7.3 Visibility Modifiers:

Classes, objects, interfaces, constructors, functions, properties and their setters can have visibility modifiers.

There are four visibility modifiers in Kotlin:

- private,
- protected,
- internal and
- public.

- The **default** visibility, used if there is no explicit modifier, is **public**.
- ***private*** will only be visible inside the file containing the declaration.
- ***internal*** is visible everywhere in the same module;
- ***protected*** is not available for top-level declarations.

public modifier

A ***public*** modifier is accessible from everywhere in the project. It is a default modifier in Kotlin.

If any class, interface etc. are not specified with any access modifier then that class, interface etc. are used in public scope.

protected modifier

A ***protected*** modifier with class or interface allows visibility to its class or subclass only.

A protected declaration (when overridden) in its subclass is also protected modifier unless it is explicitly changed.

Example:

```
open class Base{
    protected val i = 0
}

class Derived : Base(){
    fun getValue() : Int
    {
        return i
    }
}
```

In Kotlin, protected modifier cannot be declared at top level.

Overriding of protected types

```
open class Base{
      open protected val i = 5
}
class Another : Base(){
     fun getValue() : Int
     {
              return i
     }
     override val i =10
}

fun main(args: Array<String>) {
    val a1 = Another();
    println("Value : " + a1.getValue());
}
```

internal modifier

The ***internal*** modifiers are newly added in Kotlin, it is not available in Java.

Declaring anything makes that field marked as internal field.

The internal modifier makes the field visible only inside the module in which it is implemented.

Example:

```
internal class Example{

    internal val x = 5
    internal fun getValue(){

    }
}
internal val y = 10
```

private modifier

A private modifier allows the declaration to be accessible only within the block in which properties, fields, etc. are declare.

The private modifier declaration does not allow to access the outside the scope.

A private package can be accessible within that specific file.

```
private class Example {
    private val x = 1
    private val doSomething() {
    }
}
```

Example of Visibility Modifier

```
open class Base() {
     var a = 1 // public by default
    private var b = 2 // private to Base class
    protected open val c = 3  // visible to the Base and the Derived class
    internal val d = 4 // visible inside the same module
    protected fun e() { } // visible to the Base and the Derived class
}

class Derived: Base() {
    // a, c, d, and e() of the Base class are visible
   // b is not visible
    override val c = 9 // c is protected
}

fun main(args: Array<String>) {
     val base = Base()
    // base.a and base.d are visible
   // base.b, base.c and base.e() are not visible
     val derived = Derived()
    // derived.c is not visible
}
```

Kotlin Extension Function

Kotlin ***extension function*** provides a facility to **"add"** methods to class without inheriting a class or using any type of design pattern.

The created extension functions are used as a regular function inside that class.

The extension function is declared with a prefix receiver type with method name.

Syntax:

```
fun <class_name>.<method_name>()
```

Example of extension function declaration and its use

In general, we call all methods from outside the class which are already defined inside the class.In below example, a Student class declares a method is Passed() which is called from main() function by creating the object student of Student class.

Suppose that we want to call a method (say isExcellent()) of Student class which is not defined in class. In such situation, we create a function (isExcellent()) outside the Student class as Student.isExcellent() and call it from the main() function. The declare Student.isExcellent() function is known as extension function, where Student class is known as *receiver type*.

Example:

```
class Student{
    fun isPassed(mark: Int): Boolean{
        return mark>40
    }
}
fun Student.isExcellent(mark: Int): Boolean{
    return mark > 90
}
```

```
    fun main(args: Array<String>){
         val student = Student()
         val passingStatus = student.isPassed(55)
         println("student passing status is $passingStatus")

         val excellentStatus = student.isExcellent(95)
         println("student excellent status is
                                    $excellentStatus")
    }
```

```
:Output:
student passing status is true
student excellent status is true
```

Kotlin extension function example

Let's see the real example of extension function.

In this example, we are swapping the elements of MutableList<> using swap() method.

However, MutableList<>class does not provide the swap() method internally which swap the elements of it.

For doing this we create an extension function for MutableList<> with swap() function.

The list object call the extension function (MutableList<Int>.swap(index1: Int, index2: Int):MutableList<Int>) using list.swap(0,2) function call.

The swap(0,2) function pass the index value of list inside MutableList<Int>.swap(index1: Int, index2: Int):MutableList<Int>) sxtension function.

Example:

```
fun MutableList<Int>.swap(index1: Int, index2:
Int):MutableList<Int> {

     val tmp = this[index1] // 'this' represents to the list
     this[index1] = this[index2]
     this[index2] = tmp
     return this
}
```

```
fun main(args: Array<String>) {
     val list = mutableListOf(5,10,15)
     println("before swapping the list :$list")
     val result = list.swap(0, 2)
     println("after swapping the list :$result")
}
```

```
:Output:
before swapping the list :[5, 10, 15]
after swapping the list :[15, 10, 5]
```

Extension Function as Nullable Receiver

The extension function can be defined as nullable receiver type. This nullable extension function is called through object variable even the object value is null. The nullability of object is checked using this == null inside the body.

Let's rewrite the above program using extension function as nullable receiver.

```
fun MutableList<Int>?.swap(index1: Int, index2: Int): Any {
if (this == null)
     return "null"
else{
          val tmp = this[index1] //'this' represents to the list
          this[index1] = this[index2]
          this[index2] = tmp
          return this
    }
}
fun main(args: Array<String>) {
     val list = mutableListOf(5,10,15)
     println("before swapping the list :$list")
     val result = list.swap(0, 2)
     println("after swapping the list :$result")
}
```

```
:Output:
before swapping the list :[5, 10, 15]
after swapping the list :[15, 10, 5]
```

Companion Object Extensions

A companion object is an object which is declared inside a class and marked with the companion keyword.

Companion object is used to call the member function of class directly using the class name (like static in java).

A class which contains companion object can also be defined as extension function and property for the companion object.

Example of companion object

In this example, we call a **create()** function declared inside companion object using class name (MyClass) as qualifier.

```
class MyClass {
    companion object {
        fun create():String{
            return "calls create method of companion object"
        }
    }
}
fun main(args: Array<String>){
     val instance = MyClass.create()
}
```

:Output:

```
calls create method of companion object
```

Companion object extensions example

Let's see an example of companion object extensions. The companion object extension is also being called using the *class name* as the qualifier.

```
class MyClass {
   companion object {
      fun create(): String {
         return "calling create method of companion object"
      }
   }
}
```

```
fun MyClass.Companion.helloWorld() {
        println("executing extension of companion object")
}
fun main(args: Array<String>) {
        MyClass.helloWorld() //extension function declared upon the companion object
}
```

:Output:

```
executing extension of companion object
```