Mobile Application Development

Produced by

David Drohan (ddrohan@wit.ie)
Dr. Siobhan Drohan (sdrohan@wit.ie)

Department of Computing & Mathematics
Waterford Institute of Technology
http://www.wit.ie

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Introducing Kotlin Syntax - Part 1.2

Agenda

- ❑ Basic Types
- ❑ Local Variables (val & var)
- ❑ Functions
- ❑ Control Flow (if, when, for, while)
- ❑ Strings & String Templates
- ❑ Ranges (and the *in* operator)
- ❑ Type Checks & Casts
- ❑ Null Safety
- ❑ Comments

Agenda

- ❑ Basic Types
- ❑ Local Variables (val & var)
- ❑ Functions
- ❑ Control Flow (if, when, for, while)
- ❑ Strings & String Templates
- ❑ Ranges (and the *in* operator)
- ❑ Type Checks & Casts
- ❑ Null Safety
- ❑ Comments

Functions

Parameters, return types, expression body,
inferred return type

Functions – parameters and return types

Function having two `Int` parameters with `Int` return type:

```
fun sum(a: Int, b: Int): Int {
    return a + b
}

fun main(args: Array<String>) {
    print("sum of 3 and 5 is ")
    println(sum(3, 5))
}
```

```
sum of 3 and 5 is 8
```

Functions – expression body, inferred return type

```
fun sum(a: Int, b: Int) = a + b

fun main(args: Array<String>) {
    println("sum of 19 and 23 is ${sum(19, 23)}")
    println("sum of 19 and 23 is " + sum(19, 23))
}
```

Function “sum” with an expression body and inferred return type

Console

```
<terminated> Config - Main.kt [Java Application] C:\Program Files\Java\jre1.8.0_77\b
sum of 19 and 23 is 42
sum of 19 and 23 is 42
```

Functions – no return data

Function returning no meaningful value:

```
fun printSum(a: Int, b: Int): Unit {
    println("sum of $a and $b is ${a + b}")
}

fun main() {
    printSum(-1, 8)
}
```

```
sum of -1 and 8 is 7
```

`Unit` return type can be omitted:

```
fun printSum(a: Int, b: Int) {
    println("sum of $a and $b is ${a + b}")
}

fun main() {
    printSum(-1, 8)
}
```

```
sum of -1 and 8 is 7
```

Functions – no return data

Function returning no meaningful value:

```kotlin
fun printSum(a: Int, b: Int): Unit {
    println("sum of $a and $b is ${a + b}")
}

fun main() {
    printSum(-1, 8)
}
```

```
sum of -1 and 8 is 7
```

`Unit` return type can be omitted:

```kotlin
fun printSum(a: Int, b: Int) {
    println("sum of $a and $b is ${a + b}")
}

fun main() {
    printSum(-1, 8)
}
```

```
sum of -1 and 8 is 7
```

Control Flow

if, when, for, while

Control Flow – if

The traditional way to write if's

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

fun main() {
    println("max of 0 and 42 is ${maxOf(0, 42)}")
}
```

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

Control Flow – if

The traditional way to write if's

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

fun main() {
    println("max of 0 and 42 is ${maxOf(0, 42)}")
}
```

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

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

Using if as an expression

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

fun main() {
    println("max of 0 and 42 is ${maxOf(0, 42)}")
}
```

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

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

fun main() {
    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 be blocks, and the last expression is the value of a block:

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

In general, if you're using *if* as an expression rather than a statement (for example, returning its value (as above) or assigning it to a variable), the expression is **required** to have an **else** branch.

Control Flow – when

```
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")
}
```

when replaces *switch* in Java

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

- Another possibility is to check that a value *is* or *!is* of a particular type. Note that, due to smart casts, you can access the methods and properties of the type without any extra checks.

```
fun hasPrefix(x: Any) = when(x) {
    is String -> x.startsWith("prefix")
    else -> false
}
```

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.

```
when {
    x.isOdd() -> print("x is odd")
    x.isEven() -> print("x is even")
    else -> print("x is funny")
}
```

Control Flow – when

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

fun main() {
    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() {
    val items = listOf("apple", "banana", "kiwifruit")
    for (item in items) {
        println(item)
    }
}
```

```
apple
banana
kiwifruit
```

Control Flow – for

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

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

Control Flow – for

- Alternatively, you can use the *withIndex* library function:

```
fun main() {
    val array = arrayOf("a", "b", "c")
    for ((index, value) in array.withIndex()) {
        println("the element at $index is $value")
    }
}
```

```
the element at 0 is a
the element at 1 is b
the element at 2 is c
```

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.

References

Sources:

http://kotlinlang.org/docs/reference/basic-syntax.html
http://petersommerhoff.com/dev/kotlin/kotlin-for-java-devs/
https://www.programiz.com/kotlin-programming
https://medium.com/@napperley/kotlin-tutorial-5-basic-collections-3f114996692b


Thanks.