# Unidad 0
## Introducción a Kotlin

Javier Carrasco Navarro

# ¿Qué es Kotlin?

- Lenguaje creado en 2011 por **JetBrains** (creadores, entre otros, del entorno **IntelliJ IDEA** para Java)
- De propósito general, multiplataforma, con tipado estático e inferencia de tipos.
- Web oficial: https://kotlinlang.org/
- Tutoriales oficiales: https://kotlinlang.org/docs/reference/
- Para probar online: https://play.kotlinlang.org/

# Generalidades

- Hola, mundo:

```
fun main() {
    println("Hello, world!!!")
}
```

- El punto y coma final es opcional
- Los comentarios con /* ... */  y con //
- Null Safety (?, !!)

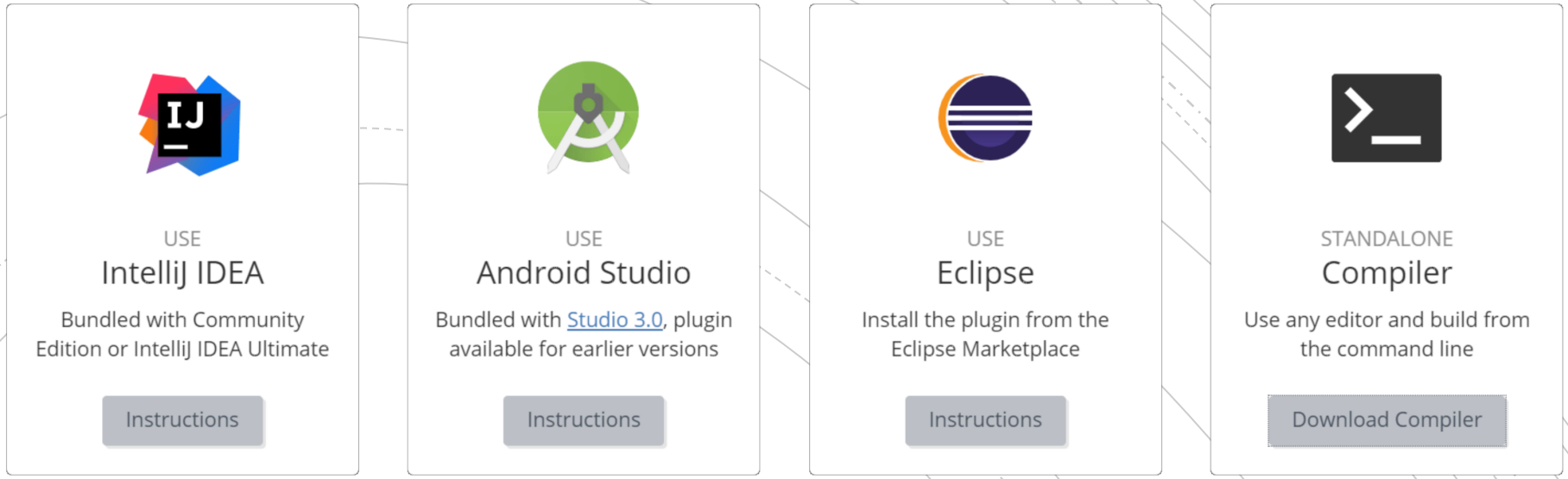
# Entornos online

No es necesario “instalar Kotlin” para probarlo.

Para apps Android, es parte de Android Studio.

Para pruebas rápidas, entornos online: play.kotlinlang.org, paiza.io, repl.it, rextester.com, ...

La mayoría de ellos no son interactivos (sin “readline”).




# Entornos offline

# Expresividad

**Java**

```java
public class Artista {
    private long id;
    private String nombre;
    private String url;
    private String mbid;

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getUrl() {
        return url;
    }

    public void setUrl(String url) {
        this.url = url;
    }

    public String getMbid() {
        return mbid;
    }

    public void setMbid(String mbid) {
        this.mbid = mbid;
    }

    @Override
    public String toString() {
        return "Artista{id=" + id + ", nombre='" + nombre + '\'' + ", url='" + url + '\'' + ", mbid='" + mbid + '\'}';
    }
}
```

**Kotlin**

```kotlin
data class Artista(
    var id: Long,
    var nombre: String,
    var url: String,
    var mbid: String
)
```

# Variables

- Se declaran con **var** → `var i: Int = 42`
- Si se conoce el valor inicial, el tipo es opcional → `var i = 42`
- No hay conversión automática:

```
var d: Double = i   // Error

val d: Double = i.toDouble()   // Ok
```

- Las constantes, con **val** → `val i = 53`
- Para evitar la obligatoriedad en la instanciación se utiliza la palabra reservada **lateinit**.

| | | |
|---|---|---|
| Byte | 8 bit | -128 a 127 |
| Short | 16 bit | -32768 a 32767 |
| Int | 32 bit | -2,147,483,648 a 2,147,483,647 |
| Long | 64 bit | -9,223,372,036,854,775,808 a +9,223,372,036,854,775,807 |
| Float | 32 bit | 1.40129846432481707e-45 a 3.40282346638528860e+38 |
| Double | 64 bit | 4.94065645841246544e-324 a 1.79769313486231570e+308 |

# Tipos numéricos

# Actividad 0001 Operaciones básicas

- Muestra la suma, resta, multiplicación y división de dos enteros largos prefijados, así como el resto de dividir el primero entre el segundo.

# Otros aspectos

- Char
- String
- Boolean

```
var a: String = "abc" // No puede tener valores nulos

var b: String? // Puede tener valores nulos
```

# Leer de consola

- Tenemos definida una función “readLine”, que devuelve un string (o null):

```
fun readLine(): String?
```

- En su uso real, normalmente deberemos añadir “!!” al final, para permitir que salte una excepción si fuera null:

```
var numero = readLine()!!.toInt()
```

# Ejemplo de consola

```
fun main() {
    println("Dime el primer dato")
    var n1 = readLine()!!.toInt()
    println("Dime el segundo dato")
    var n2 = readLine()!!.toInt()
    println("Su suma es")
    println(n1+n2)
}
```

# Otros IDE online

- Algunos IDEs ya son interactivos. Por ejemplo,
  - https://www.jdoodle.com/compile-kotlin-online/
  - https://replit.com/languages/kotlin
- Otros tienen una pestaña de entrada (paiza.io) y otros no permiten entrada de ningún tipo (play.kotlinlang.org).

# Alternativas

- Alternativa más compacta: indicar entre comillas: + nombre de variable

```
println("Su suma es" + (n1+n2))
```

- Más “al estilo Kotlin”: prefijo $

```
val suma = n1 + n2

println("Su suma es $suma")
```

- También puede ser una expresión entre llaves

```
println("Su suma es ${n1+n2}")
```

- Para mostrar el $: \$
- Para número de decimales:

```
println("Su suma es ${String.format("%.2f", n1+n2)}")
```

# Condiciones

```
if (a > b) {
    println(a)
} else {
    println(b)
}
```

```
var mayor = if (a > b) a else b
```

# Actividad 0002
## ¿Par?

- Indica si un número es par o impar, usando la sintaxis de *if* como expresión, en vez la sintaxis clásica.

# When (Switch de Java)

```
when (x) {
    0, 1 -> print("0 o 1")
    in 2..10 -> print("2 al 10")
    else -> print("otros")
}
```

- Permite incluso comprobar tipo:

```
is MiClase -> print("...")
```

# Actividad 0003
# Día de la semana

- Muestra el nombre de un día de la semana a partir de su número (del 1 al 7), usando “when”. El sábado y el domingo deberán aparecer como “fin de semana” y se deberá avisar si el número no es válido.

# Bucle While

```
while (x > 0) {
    x--
}  // existen break y continue


do {
    val y = recibirDatos()
} while (y != null) // y es visible aunque se declare dentro
```

# Bucle For (1)

- Ascendente:

```
for (i in 1..3) {   // 1 y 3 incluidos
    println(i)
}

for (i in 1 until 3) {   // 3 no incluido
    println(i)
}
```

# Bucle For (2)

- Descendente:

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

- Para recorrer listas o colecciones:

```
for (item in collección) print(item)
```

# Actividad 0004
## Tabla de multiplicar

- Escribe la tabla de multiplicar de un número introducido por el usuario, utiliza el bucle *for*.

# Actividad 0005
## Letras de una cadena

- Escribe las letras de una cadena de texto, una a una, separadas por espacios, usando *for*.

# Funciones (1)

- En las “funciones void” no hay que indicar tipo devuelto:

```
fun saludar() {
    println("Hola")
}
...
saludar()
```

# Funciones (2)

- Parámetros: tipo tras el nombre de variable
- Valor devuelto: tras la declaración

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

val resultado = doble(3)
```

# Funciones (3)

- Si una función es sólo una expresión: sin llaves, con “=”

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

- En ocasiones, se puede omitir el tipo devuelto

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

# Funciones (4)

- Posibilidad de usar parámetros con nombre y valores por defecto (no sólo al final de la llamada):

```
fun potencia(nExp: Int = 1, nBase: Int) { /*...*/ }

potencia(nBase = 2)
```

# Funciones (5)

- **Funciones de extensión**, este tipo de funciones permiten añadir funcionalidades a clases ya existentes sin necesidad de editar su código.

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

- Uso:

```
val fragment: Fragment
fragment.toast("Hello world?!")
```

# Actividad 0006
## ¿Primo?

- Crea una función que permita saber si un cierto número, que se le pase como parámetro, es primo. Úsala desde un programa.

# Arrays (1)

- Con datos prefijados: ArrayOf

  val numeros = arrayOf("Uno", "Dos", "Tres")

- Recorrer todo: for (n in numeros) println(n)
- Posiciones sueltas:

  println(numeros[1])

  println(numeros.get(1))

# Arrays (2)

- Crear a partir del tamaño, sin datos (se rellena con ceros):

```
val datos = IntArray(10)
```

- Crear a partir de datos individuales:

```
val datos: IntArray = intArrayOf(1, 2, 3)
```

- Sin indicar tipo base:

```
val numeros = arrayOf("Uno", "Dos", "Tres")
```

# Arrays (3)

- Rellenar con un operador Lambda:

  val datos = IntArray (10) { i -> i + 10 }

- Crear otros sin valores iniciales es más complejo:

  val profesores : Array<String?> = arrayOfNulls(2)

  profesores[0] = "Javier"; profesores[1] = "Nacho"

# Arrays (4)

- Tamaño: numeros.size
- Modificar:

  numeros[2] = "TresB"

  numeros.set(1,"DosB")

- Existen tipos especializados, más rápidos: ByteArray, ShortArray, IntArray:  val miArray = IntArray(5)

# Arrays (5)

- Avanzado: extraer posición y valor como un par:

```
for ((posicion, valor) in numeros.withIndex())
    println("$posicion = $valor")
```

# Actividad 0007 Array

- Prepara un array con los días de la semana. El usuario introducirá un número del 1 al 7 y le mostrarás el día correspondiente. Si introduce un valor incorrecto, le mostrarás los 7 valores aceptables (“1=lunes”, etc).

# Listas (1)

- Podemos usar listas inmutables y listas mutables.

- Las inmutables recuerdan mucho a los arrays:

```
val secciones: List<String> = listOf("Multimedia", "Móviles")
println(secciones.size)
println(secciones[1])
```

# Listas (2)

- Las listas mutables permiten añadir datos con “.add”, al final o en una posición intermedia:

```
val secciones: MutableList<String> =
        mutableListOf("Multimedia", "Móviles")
secciones.add("Proyecto  final")
secciones.add(1, "Proyecto de Unity")
```

# Listas (3)

- Varias formas de recorrer una lista. Dos de las más sencillas:

```
for (s in secciones) println(s)
```

*y...*

```
secciones.forEach { println(it) }
```

- (Y más alternativas. Por ejemplo, usando “iterators”).

# Listas (4)

- Operaciones avanzadas: filter, map, any, count, find, max, min...
- Por ejemplo:

```
val seccionesFiltradas = secciones.Filter {
    it.contains("Unity")
}
val seccionesMayusculas = seccionesFiltradas.map {
    it -> it.uppercase()
}
```

# Actividad 0008
## Listas

- Crea un programa que permita al usuario introducir cadenas de texto y las guarde en una lista, hasta terminar con “fin” (que no se almacenará). A continuación, muestra la lista en orden inverso (del último dato al primero). Finalmente, muéstrala en su orden normal usando *“.forEach”*.

# Mapas (diccionarios)

- Los “mapas” son equivalentes a los “diccionarios” de C#:

```
val numeros = mutableMapOf("uno" to 1, "dos" to 2)
numeros.put("tres", 3)
numeros["uno"] = 11
println(numeros)    // Salida: {uno=11, dos=2, tres=3}
```

- Sintaxis alternativa:

```
val numeros = mapOf(Pair("uno",1), Pair("dos",2))
```

# Conjuntos

- Para datos que no vayan a tener duplicados:

```
val setOfItems = mutableSetOf<Int>()
setOfItems.add(1)
setOfItems.add(2)
setOfItems.add(3)
println(setOfItems) // Salida: [1, 2, 3]
```

# Clases (1)

- En Kotlin las clases no tienen campos, sino propiedades, que equivalen al campo + *getter* + *setter*:

```
class Persona {
    var nombre: String = ""
    var apellido: String = ""
}
```

- Uso: `var yo = Persona()`

# Clases (2)

- Constructor (principal): “init”.

```
class Persona(nombre: String, apellidos: String) {
    var nombre: String
    var apellidos: String

    init {
        this.nombre = nombre
        this.apellidos = apellidos
    }
}
```

# Clases (3)

- Se distingue entre “constructor principal” (init) y “secundarios”:

```
constructor(nombre: String, apellido:
String, edad: Int): this(nombre,apellido) {
    this.edad = edad
}
```

- Uso: `val yo = Persona("Pepe", "Botica", 58)`

# Clases (4)

- Métodos: funciones con sintaxis normal o abreviada.

```
class Persona(nombre: String, apellido: String) {
    ...
    fun getNombreCompleto() = "$nombre $apellido"
}

fun main() {
    ...
    println(persona2.getNombreCompleto())
}
```

# Clases (5)

- Pero se pueden crear comportamientos específicos para los *getter* y *setter*:

```
class Persona(nombre: String, apellidos: String) {
    var localidad: String = ""
    set(value) {
        field = if (value == "") "No indicada"
                else value
     }
     get() = "Localidad: $field"
     ...
```

# Clases (6)

- Una clase sólo para almacenar datos es una “data class”, entre paréntesis, sin llaves:

```
data class Persona(var nombre:
String, var apellido: String)
```

- Eso crea de manera automática los *getters*, *setters*, *toString*, *equals* y *hashcode*.

# Clases (7)

- Existe un tipo de clases llamadas *“sealed class”*, este tipo de clases no permiten crear más hijos fuera del proyecto en el que se han creado.

```
sealed class Vehiculo(var nRuedas: Int)

data class Motocicleta(var ruedas: Int = 2):
Vehiculo(ruedas)

data class Turismo(var ruedas: Int = 4, var
puertas: Int = 2): Vehiculo(ruedas)
```

- Este tipo de clases evita comprobaciones del tipo “otro tipo”, por ejemplo, el *else* en una estructura *when*.
- Para que se pueda heredar de una clase, la clase padre debe tener la palabra reservada ***open*** delante de *class*.

# Clases (8)

- La clase ***Pair*** en Kotlin se utiliza para representaciones genéricas de pares.

```
val parUno = Pair("Hola", "Mundo")

val parDos = Pair("Adiós amigos", 150)

val (usuario, contrasenya) = Pair("javier", "kotlin")
```

- Uso:

```
println("Par UNO-1: ${parUno.first} ?| Par UNO-2:
${parUno.second}")

println("Par DOS-1: ${parDos.first} ?| Par DOS-2:
${parDos.second}")

println("Usu: $usuario - Pass: $contrasenya")
```

# Actividad 0009
## Clases I

- Crea una clase de datos “coche” con marca, modelo y año de lanzamiento. Crea dos objetos de esa clase.

# Actividad 0010
# Clases II

- Crea una clase libro (autor, título, año), con *setters* que dejen los valores por defecto “Anónimo”, “No indicado” y -1. Crea también un método que devuelva los tres datos en una misma línea, comenzando por el título, separados por espacios y guiones, como en **“It – Stephen King – 1986”**.

# Objetos (1)

- Un ***object*** en Kotlin es una variable con una única implementación.
- Serían los elementos *static* de Java.
- Utilizan el patrón *singleton*, por lo que solo existirá una instancia del objeto creado.

# Objetos (2)

- Implementación:

```
object MiObjeto {
   val usuario: String = "Javier"
   val base_URL: String = "http:?/www.javiercarrasco.es/"

   fun funcionDeMiObjeto() {
      println("Has llamado a la función de un objeto.")
   }
}
```

# Objetos (3)

- Uso:

```
val nombreUsuario = MiObjeto.usuario

println(nombreUsuario)

println(MiObjeto.funcionDeMiObjeto())
```

# Companion Object (1)

- Los ***companion object*** pueden implementarse en cualquier clase.
- Son comunes a todas las instancias de la clase.
- Funcionan igual que los *objects*, por lo que sus miembros pueden ser accedidos a través del nombre de la clase que los contiene.
- Utilizan el patrón ***singleton***.

# Companion Object (2)

```
class Empleados(nom: String, ape: String) {
    var idEmpleados: Int
    lateinit var nombre: String
    lateinit var apellido: String

    init {
        println("Init clase Empleado.")
        this.nombre = nom
        this.apellido = ape
        numEmpleados++
        idEmpleados = numEmpleados

    }
```

```
    companion object {
        // Se ejecutará una sola vez.
        init {
            println("Init Companion Object.")
        }

        var numEmpleados = 0
    }
}
```

# Companion Object (3)

- Uso

```
val empleado1 = Empleados("Javier", "Carrasco")
val empleado2 = Empleados("Nacho", "Cabanes")

println("Empleado 1: ${empleado1.nombre} ${empleado1.apellido}")
println("Empleado 2: ${empleado2.nombre} ${empleado2.apellido}")
println("Total empleados: ${Empleados.numEmpleados}")
```