Table of Contents

Control flow

`for` loops

Kotlin only provides for loops that iterate through iterable objects; that is, any object that has an `operator iterator` function.

```
for (item in collection) { ... }
```

The standard C-style loop is not supported. To iterate through a range of indices, you can take the classic approach:

```
var index = 0
while (index < count) {
    ...
    ++index
}
```

or make use of Kotlin's ranges.

Ranges

The syntax of a range is `lower..upper`; ranges are inclusive, unlike in languages like Python or Swift, they include both of their bounds.

```
for (index in 1..count) { ... }
for (index in 0..(count - 1)) { ... }
```

Since ranges are end-inclusive, the standard library provides the infix function `until` to simplify the creation of exclusive ranges.

```
for (index in 0 until count) { ... }
```

With the standard range-manipulating infix functions, you can create all sorts of ranges — backwards ranges, ranges that skip values, you name it.

```
println((1..10).toList()) // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
println((1 until 10).toList()) // [1, 2, 3, 4, 5, 6, 7, 8, 9]
println((10 downTo 1).toList()) // [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
println((10 downTo 1 step 3).toList()) // [10, 7, 4, 1]
```

In the special case of iterating over an array's indices and elements, you can use `forEachIndexed`:

```
val list = listOf(1, 2, 3, 4)
list.forEachIndexed { index, i -> println("$index: $i") }
```

If you need to do any complex manipulation of the current loop index, you will have to fall back to a `while` loop and a manual index. This can often be avoided; for common use cases, there is often a standard library function that does what you want. windowed can be used for the common "rolling window" approach, for example.

while / do...while

These loop types function as they do in C.

when

`when` is Kotlin's replacement for the switch statement. It is much more powerful and does not suffer from accidental fall-through; in fact, **fall-through is impossible**!

You can match values of any type, check the type of a value, or check if it is contained in a collection (the `in` operator):

```
val x = readLine()?.toIntOrNull()!!
when (x) {
    1 -> println("x = 1")
    in 12..38 -> println("x is between 12 and 38")
    !in 3..5 -> println("x is not between 3 and 5")
    else -> println("None of the conditions were satisfied")
}

val anObject: Any = ...
when (anObject) {
    is String -> println("The object is a string")
    is Number -> println("The object is a number")
    else -> println("I don't know what type the object is")
}
```

Kotlin's `when` isn't quite as powerful as Scala's `match` expression (there is no support for pattern matching), but it's a nice step up from Java.

Nested scopes (labels)

Similarly to Java, Kotlin can make use of *labels* to return from a nested scope:

```
outer:
for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 3; ++j) {
        if (i == 1 && j == 2) break outer;
    }
}
```

would become:

```
outer@for (i in 0 until 3) {
    for (j in 0 until 3) {
        if (i == 1 && j == 2) break@outer
    }
}
```

Labels can also be used to return from lambdas. We can use this to fix our earlier problem with non-local returns:

```
fun loopList() {
    list.forEach { item ->
        if (item == 3)
            return@forEach // This will only return from the lambda,
                           // skipping the current item,
                           // so this is equivalent to a `continue`
    }
}
```