“UNIVERSIDAD DE LAS FUERZAS ARMADAS”
ESPE

OBJECT ORIENTED PROGRAMMING
GROUP 7
JADIST Devs

Topic:

Object Oriented Programming in Kotlin

MEMBERS:
STEVEN POZO
JAIRO QUILUMBAQUIN
DIEGO QUIMBIULCO

TUTOR:
PHD EDISON LASCANO

NRC:
4680

DATE:
July 13,2022

SEDE MATRIZ SANGOLQUÍ
QUITO-ECUADOR
2022

1. OBJECT ORIENTED PROGRAMMING IN KOTLIN

Kotlin was created by JetBrains in 2016, it is a programming language that can run on JVM, JavaScript and LLVM, being interoperable with Java code.

1.1 Declaration of a class:

Declare a class in Kotlin the word class is used followed by the name of the class. Class attributes are declared with var or val, this being the variable.

- we can have functions inside classes.
- Init is used to set the initial value of variables.
- You can have multiple constructors.

```
3   class Concessionaire {
4       var user:String;
        var password:String;

        //otra manera de inicializar la variable las variables
        init {
9           user=""
10          password=""
11      }
12
13      //otra manera de tener un constructor
14
15      constructor(user:String,password:String){
16          var user=user;
17          var password=password;
18      }
19
20      override fun toString(): String {
21          return "Concessionaire(user='$user', password='$password')"
22      }
23
24      //FUNCIONES
25      fun logIn (){
26          val scanner = Scanner(System.`in`)
27          var userName:String;
28          var password:String;
29          var user:String="hhh";
30          var password1:String="1234";
```

Class name

Attributes with var

Start variable

Constructor

Function

2. **Instantiate a class:**

The reserved word "new" is not needed as in Java, after having the object of the class the methods of the same are called

```
fun main(args: Array<String>) {

    val admin=Concessionaire();
    val customer = Customer(name="",id="", phoneNumber = "", email = "");
    val car=Vehicle(treademark="", model = "", color = "", price =0F);

    admin.logIn();
    customer.addCustomer();
    println(customer.toString());
    println("Car's catalogue");
    admin.showCatalogue();

}
```

Instantiate teh class

```
val instancia = Clase()
```

Method

3. **Access modifiers:**

- ***Public:*** anyone can access this.
- ***Private:*** can only be accessed from the class.
- ***Internal:*** Anyone who can access the class can access the property.
- ***Protected:*** Subclasses of this class can access this class.

4. **Constructor**

When there are several constructors within a class they are defined with the reserved word "constructor", also within the same class the property and the constructor can be declared.

```
class Vehicle constructor(treademark:String, model:String, color:String, price: Float){
    var treademark:String
    var model:String
    var color:String
    var price:Float

    init {


        this.treademark = treademark;
        this.model =model;
        this.color =color;
        this.price=price;

    }
```

```
6   class Customer{
7       var name:String;
8       var id:String;
9       var phoneNumber:String;
10      var email:String
11
12      init {
13          this.name ="";
14          this.id="";
15          this.phoneNumber="";
16          this.email="";
17      }
18      constructor(name:String,id:String,phoneNumber:String,email:String){
19
```

4.1 Several constructors

There can be more constructors and they must call the primary constructor.

Primary Constructor

Secondary Constructor

```
8   class Mascota(var nombre: String, var edad: Int) {
9
10      constructor(nombre: String) : this(nombre, 0) {
11          // Opcionalmente aquí el cuerpo u otras acciones
12      }
```

5. Getter and Setter

The getter and setter in Kotlin inherit the same access as properties. So, if a property is defined as private the getter and setter will also be private and this cannot be changed.

Are defined by default:

```
var nombre: String
```

```
mascota.nombre = "Otro nombre"

var elNombre = mascota.nombre
```

There is no need to call functions like ***getName*** and ***setName***, just access or set the property.

Encapsulation

OOP encapsulation in Kotlin is enforced and has some fine grained levels (scope modifiers/keywords) which are private,public,Internal,and protected.

To encapsulate the fields we must create attributes that are declared with var or val, this being the variable,and declare it if it is private.Also the setters and getters,that are the methods which are public.

```
Main.kt
class Vehicle{

    //Create private var
    private var tipo:String = ""
    private var marca:String = ""

    //setters and getters
    public fun getTipo():String
    {
        return this.tipo
    }
    public fun setTipo(tipo:String)
    {
        this.tipo = tipo
    }

    public fun getMarca():String
    {
        return this.marca
    }
    public fun setMarca(marca:String)
    {
        this.marca = marca
    }
}
```

In the same class vehicle we create main method and create vehicle object,and call the method that are encapsulated.

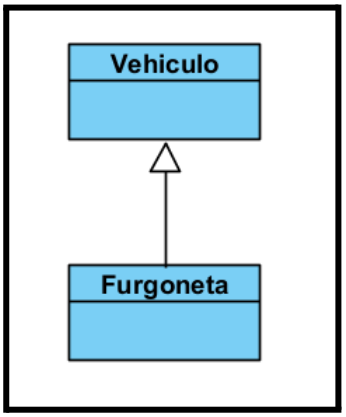
```kotlin
//create main method and create vehicle object
fun main(args: Array<String>){
    var vehicle = Vehicle()
    vehicle.setTipo("Sedan")
    vehicle.setMarca("Nissan")

    println("The type of the vehicle is: ${vehicle.getTipo()}" )
    println("The manufacturer of the vehicle is: ${vehicle.getMarca()}" )

}
```

Inheritance

In Kotlin, the inheritance mechanism can be created with **:** notation followed with the parent class name. In this example, there are two classes called Vehiculo and Furgoneta. The Vehiculo class is a parent class when the Furgoneta class is a sub class from Vehiculo class. The relation between Vehiculo class and Furgoneta class is illustrated in this picture.



The Vehiculo class is created inside the Vehiculo.kt file.

```kotlin
open class Vehiculo (val marca:String, val tipo: String) {
    open fun comprar(){
        println("Comprar vehiculo")

    }
}
```

The Furgoneta class is created inside the Furgoneta.kt file.

```kotlin
// create a Furgoneta class that inherits Vehiculo class
class Furgoneta(val color:String, marca:String, tipo:String): Vehiculo(marca,tipo){
    // create a specific implementation for comprar() method
    // in Furgoneta class
    override fun comprar() {
        super.comprar()
        println("Comprar vehiculo con las siguientes caracteristicas:")
        println("Color del vehiculo: $color")
        println("Marca del vehiculo: $marca")
```

We also create an Client class for the data of the client

```
Main.kt   Vehiculo.kt   Furgoneta.kt   Cliente.kt
1   data class Cliente(
2
3       val nombre:String,
4       val email:String,
5   )
6
```

The object from Furgoneta class is created in main() method.

```
Main.kt   Vehiculo.kt   Furgoneta.kt   Cliente.kt
1   fun main() {
2       //We create the client data and create the client object
3       val cliente = Cliente( nombre: "Jhon Spencer", email: "jhon_spencer@gmail.com")
4       println("Datos primarios del cliente:")
5       println("Nombre del Cliente: ${cliente.nombre}")
6       println("E-mail del Cliente: ${cliente.email}")
7
8       // create an object from Furgoneta class
9       val furgoneta = Furgoneta( color: "Rojo", marca: "Volkswagen", tipo: "Van")
10      // call comprar() method
11      furgoneta.comprar()
12  }
```

Output

```
MainKt
"C:\Program Files\Java\jdk-17.0.1\bin\java.exe" "-javaagent:C:\P
Datos primarios del cliente:
Nombre del Cliente: Jhon Spencer
E-mail del Cliente: jhon_spencer@gmail.com
Comprar vehiculo
Comprar vehiculo con las siguientes caracteristicas:
Color del vehiculo: Rojo
Marca del vehiculo: Volkswagen
Tipo del Vehiculo: Van
```

Polymorphism and Abstraction

Sealed class is an abstract class so the object cannot be created or instantiated from sealed class.In this example, the sealed class is created inside Vehicle.kt file.

```
Main.kt   Vehicle.kt
class Vehicle {
    // create a sealed class called Vehicle
    sealed class Vehicle
    // create two data classes that inherit Vehicle class
    data class Motorcycle(val manufacturer: String, val type: String): Vehicle()
    data class Car(val manufacturer: String, val type: String): Vehicle()

    // create a run()  method that can be called
    // from a class that inherits Vehicle class
    fun Vehicle.run() {
        // define the run() implementation
        // based on specific class
        when(this) {
            is Motorcycle -> println("Running with two wheels")
            is Car -> println("Running with four wheels")
        }
    }
}
```

Create an object from Motorcycle class that inherits sealed class called Vehicle.

```
Vehicle.kt   Main.kt
fun main() {
    // create an object from Motorcycle class
    val motorcycle = Motorcycle( manufacturer: "Honda", type: "CBR")
    // call run() method
    motorcycle.run()
}
```

Output

```
MainKt
"C:\Program Files\Java\jdk-17.0.1\bin\java.exe" "-javaagent:C:\Program
Running with two wheels

Process finished with exit code 0
```

Abstract classes and methods

An abstract method in kotlin it's easy to implement, to explain implementation of this kinf of methods we create an abstract class called CoffeMachine as we can see in the figure:

```
abstract class CoffeMachine {

    abstract fun heatWater()
    abstract fun addCoffe()

}
```

in this class we have been created two abstract methods called heatWater and addCoffe following the same concepts of abstract classes as java, implementation of this methods is done creating a new class called PremiunCoffe which implements and overriding the abstract methods:

```
class PremiumCoffeeMachine : CoffeMachine() {
    override fun heatWater() {
        println("Heating Water")
    }

    override fun addCoffe() {
        println("adding Coffee")
    }

}
```

The principal difference between kotlin and java are the use of keywords, in java to specify that a class has inheritance from another class we use the keyword **extends** but in kotlin we just set the name of the superclass and follow the same process to implement the different methods.

```
fun main() {
    val orderCoffe: PremiumCoffeeMachine=PremiumCoffeeMachine()

    orderCoffe.heatWater()
    orderCoffe.addCoffe()

}
```

```
MainKt
"C:\Program Files\Java\jdk-18.0.1\bin\java.exe" ...
Heating Water
adding Coffee

Process finished with exit code 0
```

Finally we have created an object of the child class Premium Coffee and test if the overwritten functions are working right.

Interfaces

An interface is a kind of class which allows us to create a complete abstract class without attributes just methods, in kotlin we create this kind of class using "interface" keyword as is shown below:

```
interface Person {
    fun run()
    fun breathe()
    fun sleep()
}
```

As java we can't instantiate objects of this kind of classes but we can use the methods making use of the inheritance between classes so we have created a child class called Athlete who implements all the method of his parent class:

```
class Athlete: Person {
    override fun run() {
        println("running")
    }

    override fun breathe() {
        println("breathing")
    }

    override fun sleep() {
        println("sleeping")
    }

}
```

Finally in our main class we test if the implementation of the interface was done right.

```
fun main() {
    val mike: Athlete=Athlete()

    mike.breathe()
    mike.run()
    mike.sleep()


}
```

```
MainKt
"C:\Program Files\Java\jdk-18.0.1\bin\java.exe" ...
breathing
running
sleeping

Process finished with exit code 0
```

.

As we can see the implementation of the interfaces was made successfully.