LECTURE 10: KOTLIN OOP - KOTLIN CLASS AND OBJECTS

BY LINA HAMMAD & AHMAD BARGHASH

In this lecture, you'll be introduced to Object-oriented programming in Kotlin. You'll learn what a class is, how to create objects and use it in your program. Also, you'll learn about inheritance. More specifically, what is inheritance and how to implement it in Kotlin (with the help of examples).

KOTLIN CLASS AND OBJECTS – OBJECT ORIENTED PROGRAMMING (OOP)

- Kotlin is an object-oriented programming language
- Kotlin supports pillars of OOP language such as inheritance.
- Classes are the main building blocks of any object-oriented programming language.
- Kotlin classes are declared using keyword **class**. Kotlin class has a class header which specifies its type parameters, constructor etc. and the class body which is surrounded by curly braces.
- Syntax of Kotlin class declaration:

```
class MyClass {
    // variables or data members or ptoperty
    // member functions or methods
    ..
    ..
}
```

PROPERTIES AND METHODS

- Variables inside a class are called **Properties (Data members)**.
- A function inside a class is called a **Method (Member function)**.

```
class Example {

    // data member
    var number: Int = 5

    // member function
    fun calculateSquare(): Int {
        return number*number
    }
}
```

KOTLIN CLASS EXAMPLE 2

```
class Lamp {

    // property (data member)
    var isOn: Boolean = false

    // member function (method)
    fun turnOn() {
        isOn = true
    }

    // member function
    fun turnOff() {
        isOn = false
    }
}
```

- Here, we defined a class named Lamp.
- The class has one Property isOn (defined in same way as variable), and two Methods turnOn() and turnOff().
- Make sure you initialize the property once declared.

KOTLIN OBJECTS

- To access members defined within the class, you need to create objects.
- **Object** is real time entity which has state and behavior.
- Kotlin allows to create multiple objects of a class.

KOTLIN CLASS EXAMPLE 1

- We can access the property and the data members of the class using the object as the following:

```
class Example {

    // data member
    var number: Int = 5

    // member function
    fun calculateSquare(): Int {
        return number*number
    }
}

fun main(){
    val e1 = Example()
    val e2 = Example()

    //object 1
    var result = e1.calculateSquare()
    println("result = $result")

    //object2
    e2.number = 6
    result = e2.calculateSquare()
    println("result = $result")
}
```

The output is:

```
result = 25

result = 36
```

KOTLIN CLASS EXAMPLE 2 WITH A SMALL MODIFICATION

```
class Lamp {
    var isOn: Boolean = false // property (data member)

    // member function
    fun turnOn() {
        isOn = true
    }

    // member function
    fun turnOff() {
        isOn = false
    }

    fun displayLightStatus(lamp: String) {
        if (isOn == true)
            println("$lamp lamp is on.")
        else
            println("$lamp lamp is off.")
    }
}
```

```
fun main(args: Array<String>) {
    val l1 = Lamp() // create l1 object of Lamp class
    val l2 = Lamp() // create l2 object of Lamp class

    l1.turnOn()
    l2.turnOff()

    l1.displayLightStatus("l1")
    l2.displayLightStatus("l2")
}
```

The output is:

```
l1 lamp is on.
l2 lamp is off.
```

DETAILED EXPLANATION

- In the previous program,
  - Lamp class is created.
  - The class has a property isOn and three member functions turnOn(), turnOff() and displayLightStatus().
  - Two objects L1 and L2 of Lamp class are created in the main() function.
  - Here, turnOn() function is called using L1 object: L1.turnOn(). This method sets isOn instance variable of L1 object to true.
  - And, turnOff() function is called using L2 object: L1.turnOff(). This method sets isOff instance variable of L2 object to false.
  - Then, displayLightStatus() function is called for L1 and L2 objects which prints appropriate message depending on whether isOn property is true or false.
  - Notice that, the isOn property is initialized to false inside the class. When an object of the class is created, isOn property for the object is initialized to false automatically. So, it's not necessary for L2 object to call turnOff() to set isOn property to false.

ACCESS MODIFIERS

- **Private:** Can be accessed inside the class only.
- **Public**: Can be accessed everywhere.
- **Protected**: Can be accessed to the class and its subclasses.

- If you do not specify the visibility modifier, it will be public by default.
- Access modifiers are specified before the class, properties, member function keyword.

```
private class MyClass {

}
```

```
class MyClass {
    private var num1 = 10
    protected var num2 = 20
    public  var num3 = 30
}
```

```
public class MyClass {
    private var num1 = 10
    public fun printClassName(){
        println ("Hello from MyClass")
    }
    protected fun printNum(){
        println ("num1 = $num1")
    }
}
```

KOTLIN INHERITANCE

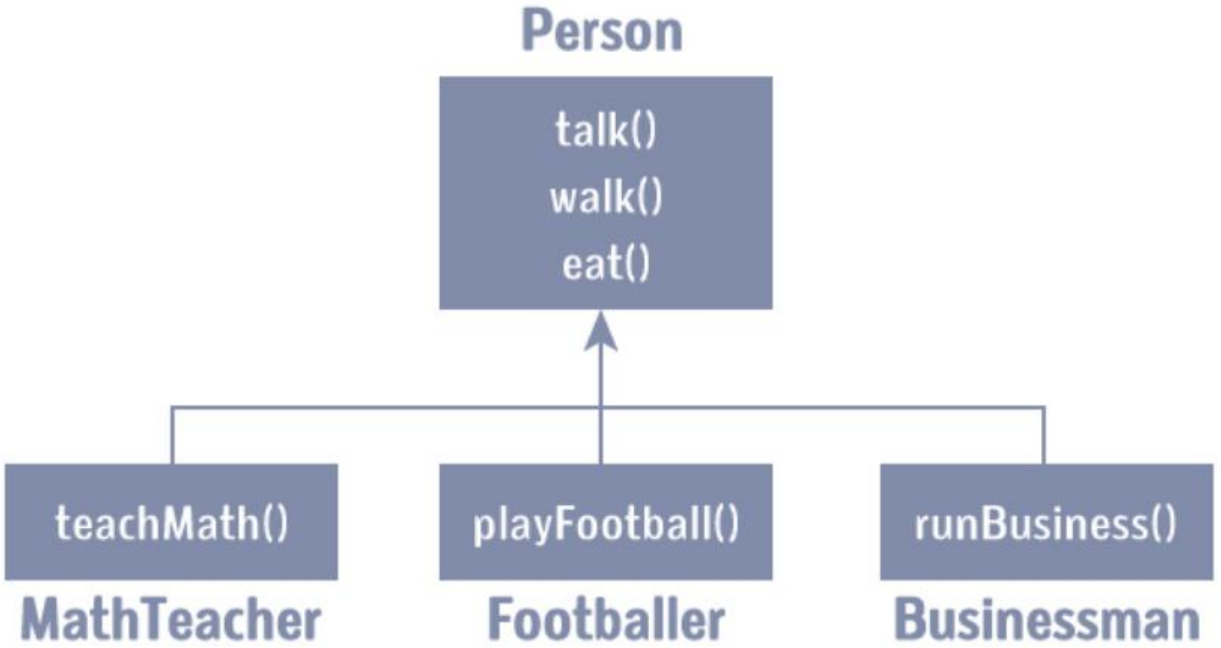
- Inheritance is one of the key features of object-oriented programming. It allows user to create a new class (derived class, child class, or sub class) from an existing class (base class, parent class or super class).
- The derived class inherits all the features from the base class and can have additional features of its own.

WHY INHERITANCE?

- Suppose in your application, you want three characters - a math teacher, a footballer and a businessman.
- Since, all of the characters are persons, they can walk and talk. However, they also have some special skills. A math teacher can teach math, a footballer can play football and a businessman can run a business.
- You can individually create three classes who can walk, talk and perform their special skill.

WHY INHERITANCE?

- In each of the classes, you would be copying the same code for walk and talk for each character.
- If you want to add a new feature - eat, you need to implement the same code for each character. This can easily become error prone (when copying) and duplicate codes.
- It would be a lot easier if we had a Person class with basic features like talk, walk, eat, sleep, and add special skills to those features as per our characters. This is done using inheritance.



WHY INHERITANCE?

- Using inheritance, now you don't implement the same code for walk(), talk() and eat() for each class. You just need to inherit them.
- So, for MathTeacher (derived class), you inherit all features of a Person (base class) and add a new feature teachMath(). Likewise, for the Footballer class, you inherit all the features of the Person class and add a new feature playFootball() and so on.
- This makes your code cleaner, understandable and extendable.
- It is important to remember: When working with inheritance, each derived class should satisfy the condition whether it "is a" base class or not. In the example above, MathTeacher is a Person, Footballer is a Person. You cannot have something like, Businessman is a Business.

KOTLIN INHERITANCE

```
open class Person {
    // code for eating, talking, walking
}

class MathTeacher: Person() {
    // other features of math teacher
}

class Footballer: Person() {
    // other features of footballer
}

class Businessman: Person() {
    // other features of businessman
}
```

- Here, Person is a base class, and classes MathTeacher, Footballer, and Businessman are derived from the Person class.
- Notice, the keyword **open** before the base class, Person. **It's important**.

EXAMPLE: KOTLIN INHERITANCE I

```
open class Person{
        var name:String=" "
        var age:Int=0
    fun PrintInfo() {
        println("My name is $name.")
        println("My age is $age")
    }
}
class MathTeacher: Person() {
    fun teachMaths() {
        println("I teach in Ette7ad School")
    }
}
class Footballer: Person() {
    fun playFootball() {
        println("I play for Dortmond")
    }
}
```

```
fun main(args: Array<String>) {
    val t1 = MathTeacher()
    t1.name="Ali"
    t1.age=30
    t1.PrintInfo()
    t1.teachMaths()

    val f1 = Footballer()
    f1.name="Mohammad"
    f1.age=23
    f1.PrintInfo()
    f1.playFootball()
}
```

The output is:

```
My name is Ali.
My age is 30
I teach in Ette7ad School
My name is Mohammad.
My age is 23
I play for Dortmond
```

EXAMPLE: KOTLIN INHERITANCE 2

```
open class Base{
    val x = 10
}
class Derived: Base() {
    fun foo() {
        println("x is equal to " + x)
    }
}
fun main(args: Array<String>) {
    val derived = Derived()
    derived.foo()
}
```

The output is:

```
x is equal to 10
```

OVERRIDING MEMBER FUNCTION EXAMPLE

- If the base class and the derived class contains a member function (or property) with the same name, you need to **override** the member function of the derived class using override.

```
// Empty primary constructor
open class Person() {
    open fun displayAge(age: Int) {
        println("My age is $age.")
    }
}
class Girl: Person() {
    override fun displayAge(age: Int) {
        println("My fake age is ${age - 5}.")
    }
}
fun main(args: Array<String>) {
    val girl = Girl()
    girl.displayAge(31)
}
```

The output is:

```
My fake age is 26.
```

OVERRIDING MEMBER FUNCTION EXAMPLE

```
open class Animal() {
    open fun sound() {
        println("Animal makes a sound")
    }
}

class Dog: Animal() {
    override fun sound() {
        println("Dog makes a sound of woof woof")
    }
}

fun main(args: Array<String>) {
    val d = Dog()
    d.sound()
}
```

The output is:

```
Dog makes a sound of woof woof
```

OVERRIDING MEMBER FUNCTIONS AND PROPERTIES EXAMPLE

```
open class Animal() {
    open var colour: String = "White"
}

class Dog: Animal() {
    override var colour: String = "Black"
    fun sound() {
        println("Dog makes a sound of woof woof")
    }
}

fun main(args: Array<String>) {
    val d = Dog()
    d.sound()
    println("${d.colour}")
}
```

The output is:

```
Dog makes a sound of woof woof
Black
```

CALLING MEMBERS OF BASE CLASS FROM DERIVED CLASS

- You can call functions (and access properties) of the base class from a derived class using **super** keyword. Here's how:

```
open class Person() {
    open fun displayAge(age: Int) {
        println("My actual age is $age.")
    }
}
class Girl: Person() {
    override fun displayAge(age: Int) {
        // calling function of base class
        super.displayAge(age)

        println("My fake age is ${age - 5}.")
    }
}
fun main(args: Array<String>) {
    val girl = Girl()
    girl.displayAge(31)
}
```

The output is:

```
My actual age is 31.
My fake age is 26.
```

CALLING MEMBERS OF BASE CLASS FROM DERIVED CLASS

```
open class Parent() {
    open var num: Int = 100
    open fun demo(){
        println("demo function of parent class")
    }
}

class Child: Parent() {
    override var num: Int  = 101
    override fun demo() {
        super.demo()
        println("demo function of child class")
    }
    fun demo2(){
        println(super.num)
    }
}

fun main(args: Array<String>) {
    val obj = Child()
    obj.demo()
    obj.demo2()
}
```

The output is:

```
demo function of parent class
demo function of child class
100
```