Introduction to Android development with Kotlin

Miguel Ángel Ruiz López

What is Kotlin?

- Lenguaje developed by Jetbrains
- It is designed to work in the JRE
- Kotlin is **interoperable** with Java
- It is **concise** and reduces the amount of boilerplate
- Kotlin is safe, reduces the number of errors, such as null pointer exceptions
- Fully compatible with Android Studio

Defining a class

This is a typical class of model in Java

```
public class UserModel {
        private String name;
        private String surname;
        private String address;
        private int Age;

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name  = name;
        }

        public String getSurname() {
                return surname;
        }

        public void setSurname(String surname) {
                this.surname = surname;
        }

        public String getAddress() {
                return address;
        }

        public void setAddress(String address) {
                this.address = address;
        }
```

This is a way to write it in Kotlin

```
class UserModel(var name: String, var surname: String, var address: String, var age: Int)
```

Note 1: To create an instance we don't need the keyword "new"

```
UserModel("Homer", "Simpson", "Evergreen Terrace", 40)
```

Note 2: A file can contain more than one class or interface

Inheritance in kotlin

Here we have an Inheritance example

```
class ExtendedClass : BaseClass(), BaseInterface
```

Classes cannot be extended by default. To do that is necessary add “open” keyword. This is needed also to override attributes or methods

Objects

The Objects are similar to a static class in Java. They are declared with the "object" keyword and can no have instances so to call them you have to use the name of the object as in Java.

```
object ObjectExample {
        const val constant = "Value"

        fun staticMethod() {}
}
```

If you need to use some constants or static functions in a class, they have to be declared as “Companion Object” inside the class.

```
class CopanionObjectExample {
        companion object {
                const val constant = "Value"

                fun staticMethod() {}
        }
}
```

Variables

Exists two types of variables in Kotlin:

Read-only variables defined by **"val"**

```
val immutableVariable: String = "Immutable"
```

Reassignable variables defined by "var"

```
var mutableVariable: String = "Mutable"
```

Functions

Functions in Kotlin are declared using the fun keyword:

```
fun sum(x: Int, y: Int): Int {
        return x + y
}
```

A better way to write this method is:

```
fun sum2(x: Int, y: Int): Int = x + y
```

Even in this case, we can remove the return type (you should do this when the return type is clear)

```
fun sum3(x: Int, y: Int) = x + y
```

Extension Functions

These functions let you add functionality to any class

```
fun Fragment.toast(message: CharSequence, duration: Int = Toast.LENGTH_SHORT) {
        Toast.makeText(context, message, duration).show()
}
```

Methods Overloading

You can have in Java some methods with the same name but different arguments

```java
public void method(String variable1) {
        method(variable1, 0);
}

public void method(String variable1, int variable2) {
}
```

And what about Kotlin?

```kotlin
fun method(variable1: String, variable2: Int = 0) {}
```

Null Safety

- In kotlin for each type exists another type nullable (for example String, String?)
- That means that if you introduce a null in a variable of String type the compiler returns an error

```
var a: String = "notNull"
a = null // Compilation error
```

What can we do?

- Just declare the variable of type "String?" This forces us to control if the variable is null everywhere.

```
val b: String? = null
```

- SmartCast: The compiler infers that a variable cannot be null later of a proper checking.

```
val b2 = b.length // it doesn't compile
val l = if (b != null) b.length else -1
```

- Safe calls: Let you access to inner variables from a nullable instance. In case this instance is null then return null for the full expression (instead of a null pointer exception). These are useful in chains.

```
homer?.departament?.head?.name
```

- Elvis Operator: Let you return some value in case an expression return null.

```
val l2 = b?.length ?: -1
```

- The operator "!!" is similar to the operator "?" but return a NullpointerException in case we have some null in the chain (No recommended)

```
homer!!.departament!!.head!!.name
```

- Safe casts: Regular casts may result into a ClassCastException if the object is not of the target type. Another option is to use safe casts that return null if the attempt was not successful

```
val aInt: Int? = a as? Int
```

STANDARD DELEGATES

- Lazy: Initialize the variable only when this is needed.

```
val dabase: SQLiteOpenHelper by lazy { MyDatabaseHelper(applicationContext) }
```

- Observable: When the variable is updated, then the lambda given is executed.

```
var name by Delegates.observable("No Name") {
        _, old, new -> println("$old -> $new")
}
```

- Lateinit: Sometimes we now that the variable will be initialized before we use it but not in the declaration, so to avoid to set it with null or a dummy value we can use "lateinit".

```
lateinit var view: View
```

Synthetic imports

The basic way to import view in Android is using “findById” method. We write this in this way in Kotlin(Some of you maybe use Butterknife):

```
val nameTextField = activity?.findViewById<TextView>(R.id.nameField)
nameTextField?.text = "Name"
```

Using synthetic imports we can access to the view just using the id of the View:

```
nameField.text = "Name"
```

Parcelable

Parcelables are useful to pass classes in Android to a Fragment or Activity. The normal way to make a class parcelable is to implement methods to read a write manually the variables.

The new way in Kotlin:

```
@Parcelize
class ParcelableExample(val param1: String, val param2: String) : Parcelable
```

Kotlin Scope Functions

- Let: Calls the specified function block with this value as its argument and returns its result.

```
val len = text?.let {
        println("get length of $it")
        it.length
} ?: 0
println("Length of $text is $len")
```

- Apply: Usually is used to make modifications in an instance created.

```
fun newInstance(arg1: String, arg2: String): MyFragment {
        val myFragment = MyFragment()
        val bundle = Bundle()
        bundle.putString(ARG1, arg1)
        bundle.putString(ARG2, arg2)
        myFragment.arguments = bundle
        return myFragment
}
```

```
fun newInstance(arg1: String, arg2: String) =
        MyFragment().apply {
                arguments = Bundle().apply {
                        putString(ARG1, arg1)
                        putString(ARG2, arg2)
                }
        }
```

- Run: Calls the specified function block with this value as its receiver and returns its result.

```
val len = text?.run {
        println("get length of $this")
        length //`this` can be omitted
} ?: 0
println("Length of $text is $len")
```

COLLECTIONS

There are two types of collections, mutable and immutable (List/MutableList, Set/MutableSet, Map/MutableMap…)

Dedicated constructors such as listOf(), mutableListOf(), setOf(), mutableSetOf()

```
val items = listOf(1, 2, 3)
```

There are many methods useful to deal with the collections in the functional way (map, filter, none, first, last, +, ...).

Those functions let you avoid fully "for" and "while" loops.

```
listOf(1, 2, 3, 4, 5).map { it * 5 }.filter { it % 2 == 0 } // [10, 20]
```

THANKS!

Twitter: **@maRuizLopez**

LinkedIn: **www.linkedin.com/in/maruiz81**

email: **maruiz81@gmail.com**