Laboratorio di Applicazioni Mobili
Bachelor in Computer Science &
Computer Science for Management

University of Bologna

Kotlin

Federico Montori
federico.montori2@unibo.it

Table of Contents

Java and Kotlin

Why Java and Kotlin?

Java has been the official language for years and most supported until 2021.

As for now, it's <u>not</u> the most used, Kotlin took over, however since we know Java we can still use it.

A few drawbacks though…

more on this later.

Kotlin

It is the **official programming language for Native Android** since 2019

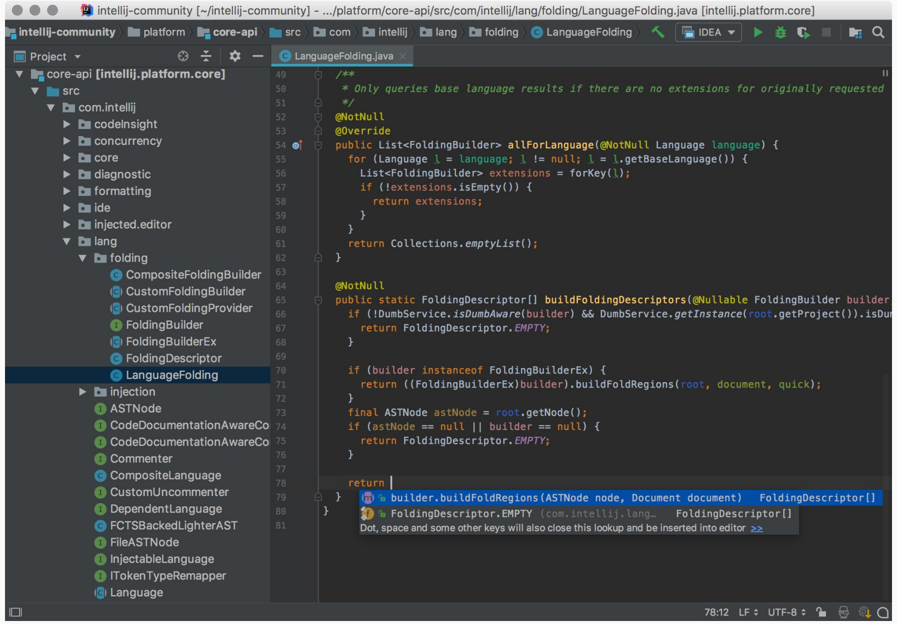
- Announced by JetBrains in 2011
- New language for the JVM
- Open source since 2012 under Apache 2 License
- Named after Kotlin Island
    - FYI Java is an island too

Kotlin General Features

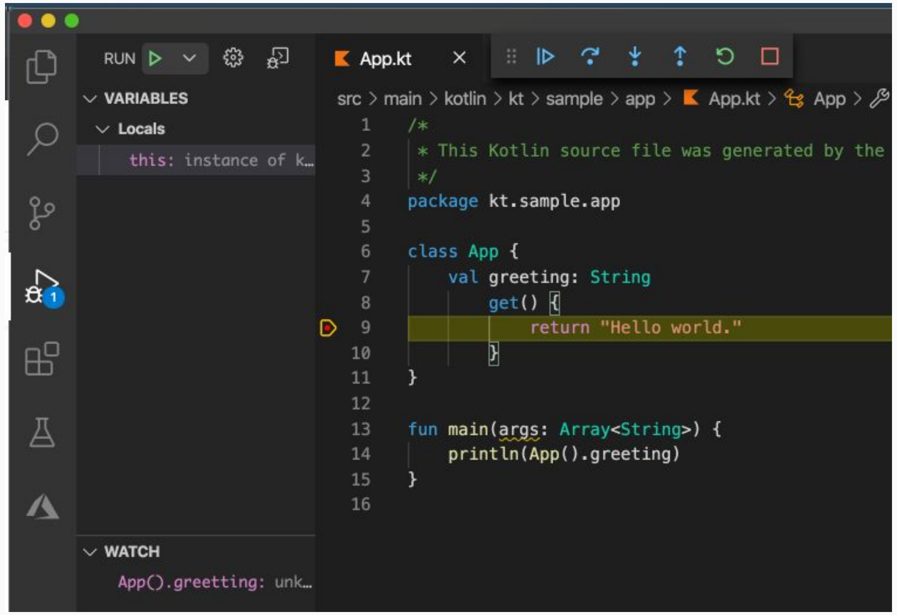
- It is a **Type Inference** language (like Python)
    - Still, it is **statically typed** (unlike Python)
- It is **Cross-Platform**
- It compiles to **Java Bytecode**
    - Fully interoperable with Java
    - You can write easily mixed code projects
    - It can also compile to Javascript and other stuff

Getting started with Kotlin

Kotlin is Cross-Platform → like Java, it is not bound to Android



Intellij IDEA (supported natively)



Visual Studio Code

Basically the brother of Android Studio…

Variables and Types

Declaration of variables and types

```
var x: Int = 42        // Declaration of a variable with type Int
var x = 42             // Declaration of a variable with inferred type Int
val x = 42             // Declaration of a constant with inferred type Int
```

Type inference does not mean that types are dynamic (like in Python…)

```
var x = 42
x = ‘c’                // This will give an error
```

Disclaimer: this is an accelerated tutorial :: Complete official guide: https://kotlinlang.org/docs/home.html

Variables and Types

Basic types:

- Int
- Long
- Short
- Byte
- Float
- Double
- Boolean
- Char
- String

Can always specify them, or:

```
var x = 42
var x = 42L


var x = 42.42f
var x = 42.42
var x = true
var x = ‘f’
var x = “fortytwo”
```

You can specify true constants:

```
const val
numRounds = 42

/* This can only be
used in top-level
declaration and it is
not evaluated at
runtime */
```

Operators

Operations in Kotlin are quite straightforward…

- Arithmetic Operators
  - + - * / %
- Logical Operators
  - && || !
- Comparison Operators
  - < > == >= <= !=

Strings and Prints

Like some other imperative languages, the access point is the **main** function.

```
// Enhanced Hello World Example
fun main() {
    val nickname: String = “stradivarius”
    println(“Hello world, my name is $nickname”)
}
```

Selection Contstruct

The IFTE construct is straightforward too…

```
if ( condition ) {
    // Then Clause
} else {
    // Else Clause
}
```

There is a contract syntax for assignments

```
var y = if (x == 42) 1 else 0
```

Selection Construct

The case construct is as follows

```
when ( x ) {
    in 0..21 -> println("One line clause")
    in 22..42 -> println {
        println("Multiple line clause")
    }
    else -> println("Default clause")
}
```

With the double dot (..) you can specify ranges, which originate Lists (see later).

Arrays and Lists

```
val arr: IntArray = intArrayOf(1, 2, 3)            // [1,2,3]
println(arr[0])
```

Arrays are a class and can be instantiated in several ways (they also have their subtypes):

Equivalent to their primitive in C: immutable in size, type-invariant

```
// Array of int of size 5 with values [0, 0, 0, 0, 0]
val arr = IntArray(5)

// Array of int of size 5 with values [42, 42, 42, 42, 42]
val arr = IntArray(5) { 42 }

// Array of int of size 5 with values [0, 1, 2, 3, 4] (lambda, you'll see…)
var arr = IntArray(5) { it * 1 }
```

Arrays and Lists

Lists can be “constants” or “variables”.

ArrayList is just one List implementation…

```
// Immutable List
val myList = listOf<String>("one", "two", "three")
println(myList)

// Mutable List (referenced by a val because it is the pointer)
val myMutableList = mutableListOf<String>("one", "two", "three")
myMutableList.add("four")
```

Loops

The iteration constructs are straightforward too…

```
// While loop
var counter = 0
while (counter < myMutableList.size) {
    println(myMutableList[counter])
    counter++
}

// For loop
for(item in myListMutable)                  // Here we can use ranges as well
    println(item)
```

Null Safety

One of the major advantages of Kotlin is the **Null Safety**

➔ The program does not crash because of null values (remember the annoying Java **NullPointerException**)

- Basically types are non-nullable, in fact variables are either:
  - Initialized
  - Explicitly null, but they throw error at compile time
- Variables that can be null are *Nullable* but calling them is safe

let’s see how…

Null Safety

Non nullable types

```
var s: String = "Hello"      // Regular initialization means non-null by default
s = null                     // compilation error
```

Nullable types

```
var s: String? = "Hello"     // Nullable initialization means it can be null
s = null                     // this is ok: e.g. if you print it, it will print “null”
```

Null safety

```
val l = s.length                              // Compiler error: “s can be null”
val l = s?.length                             // If s is null then l is null (if nullable)
val l = if (s != null) s.length else -1       // Custom workaround
```

Null Safety

This is true even for more complex scenarios, for instance:

```
val name: String? = department?.head?.getName()
```

If anything in here is null, then the function is not called

You really want it to be not null:

```
val l = s!!.length                // Casts s to non nullable, can throw exception
```

The “Elvis” operator

```
val l = s?.length ?: -1           // -1 is the default value for l if s is null
```

Functions

Ordinary functions (they support the default value)

```
fun isEven(number: Int = 0): Boolean {          // number is set to 0 if not passed
    return number % 2 == 0
}
isEven(14)
```

Extension functions

```
fun Int.isEven(): Boolean {                     // Extend the class Int
    return this % 2 == 0
}
14.isEven()
```

Higher Order Functions

Higher order functions take functions as inputs

```
// Function that counts members in a List of strings that respect a certain condition

fun List<String>.customCount(function: (String) -> Boolean): Int {
    var counter = 0
    for (str in this) {
        if (function(str))
            counter++
    }
    return counter
}
```

Higher Order Functions

They might as well take any type in (usually called “generics”)

```
// Function that counts members in a List of any type that respect a certain condition

fun <T> List<T>.customCountAllTypes(function: (T) -> Boolean): Int {
    var counter = 0
    for (anything in this) {
        if (function(anything))
            counter++
    }
    return counter
}
```

Lambdas

Lambdas are undeclared functions that are passed directly as they are and used once.

➔ Added to Java as well

Let us use the previous higher order functions…

```
val myList = listOf<String>(“one”, “two”, “three”)

val x: Int = myList.customCount { str -> str.length == 3 }

val x: Int = myList.customCountAllTypes { str -> str.length == 3 }
```

Classes

Classes are pretty much like in Java, however they typically have a primary constructor:

```
class Animal (                          // Constructor is within round brackets
        val name: String,
        val legCount: Int = 4           // Default value if not passed
) {
        var sound: String = "Hey"       // Property not initialized by the constructor

        init {
                println("Hello I am a $name")   // Function executed at instantiation time
        }
}
val dog = Animal("dog")                 // Instantiation of a class into an object
val duck = Animal("duck", 2)
```

Classes

Properties have default accessors (setters, getters...)

you can define custom ones or make it private...

```
// Equivalent notation
var sound: String = "Hey"
    get() = field
    set(value) { field = value }            // Keyword field refers to the property

// Custom notation
var sound: String = "Hey"
    get() = this.name
    private set                             // Setter is private

val dog = Animal("dog")
dog.sound                                   //  Will access the getter, not the property
```

Classes

You can obviously subclass that if the original class is **open**

```
class Dog: Animal("dog") {
    fun bark() {
        println("WOOF")
    }
}
```

```
class Duck: Animal("duck", 2) {
    fun quack() {
        println("QUACK")
    }
}
```

Classes

Let us make that abstract

```
abstract class AbstractAnimal (

    val name: String,
    val legCount: Int = 4
) {
    abstract fun makeSound()
}
```

Then you'll have to implement the abstract method

```
class Cat: AbstractAnimal("cat") {
    override fun makeSound() {
        println("MEOW")
    }
}
```

Classes

You can create an anonymous class, if used only once:

```
val bear = object: AbstractAnimal("bear") {
    override fun makeSound() {
        println("GROWL")
    }
}
```

You can also create a sealed class, to prevent third parties to extend it outside your package - *i.e.* subclasses are known at compile time.

Classes

In Kotlin every <u>object property</u> needs to be initialized upon declaring the object. You can defer that by using **lateinit**

```
class Animal (
    val name: String,
    val legCount: Int = 4
) {
    var sound: String = "Hey"
}
```

```
class Animal (
    val name: String,
    val legCount: Int = 4
) {
    lateinit var sound: String
}
```

You must make sure that the variable is initialized somewhere else, e.g. in a Unit Test or a setup function…

Classes

A **companion object** is much like a static object in Java, it creates a Singleton that is tied to the class, rather than to the instance.

```kotlin
class Animal (
	val name: String,
	val legCount: Int = 4
) {
	companion object {
		const val Kingdom: String = "Animalia"
	}
}

println(Animal.Kingdom)
```

The example shows a single constant value, but it might as well be a fully fledged object, like a **factory**.

Scope Functions

Scope functions are used to simplify multiple interaction with the same object: **apply** (context object is the receiver “this”, returns the object itself)

```
val snake = Animal("snake")                // Without “apply”
snake.legCount = 0
snake.sound = "Hiss"
```

```
val snake = Animal("snake").apply {        // With “apply”
    legCount = 0
    sound = "Hiss"
}
```

Scope Functions

Scope functions are used to simplify multiple interaction with the same object: **let** (context object is the lambda argument “it”)

```
val numbers = mutableListOf("one", "two", "three", "four", "five")
val resultList = numbers.map { it.length }.filter { it > 3 }          // Without Let
println(resultList)
```

```
val numbers = mutableListOf("one", "two", "three", "four", "five")
numbers.map { it.length }.filter { it > 3 }.let {                     // With Let
    println(it)
    // and more function calls if needed without using a result variable
}
```

Scope Functions

Scope functions are used to simplify multiple interaction with the same object: **with** (context object passed, but is the receiver “this”)

```
val snake = Animal("snake")                    // Without “with”
snake.makeSound()
```

```
val snake = Animal("snake")
with(snake) {                                  // With “with”
   makeSound()
}
```

Scope Functions

Scope functions are used to simplify multiple interaction with the same object: **run** (context object is the receiver “this”, but returns the lambda result)

```
val snake = Animal("snake")                    // Without “run”
snake.legCount = 0
val legNumbers = snake.howManyLegs()
```

```
val snake = Animal("snake")                    // With “run”
val legNumbers = snake.run() {
    legCount = 0
    howManyLegs()
}
```

Scope Functions

Scope functions are used to simplify multiple interaction with the same object: **also** (context object is the lambda argument “it”, but returns the object)

```
val numbers = mutableListOf("one", "two", "three", "four", "five")
numbers.add("six")                                        // Without Also
println(numbers)
```

```
val numbers = mutableListOf("one", "two", "three", "four", "five")
numbers.also {                                            // With Also
    it.add("six")
    println(it)
}
```

Delegation

There may be cases where your class implements an interface in the same way as it is implemented elsewhere→ you can delegate the implementation.

```
interface Animal {
    val legCount: Int
}

class Cat
    (override val legCount: Int) : Animal
```

```
class PersianCat (val cat : Cat)
    : Animal by cat {
        fun someOtherMethod () { … }
}
// This will automatically implement all the
interface members of Animal in
PersianCat by invoking the same member
on cat.
```

Delegation

```kotlin
import kotlin.reflect.KProperty
class DelegatedProperty ( private val default: String ) {
    private var _value: String? = null
    private var loaded = false
    operator fun getValue(
        thisRef: Any?, property : KProperty<*>): String? {
        if (loaded) return _value
        _value = retrieveValue()
        loaded = true
        return _value
    }
}

// In your body: the by redirects to getValue()
val name by DelegatedProperty("myDef")
```

Delegation can be used to do lazy loading (i.e. evaluating an expression only the first time it is invoked). Kotlin also has a built-in expression

```kotlin
val name: String? by lazy {
    retrieveValue()
}
```

Kotlin & Java: Differences

- Explicit types
- Strictly OOP
- Not Null Safe
- Explicit set & get

- Type inference
- Not necessarily OOP
- Null Safe
- Implicit set & get
- + Extension functions
- + Scope Functions
- + Lambdas
- + Implicit Casting
- + Structured Concurrency
  - Coroutines (TBC)

Kotlin for Android

How to set up an Android project in Kotlin?

Literally in the same way it is done for Java!

- Still uses XML resources
- Everything still applies to what we will see:
  - Resources
  - Activity Lifecycle
  - Fragments
  - Intents
  - Views
- Only thing that changes is the syntax…

Kotlin for Android

BUT…
There are certains things that can only be done with Kotlin…

- Android Jetpack Compose projects (similar to Flutter)
- Structured concurrency (Coroutines & Flows)

Mainly because Java is there for historical and retrocompatibility reasons.
Mainly library issues… because they are both Turing complete they have virtually the same capabilities!

Questions?

federico.montori2@unibo.it