



escuela**arte**granada

TEMA 2: Kotlin

Programación Multimedia y Dispositivos Móviles
2º Desarrollo de Aplicaciones Multiplataforma
Profesor: Juan Miguel González Craviotto

Índice

- ¿Por qué Kotlin?
- Principales características
- Intro. a Android Studio
- Hola mundo
- Variables
- Arrays
- Control de flujo
- Funciones
- Excepciones
- Clases
- Colecciones (listas, conjuntos y mapas)
- Depuración en Android Studio
- Buenas prácticas de programación

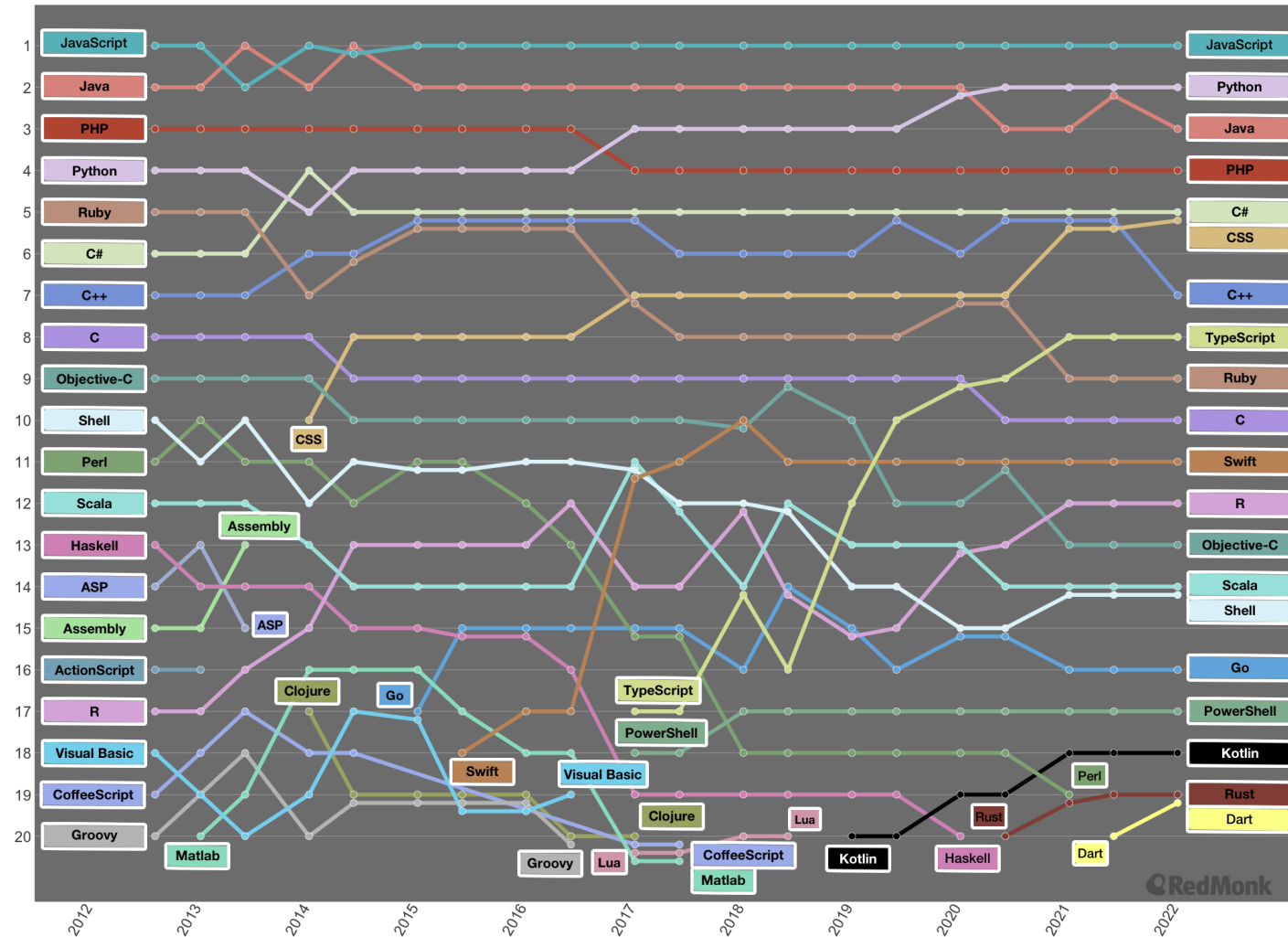
¿Por qué Kotlin?

- Junto a Java y C++, únicos lenguajes nativos para el desarrollo de apps para dispositivos Android
- En 2019 se establece como el lenguaje recomendado por Google para el desarrollo de apps
- Todas las nuevas apps de Google estarán programadas en Kotlin y muchas apps de Google han sido reescritas, parcial o totalmente, de Java a Kotlin
- No solo Google lo usa: Netflix, TikTok, Pinterest, Evernote, Slack, Tinder, Airbnb, Amazon, Plex, Trello, Foursquare, New Relic, Tuenti y muchas otras apps están programadas en Kotlin

¿Por qué Kotlin?

RedMonk Language Rankings

September 2012 - January 2022



¿Por qué Kotlin?

1 JavaScript	11 Swift
2 Python	12 R
3 Java	13 Objective-C
4 PHP	14 Shell
5 C#	14 Scala
5 C++	16 Go
5 CSS	17 PowerShell
8 TypeScript	18 Kotlin
9 Ruby	19 Rust
10 C	20 Perl

Puede parecer poco, pero en 2018 no aparecía en el ranking, en el 2019 apareció en el puesto 20 y en el 2020 subió al 18. Además hay que tener en cuenta que el ámbito principal de Kotlin es el desarrollo de apps

¿Por qué Kotlin?

- En 2019 el 59,43% de las apps programadas para Android tenían un 80% o más de su código programado en Kotlin[1]

[1]<https://www.researchgate.net/publication/334017787> An empirical study on quality of Android applications written in Kotlin language

Kotlin

- **Multiparadigma** (el cual soporta más de un paradigma de programación. Según lo describe Bjarne Stroustrup, permiten crear “programas usando más de un estilo de programación”)
- **Tipado estático**
(cuando la comprobación de tipificación se realiza durante la compilación, y no durante la ejecución)
- **Interoperable con Java**
(posibilita que Kotlin pueda usar todas los frameworks y librerías de Java. O incluso mezclar ambos códigos en un mismo proyecto. Puedes llamar código Java desde Kotlin y viceversa)
- **Compila a bytecode Java 8 que es interpretado por la JVM**
- **Se puede configurar para que compile a código JS**
- **Herramientas para la programación funcional**
- **Sintaxis concisa**
- **Gestión de los valores nulos mejorada con respecto a Java**
- **Programación concurrente sin hebras → uso de corrutinas**

Introducción a Android Studio

Hola mundo

Ejecutar código Kotlin en Android Studio

- Crea un nuevo proyecto sin actividad asociada
- Elige Kotlin como lenguaje de programación
- Crea un nuevo fichero Kotlin y crea la función main que por defecto es la función que se ejecuta al compilar el código Kotlin:

```
1 package com.example.pruebakotlin2
2
3 ▶ fun main(){
4     println("Hola mundo")
5 }
```

Variables

Variables – tipos

- Los tipos de variables básicos más comunes en Kotlin son: números enteros (Int), números enteros largos (Long), números reales de precisión simple (Float), número reales de precisión doble (Double), valores booleanos (Boolean), caracteres (Char) y cadenas de texto (String)

Variables – var vs val

- Para declarar una variable utilizar var o val, según su valor pueda cambiar o no.
- Utilizamos var cuando la variable puede cambiar su valor en tiempo de ejecución
- Utilizamos val cuando una vez asignado el valor no puede cambiar, por lo tanto cuando es una constante

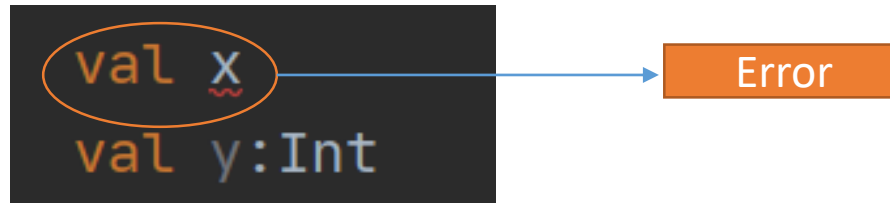
Variables – declaración de variables

- Podemos especificar el tipo de variable o podemos dejar que Kotlin lo infiera por nosotros:

```
val x = 5  
val y:Int = 3
```

- Sin embargo si no le asignamos un valor inicial estamos obligados a definir su tipo (no se puede inferir nada):

```
val x  
val y:Int
```



Variables - casting

- Para cambiar el tipo de una variable utilizamos los métodos `toInt()`, `toFloat()`, `toDouble()` y `toString()`

```
var a = 5.3  
var b:Int = a.toInt()  
println(b)
```

Variables - casting

- Sin embargo muchas veces no es necesario realizar el casting, a diferencia de Java, Kotlin es capaz de realizar automáticamente el casting

```
val a = 19  
println("Tengo "+a+" años")
```


Variables - casting

- Para comprobar el tipo de una palabra utilizamos 'is':

```
val a = 7
if(a is Int){
    println("La variable es un entero")
}
else{
    println("La variable no es un entero")
}
```

Variables - Strings

- En Kotlin existe el concepto de Strings Templates:

```
val a = 7
println("La variable tiene el valor $a")
```

- Podemos incluso ejecutar instrucciones con {} y su valor será puesto en el String

```
val provinciasAndalucia =
    arrayOf("Granada", "Málaga", "Jaén", "Almería", "Sevilla", "Cádiz", "Córdoba", "Huelva")
println("El primer elemento del array es ${provinciasAndalucia[0]}")
println("El número de elementos del array es ${provinciasAndalucia.size}")
```

Variable - lateinit

- A diferencia de en Java cuando una variable se declara se debe inicializar.
- Esto puede resultar problemático, por ejemplo al definir propiedades de una clase que son inicializadas posteriormente a la creación del objeto mediante inyección de dependencias.
- Para solventar este problema utilizamos la palabra reservada 'lateinit', anteponiéndola a 'var'

Arrays

Arrays

- Conjunto ordenado de valores de un mismo tipo
- Su tamaño es fijo
- Para crear un array usamos la función `arrayOf()`

```
val array = arrayOf(1, 7, 3)
val array2 = arrayOf("Granada", "Sevilla", "Málaga")
```

- Para conocer la longitud de un array consultamos su parámetro `.size`

Arrays

- También podemos crear arrays especificando su tipo

```
val array = intArrayOf(1, 7, 3)  
val array2 = doubleArrayOf(2.3, 2.0, 3.3)
```

- Lo cual nos permite crear arrays de un determinado tipo sin tener que inicializar los valores

```
val array = IntArray(size: 5)  
val array2 = FloatArray(size: 4)
```

Para pensar...

- ¿Es correcto el siguiente código?

```
val provincias =  
    arrayOf("Granada", "Málaga", "Jaén", "Almería", "Sevilla", "Cádiz", "Córdoba", "Huelva")  
provincias[3] = "hola"
```

- O... ¿se debería usar var en vez de val?

```
fun main() {  
    val sueldos: IntArray  
    sueldos = IntArray( size: 5)  
    //carga de sus elementos por teclado  
    for(i in 0..4) {  
        print("Ingrese sueldo:")  
        sueldos[i] = readln().toInt()  
    }  
    //impresion de sus elementos  
    for(i in 0..4) {  
        println(sueldos[i])  
    }  
}
```



```
fun main() {  
    val alturas = FloatArray( size: 5)  
    var suma = 0f  
    for(i in 0..alturas.size-1){  
        print("Ingrese la altura:")  
        alturas[i] = readln().toFloat()  
        suma += alturas[i]  
    }  
  
    val promedio = suma / alturas.size  
    println("Altura promedio: $promedio")  
    var altos = 0  
    var bajos = 0  
    for(i in 0..alturas.size-1)  
        if (alturas[i] > promedio)  
            altos++  
        else  
            bajos++  
    println("Cantidad de personas más altas que el promedio: $altos")  
    println("Cantidad de personas más bajas que el promedio: $bajos")  
}
```

split()

- Crea un array a partir de una cadena de texto, donde en cada posición habrá una subcadena
- Recibe como entrada el carácter que utilizaremos por separador
- Por ejemplo para dada una cadena de texto crear un array con las palabras que la componen hacemos
- `"Hola alumno".split(" ")` crearía un array con dos elementos: `"Hola"` y `"alumno"`

Control de flujo

Condicionales - if

- Tiene la misma sintaxis que en Java y se utilizan los mismo operadores booleanos

```
if(5%2==0){  
    println("5 es par")  
}  
else{  
    println("5 es impar")  
}
```

```
val x = 5  
if(x>0){  
    println("x es un número positivo")  
}  
else if(x<0){  
    println("x es un número negativo")  
}  
else{  
    println("x es cero")  
}
```

Condicionales - if

- En Kotlin los bucle if devuelve un valor

```
val x = 5
val y = 3
val max = if(x>y){
    x
}
else{
    y
}
println(max)
```

Condicionales - when

- Sustituto de switch

var provincia = "Granada"

```
when(provincia){  
    "Granada" -> {  
        println("Bienvenido a Granada")  
    }  
    "Málaga" -> {  
        println("Bienvenido a Málaga")  
    }  
    else ->{  
        println("Bienvenido a la nada")  
    }  
}
```

Condicionales - when

- Podemos especificar múltiples condiciones para una rama

```
when(provincia){  
    "Granada", "Málaga", "Jaén", "Almería", "Sevilla", "Cádiz", "Córdoba", "Huelva" -> {  
        println("Bienvenido a Granada")  
    }  
    else ->{  
        println("No estás en Andalucía")  
    }  
}
```

Condicionales - when

- Al igual que if, when también devuelve un valor, por lo tanto:

```
val estoyEnAndalucia = when(provincia){  
    "Granada", "Málaga", "Jaén", "Almería", "Sevilla", "Cádiz", "Córdoba", "Huelva" -> {  
        true  
    }  
    else ->{  
        false  
    }  
}
```


Bucles - for

- Para recorrer un array (o una colección como veremos más adelante) utilizamos el bucle for .. in

```
val provincias =  
    arrayOf("Granada", "Málaga", "Jaén", "Almería", "Sevilla", "Cádiz", "Córdoba", "Huelva")  
println("Las provincias de Andalucía son:")  
for(provincia in provincias){  
    println(provincia)  
}
```

Bucles - for

- Para imitar el comportamiento clásico del bucle for →
for(int i=0; i<arr.size -1; i++) quedaría como:

```
val provincias =  
    arrayOf("Granada", "Málaga", "Jaén", "Almería", "Sevilla", "Cádiz", "Córdoba", "Huelva")  
println("Las provincias de Andalucía son:")  
for(i in provincias.indices){  
    val provincia = provincias[i]  
    println("La provincia número "+i+" es: "+provincia)  
}
```

```
fun main(parametro: Array<String>) {  
    for(i in 1..100)  
        println(i)  
}
```

```
fun main(parametro: Array<String>) {  
    var suma = 0  
    for(i in 1..10) {  
        print("Ingrese un valor:")  
        val valor = readln().toInt()  
        suma += valor  
    }  
    println("La suma de los valores ingresados es $suma")  
    val promedio = suma / 10  
    println("Su promedio es $promedio")  
}
```

Para pensar...

- ¿Por qué he declarado la variable provincia dentro del bucle con val en vez de con var?

Bucles – for (Rangos)

- Para crear un bucle que se repita n veces utilizamos el operador rango (..)

```
println("Voy a escribir 100 veces 'Hola Mundo'")  
for(i in 1..100){  
    println("Hola mundo")  
}
```

Más sobre rangos

- Podemos establecer el incremento del rango con 'step'

```
println("Los números pares entre 30 y 45 son: ")
for(i in 30..45 step 2){
    println(i)
}
```

- También podemos 'contar' hacia atrás con downTo

```
println("La palabra palíndromo alreves es: ")
val palabra = "Palíndromo"
for(i in "Palíndromo".length-1 downTo 0){
    print(palabra[i])
}
```

slice()

- El método slice() devuelve una parte del array o cadena al que se lo aplicamos, recibe como entrada el rango que queremos seleccionar
- Ejemplo “Hola alumno”.slice(2..5) → devolvería “la a”
- Se puede aplicar también a arrays

Bucles while

- Los bucles while y do .. While funciona de la misma forma en Kotlin que en Java

```
fun main(parametro: Array<String>) {  
    var x = 1  
    while (x <= 100) {  
        println(x)  
        x = x + 1  
    }  
}
```

```
fun main(parametro: Array<String>) {  
    do {  
        print("Ingrese un valor comprendido entre 0 y 999:")  
        val valor = readln().toInt()  
        if (valor < 10)  
            println("El valor ingresado tiene un dígito")  
        else  
            if (valor < 100)  
                println("El valor ingresado tiene dos dígitos")  
            else  
                println("El valor ingresado tiene tres dígitos")  
    } while (valor != 0)  
}
```


Saltos y rupturas

- Al igual que en Java utilizamos break y continue
- Break hace terminar el bucle
- Continue hace terminar la iteración actual pasando a la siguiente

Ejemplo break y continue

```
fun main() {  
    while (true) {  
        print("Escribe una palabra:")  
        val word = readLine()!!  
  
        if (word == "salir") break  
  
        println("Caracteres:${word.length}")  
    }  
}
```

```
fun main() {  
    for (i in 1..20) {  
        if (i % 4 != 0) {  
            continue  
        }  
        println(i)  
    }  
}
```

Funciones

Funciones

- A diferencia de Java, cuya aproximación a la programación dirigida a objetos es mucho más ortodoxa, Kotlin permite crear funciones.
- Una función es un secuencia de instrucciones que reciben unos parámetros de entrada y producen una salida
- Toda secuencia de instrucciones susceptible de ser usada varias veces debe de ser encapsulada en una función
- En Java no existen como tal (aunque un método estático de una clase a efectos prácticos es una función)
- No podemos decir que las funciones sean métodos, ya que a diferencia de estos el valor que devuelvan no dependen del estado de un objeto

Funciones

- Ejemplo de función:
- Fíjate como debemos definir el tipo de los parámetros de entrada y el de salida

```
fun square(x: Int): Int {  
    return x * x  
}  
  
fun main() {  
    print(square(2))  
}
```

Funciones que reciben un parámetro

Funciones que retornan un dato

Funciones de una única expresión

Funciones que tienen parámetros con valor por defecto

Llamadas a una función con argumentos nombrados

```

fun retornarMayor(v1: Int, v2: Int): Int {
    if (v1 > v2)
        return v1
    else
        return v2
}

fun main(parametro: Array<String>) {
    print("Ingrese el primer valor:")
    val valor1 = readln().toInt()
    print("Ingrese el segundo valor:")
    val valor2 = readln().toInt()
    println("El mayor entre $valor1 y $valor2 es ${retornarMayor(valor1, valor2)}")
}

```

```

fun mostrarMayor() {
    fun mayor (x1: Int, x2: Int) = if (x1 > x2) x1 else x2

    for(i in 1..5) {
        print("Ingrese primer valor:")
        val valor1 = readln().toInt()
        print("Ingrese segundo valor:")
        val valor2 = readln().toInt()
        println("El mayor entre $valor1 y $valor2 es ${mayor(valor1, valor2)}")
    }
}

fun main() {
    mostrarMayor()
}

```

```
fun retornarSuperficie(lado: Int) = lado * lado

fun main(parametro: Array<String>) {
    print("Ingrese el valor del lado del cuadrado:")
    val la = readln().toInt()
    println("La superficie del cuadrado es ${retornarSuperficie(la)}")
}
```

```
fun calcularSueldo(nombre: String, costoHora: Double, cantidadHoras: Int) {  
    val sueldo = costoHora * cantidadHoras  
    println("$nombre trabajó $cantidadHoras horas, se le paga por hora $costoHora por lo tanto le corresponde un sueldo de $sueldo")  
}  
  
fun main(parametro: Array<String>) {  
    calcularSueldo(nombre = "juan", costoHora = 10.5, cantidadHoras = 120)  
    calcularSueldo(costoHora = 12.0, cantidadHoras = 40, nombre = "ana")  
    calcularSueldo(cantidadHoras = 90, nombre = "luis", costoHora = 7.25)  
}
```


Funciones valores por defecto

- Las variables de entrada pueden tener un valor por defecto que se utilizará si no se especifica un valor para dicha variable:

```
fun longitudCircunfencia(radius: Double = 1.0): Double {  
    return 2 * Math.PI * radius  
}  
  
fun main() {  
    println(longitudCircunfencia())  
    println(longitudCircunfencia(radius: 2.0))  
}
```

Funciones valores por defecto

- Ejercicio, dada la siguiente función:

```
fun f(a:Int, b:Int=1, c:Int=2):Int{  
    return a*b+c  
}
```

- Determinar el resultado de:
f(1); f(1, 2, 3); f(1, c=3); f(2, 1)

Funciones valores por defecto

```
fun tituloSubrayado(titulo: String, caracter: String = "*") {  
    println(titulo)  
    for(i in 1..titulo.length)  
        print(caracter)  
    println()  
}  
  
fun main(parametro: Array<String>) {  
    tituloSubrayado(titulo: "Sistema de Administracion")  
    tituloSubrayado(titulo: "Ventas", caracter: "-")  
}
```

Excepciones

Excepciones

- ¿Qué hace la siguiente función?

```
fun main() {  
    println("5.3".toDoubleOrDefault( defaultValue: 1.0))  
    println("5.".toDoubleOrDefault( defaultValue: 1.0))  
    println(".3".toDoubleOrDefault( defaultValue: 1.0))  
    println("dos".toDoubleOrDefault( defaultValue: 1.0))  
}  
  
fun String.toDoubleOrDefault(defaultValue: Double): Double {  
    return try {  
        toDouble()  
    } catch (e: NumberFormatException) {  
        defaultValue  
    }  
}
```

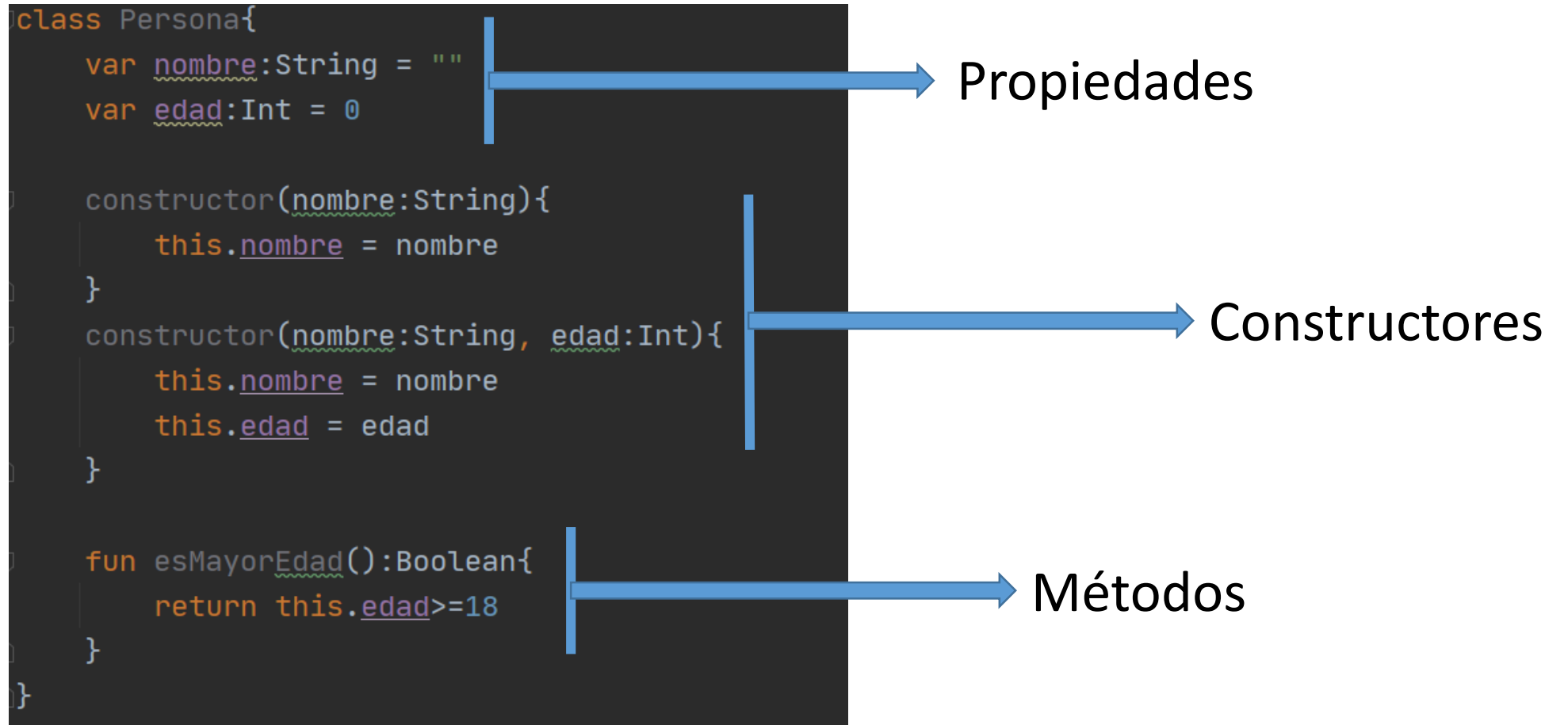
Excepciones

- Los tipos de excepciones son los mismos que en Java:
 - ArithmeticException
 - ArrayIndexOutOfBoundsException
 - ClassNotFoundException
 - FileNotFoundException
 - IOException
 - InterruptedException
 - NoSuchFieldException
 - NoSuchMethodException
 - NullPointerException
 - NumberFormatException
 - RuntimeException
 - StringIndexOutOfBoundsException

Excepciones

- Una excepción puede tener múltiples bloques catch
- Una excepción puede contener un último bloque 'finally' el cual se ejecutará haya o no haya excepción

Clases



Constructor Primario y Secundario

class ClaseEjemplo (val propiedad1:Tipo, var propiedad2:Tipo, ...)

```
class Weapon (val attack: Int, val speed: Double)
fun main() {
    val weapon1 = Weapon( attack: 3, speed: 0.5)
    println("Arma 1 (ataque:${weapon1.attack}, velocidad: ${weapon1.speed})")
}
```

```
fun main() {
    Product( x: 3, y: 7)
}

class Product {
    //Creating a secondary constructor
    constructor(x: Int, y:Int){
        var i = x * y
        println("The product of integers 3 and 7 is:  ${i} ")
    }
}
```

Instanciar clases

```
val p1 = Persona(nombre: "Borja", edad: 31)
```

- Fíjate que no utilizamos la palabra new

Clases visibilidad de las propiedades

- Por defecto propiedades y métodos son públicos, aunque puedes usar las palabras reservadas 'private', 'protected' e 'internal' si quieres cambiar este comportamiento
 - `private` : Marca una declaración como visible en la clase o archivo actual
 - `protected` : Marca una declaración como visible en la clase y subclases de la misma
 - `internal` : Marca una declaración como visible en el módulo actual
 - `public` : Marca una declaración como visible en todas partes

Si omites el modificador en una declaración, el valor por defecto asignado será `public` junto a `final` .

```
class Operaciones {  
    private var valor1: Int = 0  
    private var valor2: Int = 0  
  
    fun cargar() {  
        print("Ingrese primer valor:")  
        valor1 = readln().toInt()  
        print("Ingrese segundo valor:")  
        valor2 = readln().toInt()  
        sumar()  
        restar()  
    }  
  
    private fun sumar() {  
        val suma = valor1 + valor2  
        println("La suma de $valor1 y $valor2 es $suma")  
    }  
  
    private fun restar() {  
        val resta = valor1 - valor2  
        println("La resta de $valor1 y $valor2 es $resta")  
    }  
}  
  
fun main() {  
    val operaciones1 = Operaciones()  
    operaciones1.cargar()  
}
```

```
class Dado{
    private var valor: Int = 1
    fun tirar() {
        valor = ((Math.random() * 6) + 1).toInt()
    }

    fun imprimir() {
        separador()
        println("Valor del dado: $valor")
        separador()
    }

    private fun separador() = println("*****")
}

fun main(parametro: Array<String>) {
    val dado1 = Dado()
    dado1.tirar()
    dado1.imprimir()
}
```

Clases

- Al ser las propiedades públicas por defecto podemos acceder y cambiarlas sin necesidad de establecer setters y getters:

```
fun main(){  
    val p1 = Persona(nombre: "Borja", edad: 31)  
    p1.edad = 32  
    println(p1.nombre)  
}
```

Cuando definimos una propiedad pública podemos acceder a su contenido para modificarla o consultarla desde donde definimos un objeto.

A una propiedad podemos asociarle un método llamado set en el momento que se le asigne un valor y otro método llamado get cuando se accede al contenido de la propiedad.

Estos métodos son opcionales y nos permiten validar el dato a asignar a la propiedad o el valor de retorno.

Cuando no se implementan estos métodos el mismo compilador crea estos dos métodos por defecto

The following code in Kotlin

```
class Person {  
    var name: String = "defaultValue"  
}
```

is equivalent to

```
class Person {  
    var name: String = "defaultValue"  
  
    // getter  
    get() = field  
  
    // setter  
    set(value) {  
        field = value  
    }  
}
```

When you instantiate object of the `Person` class and initialize the `name` property, it is passed to the setters parameter `value` and sets `field` to `value`.

```
val p = Person()  
p.name = "jack"
```

Now, when you access `name` property of the object, you will get `field` because of the code `get() = field`.

```
println("${p.name}")
```


Data Class

- En ocasiones tenemos clases que lo único que hacen es albergar datos
- En dicho caso podemos crear las clases en una única línea y anteponiendo la palabra reservada 'data' a la palabra reservada 'class':

```
data class Musculo(val nombre:String, val zona:String, val descripcion: String, val ID:Int)
```

Enum Class

Utiliza para definir un conjunto de constantes

```
enum class TipoCarta{  
    DIAMANTE,  
    TREBOL,  
    CORAZON,  
    PICA  
}  
  
class Carta(val tipo: TipoCarta, val valor: Int) {  
  
    fun imprimir() {  
        println("Carta: $tipo y su valor es $valor")  
    }  
}  
  
fun main() {  
    val carta1 = Carta(TipoCarta.TREBOL, valor: 4)  
    carta1.imprimir()  
}
```

```

class Triangulo {
    var lado1: Int = 0
    var lado2: Int = 0
    var lado3: Int = 0

    fun inicializar() {
        print("Ingrese lado 1:")
        lado1 = readln().toInt()
        print("Ingrese lado 2:")
        lado2 = readln().toInt()
        print("Ingrese lado 3:")
        lado3 = readln().toInt()
    }
}

```

```

fun ladoMayor() {
    print("Lado mayor:")
    when {
        lado1 > lado2 && lado1 > lado3 -> println(lado1)
        lado2 > lado3 -> println(lado2)
        else -> println(lado3)
    }
}

fun esEquilatero() {
    if (lado1 == lado2 && lado1 == lado3)
        print("Es un triángulo equilátero")
    else
        print("No es un triángulo equilátero")
}

fun main(parametro: Array<String>) {
    val triangulo1 = Triangulo()
    triangulo1.inicializar()
    triangulo1.ladoMayor()
    triangulo1.esEquilatero()
}

```

Arrays de objetos

- Hasta ahora hemos hecho arrays de tipos básicos, pero podemos hacer arrays de objetos

```
val t1 = Triangulo(l1: 1.0, l2: 1.0, l3: 1.0)
val t2 = Triangulo(l1: 3.0, l2: 4.0, l3: 5.0)
val triangulos = arrayOf(t1, t2)
```

- Los objetos deben de ser de la misma clase

Arrays de objetos

- Para inicializar un array de objetos de una determinada clase que posteriormente rellenaremos:

```
val triangulos = arrayOfNulls<Triangulo>(size: 10)
val t1 = Triangulo(l1: 1.0, l2: 1.0, l3: 1.0)
val t2 = Triangulo(l1: 3.0, l2: 4.0, l3: 5.0)
triangulos[0] = t1
triangulos[1] = t2
```

```
class Persona(val nombre: String, val edad: Int) {  
    fun imprimir() {  
        println("Nombre: $nombre Edad: $edad")  
    }  
  
    fun esMayor() = if (edad >= 18) true else false  
}  
  
fun main(parametro: Array<String>) {  
    val personas: Array<Persona> = arrayOf(Persona( nombre: "ana", edad: 22), Persona( nombre: "juan", edad: 13), Persona( nombre: "carlos", edad: 6), Persona( nombre: "maria", edad: 72))  
    println("Listado de personas")  
    for(per in personas)  
        per.imprimir()  
    var cant = 0  
    for(per in personas)  
        if (per.esMayor())  
            cant++  
    println("Cantidad de personas mayores de edad: $cant")  
}
```