Kotlin > Java

Session 1

What will we cover?

In this session, we'll cover the following concepts:

- Variable declarations and types.
- Function declarations & named arguments.
- Nullable Types (!! operator, elvis operator, safe calls).
- If/else and expression if.
- When blocks
- Collections
- Lambdas + Anonymous Functions
- .also{} and .apply{}
- String Templating

If you need help at any point, then raise your hand or ask a mentor - they'll be more than happy to help!

Getting started with Kotlin

If you already have IntelliJ & Kotlin set up on your computer you can skip this.

If you don't have Intellij or Java

installed

- Log into one of the Lab PCs. Faster and a lot easier to set up.
- Follow the instructions below for '*If you have IntelliJ & Java installed*'.

If you have Intellij & Java installed

- Open Intellij and Create a new project.
- In the 'New Project' dialogue click on Java on the left, and make sure Kotlin/JVM is checked.
- If next to 'Project SDK' it says *<No SDK>*, click new and navigate to your Java SDK installation. If you don't have Java installed then log into one of the Lab PCs and follow these instructions from the start.
- Press 'Next'
- If all goes according to plan, you should have a new empty IntelliJ Project.

If you have trouble with any of this, feel free to ask a mentor for help!

Hello World

You knew this was coming. Create a new kotlin file (right click on src -> new -> kotlin file) and copy & paste the following code into the IDE:

```
fun main(args: Array<String>) {
    print("Hello World")
```

```
    //Your code here
}
```

Congratulations! Your first Kotlin program. Click on the green tick in the left column of the text editor to run that code and check if your log correctly outputs ‘Hello World’. If it does, great! Everything is configured correctly. If not, speak to one of the helpers to make sure you have set up IntelliJ correctly.

Variable Declarations & Types

As you could see from the code above, there are slight differences in syntax between Kotlin and Java, such as the *controvertial* optional semicolon.
Kotlin code can call Java code and vice versa. (We can use Java objects like ArrayList in our Kotlin code).

Var and Val

There are two types of basic variable declaration in Kotlin, `var`s and `val`s. A `var` declaration will create a variable that can be re-assigned:

```
var x = "Hello"
x = "World!" // all good
```

A `val` declaration however cannot be reassigned, it is a constant.

```
val bestLang = "Kotlin"
bestLang = "Java" // whoops!
```

You can also have `val`s that are compile-time constants (`const val`) i.e. cannot be assigned as the result of a runtime expression:

```
const val okay = "Kotlin123"
const val notOkay = readLine()
```

One thing you may have noticed is that none of these variable declarations have types! Variable declarations where the compiler can 'guess' the type are not necessary, but it is perfectly valid to put them in anyway for clarity. Kotlin types are specified after the variable name:

```
var x: String = "I have an explicit type"
```

Nullable Types

Every variable in kotlin be given a value on declaration. There are two 'exceptions' to this, Nullable Types and Lateinit variables.
Variables in Kotlin are not allowed to have `null` values by default:

```
Integer myInt = null; //In Java, this is allowed
```

```
var myInt: Int = null //This is not allowed in Kotlin! Wil
l not compile.
```

If you want to represent nullable values in Kotlin, it needs to be explicitly stated in the type!

```
var myNullableInt: Int? = null // Type is required in this
 case, as if only null was given, the compiler cannot gues
s what the intended type was.
```

Gone are the days of the untraceable NullPointerException. Nullable Types offer an extra layer of protection for the programmer so they can see exactly what can and can't be null. However, you run into issues when trying to *use* that value in specific circumstances:

```
var myInt: Int = myNullableInt //Wrong, Cannot convert fro
m Int? to Int
```

You cannot use a nullable type as that type directly, but there are a few ways to work around this. You can either assert the value to a non-null type using the non-null assertion operator `!!` , or you can use the *elvis operator* `?:` to give an alternative vaue if the given value is null.

```
var myInt: Int = myNullableInt!! //Asserts that myNullable
Int does not have value null. If it does, throws an NPE!
var myInt2: Int = myNullableInt ?: 10 //The elvis operator
 checks if the left value is null, and if it is, then the
expression has the value of the right value (10)
```

You can also use if-statements to determine if a nullable value is null, but we will get to that later.

Lateinit

Lastly, if a `var` is given the `lateinit` modifier, then that variable does not have to be given a value immediately. This is good for variables that cannot be populated in a constructor. You cannot have `lateinit val`s.

```
lateinit var myLateString: String // type required
...
myLateString = "I have a value now!"
```

Make sure you don't access the variable before it's given a value, or again it'll crash!

Function Declarations & Named Arguments

Function Declarations

Functions in Kotlin are of a similar structure to Java and other C style languages, but the positioning of elements is slightly different.
Take an example function:

```
fun exampleFunc(myString: String, myInt: Int): String {
```

```
    return "this is a string"
}
```

As seen earlier with our main function, kotlin functions are defined starting with the keyword `fun`. The name then follows (`exampleFunc`) as well as any parameters, similar to java (`myString: String, myInt: Int`) using the Kotlin style of typing. `val`/`var` is left out, as fuunction parameters are always `val` (non-reassignable).
We then give our return type just before the start of the code block (`String`, again this can be left out if the type can be assumed by the compiler).

Functions can also have default values:

```
fun double(myInt: Int = 1) { return myInt * 2 }
double(5) == 10
double(20) == 40
double() == 2
```

Unit Type

Kotlin doesn't have a 'void' type like Java, as most constructs in Kotlin are expressions i.e. return some type. If no return type is necessary, then the type `Unit` is given. This is the type given automaticaly if there is no return statement in a function:

```
fun printA(String: s) { print(s) }
```

```
fun printB(String: s): Unit { print(s) } // two equivalent
 function definitions
```

Single Expression Functions

Lets look again at the main function we had before.

```
fun main(args: Array<String>) = print("Hello World")
```

But wait, that's not what we had before? There's an equals sign instead of a block {}.
Kotlin allows single expression function to be simplified in this way:

```
fun addOne(operand: Int) = 1 + operand // or in java:
//int addOne(int operand) { return 1 + operand; }
```

Calling Functions

Calling functions works similarly to in Java, but allows for calling functions with *named arguments*. This is useful if you want to be more explicit with how you pass arguments into functions:

```
fun divide(dividend: Int, divisor: Int) = dividend / divis
or

val result = divide(10, 2) // Calling functions like Java
```

```
val result2 = divide(divisor = 2, dividend = 10) // Identi
cal function call with named arguments.
```

Calling Functions on Nullable Objects

If we have a `String?` object, we can't just call a function on it (like `nullableString.capitalize()`), as the compiler won't let you. We need to make sure that we're not calling that function on a null object. We can do that in two ways, using the aformentioned assertion operator `!!`, or the safe-call operator `?.`.
The safe call operator will only call the function on the object if it's non-null, otherwise it does nothing and returns null.

```
val capsString1: String = (nullableString!!).capitalize()
// brackets are optional and for representation, will cras
h if nullableString is null.
val capsString2: String? = nullableString?.capitalize() //
 if nullableString is null, capsString becomes null.
```

If-Else

If-else works similarly to Java and other C-Style languages.

```
if (true) print("isTrue")
else if (2 == 1) print("badMaths")
else print ("oof")
```

However, in Kotlin if is *always* an expression i.e. it always returns a value:

```
val myStringResult = if (2 == 1) "Hello World"
else if (2 == 3) "Hello Galaxy"
else "Hello Universe"
```

This also allows you to use if expressions in single-expression functions (`fun f(myBool: Boolean) = if (myBool) 1 else 0`)
If you use if-else as an expression (i.e. not a Unit return type) , you *must* provide an else clause, otherwise the compiler will complain about missing control paths and the expression won't be well-formed.

Loops & Ranges

While loops are pretty much identical in Kotlin and Java/C++ so I won't be covering that. For loops however are slightly different. The Java for loop syntax no longer exists, and instead loops are iterated over *iterable* types. An example of a basic iterable type is a `range`.
A range is a series of values with given start, end and step values. A simple range can be defined using the `..` operator like so: `0..5` which creates a range from 0 to 5 *inclusive*.

```
for (i in 0..5) { print(i) } //0,1,2,3,4,5 and in java:
//for (int i = 0; i <= 5; ++i) { print(i) }
```

Want a reverse order range? Use `downto`:

```
for (i in 5 downto 0) { print(i) } //5,4,3,2,1,0
```

Want a specific step value? use `step`:

```
for (i in 0..5 step 2) { print(i) } //0,2,4
```

Want to create a range with an *exclusive* end? Use `until`:

```
for (i in 0 until myStringList.size) { myStringList[i] } /
/ Equivalent to (i in 0..myStringList.size-1)
```

All of these can be combined together to get the specific values you want in your iteration.
You can also iterate over containers such as Java ArrayLists.

```
val args = arrayOf("Hello", "World", "It's", "Kotlin")
for (arg in args) println(arg)
```

When (and why it’s better than switch/case)

A `when` block is Kotlin’s equivalent to a switch/case statement.