Control flow

It is time to learn about Kotlin's features which we will use to create logic for our software.

If expression

As we have already mentioned, in Kotlin *If* is expression. In Kotlin *If* returns value! Because of that, ternary operator "*condition ? then : else*" is removed! It does not exist.

The following examples demonstrates it (*If.kt*):

```
val result = if (x == y) "It is equal." else "It is not equal."

fun ifExample(x: Int, y: Int) {
    val result = if (x >= y) {
        println("Condition ok.")
        true
    } else {
        println("Condition else.")
        false
    }
    println("Result $result")
}
```

Execute method:

```
ifExample(2, 5)
ifExample(2, 2)
ifExample(5, 2)
```

console output:

```
Condition else.
Result false
Condition ok.
Result true
Condition ok.
Result true
```

If you choose to use *If* as expression rather than statement, your *If* must use *Else* part!

When

When represents replacement for Java's *switch / case*. *When* can be used either as an expression or as a statement.

When matches its argument against all branches sequentially until some branch condition is satisfied. If it is used as an expression, the value of the satisfied branch becomes the value of the overall expression. If it is used as a statement, the values of individual branches are ignored.

The *else* branch is evaluated if none of the other branch conditions are satisfied. If *When* is used as an expression, the *else* branch is mandatory, unless the compiler can prove that all possible cases are covered with branch conditions. If many cases should be handled in the same way, the branch conditions can be combined with a comma.

Let's check out the following examples (*When.kt*) and see what we can do with *When*:

```
// ----- example 1:

fun whenExample(userType: Int) {
   when (userType) {
      0 -> println("Registered user")
      1 -> print("Administrator")
      else -> {
         println("Unknown")
      }
   }
}

// ----- example 2:

fun whenExample2(userType: Int) {
   when (userType) {
      0, 1 -> println("Welcome user.")
      else -> println("Permission denied.")
   }
}

// ----- example 3:

fun whenExample3(userType: Int){
   when (userType) {
      filterUserType(userType) -> {
         println("Subtype ok")
         whenExample2(userType)
      }
      else -> print("Subtype not ok")
   }
}

fun filterUserType(userType: Int): Int {
   if(userType >= 0 && userType < 2){
      return userType;
   }
```

```
    return -1
}
// ----- example 4:

// Checking ranges:
fun whenExample4(x: Int) {
    val from = 0
    val to = 100
    when (x) {
        in from..to -> println("PRECISE")
        in (from / 2)..(to / 2) -> print("VERY PRECISE")
        50 -> print("STRAIGHT IN TARGET")
        else -> print("MISSED")
    }
}

// ----- example 5:

fun whenExample5(fullName: String) {
    val isJohn = when (fullName) {
        is String -> fullName.startsWith("John ")
        else -> false
    }
}

// ----- example 6:

fun whenExample6(fullName: String) {
    when {
        fullName.length == 0 -> println("Please enter your name.")
        fullName.substring(0, 2).equals("X ") -> println("Hello Mr. X")
        fullName.startsWith("John ") && !fullName.endsWith(" Smith") -> println("Hello
John!")
        fullName.endsWith(" Smith") -> println("Hello agent Smith.")
        else -> println("Only secret agents allowed!")
    }
}
```

For

For loop iterates through everything that provides an iterator and has a member, or extension-function *iterator()*. It must be *Iterator* whose return type has a member, or extension-function *next()*, *hasNext()* that returns *Boolean*.

All of these three functions need to be marked as *operator*. A *For* loop over an array is compiled to an index-based loop that does not create an iterator object.

Let's take a look at example *For.kt*:

```
fun forExample(array: Array<String>) {
    for ((index, value) in array.withIndex()) {
        println("[ $index ][ $value ]")
    }
}
```

While and Do..While

We will demonstrate both in code example *While.kt*:

```
fun whileExample() {
    var x = 0
    while (x < 100) {
        println("X: $x")
        x++
    }
}

fun doWhileExample() {
    var x = 0
    do {
        println("X: $x")
        x++
    } while (x < 100)
}
```

Jump operators

In Kotlin we have the following jump operators:

- *return*, by default it returns value from nearest function or anonymous function
- *break*, terminates nearest enclosing loop
- *continue*, goes to next step of nearest enclosing loop.

Jump operator labels

In Kotlin, any expression can be marked with label which has the following form: *@label expression*.

Let's try out labels with break (*JumpOperatorLabels.kt*):

```
myLoop@ for (a in 0..x) {
        for (b in 0..y) {
            println("[ $a ][ $b ]")
            if (b == 10) {
                break@myLoop // Does not break current loop, but the one above it!
            }
```

```
        }
    }
```

then we'll call the method:

```
labelsExample(20, 20)
```

console output looks like this:

```
[ 0 ][ 0 ]
[ 0 ][ 1 ]
[ 0 ][ 2 ]
[ 0 ][ 3 ]
[ 0 ][ 4 ]
[ 0 ][ 5 ]
[ 0 ][ 6 ]
[ 0 ][ 7 ]
[ 0 ][ 8 ]
[ 0 ][ 9 ]
[ 0 ][ 10 ]
```

Return with labels

With function literals, local functions and object expression, functions can be nested in Kotlin. Qualified returns allow us to return from an outer function. **The most important use case is returning from a lambda expression!**

Let's take a look at example *ReturnAtLabels.kt*:

```
fun returnAtLabelsExample(ints: List<Int>) {
    ints.forEach {
        if (it == 0) return
        println("Item: $it")
    }
}

returnAtLabelsExample(arrayListOf(1, 2, 3, 4, 5, 0, 10, 11, 12, 13))
```

Console output:

```
Item: 1
Item: 2
Item: 3
Item: 4
Item: 5
```

```
fun returnAtLabelsExample2(ints: List<Int>) {
    ints.forEach myLabel@ {
```

```
        if (it == 0) return@myLabel
        println("Item: $it")
    }
}

returnAtLabelsExample2(arrayListOf(1, 2, 3, 4, 5, 0, 10, 11, 12, 13))
```

Console output:

```
Item: 1
Item: 2
Item: 3
Item: 4
Item: 5
Item: 10
Item: 11
Item: 12
Item: 13
```

Return returns from the nearest enclosing function. **Such non-local returns are supported only for lambda expressions passed to inline functions.** If we need to return from a lambda expression, we have to label it and qualify the R*eturn*!

We can achieve the same with the following example:

```
fun returnAtLabelsExample3(ints: List<Int>) {
    ints.forEach {
        if (it == 0) return@forEach
        println("Item: $it")
    }
}

returnAtLabelsExample3(arrayListOf(1, 2, 3, 4, 5, 0, 10, 11, 12, 13))
```

Console output:

```
Item: 1
Item: 2
Item: 3
Item: 4
Item: 5
Item: 10
Item: 11
Item: 12
Item: 13
```

We can replace lambda expression with anonymous function:

```
ints.forEach(
```

```
        fun(value: Int) {
            if (value == 0) return
            println("Item: $value")
        }
    )

returnAtLabelsExample4(arrayListOf(1, 2, 3, 4, 5, 0, 10, 11, 12, 13))
```

Console output:

```
Item: 1
Item: 2
Item: 3
Item: 4
Item: 5
Item: 10
Item: 11
Item: 12
Item: 13
```