are met
    break;
}
```

SWITCH statement – flowchart



Java – Basics: Loops

Features and types of loop design

Loop features:

- Perform the selected code block as long as conditions are met
- The code is executed by a finite number of circuits (loops)
- Loops that have no end are called infinite loops

In Java we distinguish the following types of loops:

- `for`
- `while`
- `Do while`

'for' loop

The general form of the `for` loop is as follows:

```
for(counter declaration; condition; modification of the counter) {
    // instructions
}
```

Phases of execution:

- **counter declaration** and the so called **initial condition** (loop counter is set)
- **the final condition** – this is nothing more than a check of the condition: if the condition is `true` – the code inside the loop is executed, otherwise – it's not
- **modification of the counter** – after the loop is completed, the loop counter is updated

'for' loop – examples

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

The `for` loop is also ideal for iterating through collections/tables:

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

'while' loop

The `while` loop is most often used in places where the assumed number of repetitions is undefined, but we know the condition that must be met. The general scheme is as follows:

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

The simplest way to describe the operation of the `while` loop is to describe it this way: "Follow the instructions inside the block as long as the condition in the loop is fulfilled".

'while' loop – examples

A simple example can look like this (the action will be the same as in the `for` loop example):

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

'do while' loop

All instructions executed in the loop block will be executed at least once, as the condition is checked at the end (after executing the instructions inside the loop).

The general scheme is as follows:

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

'do while' loop – examples

An example similar to the previous ones (loops: `for` and `while`), would look like this:

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

---

break

The `break` statement will terminate the loop that is currently executing. The next iteration will not be started, e.g:

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

---

continue

Using the `continue` statement will jump to the next loop iteration, e.g:

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

Java – Basics: String class

String class – Introduction

The `String` class is used to represent character strings. The text as a `String` can be declared in two ways:

- Using literal:

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

- Using the constructor – with usage of `new` keyword:

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

Immutability of String objects

The `immutability` of String class 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";
```

What does the immutability give us?

- safety – is safe in multithread-safe applications
- performance
- the `String pool` – multiple references can point to the same object thanks to the pool – the Java optimization mechanism

String pool

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

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

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

Concatenation of Strings

`Concatenation` of Strings is nothing more than a combination of them:

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

String text3 = "This is ";
String text4 = "a test";
String finalText2 = text3.concat(text4);
```

Comparing Strings

To check if the two Strings are identical, use the `equals` method:

```
String text1 = "Text to compare";
String text2 = "Text to compare";
```

```
System.out.print(text1.equals(text2)); // the value true will be displayed because the
two Strings are identical
```

String class methods

- length() – returns length of String
- toUpperCase() and toLowerCase() – they replace the original String with the same String consisting of capital letters and, in the second case, lower case letters:

```
System.out.print("This is test value".length()); // 18
String testValue = "This is test value";
System.out.print(testValue.toUpperCase()); // THIS IS TEST VALUE
System.out.print(testValue.toLowerCase()); // this is test value
```

String class methods – indexOf()

- indexOf() – returns the position of the first occurrence of the specified text in a String:

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

String class methods – replaceAll()

The `replaceAll()` method allows you to replace all occurrences of a given substring with another:

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

Standard input/output

Scanner – basic input

The `Scanner` class is ideal for retrieving user input as well as the basic reading of the file:

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

Scanner class methods

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

Standard output – print() and println() method

`print()` method prints a given string or a numeric variable on the screen, which it converts previously to the `String` type:

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

`println()` – works the same as `print()`, but also adds a new line character at the end.

Standard output – printf()

`printf()` – a method that can format data in addition to outputting it. Special conversion operators are used for this purpose:

```
* e – floating point number in exponential notation
* f – floating-point number
* x – integer in hexadecimal system
* o – integer in the octal number system
* s – string
* c – one character (char)
* b – logical value
```

System.out.printf() – examples

```
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 result of the division will be rounded off
```

Java – Basics: Regular expressions

What are regular expressions?

Regular expressions (also called `regex`) are patterns that allow us to check if a string has a format we have specified (e.g. if it can be a date). Regular expressions allow us to check if the user data has the correct format, e.g. regex representing a name looks like this:

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

Structure of regular expressions

- consist of a sequence of atoms
- The simplest atom is a letter, a number, a special character.
- the letters can be grouped in parentheses
- `quantifiers` – number of occurrences of a given atom

The simplest regular expression might look like:

```
abcde
```

Quantifiers

The quantifier describes the number of repetitions of a given substring in the character string under consideration. We will check it with an example:

```
a+bcde
```

Table of most popular quantifiers

Quantifier	Meaning	Example	The example matches
*	Zero or more occurrences	a*b	ab, b, aab, aaaaaab, aaab (and similar)
+	One or more occurrences	a+b	ab, aab, aaaaaaab, aab (and similar

?	Zero or one occurrence	a?b	ab, b
{n,m}	At least n and maximum m of occurrences	a{1,4}b	ab, aab, aaab, aaaab
{n,}	At least n occurrences	a{3,}b	aaab, aaaab aaaaab (and similar)
{,n}	Maximum of n occurrences	a{,3}b	b, ab, aab, aaab
{n}	Exactly n occurrences	a{3}b	aaab

Ranges and groups

- ranges in regular expressions as character groups
- the range is also an atom
- the ranges are defined in square brackets in two ways:
    - by exchanging all possible characters (without commas, one next to the other)
    - by introducing a range (e.g.: 1-3 ; 0-9; 1-5, a-z ; A-Z)

Examples of ranges used in regular expressions

Expression	Description
[abcde]	one of the letters: a, b, c, d or e
[a-zA-Z]	One of the letters a to Z lowercase or uppercase
[a-c3-5]	Letter a to c or number 3 to 5
[a-c14-7]	Letter a to c or number 1 or number 4 to 7
[abc\[\]]	The letter a or b or c or a square parenthesis (why we also added inverted slashes, read more)
[.]	Any letter (read more)

Implementation of regular expressions in Java

In Java, most operations on regular expressions are performed using classes:

- `Pattern`
- `Matcher`

Pattern

- Class `Pattern` as representation of a compiled regular expression
- An object with an expression representation is obtained by calling the `compile(regexAsString)` static method:

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

Matcher

- a class instance of `Pattern` has a method `matcher()` that returns a class instance of `Matcher`
- an object of type `Matcher` has a method of `matches()` – checks whether the string used to create an instance of a class `Matcher` matches a regular expression

```
Matcher matcher = pattern.matcher("aaaaabcd aaaaaaabbcd");
matcher.matches();  // returns true/false
```

Matcher

The `Matcher` class also has the `find()` method – it checks if there is anything in the character string that matches a regular expression:

```
matcher.find();     // returns true/false
```

Java – Basics: Arrays

Definition and types of arrays

Features:

- an array as a container to hold structured data of one type
- referring to individual elements by means of **index**
- the array size should be fixed

There are two basic types of arrays:

- one-dimensional
- multidimensional

One-dimensional arrays

Declaration

Table-type declaration scheme:

```
type[] table_name = new type[number_of_elements];
```

Example:

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

Declaration with initialization of the elements

It is also possible to declare an array without specifying the size of it, but individual elements of the array must be explicitly initialized:

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

Examples of array 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];
```

Array initialization and default values

If we declare an array and do not specify the data to be filled in, it will be filled in with default values for the selected type, e.g. for numbers it will be 0, and for reference variables: `null`:

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

Indexes of arrays

Writing/reading elements:

Writing/reading elements:

- by means of indices:
  - natural numbers
  - the numbering starts with `0`
  - the last element: `Size_table-1`
- Beware of the `java.lang.ArrayIndexOutOfBoundsException` exception .

Setting array values with indexes

```
String[] names = new String[4];
names[0] = "Jan";
names[3] = "roman";
System.out.println("Element number 1: " + names[0]); // Element number 1: Jan
System.out.println("Element number 2: " + names[1]); // Element number 2: null
System.out.println("Element number 3: " + names[2]); // Element number 3: null
System.out.println("Element number 4: " + names[3]); // Element number 4: roman
```

Iteration over the array using the loop for()

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

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

Getting the array size

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

```
String[] myArray = new String[10];
System.out.println(myArray.length); // 10
```

Multidimensional arrays

Definition

- The array can also store other arrays because it is an object
- Two, three or multi-dimensional arrays can be created
- A two-dimensional array is nothing more than a structure containing rows and columns that store data

Declaration

Basic declaration and initialization of the two-dimensional array:

```
type[][] array_name;                                          // array declaration
array_name = new type[number_of_rows][number_of_columns];     // array
initialization
type[][] array_name = new type[number_of_rows][number_of_columns]; // declaration and
initialization at once
```

initialization

The number of columns and rows do not have to be identical, so you can create any rectangular arrays:

```
String[][] myArray = new String[2][];
myArray[0] = new String[]{"Alice", "has", "the", "cat"}; // creating the first row (index
number 0)
myArray[1] = new String[]{"The", "cat", "has", "Alice"};  // creating the second row
(index number 1)
```

Retrieving any array value

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

Iteration over an array

- The first loop of `for()` iterates on the rows
- The second loop iterates on the columns

```
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();
}
```

Iteration over an array – things good to know

What's worth remembering?

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

---

Java – Basics: methods

Methods

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

- Reusability of repeating code fragments
- The sensible division of the code into smaller parts makes it easier to understand

---

Definition of methods

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

Example:

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

Arguments of the methods

The methods may accept any number of arguments of any type or may not accept them at all. If a method contains several arguments, they are separated by commas.

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

Returned value

The method can return some value. The value returned by the method is preceded by the keyword `return`. In the method signature we additionally declare the returned type.

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

Returned value void

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 `void` type.

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

Early interruption of the void return method

The returned value can be any simple or object type. We can also prematurely interrupt a method that does not return any values and exit it using the keyword `return` without providing a value.

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

Body of the method

The body of a method is the entire code contained between the curly brackets of the `{}` method definition.

```
{
    return arg1 - arg2;
}
```

```
{
    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.

Calling the method

Calling a method is to refer to the name of the method with or without the relevant parameters.

Definition example:

```
int multiple(int arg1, int arg2, arg3) {
    return arg1 * arg2 * arg3;
}
```

and its calling:

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

Methods naming convention

In Java it has been established that the names of the methods are written in the so-called **camelCase** standard, which means that the subsequent words are started each subsequent one with a capital letter (except for the first one), e.g.

- someMethod
- thisIsSimpleMethodName

The same principle applies to variable name declarations.

Java – Basics: Classes and Objects

Classes

Introduction

Classes:

- They are used to describe the objects, activities, states and relations that exist between them.
- They are defined in applications and they can be used to create variables with **complex** or otherwise **referential** types.
- They have two basic components:
    - **field** – a variable feature that describes an object of a given class
    - **method** – operation

A class is a schema consisting of fields and methods defined in it.

A simple example of a class

```
public class Movie {
    private String title;
    private String description;
    private int productionYear;

    public void play() {
        // instructions
    }
}
```

A slightly more complex class example

```
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));
    }
}
```

Class definition

The class definition can be defined as follows:

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

    // method definitions
}
```

Objects

Introduction

What are objects?

You can create copies (**instances**) of classes from a template. These instances will contain fields defined within the class and you can call methods on them defined in the class. Instances that are created from the class are called **objects**.

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 `car1` and `car2`. **constructor**.

Calling methods and setting object field values

```
Car car1 = new Car();
car1.setBrand("Mercedes");
car1.setColor("white");
car1.setMaxSpeed(250);
car1.printCarParameters();
```

- To call a method on an object, refer to the name of that method using the dotted notation on the object reference variables.
- We also use the dotted notation to retrieve field values.

Access Modifiers

- They help determine the scope and visibility of methods and class fields
- They set the visibility of classes
- Who, and in what situation, will be able to use the fields and methods

There are four basic modifiers:

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

- `private`

Private and public modifiers

- the `public` modifier gives access to our class, methods, fields from anywhere in our application – they are **public**.
- the `private` modifier gives access only from a defined class – the fields and methods are then **private** for our class.

Example of private and public modifiers

```
public class Book {
    public String title;        // public field
    public String author;
    private int numberOfPages;  // private field

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

    private boolean isNumberOfPagesIsCorrect(int numberOfPages) {   // private method
        return numberOfPages > 0;
    }
}
```

Referring to the public and private components

```
public class BookTest {
    public static void main(String[] args) {
        Book testBook = new Book();
        testBook.author = "Michael Crichton";
        testBook.title = "Jurrasic Park";
        testBook.setNumberOfPages(250);

        System.out.println("Book title: " + testBook.title);
        System.out.println("Author of the book: " + testBook.author);

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

When should you use appropriate access modifiers?

- All class fields should be private if possible – this includes the use of accessors (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 classes shall be public.

Getters and setters

- Encapsulation
- The fields should be private where possible (the use of accessors)
- The division of the accessors:
    - **getters** – are used to retrieve the value of a given field
    - **setters** – are used to set the value of a given field

Accessors naming convention

The getter methods take the prefix **get** in the name, and the setter methods take the setter: **set**. The second part of the name

should be the name of the field being set up or downloaded, e.g.:

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

Example of the definition of accessors

```
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;
    }
}
```

Calling accessors outside the class definition

```
public class BookTest {
    public static void main(String[] args) {
        Book testBook = new Book();
        testBook.setAuthor("Michael Crichton");
        testBook.setTitle("Jurassic Park");

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

Packages

Introduction

Using packages:

- Minimize potential problems with repeating the same class name.
- Separate classes into separate packages.
- **Package**, as an integral name of our class.

Defining

We declare packages using the keyword `package`:

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

The packages reflect the name of the reversed Internet domain.

Definition of classes with the same names

When defining several classes with the same names, we have to put them in separate packages:

```
package pl.sdacademy.myfirstbookexample;

public class Book {
```

```
    // definitions
}
```

```
package pl.sdacademy.mysecondbookexample;

public class Book {
    // definitions
}
```

Rules of creating packages

- The class belonging to the package (keyword `package`) must be at the beginning of the file,
- The only thing that can precede the use of the `package` is the comments,
- It is best that package names contain only letters and numbers,
- The package names are separated by dots,
- The package name should correspond to the directory structure on the disk:

```
pl
|
--sdacademy
|
--myfirstbookexample
```

Importing classes

- Importing classes using the keyword `import`.
- Import all classes from a package using the * keyword instead of the imported class name.

```
package pl.sdacademy;

import pl.sdacademy.bookexample.Book;

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

Import of several classes with the same name

In case you need to import two classes with the same name, you should openly import one of them and refer to the other one in code after the full package name:

```
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)
    }
}
```

Static import

In order to make it easier for us to use constants and static methods from other packages, we are helped by the so called **static import**. All we need to do is import our methods/constants together with the keyword `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));
    }
}
```

Default access

- **default** – otherwise known as **package-private**
- If we don't use any access modifier, our field, method or class will just get access **default**.
- It is almost as restrictive as the `private` modifier, but allows access to classes that are in the same package.

Default access – example

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;
    }
}
```

Modifier protected

- It has a common feature with the default modifier, i.e. it allows access to fields and methods from classes in the same package.
- Elements of the class are available for the class itself and its subclasses.

Constructors

Definition

The 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.

- The name of the constructor is identical to the name of the class in which we define it.
- The constructor method does not return anything, or more precisely, does not even return the `void` type – there is no return type declaration before the constructor method name.
- Constructor methods, like normal methods, can accept any number of input arguments.

Example of the definition of constructor

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

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

The definition of a constructor that we are analysing is the definition of the so called **Parametric constructor**. There is another variant of the constructor – **default constructor**.

Default constructor

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

- The example above is to create an instance of `Car` using a non-parameter or otherwise default constructor.
- The default constructor method does not take any parameters.
- The non-parameterless constructor is the default – you don't need to define it because you inherit it from the `Object` class.

Explicit definition of a parameterless constructor

```
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.

Constructor overloading

- Each class can have multiple constructors.
- To have more than one constructor, each must differ in number, type or order of arguments (according to the method overload rules):

```
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";
    }
```

Constructor overloading – continuation

```
    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;
    }
}
```

Calling overloaded constructor

The calling of overloaded constructors is performed in the same way as the calling of normal methods:

```
Car car1 = new Car();
Car car2 = new Car(250, "Mercedes");
```

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

Java – Basics: Static methods and classes

Static methods – definition

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 in the project. The declaration is very simple: before the type returned by the method, just add the keyword `static`:

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

Calling static methods

To call the static method, we do not need to create an instance of the object, but simply refer directly to the class:

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

Internal static classes

What is an internal class?

```
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`.

Creating an internal class instance

To create an internal class instance we need an external class instance:

```
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
}
```

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 you can see 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 here with a static internal class declaration.

Creating an internal class static instance

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();
}
```

Static fields

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 type:

```
public class StaticFieldExample {
    public int myNumber = 10;
    public static int myStaticNumber = 15;
}
```

Reference to static fields

```
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 refer to the static field `myStaticNumber` based on the class type and not the instance of the object of that class.

Java – Basics: Date and time

Local time (without TimeZone)

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

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

LocalTime

`LocalTime` class represents time, without linking it to a specific time zone or even a date.

- **now()** – returns the current date in the format **HH:mm:ss.mmm** (hours:minutes:seconds.milliseconds).

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

LocalTime class- overview of selected methods

- **withHour(), withMinute(), withSecond(), withNano()** – they set the time, minute or second that we indicate at the current time
- **plusSeconds(x), plusMinutes(x), plusHours(x), minusSeconds(x), minusMinutes(x), minusHours(x)**

```
LocalTime localTime = LocalTime.now()
        .withSecond(0)                                  // set the seconds to 0
        .withNano(0);                                   // set the nanoseconds to 0
System.out.println("Now is the time: " + localTime);    // Now is the time: 22:41
LocalTime now = LocalTime.now();
System.out.println("Now is the time: " + now);          // Now is the time: 22:49:01.241
now = now.plusMinutes(10).plusHours(1);
System.out.println("Now is the changed time: " + now);  // Now is the changed time:
23:59:01.241
```

LocalTime class- overview of selected methods

- **getHour(), getMinute(), getSecond()** – Returns respectively: hour, minute, second of time that the object represents

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

LocalDate – overview of selected methods

The essence of the existence of the class `LocalDate` is to represent the date (year, month, day) without taking into account the time zone in the `ISO-8601` calendar system.

- **now()** – returns the current date in the format **YYYY-mm-dd**:

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

LocalDate – overview of selected methods

- **of(year, month, dayOfMonth)** – creates an object representing a date (year, month, day):

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

- **getYear()** – returns the year in **YYYY** format
- **getMonth()** – returns a month in **MM** format
- **getDayofTheMonth()** – returns the day of the month in **DD** format
- **getDayofTheWeek()** – returns the day of the week in **XX** format
- **getDayOfYear(), getDayOfWeek(), getDayOfMonth()** – returns the information which is respectively: day of the year, week, month

LocalDateTime

The essence of the existence of the class `LocalDateTime` is the representation of date (year, month, day) and time (hour, minute, second, millisecond) without taking into account the time zone in the `ISO-8601` calendar system.

- **now()** – returns the current date and time in the format `YYYY-MM-ddThh:mm:ss.mmm`.

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

LocalDateTime – overview of selected methods

- **of(year, month, dayOfMonth, hour, minutes, seconds, milliseconds)** – a static method that returns a local date and time according to set parameters (year, month, day of the month, hour, minutes, seconds, milliseconds):

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

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 nanosecond). 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 **UTC** . In short, it internally stores the number of seconds (with accuracy to nanoseconds) from a certain fixed point in time (January 1, 1970 – the so-called **Epoch time**).

Classes representing time intervals

`JSR-310` (**JSR –Java SpecificationRequest**) introduces the concept of time interval as the time that elapsed between moments A and B. Two classes are used 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)).

Duration and Period – use examples

```
System.out.println(Duration.ofHours(10).toMinutes());   // 10 hours expressed in minutes:
600
// In the example below, the difference in minutes between the current time and the time
2 days later is 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
```

Display date format

To format objects of type: `LocalDate LocalTime LocalDateTime` the `format(formatter)` method is used:

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

List of predefined formatters

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'

Java – Basics: Varargs

Definition

- `varargs` are methods with a changeable number of arguments
- this mechanism allows you to create methods that can contain different number of arguments without the need to use arrays that store them
- the declaration concerning changeable number of arguments should be the last one
- with `varargs`, any number of arguments of the same type can be given in the method

```
[type] method (type fixedArgument), type ... varargs)
```

Examples

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

Example method calls:

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

Example with a fixed argument

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

Example method calls:

```
printArgs();            // Compilation error!
printArgs(3);           // Fixed argument: 3
printArgs(1, 2, 3);     // Fixed argument: 1
 varargs argument: 2
 varargs argument: 3   /*
 means a new line */
```

Complete Lesson