KOTLIN FOR ANDROID

WHAT'S KOTLIN

- JVM based Object oriented | Functional programming language by Jetbrains
- Kotlin compiles to JVM bytecode, JavaScript or Native.

WHY KOTLIN

- Concise : Drastically reduce the amount of boilerplate code.
- Safe: Avoid entire classes of errors such as null pointer exceptions.
- Interoperable: Leverage existing libraries for the JVM, Android, and the browser.
- Tool-friendly: Choose any Java IDE or build from the command line.

Source: Kotlin

NULLS IN KOTLIN

NULLS IN KOTLIN

- Nulls in Kotlin don't exist until you say otherwise.
- No variable, by default, can be set to null

```
val x: Int = null //Won't compile
val x: Int? = null

val y = x.toDouble() //Won't compile
if (x != null) {
    val y = x.toDouble()
}
```

Source: Antonio Leiva

NULLS IN KOTLIN

- Secure access expression

```
val y = x?.toDouble()
```

  - In this case, if x is null, then the expression willl return null as well. So y will be of type Double?.

- The Elvis operator

```
val y = x?.toDouble() ?:0.0
```

```
val y = if (x != null) {
    x.toDouble()
} else {
    0.0
}
```

Source: Antonio Leiva

NULLS IN KOTLIN

- Avoiding the null check

```
val x: Int? = null
val y = x!!.toDouble()
```

  - This would compile and produce a NullPointerException

Source: Antonio Leiva

VARIABLES IN KOTLIN

VARIABLES IN KOTLIN

- The variables can be mutable and immutable
- Variables are declared using `val` or `var` , provided as they are immutable or mutable

```
var x = 7
var y: String = "my String"
var z = View(this)
```

  - As you see, you do not need to use `*new*` to create a new instance of an object.

Source: Antonio Leiva

VARIABLES IN KOTLIN

- Type casting is done automatically

```
val z: View = findViewById(R.id.my_view)


if (z is TextView) {
    z.text = "I've been casted!"
}
```

  - I did not call setText()

Source: Antonio Leiva

VARIABLES IN KOTLIN

- In Kotlin everything is an object
  - There are no basic types, and there is no void
  - If something does not return anything, it actually returns the Unit

```
val x: Int = 20
val y: Double = 21.5
val z: Unit = Unit
```

Source: Antonio Leiva

VARIABLES IN KOTLIN

- Simpler numerical types cannot be assigned to more complex types
  - an integer cannot be assigned to a long variable. This does not compile:

```
val x: Int = 20
val y: Long = x
```

  - You need to do an explicit casting:

```
val x: Int = 20
val y: Long = x.toLong()
```

Source: Antonio Leiva

CLASSES IN KOTLIN

CLASSES IN KOTLIN

- Declare the class

```
class Person
```

  - You don't need to use braces if the class doesn't contain any code.
  - class is not using public visibility modifier? That's because everything is public in Kotlin by default.

Source: Antonio Leiva

CLASSES IN KOTLIN

- Add properties
  - Fields don't exist in Kotlin. A class has poperties.
  - A property substites a Java field + getter + setter.

```
class Person {
    var name = "Name"
        set(value) {
            field = "Name: $value"
        }
    var surname = "Surname"
}
```

Source: Antonio Leiva

CLASSES IN KOTLIN

- Add constructor

```
class Person(val name: String, val surname: String)
```

  - If the constructor arguments are annotated with var or val, the properties are generated inline
  - The constructor is written just after the definition of the class.

Source: Antonio Leiva

CLASSES IN KOTLIN

- Functions inside the class

```
class Person(val name: String, val surname: String) {
    fun getFullName() = "$name $surname"
}
```

  - Functions can be written in a contracted way when a value is directly assigned.

```
fun getFullName(): String {
    return "$name $surname"
}
```

Source: Antonio Leiva

CLASSES IN KOTLIN

- Kotlin is closed by default
  - So it can't be extended, and children (in case a class can be extended) can't override its functions, unless it's indicated with the reserved word open:

```
open class Person(val name: String, val surname: String)

class Cop(surname: String) : Person("Mr", surname)
```

Source: Antonio Leiva

DEFAULT / NAMED ARGUMENTS

DEFAULT / NAMED ARGUMENTS

- There is no need to define several similar methods with varying arguments:

```
fun build(title: String, width: Int = 800, height: Int = 600) {
    Frame(title, width, height)
}
build("PacMan", 400, 300)                          // equivalent
build(title = "PacMan", width = 400, height = 300)  // equivalent
build(width = 400, height = 300, title = "PacMan")  // equivalent
```

Source: Medium

RANGES IN KOTLIN

RANGES IN KOTLIN

- There is no need to define several similar methods with varying arguments:

```
for (i in 1..100) { ... }
for (i in 0 until 100) { ... }
for (i in 2..10 step 2) { ... }
for (i in 10 downTo 1) { ... }
if (x in 1..10) { ... }
```

Source: Medium

LAMBDAS IN KOTLIN

LAMBDAS IN KOTLIN

- A lambda is a way of representing a function

```
val view = findViewById(R.id.welcomeMessage)
view.setOnClickListener { v -> navigateWithView(v) }
```

Source: Antonio Leiva

LAMBDAS IN KOTLIN

- How to define a function that accepts lambdas

```
fun setOnClickListener(listener: (view:View) -> Unit){}
```

  - This is known as a Higher-Order Function, If a function that receives a function by parameter, or that returns a function is a Higher-Order Function.

- The natural way of calling this function

```
view.setOnClickListener({ v -> navigateWithView(v) })
```

Source: Antonio Leiva

LAMBDAS IN KOTLIN

- If the last parameter of a function is a function, we can extract it from the parentheses

```
view.setOnClickListener(){ v -> navigateWithView(v) }
```

- If there is only one function as a parameter, we can just get rid of the parentheses

```
view.setOnClickListener { v -> navigateWithView(v) }
view.setOnClickListener { navigateWithView(v) }
```

Source: Antonio Leiva

EXTENSION PROPERTIES / FUNCTIONS IN KOTLIN

EXTENSION PROPERTIES / FUNCTIONS IN KOTLIN

- Extension functions are functions that help us to extend the functionality of classes without having to touch their code.

```
fun ImageView.loadUrl(url: String) {
    Picasso.with(context).load(url).into(this)
}


imageView.loadUrl(url)
```

Source: Antonio Leiva

EXTENSION PROPERTIES / FUNCTIONS IN KOTLIN

- Just as you can do extension functions, so can you do with properties.
- The only thing you need to remember is that extension properties cannot save state.

```
fun View.visible() {
    visibility = View.VISIBLE
}
val ViewGroup.children: List
    get() = (0..childCount -1).map { getChildAt(it) }
parent.children.forEach { it.visible() }
```

Source: Antonio Leiva

DATA CLASSES IN KOTLIN

DATA CLASSES IN KOTLIN

Source: Antonio Leiva

WHEN EXPRESSION IN KOTLIN

WHEN EXPRESSION IN KOTLIN

- you can use it as a regular switch

```
when(view.visibility){
    View.VISIBLE -> toast("visible")
    View.INVISIBLE -> toast("invisible")
     else -> toast("gone")
}
```

Source: Antonio Leiva

WHEN EXPRESSION IN KOTLIN

- Auto-casting
  - If you check something in the left side, you'll get the cast in the right side.

```
when (view) {
    is TextView -> toast(view.text)
    is RecyclerView -> toast("Item count = ${view.adapter.itemCount}")
    is SearchView -> toast("Current query: ${view.query}")
    else -> toast("View type not supported")
}
```

Source: Antonio Leiva

WHEN EXPRESSION IN KOTLIN

- when without arguments
  - With this option, we can check basically anything we want in the left side of the when condition.

```
val res = when {
    x in 1..10 -> "cheap"
    s.contains("hello") -> "it's a welcome!"
    v is ViewGroup -> "child count: ${v.getChildCount()}"
    else -> ""
}
```

Source: Antonio Leiva

INTERFACES IN KOTLIN

INTERFACES IN KOTLIN

- interfaces can have code.

```
interface Interface1 {
    fun function1() {
        Log.d("Interface1", "function1 called")
    }
}
```

Source: Antonio Leiva

```
interface Interface2 {
    fun function2() {
        Log.d("Interface2", "function2 called")
    }
}

class MyClass : Interface1, Interface2 {
    fun myFunction() {
        function1()
        function2()
    }
}
```

INTERFACES IN KOTLIN

- Interfaces can't keep state

```
interface Toaster {
    val context: Context

    fun toast(message: String) {
        Toast.makeText(context, message,
        Toast.LENGTH_SHORT).show()
    }
}
```

```
class MyActivity : AppCompatActivity(), Toaster {
    override val context = this

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        toast("onCreate")
    }
}
```

OBJECTS IN KOTLIN

OBJECTS IN KOTLIN

- Object declaration
- Companion Object

Source: Antonio Leiva

NEXT THINGS

- Subtle differences between Kotlin's with(), apply(), let(), also(), and run() at here.
- Collections: List, Set, Map at here.
- Collection operations in Kotlin at here.

- Android Development with Kotlin — Jake Wharton at YouTube
- Introduction to Kotlin (Google I/O '17) at YouTube