Kotlin - Control Flow

In the previous chapter we have learned about different types of data types available in Kotlin system. In this chapter, we will discuss different types of control flow mechanism available in the Kotlin.

If - Else

Kotlin is a functional language hence like every functional language in Kotlin **“if”** is an expression, it is not a keyword. The expression **“if”** will return a value whenever necessary. Like other programming language, **“if-else”** block is used as an initial conditional checking operator. In the following example, we will compare two variables and provide the required output accordingly.

Live Demo

```
fun main(args: Array<String>) {
   val a:Int = 5
   val b:Int = 2
   var max: Int

   if (a > b) {
      max = a
   } else {
      max = b
   }
   print("Maximum of a or b is " +max)

   // As expression
   // val max = if (a > b) a else b
}
```

The above piece of code yields the following output as a result in the browser. Our example also contains another line of code, which depicts how to use **“If”** statement as an expression.

```
Maximum of a or b is 5
```

Use of When

If you are familiar with other programming languages, then you might have heard of the term switch statement, which is basically a conditional operator when multiple conditions can be applied on a particular variable. **“when”** operator matches the variable value against the branch conditions. If it

is satisfying the branch condition then it will execute the statement inside that scope. In the following example, we will learn more about “when” in Kotlin.

Live Demo

```
fun main(args: Array<String>) {
   val x:Int = 5
   when (x) {
      1 -> print("x = = 1")
      2 -> print("x = = 2")

      else -> { // Note the block
         print("x is neither 1 nor 2")
      }
   }
}
```

The above piece of code yields the following output in the browser.

```
x is neither 1 nor 2
```

In the above example, Kotlin compiler matches the value of **x** with the given branches. If it is not matching any of the branches, then it will execute the else part. Practically, when is equivalent to multiple if block. Kotlin provides another flexibility to the developer, where the developer can provide multiple checks in the same line by providing “,” inside the checks. Let us modify the above example as follows.

Live Demo

```
fun main(args: Array<String>) {
   val x:Int = 5
   when (x) {
      1,2 -> print(" Value of X either 1,2")

      else -> { // Note the block
         print("x is neither 1 nor 2")
      }
   }
}
```

Run the same in the browser, which will yield the following output in the browser.

```
x is neither 1 nor 2
```

For Loop

Loop is such an invention that provides the flexibility to iterate through any kind of data structure. Like other programming languages, Kotlin also provides many kinds of Looping methodology, however, among them **"For"** is the most successful one. The implementation and use of For loop is conceptually similar to Java for loop. The following example shows how we can use the same in real-life examples.

Live Demo

```
fun main(args: Array<String>) {
   val items = listOf(1, 2, 3, 4)
   for (i in items) println("values of the array"+i)
}
```

In the above piece of code, we have declared one list named as "items" and using for loop we are iterating through that defined list and printing its value in the browser. Following is the output.

```
values of the array1
values of the array2
values of the array3
values of the array4
```

Following is another example of code, where we are using some library function to make our development work easier than ever before.

Live Demo

```
fun main(args: Array<String>) {
   val items = listOf(1, 22, 83, 4)

   for ((index, value) in items.withIndex()) {
      println("the element at $index is $value")
   }
}
```

Once we compile and execute the above piece of code in our coding ground, it will yield the following output in the browser.

```
the element at 0 is 1
the element at 1 is 22
the element at 2 is 83
the element at 3 is 4
```

While Loop and Do-While Loop

While and Do-While work exactly in a similar way like they do in other programming languages. The only difference between these two loops is, in case of Do-while loop the condition will be tested

at the end of the loop. The following example shows the usage of the **While loop**.

Live Demo

```
fun main(args: Array<String>) {
   var x:Int = 0
   println("Example of While Loop--")

   while(x< = 10) {
      println(x)
      x++
   }
}
```

The above piece of code yields the following output in the browser.

```
Example of While Loop--
0
1
2
3
4
5
6
7
8
9
10
```

Kotlin also has another loop called Do-While loop, where the loop body will be executed once, only then the condition will be checked. The following example shows the usage of the **Do-while loop**.

Live Demo

```
fun main(args: Array<String>) {
   var x:Int = 0
   do {
      x = x + 10
      println("I am inside Do block---"+x)
   } while(x <= 50)
}
```

The above piece of code yields the following output in the browser. In the above code, Kotlin compiler will execute the DO block, then it will go for condition checking in while block.

```
I am inside Do block---10
I am inside Do block---20
```

```
I am inside Do block---30
I am inside Do block---40
I am inside Do block---50
I am inside Do block---60
```

Use of Return, Break, Continue

If you are familiar with any programming language, then you must have an idea of different keywords that help us implement good control flow in the application. Following are the different keywords that can be used to control the loops or any other types of control flow.

Return − Return is a keyword that returns some value to the calling function from the called function. In the following example, we will implement this scenario using our Kotlin coding ground.

Live Demo

```
fun main(args: Array<String>) {
   var x:Int = 10
   println("The value of X is--"+doubleMe(x))
}
fun doubleMe(x:Int):Int {
   return 2*x;
}
```

In the above piece of code, we are calling another function and multiplying the input with 2, and returning the resultant value to the called function that is our main function. Kotlin defines the function in a different manner that we will look at in a subsequent chapter. For now, it is enough to understand that the above code will generate the following output in the browser.

```
The value of X is--20
```

Continue & Break − Continue and break are the most vital part of a logical problem. The “break” keyword terminates the controller flow if some condition has failed and “continue” does the opposite. All this operation happens with immediate visibility. Kotlin is smarter than other programming languages, wherein the developer can apply more than one label as visibility. The following piece of code shows how we are implementing this label in Kotlin.

Live Demo

```
fun main(args: Array<String>) {
   println("Example of Break and Continue")
   myLabel@ for(x in 1..10) { // appling the custom label
      if(x = = 5) {
         println("I am inside if block with value"+x+"\n-- hence it will close the c
         break@myLabel //specifying the label
      } else {
         println("I am inside else block with value"+x)
         continue@myLabel
```

```
        }
    }
}
```

The above piece of code yields the following output in the browser.

```
Example of Break and Continue
I am inside else block with value1
I am inside else block with value2
I am inside else block with value3
I am inside else block with value4
I am inside if block with value5
-- hence it will close the operation
```

As you can see, the controller continues the loop, until and unless the value of **x** is 5. Once the value of **x** reaches 5, it starts executing the if block and once the break statement is reached, the entire control flow terminates the program execution.