FITPED

Java classes

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

www.fitped.eu

2021

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

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

Java Classes

Published on

November 2021

Authors

Ján Skalka | Constantine the Philosopher University in Nitra, Slovakia

Ľubomír Benko | Constantine the Philosopher University in Nitra, Slovakia

Jaromír Landa | Mendel University in Brno, Czech Republic

Mariusz Boryczka | University of Silesia in Katowice, Poland

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

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

Reviewers

Anna Stolińska | Pedagogical University of Cracow, Poland

Peter Švec | Teacher.sk, Slovakia

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

Piet Kommers | Helix5, Netherland

Graphics

Ľubomír Benko | Constantine the Philosopher University in Nitra, Slovakia

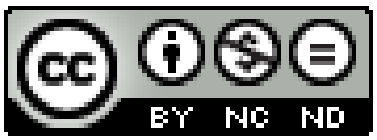
David Sabol | Constantine the Philosopher University in Nitra, Slovakia

Erasmus+ FITPED

Work-Based Learning in Future IT Professionals Education

Project 2018-1-SK01-KA203-046382

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



ISBN 978-80-558-1778-1

Table of Contents

Introduction to Object Oriented Programming

Chapter 1

1.1 Classes

📖 1.1.1

In the 1980s, programming was a complex and challenging activity. The programs were long and confusing, and each modification required a lot of time and understanding of the original programmer's thinking.

Object-oriented programming (OOP) came into practice in the 1990s, bringing a new program organization that follows the real-world layout and clearer code.

The concept is based on an **object** that represents an element with the features and capabilities that it provides to the programmer. When writing a program, objects are created and used as needed, with the programmer not interested in detailing the object in the first step but concentrating on solving the problem from a human perspective and assuming that the objects have all the skills he needs for his program.

📝 1.1.2

True or false? Object-oriented programming follows real-world ordering.

- True
- False

📖 1.1.3

Object-oriented programming (OOP) focuses on expressing the problem in such a way that it does not match the programmer's thinking to the programming language but to the programmer's language.

The programmer uses objects as if they had all the capabilities he needed. If they do not have them, he will add them as needed and code the details. When designing a solution, however, thanks to the objects do not need to deal with details but the problem itself.

Example: We have an Employee object that contains all the necessary information about the employee, it has the ability to change its salary or marital status, or move it to another workplace. In the code, the James object can then be manipulated as follows.

```
if (businessIsGood) {
     James.changeSalary(+100);
```

```
} else {
      James.changeSalary(-50);
}
// if James got married
if (Wedding) {
  James.changeStatus("married");
  James.changeSalary(+20);
}
```

We will only think about the change in status or salary change after we describe the whole process program behavior.

All the procedures and principles that we have learned are applied and applied when working with objects.

1.1.4

Which structures can be used for programming in Object-oriented programming?

- sequence
- branching
- cycle
- compound conditions
- arrays

1.1.5

From the point of view of using a programmer, we can imagine the object as a black box that has some memory in which it keeps the necessary information and abilities that we can use.

Memory is a variable and is referred to as **attributes** in Object-Oriented Programming terminology.

Object capabilities are described using programming language commands and represent partial algorithms that solve simple tasks. We call them **methods**.

1.1.6

Select true statements:

- Object memory are variables and are referred to as attributes.
- Object capabilities are partial algorithms and are referred to as methods.
- Object memory is a simple algorithm and is called an attribute.
- Object memory are variables and are referred to as methods.
- Object capabilities are partial algorithms and are referred to as attributes.

🕮 1.1.7

The concept of an object is retreating into the background in Java and is replaced by the terms **class** and **instance**.

A **class** is a rule that determines what objects will be created based on and what attributes to use. However, it does not yet exist in memory, so its variables cannot be used or called its methods.

An **instance** is already a specific object created in memory that provides us with its attributes and capabilities represented by methods.

The class represents a template that can be used to create instances with different attributes (appearance, ingredients, quality, etc.)

Class:

Instances:

A class is a rule that tells what attributes and abilities an instance has.

An instance is already a specific element that **HAS** characteristics and **IS CAPABLE** of performing activities.

1.1.8

Select true statements:

- Class is a prescription that creates instances.
- If we do not have an instance created, we do not have access to the attributes or methods defined in the class.
- If we have an instance created, we can use both the attributes and methods defined in the class.
- If we have created a class, we can use both the attributes and methods defined in the instance in the program.
- An instance is a prescription that generates classes.

1.1.9

Fill in correct terms:

A rule or template that is used to create Java objects is called _____.

An object that contains data and is capable of performing activities in Java is called _____.

The variable used to store data in an object is called _____.

The partial algorithm representing the capabilities of the object is called _____.

1.2 Classes in Java

📖 1.2.1

The simplest example of a class is a class that has only attributes defined and does not use any methods.

The general declaration:

```
class ClassName {
   typ1 variableName1;
   typ2 variableName2;
   ...
}
```

It is established that the class name begins with a capital letter.

📝 1.2.2

What command do we use to define a class?

📖 1.2.3

Design a Tree class that will have its kind, age, height, and perimeter.

The class name will be Tree and each attribute will be a variable for storing numeric and text values, such as:

```
class Tree {
   String type;
   int age;
   double height;
   double perimeter;
}
```

This structure represents any tree:

1.2.4

True or false?

Class attributes are represented by variables of any type (both primitive and reference).

- True
- False

1.2.5

The class represents only the structure by which instances are created in memory. To store data about a particular tree, we need to create instances of the Tree class - a separate instance for each tree.

An instance is just a class variable. We declare it as a variable, its type being a class. *Example:*

```
Tree oak;
```

To create an instance in memory, you must use the **new** command followed by specifying the class (template) to create:

```
oak = new Tree();
```

or we can use the declaration with creation in one command:

```
Tree pine = new Tree();
```

We will use the **new** keyword to allocate memory for instances, just as we do with memory allocation for arrays.

We can create any number of instances of any class in the program.

```
public class App {
 public static void main(String[] args) {
```

```
    Strom oak = new Tree();
    Strom pine = new Tree();
    Strom pear = new Tree();
    ...
  }
}
```

1.2.6

What command do we use to create an instance of any class?

1.2.7

A class selector - character "." is used to access the class attributes. Allows you to insert or read content into an instance attribute:

```
public class App {
  public static void main(String[] args) {
    Tree oak = new Tree();
    oak.type = "deciduous";
    oak.age = 21;

    Tree pine = new Tree();
    pine.type = "coniferous";
    pine.height = 15.2;
    System.out.println("Pine: " + pine.type + " - " +
pine.height);
  }
}
```

If some attributes are not set, the default value will be stored (0 for numeric types).

1.2.8

Fill the code to create the instance and set its attributes.

```
_____ Person {
    String name;
    int age;
}
```

```
public class App {
   public static void main(String[] args) {
     Person men1 = _____ Person();
     man1_____name = "Peter";
     man1_____age = 25;
     System.out.println(man1.name + ":" + man1.age);
     Person director = _____ Person();
     director_____name = "Paul";
     director_____weight = 120;
     System.out.println(director.name + ":" + director.age);
  }
}
```

📖 1.2.9

Manipulation of attributes is the same as for simple variables. We load and write to them using the dot selector.

Usually, a class is created in a separate file but it is possible to have multiple class definitions and a start code in the same file.

```
class Person {
   String name;
   int age;
   double salary;
}

public class App {
   public static void main(String[] args) {
     Person man1 = new Person();
     man1.name = "Peter";
     man1.age = 25;
     man1.age++;

     if (man1.salary < 800)  man1.salary = 800;
     ...
  }
}
```

1.2.10

Fill in the code. If the car brand is BMW or Audi set the maximum speed to 250, otherwise set it to 210.

```
class Car {
   String brand;
   int price;
   int maxSpeed;
}

public class App {
   public static void main(String[] args) {
     Scanner scn = new Scanner(System.in);
     String b = scn.nextLine();
     Car car1 = _____ _____;
     car1.brand = b;
     if ((car1.brand._____("BMW")) _____
(car1.brand._____("Audi")) {
        car1._____ = _____;
     } else {
        car1._____ = 210;
     }
     System.out.println(car1.brand + "have maximal speed " +
car1._____);
  }
}
```

1.2.11

In some cases, setting the attributes in the code is unnecessarily lengthy. This is particularly the case when some attributes have the same default value. In such cases, we can put the default value into the attributes just as you would when declaring ordinary variables.

When you create an instance, these values are then set to the appropriate attributes.

```
class Employee {
   String name;
   double salary = 998;
   String degree = "Ing.";
}
```

In general, it is not appropriate to determine the same value for a name unless we assume that all of our employees are called Peter or James.

If all newly recruited employees have the same starting salary, it is a good idea to set it directly in the declaration - saving each line of code after creating each new instance.

If we assume that employees with a technical university education will be recruited for the given positions, they are usually Ing. If we hire an employee with a different title, we will simply overwrite it.

In this case, the code

```
public class App {
   public static void main(String[] args) {
     Employee new1 = new Employee();
     new1.name = "Carlos";
     System.out.println(new1.name + ", " + new1.degree + " ,
salary: "  + new1.salary);
  }
}
```

will print the following information:

```
Carlos, Ing., salary: 998
```

1.2.12

Fill the code so that the newly created Tree instances are set to 2 years old and 0.8 m high.

```
class Tree {
   String type_____
   int age_____
   double height_____
   double perimeter_____
}
```

1.3 Class definition (programs)

1.3.1 House

Create a **House** class that will have the following attributes:

- **color** - plaster color
- **height** - the height of the house
- **width** - width of the house
- **rooms** - number of rooms

Choose the appropriate data types and set the following default values for the attributes:

- number of rooms - 5
- width - 10.5

An illustration of creating instances and setting attributes is available in the **MainApp.java** file.

MainApp.java

```
public class MainApp {

    public static void main() {
        // this is just an example usage, this code is not
executed
        House example = new House(); // creates instance
        example.color = "green";
        example.height = 6.2;
      example.width = 12.5;
      example.rooms = 8;
    }
}
```

⌨ 1.3.2 Tree

Create a **Tree** class that will have the following attributes:

- **name** - the name of the tree
- **height** - the height of the tree
- **leafs** - whether it has or leaves
- **age** - age of the tree

Select the data types appropriately and set the following default values for the attributes:

- name - elm
- height - 17.8

An illustration of creating instances and setting attributes is available in the **MainApp.java** file.

MainApp.java

```
public class MainApp {

    public static void main() {
        // this is just an example usage, this code is not
executed
        Tree example = new Tree();
        example.name = "maple";
        example.height = 16.2;
      example.leafs = true;
      example.age = 12;
    }
}
```

⌨ 1.3.3 Person

Create a **Person** class that will have the following attributes:

- **name**
- **job** - occupation
- **age** - age
- **kids** - number of children

Select the data types appropriately and set the attributes to the following default values:

age - 23

number of children - 1

An illustration of creating instances and setting attributes is available in the **MainApp.java** file.

MainApp.java

```
public class MainApp {

    public static void main() {
        // this is just an example usage, this code is not
executed
        Person example = new Person();
        example.name = "Ivan";
        example.job = "manager";
       example.age = 28;
       example.kids = 2;
    }
}
```

Methods

Chapter 2

2.1 Methods

📖 2.1.1

The object-oriented concept was to preserve the attributes related to one object together but mainly to transfer to the object the operations that are performed with these attributes.

Methods define the abilities and behavior of an object. Methods are designed to deal with elementary object operations and very often contain one or only a few command lines. If we need to solve a more complex problem, we simply design a method that uses the methods already defined.

Some sources state that the number of lines of code in a method should not exceed 20.

The method can:

- just execute a sequence of commands (sometimes referred to as a **procedure**)
- execute a sequence of commands and return a result (referred to as a **function**)

📝 2.1.2

How do the methods work?

- the method executes a sequence of commands and can end or return a result
- the method executes a sequence of commands and ends
- the method executes a sequence of commands and returns the result

📖 2.1.3

We declare a method in a class body, the method must have its type, name, optional parameters in brackets, and the commands enclosed in {}:

```
class MyClass {
   type1 atrribute1;
   type2 attribute2;
   type3 nameOfMethod1() {
      // method code
```

```
    }
    type4 methodWithParameters(type1 par1, type2 par2) {
       // method code
    }
}
```

If the method has no parameters, we give it only the name and the empty pair of brackets "()".

For a method that only executes commands and returns no value, we specify **void** as the type. *Example:*

```
class Tree {
    String type;
    int age;
    double height;

    void growOld() {
      /* code for increasing the age by one year */
    }

    void growUp() {
      /* code for changing height, for example up by 10 cm */
    }
}
```

2.1.4

Which parameters are required to define a method?

- return value type
- method name
- parameter or more parameters
- void

2.1.5

If we use attributes inside a class, we refer to them by their names, we do not use a selector because the attributes are part of the method and because we cannot refer to an instance of the class because there is none yet - we write the class code to create the instance.

Fill the code for the ***Tree*** class where the *growOld()* method is called, the age is increased by 1, where the *growUp()* method is called, the height is increased by 0.1 m.

```
class Tree {
   String type;
   int age;
   double height;

   void growOld() {
     age++;
   }

   void growUp() {
     height = height + 0.1;
   }
}
```

2.1.6

Fill the Car class code where *accelerate()* method increments the speed by 10 and in the *stop()* method sets the speed to 0.

```
class Car _____
   String brand;
   int speed;

   _____ accelerate _____ {
      speed = speed + _____;
   }

   _____ stop() {
      speed = 0;
   }
}
```

2.1.7

Class instances are created by the **new** command, attributes are referenced by the selector "**.**" and the methods are called by the selector "**.**" too. *Example:*

```
class Tree {
   String type;
   int age;
```

```
   double height;

   void growOld() {
     age++;
   }

   void growUp() {
     height = height + 0.1;
   }

   void listing() { // add a method to listing the current
state of the instance


     System.out.println(type+ ", " + age +" old, height" +
height + "m ");
   }
}
```

We create instances from the *Tree* class and change the attributes of the methods by calling the methods multiple times.

```
public class App {
   public static void main(String[] args) {
      Tree pear = new Tree();
      Tree pine = new Tree();
      pear.growOld();
      pear.growUp();
      pear.listing();

      pine.type = "coniferous”;
      pine.growUp();
      pine.growUp();
      pine.listing();
   }
}
```

2.1.8

Fill in the code that uses *Car* methods to change attributes and listing.

```
public class App {
   public static void main(String[] args) {
      Car car1 = new _____();
```

```
        Car peugeot = new _____();
        car1_____accelerate();
        car1_____accelerate();
        peugeot_____accelerate();
        peugeot_____stop();
        car1_____listing_____;
    }
}
```

📖 2.1.9

In the method code, we can call another existing method of that class. For example, in the case of a tree, it is logical if the tree changes its height as age increases.

```
class Tree {
    ...

    void growOld() {
      age++;
      growUp();
    }

    void growUp() {
      height = height + 0.1;
    }
}
```

The method is called by its name followed by a pair of brackets. We don't use a selector here like referring to attributes in a class.

2.1.10

Fill the *Employee* class code to automatically increase salary by 10 EUR when increasing the period of employment. Secure the salary increase by the already defined method.

```
class Employee {
    int _____;
    int periodOfEmployment;

    _____ incSalary() {
      salary = salary + _____;
```

```
    }

    void incPeriodOfEmployment() {
      periodOfEmployment++;
      _____;
    }
}
```

2.2 Methods (programs)

⌨ 2.2.1 Greetings

Create a **Hello** class that will have a **sayHello()** method. Make sure that the **sayHello()** method prints the text

```
Hello, my friend
```

An illustration of creating instances and calling methods is available in the **MainApp.java** file.

MainApp.java

```
public class MainApp {

    public static void main() {
        // this is just an example usage, this code is not
executed
        Hello a = new Hello();
        a.sayHello();                /* prints "Hello, my
friend" */
    }
}
```

⌨ 2.2.2 Greetings text

Design a **Greeting** class that will have:

- a **txt** attribute that will contain the greeting method, e.g. "Hi", "Greetings", "Hello", etc.,
- an **output()** method that outputs the given greeting and adds an exclamation mark after it.

An illustration of creating instances and calling methods is available in the **MainApp.java** file.

For parameters:

```
txt = "Hello"
```

will be the result of the **output()** method:

```
Hello!
```

Greeting.java

```
public class Greeting {
  // create a txt attribute that will contain the greeting
method, e.g. "Hi", "Greetings", "Hello", etc.,
  // an output() method that outputs the given greeting and
adds an exclamation mark after it.

}
```

MainApp.java

```
public class MainApp {

    public static void main() {
        // this is just an example usage, this code is not
executed
        Greeting a = new Greeting();
        a.txt = "Hi";        // sets text for greeting
        a.output();                 /* prints "Hi!" */
    }
}
```

⌨ 2.2.3 Framing of the greeting

Design the **BorderedGreeting** class.

- Let the class contain a txt attribute that will contain the text of the greeting, e.g. "Hi", "Greetings", "Hello", etc.,
- Add create a symbol attribute containing the character with which the greeting will be framed, use the char data type.
- Create an output method that prints the given greeting, adds an exclamation mark after it, and frames it.

An illustration of creating instances and calling methods is available in the **MainApp.java** file.

For parameters:

```
txt = "Good day", symbol = "x"
```

will result:

```
xxxxxxxxxxx
xGood day!x
xxxxxxxxxxx
```

BorderedGreeting.java

```
public class BorderedGreeting {
  // create a txt attribute that contains the text of the
greeting, e.g. "Hi", "Greetings", "Hello", etc.,
  // create a symbol attribute containing the character that
will frame the greeting, use the char data type
  // create an output method that prints the given greeting,
adds an exclamation mark after it and frames it

}
```

MainApp.java

```
public class MainApp {

    public static void main() {
        // this is just an example usage, this code is not
executed
        BorderedGreeting a = new BorderedGreeting();
        a.txt = "Cau";       // sets text for greeting
        a.symbol = 'x';      // sets the border char
        a.output();                /* vypise "xxxxx
                                          xHi!x
                                          xxxxx" */
    }
}
```

⌨ 2.2.4 Car

Create a **Car** class.

- Let the class contain a **car_type** attribute, which will contain the make of the car,
- Add an integer attribute **speed** containing the current speed of the car. Let the speed be initially set to 50.
- Create a **faster()** method that will increase the car's speed by 10 when called.
- Create a **slower()** method that, when called, decreases the car's speed by 5. Make sure to check that the speed does not drop below 0.
- Create an **output()** method that outputs the current data in the form type: speed.

An illustration of creating instances and calling methods is available in the **MainApp.java** file.

MainApp.java

```
public class MainApp {

    public static void main() {
        // this is just an example usage, this code is not
executed
        Car example = new Car();
        example.car_type = "Ford";
        example.faster(); // increases speed, current speed is
60
        example.output(); // writes "Ford: 60"
        example.slower();
        example.slower();
        example.slower();
        example.output(); // writes "Ford: 45"
    }
}
```

2.3 Parameters

🕮 2.3.1

In the *Employee* class, edit the *incSalary()* method so that you can determine how much you want to raise each time you call.

```
class Employee {
   int salary;
   int jobLength;
```

```
   void  incSalary() {
     salary = salary + 10;
   }

   void incJobLength() {
     jobLength++;
   }
}
```

Parameters are used to transfer this information to the method. A parameter is a variable that is declared in the method header and is used to transfer values to that method from its call point. The form of the parameter is as follows:

```
  void  incSalary(int amount) {
     salary = salary + amount;
  }
```

and form of calling from the *jobLength()* method, for example:

```
  void incJobLength() {
     jobLength++;
     if (jobLength == 10) incSalary(50);
     if (jobLength == 20) incSalary(100);
  }
```

The call parameter is a specific value or a variable from which its value is taken.

When running the *incSalary()* method, an integer variable *amount* is created into which the value that is part of calling the *incSalary()* method in the *incJobLength()* method is inserted.

The *amount* variable can be used as a common variable in the *incSalary()* method. However, its main purpose is to bring value from another part of the program into the method.

The variable *amount* expires upon the termination of the *incSalary()* method in which it was declared.

2.3.2

Fill the code so that the *incPerimeter()* method adds the specified decimal number to the current perimeter of the tree.

```
class Tree {
  _____ perimeter;
```

```
  void incPerimeter(_____ addition_____ {
      perimeter = perimeter + _____};
  }
}
```

📖 2.3.3

There is no limit to the number of parameters that enter the method. However, we must determine the type separately for each parameter.

For example, we will create a *Calculator* class that will have *sum()* and *product()* methods, listing the result of the sum and the product of two numbers that enter it as parameters.

```
class Calculator {

  void sum(int a, int b) {
    System.out.println(a + b);
  }

  void product(int a, int b) {
    System.out.println(a * b);
  }
}
```

Parameters are separated by a comma, in the method we can execute the calculation directly in the statement and within the class, and we do not need any attributes. Parameters in different methods can have the same names, their existence and validity end with the last statement of the method in which they are declared.

📝 2.3.4

Fill the code to subtract two integer inputs after starting and calculate the remainder.

```
class Calculator2 {

   _____ subtract(_____ x _____ int _____) {
     System.out.println(_____ - y);
    }
```

```
    _____ remainder(_____ a _____ int _____) {
      System.out.println( a _____ b);
    }
}
```

2.3.5

To use the calculator's methods in the program, we need to instantiate it and then use all of its available methods:

```
class Calculator {

    void sum(int a, int b) {
      System.out.println(a+b);
    }

    void product(int a, int b) {
      System.out.println(a * b);
    }
}

public class App {
  public static void main(String[] args) {

      Calculator myCal = new Calculator();  // create an
instance
      myCal.sum(10,20);          // prints the sum of two entered
numbers
      myCal.sum(5,8);            // prints the sum of the other
two numbers
      myCal.product(9,9);        // prints the product of
completely different entered numbers
      ...

  }
}
```

2.3.6

Fill the code so that the program creates an instance of the *Multiples* class that prints three times the number 86 and five times the number 189.

```
class Multiples {

  void multiple3(int input) {
    System.out.println(input * 3);
  }

  void multiple5(int vstup) {
    System.out.println(input * 5);
  }
}

public class App {

  public static void main(String[] args) {
     _____ cal = new _____();
     cal._____(86);
     cal_____multiple5_____189_____;
  }
}
```

📖 2.3.7

We treat text parameters in the same way as numeric parameters.

Create a *Mirror* class that in the *rotate()* method prints a mirror image of the string specified as its parameter.

The solution is relatively simple, we have already created a mirror image without using methods.

```
class Mirror {

  void rotate(String str) {
    String aux = "";
    for(int i = 0; i < str.length(); i++)
      aux = str.substring(i,i+1) + aux; // inserts the
following characters at the start
    System.out.println(aux);
  }
}
```

The *rotate()* method enters a string to rotate. We treat the variable that represents this parameter in the same way as any other. We insert a cycle into the body of the method, in which we copy the string from the end to the auxiliary variable and then write it.

2.3.8

Add the *Counter* class, which in the *count()* method detects and lists the count of digits in a string that enters it as a parameter.

```
class Counter {

  _____ count(_____ str) {
    int c = _____;
    for(int i = 0; i < str._____; i++)
      if (str._____(i) >= '0' && str._____(i) <= '9') c++;
    System.out.println( c );
  }
}
```

2.3.9

Which statements about the parameters are true?

- the parameter exists only during the run of the method in which it is declared
- the parameter is declared in the method header as well as the variable
- if we use multiple parameters in a method, each type must be defined separately
- parameters are separated by commas in the method header
- parameters are separated by commas when entering (calling a method)
- the parameter is available for the lifetime of the instance
- the parameter cannot be used in a method as a variable because it has a constant value after it is created
- parameters can only be used in void methods

2.4 Parameters (programs)

2.4.1 Calculator

Create a **Calculator** class that has methods

- **sum()** - adds two integers that enter the method as a parameter and prints the result,
- **division()** - divide two integers and print the result as 15 : 3 = 5, remainder 0. If 0 was specified as the divisor, it will print "Zero cannot be divided by",
- **digit_sum()** - prints the digit sum of the number specified as a parameter.

An illustration of creating instances and calling methods is presented by the following procedure:

```
public static void main(String[] args) {
   Calculator calc = new Calculator();
   calc.sum(2,8);        // writes 10
   calc.division(15,4); // writes 15 : 4 = 3, remainder 3
   calc.division(15,0); // writes Zero cannot be divided by
   calc.digit_sum(987); // wriites 24
}
```

2.4.2 Car

In the Car class

```
public class Car {
  String car_type;
  int speed;
  fuel_consumption;

}
```

ensure that:

- the **speed** value is set to 40 when the instance is created,
- by calling the **faster()** method, increase the current speed by 10, but make sure that it cannot be higher than 180,
- at the same time as increasing the speed by 10, increase the consumption by 0.5 litres in the **addConsumption()** method,
- the current state is output via the **output()** method in the form:

```
car_type: speed / fuel_consumption
```

The following procedure illustrates the creation of instances and method calls:

```
public class MainApp {

   public static void main() {
       Car x = new Car(); // creates the instance
       x.car_type = "Ford";
       x.faster(); // increases speed and consumption, current
speed is 40 + 10
       x.output(); // writes "Ford: 50 / 0.5
       x.faster();
       x.output(); // writes "Ford: 60 / 1
       x.addConsumption(); // adds only consumption
       x.output(); // writes "Ford: 60 / 1.5
   }
}
```

2.4.3 Robot

Design a **Robot** class to represent a robot moving on a 10x10 chessboard in four directions. The robot will remember its position in the **x** and **y** attributes.

- Set the starting position to [4,4].
- Implement the methods **left(), right(), up(), down()**, which will move the robot one step in a given direction each time. If the robot tries to exit the checkerboard, output the text "BOOM" and do not allow it to change position.
- Add an **output()** method that outputs the robot's position in the form: [x,y]

The following procedure illustrates the creation of instances and method calls:

```
public class MainApp {
 public static void main(String[] args) {
   Robot r = new Robot();
   r.output(); // writes [4,4]
   r.left();
   r.left();
   r.up();
   r.up();
   r.output(); // writes [2,6]
   r.left();
   r.left();
   r.left(); // writes BOOM
   r.left(); // writes BOOM
 }
}
```

⌨ 2.4.4 Pyramid

Design a **Pyramid** class that will be able to render a pyramid with the specified number of **floors** based on the floor attribute using the * character.

E.g. for floors = 3 it will be of the form:

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

In addition to rendering the pyramid in the standard form using the **classis()** method, add a **reversed()** method that will reverse the pyramid so that the top row contains the most elements and the last row contains one.

Add a **river()** method that renders the pyramid and draws a river below the pyramid made of "=" characters with the length of the longest line of the pyramid.

```
  *
 ***
*****
=====
```

The following procedure illustrates the creation of instances and method calls:

```
public static void main(String[] args) {
    Pyramid p = new Pyramid();
    p.floors = 3;
    p.classic(); // writes:
    /*  *
       ***
      *****
    */
    p.reversed(); // writes:
    /*  *****
         ***
          *
     */
    p.river(); // writes:
    /*    *
         ***
        *****
        =====
     */
}
```

⌨ 2.4.5 TextProcesor

Create a **TextProcessor** class, which will work with the text entered as a parameter of each method and will have methods:

- **getLength()** - prints the number of characters of the specified string in the form

```
The string "xyz" has 3 characters
The string "xyzwer" has 6 characters
The string "x" has 1 character
The string "" has 0 characters
```

- **getDigits()** - prints digits from the text by omitting all characters that are not digits
- **getDigitCount()** - prints the number of digits from the text
- **getSumDigits()** - prints the sum of digits from the text
- **getNumbers()** - prints a comma-separated list of numbers from the text (a number is a sequence of digits with no other character in between), e.g. for ma12m3am456aemu7 returns

```
12,3,456,7
```

Make sure that the class has a mechanism to count how many operations (method calls) its instance has performed and use the **getOperationsCount()** method to list the number of operations or method calls listed above.

The following procedure illustrates the creation of instances and method calls:

```
public static void main(String[] args) {
    TextProcesor tp = new TextProcesor();

    tp.getLength("What is the length of this string?");
// writes: The string "What is the length of this string?" has
34 characters
    tp.getDigits("It is necessary 1 to write out 23 digits out
of 123 of this text."); // writes: 123123
    tp.getDigitCount("How many 123 digits is 1 in this 2
text?");// writes: 5
    tp.getSumDigits("What 1 is the 23 sum of 1 of these
digits?");// writes: 7
    tp.getNumbers("List 12 of these 22 numbers. 87");//
writes: 12, 22, 87
    tp.getOperationCount(); // writes the number of operations
performed in this instance: 5
}
```

2.4.6 Square

Create a **Square** class that contains:

- a method to calculate the content of **getContent()**, with an integer parameter, which calculates and prints the result in the form: the content of a square with side 4 is 16.
- a method to calculate the perimeter **getCircumference()**, with an integer parameter, which calculates and prints the result in the form: The perimeter of a square with side 4 is 16.
- method to draw a full square **drawFull()**, which for the given parameter draws a full square and under the drawn image prints (calculates) its content using an existing method
- method for drawing an empty square **drawEmpty()**, which for the given parameter draws the outline of the square and under the drawn image prints (calculates) its perimeter using an existing method.

An illustration of creating instances and calling methods is presented in the following procedure:

```
public static void main(String[] args) {
    Square s = new Square();
    s.getContent(10); // writes: The content of a square with
side 10 is 100.
    s.getCircumference(5); // writes: The perimeter of a
square with side 5 is 20.
    s.drawFull(5); // writes:
    /* xxxxx
       xxxxx
       xxxxx
       xxxxx
       xxxxx
       The content of a square with side 5 is 25.
     */
    s.drawEmpty(5);// writes:
    /* xxxxx
       x   x
       x   x
       x   x
       xxxxx
       The perimeter of a square with side 5 is 20.
     */
```

```
}
```

2.4.7 Drawer

Create a **Drawer** class that will draw different shapes.

Based on the boolean parameter, ensure that a full (if true) or empty shape (false) is drawn. The individual methods will have the first parameter to indicate whether the shape is full or empty, followed by the other necessary integer parameters

- **square()** draws a full or empty square with the specified character - let the character be of type **String**,
- **diagonal()** draws a diagonal from left to right, in case of an empty shape draws a full square with the diagonal characters omitted, use "x" as the draw character.
- **rectangle()** renders a rectangle, where the first integer parameter represents the number of rows, the second the number of columns, use "x" as the rendering character.
- **triangle()** draws a right-angled isosceles triangle in the form e.g. for 4 characters

```
xxxx
xxx
xx
x
```

for empty

```
xxxx
x x
xx
x
```

use "x" as the rendering character.

The following procedure illustrates how to create instances and call methods:

```
public static void main(String[] args) {
    Drawer d = new Drawer();
    d.square(true,"a",5);
    /* writes:
        aaaaa
        aaaaa
        aaaaa
        aaaaa
        aaaaa
```

```
     */
    d.diagonal(true,4);
    /* writes:
        x
         x
          x
           x
     */
    d.diagonal(false,4);
    /* writes:
         xxx
        x xx
        xx x
        xxx
     */
    d.rectangle(true,2,3);
    /* writes:
        xxx
        xxx
     */
    d.triangle(false,5);
    /* writes:
        xxxxx
        x   x
        x x
        xx
        x
     */
}
```

⌨ 2.4.8 Tickets

Design a **Tickets** class listing the price of tickets to a specified destination for a specified number of users.

The class will have a single **getData()** method that will work as follows:

For a call to **getData("Madrid", 5)**, it will find or find the ticket price for **Madrid** and multiply it by 5 (5 people).

The result is printed in the form:

```
Your x tickets to Madrid cost xxx EUR.
```

If the destination is not offered, it will print:

```
Tickets to Madrid are not available.
```

The following destinations and prices are available:

```
Madrid - 500
Prague - 200
New York - 1200
Amsterdam - 400
Rome - 350
Moscow - 800
Krakow - 450
Paris - 333
Vienna - 800
London - 678
```

The following procedure illustrates how to create instances and call methods:

```
public static void main(String[] args) {
    Tickets t = new Tickets();
    t.getData("Moscow", 5); // writes: Your 5 tickets to
Madrid cost 4000 EUR.
    t.getData("Paris", 3);  // vypíše: Your 3 tickets to
Madrid cost 999 EUR.
    t.getData("Viena", 19); // vypíše: Your 19 tickets to
Madrid cost 15200 EUR.
    t.getData("Rio", 2);    // vypíše: Tickets to Rio are not
available.
}
```

2.5 Functions

🕮 2.5.1

Much more often than the extract from the method itself, we demand a return of the result that we can work with. The method that returns the result is often referred to as a **function** and must have a data type defined for the return value, just as with variables.

We generally define a function as a method, but it is necessary to return a value that we consider to be the result of the return statement.

```
typ methodName(typParameter p1,....) {
   command1;
```

```
   ...
   return result;
}
```

The result must be of the same type as the method type. For example: for an integer method, a text string (of type *String*) cannot be returned as a result.

2.5.2

We call a method that returns a result as a result of its activity as:

- function
- procedure
- algorithm

2.5.3

The simplest example is a method that returns the absolute value of an integer.

The input to the method is an integer parameter whose absolute value we want to find out, the output will be an absolute value. As we work with integers, the function will naturally result in an integer.

Method definition:

```
int abs(int x) {
}
```

where, as a result, we expect an integer value - the type of method, denoting the number for which we want to determine the absolute value as x.

The code will be:

```
int abs(int x) {
   if (x >= 0) result = x;      // for non-negative numbers,
the absolute value is the original number
    else result = -x;           // for negative numbers, the
absolute value is the opposite return result;              //
the result stored in the variable result is returned as a
function result
}
```

Calling a method within a class where the method is defined will look like this:

```
int b = -100;
int a = abs(b);
```

or

```
int c = abs(-58);
```

2.5.4

Fill the code in a method returning twice the entered integer value:

```
_____ twice(int a) {
   int result = _____ * a;
   _____ result;
}
```

and calling:

```
void test() {
   int m = _____(-20);
}
```

2.5.5

We do not need to use a separate variable in the method to calculate the result but we can also perform the calculation within the *return* statement - just like a statement.

```
int twice (int a) {
   return a*2;
}
```

2.5.6

Is it necessary to use variables in the body of function?

- no
- yes

2.5.7

We can use the *return* statement several times in the body of the method, everything depends on the logic we insert into the method. For example:

```
int abs(int x) {
   if (x >= 0) return x;
    else return -x;
}
```

In this way, if we have the result before all the commands in the method body are executed, we can return the result with the return statement to finish the method; no further instructions are executed.

Type a method to see if "a" is in the string.

As a result of the method we expect true/false information, so it will be of the *boolean* type. The input is a text string to be searched - type *String*.

```
boolean isThereA(String str) {
  for(int i = 0; i < str.length(); i++) {
    if (str.charAt(i) == 'a') return true;
  }
  return false;
}
```

If we find 'a' while searching for a string, we may quit because we know the answer. If the entire string is searched and the character is not found, return the function by returning *false*.

2.5.8

Fill in the code of method to see if the number you entered is 0.

```
_____ isThereZero(String number) {
   for (int i = 0; i _____ number.length(); i++) {
      if (number._____(i) == _____0_____) _____ true;
   }
   _____ _____;
}
```

📖 2.5.9

We usually expect a correct result from a return method but a problem may arise if the result is simply not.

We often examine whether a value is in the list, if so, on what position.

However, the method does not allow the return of two values or two different forms, for example, occurrence and position or that it does not occur.

Find out the position of the 'a' character in the string.

If a character is in a string, we can return its position. The problem is if the character is not found in the string. In this case, we must select a return value that can never occur when a character is found.

This value cannot be 0, because if the character is in the 1st position (index 0) the result is just 0.

A suitable value is -1 because position -1 does not exist in the string and when evaluating the result, we can check based on this value whether the search was successful or not.

```
int isThereA(String str) {
   for(int i = 0; i < str.length(); i++) {
     if (str.charAt(i) == 'a') return i;
   }
   return -1;
}
```

and calling:

```
void test() {
   String s = "Sagarmatha";
   int index = isThereA(s);
   if (index > -1) System.out.println("Contains 'a' on
possition " + index);
    else System.out.println("Not contains 'a' ");
}
```

Does this not remind you of a function for a *String indexOf*?

2.5.10

Fill the code of method to find out what position of the number entered as a string is the **last** zero. Returns -1 if the specified number is not 0.

```
_____ lastZero(String number) {
   for (int i = number.length() - 1; i _____ 0; i_____) {
      if (number._____(i) == _____0_____) return _____;
   }
   return _____;
}
```

2.5.11

You can also use the *return* command in a *void* method, which usually handles the premature termination of the method - the commands after the *return* command are not executed because the *return* command leaves the method.

This use of the *return* command does not require a return value.

List the numbers in the array until it hits 0 or the end of the array.

```
void listing(int[] arr) {
   for(int i = 0; i < arr.length; i++) {
      if (arr[i] == 0) return;            // if it encounters
0, it stops executing
                                           // other commands
                                           // in the method
      System.out.println(arr[i]);          // otherwise, it
prints the element
   }
}
```

2.5.12

Can we use the *return* command in a *void* method?

- yes
- no

2.6 Functions I. (programs)

2.6.1 Addition

In the **MyClass** class, create a **sum()** method with two real parameters that returns the sum of the specified numbers rounded to three decimal places.

The following code shows the usage of the class and its method:

```
public static void main(String[] args) {
   MyClass mc = new MyClass();
   System.out.println(mc.sum(3.464654,8.2132131)); // writes:
11.678
}
```

2.6.2 Comparison

In the **MyClass** class, create a **getMax()** method with two integer parameters that returns the larger of the two values as an integer as the result of its action. If they are equal, it will return any one of them.

The following code demonstrates creating instances and calling methods:

```
public static void main(String[] args) {
   MyClass mc = new MyClass();
   System.out.println(mc.getMax(800,200)); // writes 800
}
```

2.6.3 Product 3

In the **MyClass** class, create a **product()** method with three integer parameters that return the integer product of the specified numbers rounded to tens as the result of its operation.

E.g. for inputs 5, 7, 9

the result will be 320 (315 rounded to tens)

The following code demonstrates creating instances and calling methods:

```
 public static void main(String[] args) {
    MyClass mc = new MyClass();
   System.out.println(mc.product(-1,21,33)); // writes -690
}
```

⌨ 2.6.4 Power

In the **MyClass** class, create a **power()** method with two integer parameters that take the value an as the result of its operation a^n for the specified values a, n

E.g. for inputs 2 and 3, it will be 23 = 8

If the exponent is negative, return the value 0.

The following code demonstrates creating instances and calling methods:

```
public static void main(String[] args) {
   MyClass mc = new MyClass();
   System.out.println(mc.power(2,5)); // writes 32
   System.out.println(mc.power(2,-5)); // writes 0
}
```

⌨ 2.6.5 Divisible by 5

In the **MyClass** class, create a **get5()** method with an integer parameter that returns a list of numbers less than or equal to the specified parameter that are divisible by 5.

Let the numbers are separated by commas and the last one be followed by a period. If a value of 0 is entered, print a dash - "-".

Creating instances and calling methods is presented by the following code:

```
public static void main(String[] args) {
   MyClass mc = new MyClass();
   System.out.println(mc.get5(21)); // writes: 5,10,15,20.
}
```

⌨ 2.6.6 Addition and Product

In the **MyClass** class, create a **process()** method with two integer parameters that returns the sum and product of the specified parameters as the result of its operation.

Return the results placed in rows below each other - the first row will contain the sum, the second the product.

Creating instances and calling methods is presented by the following code:

```
public static void main(String[] args) {
   MyClass mc = new MyClass();
   System.out.println(mc.process(21,31));
   /* writes:
      52
      651 */
}
```

2.6.7 Row and column

Create methods in the **MyClass** class:

- **row()** with an integer parameter, which will return a text string as the result of its action, containing the specified number of "o" characters stored in the row after the
- **column()** with an integer parameter, which will return a text string containing the specified number of "o" characters stored below it.

Creating instances and calling methods is presented by the following code:

```
public static void main(String[] args) {
   MyClass mc = new MyClass();
   System.out.println(mc.row(6)); // writes: oooooo
   System.out.println(mc.column(4));
   /* writes:
      o
      o
      o
      o */
}
```

2.6.8 Sequence

In the **MyClass** class, create a **stars()** method with an integer parameter that will return a text string with one "x" character in the first line, two in the second, three in the third, etc. up to the nth line with n elements.

The following code demonstrates the creation of instances and method calls:

```
public static void main(String[] args) {
   MyClass mc = new MyClass();
   System.out.println(mc.starts(6));
```

```
    /* writes:
       x
       xx
       xxx
       xxxx
       xxxxx
       xxxxxx */
}
```

2.7 Functions II. (programs)

⌨ 2.7.1 Maximum in array

In the **MyClass** class, create a **getMax()** method with a parameter of a type integer array that returns the largest element of the array as the result of its action.

The following procedure illustrates how to create instances and call methods:

```
public static void main(String[] args) {
    MyClass mc = new MyClass();
    int[] input_array = {2,9,4,6,12,0,7,8};
    System.out.println(mc.getMax(input_array));  // writes 12
}
```

⌨ 2.7.2 Sum and average of elements in the array

In the **MyClass** class, create a **getData()** method with a parameter of a type integer array that returns the sum of the array elements and the average of the elements in the array rounded to one decimal place as the result of its operation in the form of a list below:

```
sum: 20
avg: 15.3
```

An illustration of creating instances and calling methods is presented by the following procedure:

```
public static void main(String[] args) {
    MyClass mc = new MyClass();
    int[] input_array = {2,9,4,6,12,0,7,8};
    System.out.println(mc.getData(input_array));
    /* writes:
       sum: 48
       avg: 6.0 */
```

```
}
```

2.7.3 Above-average elements

In the **MyClass** class, create a **getData()** method with a parameter of type array of real values, which returns a comma-separated list of the array's superscript elements, with a period after the last element of the array as the result of its operation.

An illustration of creating instances and calling methods is presented by the following procedure:

```
public static void main(String[] args) {
   MyClass mc = new MyClass();
   int[] input_array = {2.0,9.0,4.0,6.0,12.0,0.0,7.0,8.0};
   System.out.println(mc.getData(input_array)); // writes:
9.0,12.0.
}
```

2.7.4 Comparison of arrays

In the **MyClass** class, create a **compareData()** method with two parameters of the integer array type that returns a truth value indicating whether the arrays contain the same elements. The elements can be placed in different positions.

For arrays:

```
1,2,5
5,2,1
```

returns the result **true**

For arrays:

```
1,2,5,0,3
1,2,5
```

returns **false**.

The following procedure illustrates how to create instances and call methods:

```
public static void main(String[] args) {
   MyClass mc = new MyClass();
   int[] input_array1 = {2,9,4,6,12,0,7,8};
   int[] input_array2 = {2,9,12,4,6,0,7,8};
```

```
   int[] input_array3 = {2,9,12,4,6,0};
   System.out.println(mc.compareData(input_array1,
input_array2)); // writes true
   System.out.println(mc.compareData(input_array2,
input_array3)); // writes false
}
```

⌨ 2.7.5 Arranged array?

In the **MyClass** class, create a **getInfo()** method with a parameter of type integer array, which returns a truth value as the result of its action, indicating whether the elements of the array are arranged in ascending order.

For array:

```
1,2,5
```

returns the result **true**

For array:

```
1,2,5,0,3
```

returns **false**.

The following procedure illustrates the creation of instances and method calls:

```
public static void main(String[] args) {
   MyClass mc = new MyClass();
   int[] input_array1 = {2,9,14,26,112,200,371,805};
   int[] input_array2 = {2,9,12,4,6,0,7,8};
   System.out.println(mc.getInfo(input_array1)); // writes
true
   System.out.println(mc.getInfo(input_array2)); // writes
false
}
```

⌨ 2.7.6 Interval?

In the **MyClass** class, create a **compareData()** method with three parameters:

- the first parameter will be an integer array

- the next two parameters will represent the bounds of the closed interval (it is not said whether the first value must be greater than the second)

Determine if all elements of the array lie within the interval defined by the second and third parameters of the method. The result of the method is **true** if they lie, the result of the method is **false** if even one element of the array lies outside the interval.

E.g., for:

```
[2,8,9,7,3], 1, 10
```

is the result:

```
true
```

The following procedure illustrates the creation of instances and method calls:

```
public static void main(String[] args) {
   MyClass mc = new MyClass();
   int[] input_array = {2,9,14,26,10,2,1,8};
   int[] input_array2 = {2,9,12,4,6,0,7,8};
   System.out.println(mc.compareData(input_array1,1,30)); //
writes true
   System.out.println(mc.compareData(input_array2,15,1)); //
writes false
}
```

2.7.7 Position of the longest

In the **MyClass** class, create a **getMax()** method with a parameter of type array of strings, which returns the index of the longest element in the array of strings as the result of its action which returns the index of the longest element in the array of strings.

If the array contains multiple equal-length elements with the longest length, return the index of the one that is earlier in the sequence.

Assume that the array of strings is always non-empty.

E.g., for an array:

```
[Python, Pascal, Java, C, C++]
```

will output

```
0
```

An illustration of creating instances and calling methods is presented by the following procedure:

```
public static void main(String[] args) {
   MyClass mc = new MyClass();
   String[] firstArray = {"Python", "Pascal", "Java",
"C","C++"};
   System.out.println(mc.getMax(firstArray)); // writes 0
}
```

⌨ 2.7.8 Process the array

In the MyClass class, create a process() method with two parameters:

- the first parameter will be an array of integers
- the second is an integer value

Make sure that the process() method returns an array as the result of its operation, which will have the value sent to the method as the second parameter set instead of the negative elements. The contents of the original array must remain unchanged. Do not print the result.

For example, for input:

```
[1,-2,3,5,-6],10
```

the output will be

```
[1,10,3,5,10]
```

An illustration of creating instances and calling methods is presented by the following procedure:

```
public static void main(String[] args) {
```

```
   public static void main(String[] args) {
   MyClass mc = new MyClass();
   int[] arr1 = {1,-2,3,5,-6};
   int[] arr2 = mc.process(myArray,99));
   /*
     in arr1 will be saved [1,-2,3,5,-6]
     in arr2 will be saved [1,99,3,5,99]
    */
}
```

Encapsulation

Chapter 3

3.1 Encapsulation principle

🕮 3.1.1

We refer to variables that we use in the body of a method as **local**. They are created by running the method and disappear after its last statement or after the *return* command is executed.

The variables that are available across all methods in a class are actually attributes - sometimes referred to as **class variables.**

When creating classes, we try to be independent of other classes and variables that do not belong to them.

3.1.2

What do we refer to as variables that exist only during the execution of a method in that class?

- local
- global
- class variables
- static

🕮 3.1.3

When creating a class, we insert logic into the code that copies the real world. For example, the *Employee* class describes an employee and the functionalities necessary to ensure the operations that apply to it.

```
class Employee {
   String name;
   String function;
   double salary;
   int age;
}
```

We don't need any methods to get started, just set up the employee's attributes in the main program body - everything is fine:

```
Employee emp1 = new Employee();
emp1.name = "Adam";
emp1.salary = 1000;
```

```
emp1.age = 33;
```

If we perform the following assignment in the code

```
emp1.age = -28;
```

externally, everything is fine. Logic says age is a positive number, so entering a negative value can cause problems - for example, when calculating the average age of employees.

A similar problem may arise in an employee's salary variable, which may fall below the minimum wage or even to negative values by inappropriate operations.

3.1.4

Select operations that conflict with the real world image:

- int salary = -840;
- String surname = "x";
- int age = -11;
- int age = 245;
- int salary = 100;
- int age = 7;
- int age = 0;
- String surname = "Li";
- String surname = "Smith";

3.1.5

If the attribute value is set to the correct range, the application is seamless. Otherwise, an error occurs. For example, if we reduce wage payments by a loan that an employee repays, the operations are in order until the salary falls below a certain threshold due to the repayment of several loans.

Sometimes the programmer may not realize such situations, the user will apply the operation automatically and the data will become a form that no longer reflects reality.

In order to prevent such situations, object-oriented programming uses **methods** to modify the attributes. They are usually very simple but they **contain a check** to ensure that unrealistic values do not get into the attributes.

For example, the pay cut method:

```
class Employee {
   final int minSalary = 500;
   String name;
   String funnction;
   double salary;
   int age;

void reduceSalary(int amount) {
   if (salary - amount > minSalary) salary = salary - amount;
    else {
          salary = minSalary;
          System.out.println("The salary can no longer be
reduced. I set employee's salary to a minimum" + minSalary);
   }
}
```

We define a constant for setting the minimum possible salary and do not allow the current salary to be lowered to the amount below the *minSalary*. If the difference between the current salary and the reduction is less than the defined minimum, I will set the salary to a minimum and inform the user that the salary cannot be reduced.

The use of the class may take the following form:

```
...
Employee emp1 = new Employee();
emp1.salary = 1000;
emp1.name = "Adam";
emp1.reduceSalary(300);
emp1.reduceSalary(500);
```

3.1.6

Fill in the code so that it does not allow the device price to fall below 100€.

```
_____ Device {
   String title;
   double price;

   void useDiscount(double discount) {
      if (_____ - _____ >= 100) price = price _____ discount;
       else {
             System.out.println("Discount is too big, set
price to 100€");
             price = _____;
```

```
        }
    }
}
```

🕮 3.1.7

We should always use methods for responsible attribute setup at the outset, i.e. with a check on the admissibility of values, not only if we perform an operation that corresponds to the real-world operation.

Again, an example employee:

```
class Employee {
   String name;
   int age;
}
```

and creating an instance:

```
Employee emp1 = new Employee();
emp1.name = "Adam";
emp1.age = -19;
```

Now we can assign a value to an attribute at any time, and we can insert any value that matches the data type used - which is not a good solution.

To ensure that values are checked, it is necessary to ensure that the attributes cannot be changed directly, but only through methods that check whether the permissible values are assigned.

To meet this requirement, Java defines access levels for both attributes and methods. Each attribute and each method can have three access levels:

- **public** – is available from anywhere within the package/app,
- **private** – available only from that class the attribute is created, so it can only be read and changed from the methods of that class,
- **protected** –available from the class the attribute is created and its descendants (who are inherited).

The access level is defined by the keyword preceding the type of the variable used.

```
class Employee {
   private String name;
   private int age;

   public void changeAge(int a) {
```

```
      if (a > 0) age  = a;
   }
}
```

If no type of availability is specified, it is expected that the public level is set.

3.1.8

Identify the right keywords to define attribute accessibility:

_____ - ensures the availability of methods and attributes from anywhere in the program

_____ - ensure that methods and attributes are only available from the methods of the class to which they belong

3.1.9

For our current needs it is sufficient to use two levels of access:

private – usually set to all attributes, that is, they will no longer be available to read or write from anywhere in the program except the class methods they belong to.

```
public class Employee {
   private String name;
   private int age;

   public void changeAge(int a) {
      if (a > 0) age  = a;
   }
}
```

This code:

```
Employee emp1 = new Employee();
zemp1.name = "Adam";
emp1.age = -19;
```

it will not be accepted by the precompiler from the second line because the *name* attribute is private, so it is unavailable from outside the class.

public – it is usually set for all methods, except for those that do not make sense to call from outside the class (if they are used, for example, to perform partial

operations used in other class methods) as well as the class definitions themselves.

3.1.10

Select the correct accessibility level and complete the code for:

```
_____ class Device {
   _____ String title;
   _____ int price;

   _____ void setPrice(int p) {
      if (p > 0) _____ = p;
   }
}
```

3.2 Setters and getters

3.2.1

Of course, blocking the availability of attributes from the outside also has disadvantages. In addition to class methods, you cannot assign or query the contents of an attribute.

The code

```
Employee emp1 = new Employee();
...
System.out.println(emp1.name);
```

will not work because the name attribute is not accessible - its access level is set to *private*.

To define classes correctly:

- we need to define a public method for each attribute that sets its value
- we need to define a public method for each attribute that returns its value

The methods that set values are called **setters** and take the form of:

```
public void setAtribut(type a) { atribut = a; } // possibly
with the necessary controls
```

Methods that return values are called **getters** and always have a type that corresponds to the type of attribute whose value they return and take the form of:

```
public type getAtribut() { return atribut; }
```

The form names *getAttribute* and *setAttribute* are used, not mandatory.

3.2.2

Fill in the sentence.

The methods used to insert values into attributes are called _____ and have the accessibility level set to_____.

The methods used to return attribute values are called _____ and their return value must be of the same type as the attributes they return.

- public
- setters
- getters

3.2.3

For the *Employee* class, we could add individual setters and getters as follows:

```
public class Employee {
   private String name;
   private int age;

   // setters
   public void setAge(int a) {  if (a > 0) age  = a; }
   public void setName(String n) { name  = n;}

   // getters
   public int getAge() {  return age; }
   public String getName() { return name;}
}
```

In some cases of setters, it is necessary to check the contents of the set values, in others it is not.

3.2.4

Fill in the code:

```
public class Car {
   _____ String brand;
   _____ int maxSpeed;

   public _____ setMaxSpeed(int s) {
      if (s > 0) _____ = s;
   }
   public _____ getmaxSpeed() {
      _____ maxSpeed;
   }

   public _____ setBrand(String b) {
      brand = b;
   }
   public _____ getBrand() {
      _____ brand;
   }
}
```

3.2.5

After creating the *Employee* class, which is defined as follows:

```
public class Employee {
   private String name;
   private int age;

   // setters
   public void setAge(int a) {  if (a > 0) age  = a; }
   public void setNeme(String n) { name  = n;}

   // getters
   public int getAge() {  return age; }
   public String getName() { return name;}
}
```

we use the setters after creating the instance as follows:

```
Employee emp1 = new Employee();
emp1.setName("David");
```

```
emp1.setAge(28);
```

The use of getters can take the form of:

- inserting a value into variables, for example:

```
int aux = emp1.getAge();
```

- use of value in statements or calculations:

```
double average = (emp1.getAge() + emp2.getAge()) /2;
System.out.println(emp1.getName() + " is old "  +
emp1.getAge() + " years");
```

3.2.6

Fill the code to create an instance, set up, and list the attributes

```
public class Tree {
   private String title, type;
   private int height;

   public void setTitle(String ttl) { title = ttl;}
   public void setType(String tp) { type = tp;}
   public void setHeight(int h) { if (h>0) height = h;}

   public String getTitle() { return title;}
   public String getType() { return type;}
   public int getHeight() { return height;}
}
```

```
Tree tr1 = new Tree();
tr1.setType("deciduous");
tr1.setTitle("linden");
tr1.setHeight(50)
int x = tr1._____();
String ttl = tr1._____Type()
// use variables where attributes are loaded if possible
System.out.println("My tree " +  _____.getTitle_____+" is " +
_____ + "and is" + _____} + " m height");
```

📖 3.2.7

When we write the code, we try to make sure that all the necessary attribute operations perform only the methods of that class. This makes the class work as a black box - providing us with the necessary methods without knowing how they are programmed internally - and we don't care.

In a form in which the class is independent of the surrounding classes and its attributes are accessible only from the methods of that class or through getters and setters, we say that the class is **encapsulated**. The internal activity of the instance is thus independent of the surrounding environment, which means that the class can be converted into any program and used as a separate functional unit.

If necessary, we can replace the class at any time with a more in-build performance, but externally providing the same functionality. The exchange can be done without affecting the other parts of the program - they will communicate with the new class using the same methods ...

Encapsulation is the first necessary feature of object-oriented programming and makes it easy to program programs from finished parts (classes, components, modules, etc.).

3.2.8

How do we call the properties of classes that allow them to hide their internal implementation and communicate only through public methods?

3.3 Bulk attributes setup

📖 3.3.1

While enforcing encapsulation values cannot be assigned directly to attributes but we must use setters that often check the admissibility of the inserted values.

Calling more to many setters after creating a class unnecessarily lengthens the code, e.g.:

```
Tree s = new Tree();
s.setTitle("pine");
s.setAge(12);
s.setHeight(6.5);
```

etc.

It is much easier to make all the necessary settings in one call:

```
Tree s = new Tree();
s.setValues("pine", 12, 6.5);
```

where the method in class *Tree* can have a form of the method where we check the input values:

```
class Tree {
   private String title;
   private int age;
   private double height;

   public void setValues(String ititle, int iage, double
iheight) {
      if (ititle.length() > 0) title = ititle;
      if (iage > 0) age = iage;
      if (iheight > 0) height = iheight;
   }
}
```

3.3.2

Fill in the class and method definition for the default setup of attributes:

```
class Person {
   private String name;
   private int age;
   private int weight;
   private double height;

   public void setValues(String iname, int iage, int iweight,
double iheight) {
      if (iname._____ > 0) name = iname;
      if (iage > 0) age = iage;
      if (_____ > 0) weight = _____;
      if (iheight _____ 0) height = iheight;
   }
}
```

and the call:

```
Person p1 = _____ Person();
p1._____("Michael", 15, 72, 1.75);
```

3.3.3

If we assume that, in addition to the initial setup, we will also set individual attributes using setters in the next code, it is more efficient to move the control to them and the setters from the multi-attribute method just call:

```
class Tree {
   private String title;
   private int age;
   private double height;

   public void setValue(String ititle, int iage, double
iheight) {
     setTitle(ititle);
     setAge(iage);
     setHeight(iheight);
   }

   public void setTitle(String ititle) {
     if (ititle.length() > 0) title = ititle;
   }

   public void setAge(int iage) {
     if (iage> 0) age = iage;
   }

   public void setHeight(double iheight) {
     if (iheight > 0) height = iheight;
   }
}
```

3.3.4

Use the setters to set the values of each attribute in the method *setValues()*:

```
class Person {
   private String name;
   private int weight;
   private double height;

   public void setValues(String iname, int iweight, double
iheight) {
     _____(iname);
     _____(iweight);
```

```
      _____(iheight);
   }

   public _____ setName(_____ iname)   {
      if (iname.length() > 0) name = iname;
   }

   public _____ setWeight(_____ iweight)   {
      if (iweight > 0) weight = iweight;
   }

   public _____ setHeight(_____ iheight)   {
      if (iheight > 0) height = iheight;
   }
}
```

📖 3.3.5

In the previous examples we did not solve the situation when the set value did not meet the setter condition, e.g.:

```
public void setName(String iname)   {
   if (iname.length() > 0) name = iname;
}
public void setWeight(int iweight)   {
   if (iweight > 0) weight = iweight;
}
```

If the new value did not satisfy the condition (variable more than 0 characters, weights greater than 0, etc.), the attribute remained unchanged, the user was not informed that the change was not made.

Although at first glance the class looks unfinished, it is not a mistake, it is only a definition of **class behavior** that depends on the decision of the programmer or class designer.

When creating setters, we can decide that:

- **we will ignore the incorrect values** and leave the original values in the attributes, and the apparatus that will prepare the values for our instances should ensure that they are correct:

```
public void setWeight(int iweight)   {
   if (iweight> 0) weight = iweight;
}
```

- if the value is incorrect, **we set the default/base value**:

```
public void setWeight(int iweight)  {
   if (iwieght > 0) weight = iweight;
   else weight = 10;
}
```

What will be the basic value again depends on the programmer or a situation copying the real world. Here we have chosen 10 as the minimum value we are willing to accept in reality.

- if the value is incorrect, **we set a specific value to reflect that the attribute is incorrect**:

```
public void setWeight(int iweight)  {
   if (iweight > 0) weight = iweight;
   else weight = -1;
}

public void setName(String iname)  {
   if (iname.length() > 0) name = iname;
   else name = "not set";              // or name = null
}
```

A value of -1 is meaningless to the weight, and if we encounter an attribute when checking the value of an attribute, we can point out in the application that this value is problematic. Also, this applies to *String* variables too.

- if we try to enter a value that is outside the allowed range, we will **generate an exception**:

```
public void setNumber(int num) {
   if (num < 10 || num > 100) {
      throw new IllegalArgumentException();
   }
   this.number = num;
}
```

For each of these alternatives, we may choose to inform the user that the attribute setup attempt has failed, e.g.:

```
public void setWeight(int iweight)  {
   if (iweight > 0) weight = iweight;
   else {
      System.out.println("The set weight is not valid, I set
up the default value to 10" );
```

```
        weight = 10;
    }
}
```

3.3.6

Fill in the code that will check the validity of the value and warn whether the value of age is not valid or is under 18 or above 70.

```
class Person {
  ...
  private int age;

  public void setAge(int _____ {
     if (_____ <= 0) {
         System.out.println("Age contains an invalid value,
default set to 1");
         age = 1;
         _____; // ending the method
     }
     if (_____ < 18) {
         System.out.println("Age is under 18");
     }
     if (_____ > 70) {
         System.out.println("Age is above 70");
     }
     age = iage;
  }
}
```

3.3.7

Ensure that, if an incorrect parameter value is used, the appropriate values listed in the notes are set as minimum and maximum for the attributes:

```
class Person {
   private int height;  // min: 100, max: 240
   private int weight; // min: 40, max: 150

   public void setHeight(int h) {
     height = _____;
     if (h < 100) _____ = 100;
     if (h > 240) _____ = 240;
```

```
   }

   public void setWeight(int w) {
     if (w < 40) _____ = 40;
     if (w > 250) _____ = 250;
     _____ = w;
   }
}
```

🕮 3.3.8

An alert statement itself when setting an incorrect value has only an informative value and does not allow you to programmatically respond to an incorrect value.

Again, depending on the situation, we can get information on whether the setpoint has been set directly from the value-setting method - it is enough to change the return value type from void to *boolean*. In this case, however, we are no longer talking about conventional setter.

If the setting succeeds, it returns *true*, otherwise it returns *false* and this can be handled in the program (usually not handled directly in the method)

```
class Employee {
   private String name;
   private double salary;

   public boolean setSalery(double s) {
      if (s <= 0 ) return false;
      salary = s;
      return true;
  }
...
```

and usage in the class code:

```
   public void setValues(...., double s) {
      if (setSalary(s)) {
         System.out.println("Salary set on the first try");
       else
         setSalary(500);     // we set a default value
      ....
      }
   }
}
```

Again, we emphasize that this is only a variant of possible situations and it cannot be said that one approach is more appropriate than another - it always depends on the particular situation and custom of the programmer.

3.3.9

Fill in the code so that we get information about whether the value assigned to the attribute was successful:

```
class Person {
   private int height;  // min: 100, max: 240
   private int weight; // min: 40, max: 150

   public _____ setHeight(int h) {
     if (h < 100 || h > 240) _____ false;
     height = h;
     _____ true;
  }

   public _____ setWeight(int w) {
     if (w _____ 40 && w _____ 250) {
        weight = w;
        return _____;
     }
     return _____;
   }
}
```

3.3.10

Parameters that we send to setters or into any methods can also have the same name as class attributes.

```
class Person {
   ...
   int weight;

   public void setWeight(int weight)  {
   }
}
```

This choice of the name makes it easier to read the program, and we can distinguish it by using **this** keyword, which indicates that it is a class attribute or of instance:

```
class Person {
   ...
   int weight;

   public void setWeight(int weight)  {
      if (weight > 0) this.weight = weight;
   }
}
```

This is used when we need to distinguish whether it is an instance variable (attribute) or a local variable that has entered the method as a parameter. **This** indicates class/instance affiliation.

For the attribute **this** is used:

```
this.weight
```

for the parameter, it is not used and we access it without **this**:

```
weight
```

If two variables with the same name do not appear in the method and the attribute identification is unambiguous, we can omit the keyword **this** (but we can use it too).

3.3.11

Fill in the code to setup the parameters *age* and *salary*:

```
class Employee {
   private String name;
   private double salary;
   private int age;

   public void setSalary(double salary) {
     if (_____ >= 0)
        _____ = _____;
   }

   public void setAge(int age) {
     if (_____ >= 0)
```

```
        _____ = _____;
    }
}
```

3.4 Encapsulation - programs (programs)

⌨ 3.4.1 Power

Create a **Power** class that

- sets the values of its integer attribute using the **setValues()** method,
- using the **getPower3()** method to return the third power of its attribute,
- using the **getPower()** method with one integer parameter returns the specified power for the number stored in the attribute.

An illustration of creating instances and calling methods is presented by the following procedure:

```
public static void main(String[] args) {
    Power p = new Power();
    p.setValues(5); // sets the value to 5
    System.out.println(p.getPower3()) // writes 5 powered by 3
= 125
    System.out.println(p.getPower(2)) // squared of 5= 25
    System.out.println(p.getPower(4)) // writes 4 powered by 5
= 625
}
```

⌨ 3.4.2 GreetMe

Create a **GreetMe** class that will have the following available:

- a **setGreeting()** method that sets an attribute for remembering the greeting text,
- a **greet()** method, with a name parameter, that returns a greeting in the form, e.g.

```
Hello, Fero!
```

An illustration of creating instances and calling methods is presented by the following procedure:

```
public static void main(String[] args) {
    GreetMe g = new GreetMe();
    g.setGreeting("Hello"); // sets greeting to Hello
    System.out.println(g.greet("Ivan")) // writes Hello, Ivan!
    g.setGreeting("Hi"); // sets greeting to Hi
    System.out.println(g.greet("Michael")) // writes Hi,
Michael!
}
```

⌨ 3.4.3 Prism

Create a **Prism** class that has three integer attributes to hold the dimensions of the prism.

- Define a **setValues()** method that populates the triple of attributes. If any value is negative or null, set it to 1.
- Define a **getCapacity()** method that returns the volume of the prism.
- Define a **getSurface()** method that returns the surface of the prism.

An illustration of creating instances and calling methods is presented by the following procedure:

```
public static void main(String[] args) {
    Prism p1 = new Prism();
    p1.setValues(10,20,30); // sets the dimensions of the
prism to the specified values
    Prism p2 = new Prism();
    p2.setValues(10,-20,30); // sets the dimensions of the
prism to the values 10,1,30
    System.out.println(p1.getSurface()) // writes 2200
    System.out.println(p2.getCapacity()) // writes 300
}
```

⌨ 3.4.4 Tree

Create a **Tree** class that will have defined:

- integer attributes for age, height and trunk thickness

- a **setValues()** method that will populate the attributes with the specified values in the order: age, thickness, height. If any of the values are negative, set its absolute value.
- method **growOld()**, which:
- increment the age by 1,
- increase the thickness by the value obtained by calculating age * 0.5 + 1 - taking only the whole part
- increase the height by age/10 - again taking only the whole part

Define getters:

- **getAge()**, which returns age,
- **getWidth()**, which returns the thickness of the trunk,
- **getHeight()**, which returns the height of the tree
- **getInfo()**, which returns all the information in the form "Age: " + age + "\nWidth: " + thickness + "\nHeight: " + height

The following procedure illustrates the creation of instances and method calls:

```
public static void main(String[] args) {
    Tree t = new Tree();
    t.setValues(10,15,30); // sets the age to 10, thickness to
15 and height to 30
    t.growOld();
    t.growOld();
    System.out.println(t.getWidth()); // writes 28
    System.out.println(t.getInfo());
    /* writes:
        Age: 12
        Width: 28
        Height: 32   */
}
```

Constructors

Chapter 4

4.1 Constructors

📖 4.1.1

Because of encapsulation in class design, we don't allow direct access to attributes - we protect them by marking them *(private)*.

However, this does not mean that there is no need to address the default setting requirements, which are usually a prerequisite for the proper functioning of the instances created.

Up to now, we have used the following ways to enter initial values:

- assigning a default value in a variable declaration that is a simple solution, but not sufficient if initialization requires some logic (such as handling an inappropriate value)
- setters that set only one attribute at a time,
- one method of "setValue" style to which we have sent the input values and verified them inside.

4.1.2

Choose the right statements:

- setter ensures that the input value is correct
- each getter must contain a return statement
- the attribute setting you cannot use for declaration instead of a full setter
- setter does not ensure that the input value is correct
- the getter does not need to contain a return statement
- the attribute setting for the declaration can be used instead of a full setter

📖 4.1.3

We usually set the default attribute settings after instantiating, before calling other methods, such as:

```
Triangle tr1 = new Triangle();
tr1.setValue(4,3,5);
double area = tr1.getArea();
```

Sometimes can happen that the programmer omits setting values or sets the values in the wrong place and the results are then incorrect.

4.1.4

Which number will be the output of the next program?

```
class Triangle {
   int a  = 2;
   int b  = 2;
   int c  = 2;

   public void setValues(int a , int b, int c) {
     if (a > 0) this.a = a;
     if (b > 0) this.b = b;
     if (c > 0) this.c = c;
   }

   public int getPerimeter() {
     return a + b + c;
   }
}

...
Triangle tr1 = new Triangle();
tr1.setValues(3,-1,0);
System.out.println(tr1.getPerimeter());
```

4.1.5

Java is a language that assumes responsibility for minimizing programmer errors, as well as a mechanism that forces the programmer to enter initial values when creating an instance.

Java provides us with a mechanism that:

- can ensure the initialization of attributes
- it also creates an instance of the class
- allows you to run other necessary methods when creating an instance

The method of doing this is referred to as a **constructor**, and we have used it whenever we created any variable (except primitive type variables), such as:

```
int[] myArray = new int[2];                    // create an
array
```

```
FileWriter ff = new FileWriter("data.dat");  // create a
stream
Person man1 = new Person();                  // create a
person
```

When using classes, the constructor has the same name as the class itself (*FileWriter, Person*, etc.)

4.1.6

What is the method responsible for creating an instance and initializing its attributes?

- constructor
- setter
- getter

4.1.7

The constructor provides memory allocation and a new instance. Up to now, we have not created our own constructor, we have built-in, which is available in a simple form as part of each class. Such a constructor has the same name as a class and only provides memory allocation and setting of default attribute values if specified in their declaration.

For example, we used it in the following cases:

```
Person p1 = new Person();
Tree t1 = new Tree();
```

A constructor is a method that can also contain code. In Java, the method is written with parameters in parentheses after its name, and if the method has no parameters, the parentheses are empty. As with the *Person()* and *Tree()* constructors.

In case the programmer does not define his own constructor, the compiler himself creates a very simple one, which ensures only memory allocation and defines it itself - without parameters.

In case the programmer creates his own constructor, the system will not create or allow the use of the base constructor.

4.1.8

Which entries represent the *Person* class constructor?

- Person()
- setPerson()
- getPerson()
- _Person()
- setParameter(Person)
- Person.setConstructor()

4.1.9

The constructor is a method that:

- has the same name as the class name
- does not have a return value known to us
- can have any number of parameters
- is used in conjunction with the *new* command
- in code is part of the class for which it instances

If we do not define a constructor for the *Person* class, create it as follows:

```
class Person {
  ...
}
...
 Person p1 = new Person();
```

If we add for example two parameters to the constructor:

```
class Person {
   private String name;
   private int weight;

   public Person(String name, int weight) {
     this.name = name;
     if (weight > 40) this.weight = weight;
      else this.weight = 40;
  }
```

we can no longer use the previous code:

```
Person p1 = new Person();
```

but we have to use the created constructor:

```
Person p1 = new Person("Peter",56);
```

which executes the commands specified in the constructor body after reserving space for the instance,

4.1.10

Fill the code where the constructor ensures that the read values are positive.

```
class Triangle {
  private int a, b, c;

  public _____(int a, int b, int c) {
    if (a _____ 0) this.a = _____; else this.a = 1;
    if (b _____} 0) this.b = _____; else this.b = 1;
    if (c _____ 0) this.c = _____; else this.c = 1;
}
```

4.2 Constructors usage

4.2.1

As with the method, other methods can be called in the constructor. So we can use setters if they are defined in the class:

```
class Person {
   private String name;
   private int weight;
   private double height;

   public void Person(String name, int weight, double height)
{
     setName(name);
     setWeight(weight);
     setHeight(height);
   }

   public void setName(String name)      { if (name.length() >
0) this.name = name; }
```

```
   public void setWeight(int weight)     { if (weight > 0)
this.weight = weight;     }
   public void setHeight(double height) { if (heitht > 0)
this.height = height;     }
}
```

4.2.2

Fill the constructor code that effectively uses the existing method to check the value and ensure that the sides of the triangle are set to at least 1:

```
class Triangle {
  private int a, b, c;

  _____ Triangle(int a , int b, int c) {
    this.a = _____(a);
    this.b = _____(b);
    this.c = _____(c);
  }

  public _____ process(int x) {
    if (x > _____) return _____;
     else return _____;
  }
}
```

4.2.3

Let's have a *Tree* class with a constructor initializing all of its attributes:

```
class Tree {
   private String title;
   private String type;
   private int age;
   private double height;

   public Tree(String title, String type, int age, double
height) {
     if (title.length() > 0) this.title = title;
     if (type.length() > 0) this.type = type;
     if (age > 0) this.age = age;
     if (height > 0) this.height = height;
   }
```

```
}
```

In some situations, we do not need to initialize all attributes or the attributes of the instances created are set to the same values. We do not need to rewrite the code in this case but we can add a new constructor.

Each constructor must have the same name as the class its instances, so we will have **multiple constructors with the same name.**

To avoid translation problems, it is necessary that the individual constructors differ in the types, numbers or order of parameters so that the compiler has a clear indication of which constructor to call when calling the constructor.

Using multiple constructors or multiple methods with the same names that differ in number and/or type of parameters is called **overloading**.

4.2.4

Can one class have more constructors?

- yes
- no

4.2.5

Can one class have multiple methods with the same name?

- yes
- no

4.2.6

Let's add a code for a constructor that will create one-year-old apple trees of varying heights. This constructor will need a single parameter - *height*. Other attributes can be set to fixed values.

```
class Tree {
   private String title;
   private String type;
   private int age;
   private double height;
```

```
   public Tree(double height) {
     if (height > 0) this.height = height;
     title = "apple";
     type = "deciduous";
     age = 1;
   }

   public Tree(String title, String type, int age, double
height) {
     if (title.length() > 0) this.title = title;
     if (type.length() > 0) this.type = type;
     if (age > 0) this.age = age;
     if (height > 0) this.height = height;
   }
}
```

Now we can choose which constructor to use when creating instances. If we create a one-year-old apple tree we use a constructor with one parameter, constructor with all parameters we use at another tree type:

```
Tree a1 = new Tree(0.82);
Tree a2 = new Tree(1.1);
Tree p1 = new Tree("pine", "coniferous", 3, 2.1);
```

4.2.7

Fill the code for the two constructors that make up the person: the child and the adult.

```
class Person {
   private String name;
   private String education;
   private double salary;

   public _____(String name, String education, double salary)
{ // adult
     this.name = name;
     this.education = education;
     this.salary = salary;
   }

   public _____}(String name) {
// child
```

```
      this.name = _____;
      this._____ = "none";
      this._____ = 0;
   }
}
```

📖 4.2.8

Constructor and methods communication rules :

- methods are equal
- a method of that class can call any other method in an instance of that class
- the constructor can call other methods as well as other constructors of that class
- the method can call the constructor only through the *new* command

An important piece of information for us is the possibility to combine a new and existing constructor. This allows us to assign values to attributes in one place in the class.

For example, in a constructor of a *Tree* that has only one parameter, all we need to do is call a "complete" constructor with the parameters placed in the correct positions:

```
class Tree {
   private String title;
   private String type;
   private int age;
   private double height;

   public Tree (String title, String type, int age, double
height) {
      if (title.length() > 0) this.title = title;
      if (type.length() > 0) this.type = type;
      if (age > 0) this.age = age;
      if (height > 0) this.height = height;
   }

   public Tree(double height) {
      this("apple","deciduous",1 , height); // constructor call
   }
```

instead of this code:

```
   public Tree(double height) {
```

```
      if (height > 0) this.height = height;
      title = "apple";
      type = "deciduous";
      age = 1;
   }
```

To call the constructor we used *this* keyword to represent the instance (and thus its constructor). We know it from the membership of attributes to an instance.

An important rule is that there must be **no other command in the constructor before calling the constructor**. Since the constructor creates an instance, it is logical that no constructor must be preceded by any command.

4.2.9

Fill the code to call the constructor in the constructor to create a child-type person:

```
class Person {
   private String name;
   private String education;
   private double salary;

   public _____(String name, String education, double salary)
{   // adult
      this.name = name;
      this.education = education;
      this.salary = salary;
   }

   public _____(String name) {
// child
      _____(_____, "none", 0);
   }
```

calling:

```
Person p1 = new Person("Edmond", "PhD.", 2800);
Person child = new _____("Sara");
```

4.2.10

In these cases, the constructor with fewer parameters will add some default values to the non-specified parameters and call the constructor setting for all the parameters.

```
public Tree(double height) {
  this("apple","deciduous",1 , height);
}
```

The opposite approach is also possible. A multi-parameter constructor calls an existing constructor to check, set and secure the rest of the parameters in its code. For example:

```
class Tree {
   private String title = "undef.";
   private String type = "undef.";
   private int age = 1;
   private double height;

   public Tree(double height, double age) {
     if (height > 0) this.height = height;
      else this.height = 0.5;
     if (age > 0) this.age = age;
      else this.age = 1;
   }

   public Tree(String title, String type, int age, double
height) {
     this(height, age);
     this.title = title;
     this.type = type;
  }
}
```

Again, in this case, the second constructor must be called the first command.

4.2.11

Fill the code to use the existing constructor to adjust the *height* and *weight*:

```
class Person {
   private String name;
   private String education;
```

```
   private double height;
   private int weight;

   public _____(String name, String education, double height,
int weight) {
      _____(height, weight);
      this.name = name;
      this.education = education;
   }

   public _____(double height, int weight) {
      if (height >0) this.height = height;
      if (weight > 0) this.weight = weight;
   }
}
```

4.3 Constructors (programs)

⌨ 4.3.1 Calculator

Create a **Calculator** class that will return results for numerical calculations with two integers stored as attributes.

Add a **constructor** with two parameters that set both attributes. Checks are not necessary.

Add getters and setters for attributes **a** and **b**: **getA(), getB(), setA(), setB()**.

Ensure the following methods are functional:

- **sum()** - sum of attributes,
- **product()** - product of attributes,
- **difference()** - difference of attributes,
- **quotient()** - integer proportion of attributes, if the second attribute is 0, return -1 as the result,
- **modulo()** - remainder when dividing attributes, if the second attribute is 0, return -1 as result.

The following procedure illustrates the creation of instances and method calls:

```
public static void main(String[] args) {
    Calculator c = new Calculator(10,20);
    System.out.println(c.sum()); // writes 30
```

```
    c.setA(5);
    System.out.println(c.product()); // writes 100
    c.setB(3);
    System.out.println(c.getA()); // writes 5
    System.out.println(c.difference()); // writes 2
    System.out.println(c.quotient()); // writes 1
    System.out.println(c.modulo()); // writes 2
}
```

⌨ 4.3.2 Stopwatch

Create a **Stopwatch** class that will have a defined number of hours, minutes and seconds during which it runs.

Define the following methods:

- A **constructor** that populates the start state: hours, minutes, seconds in that order of parameters. If any of the values for minutes or seconds exceed 59, set them to 0; if the value for hours exceeds 23, set it to 0. If any value is negative, set it to 0.
- A second constructor, with no parameters, that sets all values to 1,
- a **tick()** method that increases the running time by 1 second; if the seconds are 60, increase the minutes by 1 and set the number of seconds to 0; if 60 minutes is reached, set the value to 0 and increase the number of hours; if the number of hours is 24, set it to 0
- **getTime(),** which returns the current state of hours, minutes, seconds in the form:

```
00:05:17
```

- i.e. each of the variables will be padded with zeros to two places and the ":" character will be inserted between the values.

The following procedure illustrates the creation of instances and method calls:

```
public static void main(String[] args) {
    Stopwatch s = new Stopwatch(10,20,30);
    for (int i = 1; i < 60; i++) s.tick();
    System.out.println(s.getTime()); // writes 10:21:29
    Stopwatch s2 = new Stopwatch();
    s2.tick();
    System.out.println(s2.getTime()); // writes 01:01:02
}
```

⌨ 4.3.3 Label

Create a **Label** class that returns a name label for the given first name, last name, and title before the name stored in the attributes:

Create two constructors:

- **A constructor with two parameters** sets the first and last name. If either value is empty, it will set it to "Anonymous", the attribute for the title will be set to empty.
- **A constructor with three parameters** will additionally contain a title in the first position, which if empty will get the value "titled".

Add getter and setter for title: **setTitle()** and **getTitle()**.

Add a **getLabel()** method that will return a title of the form title + dot + space + name + surname based on the attributes.

The following procedure illustrates the creation of instances and method calls:

```
public static void main(String[] args) {
    Label l1 = new Label("James","Bond");
    System.out.println(l1.getLabel()); // writes James Bond
    Label l2 = new Label("","James","Bond");
    System.out.println(l2.getLabel()); // writes titled. James
Bond
    l2.setTitle("Mr.");
    System.out.println(l2.getLabel()); // writes Mr. James
Bond
}
```

⌨ 4.3.4 Car

Create a **Car** class that has information about:

- the make of the car
- registration number (registration number)
- speed (integer value)

Define a **constructor** that will populate all attributes with the appropriate values in the order make, license plate, speed. If:

- tag is an empty (or null) value, sets the text "Anonymous"
- The license plate number is an empty (or null) value, it sets XX-000XX
- speed is less than 0, sets 0.

Define a **faster()** method that will increase the speed by 1 km/h.

Define a **slower()** method that decrements the speed by 1 km/h. If the slower method is used at zero speed, it sets the speed value to 0.

Define a **stop()** method that sets the speed to 0 km/h.

Define a **getInfo()** method that returns all information in the form: sign +": " + regni+ ", " + speed, e.g..:

```
BMW: AA-101XX, 58
```

The following procedure illustrates the creation of instances and method calls:

```
public static void main(String[] args) {
    Car c = new Car("Renault","XX-123AB",12);
    System.out.println(c.getInfo()); // writes Renault: XX-
123AB, 12
    c.faster();c.faster();c.faster();c.faster();
    System.out.println(c.getInfo()); // writes Renault: XX-
123AB, 16
    c.slower();
    System.out.println(c.getInfo()); // writes Renault: XX-
123AB, 15
    c.stop();
    System.out.println(c.getInfo()); // writes Renault: XX-
123AB, 0
}
```

⌨ 4.3.5 Spider

Create a **Spider** class that will have defined:

- dimensions of the room where the spider lives **rows, columns**
- **x, y** position of the spider
- a character variable (char) that will be used to draw the edges of the room.

Our goal is to place the spider in the room at the specified position, e.g..:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
x                              x
x                              x
x              o               x
x                              x
x                              x
x                              x
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

where "x" is the bounding box and "o" is the spider marker

The rows and columns, as well as the position of the spider, start counting from 1, i.e. the coordinate of the upper left corner is [1,1] (the "x" character is placed there).

Define a **constructor** that populates the number of rows, the number of columns, the x, y and the delimiter character

- if the number of rows or columns is less than 5, set the small value to 10
- if x or y is on the boundary or outside the room area, set its value to 3 if x is faulty and its value to 2 if y is faulty.

Define methods **left(), right(), up(), down()** that will move the spider position by 1 position in the appropriate direction. Let the methods return "OK" if the shift is successful, if the spider tries to step on the boundary, let its position not change and return "Ouch!"

Define a **getInfo()** method that returns the current spider and room situation, see the image above.

Define a **getSpider()** method that returns the spider's position in the form: "[" + x + "," + y + "]"

Define a **getDimensions()** method that returns the dimensions of the room in the form of the number of rows + " x " + the number of columns.

The following procedure illustrates how to create instances and call methods:

```
public static void main(String[] args) {
   // creates a room with 10 rows and 20 columns with borders
"s",
```

```
   // and with spidet on [5,6]
    Spider s = new Spider(10,20,5,6,'s');
    s.setValues(10,20,5,10,'s'); // n
    String res = s.left(); // returns OK
    res = s.left(); // returns OK
    res = s.left(); // returns OK
    res = s.left(); // returns OK
    res = s.left(); // returns Au!
    System.out.println(s.getSpider()); // writes [1,6]
    System.out.println(s.getDimensions()); // writes 10x20
}
```

Class Examples

Chapter 5

5.1 Class examples

📖 5.1.1

Create an *Animal* class that will be able to say hello as the animal.

An attribute that stores the sound of the animal and the *greeting()* method will suffice to solve the task.

The constructor is enough for us to set the attribute.

```
class Animal {
  private String sound;

  public Animal(String sound) {
     this.sound = sound;
  }

  public void greeting() {
     System.out.println(sound);
  }
}
```

We create instances very easily:

```
Animal dog = new Animal("woof");
Animal cat = new Animal("meow");
```

and call the appropriate method for everyone as well:

```
dog.greeting();
cat.greeting();
```

📝 5.1.2

Fill in the code with setter and getter for the *Animal* class:

```
class Animal {
   private String sound;

   public Animal(String sound) {
      this.sound = sound;
   }

   public _____ setSound(String sound) {
```

```
        _____.sound = sound;
    }

    public _____ getSound() {
        _____ sound;
    }

    public void greeting() {
      System.out.println(sound);
    }
}
```

📖 5.1.3

Create a *Stopwatch* class that will be able to increase the time by one second (tick) and return/write the time.

The attributes will represent the time represented by seconds, minutes, and hours.

We create a class using two constructors - the first (without parameters) sets the values to 0, the second (with parameters) allows you to set a specific time.

The setters can be solved by the *setTime()* method which, after checking the suitability of the parameters for time, sets the attributes to the specified hours, minutes and seconds. This method can also be used in the constructor.

We can also replace the triple getters with a method that returns the measured time in text form.

The key method will be the *tick()* method, which shifts the time by one second and provides, minutes and hours if it is necessary.

5.1.4

Fill the code for the default parameter settings and the constructor in the *Stopwatch* class.

```
class Stopwatch {
    _____ int h, m, s;

    public _____() {
        h = _____;
        m = _____;
        s = _____;
```

```
    }
}
```

5.1.5

Fill the code for the method to set the time parameters and the second constructor in the *Stopwatch* class.

```
class Stopwatch {
...
    public _____(int h, int m, int s) {
        _____(h, m, s);
    }

    public _____ setTime(int h, int m, int s) {
        if (h >= 0 && h < 24) _____.h = h;
         else _____.h = 0;
        if (m >= 0 && m < _____) _____.m = m;
         else _____.m = 0;
        if (s >= 0 && s < _____) _____.h = h;
         else _____.h = 0;
    }
}
```

5.1.6

Fill the code for the method that returns the current time on the stopwatch:

```
public _____ getTime() {
  String txt = h + ":" + m + ":" + s;
  _____ txt;
}
```

5.1.7

Fill the code for the *tick()* method which shifts the time by one second and increases the minutes and hours when the count reaches 60.

```
 public void tick() {
    s_____;                    // increases the seconds
    if (s == 60)    {
        s = 0;                      // if you exceed 60
seconds, increase the minutes
```

```
        m++;
        if (m _____ 60) {
            m = _____;
            h_____;
            if (h == 24) h = 0;
        }
    }
}
```

5.1.8

Create a *Fraction* class in which will be possible to:

- create a fraction using the numerator and denominator,
- we can omit the addition of getters and setters for this time,
- the *returnFraction()* method which returns a fraction in text form *n/d*
- the *reduce()* method (reduce a fraction to lowest terms) which reduce the fraction by first finding the largest common divisor and then dividing the number above and below the fractional line.

5.1.9

Create a constructor for a fraction that cannot be entered in denominator 0. If the user enters 0 replace it with 1.

```
class Fraction {
   _____ int numerator;
   _____ int denominator;

   public _____(int numerator, int denominator) {
      this.numerator  = numerator;
      if (denominator _____ 0) _____ = 1;
      this.denominator = denominator;
   }
}
```

5.1.10

Fill the code for the *returnFraction()* method which returns the fraction in text form *n/d*.

```
public _____ returnFraction() {
  _____ numerator + "/" + denominator;
}
```

5.1.11

Fill the code for the *reduce()* method which reduces fraction and the *getGCD()* method (the greatest common divisor) that finds the greatest common divisor.

```
public void reduce() {
   _____ gcd = getGCD(numerator, denominator);
   numerator _____= gcd;
   denominator _____= gcd;
}

private int getGCD(int n, int d) {
   int min = c;
   if (d < min && d _____ 0) min = d;
   for(int i = _____; i > 1; i--)
     if (c _____ i == 0 && d _____ i == 0) _____ i;
   return 1;
}
```

More Classes

Chapter 6

6.1 Parameters passed by reference

📖 6.1.1

Parameters of the primitive types that enter the method always create a new variable named parameter. It can be used as a common variable in a method and will disappear along with its value when the method ends.

E.g.:

```
class Test {
   public void listing(int i) {
      i++;
      System.out.println(i);
   }

   public void start() {
      int i  = 10;
      listing(i);
      System.out.println(i)
   }
}
```

In the example calling the *start()* method prints the following values:

```
11
10
```

because the *listing()* method has changed the copy of variable "*i*". After completing the method, we returned to the original variable whose value was set to 10 in the *start()* method and unchanged.

6.1.2

How will be the following code output after calling the *start()* method?

```
class Test {
   public void output(int i) {
      i = i + 3;
      System.out.print(i);
   }

   public void start() {
      int i  = 5;
```

```
        output(i);
        System.out.print(++i);
    }
}
```

6.1.3

If we want to "get" a calculated value from a method we use a function that can return the contents of one variable of any type (i.e., an instance filled with multiple values) as a result. The method must be of the appropriate type, for example:

```
class Test {
    public int squared(int i) {
        i = i * i;
        return i;
    }

    public void start() {
        int i  = 5;
        int s = squared(i);
        System.out.print(i);
        System.out.print(s);
    }
}
```

The value of the parameter *"i"* does not change, the calculated value is stored in the variable *"s"*, where it is returned as the return value of the *squared()* method.

6.1.4

What will be the output of the following code after calling the method *start()*?

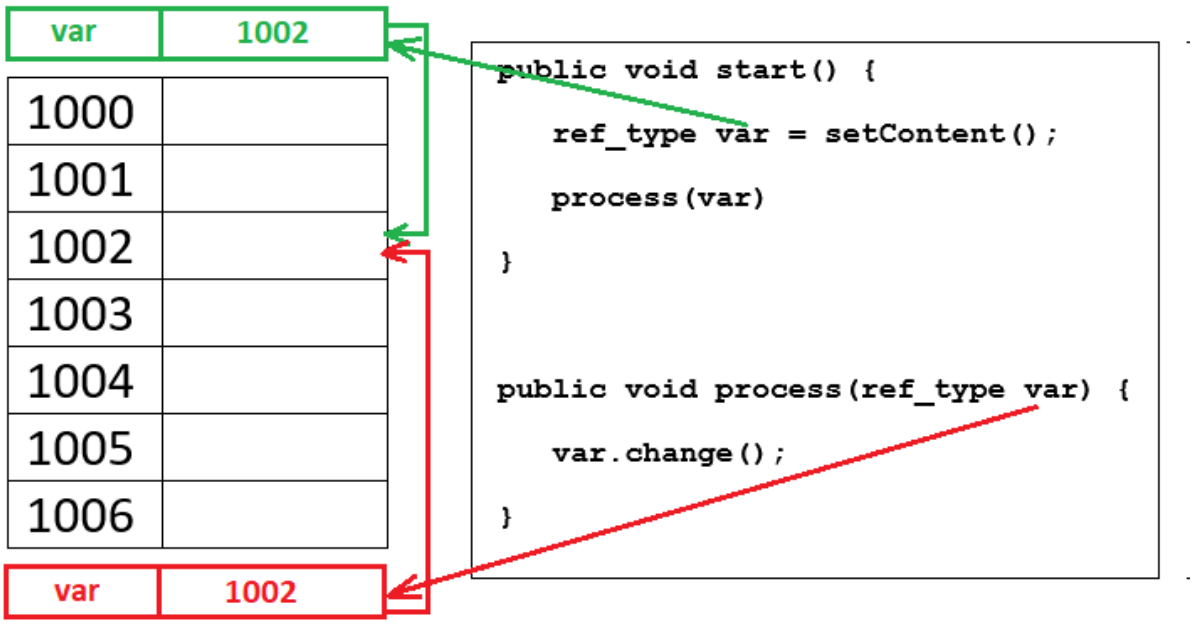
```
class Test {
    public int operation(int i, int j) {
        i = i * j;
        return  i + j;
    }

    public void start() {
        int i  = 5, j = 10;
        int m = operation(i, j);
        System.out.print(i);
        System.out.print(j);
```

```
        System.out.print(m);
    }
}
```

📖 6.1.5

If we send a variable to the method that is not a primitive but a ***reference type***, a copy of the content pointed to by the reference variable but only a copy of the reference variable - pointer, is not created when entering the method.

Changes to the content of the reference variable are then made at the same location as the variable that was sent to the method.



If we change the content of such a variable in the method, then the changes are retained even after the method is over.

A special case is the *String* type, which behaves like a primitive variable, even though it is a reference type.

6.1.6

For which group of data types will the changes in the variable's content in the method be reflected in the content of the variables involved in the method call?

- reference
- primitive
- all

- none

📖 6.1.7

Probably the simplest example of using this behavior is to manipulate the data in an array.

Create a method that sets all elements to 100 for an integer array specified as a parameter.

This is a relatively simple operation where the method input is an array with an unspecified number of elements (the method enters only a reference to the place where the array has reserved memory). The method sets each value to 100 by browsing through the elements.

```
class Test {
   public void setArray(int[] arr) {
      for(int i = 0; i < arr.length; i++)
        arr[i] = 100;
   }

   public void start() {
      int[] arr1 = new int[10];
      setArray(arr1);
      int[] arr2 = new int[6];
      setArray(arr2);
   }
}
```

We see the importance of this approach when working with multiple arrays - regardless of the name or number of array elements, all array elements are always set to the desired value.

6.1.8

Fill the code so that the elements in the array are set to the desired value.

```
class Test {
   public void setArray(int[] arr, _____ x) {
      for(int i = 0; i < arr._____; i++)
        arr[i] = _____;
   }
```

```
   public void start() {
      int[] arr1 = new int[10];
      _____(arr1, 5);
      int[] arr2 = new int[6];
      _____(arr2, 8);
   }
}
```

6.1.9

Fill the code so that the array is filled with random numbers within the specified range.

```
class Test {
   public void setArray(int[] arr, int a, int b) {
      for(int i = 0; i < arr._____; i++)
        arr[i] = (_____) ( _____ + Math._____() * (b - a +
1));
   }

   public void start() {
      int[] arr1 = new int[30];
      _____(arr1, -10, 20); // set values from -10 to 20
   }
}
```

6.2 Methods' return value

6.2.1

T: Create a method that creates a mirror image from the array specified as a parameter.

An input to a method is an array, more precisely its reference - if we change the array values its content will also change where we called the method. The exchange of elements is mirrored - the first element is exchanged with the last, the second from the penultimate, etc. up to the middle of the array.

If the array has an even number of elements, then two adjacent elements in the middle of the array will be replaced, if the number of elements is odd, the middle element will remain in place. In the loop, we proceed to the middle of the array.

```
public void mirror(int[] arr) {
   for(int i = 0; i < arr.length / 2; i++) {
     int aux = arr[arr.length - 1 - i];              // i-th
element from the end
     arr[arr.length - 1 - i] = arr[i];
     arr[i] = aux;
    }
}
```

6.2.2

Fill the method code to arrange the array from largest to smallest.

```
public void sort(int[] arr) {
   for(int i = 0; i < arr.length; i++) {
     for(int j = _____; j < arr.length - i - 1; _____++) {
         if (arr[j] < arr[_____]) {
             int aux = arr[j];
             arr[j] = arr[_____];
             arr[_____] = aux;
         }
     }
   }
}
```

6.2.3

In addition to changing the content of the reference variable, the result can be returned as well as the result of the method's work.

If we modify the previous task to use a function to solve it, the code will change only minimally.

T: Create a method that returns a mirror image for an array specified as a parameter.

The return value of the method will be an integer array.

If we do not want to change the contents of the original array, we can create a new array in which its elements are already mirrored.

```
public int[] mirror2(int[] arr) {
// reserve the same number of elements for the new array as
arr
```

```
   int arr2[] = new int[arr.length];
   for(int i = 0; i < arr.length; i++) {
      arr2[arr.length - i - 1] = arr[i]; // put the first
element in the last position, etc.
    }
    return arr2;                          // return the rotated
array as a result
}
```

and use the return value to populate the array from the call point:

```
int[] arr1 = new int[9];
setArray(arr1);                  // filled with random values
int[] myarr = mirror2(arr1);     // create a new array by
inserting the inverted array values
                                 //from the method
```

6.2.4

Fill a method to return an array containing elements with twice the original array value.

```
public _____ twice(int[] arr) {
   int arr2[] = new int[arr._____];
   for(int i = 0; i < arr._____; i++) {
     arr2[_____] = arr[_____] * _____;
   }
   _____ arr2;
}
```

and the method call:

```
_____ result = twice(arr1);
```

6.2.5

T: Create a method that detects the largest and smallest element of an array in one run.

Although there are object twins for primitive types (for *int* - *Integer*, for *double* - *Double*, etc.) they are not referencing types in the true sense. Values do not change by reference. They are usually used in places where a primitive variable cannot be used but an object variant is required.

The first solution may be to create and return an array with two elements, the first being the minimum and the second the maximum.

```
public int[] minMax(int[] arr) {
   int min =  arr[0];
   int max =  arr[0];
   for(int i = 0; i < arr.length; i++) {
     if (arr[i] < min) min = arr[i];
     if (arr[i] > max) max = arr[i];
   }
   int[] res = {min, max};                // create a result
array
   return res;                            // return the array
}
```

The call can take the form of:

```
int res[] = minMax(arr1);                // return the result
of the method to the integer array
System.out.println("min: " + res[0]);
System.out.println("max: " + res[1]);
```

This solution is only suitable if all results are of the same type.

6.2.6

Fill the method code to "output" the contents and the circuit of the circle at once.

```
public _____ circle(int c) {
  double[] result = new double[_____];
  result[0] = 3.14 * r * r;
  result[1] = 3.14 * 2 * r;
  _____ result;
}
```

the calling:

```
double res[] = circle(32);
System.out.println("circuit: " + res[_____]);
System.out.println("contents: " + res[_____]);
```

📖 6.2.7

The second solution is to create an instance of the class whose attributes (including various types) we use to:

- populating a variable (via a parameter called by a reference)
- or to return values (method type will be given class).

To output values through a variable we can proceed as follows:

We define an instance for data output:

```
class Memory {
   int min;
   int max;
}
```

and a method in which we find the minimum and maximum for their output using the reference parameter types *Memory*.

```
public void minMax(int[] arr, Memory m) {
   int min = arr[0];
   int max = arr[0];
   for(int i = 0; i < arr.length; i++) {
     if (arr[i] < min) min = arr[i];
     if (arr[i] > max) max = arr[i];
   }
   // assign values to the instance
   m.min = min;
   m.max = max;
}
```

The call will take the form of:

```
Memory m = new Memory();                    // create an instance
to transfer values
minMax(arr1,p);
System.out.println("min: " + m.min);
System.out.println("max: " + m.max);
```

where, in the first step, we create an empty instance, populate it in the *minMax()* method, and then write it through the public attributes.

If we wanted to be fair, it would be desirable to make the attributes private and to use getters and setters.

6.2.8

Fill the method code so that one call to get the average and the sum of the array elements with decimal values.

```
class Memory {
   _____ sum;
   _____ avg;
}
```

```
public void sucmAvg(double[] arr, _____ m) {
   _____ sum = 0;
   for(int i = 0; i < arr.length; i++)
      sum += arr[i];
   m.sum = sum;
   m.avg = sum/_____._____;
}
```

the calling:

```
double [] arr1 = new double[10];
setArray(arr1);
Memory m = new Memory();
sumAvg(_____, _____);
```

6.2.9

An alternative is to rewrite the method to a form in which the return value is of the *Memory* type. The result of the method is saved to a *Memory* instance.

```
class Memory {
   double sum;
   double avg;
}
```

```
public Memory minMax(int[] arr) {
   int min = arr[0];
   int max = arr[0];
```

```
   for(int i = 0; i < arr.length; i++) {
      if (arr[i] < min) min = arr[i];
      if (arr[i] > max) max = arr[i];
   }

   Memory m = new Memory();      //create an instance and
insert values into it
   m.min = min;
   m.max = max;
   return m;                     //return the populated
instance as a result of the method
}
```

and finally the call:

```
Memory m = minMax(arr1);
System.out.println("min: " + m.min);
System.out.println("max: " + m.max);
```

6.2.10

Fill the method code so that one call to get the average and the sum of the array elements with decimal values as the return value of the *sumAvg()* method.

```
class Memory {
   _____ sum;
   _____ avg;
}
```

```
class Memory {
   _____ sum;
   _____ avg;
}
```

the calling:

```
double[] arr1 = new double[10];
setArray(arr1);
_____ m = sumAvg(_____);
System.out.println("sum: " + m.sum);
System.out.println("average: " + m.avg);
```

6.3 Throw

📖 6.3.1

There are two types of errors/exceptions in Java:

- ***checked***: it is necessary to treat them in code - they include e.g. errors when working with files
- ***unchecked***: do not need to be treated and are processed after the program starts - this includes e.g. division by zero, error when converting string to a number, etc.

Java requires a *try - catch - finally* structure to catch checked exceptions.

- the ***try*** block contains a code that monitors for an error
- the ***catch*** block contains a description of the actions to take when an error occurs. We can set the same behavior for all kinds of errors or perform a different code for each identified error
- the sequence of the commands to be executed, both in the event of an error and in the absence of an error, is listed in the ***finally*** block.

```
try {
    sequence of guarded commands
} catch (Exception e) {
    error notification or error resolution code
} finally {
    block of commands to be executed in each case
}
```

This structure may or may not be used in the case of uncontrolled exceptions. If the error does not occur the code continues to execute. If the data causes an exception then the execution of the code is interrupted without *try-catch* treatment.

📝 6.3.2

Determine the correct order of the keywords used to handle exceptions.

- finally
- try
- catch

6.3.3

Select the correct statements:

- checked exceptions must be handled in the code
- unchecked exceptions need not be handled in the code
- controlled exceptions include e.g. IOException
- uncontrolled exceptions include e.g. NumberFormatException
- checked exceptions need not be handled in the code
- unchecked exceptions must be handled in code
- uncontrolled exceptions include e.g. IOException
- controlled exceptions include e.g. NumberFormatException

6.3.4

The throw statement is generated by the *throw* statement, which can be used in multiple forms.

In simple form, it provides interruption of code execution in the try block and moves control to the catch block. It does so by creating an exception, and can use:

- exception constructor without parameter, *e.g.*:

```
throw new ArithmeticException();
```

- a constructor with a text parameter that also allows a user-defined error description, e.g.:

```
throw new ArithmeticException("attempt to divide by zero");
```

Example of use:

```
try {
   int a = 3, b = 0;
   if (b == 0) throw new ArithmeticException("attempt to
divide by zero");
   int c = a/b;
   System.out.println(c);
} catch (NumberFormatException e) {
    System.out.println(e.getMessage());
} catch (ArithmeticException e) {
    System.out.println(e.getMessage());
}
```

The advantage of using the *throw* command is that you can enter your own text and catch the error before it occurs.

The generic exception generation is:

```
throw new exception_class("error message");
```

For the mechanism to work properly, the type of exception created must be caught in the *catch* branch.

6.3.5

Interrupted performance of the code by using the *throw* command if the user enters a negative value or a value over 120 for input. Ensure that the message that was used in the exception constructor is printed.

```
Scanner input = new Scanner(System.in);
try {
   int age = input.nextInt();
   if (age < 0) _____ new ArithmeticException("A negative
value for age is not allowed");
   if (age < 18) System.out.println("Child");
   if (age >= 18 && age < 70) System.out.println("Adult");
   if (age >= 70 && age < 120) System.out.println("Senior");
   if (age > 120) throw {1:SA:=new} ArithmeticException("Age
over 120 is not allowed");
} _____ (ArithmeticException e) {
    System.out.println(e._____);
}
```

6.3.6

If there is no *catch* branch for the *throw* statement, execution of the program at the exception creation point is interrupted.

Example:

```
Scanner input = new Scanner(System.in);
int age = input.nextInt();
if (age < 0) throw new ArithmeticException("Ending, the
program is not designed for age < 0");
if (age < 18) System.out.println("Child");
if (age >= 18 && vek < 70) System.out.println("Adult");
```

```
if (age >= 70 && vek < 120) System.out.println("Senior");
```

If you enter a negative value, exit the program with an error:

```
Exception in thread "main" java.lang.ArithmeticException:
Ending, the program is not designed for age < 0
```

6.3.7

Fill the program code so that the program ends with an error if an odd value is entered :

```
Scanner input = new Scanner(System.in);
System.out.println("Enter an even value!");
int data = input.nextInt();
if (data _____ 2 != 0) _____ _____ ArithmeticException("An odd
value was specified");
System.out.println("Half of the entered value is " + data
_____ 2);
```

6.3.8

Generating errors in classes will help us in particular in checking the correct range of parameters, both in the constructor and in different setters.

To prevent the programmer from using a *try-catch* block when creating a new class, you must choose one of the exception classes, which are unchecked exceptions. The ideal type is *IllegalArgumentException*.

6.3.9

In which parts of the class does the error generate an error in case of incorrect values?

- setter
- constructor
- getter
- getInfo – returning instance information

📖 6.3.10

The same *throw* statement generates an exception and aborts the execution of code in any method.

If used in the constructor (the constructor does not end with the last statement), the instance is not created - this will prevent the generation of instances with incorrect attributes.

T: Ensure that an *IllegalArgumentException exception* with a description of the problem is generated in the *Person* class constructor in case of a negative age.

```
public class Person {
   private String name;
   private int age;

    public Person(String name, int age) {
        this.name = name;
        if (age < 0) throw new IllegalArgumentException("age <
0");
        this.age = age;
    }
...
}
```

Using the following listing in your code:

```
Person p1 = new Person("Peter", 10);
```

the instance is created and the program continues.

If we try to create the following instance:

```
Person p2 = new Person("Anna", -20);
```

program execution is interrupted with the following listing:

```
Exception in thread "main" java.lang.IllegalArgumentException:
age < 0
```

6.3.11

Fill the *Employee* class code so that an exception is generated if you enter a negative salary.

```
public class Employee {
   private String name;
   private int salary;

    public Employee(String name, int salary) {
        _____.name = name;
        if (salary < 0) _____ _____
IllegalArgumentException("Attempt to assign a negative value
for salary");
        _____.salary = salary;
    }
...
}
```

📖 6.3.12

If we use an unchecked exception (*eg. IllegalArgumentException*) then we can (but do not have to) treat the place where the exception can occur with a try-catch structure. In this case, the execution of the code is not interrupted but continues after the catch block.

```
public class Person {
   private String name;
   private int age;

   public Person(String name, int age) {
      this.name = name;
      if (age < 0) throw new IllegalArgumentException("age <
0");
      this.age = age;
   }
...
}
```

with calls e.g.:

```
Person p2;
try {
   p2 = new Person("Anna", -10);                    // the
constructor generates an exception
}
catch (Exception e) {
   System.out.println(e.getMessage());
}
```

```
...
```

The same method can be used for risk setters.

6.3.13

Fill the class code to handle the situation where a negative value for the *salary* variable is sent to the *Employee* class constructor.

```
public class Employee {
   private String name;
   private int salary;

    public Employee (String name, int salary) {
        this.name = name;
        if (salary < 0) throw new
IllegalArgumentException("negative salary");
        this.salary = salary;
    }
...
}
```

```
Employee e1;
try {
   e1 = new _____("Jacob", -305);
} _____ (Exception e) {
   System.out.println(e._____);
}
```

6.4 References in fuctions (programs)

6.4.1 Same elements of arrays

In the **MyClass** class, create a **compareData()** method with two parameters of type array of strings that returns a list of elements that are in both arrays as the result of its action.

The elements can be placed in different positions and in the result let them be listed below each other in alphabetical order.

If an element is listed more than once, let it be listed more than once.

Make sure that the source arrays are untouchable when transferring their references.

For arrays:

```
mom, dad, grandma, grandpa, grandma
mama, grandpa, grandma, mama, winter
```

returns the result in the form:

```
grandma
grandpa
mama
mama
are the common elements of the arrays:
[mama,dad,grandma,grandpa,grandpa,mama]
[mama,grandpa,grandma,mama,winter]
```

An illustration of instance creation and method calls is presented by the following procedure:

```
public static void main(String[] args) {
    MyClass mc = new MyClass();
    String[] firstArray = {"mama", "dad", "grandma",
"grandpa"};
    String[] secondArray = {"mama", "grandpa", "grandma",
"winter"};
    System.out.println(mc.compareData(firstArray,secondArray))
;
    /* writes:
       grandma
       grandpa
       mama
       mama
       are the common elements of the arrays:
       [mama,dad,grandma,grandpa,mama]
       [mama,grandpa,grandma,mama,winter]
    */
}
```

⌨ 6.4.2 Student

Create a **Student** class that will have information about:

- first and last name
- the student's year
- the amount of the scholarship

Define a **constructor** with first name, last name, year and scholarship amount (integer), ensuring that the attributes are populated as follows:

- set the first and last name as they came in
- if the year is less than 1, generate an IllegalArgumentException with the text "too low year of study"
- if the year is greater than 5, generate an IllegalArgumentException with the text "too high year of study"
- If the scholarship is less than 0, generate an exception: NumberFormatException with text "negative scholarship"
- If the scholarship is less than 10000, generate an exception: NumberFormatException with the text "too expensive scholarship"

Define a **getInfo()** method that returns information about the student in the form of name, surname, year, sholarship, e.g.:

```
first name last name, 5th, scholarship EUR
```

An illustration of creating instances and calling methods is presented by the following procedure:

```
public static void main(String[] args) {
    Student s1 = new Student("John","Smith", 2, 5000);
    System.out.println(s1.getInfo()); // writes John Smith,
2., 5000 EUR
    Student s2 = new Student("John","Doe", 8, 5000); //
constructor ends with error: IllegalArgumentException: too
high year of study
    Student s3 = new Student("John","Doe", 2, -5000); //
constructor ends with error: NumberFormatException: negative
scholarship
}
```

⌨ 6.4.3 Bus

Create a **Bus** class that will have information about:

- the maximum number of passengers that can fit on the bus
- the current number of passengers,
- the state of the bus - whether it is moving or stationary, in case of movement it contains the value **true**
- door status: open/closed - if open, it contains the value **true**

Define **a constructor with one parameter** that populates the attributes as follows:

- the maximum number of passengers based on the method parameter, if less than 10, sets the value to 10.
- sets the current passenger count to 0,
- and let the bus not move and keep the doors open.

Define **a constructor with two parameters**, where the first parameter is the current number of passengers and the second is the maximum number of passengers.

- If either of the parameters is negative or the current count is greater than the maximum count, generate an exception: an IllegalArgumentException with the text "<0 or too much"
- and keep the bus stationary and the doors open.

Define a **closeDoor()** method that closes the door.

Define an **openDoor()** method that opens the door.

Define a **move()** method that moves the bus, but only if the door is closed.

Define a **stop()** method that stops the bus.

Define an **exiting()** method with an integer parameter that provided,

- that the specified parameter is less than or equal to zero does nothing,
- the door is open, let the specified number of passengers off the bus,
- if it wants to discharge more than the current number of passengers, it sets the current state to 0 and generates an IllegalArgumentException exception with the text "empty bus"

Define a **boarding()** method with an integer parameter that provided,

- the specified parameter is less than or equal to zero does nothing,
- that the door is open, adds the specified number of passengers to the current state,
- if more passengers want to board than can fit on the bus, the current state is set to full and an IllegalArgumentException is generated with the message "full bus " + count + " more" where the count is those who didn't fit on the bus.

Define a **getInfo()** method that returns information about the bus in the form of current status/maximum number of people, open/moving, e.g.:

```
10/45 true/false
```

An illustration of creating instances and calling methods is presented by the following procedure:

```
public static void main(String[] args) {
    Bus b1 = new Bus(30,20); // constructor ends with error:
IllegalArgumentException: <0 or too much
    Bus b2 = new Bus(30,40);
    b2.closeDoor();
    b2.move();
    b2.stop();
    b2.open();
    b2.boarding(10);
    System.out.println(b2.getInfo()); // writes 40/40
true/false
    b2.exiting(50); ends with error IllegalArgumentException:
empty bus
}
```

Static Variables

Chapter 7

7.1 Static variables

📖 7.1.1

T: Create a *Bottle* class to record the maximum volume, the currently filled volume, and a unique serial number. Allow adding and removing bottle content.

The class may take the form of:

```
public class Bottle {
   private int serialNum;
   private int maxVol;
   private int curVol;

   public Bottle(int maxVol, int curVol) {}
   public void addFluid(int vol) {}
   public void takeFluid(int vol) {}
   public String getInfo() {}
}
```

A new feature is the requirement to ensure the uniqueness of the serial number. The most sensible thing would be to create a mechanism that can count the bottles and assign each new number one larger than the previous one.

This information cannot be stored in an instance known to us because every new instance that is created is the same - it reads the same default values and reserves the same memory.

We could send the serial number information to the constructor as an additional parameter but we would need a global variable in the service code to count the bottles. Such a solution is possible but carries the risk of errors and encapsulation problems.

📝 7.1.2

How do we define the term encapsulation?

- class independence from surrounding code
- the ability to transfer a class to another program, regardless of code outside the body
- not using other than public attributes
- not using class attributes
- not using class method

📖 7.1.3

In order for our class to be encapsulated (and thus transferable as a whole to other applications), the serial number allocation mechanism needs to be incorporated into it.

In addition to the class instance attributes, where each instance has its own attributes, there are also class attributes that:

- they are common to and independent of, all instances of that class
- they exist before the first instance is created, they are created at program startup

These attributes are called ***static*** and use the *static* keyword to identify them. It is referred to as the visibility attribute of a variable or method, such as:

```
private static int num;
private static void load() {}
```

The following applies to the static attribute and the static method (also called class method) due to independence from any instance of the class:

- there is no need to create an instance for their use
- a static method cannot directly refer to instance methods or their attributes from a class instance, you can call
- static classes and use static attributes
- the static method can use static variables

We use static methods when writing programs from the beginning. These are all the "commands" that we use without instantiating:

- *System.out.print*
- *Integer.ParseInt*
- *Math.random*

📝 7.1.4

Which keyword is used to define static attributes and methods?

📖 7.1.5

To ensure a unique serial number:

- remember the number of bottles created so far
- assign a new number to each newly created one and increase the number

We will use the class attribute to store the serial number because it is a value that belongs only to the instance and other instances have no reason to access it.

We use a static variable that is common to all instances due to the use of the *static* keyword for the number of bottles we create. If one instance changes the contents of the variable all other instances see the updated value.

The default value that we assign to it is set when the program starts.

The modified class will be:

```
public class Bottle {
   private int serialNum;                  // instance
attribute
   private static int count = 0;           // common to all
instances
   private int maxVol;
   private int curVol;
...
}
```

The *count* attribute is private because it is not required from the outside it is only to ensure the uniqueness of the serial number and is only handled within the instance.

7.1.6

When is a static variable created and its default value set?

- when running the program
- when creating the first instance of the class to which the static variable belongs
- every time you create a new instance

📖 7.1.7

We will use the current serial number and increase it in the constructor:

```
public class Bottle {
   private int serNum;
   private static int count = 0;
   private int maxVol;
   private int curVol;

   public Bottle(int mO, int cO) {
      count++;                          // we increase the
number of created instances
      serNum = count;                   // we set the serial
number for the current instance
      if (mO > 0) maxVol = mO;
        else maxVol = 1;
      if (cO > 0) curVol = cO;
        else curVol = 0;
   }
...
}
```

Increase the number of bottles created by one in the code and set a new serial number.

We use static variables in instance methods by default because they exist since the program started.

Instance variables cannot be used in static methods because a static method can be run even if there is no instance yet and a non-existent variable would be used.

In addition, the static method for multiple instances created cannot determine which one to work with.

📝 7.1.8

For static attributes and static methods:

- there is no need to create an instance for their use
- static method cannot use instance attributes
- the static method can use static variables
- at least one instance must be created to use them
- the static method can directly refer to instance attributes

- static method cannot use static variables

7.2 Static variables usage

📖 7.2.1

Create an *Account* class that is characterized by:

- owner name
- the amount
- interest rate
- account number
- necessary methods.

Ensure:

- account numbers will automatically increase for each new account
- if the interest rate changes, interest on all accounts of that class will change
- each month, the amount on the account shall be 1/12 of the prescribed interest

We will increase account numbers in the same way as in the previous example by using the static class attribute to use as a counter. We always raise it in the constructor and assign that value as the account number to the instance.

The solution of interest rate will be a little differently. If *interest_rate* were to be defined as an instance attribute (not a static attribute), any change in interest would have to change its value in all existing instances. Let's take advantage of the fact that a static variable is shared by all instances and its change is reflected at once in all instances. If we define the *interest_rate* attribute as a static variable, it will be common to all accounts and changing it in one place will be reflected in all instances.

📝 7.2.2

Select which attributes will be static:

```
class Account {
   private ? int accountCount = 0;
```

```
   private ? double interest_rate = 0.5;
   private ? int accountNum;
   private ? String owner;
   private ? double amount;
...
}
```

- accountCount
- interest_rate
- accountNum
- owner
- amount

7.2.3

Increasing the account number and associating the current account number with the instance is provided by the constructor. Fill in the method code:

```
public Account(String owner, double amount) {

   accountCount_____;                          // increase the
number of accounts
   _____ = _____;    // set current number
   this.owner = owner;                              // set owner
name
   this.amount = amount;                            // deposit a
primary deposit
}
```

7.2.4

The interest rate is defined in a static *interest_rate* variable. It will be changed by a method that will be static for the following reasons (but not necessary):

- works with a static variable - it is customary to write methods that only work with a static variable as static,
- you should be able to change/set interest before creating the first account - the method must be static in this case so that it can be used before the first instance of the *Account* class is created.

Fill the code for the static interest rate change method:

```
class Account {
   private static int accountCount = 0;
   private static double interest_rate = 0.5;
   private int accountNum;
   private String owner;
   private double amount;
   ...
   _____ void ChangeIR(int interest_rate) {
      if (interest_rate _____ 0) _____ new
IllegalArgumentException("negative interest_rate")
      _____.interest_rate = interest_rate;
   }
}
```

7.2.5

Suppose the *interest_rate* calculation is only run once at the end of the month. The amount will be increased by 1/12 of the interest.

The method works with instance attributes, so it must not be static = it can only exist after the instance has been created.

Fill the *interest_rate* calculation code:

```
_____ void AddMonIR() {
   amount = amount + _____ * interest_rate/12;
}
```

7.3 Static variables (programs)

7.3.1 Passport

Create a **Passport** class that will handle the production of instances representing passports.

It will store the owner's name, year of birth, and the serial number of the passport.

Let the class mechanism ensure that the serial number is incremented by 1 for each new passport, and let it start at 100001.

Define a **constructor** that:

- populates the owner and its year of birth (if the owner's name is empty or the year value is less than 1900, let it set the name to "invalid" and the year of birth to -1),
- sets the following serial number.

Add a **getInfo()** method that returns passport information of the form: owner + " (" + year of birth + "): " + serial number e.g.:

```
Pear (1995): 100005
```

An illustration of creating instances and calling methods is presented by the following procedure:

```
public static void main(String[] args) {
    Pasport p1 = new Pasport("Mak",1920);
    Pasport p2 = new Pasport("Rak",1840);
    Pasport p3 = new Pasport("Drak",1998);
    System.out.println(p1.getInfo()); // writes Mak (1920):
100001
    System.out.println(p2.getInfo()); // writes invalid:
100002
    System.out.println(p3.getInfo()); // writes Drak (1998):
100003
}
```

⌨ 7.3.2 Landrover

Create a **Landrover** class that will cater for the production of expensive cars for specific customers

- It will have setters and getters defined for the attributes color, serial number (int), operating system, owner.
- Let the serial number be incremented by 1 for each new car, and let it start at 100.
- The system will work by creating new instances with the color that is currently set in the system - set the default color to "red"

Define a **constructor** that:

- populates the owner and the operating system (if any of the input values are empty, have it set the text value to "x"),
- sets the serial number and color of the car

Provide a **changeColor()** method that can change the color so that after the color is changed, all other manufactured units will in turn use that set color. The color of already produced cars is not changed.

Add a **getOrder()** method that returns information about how many in order the car was produced.

Add a **getInfo()** method that returns information about the car in the form: owner + " (" + serioveCislo + "): " + color + ", OS: " + operacnySystem, e.g..:

```
Pear (203): green, OS: Windows 3.1
```

An illustration of creating instances and calling methods is presented by the following procedure:

```
public static void main(String[] args) {
   Landrover l1 = new Landrover("El Chef","Android Oreo");
   Landrover l2 = new Landrover("El Toro","Android Kit-kat");
   l1.changeColor("pink");
   Landrover l3 = new Landrover("El Cat","Windows Mobile 10");
   System.out.println(l3.getOrder()); // writes 3
   System.out.println(l1.getinfo()); // writes El Chef (100):
red, OS: Android Oreo
   System.out.println(l2.getinfo()); // writes El Toro (101):
red, OS: Android Kit-kat
   System.out.println(l3.getinfo()); // writes El Cat (102):
pink, OS: Windows Mobile 10
}
```

Inheritance

Chapter 8

8.1 Preparation

📖 8.1.1

T: Create an *Account* class that contains information about:

- account number
- account owner
- current bank account status (balance)

Add methods that allow:

- deposit/accept an amount to account
- withdraw/pay an amount from your account
- list the last 100 operations performed
- print out the account information

Remember that banks charge for each deposit and withdrawal.

8.1.2

We'll need attributes for account number, owner name, amount and fee amount. The necessary methods will be *constructor*, *deposit* and *withdrawal*, *output* of operations information, respective return account information.

The fee will be changed in all accounts at the same time, so it is appropriate that it is a "common" variable.

In addition, we implement a mechanism for assigning account numbers using a static variable.

To store statements we will use a matrix with two columns where the first column will contain the description of the operation or account number and the second column will contain the amount. If the amount is credited to the account, it will contain a positive value, if it is debited it will contain a negative value.

Now, according to these considerations add the first definition of the class:

```
public class Account {
   private int AccNum;
   private String Owner;
```

```
   private double accStatus;
   private _____ operationOutput;
   private _____ int accountCount;
   private _____ double fee = 0.05;

   _____ Account(String owner, double deposit) {}
   public void deposit(double amount) {}
   public void withdraw(double amount) {}
   public void operationOutput() {}
   public String getInfo() {return "";}
   public _____ void feeChange(double f) {}
}
```

8.1.3

The constructor increases the number of existing accounts and sets the instance number of the current account. This will create space for 100 account statements.

According to these considerations fill the code:

```
public Account(String owner, double amount) {
   if (amount < 0) _____ new IlegalArgumentException("negative
deposit");
   accCount_____;
   accNum = _____;
   this.owner = owner;
   accStatus = amount;
   operationOutput = _____ String[_____][100];
}
```

8.1.4

The method for depositing an account is quite simple. The deposit amount is added to your account balance and the fee is deducted. The operation is written: the first parameter is the account from which the deposit was made, or the description of the operation (e.g. fee, deposit), the second parameter is the deposit amount. In addition, we will ensure that it is not possible to deposit less than the fee deducted from the account during the transaction.

According to these considerations fill the code:

```
public _____ deposit(String accNum, double amount) {
   if (amount > _____){
     accStatus = accStatus + amount - _____;
```

```
        transaction(accNum, amount);
        transaction("fee",-fee);
    }
}
```

8.1.5

Let's create an *override* deposit method that will be used if it is a cash deposit. It is different from the previous method by the number of parameters.

In the method body, we simply call the already implemented *deposit()* method with two parameters, the account number replacing with "cash deposit" text.

According to these considerations fill the code:

```
public void deposit(double amount) {
    _____("cash deposit", _____);
}
```

8.1.6

Writing operations is relatively simple. The current position of the matrix will indicate which account has sent or received the finance and the number of entries will increase. To do this we need to add an instance variable that will remember the number of operations that were written till now.

The method cannot be called from the outside because writing incorrect values could compromise integrity. Only class methods use it, so its visibility will be *private*.

If we wanted to ensure the correct behavior of the class after exceeding 100 operations in the account, it would be necessary to move the elements of matrix 1 forward and insert a new operation to the last place when exceeding 100 records.

According to these considerations fill the code:

```
_____ void transaction(String acc, double amount) {
    operationOutput[0][accCount] = acc;
    operationOutput[1][accCount] = Double._____(amount);
    transCount_____;
}
```

and edit the constructor:

```
public Account(String owner, double amount) {
   if (amount < 0) throw _____
IlegalArgumentException(“negative deposit”);
   accCount_____;
   accNum = accCount;
   this.owner = owner;
   accStatus = amount;
   operationOutput = new String[2][100];
   transCount = _____;
   // we will write the initial deposit - no fee
   transaction("deposit",amount);
}
```

8.1.7

Withdrawal/payment is a stressful operation not only for the real user but also for our class. The first step is to make sure that the amount can be withdrawn (whether an account balance is needed and whether we have a fee).

Feedback on a successful/unsuccessful operation must be provided to the user immediately after attempting to call the operation. We encode it into the Boolean return value of the method - if the operation succeeds method to return *true* otherwise *false*.

According to these considerations fill the code:

```
public _____ deposit(String acc, double amount) {
   if ((amount > 0) && (accStatus > amount + fee)) {
      accStatus = accStatus - amount - fee;
      transaction(acc, _____amount);
      transaction("fee", -fee);
      return _____;
    } else
       return _____;
}
```

8.1.8

Account information displays information about the attributes being retained. In previous programs, we tried to provide object instance info in the form of a return value so that a user of our class could decide how to display the data, so fill in the code in accordance with this practice:

```
public _____ getInfo() {
   _____ accNum + ": " + owner + " " + accStatus;
}
```

8.1.9

The last required ability is to provide a statement of transactions on the account. The header of the statement is secured by the existing *getInfo()* method, and the rest of the data is filled with the list of operations. In this case, we will not return the value but write it directly to the console.

According to these considerations fill the code:

```
public _____ operationListing() {
   System.out._____(getInfo());
   for(int i = 0; i < transCount; i++)
      System.out.println(operationOutput[0][_____] + " - " +
OperationOutput[_____][i]);
}
```

8.1.10

The last unresolved issue is the change of the fee for all existing (even created) instances. It is necessary to ensure that the content of the static variable *fee* is changed.

```
public class Account {
...
private static double fee = 0.05;
 ...
```

We can also use the class method (not static) to change it, but it can only be used when an instance exists.

If we use a static method, its code can be executed before the first instance is created by specifying the class name and method.

Fill in the code for the static fee changing method:

```
public _____ void feeChange(_____ fee) {
   if (fee > 0) this.fee = fee;
}
```

Calling a method before creating the first instance will take the following form, e.g.:

```
_____.feeChange(0.02);
```

8.1.11

Fill the code to verify the functionality of the created class:

```
public static void main(String[] args) {

   Account._____(0.03);                    // set the fee to
3 cents
   Account a1 = new _____("Herman", 300);
   a1.deposit(100);

   System.out.println(a1._____);           // output the info
of the account
   a1.deposit("123",100);
   a1.withdraw("123",50);

   a1._____();                        // output the operations
}
```

8.2 Inheritance I.

8.2.1

T: Create a junior account with the same characteristics as the bank account but ensure that operations on it are free of fee.

We could very easily make adjustments by changing the charge variable to 0 but this variable is static and would also provide zero value for non-junior accounts.

In addition, the code has a hard-coded insertion of the feed line into statements - the line would be unnecessarily inserted with zero value.

Given that we already have a number of methods in place and we have a programmatically secure whole class, making it into a universal form would be destructive rather than beneficial. Editing a class would also involve modifying all programs that use it.

We could also copy the whole class and make adjustments in the new one, but if there was a request to change the common methods, we would have to make the adjustments in two places (which is ineffective and often flawed).

Creating a new class based on the existing one without having to rewrite the existing code allows us to have an object-oriented language feature called ***inheritance***.

The new class will inherit all characteristics of the original class and if we make changes to the attributes or methods of the original class, they will immediately be reflected in the methods of the class derived by inheritance.

The original class is referred to as ***parent*** (parent), new as derived, ***child*** or heir.

8.2.2

Inheritance allows:

- creating a new class based on an existing class
- automatically translate changes in the parent class into a derived class
- translate the modifications in the derived class into the parent class

8.2.3

Benefits of inheritance:

- less code in the app, thanks to the ability to download existing code without copying it
- reusability of existing code - if we want to create a new class and there is already a class that contains some of the code we need, we can deduce the new class from the existing class
- less error due to code rewriting in one place - changing the code in the parent class also affects all derived classes

8.2.4

Inheritance provides us:

- shorter total code in the app
- ability to use existing code without duplicating it
- encapsulation and independence of classes around
- the ability to use multiple attributes

📖 8.2.5

Creating a new class based on an existing class means that the derived class will have the same attributes and all methods as the parent.

This step provides the keyword ***extends*** as follows:

```
public class JuniorAccount extends Account {
    // there will be definitions of new properties and methods
}
```

We'll add the keyword *extends* along with the name of the class we will inherit from. In our case, *JuniorAccount* is an extension (*extends*) of the *Account* class.

This notation creates the same properties and methods for the child as the parent class, plus we have space to add code to provide new and specific properties.

The thoughtful design of attributes and methods in the parent class is a prerequisite for simplifying programming - these become part of all derived classes.

8.2.6

What keyword is creating a derived parent class?

📖 8.2.7

A class that is derived from another class is called a ***subclass*** (also a derived class, extended class or daughter class).

The class from which a subclass is derived is called a ***superclass*** (also a parent class, base class, parent class or **superclass**)

Each class has only one direct parent class (simple inheritance), except for the *Object* class, which has no superclass.

Any newly created class, unless it is derived from an existing class, is always a subclass of the *Object* class.

Classes can be derived from classes that are derived from classes, and so on, up to the ultimate top-class *Object*.

8.2.8

What is the class that is the ancestor of all Java classes?

8.3 Inheritance II.

8.3.1

```
public class JuniorAccount extends Account {
   // there will be definitions of new properties and methods
}
```

There is no constructor defined in the new class. Since there is no user constructor defined in the child, the constructor is expected without parameters. This is first searched for in the *JuniorAccount* class but if not available, then in the parent class where it was "disabled" because there is a constructor with parameters.

```
public Account(String owner, double amount) {...}
```

This step identifies a problem that requires defining a custom constructor.

For our needs, the constructor can have exactly the same parameters as the constructor in the ancestor and can use the already defined sequence of orders entered in the bank account.

```
public JuniorAccount(String owner, double amount) {
   super(owner, amount);
}
```

The keyword ***super*** refers to a superclass (parent class). If the method name is specified, the method will be called, if it is specified without the method specification, the ancestor **constructor** is called.

In this way, we actually call an ancestor constructor with our parameters to take care of creating an account, filling and depositing the first deposit.

8.3.2

What keyword is used to call the parent constructor?

📖 8.3.3

Let's go through each method.

The deposit() method ensures that the amount is added to the account:

```
public void deposit(String account, double amount) {
   if (amount > fee) {
     accStatus = accStatus + amount - fee; // problem
     transaction(account, amount);
     transaction("fee", -fee);
   }
}
```

The deposit method is written in such a way that it automatically deducts a fee from the account which we don't want because we created a new class for that purpose.

Inheritance allows us not only to add new but also overwrite existing superclass methods.

A method in a subclass overlaps a method from a parent class if the overlapped method has the same name, number of parameters of the type and a return type as the method that overlaps it.

So let's add a method to the JuniorAccount class that, thanks to the same name and parameters, will replace the method inherited from the Account class.

```
@Override      // an optional tag that indicates that it is an
overlapped method
public void deposit(String acc, double amo) {
   accStatus = accStatus + amo;
   transaction(acc, amo);
}
```

The method has the same parameters (it does not matter that they have different names) and if the method is called in the JuniorAccount instance, this method is used, if the method is called in the Account instance, the original method is used.

When calling the transaction() method, the content of the JuniorAccount class is first checked, if it does not find the method, it executes the one that is defined in the superclass.

8.3.4

What is the order of class browsing when calling a method?

- methods of the Object class
- current class
- parent class
- superclass parent class

8.3.5

We have defined the classes to meet all safety requirements:

- attributes are private
- methods it does not need, the user does not see

... and this is causing us problems because we do not have access to the deposit() state variable - its visibility is

```
class Account {
   ...
   private double state;
   ...

   public void deposit(String u, double s) {
      state = state + s;
      entry(u, s);
   }
}
```

The private level allows access to the attribute only by methods of that class, preventing access from other classes but also from children. If you do not want the attribute to be public but want to make it available to all descendants of the class, you must set the access level to protected (in the Account).

The same applies to the entry() method, which we call in the deposit() method, added to the JuniorAccount class.

8.3.6

Fill the code for the properties of the attributes of each variable in the *Account* class:

```
public class Account {
   _____ int accNum;
   _____ String Owner;
   _____ double accStatus;
   _____ String[][] transaction;
   private _____ int accCount;
   private _____ double fee = 0.05;
   ...
}
```

8.3.7

The subclass does not have access to members of the parent class that have a private access level. When a derived class uses the methods of its parent it has indirect access to its members.

In our case, the attributes owner, statements, etc. - are private but the JuniorAccount class is accessed through the methods for the Account class. E.g.: the constructor for JuniorAccount sent the parameters to the constructor of the Account class:

```
class Account {
   ...
   public Account(String owner, double deposit) {}
      if (amount < 0) throw new
IlegalArgumentException("negative deposit");
      AccCount++;
      AccNum = AccCount;
      this.owner = owner;
      accStatus = amount;
      transaction = new String[2][100];
      transCount = 0;
      transaction("deposit",amount);
   }
   ...
}
class JuniorAccount extends Account {
   public JuniorAccount(String owner, double amount) {
      super(owner, amount);
   }
```

```
...
}
```

8.3.8

What keyword is used to make attributes and descendants of a derived class available?

8.3.9

In the Java programming language, each class can have one superclass and each parent class has an unlimited number of subclasses. It is not possible to combine multiple parents (multiple inheritances) for one class such as in C/C++ languages. However, there are ways to overcome this deficiency.

8.3.10

How many superclasses can a new class be derived from?

- just from one
- from none
- from a maximum of one
- from a maximum of two
- the number is not limited

8.4 Inheritance (programs)

8.4.1 Rectangle

Create a **Rectangle** class that can:

- retrieve positive integers as the dimensions of a rectangle via a **constructor**. If any value is less than or equal to zero, set it to 1.
- return the content of the rectangle using the **getContent()** method
- return the perimeter of the rectangle using the **getCircuit()** method

In the **getDraw()** method, return the rectangle drawn using the "o" characters so that the width is always greater than the height.

Override the **toString()** method to return the dimensions of the rectangle in the form: "a x b: " + a + " x " + b

Create a **TurnedRectangle** class, which will be derived from the **Rectangle** class

- will have the same methods,
- The rectangle will be drawn in the **getDraw()** method so that the width is less than the height.

An illustration of creating instances and calling methods is presented by the following procedure:

```
public static void main(String[] args) {
   Rectangle r1 = new Rectangle(4,2);
   System.out.println(r1.getContent()); // writes 8
   System.out.println(r1.getCircuit()); // writes 12
   System.out.println(r1.toString()); // writes a x b: 4 x 2
   System.out.println(r1.getdraw());
   /* writes
      oooo
      oooo   */
   TurnedRectangle r2 = new TurnedRectangle(4,2);
   System.out.println(r2.getContent()); // writes 8
   System.out.println(r2.getCircuit()); // writes 12
   System.out.println(r2.toString()); // writes a x b: 4 x 2
   System.out.println(r2.getdraw());
   /* writes
      oo
      oo
      oo
      oo   */
}
```

8.4.2 Animal

Create an **Animal** class that can:

- store information about the species, the sound the animal makes, the maximum weight, and the average age of life (integer attributes).

Create a **constructor** that populates the attributes with parameters in the order of species, sound, max weight, and age of survival. If an integer value is less than 1, set it to 1.

Add getters **getSound()** and **getAge()**.

Add a **sing()** method that returns a string containing the sound of the animal 3 times in a row separated by a space, e.g. miau miau miau

Add a **toString()** method that outputs data in the form:

```
species: weight/age (sound)
```

Create a **Bird** class that is derived from the Animal class

- It will additionally have an attribute informing about the wingspan.

Modify the **constructor** by adding the wingspan as the last (integer) parameter and setting it to 1 if its value is less than 1.

Add a **toString()** method that prints the data in the form:

```
species: weight/age/dispersion (sound
```

Create a **Fish** class, which will be derived from the Animal class

- will additionally have an attribute informing whether it is freshwater or marine (it will contain either freshwater or marine).

Modify the **constructor** by adding the type of fish as the last parameter, and if its value is less than the allowed value, set it to "marine".

Add a **toString()** method that will use the ancestor method and output the data in the form:

```
species: weight/age (sound) - fish type
```

The following procedure illustrates how to create instances and call methods:

```
public static void main(String[] args) {
   Animal a1 = new Animal("bear","brum",300,50);
   System.out.println(a1.sing()); // writes brum brum brum
   System.out.println(a1.toString()); // bear: 300/50 (brum)
   Bird b1 = new Bird("lark","cvrlik",1,10,15);
   System.out.println(b1.getAge()); // inherited from Animal,
writes 10
   System.out.println(b1.toString()); // lark: 1/10/15
(cvrlik)
   Fish f1 = new Fish("trout","hap",2,20,"freshwater");
   System.out.println(f1.sing()); // inherited from Animal,
writes hap hap hap
   System.out.println(f1.toString()); // trout: 2/20 (hap) -
freshwater
}
```

Polymorphism

Chapter 9

9.1 Concept and principle

📖 9.1.1

The *toString()* method has a special role in the class hierarchy. Usually, it returns information about the instance in an understandable form or converts the contents of the attributes into a string.

We used it for conversion, listing the contents of an error, etc., while its use is in use and editing the output for a particular class is a programmer's grace. The source of this method is the *Object* class, that is, for each class is available the *toString()* method.

For example, if you use it for an instance of the *Account* class defined as follows:

```
public class Account {
    ...
    private static int accCount = 0;
    ...
}
```

we get a string in the form:

```
p01_account.Account@9304b1
```

which provides information about the package, the class in it and the @ sign of the (unique) identifier of the instance.

This method seems unlikely to suit us, as we would expect it to replace the *getInfo()* method and get account information in a user-friendly form.

If we override this method, coming from the *Object* class, we can give up on the *getInfo()* method.

📝 9.1.2

Add the *toString()* method code to return information about the *Account* class instance.

```
@Override
public _____ toString() {
    _____ accNum + ": " + owner + " " + accStatus;
}
```

9.1.3

T: To register customer data, create the *Person* structure (name, address, birth year), it is sufficient to define the constructor and the *toString()* method.

- from the *Person* class derive the *Man* class which will also have an income attribute,
- from the *Person* class derive the *Woman* class which will also have height and number of kids,
- from the *Person* class derive the *Child* class which will also have a favorite toy parameter.

Create a customer list and output the content effectively.

9.1.4

Add code to define the *Person* class with the constructor and the redefined *toString()* method.

```
public class Person {
   private String name;
   _____ String adress;
   private _____ birth;

   public Person(String name, String adress, int birth) {
      _____.name = name;
      _____.adress = adress;
      _____.birth = birth;
   }

   public _____ toString() {
      _____ name + ": " + adress + ", year of birth.: " +
birth;
   }
}
```

9.1.5

The *Man* class has all the attributes of the *Person* class. Attributes are not accessed directly, but through methods of the *Person* class, so there is no problem that they are private.

In the constructor, it fills the attributes through the parent constructor and in the statement it also uses the parent method to obtain that portion of the attributes that belong to the *Person* class.

According to these considerations fill the code:

```
public class Man _____ Person {
   private int salary;

   public Man(String n, String a, int b, int s) {
      _____(n, a, b);
      salary = s;
   }

   _____ String toString() {
      return _____.toString() + ", salary: " + salary;
   }
}
```

9.1.6

The same rules apply to the definition of *Woman* and *Child* class. Fill in the following code:

```
public _____ Woman _____ Person {
   private int height, kids;

   public Woman(String n, String a, int b, int h, int k) {
      _____(n, a, b);
      height = h;
      kids = k;
   }

   public _____ toString() {
      return _____.toString() + ", height: " + height + ";
kids: " + kids;
   }
}
public _____ Child _____ Person {
   private String toy;

   public Child (String n, String a, int b, String t) {
      _____(n, a, b);
      toy = t;
```

```
    }

    _____ String toString() {
        return _____._____() + ", favorite toy: " + toy;
    }
}
```

📖 9.1.7

We will create a customer list that will list all types of people we register.

Each customer comes from the *Person* class regardless of whether it is an instance of the *Person* class or an instance of the derived class. Every customer **is** therefore a *Person*. We can create e.g. an array of people:

```
Person[] customerList = new Person[10];
```

However, the specific customer we will be adding to the list is no longer a person but a male, female or child. Because they were derived from the *Person* class we can also treat them as people. The following listing

```
Man c1 = new Man("Peter", "London", 1990, 1500);
customerList[0] = c1;
```

is perfectly fine.

You can also fill array elements by directly inserting an instance reference returned by the constructor:

```
customerList[1] = new Man("Pete", "Casablanca", 1986, 2000);
customerList[2] = new Woman("Anne", "Paris", 1985, 168, 2);
customerList[3] = new Child("Joan", "Amsterdam", 2010,
"Lego");
```

Each of the created instances is, except its class instance, also an instance of the *Person* class.

9.1.8

The *Plant* class is given from which the *Tree*, *Flower* and *Mushroom* classes are derived. Fill the code so that instances created by different constructors can be stored in a common field.

```
_____[] list = new _____[10];
```

```
list[0] = _____ Tree("birch", 10, 100);
list[1] = _____ _____("cornflower", "blue");
list[2] = _____ Mushroom("amanita", "red", "poison");
```

📖 9.1.9

The customer list output uses a feature called ***polymorphism***.

```
for(int i = 0; i < 4; i++) {
   System.out.println(customerList[i].toString());
}
```

Calling *toString()* method can do something different for each array element. The call depends on the type of instance that is stored in the element. Calling a method with the same name always executes the version of the method that belongs to the instance being processed. For each type of person different information will be displayed:

- man: name, address, year of birth, salary
- woman: name, address, year of birth, height, number of kids
- child: name, address, year of birth, favorite toy

Each instance knows its origin and uses the version of the method closest to it ...

9.1.10

Fill the code to list the different characteristics of the *Plant* class children:

```
public class Plant {
   private String title;

   public Plant(String title) { _____.title = title; }
   public _____ toString() { _____ title;   }
}
public class Tree extends Plant {
   private int age, height;

   public Tree(String title, int age, int height) {
      _____(title);
      this.age = age;
      this.height = _____;
   }
```

```
   public String toString() {
      return super._____ + ", height: " + height + ", age" +
age;
   }
}
public class Flower extends Plant {
   private String color;

   public Flower(String title, String color) {
      _____(title);
      this.color = color;
   }

   public _____ toString() {
      return _____.toString_____ + ", color: " + color;
   }
}
```

Output:

```
_____[] list = new _____[10];
...
for(int i = 0; i < plantCount; i++) {
   System.out.println(list[i]._____);
}
```

📖 9.1.11

Polymorphism is a consequence of the implementation of inheritance. The basic idea of polymorphism is that a child can replace a parent. This practically means that we can assign a reference to a child instance in the reference variable declared by the parent type. Thus, every instance can behave as if it were an instance of any of its superclasses.

Nevertheless, each instance 'knows' its true origin and the body selection method to be used will only take place at the time of program execution. If it finds the called method code in its body, it will use it. If not, searches in superclass etc.

As a result, calling a method with the same name for two instances with the same parent (at different levels of the hierarchy) can perform different activities.

9.1.12

Will two different method calls with the same name perform different actions (command sequences) in the case of such a writer?

```
MyClass my = list[0];
my.do();
MyClass my = list[1];
my.do();
```

- yes
- no

9.2 Polymorphism (programs)

9.2.1 Person

Create a **Person** structure to store the name, address, and year of birth of a person to record customer data.

- Define a constructor that will set the name, address and year of birth, checking the year of birth to be in the range 1900-2020, if outside this range let it set the year of birth to 2000.
- Define a **toString()** method that returns data of the form: name + " (" + yearBirth + "), " + address.

Derive the **Man** class from the **Person** class and add an extra integer attribute to hold the salary amount.

Modify the **constructor** so that

- add an integer parameter at the end to represent the salary amount
- use the ancestor constructor
- you will check the salary amount: if it is outside the range of 100-10000, set it to 0.

Override the **toString()** method, which

- returns data of the form: name + " (" + yearBirth + "), " + address + ": " + income + " EUR",
- uses the **toString()** method on the ancestor.

Derive the **Woman** class from the **Person** class and add an extra text attribute to store hair color and an integer attribute to store hair length.

Modify the **constructor** so that

- add a text parameter at the end to hold the hair colour and an integer parameter to hold the hair length
- use the ancestor constructor
- check the hair length: if it is less than 0 set it to 1, if it is more than 120 set it to 120.
- only allow four colors: black, blonde, brown, red, if other input set neutral :)

Override the **toString()** method which:

- returns data of the form: name + " (" + yearBirth + "), " + address + ", hairs: " + colorHair + " - " + lengthHair,
- invokes the **toString()** method on the ancestor.

Derive the **Kid** class from the **Person** class and add a text attribute to it storing the parent's name.

Modify the **constructor** so that

- use the ancestor constructor

Override the **toString()** method that:

- returns data of the form: name + " (" + yearBirthday + "), " + address + ", parent: " + nameFamily
- invokes the **toString()** method on the ancestor.

Create a **CustomerList** class that will contain:

- an array of persons as a list into which you will insert customers of type person (max 20)
- define an attribute that will contain the number of populated elements in the list

Define a **constructor** without parameters, in which you create an array and set the count to 0

Add an **addMan()** method in which you create and add a man to the list. Let the parameters of the method be the same as in the man constructor - you will use them when creating the man.

Add an **addWoman()** method to create and add a woman to the list. Let the method parameters be the same as in the woman constructor.

Add an **addKid()** method to create and add a child to the list. Let the method parameters be the same as in the child constructor.

Add a **getList()** method in which you use the **toString()** method to return complete customer information from the customer list. Let each customer's information be listed on a separate line.

The following procedure illustrates the creation of instances and method calls:

```
public static void main(String[] args) {
   Man m = new Man("George", "Kahira", 2026, 3000);
   Woman w = new Woman("Adela", "London", "1990", "black",
160);
   Kid k = new Kid("Adam", "London", 1880, "Ivan");
   System.out.println(m.toString()); // writes: George (2000),
Kahira: 0 EUR
   System.out.println(w.toString()); // writes: Adela (1990),
London, hairs: black - 0
   System.out.println(k.toString()); // writes: Adam (2000),
London, parent: Ivan

   CustomerList cl = new CustomerList();
   cl.addMan("George", "Kahira", 2026, 3000);
   cl.addKid("Adam", "London", 1880, "Ivan");
   cl.addWoman("Adela", "London", "1990", "black", 160);
   System.out.println(cl.getList());
   /* writes
     George (2000), Kahira: 0 EUR
     Adam (2000), London, parent: Ivan
     Adela (1990), London, hairs: black - 0
   */
}
```

PRISCILLA

priscilla.fitped.eu