

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

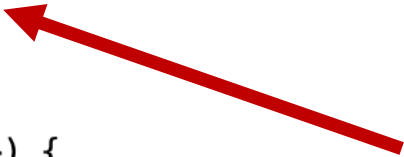
Parameters, return types, expression body,
inferred return type



Functions – parameters and return types

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

```
1 fun sum(a: Int, b: Int): Int {  
2     return a + b  
3 }  
4  
5 fun main(args: Array<String>) {  
6     print("sum of 3 and 5 is ")  
7     println(sum(3, 5))  
8 }
```





```
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:

```
1 fun printSum(a: Int, b: Int): Unit {  
2     println("sum of $a and $b is ${a + b}")  
3 }  
4  
5 fun main() {  
6     printSum(-1, 8)  
7 }
```

sum of -1 and 8 is 7

Unit return type can be omitted:

```
1 fun printSum(a: Int, b: Int) {  
2     println("sum of $a and $b is ${a + b}")  
3 }  
4  
5 fun main() {  
6     printSum(-1, 8)  
7 }
```

sum of -1 and 8 is 7

Functions – no return data

Function returning no meaningful value:

```
1 fun printSum(a: Int, b: Int): Unit {  
2     println("sum of $a and $b is ${a + b}")  
3 }  
4  
5 fun main() {  
6     printSum(-1, 8)  
7 }
```

sum of -1 and 8 is 7

Unit return type can be omitted:

```
1 fun printSum(a: Int, b: Int) {  
2     println("sum of $a and $b is ${a + b}")  
3 }  
4  
5 fun main() {  
6     printSum(-1, 8)  
7 }
```

sum of -1 and 8 is 7

Control Flow

if, when, for, while



Control Flow – if

The traditional way to write **if**'s

```
1 fun maxOf(a: Int, b: Int): Int {  
2     if (a > b) {  
3         return a  
4     } else {  
5         return b  
6     }  
7 }  
8  
9 fun main() {  
10     println("max of 0 and 42 is ${maxOf(0, 42)}")  
11 }
```

max of 0 and 42 is 42

Control Flow – if

The traditional way to write *if*'s



```
1 fun maxOf(a: Int, b: Int): Int {  
2     if (a > b) {  
3         return a  
4     } else {  
5         return b  
6     }  
7 }  
8  
9 fun main() {  
10     println("max of 0 and 42 is ${maxOf(0, 42)}")  
11 }
```

max of 0 and 42 is 42

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

Control Flow – if

Using **if** as an expression

```
1 fun maxOf(a: Int, b: Int): Int {  
2     if (a > b) {  
3         return a  
4     } else {  
5         return b  
6     }  
7 }  
8  
9 fun main() {  
10     println("max of 0 and 42 is ${maxOf(0, 42)}")  
11 }
```

max of 0 and 42 is 42

```
1 fun maxOf(a: Int, b: Int) = if (a > b) a else b  
2  
3 fun main() {  
4     println("max of 0 and 42 is ${maxOf(0, 42)}")  
5 }
```

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.

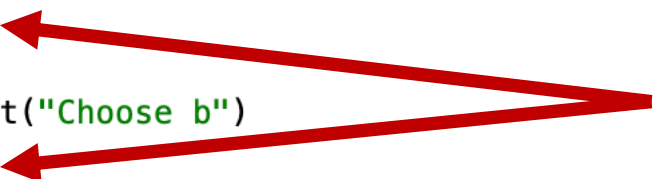
The first two examples use *if* as a statement.

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

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

- ❑ 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

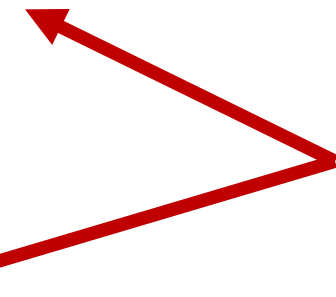
- ❑ *when* can also be used as a replacement for an *if-else if* chain.
- ❑ If no argument is supplied, the branch conditions are simply **boolean** expressions, and a branch is executed when its condition is true.

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

Control Flow – when

```
1 fun describe(obj: Any): String =  
2   when (obj) {  
3       1        -> "One"  
4       "Hello"   -> "Greeting"  
5       is Long    -> "Long"  
6       !is String -> "Not a string"  
7       else      -> "Unknown"  
8   }  
9  
10 fun main() {  
11     println(describe(1))  
12     println(describe("Hello"))  
13     println(describe(1000L))  
14     println(describe(2))  
15     println(describe("other"))  
16 }
```

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)
```

```
1 fun main() {  
2     val items = listOf("apple", "banana", "kiwifruit")  
3     for (item in items) {  
4         println(item)  
5     }  
6 }
```

```
apple  
banana  
kiwifruit
```

Control Flow – for

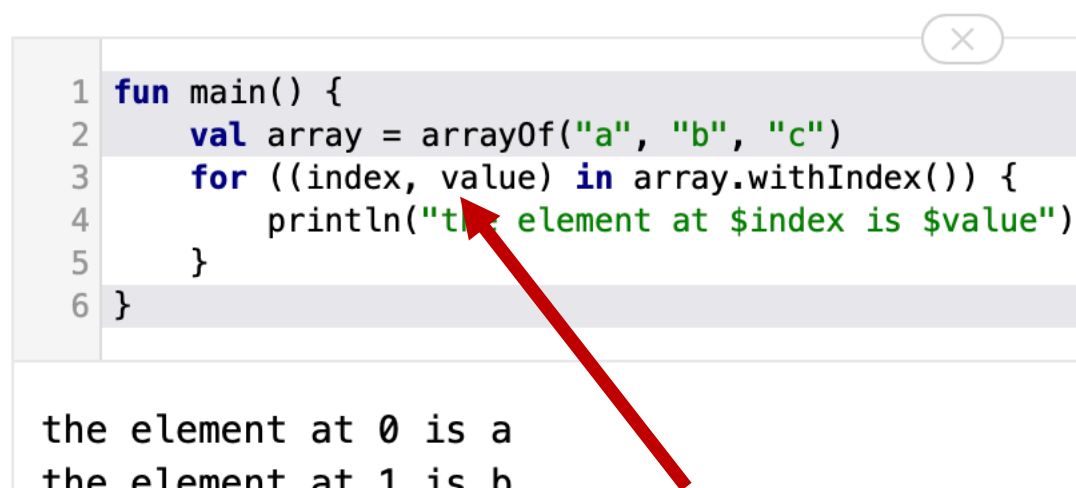
- ❑ If you want to iterate through an array or a list with an index, you can do it this way:

```
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:



```
1 fun main() {  
2     val array = arrayOf("a", "b", "c")  
3     for ((index, value) in array.withIndex()) {  
4         println("the element at $index is $value")  
5     }  
6 }
```

```
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>

