
# Kotlin Control Flow & Ranges

Sources: http://kotlinlang.org/docs/reference/basic-syntax.html
http://petersommerhoff.com/dev/kotlin/kotlin-for-java-devs/

# Control Flow – if

```
fun maxOf(a: Int, b: Int): Int {
    if (a > b) {
        return a
    } else {
        return b
    }
}

fun main(args: Array<String>) {
    println("max of 0 and 42 is ${maxOf(0, 42)}")
}
```

```
max of 0 and 42 is 42
```

The traditional way to write if's

# Control Flow – if

```kotlin
fun maxOf(a: Int, b: Int): Int {
    if (a > b) {
        return a
    } else {
        return b
    }
}

fun main(args: Array<String>) {
    println("max of 0 and 42 is ${maxOf(0, 42)}")
}
```

```
max of 0 and 42 is 42
```

The traditional way to write *if*'s

HOWEVER….in Kotlin, *if* is an expression, i.e. it returns a value. Therefore there is <u>no ternary operator</u> (condition ? then : else), because ordinary *if* works fine in this role

# Control Flow – if

```
fun maxOf(a: Int, b: Int): Int {
    if (a > b) {
        return a
    } else {
        return b
    }
}

fun main(args: Array<String>) {
    println("max of 0 and 42 is ${maxOf(0, 42)}")
}
```

```
max of 0 and 42 is 42
```

Using if as an expression

```
fun maxOf(a: Int, b: Int) = if (a > b) a else b

fun main(args: Array<String>) {
    println("max of 0 and 42 is ${maxOf(0, 42)}")
}
```

```
max of 0 and 42 is 42
```

# Control Flow – if

```
// Traditional usage
var max = a
if (a < b) max = b

// With else
var max: Int
if (a > b) {
    max = a
} else {
    max = b
}

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

Some examples without using functions.

The first two examples use *if* as a statement.

The last example uses *if* as an expression.

# Control Flow – if

*if* branches can also be blocks.
The last expression is the value of a block:

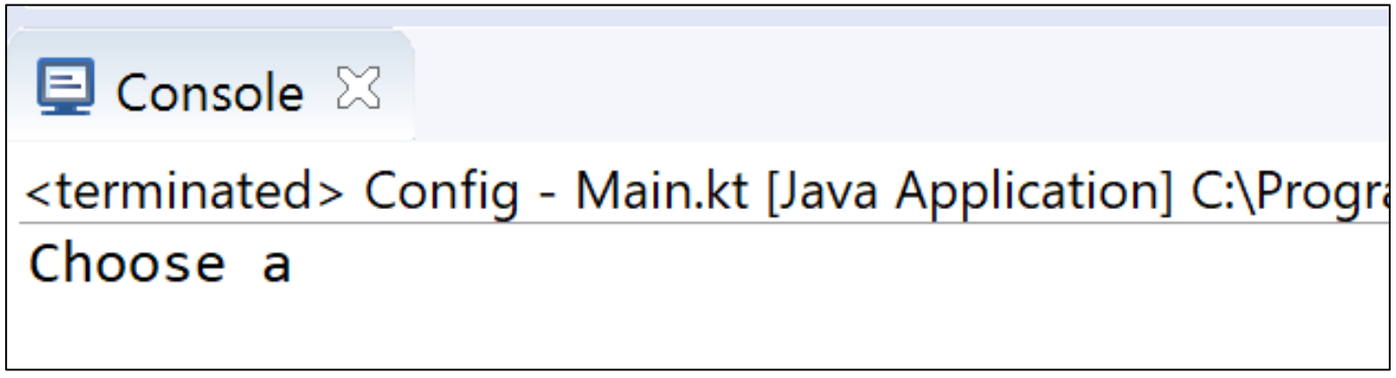
```
val a = 10;
val b = 5;
val max = if (a > b) {
                 print ("Choose a")
                 a
              }
              else {
                 print ("Choose b")
                 b
              }
```

# Control Flow – if

*if* branches can also be blocks.
The last expression is the value of a block:

```
val a = 10;
val b = 5;
val max = if (a > b) {
                print ("Choose a")
                a
            }
            else {
                print ("Choose b")
                b
            }
```

When *if* is used as an expression, the *else* part is mandatory.



# Control Flow – when

Replaces switch in Java

```
val x = 10;
when (x){
   1 -> print("x is 1")
   2 -> print("x is 2")
   in 3..10 -> print ("x is between 3 and 10")
}
```

Console

```
<terminated> Config - Main.kt [Java Application] C:\Program Files\Java
x is between 3 and 10
```

# Control Flow – when

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

# Control Flow – when

```
when (x) {
    0, 1 -> print("x == 0 or x == 1")
    else -> print("otherwise")
}
```

Branch conditions may be combined with a comma.

```
when (x) {
    parseInt(s) -> print("s encodes x")
    else -> print("s does not encode x")
}
```

We can use arbitrary expressions (not only constants) as branch conditions.

```
when (x) {
    in 1..10 -> print("x is in the range")
    in validNumbers -> print("x is valid")
    !in 10..20 -> print("x is outside the range")
    else -> print("none of the above")
}
```

We can also check a value for being *in* or *!in* a range or a collection.

# Control Flow – when

```
val x = "I am a String"

val contains = when (x){
    is String  -> x.contains("I am a")
    else -> false
}

println(contains)
```

Another possibility is to check that a value *is* or *!is* of a particular type.

# Control Flow – when

*when* can also be used as a replacement for an *if-else if* chain.

If no argument is supplied, the branch conditions are simply boolean expressions, and a branch is executed when its condition is true.

```
val aString = "I am a String"

when {
  aString.equals("I am a String") -> println("Equal");
  else -> println("Not Equal")
}
```

Console

```
<terminated> Config - Main.kt [Java Application] C:\Program Files\Java\jre1.8.0_77\bin\javav
Equal
```

# Control Flow – when

```
fun describe(obj: Any): String =
when (obj) {
    1           -> "One"
    "Hello"     -> "Greeting"
    is Long     -> "Long"
    !is String  -> "Not a string"
    else        -> "Unknown"
}

fun main(args: Array<String>) {
    println(describe(1))
    println(describe("Hello"))
    println(describe(1000L))
    println(describe(2))
    println(describe("other"))
}
```

```
One
Greeting
Long
Not a string
Unknown
```

# Control Flow – for

The *for* loop iterates through anything that provides an iterator. It is similar to the *for-each* loop in Java.

```
for (item in collection) print(item)
```

```
fun main(args: Array<String>) {
    val items = listOf("apple", "banana", "kiwi")
    for (item in items) {
        println(item)
    }
}
```

```
apple
banana
kiwi
```

# Control Flow – for

If you want to iterate through an array or a list with an index, you can do it this way:

```
for (i in array.indices) {
    print(array[i])
}
```

```
fun main(args: Array<String>) {
    val items = listOf("apple", "banana", "kiwi")
    for (index in items.indices) {
        println("item at $index is ${items[index]}")
    }
}
```

```
item at 0 is apple
item at 1 is banana
item at 2 is kiwi
```

# Control Flow – for

Alternatively, you can use the withIndex library function:

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

```
fun main(args: Array<String>) {

  val items = listOf("apple", "banana", "kiwi")

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

Console  Gradle Tasks  Gradle Executions

```
<terminated> Config - Main.kt [Java Application] C:\Program File
the element at 0 is apple
the element at 1 is banana
the element at 2 is kiwi
```

# Control Flow – while

The *while* and *do-while* work as usual:

```
while (x > 0) {
    x--
}

do {
    val y = retrieveData()
} while (y != null) // y is visible here!
```

Note: Kotlin also supports traditional *break* and *continue* operators in loops.

# Control Flow – while

```
val items = listOf("apple", "banana", "kiwi")

var index = 0
while (index < items.size){
   println("item at $index is ${items[index]}")
   index++
}
```

Console

```
<terminated> Config - Main.kt [Java Application] C:\Program Files\Java\jre1.8.0_77\bin\
item at 0 is apple
item at 1 is banana
item at 2 is kiwi
```

# Ranges

## The in operator

# Range

Check if a number is within a range using *in* operator:

```
fun main(args: Array<String>) {
    val x = 10
    val y = 9
    if (x in 1..y+1) {
        println("fits in range")
    }
}
```

```
fits in range
```

Check if a number is out of range:

```
fun main(args: Array<String>) {
    val list = listOf("a", "b", "c")

    if (-1 !in 0..list.lastIndex) {
        println("-1 is out of range")
    }
    if (list.size !in list.indices) {
        println("list size is out of valid list indices range too")
    }
}
```

```
-1 is out of range
list size is out of valid list indices range too
```

# Range

Iterating over a range:

```
fun main(args: Array<String>) {
    for (x in 1..5) {
        print(x)
    }
}
```

```
12345
```

Iterating over a progression:

```
fun main(args: Array<String>) {
    for (x in 1..10 step 2) {
        print(x)
    }
    for (x in 9 downTo 0 step 3) {
        print(x)
    }
}
```

```
135799630
```