GALAXY

TRAINING

ENTRENAMIENTO
EXPERIENCIA

GALAXY
TRAINING

ANDROID DESDE CERO CON KOTLIN

Ing. Marco Estrella
Instructor Tecnologías Java y Android

mestrella@galaxy.edu.pe

AGENDA

SESIÓN
01 ANDROID DESDE CERO CON KOTLIN

- Historia de kotlin.
- Visión general de kotlin.
- Variables y Constantes.
- Tipos de datos y Conversiones.
- Arreglos.
- Condicionales(if, if else, when).
- Bucles(for, while, do while)
- Funciones.
- Clases e instancias.
- Herencia.
- Sobreescritura de métodos.
- Sobrecarga de métodos.

HISTORIA DE JET BRAINS

Basico

Antes fue IntelliJ.

Fue fundado en el año 2000.

Su sede principal el en República Checa(Praga).

Su producto más famoso es IntelliJ IDEA.

Sus fundadores

° Sergey Dmitriev
° Valentin Kipiatkov
° Eugene Belyaev

HISTORIA DE KOTLIN

Kotlin

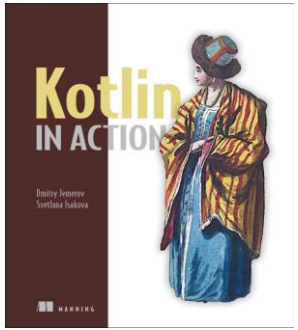
Fue creado por JetBrains

Se reveló en Julio del 2011.

Tiene la licencia Apache 2

Es un proyecto Open Source

Dmitry Jemerov escribió :



Lider de proyecto: Dmitry Jemerov

Oficializado en el Google I/O 2017 con otro lenguaje de programación para el desarrollo de aplicaciones ANDROID aparte de Java.

VISIÓN GENERAL DE KOTLIN

Kotlin es un lenguaje moderno

Open Source

Estaticamente Tipado

Orientado a Objetos

y con características de programación funcional

Podemos ejecutar Aplicaciones en

VISIÓN GENERAL DE KOTLIN

Basico

Sus características son

Concise

Drastically reduce the amount of boilerplate code.

Safe

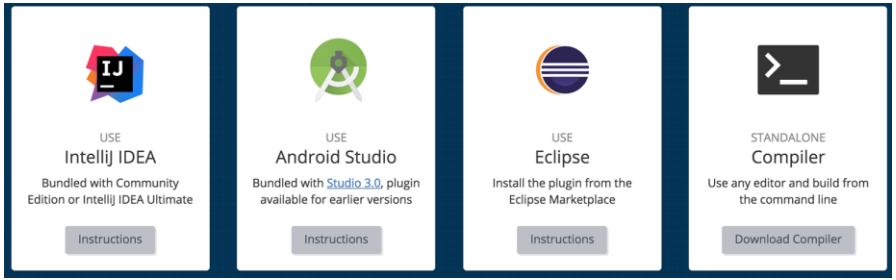
Avoid entire classes of errors such as null pointer exceptions.

Interoperable

Leverage existing libraries for the JVM, Android, and the browser.

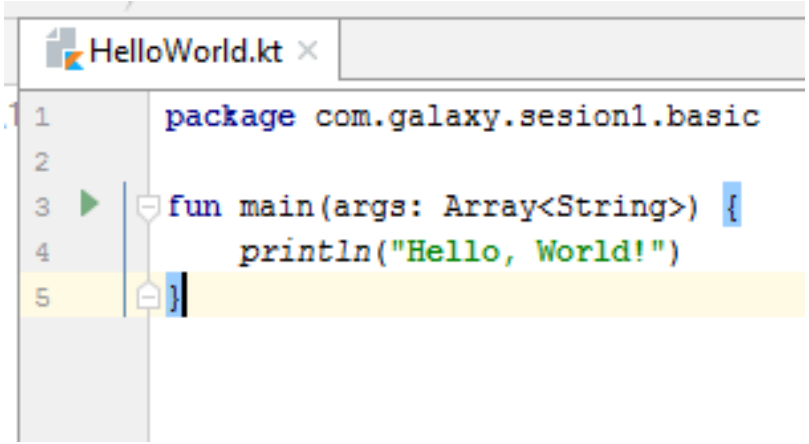
Tool-friendly

Choose any Java IDE or build from the command line.



Usaremos

Hello World

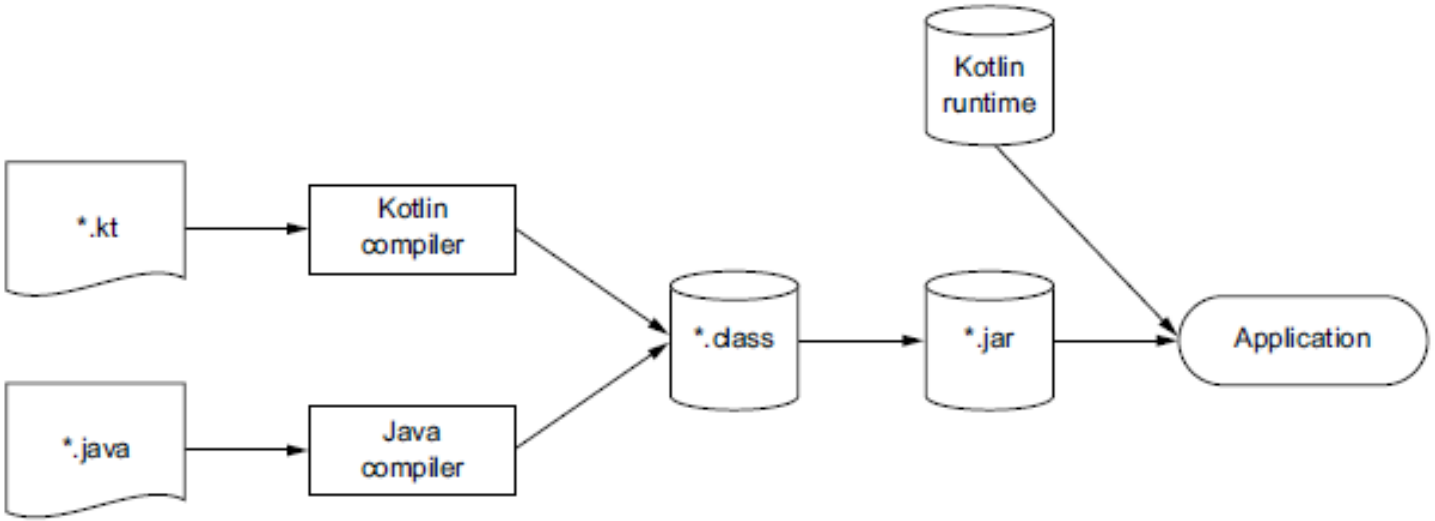


El método main

Es el método por la cual kotlin inicia el programa.

Proceso de construcción de Kotlin

Basico



Vals y Vars

Basico

Kotlin tiene 2 palabras claves para declarar variables, **val** y **var**.

El **var** es una variable mutable, la cual es, una variable que puede ser cambiado a otro valor.

```
var name = "Luisa"

var businessName: String
businessName = "Galaxy Training"
```

El **val** es usado para declarar una variable de solo lectura. es equivalente a crear una variable *final* en java.

```
val PI = 3.14
```

Vals y Vars

Basico

En kotlin no se puede crear múltiples variables en una sola línea

No, there is no way to do that. Declaring multiple properties on the same line is frowned upon by many Java style guides, so we did not implement support for that in Kotlin.

51,259 REPUTATION

10 122 127

yole top 0.10% this year

Principal Engineer at JetBrains

Working or having worked on Kotlin, IntelliJ IDEA, PyCharm and other projects.

answered Jan 27 '16 at 12:41

yole

51.3k • 10 • 122 • 127

Tipos de datos

Los Numéricos

Type	Bit width
Double	64
Float	32
Long	64
Int	32
Short	16
Byte	8

```
fun main(args: Array<String>) {

    val myDouble = 100000.085965
    val myFloat = 15255.2551F
    val myLong = 1000000000L
    val myInt = 1000
    val myShort: Short = 100
    val myByte: Byte = 1

}
```

Usar diferentes sistemas numéricos

Decimal: 15

A los **Long** set tiene que añadir la L al final: **15L**

Hexadecimales: 0x0F

Binarios: 0b00001111

Se puede usar el ‘_’

```
val oneMillion = 1_000_000
val creditCardNumber = 1234_5678_9012_3456L
val socialSecurityNumber = 999_99_9999L
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
```

Caracteres

Los caracteres son representados por el tipo **Char.** Ellos no son tratados directamente como números.

Se usan apóstrofos para la asignación de su valor.

```
val myChar = 'C'

val myChar2 = '8'

if(myChar2 in '0'..'5')
    println("myChar2 esta dentro del rango")
else
    println("myChar2 no esta dentro del rango")

val myInt: Int = myChar2.toInt()

print("int = $myInt")
```

Booleano

El tipo Boolean representa a los booleanos, y tiene 2 valores: **true** y **false**.

Sus operaciones:

- || - disyunción
- && conjunción
- ! - negación

Conversiones de tipos

Para hacer las conversiones entre wrappers de tipos primitivos de usan los métodos que el tipo ya tiene.

- `toByte(): Byte`
- `toShort(): Short`
- `toInt(): Int`
- `toLong(): Long`
- `toFloat(): Float`
- `toDouble(): Double`
- `toChar(): Char`

```
val myDouble = 100000.085965
val myFloat = 15255.2551F

val myFloat2 = myDouble.toLong()

val myLong = 1000000000L
val myInt = 11111111
val myShort: Short = 100
val myByte: Byte = 1

val myShort2 = myInt.to
print(myShort2)

l myChar = 'C'

val myChar2 = '
```

```
to(that: B) fo
toShort()
toByte()
toChar()
toDouble()
toFloat()
toInt()
toLong()
toString()
```

Arreglos o Arrays

Basico

Un array es una colección de un número establecido de valores. Se puede acceder a los elementos mediante un índice. La indexación del array comienza en 0.

Se pueden crear Array de varias formas:

Usando la función genérica arrayOf()

```
val myArrayExample = arrayOf(1, 2, 3, 4, 5, 6)
var myArrayExample2 = arrayOf('a', 'b', 'c')
var myArrayExample3 = arrayOf(true, false, true)
```

Arreglos o Arrays

Basico

Usando las funciones específicas se acuerdo al tipo de dato.

```
val myDoubleArray = doubleArrayOf(100.255, 2.66, 3.85, 4.63)
val myFloatArray = floatArrayOf(1.5f, 2.8F, 38.63f, 44.6f)
val myLongArray = longArrayOf(1000L, 20000L, 30000L, 40000L)
val myIntArray = intArrayOf(1,2,3,4)
val myShortArray = shortArrayOf(1,2,3,4)
val myByteArray = byteArrayOf(1,2,3,4)
val myCharArray = charArrayOf('a', 'b', 'c')
val booleanArray = booleanArrayOf(true, true, false)
```

Usando la clase Array con el tipo específico a crear.

```
val myDoubleArray2 = DoubleArray( size: 5)
myDoubleArray2[0] = 22.55

val myFloatArray2 = FloatArray( size: 5)
val myLongArray2 = LongArray( size: 8)
val myIntArray2 = IntArray( size: 2)
val myShortArray2 = ShortArray( size: 4)
val myByteArray2 = ByteArray( size: 3)
val myChartArray2 = CharArray( size: 10)
val booleanArray2 = BooleanArray( size: 2)
```

Operaciones con los Arreglos

Función Foreach

```
myCharArray.forEach { it: Char
    println(it)
}
```

Función Foreach con Índice

```
myCharArray.forEachIndexed { index, item ->
    println("$index - $item")
}
```

Asignar valor a un ítem

```
myDoubleArray2[0] = 22.55
```

Obtener valor a un ítem

```
println(myDoubleArray2[0])
```

Obtener cantidad de ítems

```
myDoubleArray2.size
myDoubleArray2.count()
```

String

Comillas simples

```
val myString = "Hola Mundo"

val myString2: String
myString2 = "Hello World"
```

Comillas triples

```
val myStringMultiLineas = """
    Linea numero uno
    Linea numero dos
    Linea numero 3
    Linea numero 4
"""
```

Secuencias de escape

\n -----> Salto de Línea
\t -----> Tabulador
\r -----> Borra a la izquierda
\\ -----> El carácter barra inversa \
\' -----> Comilla simple '
\" -----> Comillas dobles "
\$ -----> Une ($) a una cadena

El signo Dólar ($)

Permite concatenar una variable dentro de una cadena.

```
val edad = 25

print("edad = $edad")
```

INTRODUCCIÓN A NULL SAFETY

Basico

°Permite que nuestras variables sean nulos, simplemente debemos emplear el carácter **?** luego de iniciar el tipo de dato.

°En kotlin los datos por defecto son not-null.

Operador de Invocación segura

Emplea el operador **?**.

Gestiona los nulos de forma segura .

```
var nombre: String?
nombre = null
```

El Operador ! ! .

Convierte cualquier valor a un tipo no nulo y lanza una excepción si el valor es nulo.

```
var nombre: String? = null
var longitud: Int = nombre!!.length
```

El Operador Elvis

Emplea el operador **?:** .

Provee un valor alternativo si encuentra un nulo .

```
var nombre: String? = null
var longitud: Int = nombre?.length ?: 0
println("Longitud: ${longitud}")
// Longitud: 0
```

Condicionales

Basico

Condicional **if**

```
val x = 5

if(x > 2){
    println("x es mayor que 2")
}

if(x > 3) println("x es mayor que 3")
```

Condicional **if else**

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

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

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

Condicionales

Basico

if else if

```
val number = 18

if(number > 0){
    println("Es positivo")
}else if(number < 0){
    println("Es Negativo")
}else {
    println("Es cero")
}
```

if anidado

```
val edad = 40

if(edad >= 12){
    if(edad < 18)
        println("Usted es un adolescente")
    else
        println("Usted es un mayor de edad")
}else{
    println("Usted es un Niño")
}
```

Expresión WHEN

Basico

Estructura básica

```
val x2 = 3
when(x2){
    1 -> println("x == 1")
    2 -> println("x == 2")
    else -> println("x no es 1 ni 2")
}
```

Soporta valores dinámicos

```
val arg = "1"
val x2 = 3
when(x2){
    arg.toInt() -> println("x == 1")
    2 -> println("x == 2")
    else -> println("x no es 1 ni 2")
}
```

When sin argumento

```
val edad = 18

val esAdulto = when{
    edad in 1..17 -> false
    else -> true
}

println(esAdulto)
```

Bucles

Basico

For con un Rango

```
for (x in 1..5){
    println(x)
}

val maxNumber = 10
for ( a in 1..maxNumber){
    println(a)
}
```

```
for (x in 1..10 step 2){
    println(x)
}

val rango = 1..10
for (x in rango.reversed()){
    println(x)
}

val rango = 10 downTo 1
for (x in rango){
    println(x)
}
```

Foreach

```
val nombres = arrayOf("Pedro", "Paola", "María")

for(nombre in nombres){
    println(nombre)
}
```

While

```
var c = 10

while(c > 0){
    println(c)
    c--
}
```

do while

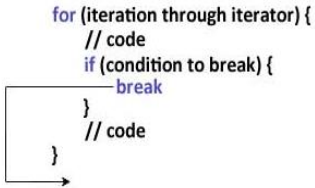
```
var contador = 10

do {
    println(contador)
    contador--
}while (contador > 5)
```

Expresiones de salto

Break

Finaliza el bucle




```
for (elemento :Int in 1..5){
    if (elemento==3){
        break//Rompe el ciclo
    }
     print(" "+elemento)
    //Rpta: 1 2
```

Continue

No termina el el bloque y salta al otro ciclo.

Se usa cuando queremos controlar algo en especifico.

```
for (elemento :Int in 1..5){
    if (elemento==3){
        continue        //regresa al bucle
    }
     print(" "+elemento)
    //Rpta: 1 2 4 5
```

return

Termina la ejecución de una función sin necesidad de ejecutar todo el bloque.

```
fun permitirIngreso(x: Int){
    println("Función Iniciado")
    if(x < 18){
        println("Es menor de edad")
        return
    }
    println("Función Terminado")
}
```

Expresiones de salto con etiquetas

break@<etiqueta>

```
fun main(args: Array<String>) {

    loop@for (contador1 :Int in 1..2) {
            for (contador2 :Int in 1..5){
                println("contador1: $contador1")
                if (contador2==2) {
                    break@loop
                }
            }
     }
}
```

continue@<etiqueta>

```
fun main(args: Array<String>) {

    loop@for (contador1 :Int in 1..2) {
            for (contador2 :Int in 1..5){
                println("contador1: $contador1")
                if (contador2==2) {
                    continue@loop
                }
            }
     }
}
```

Funciones

Basico

Es un conjunto de instrucciones que realizan una determinada tarea

Usan la palabra clave **fun**

```
fun sumar(numero1: Int, numero2: Int): Int {
    return numero1 + numero2
}
```

```
    val numero1=1
    val numero2=5
    fun sumar():Int{
        return numero1 + numero2
    }
```

```
// Sin indicar el tipo de retorno
fun despedirse(nombre: String) {
    println("Adios, " + nombre)
}
```

Sintaxis

```
fun nombreFuncion(parametro1:Tipo, ..., parametroN:Tipo): TipoRespuesta {
    // Cuerpo de la función
}
```

Expresión **Unit**

Indica que la función no devuelve ningún valor.
Su uso es muy parecido al *void* de Java.

sintaxis

```
fun funcionUnit(): Unit {
    //código
}
```

```
// Indicando que la función no devuelve nada
fun saludar(nombre: String): Unit {
    println("Hola, " + nombre)
}
```

Expresión **Nothing**

Se usa para representar un valor que nunca existe.
Siempre arroja una excepción.

Sintaxis

```
fun reportError(): Nothing {
    throw RuntimeException()
}
```

```
fun reportError():Nothing{

    throw  IllegalArgumentException("Ocurrio un error")
}
reportError()
//Rpta: Exception ......IllegalArgumentException: Ocurrio un error
```

Clases y Constructores

Basico

Se usa la palabra clase *class.*

```
class Boleta

class Factura {}

class Persona(nombre: String, edad: Int)
```

Declarar el cuerpo del constructor primario

```
class Animal(nombre: String, paisOrigen: String){

    init {
        println("Cuerpo del contructor primario")
    }
}
```

Constructores Secundarios

```
class Empleado(codigoEmpleado: Int?, nombre: String, role: String){

    constructor(nombre: String, role: String): this( codigoEmpleado: null, nombre, role)
}
```

Declaración de propiedades

```
class Pais(val nombre: String, var poblacion: Int, val continente: String){

    fun resumen(){
        println("El pais $nombre esta en el continente $continente y viven $poblacion personas.")
    }
}

fun main(args: Array<String>) {
  val pais = Pais( nombre: "Peru",  poblacion: 32000000,  continente: "America")
    pais.resumen()
}
```

Modificadores de visibilidad

Basico

Puedes usar modificadores de visibilidades en las declaraciones de clases, variables y métodos.

private .- solo será visible dentro de la clase

protected .- private + visible en las subclases.

internal .- cualquier clase dentro del módulo.

```
// file name: example.kt
package foo

private fun foo() { ... } // visible inside example.kt

public var bar: Int = 5 // property is visible everywhere
    private set         // setter is visible only in example.kt

internal val baz = 6    // visible inside the same module
```

public .- visible en cualquier parte, inclusive fuera del módulo. ***Por defecto es public***.

Herencia de clases

Basico

Las clases, variables y funciones por defecto son ***final***. Lo que significa que no puede ser heredada.

Para hacer que una clase sea heredable y/o un función sobreescribible se le tiene que asignar el keyword ***open***.

Si una función tiene el keyword ***override*** significa que también es ***open***.
Par evitar eso, solo tenemos que añadir el keyword ***final***.

```
class ClaseNieta: ClaseDerivada(){

    override fun funcion1() {
        super.funcion1()
    }
}
```

```
open class ClaseBase {

    open fun funcion1(){
        println("ClaseBase - funcion1")
    }

    fun funcion2(){
        println("ClaseBase - funcion2")
    }
}

open class ClaseDerivada: ClaseBase() {

    final override fun funcion1() {
        super.funcion1()

        println("ClaseDerivada - funcion1")
    }
}
```

Herencia de clases

Basico

Sobreescribiendo propiedades

```
open class ClaseBase {

    open val a: Int = 200

    open val x : Int get() {
        return 100
    }
}

open class ClaseDerivada: ClaseBase() {

    override val a: Int = 2000

    override val x: Int
        get() {
            return 1000
        }
}
```

Inicializadores

```
open class ClaseBase(val nombre: String) {

    init {
        println("Iniciando la clase base")
    }
}

open class ClaseDerivada(nombre: String, apellido: String): ClaseBase(nombre) {

    init {
        println("Iniciando la clase derivada")
    }
}
```

Sobrecarga de métodos

Basico

En Kotlin, podemos asignarle un valor por defecto a los parámetros de las funciones y constructores. Esa funcionalidad nos puede ayudar ha hacer sobrecarga sin necesidad de crear muchas funciones.

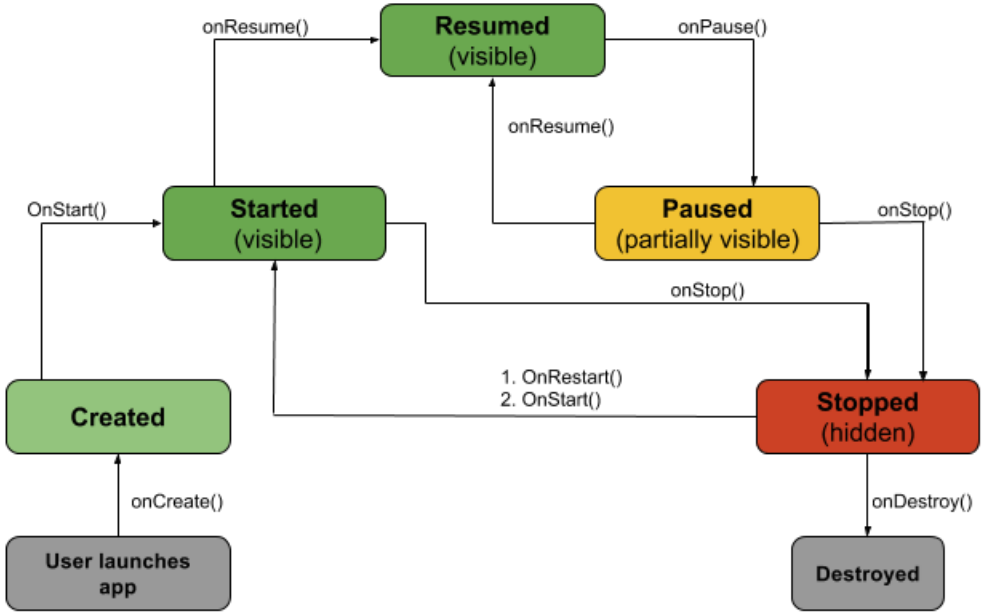
```
fun escribirNombreCompleto(primerNombre: String = "",
                           segundoNombre: String = "",
                           primerApellido: String = "",
                           segundoApellido: String = ""){

    println("$primerNombre $segundoNombre $primerApellido $segundoApellido")
}

fun main(args: Array<String>) {
    escribirNombreCompleto( primerNombre: "Luis",  segundoNombre: "Miguel",  primerApellido: "Torres",  segundoApellido: "Albarado")
    escribirNombreCompleto(primerNombre = "Luis", primerApellido = "Torres", segundoApellido = "Albarado" )
    escribirNombreCompleto(primerNombre = "Luis", primerApellido = "Torres" )
}
```

El ciclo de vida de un activity

Basico



Thank you!
Questions?