Kotlin - Class & Object

In this chapter, we will learn the basics of Object-Oriented Programming (OOP) using Kotlin. We will learn about class and its object and how to play with that object. By definition of OOP, a class is a blueprint of a runtime entity and object is its state, which includes both its behavior and state. In Kotlin, class declaration consists of a class header and a class body surrounded by curly braces, similar to Java.

```
Class myClass { // class Header

   // class Body
}
```

Like Java, Kotlin also allows to create several objects of a class and you are free to include its class members and functions. We can control the visibility of the class members variables using different keywords that we will learn in Chapter 10 – Visibility Control. In the following example, we will create one class and its object through which we will access different data members of that class.

Live Demo

```
class myClass {
   // property (data member)
   private var name: String = "Tutorials.point"

   // member function
   fun printMe() {
      print("You are at the best Learning website Named-"+name)
   }
}
fun main(args: Array<String>) {
   val obj = myClass() // create obj object of myClass class
   obj.printMe()
}
```

The above piece of code will yield the following output in the browser, where we are calling printMe() of myClass using its own object.

```
You are at the best Learning website Named- Tutorials.point
```

Nested Class

By definition, when a class has been created inside another class, then it is called as a nested class. In Kotlin, nested class is by default static, hence, it can be accessed without creating any object of that class. In the following example, we will see how Kotlin interprets our nested class.

Live Demo

```
fun main(args: Array<String>) {
   val demo = Outer.Nested().foo() // calling nested class method
   print(demo)
}
class Outer {
   class Nested {
      fun foo() = "Welcome to The TutorialsPoint.com"
   }
}
```

The above piece of code will yield the following output in the browser.

```
Welcome to The TutorialsPoint.com
```

Inner Class

When a nested class is marked as a “inner”, then it will be called as an Inner class. An inner class can be accessed by the data member of the outer class. In the following example, we will be accessing the data member of the outer class.

Live Demo

```
fun main(args: Array<String>) {
   val demo = Outer().Nested().foo() // calling nested class method
   print(demo)
}
class Outer {
   private val welcomeMessage: String = "Welcome to the TutorialsPoint.com"
   inner class Nested {
      fun foo() = welcomeMessage
   }
}
```

The above piece of code will yield the following output in the browser, where we are calling the nested class using the default constructor provided by Kotlin compilers at the time of compiling.

```
Welcome to the TutorialsPoint.com
```

Anonymous Inner Class

Anonymous inner class is a pretty good concept that makes the life of a programmer very easy. Whenever we are implementing an interface, the concept of anonymous inner block comes into picture. The concept of creating an object of interface using runtime object reference is known as anonymous class. In the following example, we will create an interface and we will create an object of that interface using Anonymous Inner class mechanism.

Live Demo

```
fun main(args: Array<String>) {
   var programmer :Human = object:Human // creating an instance of the interface {
      override fun think() { // overriding the think method
         print("I am an example of Anonymous Inner Class ")
      }
   }
   programmer.think()
}
interface Human {
   fun think()
}
```

The above piece of code will yield the following output in the browser.

```
I am an example of Anonymous Inner Class
```

Type Aliases

Type aliases are a property of Kotlin compiler. It provides the flexibility of creating a new name of an existing type, it does not create a new type. If the type name is too long, you can easily introduce a shorter name and use the same for future usage. Type aliases is really helpful for complex type. In the latest version, Kotlin revoked the support for type aliases, however, if you are using an old version of Kotlin you may have use it like the following −

```
typealias NodeSet = Set<Network.Node>
typealias FileTable<K> = MutableMap<K, MutableList<File>>
```