KAZAKH-BRITISH TECHNICAL UNIVERSITY

KBTU

Kazakh-British Technical University
School of Information Technologies and Engineering

Assignment #1

Mobile programming
Introduction to a Kotlin

Prepared by: Maratuly T.
Checked by: Serek A.

Almaty, 2024

CONTENT

INTRODUCTION

This work is responsible for getting familiar with Kotlin programming language along with the Android studio as a developer IDE. Windows 10 was an operational system (OS) which we used to cover this lab. Pay attention to a user's username (here 'temir is a username of Maratuly Temirbolat with ID 23MD0409).

1. Exercise 1: Kotlin Syntax Basics

Before the beginning of the lab's process pay attention that username 'temir' (in the project's path) belongs to Maratuly Temirbolat (ID 23MD0409). Figure 1.1 illustrates who is the owner of the account which will be used to cover the work.

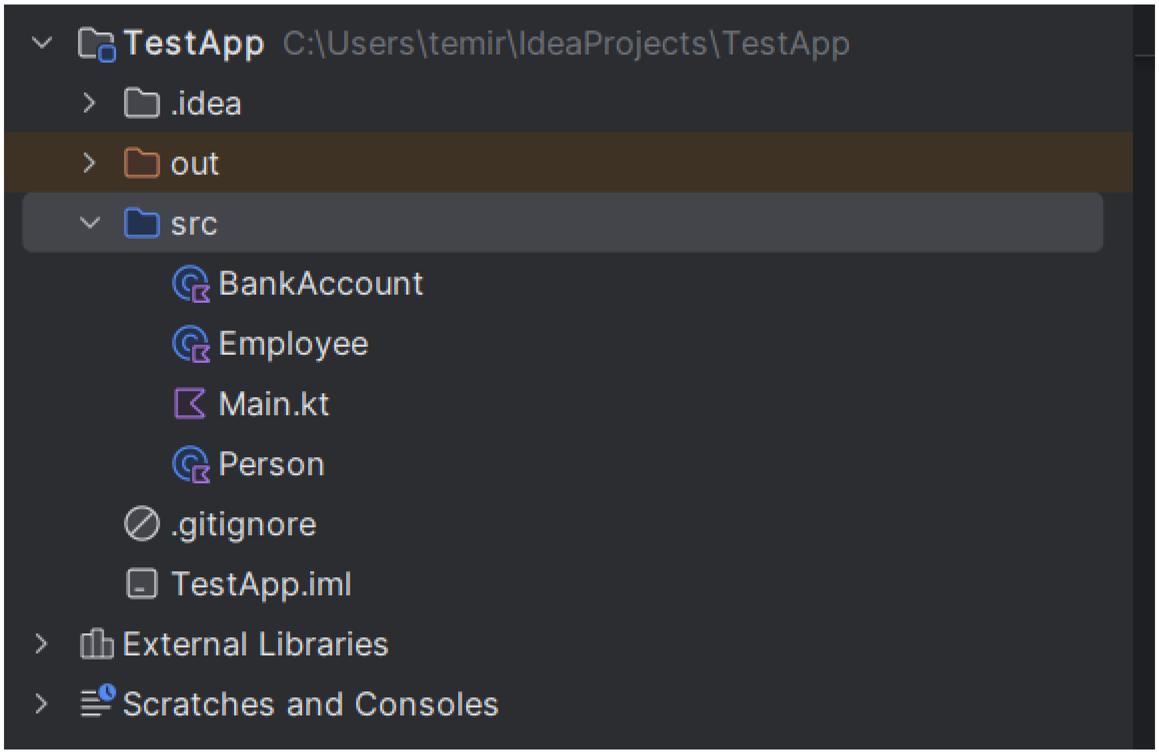


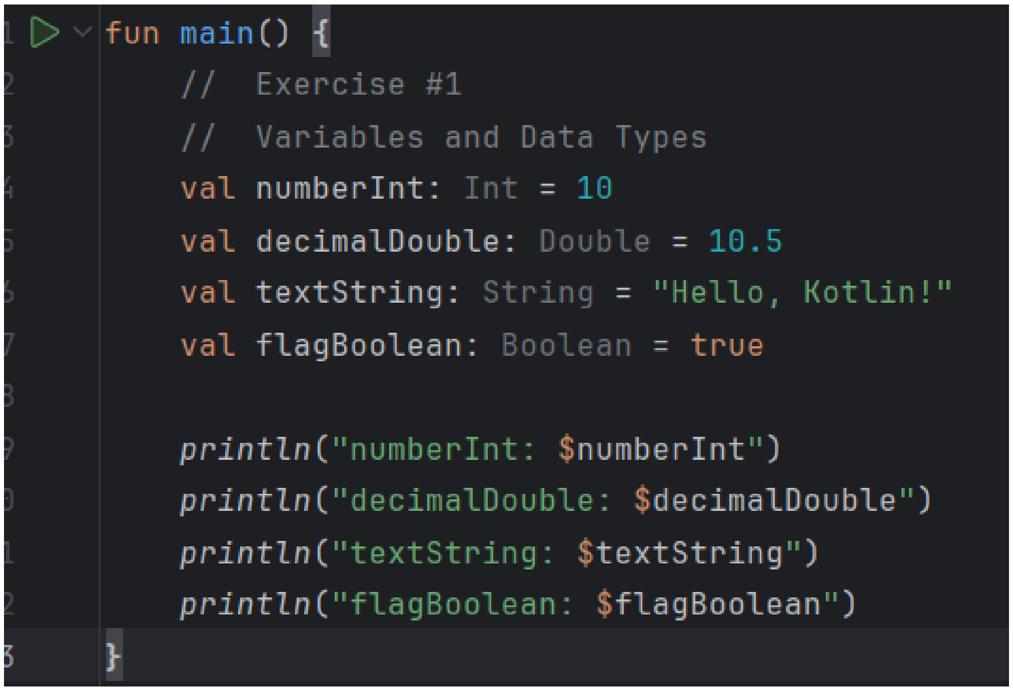
Figure 1.1 – Working tree of a project

According to a figure, the project consists of the '.idea' (hidden) folder with the idea's configuration, 'src' folder having data about the projects files (models, main code, etc.), '.gitignore' to tell git what files or folders we should ignore, emulator of a project. 'External Libraries' is a folder that has all the third party packages which are used in a project, 'out' is used to show the results.

1.1 Variables and Data Types

As soon as we verified the account's credentials, we can start covering the tasks' exercises. For the first task we need to create variables and annotate their data types. We are required to create the variables of types: Int, Double, String and Boolean. Then it is necessary to print them in a console to view the results using 'println' built in command. All the compiled exercises were done in 'Main.kt' file with extension for Kotlin syntax.

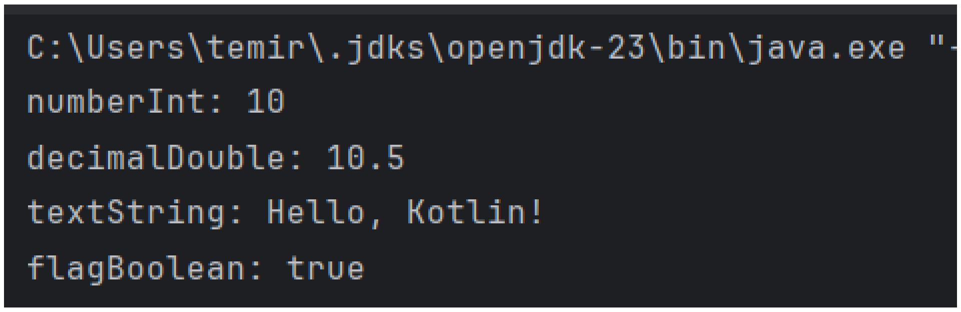
Figure 1.2 illustrates the first task's code and comments that were used to leave notations about the tasks' number and their titles.



```kotlin
fun main() {
    //  Exercise #1
    //  Variables and Data Types
    val numberInt: Int = 10
    val decimalDouble: Double = 10.5
    val textString: String = "Hello, Kotlin!"
    val flagBoolean: Boolean = true

    println("numberInt: $numberInt")
    println("decimalDouble: $decimalDouble")
    println("textString: $textString")
    println("flagBoolean: $flagBoolean")
}
```

Figure 1.2 – Basic common data type in Kotlin

We can see that Int, Double, Sting and Boolean data types are basically just classes because they start with a capital letter. The 'println' results are hold in Figure 1.2.



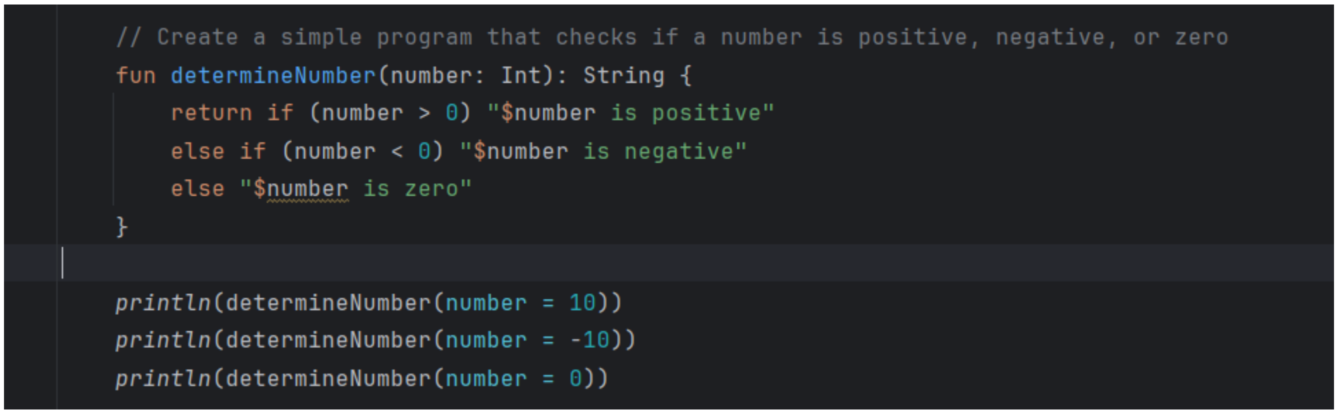
```
C:\Users\temir\.jdks\openjdk-23\bin\java.exe "
numberInt: 10
decimalDouble: 10.5
textString: Hello, Kotlin!
flagBoolean: true
```

Figure 1.2 – Results of printing data types of values using 'println'

Thus, each variable has its own address where the value is stored. Here camel case was used to notate the names of variables (Kotlin's approach of naming) as it bases on Java programming language. It means that Kotlin language was obtained from Java and we can use it instead of each other while the application can continue working with no issues.

1.2. Conditional statements

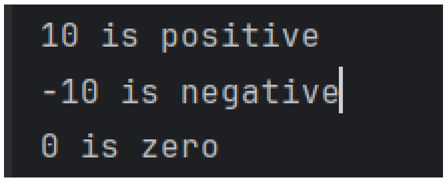
As we learnt how to create variables, now we move to conditions according to which we can decide what code should be executed. Here we need to create a simple program that checks if a number is positive, negative, or zero. For this purpose a function 'determineNumber' was implemented which takes a single 'Int' variable and returns a 'String' which says whether the number is positive, negative or none of them (means zero). Figure 1.3 shows the code of the task.



```
// Create a simple program that checks if a number is positive, negative, or zero
fun determineNumber(number: Int): String {
    return if (number > 0) "$number is positive"
    else if (number < 0) "$number is negative"
    else "$number is zero"
}

println(determineNumber(number = 10))
println(determineNumber(number = -10))
println(determineNumber(number = 0))
```

Figure 1.3 – Code of a function to determine what number belongs to

We can see that function returns immediately a conditional statement's result and we don't crate extra variables and blocks of code. The results of an executed code could be viewed in Figure 1.4.



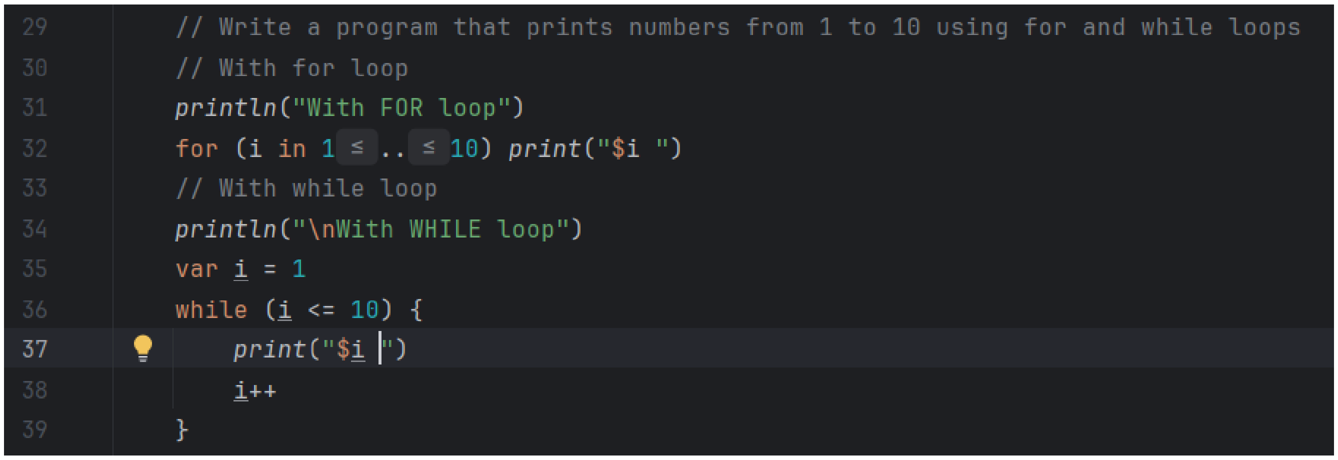
```
10 is positive
-10 is negative
0 is zero
```

Figure 1.4 – Results of a 'determineNumber' function

So, the console shows that 10 is a positive number, -10 is a negative one while 0 is zero. As it was expected, the tests are passed.

1.3 Loops

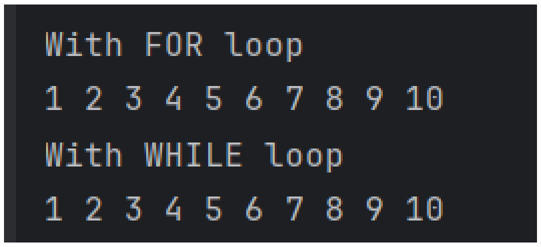
One of the most crucial topics in programming is the usage of loops. The loops are basically used to repeat some logic more than one time in order not to have duplicated code. Another usage is to iterate through the object, or collection of objects. Here, we needed to use ‘for’ and ‘while’ loops to print numbers from 1 to 10 inclusively. The code that combines these two approaches is shown in Figure 1.5



```kotlin
// Write a program that prints numbers from 1 to 10 using for and while loops
// With for loop
println("With FOR loop")
for (i in 1 ≤ .. ≤ 10) print("$i ")
// With while loop
println("\nWith WHILE loop")
var i = 1
while (i <= 10) {
    print("$i ")
    i++
}
```

Figure 1.5 – Code to print number from 1 to 10 using ‘for’ and ‘while’ loops

It can be clearly seen that the usage of ‘for’ compared to a ‘while’ is much simpler and looks more convenient since it has its own block of code where variable is initialized, whereas ‘while’ loop requires to create a variable manually and outside of the loop block. The results of a code are shown in Figure 1.6



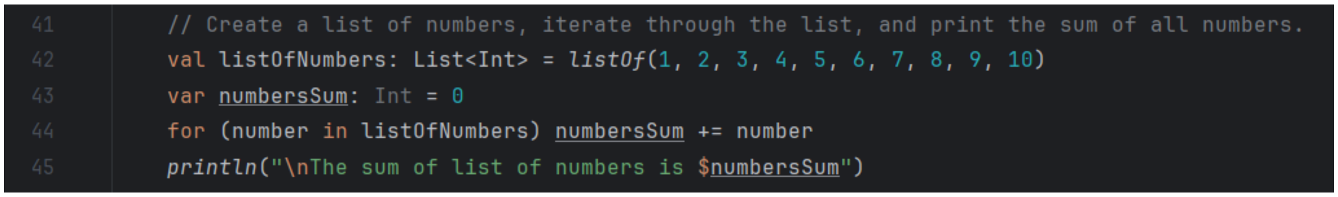
```
With FOR loop
1 2 3 4 5 6 7 8 9 10
With WHILE loop
1 2 3 4 5 6 7 8 9 10
```

Figure 1.6 – Results of using ‘for’ and ‘while’ loops

Generally, each loop has its own advantages and disadvantages, but here we just needed to get the same results demonstrating the usage both. Commonly, we can use any loop type as we want or simpler to use in a concrete situation.

1.4 Collections

Next, moving to more advanced topic we come to collections. Collections is just a box that contains plenty of objects inside of it to work with a group pointing at one object. These objects inside of a box has addresses just next to a single one + its number of bytes that is used. The exercise wants us to create a list of numbers, iterate through the list, and print the sum of all numbers. The solution code of a task is represented in Figure 1.7 just below.



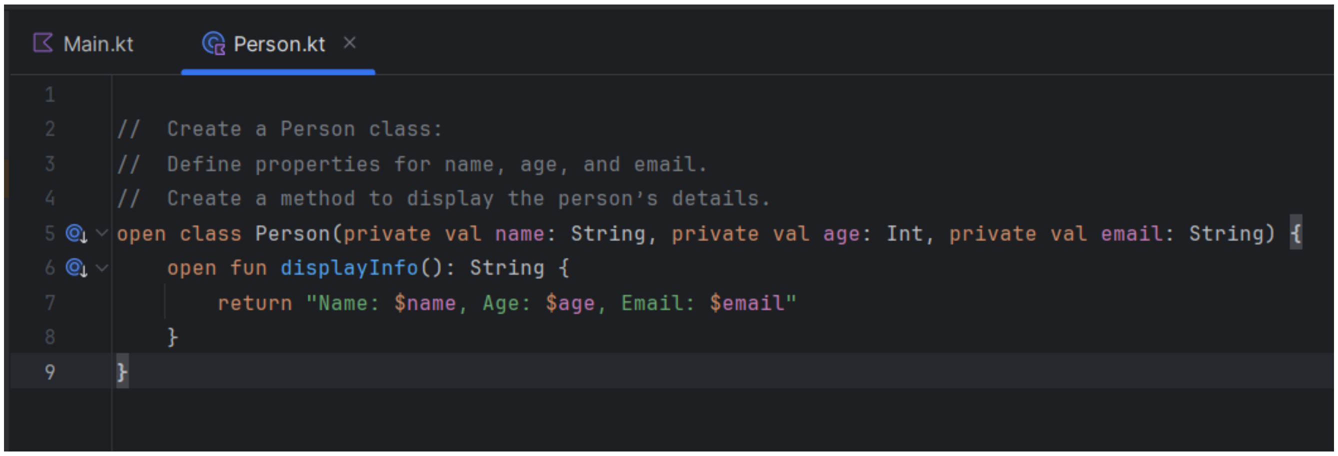
```kotlin
    // Create a list of numbers, iterate through the list, and print the sum of all numbers.
    val listOfNumbers: List<Int> = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    var numbersSum: Int = 0
    for (number in listOfNumbers) numbersSum += number
    println("\nThe sum of list of numbers is $numbersSum")
```

Figure 1.7 – Code with a collection of numbers and its sum

We can see that we used ‘listOf’ built in function to generate a collection of numbers. Additionally, we set the data type of out list to tell compiler what is inside.

2. Exercise 2: Kotlin OOP (Object-Oriented Programming)

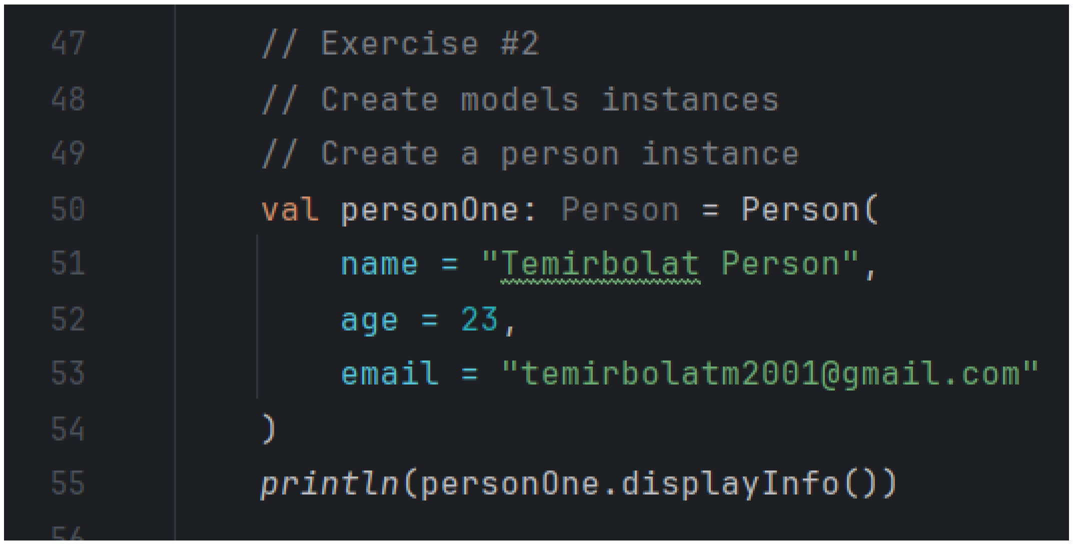
OOP or Object-Oriented Programming is an approach where models are involved and aimed to represent a real object with its own behavior. OOP concept is widely used in plenty companies and programming languages, especially Java has all the features to use OOP’s full power since it is mostly class based. As a simple task we need to create a class ‘Person’, define properties like ‘name’, ‘age’, ‘email’ and provide a method to show person’s details. Figure 1.1 shows that in project tree that Person class is present, whereas Figure 1.8 contains the code of its implementation.



```kotlin

//  Create a Person class:
//  Define properties for name, age, and email.
//  Create a method to display the person's details.
open class Person(private val name: String, private val age: Int, private val email: String) {
    open fun displayInfo(): String {
        return "Name: $name, Age: $age, Email: $email"
    }
}
```

Figure 1.8 – Class Person implementation

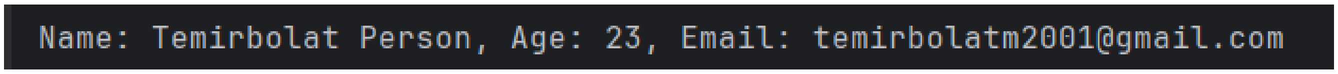
As a class feature we can define variables assign in Person’s constructor and set them private as mode, so, we can access them only inside of the class. We have also function ‘displayInfo’ which returns person’s name, age and email. We can create instances of any model to specify what are the values of fields and test its behavior in different situations. Figure 1.9 demonstrates it.



```kotlin
    // Exercise #2
    // Create models instances
    // Create a person instance
    val personOne: Person = Person(
        name = "Temirbolat Person",
        age = 23,
        email = "temirbolatm2001@gmail.com"
    )
    println(personOne.displayInfo())
```

Figure 1.9 – Person’s instance with specified name, age and email

The log of person’s information is shown in Figure 1.10.



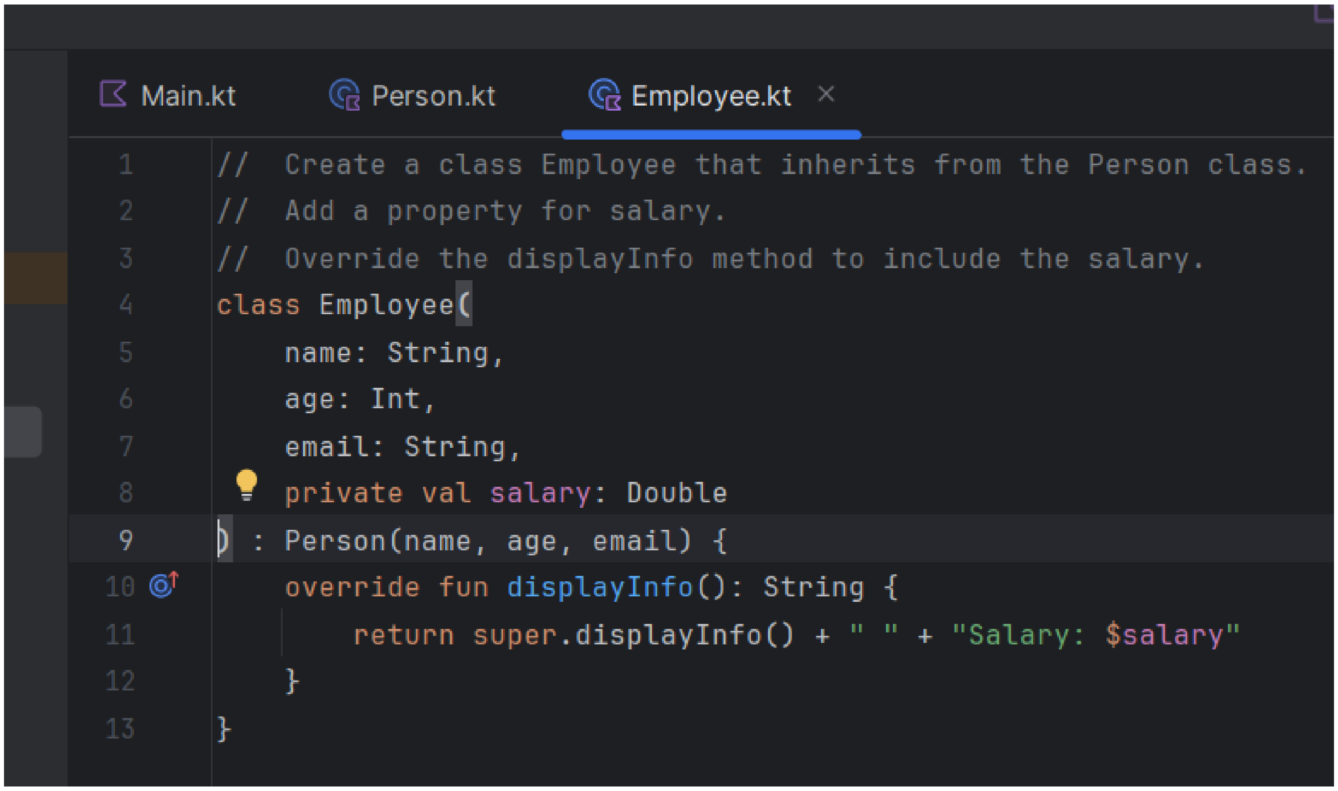
```
Name: Temirbolat Person, Age: 23, Email: temirbolatm2001@gmail.com
```

Figure 1.10 – Log about person’s data

Thus, everything works correctly.

2.1 Inheritance

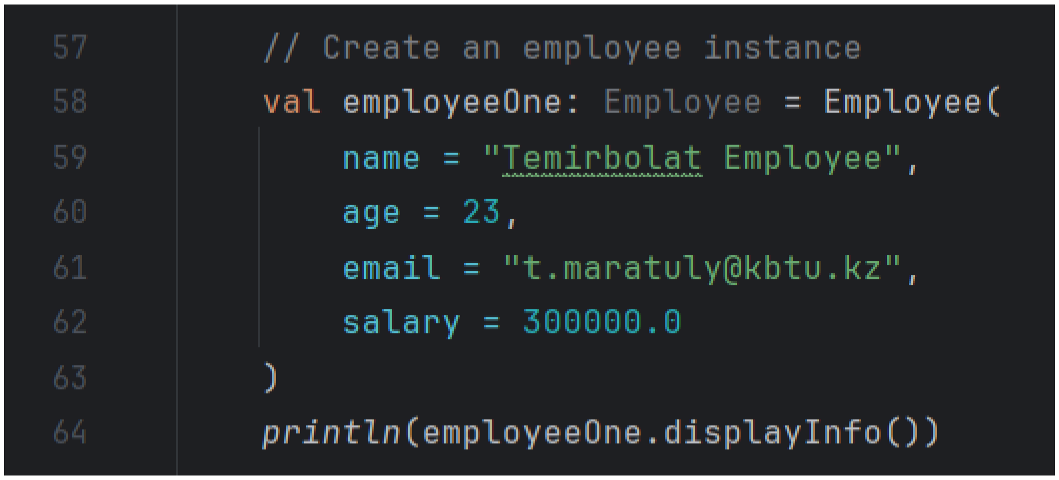
Inheritance is an approach where one class extends all the scheme of another class. It means that, for example, class A can acquire all the fields and methods from class B as it extends it. It is very convenient as we follow the rules of its implementation. To demonstrate an inheritance, we need to create some more classes. Let’s create class ‘Employee’ which extends class ‘Person’ as any employee is also a person. We can also add extra field ‘salary’ for a ‘Employee’ class and override ‘displayInfo’ method from a parent class. Figure 1.11 demonstrates an implementation code of a class ‘Employee’ which extends ‘Person’ while Figure 1.1 revises the file’s location in a project. So, they are in the same directory and in different files, however they see each other as they are set to be in the same package.



```kotlin
//  Create a class Employee that inherits from the Person class.
//  Add a property for salary.
//  Override the displayInfo method to include the salary.
class Employee(
    name: String,
    age: Int,
    email: String,
    private val salary: Double
) : Person(name, age, email) {
    override fun displayInfo(): String {
        return super.displayInfo() + " " + "Salary: $salary"
    }
}
```

Figure 1.11 – Implementation of the ‘Employee’ class using inheritance

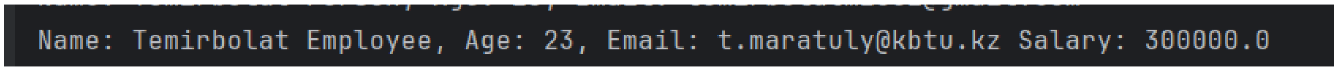
According to a figure we provide the same parameters for an employee constructor and redirect them into Person class from which we inherit. The inheritance is done using ‘:’ sign, while we override ‘displayInfo’ method using ‘override’ construction, however, this method in class ‘Person’ must be marked as ‘open’ to be overridden. The instance of a class ‘Employee’ is shown in Figure 1.12.



```kotlin
// Create an employee instance
val employeeOne: Employee = Employee(
    name = "Temirbolat Employee",
    age = 23,
    email = "t.maratuly@kbtu.kz",
    salary = 300000.0
)
println(employeeOne.displayInfo())
```

Figure 1.12 – Instance of a class 'Employee’ with extra field

The ‘println’ results are shown in Figure 1.13. The details are presented with additional field ‘salary’.



```
Name: Temirbolat Employee, Age: 23, Email: t.maratuly@kbtu.kz Salary: 300000.0
```

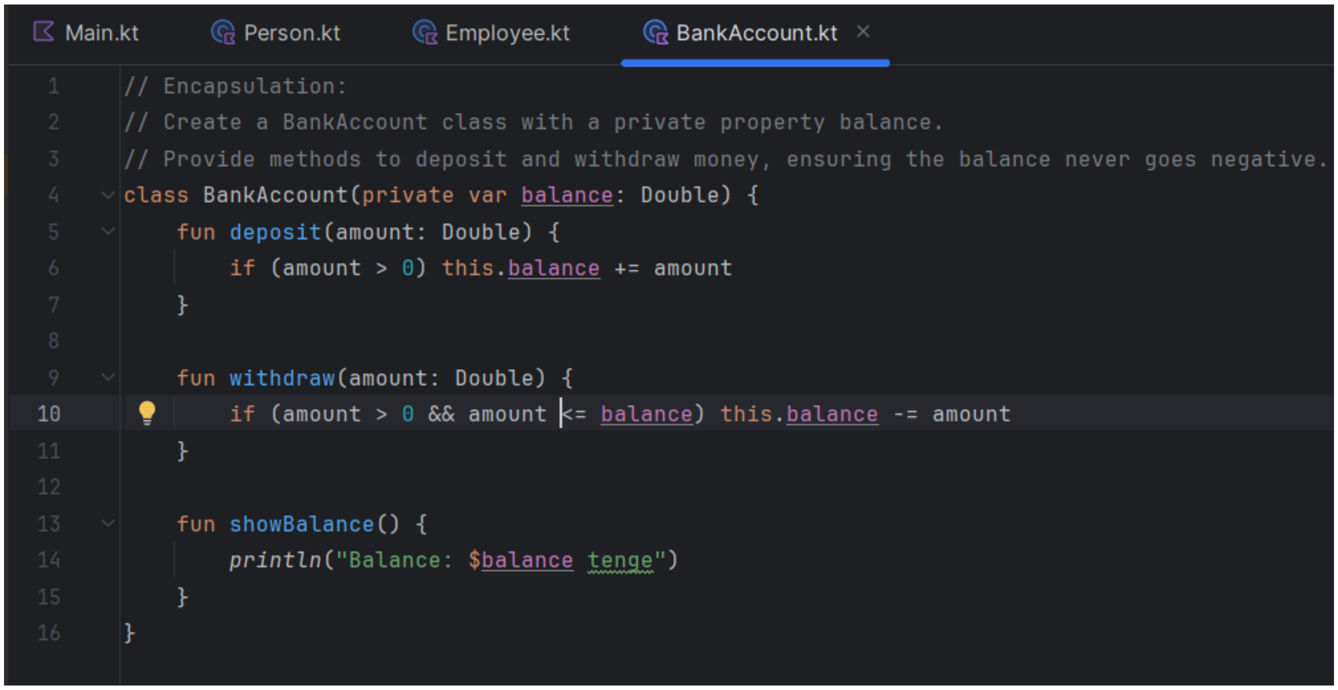
Figure 1.13 – Log about employee’s data

It can be clearly seen that we expanded ‘displayInfo’ from the class ‘Person’ in ‘Employee’ class as we used it and added ‘salary’ information.

2.2 Encapsulation

Encapsulation is a technique of restricting a user from directly modifying the data members or variables of a class in order to maintain the integrity of the data. Simply saying, it is used to make data (fields or methods) private in classes to work with them only inside of a class, while outside of a class they are not visible.

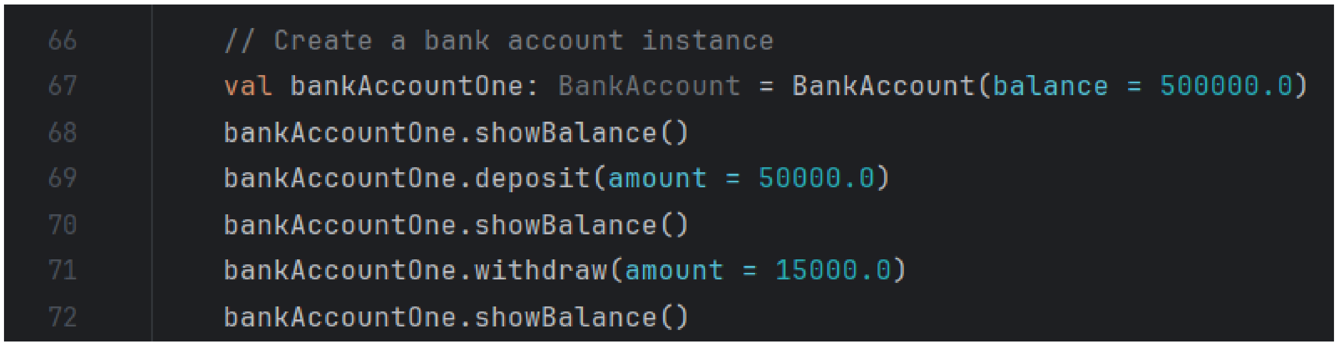
For the current task we need to create a class ‘BankAccount’ with private property ‘balance’. Moreover, we need to provide methods to deposit and withdraw money making sure that balance never becomes negative. Figure 1.14 shows the implementation of a class and Figure 1.1 allocates the file’s location.



```kotlin
// Encapsulation:
// Create a BankAccount class with a private property balance.
// Provide methods to deposit and withdraw money, ensuring the balance never goes negative.
class BankAccount(private var balance: Double) {
    fun deposit(amount: Double) {
        if (amount > 0) this.balance += amount
    }

    fun withdraw(amount: Double) {
        if (amount > 0 && amount <= balance) this.balance -= amount
    }

    fun showBalance() {
        println("Balance: $balance tenge")
    }
}
```

Figure 1.14 – Implementation of ‘BankAccount’ class using encapsulation

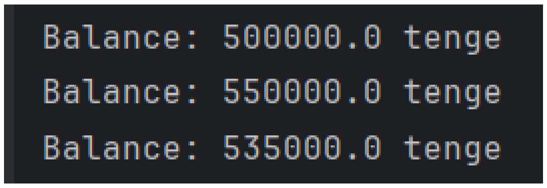
It can be clearly seen from a figure that field ‘balance’ is private in a constructor, that’s why we won’t manage to reach it outside of the class. We have also 3 methods to work with balance: ‘deposit’, ‘withdraw’ and ‘showBalance’. ‘Deposit’ is responsible for adding money to a balance (here we make sure that provided amount is positive), ‘withdraw’ is responsible the take set amount of money (here make sure that requested number of moneys is no more than a balance and positive as well), ‘showBalance’ is used to just demonstrate the values of fields. Figure 1.15 shows the instance creation and manipulation of the methods.



```
    // Create a bank account instance
    val bankAccountOne: BankAccount = BankAccount(balance = 500000.0)
    bankAccountOne.showBalance()
    bankAccountOne.deposit(amount = 50000.0)
    bankAccountOne.showBalance()
    bankAccountOne.withdraw(amount = 15000.0)
    bankAccountOne.showBalance()
```

Figure 1.15 – Instance creation and manipulation with its methods

The results and logs about account’s balance is illustrated in Figure 1.16.



```
Balance: 500000.0 tenge
Balance: 550000.0 tenge
Balance: 535000.0 tenge
```

Figure 1.16 – Logs of methods manipulations

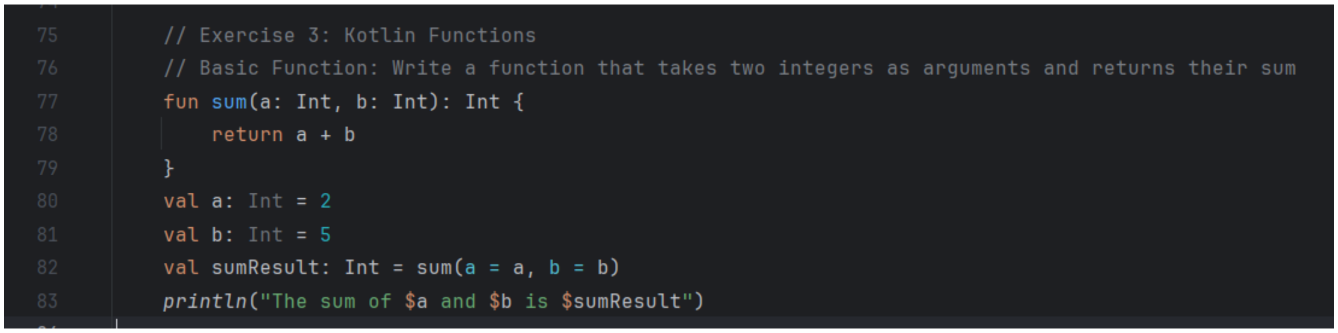
So, we can see that all the methods works perfectly and balance changes as we withdraw or add money.

3. Exercise 3: Kotlin functions

Function is a block of a code which can be called almost from any part of a project as it is visible and accessible. They are used in cases where we don’t want to duplicate a logic code and want to just reuse it. Functions can take parameters which could we used to specify the flow of a code by checking for specified values. The function can be anonymous and basic one. Anonymous function can be used as a callback one (provided as a parameter).

3.1 Basic functions

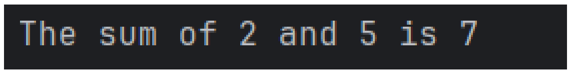
For the first task, we need to implement a basic function that takes two integers as arguments and returns their sum. Figure 1.17 illustrates function's implementation.



```kotlin
// Exercise 3: Kotlin Functions
// Basic Function: Write a function that takes two integers as arguments and returns their sum
fun sum(a: Int, b: Int): Int {
    return a + b
}
val a: Int = 2
val b: Int = 5
val sumResult: Int = sum(a = a, b = b)
println("The sum of $a and $b is $sumResult")
```

Figure 1.17 – Implementation of a 'sum' function

According to a figure we provide two arguments into the function which sum them and returns a result. The result of function is set into a variable. Figure 1.18 shows the results the 'println'.



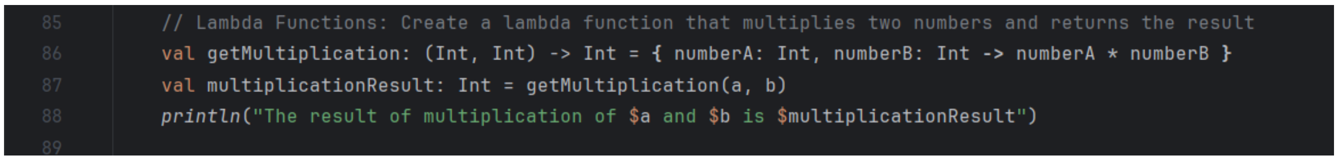
```
The sum of 2 and 5 is 7
```

Figure 1.18 – Results of 'println' function of the summation

So, the result is correct as the summation of 2 and 5 equals 7.

3.2 Lambda functions

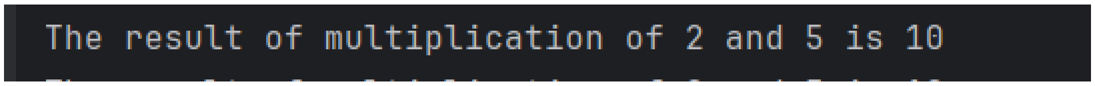
Lambda function is a small, anonymous **function** that take any number of arguments but only have one expression. To cover a task, we need to create a lambda function that multiplies two numbers and returns the result. Figure 1.19 contains the code of a lambda function.



```kotlin
// Lambda Functions: Create a lambda function that multiplies two numbers and returns the result
val getMultiplication: (Int, Int) -> Int = { numberA: Int, numberB: Int -> numberA * numberB }
val multiplicationResult: Int = getMultiplication(a, b)
println("The result of multiplication of $a and $b is $multiplicationResult")
```

Figure 1.19 – Implementation of a lambda function for multiplication

Thus, we create a lambda function using special notation, take two parameters that are our numbers and return immediately the result of their multiplication. The results of the multiplication could be seen in Figure 1.20



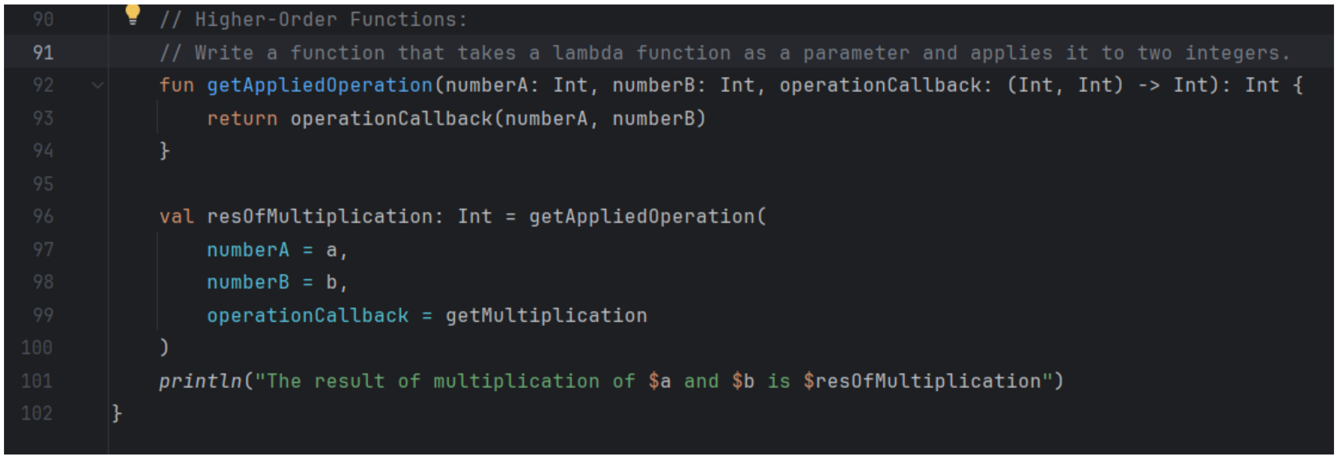
```
The result of multiplication of 2 and 5 is 10
```

Figure 1.20 – Results of a multiplication using a lambda function

So, the result is correct as the multiplication of 2 and 5 equals 10.

3.3 Higher-order functions

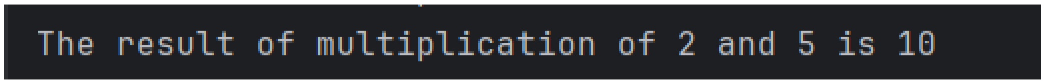
In a context of 'high-order functions' we want to say that they use at least one function which was sent via parameters as a callback one (anonymous). To show an example we need to create a function that takes a lambda function as a parameter and applies it to two integers. Figure 1.21 demonstrates the results of a code.



```
    // Higher-Order Functions:
    // Write a function that takes a lambda function as a parameter and applies it to two integers.
    fun getAppliedOperation(numberA: Int, numberB: Int, operationCallback: (Int, Int) -> Int): Int {
        return operationCallback(numberA, numberB)
    }

    val resOfMultiplication: Int = getAppliedOperation(
        numberA = a,
        numberB = b,
        operationCallback = getMultiplication
    )
    println("The result of multiplication of $a and $b is $resOfMultiplication")
}
```

Figure 1.21 – Implementation of a high-order function with multiplication as a callback

The results of function could be seen in Figure 1.22.



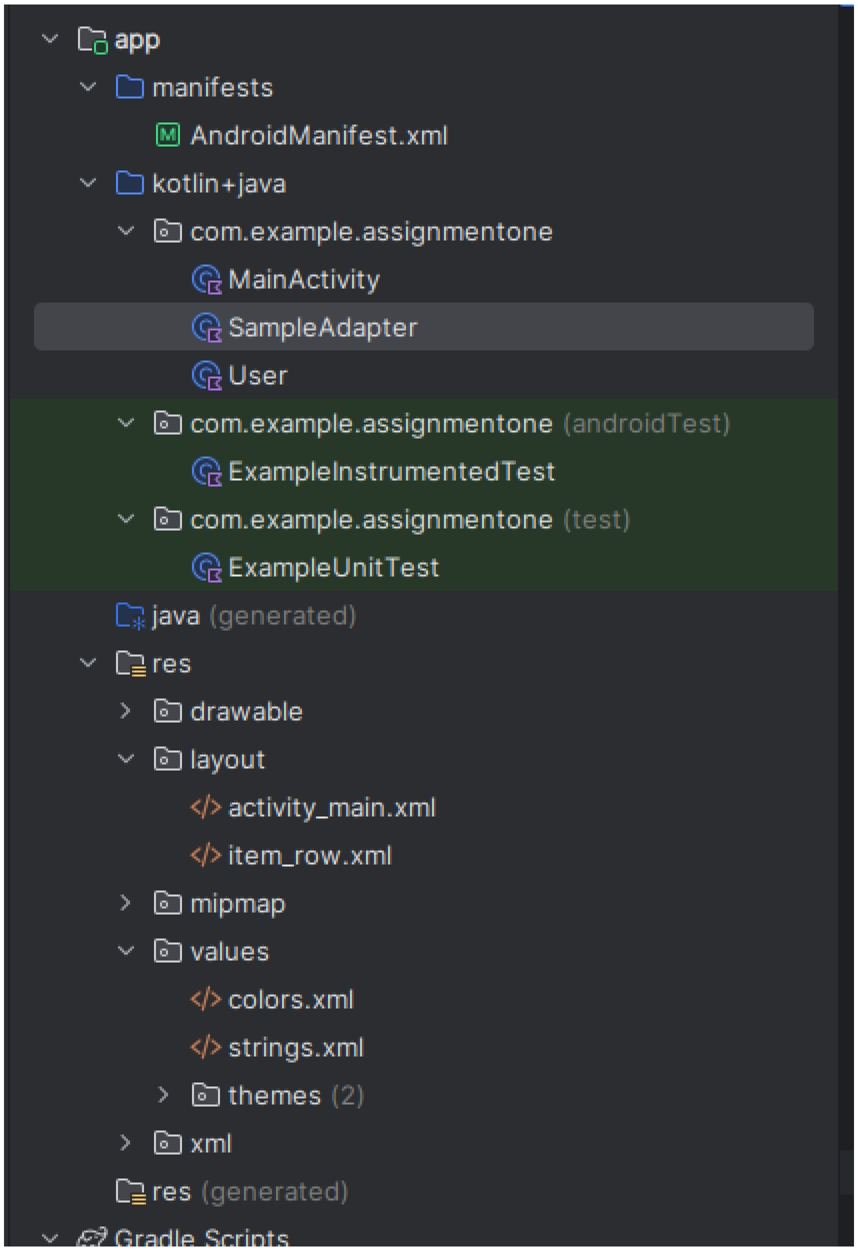
```
The result of multiplication of 2 and 5 is 10
```

Figure 1.22 – Results of a multiplication using callback function

Thus, the result maintains the same as it was in an example above.

4. Android Layout in Kotlin (Instagram-like Layout)

As we finished with basic introduction to a Kotlin and its syntax in base of a language, we can go to an area where this language is applied, for apps development. In this exercise we need to create an app with a layout like Instagram using the following components (views): ‘ImageView’, ‘TextView’ and ‘RecyclerView’. For this purpose we created another project which explorer tree is shown in Figure 1.23.



```
app
  manifests
    AndroidManifest.xml
  kotlin+java
    com.example.assignmentone
      MainActivity
      SampleAdapter
      User
    com.example.assignmentone (androidTest)
      ExampleInstrumentedTest
    com.example.assignmentone (test)
      ExampleUnitTest
  java (generated)
  res
    drawable
    layout
      activity_main.xml
      item_row.xml
    mipmap
    values
      colors.xml
      strings.xml
      themes (2)
    xml
  res (generated)
Gradle Scripts
```

Figure 1.23 – Project tree of a Instagram like app

This Android project has an "AndroidManifest.xml" file in the "manifests" folder. In the "kotlin+java" folder, there are three files: "MainActivity", "SampleAdapter", and "User". There are also test files: "ExampleInstrumentedTest" in the "androidTest" folder and "ExampleUnitTest" in the "test" folder. The "java (generated)" and "res (generated)" folders hold generated files. The "res" folder has a "drawable" folder for images, "layout" for screens with "activity_main.xml" and "item_row.xml", "mipmap" for app icons, "values" for app colors, text, and themes with "colors.xml", "strings.xml", and "themes.xml", and an "xml" folder for other settings. Finally, there are "Gradle Scripts" for building the project.

Project has two xml files: ‘activity_main.xml’ and ‘item_row.xml’. The last one is for ‘RecyclerView’ to set its appearance there of an individual row, while the first one combines all the project’s layout there. Figure 1.24 illustrates the content of a ‘item_row.xml’ file.

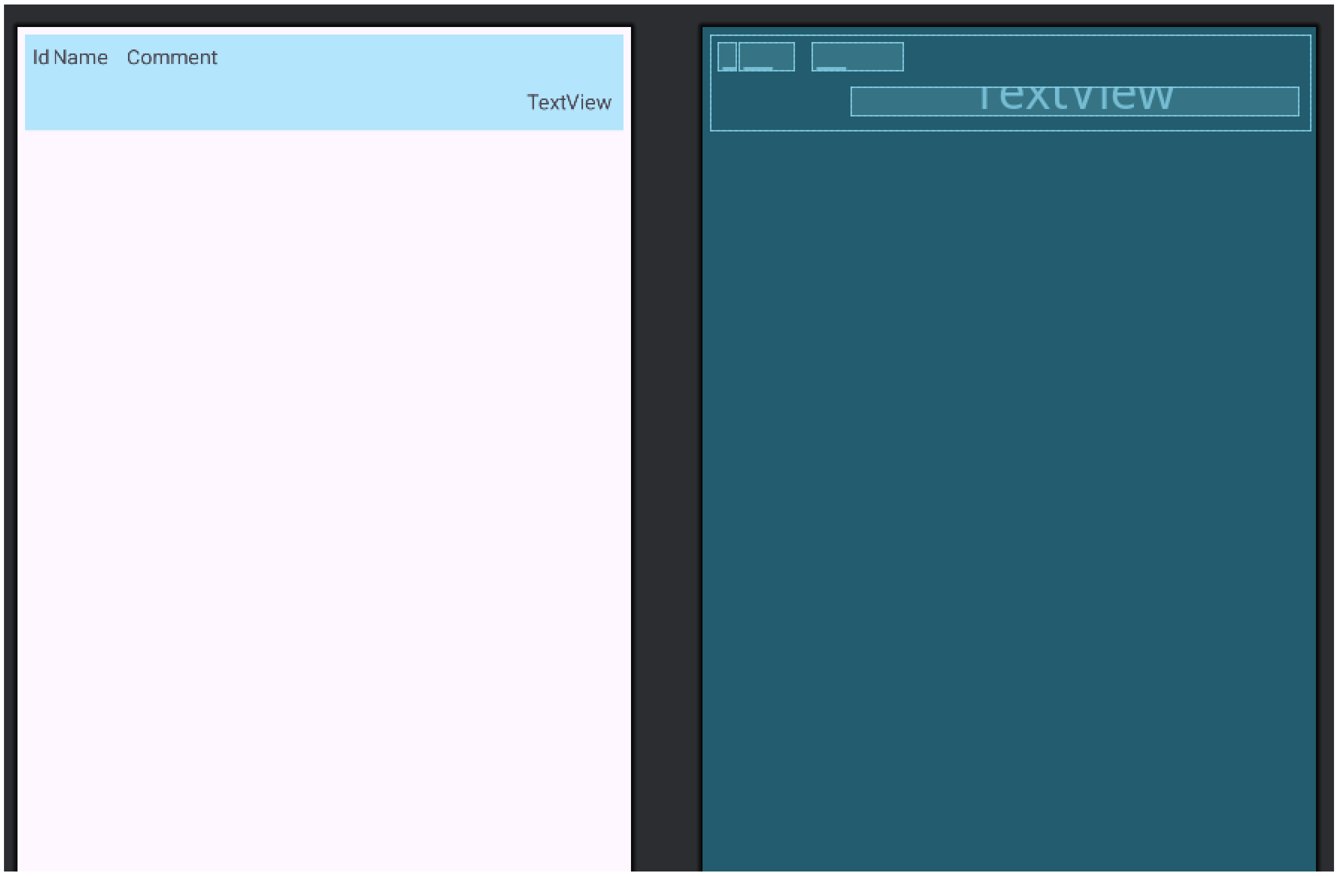


Figure 1.24 – The content of th ‘items_row.xml’ file

According to this figure each recycler view’s item contains information about use’s id, name, comment that was left and a response of an owner to this comment if it is present. The content of the ‘activity_main.xml’ could be viewed in Figure 1.25 with all the necessary views that were supplied as requirements. The final results will be provided in the end.

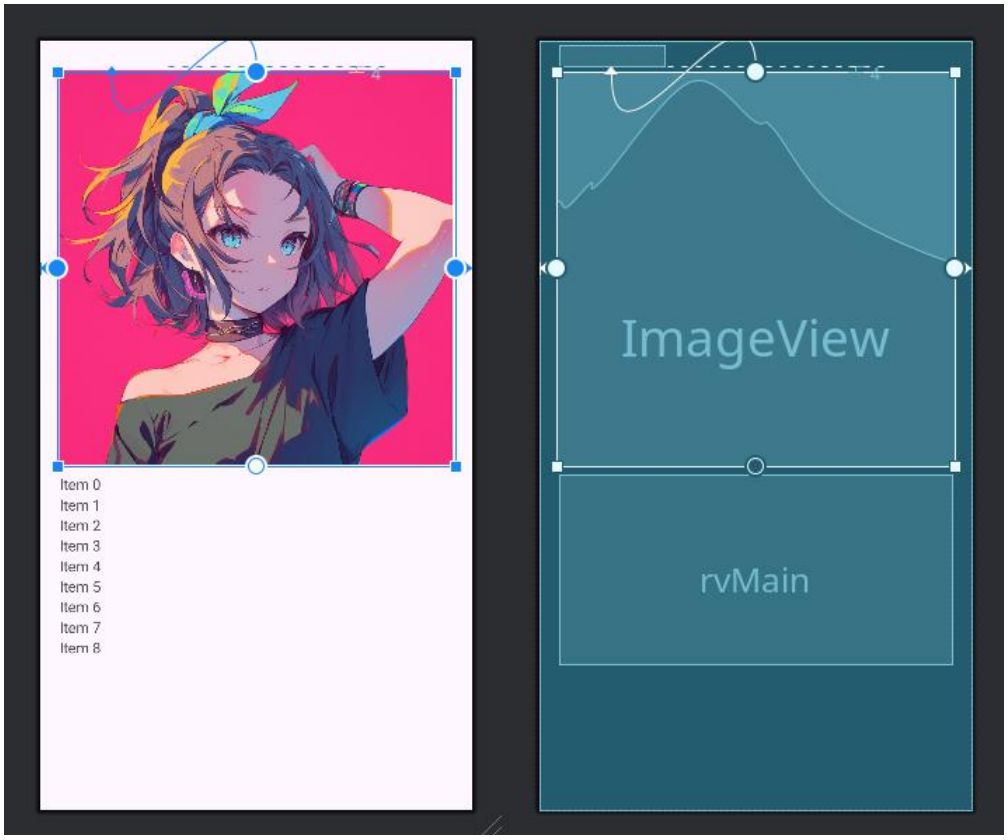
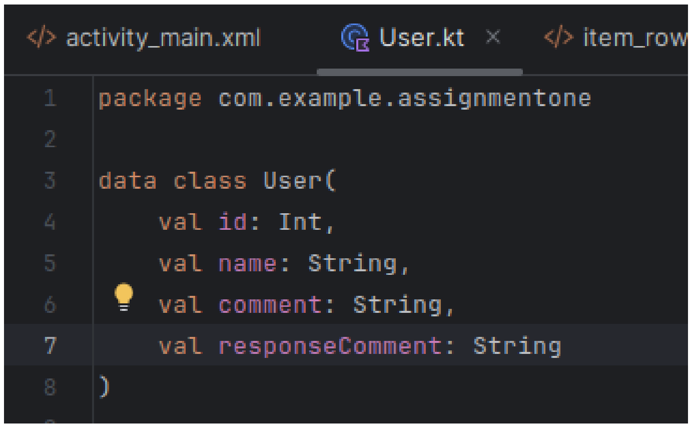


Figure 1.24 – The content of the 'activity_main.xml'

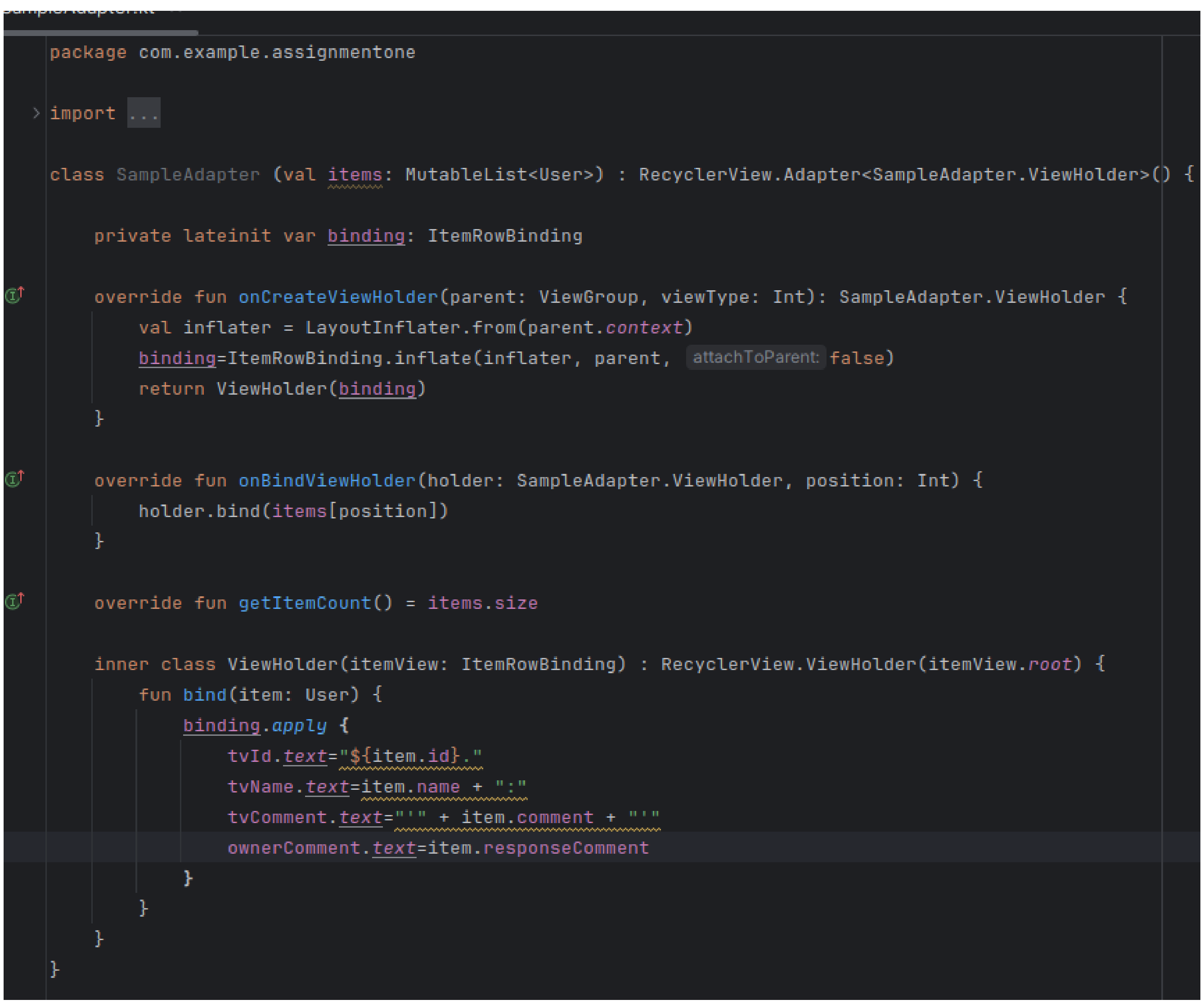
It can be clearly seen from the right part that it contains 'ImageView', 'RecyclerView' and 'TextView'. Next, it was decided to create a model for Comment to follow an OOP principle. Figure 1.25 represents model.



```
package com.example.assignmentone

data class User(
    val id: Int,
    val name: String,
    val comment: String,
    val responseComment: String
)
```

Figure 1.25 – Model for User

User model contains the following: id (Int), name (String), comment (String), responseComment (String). The last own is an owner's response message if it is present. The adapter to a Recycle View is represented in Figure 1.26.



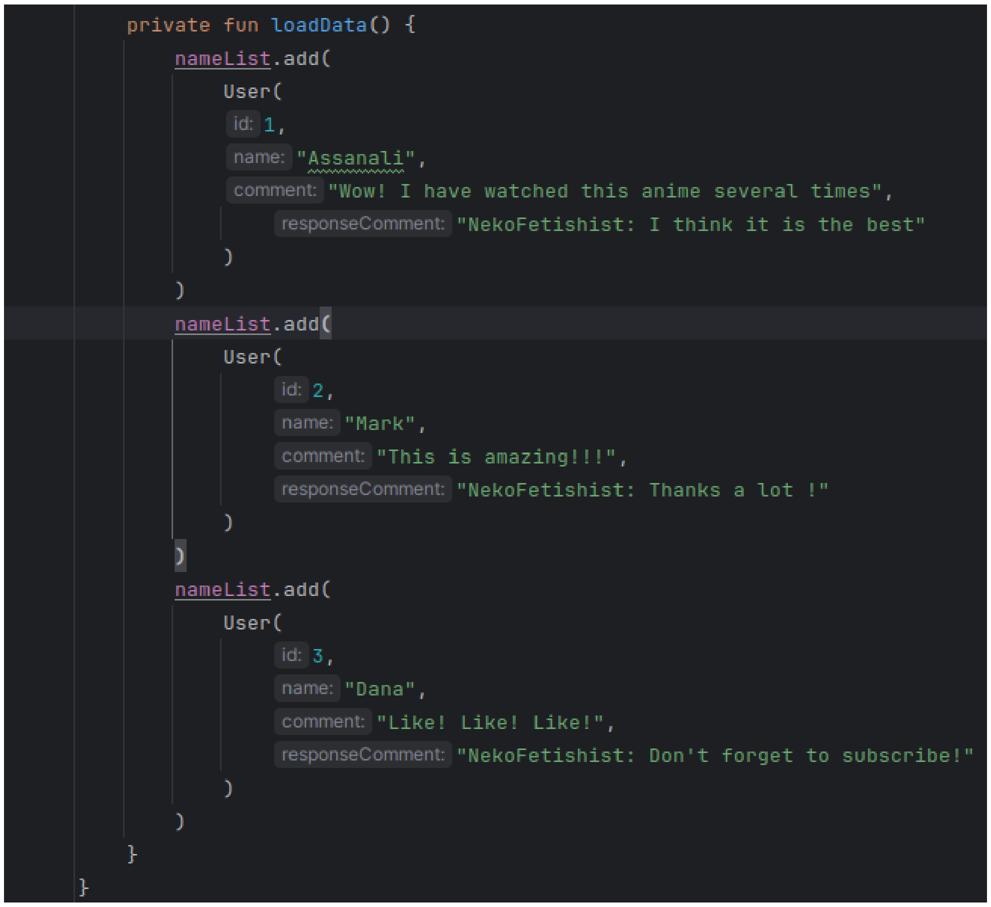
```
package com.example.assignmentone

import ...

class SampleAdapter (val items: MutableList<User>) : RecyclerView.Adapter<SampleAdapter.ViewHolder>() {

    private lateinit var binding: ItemRowBinding

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): SampleAdapter.ViewHolder {
        val inflater = LayoutInflater.from(parent.context)
        binding=ItemRowBinding.inflate(inflater, parent,  attachToParent: false)
        return ViewHolder(binding)
    }

    override fun onBindViewHolder(holder: SampleAdapter.ViewHolder, position: Int) {
        holder.bind(items[position])
    }

    override fun getItemCount() = items.size

    inner class ViewHolder(itemView: ItemRowBinding) : RecyclerView.ViewHolder(itemView.root) {
        fun bind(item: User) {
            binding.apply {
                tvId.text="${item.id}."
                tvName.text=item.name + ":"
                tvComment.text="'" + item.comment + "'"
                ownerComment.text=item.responseComment
            }
        }
    }
}
```

Figure 1.26 – Adapter for a Recycle View

The class 'SampleAdapter' takes a list of 'User' items as input. This list is used to display the data in the 'RecyclerView'. A private binding variable of type 'ItemRowBinding' is declared. This is used to bind the layout of each item in the RecyclerView. The method 'onCreateViewHolder' is called when a new 'ViewHolder' is created. A 'LayoutInflater' inflates the 'ItemRowBinding' layout, which is passed to the 'ViewHolder'.

The parameter 'attachToParent' is set to false, meaning the inflated layout will not be immediately added to the parent view. The method 'onBindViewHolder' is called for each item in the list to bind the data to the views. It retrieves the corresponding ‘User’ item from the items list based on the position and binds it using the bind method in the ‘ViewHolder’. ‘ViewHolder’ is a inner class that defines a ‘ViewHolder’, which holds references to the views in each list item. ‘bind’ is a method that is responsible for binding the data of a ‘User’ item to the views. It uses ‘binding.apply’ to set the values: ‘tvId’ (id of a user), ‘tvName’ (name of a user), ‘tvComment’ (user’s comment message), ‘ownerComment’ (owner’s response message to a comment). So, generally, adapter is used to display a list of ‘User’ items in a ‘RecyclerView’ component.

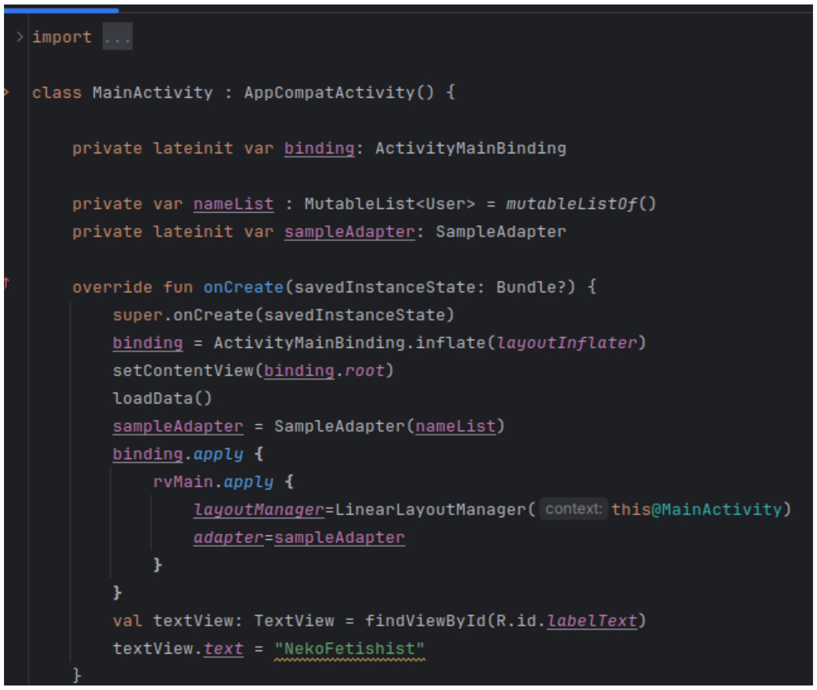
Test data loading for this exercise are hold in Figure 1.27.



```
    private fun loadData() {
        nameList.add(
            User(
                id: 1,
                name: "Assanali",
                comment: "Wow! I have watched this anime several times",
                    responseComment: "NekoFetishist: I think it is the best"
            )
        )
        nameList.add(
            User(
                id: 2,
                name: "Mark",
                comment: "This is amazing!!!",
                responseComment: "NekoFetishist: Thanks a lot !"
            )
        )
        nameList.add(
            User(
                id: 3,
                name: "Dana",
                comment: "Like! Like! Like!",
                responseComment: "NekoFetishist: Don't forget to subscribe!"
            )
        )
    }
}
```

Figure 1.27 – Test data loading of ‘User’ instances

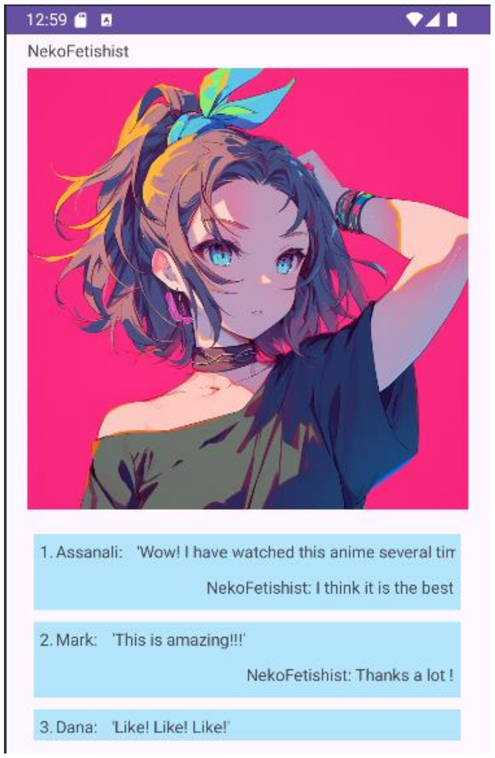
There are only three users who left comments under the post’s photo in a list. We decided to use this amount as a demonstration of a working process. The last thing before the project’s launch was to combine all the parts into one activity (file) to feed data along with the template. Figure 1.29 contains the adapter, generated test data and binding to a root xmls to bind all the references (‘TextView’, ‘RecyclerView’) by IDs. Image view is set as a static image.



```
import ...

class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding

    private var nameList : MutableList<User> = mutableListOf()
    private lateinit var sampleAdapter: SampleAdapter

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)
        loadData()
        sampleAdapter = SampleAdapter(nameList)
        binding.apply {
            rvMain.apply {
                layoutManager=LinearLayoutManager( context: this@MainActivity)
                adapter=sampleAdapter
            }
        }
        val textView: TextView = findViewById(R.id.labelText)
        textView.text = "NekoFetishist"
    }
```

Figure 1.29 – Main activity code

The results after launching the project are shown in Figure 1.30.



12:59

NekoFetishist

1. Assanali: 'Wow! I have watched this anime several tir

NekoFetishist: I think it is the best

2. Mark: 'This is amazing!!!'

NekoFetishist: Thanks a lot !

3. Dana: 'Like! Like! Like!'

Figure 1.30 – The results after launching the project

So, it contains all the comments, the username of an owner, his responses to the comments and the image.

CONCLUSION

In conclusion, we managed to build a mobile Instagram like app in Kotlin. Moreover, we used ‘.xml’ files as activities, OOP principles, etc.