# Exercise 1

# Developing Java application from two classes

## Objectives

Create a Java application comprising two classes. Learn to create objects, invoke methods, and execute the application.

## Tasks

1. Create the **HelloWorld** Java class, define the **main()** method.
2. Create the **MessagePrinter** class and the **String sayHello()** method within it.
3. Create a **MessagePrinter** class instance within the **main()** method and invoke the **sayHello()** method.
4. Compile and launch the application.

## Detailed guidelines

1. Create the **HelloWorld** Java class, define the **main()** method.
2. Create the **MessagePrinter** class and the **String sayHello()** method within it. The method must return the **“Hello, world”** message.
3. Create a **MessagePrinter** class instance in the **main()** method and invoke the **sayHello()** method.
4. Compile and launch the application. Make sure that the compiler automatically looks up and compiles dependent file **MessagePrinter.java**.

# Exercise 2 (Project m2-jva007-bank-app)

# Bank Application: Creating Bank Application

## Objectives

1. Learn principles of class creation and constructor definition.
2. Learn how to work with Java **enum**.
3. Understand the principles of polymorphism, overloading, and method overriding.
4. Learn to create abstract classes and interfaces.
5. Learn package-wise class division and the usage of access modifiers, as well as the **final** modifiers.

## Model

# 

## Tasks

### **Task 1**

1. Create the classes defined by the diagram. For the sake of simplicity, on this step, bank has one ten-client-array where one client can have one account.
2. The **Account** class has the constructor that accepts the **id** and the starting **balance** value, and also has **deposit(x)** method adding the **x** value to the balance and **withdraw(x)**, that decreases the balance by **x**, in case **balance >= x**. Implement the **getBalance()** method that returns the current account balance value.
3. Create the **BankApplication** class. **BankApplication** creates a **Bank** class object and adds several new clients to the bank. Each new client must have one account with some starting value.
4. Implement the **BankApplication.modifyBank()** method that changes balance values (using the **deposit()** and **withdraw()** methods) for some bank clients’ accounts.
5. Implement the **BankApplication.printBalance()** method that iterates through the bank clients and prints the balance value of their accounts.

### **Task 2**

1. Add the feature of specifying client’s gender and name. Define corresponding fields of the **Client** class and constructor.
2. The field value shall be specified with the help of the **enum Gender**. Create a corresponding **enum** with instances **MALE**, **FEMALE**.
3. Create the **getClientGreeting()** method of the **Client** class that displays hello message in a way: Mr. NAME or Ms. NAME depending on client’s gender.
4. If the previous step has been performed with the help of conditional operator (for example **if (gender == Gender.MALE) {}**), change the conditional statement. To do this, create a constructor and access methods for accessing gender string representation (“Mr.” and “Ms.”) in **Gender** enumeration.

### **Task 3 (improving the application, “layering”, refactoring)**

1. Create the **BankService** class that will add clients to the bank and will perform other services. The **BankService.addClient(bank, client)** static method adds a client to a bank. The **BankService** class methods are invoked from **BankApplication**.
2. Execute code refactoring. Place the **Bank, Client, Account, Gender** classes into the **com.luxoft.bankapp.domain** package. Place the **BankService** class to **com.luxoft.bankapp.service**.
3. Review domain objects, limit visibility scope of methods that are not part of client’s API (i.e. inner and utility methods).

### **Task 4 (polymorphism)**

1. New business logic requirement: there are two kinds of accounts. One kind is a **SavingAccount**. Second kind is a **CheckingAccount**. The diagram of this operation is different in a way that when creating this account the positive **overdraft** value is indicated, i.e. a loan that a Bank issues to a Client. In case if the **x** value of the **withdraw(x)** method is larger than the current **balance** value, an additional amount of money is allowed to be withdrawn within the limit of the **overdraft** value.
2. Implement the **Account** interface (that defines the methods **deposit(double x)** and **withdraw(double x))** and the abstract class **AbstractAccount** that implements the **Account** interface and has two subclasses: **SavingAccount** and **CheckingAccount**.
3. Change the code that used the **Account** class to use the **Account** interface.
4. Define into the **Account** interface the **maximumAmountToWithdraw()** methodthat returns the amount of money that can be paid to a client (taking into account the **overdraft** for **CheckingAccount**), and implement it in **SavingAccount** and **CheckingAccount**.
5. **BankApplication** must create accounts of both types. Implement the **BankService.printMaximumAmountToWithdraw(bank)** method that prints corresponding value for each account. Learn the work of virtual methods and polymorphism principles.

# Exercise 3

# Designing and developing an application.

## Objectives

Learn how to design and develop oop applications.

**Detailed guidelines**

Choose one of the following tasks:

1) **University**: create a system for accounting students, attendance, and academic achievements. Optional: it must contain reviews of teachers.

2) **Shop**: should keep information about the goods and sales of the goods. Optional: The system should provide discounts for regular customers.

3) **Clinic**: to create a system for the appointment to the doctors, to the time of the doctors. Optional: keeping patient cards by physicians.

4) **Airport**: you must manage the landing of the aircrafts, with their distribution in time and runways. Optional: notifications about flight delays to passengers, via SMS.

5) **Restaurant**: to create a system of accounting for the employment of tables, selection of dishes from the menu, bill payment. Optional: allow the reservation of tables.

# Exercise 4 (T1\_Enums project m2-jva007)

# Enums

## Objectives

Understand Java enums, how they work and how to use them.

## Tasks

Execute the Enums program that works with Java enums. Make analysis for the results that are displayed and follow the behavior.