Kotlin - Delegation

Kotlin supports **“delegation”** design pattern by introducing a new keyword **“by”**. Using this keyword or delegation methodology, Kotlin allows the derived class to access all the implemented public methods of an interface through a specific object. The following example demonstrates how this happens in Kotlin.

Live Demo

```
interface Base {
   fun printMe() //abstract method
}
class BaseImpl(val x: Int) : Base {
   override fun printMe() { println(x) }   //implementation of the method
}
class Derived(b: Base) : Base by b  // delegating the public method on the object b

fun main(args: Array<String>) {
   val b = BaseImpl(10)
   Derived(b).printMe() // prints 10 :: accessing the printMe() method
}
```

In the example, we have one interface “Base” with its abstract method named “printme()”. In the BaseImpl class, we are implementing this “printme()” and later from another class we are using this implementation using “by” keyword.

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

```
10
```

Property Delegation

In the previous section, we have learned about the delegation design pattern using “by” keyword. In this section, we will learn about delegation of properties using some standard methods mentioned in Kotlin library.

Delegation means passing the responsibility to another class or method. When a property is already declared in some places, then we should reuse the same code to initialize them. In the following examples, we will use some standard delegation methodology provided by Kotlin and some standard library function while implementing delegation in our examples.

Using Lazy()

Lazy is a lambda function which takes a property as an input and in return gives an instance of **Lazy<T>**, where <T> is basically the type of the properties it is using. Let us take a look at the following to understand how it works.

Live Demo

```
val myVar: String by lazy {
   "Hello"
}
fun main(args: Array<String>) {
   println(myVar +" My dear friend")
}
```

In the above piece of code, we are passing a variable “myVar” to the Lazy function, which in return assigns the value to its object and returns the same to the main function. Following is the output in the browser.

```
Hello My dear friend
```

Delegetion.Observable()

Observable() takes two arguments to initialize the object and returns the same to the called function. In the following example, we will see how to use Observable() method in order to implement delegation.

Live Demo

```
import kotlin.properties.Delegates
class User {
   var name: String by Delegates.observable("Welcome to Tutorialspoint.com") {
      prop, old, new ->
      println("$old -> $new")
   }
}
fun main(args: Array<String>) {
   val user = User()
   user.name = "first"
   user.name = "second"
}
```

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

```
first -> second
```

In general, the syntax is the expression after the “by” keyword is delegated. The **get()** and **set()** methods of the variable **p** will be delegated to its **getValue()** and **setValue()** methods defined in the Delegate class.

```
class Example {
   var p: String by Delegate()
}
```

For the above piece of code, following is the delegate class that we need to generate in order to assign the value in the variable **p**.

```
class Delegate {
   operator fun getValue(thisRef: Any?, property: KProperty<*>): String {
      return "$thisRef, thank you for delegating '${property.name}' to me!"
   }
   operator fun setValue(thisRef: Any?, property: KProperty<*>, value: String) {
      println("$value has been assigned to '${property.name} in $thisRef.'")
   }
}
```

While reading, getValue() method will be called and while setting the variable setValue() method will be called.