Kotlin Programming Language

CONSTRUCCIÓN DE INTERFACES DE USUARIO

2do Cuatrimestre de 2019

Universidad Nacional de Quilmes

Hello World

```kotlin
class Greeter(val name: String) {
  fun greet() {
    println("Hola $name")
  }
}

fun main(args: Array<String>) {
  Greeter(args[0]).greet()
}
```

- Package opcional
- Entry point como función (no necesita estar en una clase)
- Los ; son opcionales
- Scope public por default
- Constructor primario en la declaración

```java
package ar.edu.unq.ui.greeting;

public class Greeter {
  private String name;

  public Greeter(String name) {
    this.name = name;
  }

  public void greet() {
    System.out.println("Hello " + name);
  }

  public static void main(String[] args)
  {
    new Greeter(args[0]).greet();
  }
}
```

Consideraciones

- Esta clase es una **presentación** de Kotlin.
- **No es** una guía exhaustiva.
- Todo nuevo lenguaje se aprende codeando.
- Cuando surjan dudas primero vayan a https://kotlinlang.org/docs/reference/
- También tienen guías nativas en https://play.kotlinlang.org/koans/
- Y tienen las Prácticas y los TPs para codear.

Acerca de Kotlin

- Lenguaje de Propósito General
- Multi Plataforma
- Tipado Estático
- Inferencia de Tipos

- Desarrollado por JetBrains
- El nombre fue dado por la Isla de Kotlin, cerca de San Petersburgo (Rusia)
- Vamos a usar la versión 1.3

- Puede compilar a
  - JVM (por ahora solo nos importa este)
  - Android
  - Javascript
  - Assembler Nativo
    - iOS
    - MacOS
    - Windows
    - Linux
    - WebAssembly

Funciones

```
fun printMessage(message: String): Unit {
  println(message)
}

fun printMessageWithPrefix(message: String, prefix: String = "Info") {
  println("[$prefix] ${message}")
}

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

fun multiply(x: Int, y: Int) = x * y

fun main() {
  printMessage("Hello")
  printMessageWithPrefix("Hello", "Log")
  printMessageWithPrefix("Hello")
  printMessageWithPrefix(prefix = "Log", message = "Hello")
  println(sum(1, 2))
}
```

Variables

```
// Variable mutable tipada e inicializada
var a: String = "initial"

// Variable inmutable tipada e inicializada
val b: Int = 1

// Variable inmutable inicializada con inferencia de tipo
val c = 3

// Variable mutable declarada con tipo sin inicializar
var d: Int

// Variable inmutable declarada con tipo sin inicializar
val e: Int

// No se rompe la inmutabilidad con la asignación condicional
if (someCondition()) { e = 1 } else { e = 2 }
```

Null Safety

```
var neverNull: String = "Esta variable no puede ser null"
neverNull = null // Compilation Error

var nullable: String? = "Esta variable puede ser null"
nullable = null  // Todo bien

var inferredNonNull = "La inferencia de tipo no permite nullable"
inferredNonNull = null // Compilation Error

// Argumento no nullable
fun strLength(strNotNull: String): Int = strNotNull.length
strLength(neverNull) // Todo bien
strLength(nullable)  // Compilation Error

// Argumento nullable
fun describeString(maybeString: String?): String {
  if (maybeString != null) { return "Caracteres = ${maybeString.length}" }
  else { return "Eso no es un string, es null" }
}

// Argumento y resultado nullable
fun strLengthNulleable(maybeString: String?): Int? = strNotNull?.length
```

Clases (I)

```
// Si se declara una clase "vacía", se pueden omitir las llaves
class Customer

// El header puede contener las propiedades
// y funciona como constructor
class Contact(val id: Int, var email: String)

fun main() {
  val customer = Customer() // No se necesita new, sí los ()
  val contact = Contact(1, "mary@gmail.com")

  // Accessors automáticos
  println(contact.id)
  contact.email = "jane@gmail.com"
}
```

Clases (II): Herencia

```
class FinalDog // Por defecto las clases son final
class Bulldog : FinalDog // Error de compilación, Dog es final

// Para que pueda ser heredada hay que marcarla como open
open class Dog {
  // Lo mismo para los métodos, por defecto no se pueden sobreescribir
  open fun sayHello() = println("wof wof!")
  fun finalHello() = println("soy irremplazable")
}
class Yorkshire : Dog() {
  override fun sayHello() = println("wif wif!")

  // Error de compilación
  override fun finalHello() = println("soy irremplazable")
}
fun main() {
    val dog: Dog = Yorkshire()
    dog.sayHello() // Output: wif wif!
}
```

Uso del Condicional (if)

- En Kotlin `if` **no es** una **estructura de control** sino una **expresión**.
- Con lo cual siempre **retorna un valor**.

```
var max = a

// Uso tradicional
if (a < b) {
  max = b
}

if (a > b) {
  max = a
} else {
  max = b
}
```

```
// Uso como expresión
val max = if (a > b) a else b

// Si se utilizan bloques,
// el valor del bloque
// es la última expresión
val max = if (a > b) {
  print("a es mayor")
  a
} else {
  print("b es mayor")
  b
}
```

Igualdad

- = asignación
- == comparación estructural
- === comparación referencial

```
class User(val name: String) {
  override fun equals(other: Any?): Boolean =
    other is User && other.name == this.name
  override fun hashCode(): Int =
    name.hashCode()
}
```

```
val blue = "Blue"
val sameBlue = "Blue"
println(blue == sameBlue) // true
println(blue === sameBlue) // true, son la misma referencia

val numbers = setOf(1, 2, 3)
val sameNumbers = setOf(2, 1, 3)
println(numbers == sameNumbers)  // true, son conjuntos iguales
println(numbers === sameNumbers) // false, son distintas referencias

val bart = User("Bart")
val bort = User("Bart")
println(bart == bort) // true, se llaman igual
println(bart === bort) // false, son distintas instancias
```

Loops (for, while, do while)

- Funcionan de manera similar a otros lenguajes

```
val cakes = listOf(
  "frutilla",
  "merengue",
  "chocolate"
)

// Se puede iterar directamente
// sobre los elementos
for (cake in cakes) {
  println("Torta de $cake")
}

// Y también sobre los índices
for (i in cakes.indices) {
    println(cakes[i])
}
```

```
var cakesEaten = 0
while (cakesEaten < 5) {
  eatACake()
  cakesEaten++
}

var cakesBaked = 0
do {
  bakeACake()
  cakesBaked++
} while (cakesBaked < 5)
```

Rangos

```
// Rango ascendente
for(i in 0..3) print(i)
// 0 1 2 3

// Rango ascendente con salto
for(i in 2..8 step 2) print(i)
// 2 4 6 8

// Rango descendente
for (i in 3 downTo 0) print(i)
// 3 2 1 0

// Rango descendente con salto
for (i in 9 downTo 0 step 3)
  print(i)
// 9 6 3 0
```

```
// Rango ascendente de caracteres
for (c in 'a'..'d') print(c)
// a b c d

// Rango desc. de caract. con salto
for (c in 'z' downTo 's' step 2)
  print(c)
// z x v t

// Se puede evaluar pertenencia
val x = 2
val inRange =
  if (x in 1..10) print(x)
// true, 2

if (x !in 1..4) print(x)
// false
```

Clases Especiales (I): Data Classes

```
// Las Data Classes son ideales para almacenar información
// Definen automáticamente métodos como:
// equals(), copy(), hashCode(), toString()
data class User(val name: String, val id: Int)
fun main() {
  val user = User("Alex", 1)
  println(user) // User(name=Alex, id=1)
  val secondUser = User("Alex", 1)
  val thirdUser = User("Max", 2)

  println(user == secondUser) // true
  println(user == thirdUser)  // false

  // copy() genera una nueva instancia con las mismas propiedades
  println(user.copy()) // User(name=Alex, id=1)
  // Pero se pueden sobreescribir las propiedades deseadas
  println(user.copy("Max"))  // User(name=Max, id=1)
  println(user.copy(id = 2)) // User(name=Alex, id=2)
}
```

Clases Especiales (II): Object

- Un `Object` se puede pensar como la **única instancia** de una Clase
- Es en definitiva una aplicación del patrón **Singleton**

```
// Singleton of Prices
object Prices {
  var standard: Int = 30
  var holiday: Int = 50
  var special: Int = 100
  fun average() = sumAll() / 3
  private fun sumAll() =
      standard + holiday + special
}

fun main() {
  println(Prices.standard) // 30
  println(Prices.average()) // 60
}
```

```
fun main() {
  // Se puede utilizar como una expresión
  // con propiedades y comportamiento
  val unique = object {
    var unique: Int = 42
    fun average() =
      (Prices.average() + unique) / 2
  }
  println(unique.average()) // 51

  // Se le puede cambiar el estado
  // (porque es una instancia ;)
  unique.unique = 0
  println(unique.average()) // 30
}
```

Colecciones (I): Listas & Conjuntos

- Una **`list`** es una colección *ordenada* de elementos.
- Puede ser mutable o inmutable.
- Un **`set`** es una lista sin repetidos.

```
val roots: List<String> = listOf("root", "groot") // Inmutable
val sudoers: MutableList<String> = // Mutable
                    mutableListOf("tony", "steve", "bruce")
val planets: Set<String> = // Inmutable
     setOf("tierra", "titan", "titan", "asgard", "tierra")

fun main() {
  roots.add("fury") // <- Compilation Error
  sudoers.add("carol")
  println("${roots.size}") // 2

  for (root in roots) println(root)   // root, groot
  sudoers.forEach { e -> println(e) } // tony, steve, bruce, carol
  planets.forEach { println(it) }     // tierra, titan, asgard
}
```

Colecciones (II): Maps

- Un **map** es una colección clave/valor. Puede ser mutable o inmutable.

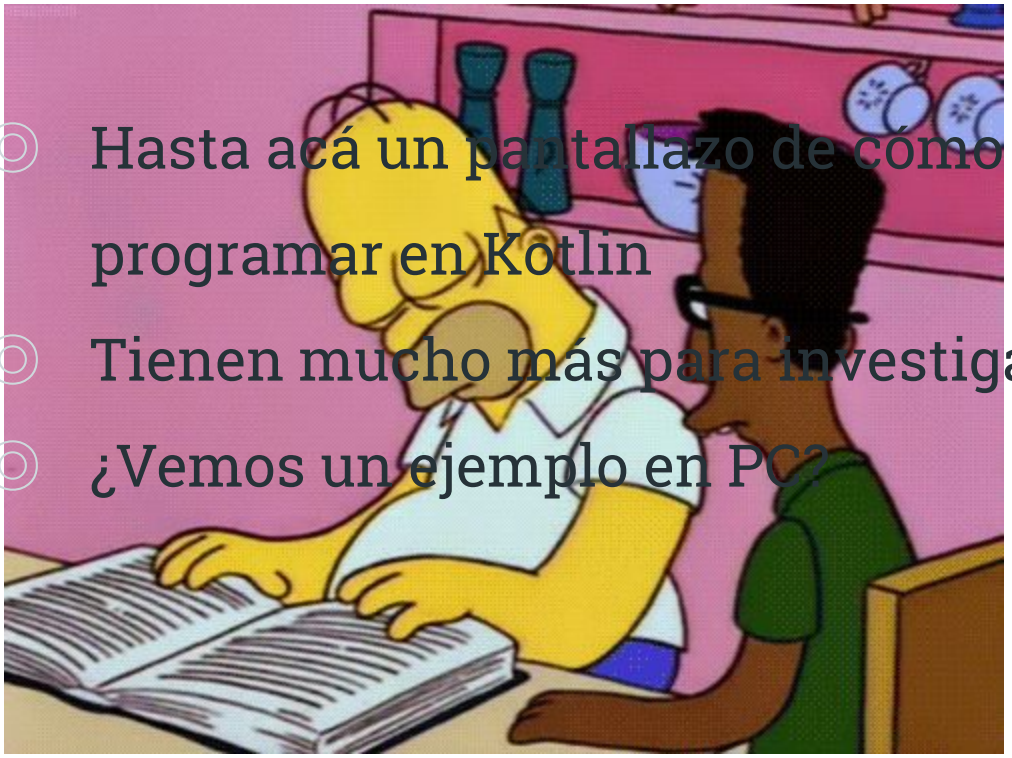
```
val codes: Map<Int, String> = mapOf(1 to "Intro", 2 to "Orga", 3 to "Mate1")
val approved: MutableMap<String, Int> =
     mutableMapOf("Intro" to 10, "Orga" to 8, "Obj1" to 9)

fun approved(code: Int) = approved.containsKey(codes[code])
fun score(name: String): Int? =
      if (approved.containsKey(name)) approved.getValue(name) else null

fun main() {
  println(approved(1))    // true
  println(approved(3))    // false
  println(score("Intro")) // 10
  println(score("UI"))    // null
  approved.forEach { key, value -> println("$key: $value") }
  // Intro: 10, Orga: 8, Obj1: 9

  codes.forEach { println("${it.key}: ${it.value}") }
  // 1: Intro, 2: Orga, 3: Mate1
}
```

Operaciones con Colecciones

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

Filter	`list.filter { it > 2 }  // [3]`
Map	`list.map { it * 2 }  // [2,4,6]`
Any, All, None	`list.any { it == 2 }  // true`
Find, FindLast	`list.find { it > 3 }  // null`
First, Last	`list.first()  // 1`
Count	`list.count { it > 1 }  // 2`
Min, Max	`list.max()  // 3`
Sorted	`list.sortedBy { -it }  // [3,2,1]`
Get or Else	`list.getOrElse(7) { 42 }// 42`

Uff...



- Hasta acá un pantallazo de cómo se puede programar en Kotlin
- Tienen mucho más para investigar
- ¿Vemos un ejemplo en PC?