Android Development

IDE: Android Studio

Android SDK

Introduction to the Kotlin language

Development Environment

- "*Android Studio*" installation
  - *Java SDK*
    - Optional for Android Studio 2.2 or higher
  - *Android Studio*
  - *Android SDK*

Java SDK (optional)

- Download *Java SDK IF*
  - https://www.oracle.com/java/technologies/javase-downloads.html

- Install

Android Studio

- Download *Android Studio*
  - https://developer.android.com/studio

- Install

Android SDK

- Execute *Android Studio*
- Open the *Configure* menu and choose SDK Manager
- Options to select
  - SDK Platforms
    - Choose a recent platform >= 8.0 (API >= 26)
      - Android SDK Platform
      - Sources (opt.)
      - A system image (Google APIs or Google Play)
  - SDKTools
    - Android SDK Build-Tools (latest)
    - Android SDK Command-line Tools (latest)
    - Android Emulator
    - Android SDK Platform-Tools
    - Google Play Services
    - Intel x86 Emulator (HAXM)
      - No longer required for the Android Emulator in Android Studio versions 33.x.x and later
    - (opt: Google USB driver)
- Accept/Install

Android Development

Kotlin quick start

Source : http://kotlinlang.org

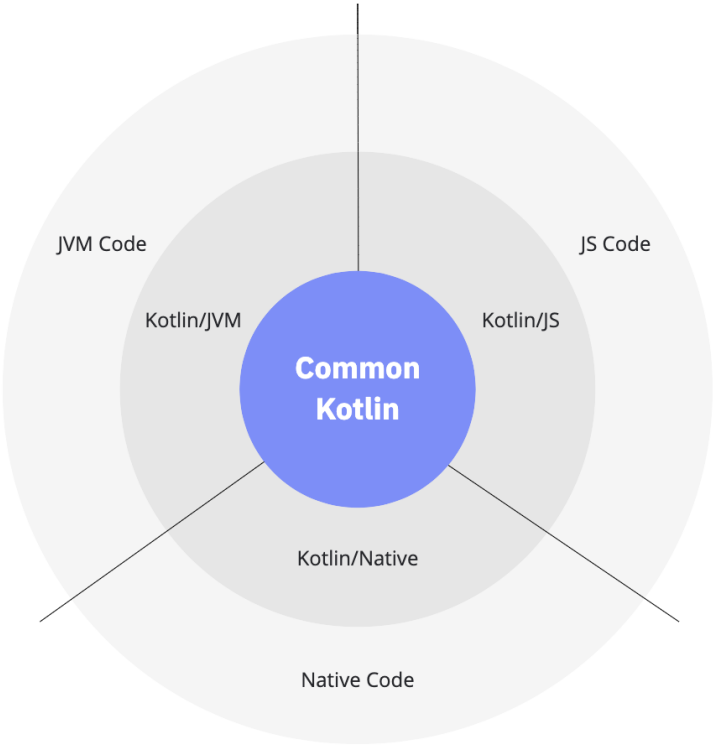
Kotlin

- The Kotlin language is the result of a project started by JetBrains in 2010-2011 with the goal of creating a more concise and modern open-source language to meet the needs of programmers

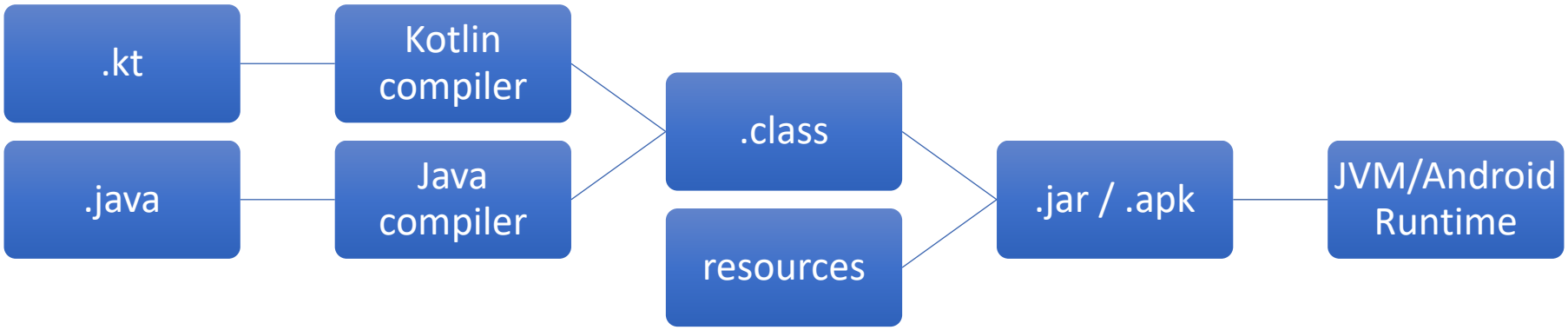
- Inspired by other languages: Java, C#, Scala, Groovy , ...

Kotlin

- Recognized as the language for Android development but...
  - Fully interoperable with the Java language
  - Interoperable with many other server-side or client-side development languages and platforms (desktop, mobile, and web)
  - Multiplataform



Java/Kotlin interoperability



Hello world

```
fun main() {
    println("Hello World!!!")
}
```

https://developer.android.com/training/kotlinplayground

Concepts and basic syntax

- Easy to learn for those with some knowledge of other object-oriented languages.
  - Especially for those with experience in Java, C#, or Swift
- Obvious differences in the first approach to the language
  - Non-use of `;` to end lines of code
  - The declaration of variables is carried out in the opposite order to the "usual"
    - First indicate the name and then the type
      - Additionally, declarations are preceded by a keyword indicating whether it is immutable (" `val`") or mutable (" `var`")
  - Functions/methods begin with the keyword "`fun`" and the return type is indicated after the parameters
  - Enables functional and object-oriented programming
  - It is not necessary to use a specific word ("`new`") to create object instances
  - The traditional for-cycle format "`for(ini;cond;inc)`" is not accepted
    - There is only the for loop in the *for each* format: `for( x in xxx )`
  - The `switch` is replaced by `when`
    - Enables the definition of more flexible cases

Mutable and immutable variables

- In the declaration of variables, the word '`val`' is used for immutable variables and '`var`' for mutable variables

  {[const] <val> | var>} <name> [: <type>] [= <value>]

```
var a : Int
val b: Double = 5.5
var c = 123
const val d = "ISEC"
```

- Variable types can be inferred from the assigned values

- Basic types
  - `Byte, Short, Int, Long, UByte, UShort, UInt, ULong`
  - `Float, Double`
  - `Char, String`
  - `Boolean`

input and *output* (console)

- *output*
  - `print`
  - `println`
    - `println("DEIS")`

- *input*
  - `readLine`
    - `var s = readLine()`
  - …it is also possible to import and use the `Scanner` class normally used in Java
    - `val sc = Scanner(System.`in`)`

Strings and Ranges

- *Strings*

```
val i = 123
println("i = $i ${i+1}")

var s = """
   texto mais longo
   com varias linhas
"""
```

- *Ranges*
  - `1..20`
  - `20 downto 1`
  - `1..20 step 3`
  - `20 downto 1 step 2`

Arrays

- They are objects of the `Array` class

- Examples of creating arrays

```
val tab1 = arrayOf(1,2,3,4,5)
val tab2 = arrayOf(1,2,3,4.5,6)

val tab3 = Array(5, { it * it } )
val tab4 = Array(4) { i -> (i * 10).toString() }
```

- Example of use

```
tab1[1] = tab1[0] + 1
```

Flow control: if

- It works in a similar way to the usual, but it is an expression
  - "`a?b:c`" operator does not exist because it is not necessary
  - When used as an expression, `else` is mandatory

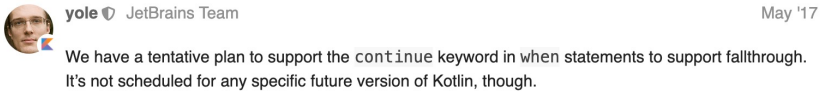
```
// Traditional usage
var max = a
if (a < b) max = b

// With else
var max: Int
if (a > b) {
    max = a
} else {
    max = b
}

// As expression
val max = if (a > b) a else b
```

Flow control: when

- The '`when`' control structure has similar functionality to '`switch`' in other languages, although it allows a different type of flexibility in *'case'* analysis
  - Expressions can be used in *'case'* statements
  - It is not necessary to use '`break`' between each case
  - If none of the cases include the value under analysis, then the code corresponding to the '`else`' case is executed
  - Like '`if`', the '`when`' can be used as an expression




```
when (x) {
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> { // Note the block
        print("x is neither 1 nor 2")
    }
}
```

```
when (x) {
    0, 1 -> print("x == 0 or x == 1")
    else -> print("otherwise")
}
```

```
when (x) {
    parseInt(s) -> print("s encodes x")
    else -> print("s does not encode x")
}
```

```
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")
}
```

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

Flow control: cycles

- `for`
  - The `for` structure in *Kotlin* corresponds to *for-each* in other languages
- `while`
- `do…while`

```
for (item in collection) print(item)
```

```
for (i in 1..3) {
    println(i)
}
for (i in 6 downTo 0 step 2) {
    println(i)
}
```

```
for (i in array.indices) {
    println(array[i])
}
```

```
for ((index, value) in array.withIndex()) {
    println("the element at $index is $value")
}
```

```
while (x > 0) {
    x--
}

do {
    val y = retrieveData()
} while (y != null) // y is visible here!
```

break, continue and return

- They work in a similar way to Java but…

```
loop@ for (i in 1..100) {
    for (j in 1..100) {
        if (...) break@loop
    }
}
```

```
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach {
        if (it == 3) return // non-local return directly to the caller of foo()
        print(it)
    }
    println("this point is unreachable")
}
```

```
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach lit@{
        if (it == 3) return@lit // local return to the caller of the lambda
        print(it)
    }
    print(" done with explicit label")
}
```

Functions

- Defined with the help of the word `fun`

```
fun double(x: Int): Int {
    return 2 * x
}
```

- Unlike Java, global functions can exist, not encapsulated in a class
  - Example:
    - `main` function – first function to be executed

```
fun main() {
   . . .
}
```

Functions

- Functions can have multiple parameters
  - These parameters can have default values

```
fun reformat(
    str: String,
    normalizeCase: Boolean = true,
    upperCaseFirstLetter: Boolean = true,
    divideByCamelHumps: Boolean = false,
    wordSeparator: Char = ' ',
) {
/*...*/
}
```

```
fun double(x: Int): Int = x * 2
```

```
fun <T> asList(vararg ts: T): List<T> {
    val result = ArrayList<T>()
    for (t in ts) // ts is an Array
        result.add(t)
    return result
}
```

```
reformat('This is a long String!')

reformat('This is a short String!', upperCaseFirstLetter = false, wordSeparator = '_')
```

Functions and Lambda functions

```
infix fun Int.shl(x: Int): Int { ... }

// calling the function using the infix notation
1 shl 2

// is the same as
1.shl(2)
```

```
fun dfs(graph: Graph) {
    val visited = HashSet<Vertex>()
    fun dfs(current: Vertex) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v)
    }

    dfs(graph.vertices[0])
}
```

```
// Parameter types in a lambda are optional if they can be inferred:
val joinedToString = items.fold("Elements:", { acc, i -> acc + " " + i })
```

```
ints.filter { it > 0 } // this literal is of type '(it: Int) -> Boolean'
```

Scope functions

Function	Object reference	Return value	Is extension function
`let`	`it`	Lambda result	Yes
`run`	`this`	Lambda result	Yes
`run`	-	Lambda result	No: called without the context object
`with`	`this`	Lambda result	No: takes the context object as an argument.
`apply`	`this`	Context object	Yes
`also`	`it`	Context object	Yes

```
fun main() {
    val str = "Hello"
    // this
    str.run {
        println("The receiver string length: $length")
        //println("The receiver string length: ${this.length}")
    }

    // it
    str.let {
        println("The receiver string's length is ${it.length}")
    }
}
```

```
val numberList = mutableListOf<Double>()
numberList.also { println("Populating the list") }
    .apply {
        add(2.71)
        add(3.14)
        add(1.0)
    }
    .also { println("Sorting the list") }
    .sort()
```

```
val numbers = mutableListOf("one", "two", "three")
with(numbers) {
    println("'with' is called with argument $this")
    println("It contains $size elements")
}
```

Classes and main constructors

- Classes are declared using the '`class`' keyword
- Within the class declaration, parameters can be defined
  - These parameters…
    - Serve as inputs when creating instances and correspond to the primary constructor of the class
    - Values are used to initialize class properties
    - They can also become class properties, adding 'var' or 'val' to their definition
- '`init`' code blocks can be defined, executing in the order they are declared when creating an instance
  - Multiple '`init`' blocks can be used

```
class InitOrderDemo(name: String) {
    val firstProperty = "First property: $name".also(::println)

    init {
        println("First initializer block that prints ${name}")
    }

    val secondProperty = "Second property: ${name.length}".also(::println)

    init {
        println("Second initializer block that prints ${name.length}")
    }
}
```

Secondary constructors

- Classes can have 0 or more secondary constructors
  - Defined with the word `constructor`
  - They must call the main constructor if it exists

```
class Person {
    var children: MutableList<Person> = mutableListOf<>()
    constructor(parent: Person) {
        parent.children.add(this)
    }
}
```

```
class Person(val name: String) {
    var children: MutableList<Person> = mutableListOf<>()
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

More about classes

- There can be *nested classes*, *inner classes* and *anonymous inner classes*

```
class Outer {
    private val bar: Int = 1
    class Nested {
        fun foo() = 2
    }
}

val demo = Outer.Nested().foo() // == 2
```

```
class Outer {
    private val bar: Int = 1
    inner class Inner {
        fun foo() = bar
    }
}

val demo = Outer().Inner().foo() // == 1
```

- Instances of objects are created without using the '`new`' keyword (typical in other languages)

Properties, getters and setters

- Properties are defined with `var` or `val`
- They all need to be initialized by default, or they need to be initialized in the constructors

```
class Address {
    var name: String = "Holmes, Sherlock"
    var street: String = "Baker"
    var city: String = "London"
    var state: String? = null
    var zip: String = "123456"
}
```

- *Getters* and *Setters*

```
var <propertyName>[: <PropertyType>] [= <property_initializer>]
    [<getter>]
    [<setter>]

var stringRepresentation: String
    get() = this.toString()
    set(value) {
        setDataFromString(value) // parses the string
    }
```

  - The keyword `field` can be used to refer to the value itself

Inheritance

- Classes can inherit characteristics from other classes and redefine behaviors

- All classes in Kotlin have a common superclass called Any
  - The Any class has 3 methods already known from Java that can be redefined: `equals`, `hashCode` and `toString`

- For a class to be inherited by another, the word `open` must be included in its definition
  - Methods that can be redefined must also have the `open` tag

Inheritance

- The syntax for inheriting characteristics in a new class is
  - `class NewClass : BaseClass() { … }`

- Method redefinition must be explicit, indicating the `override` keyword in its definition

- Access to base class methods is carried out using the word `super`

- The instance itself is designated by `this`

Inheritance

```
open class Shape {
    open fun draw() { /*...*/ }
    fun fill() { /*...*/ }
}

class Circle() : Shape() {
    override fun draw() { /*...*/ }
}

open class Rectangle() : Shape() {
    final override fun draw() { /*...*/ }
}
```

Inheritance: more examples

```
open class Rectangle {
    open fun draw() { /* ... */ }
}

interface Polygon {
    fun draw() { /* ... */ } // interface members are 'open' by default
}

class Square() : Rectangle(), Polygon {
    // The compiler requires draw() to be overridden:
    override fun draw() {
        super<Rectangle>.draw() // call to Rectangle.draw()
        super<Polygon>.draw() // call to Polygon.draw()
    }
}
```

```
open class Polygon {
    open fun draw() {}
}

abstract class Rectangle : Polygon() {
    abstract override fun draw()
}
```

Interfaces

```
interface MyInterface {
    fun bar()
    fun foo() {
      // optional body
    }
}

class Child : MyInterface {
    override fun bar() {
        // body
    }
}
```

```
interface MyInterface {
    val prop: Int // abstract

    val propertyWithImplementation: String
        get() = "foo"

    fun foo() {
        print(prop)
    }
}

class Child : MyInterface {
    override val prop: Int = 29
}
```

```
interface A {
    fun foo() { print("A") }
    fun bar()
}

interface B {
    fun foo() { print("B") }
    fun bar() { print("bar") }
}

class C : A {
    override fun bar() { print("bar") }
}

class D : A, B {
    override fun foo() {
        super<A>.foo()
        super<B>.foo()
    }

    override fun bar() {
        super<B>.bar()
    }
}
```

Other concepts

- enums
- extensions
- data classes
- sealed classes
- object
  - singleton
- companion objects
  - They enable access to features like those found in Java's static elements
- coroutines
- collections
  - List, Set, Map, …
- delegation
- generics
- exceptions

```
enum class Direction {
    NORTH, SOUTH, WEST, EAST
}
```

```
enum class Color(val rgb: Int) {
        RED(0xFF0000),
        GREEN(0x00FF00),
        BLUE(0x0000FF)
}
```

```
class Example {
    fun printFunctionType() { println("Class method") }
}

fun Example.printFunctionType(i: Int) { println("Extension function") }

Example().printFunctionType(1)
```

```
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}

val instance = MyClass.create()
```

```
val numbersSet = setOf("one", "two", "three", "four")
val emptySet = mutableSetOf<String>()
```

```
val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)
```

```
data class User(val name: String, val age: Int)
```