Platzi

Curso de Kotlin

Anahí Salgado
@anncode

¿Por qué debo aprender Kotlin?


JET
BRAINS

kotlinlang.org

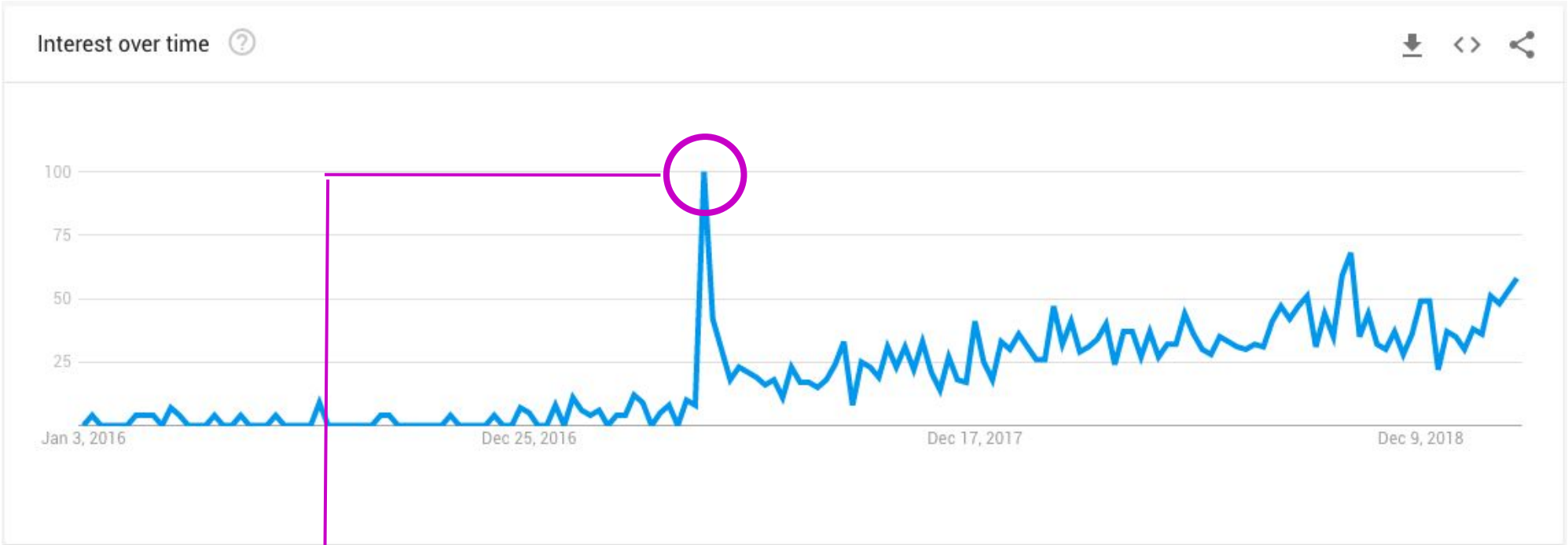
2010

2016

2017


Interest over time
100
75
50
25
Jan 3, 2016
Dec 25, 2016
Dec 17, 2017
Dec 9, 2018

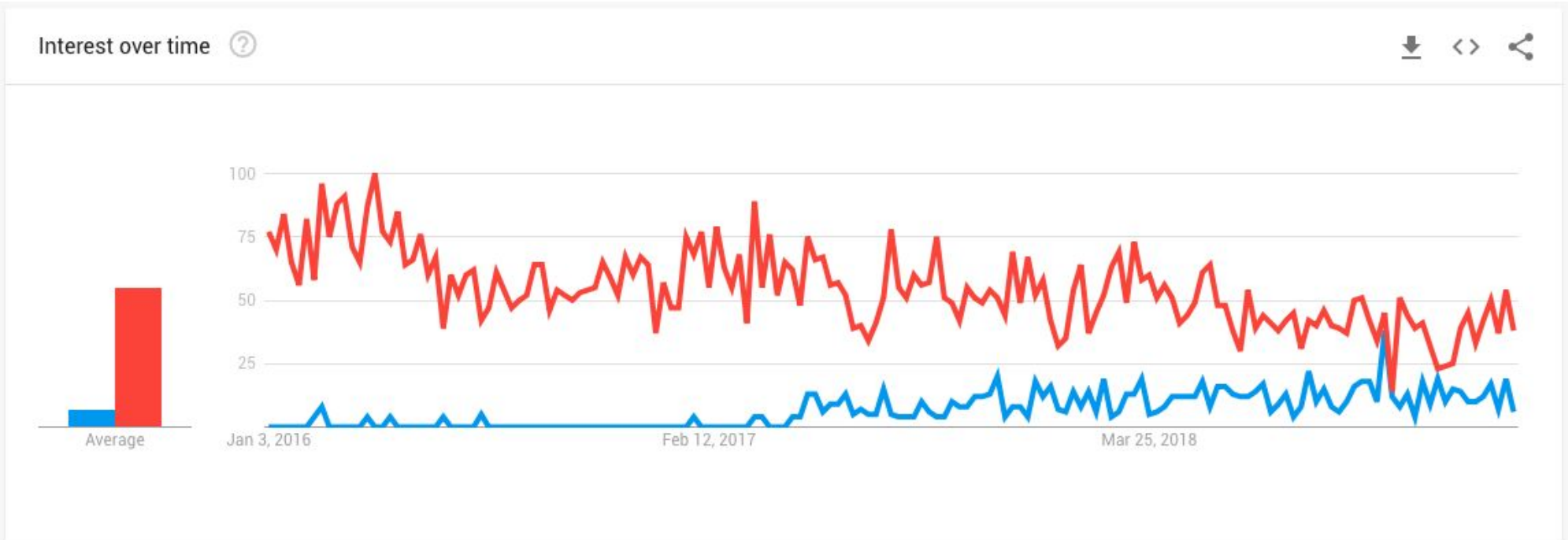
2017


Interest over time
100
75
50
25
Jan 3, 2016
Dec 25, 2016
Dec 17, 2017
Dec 9, 2018

2017

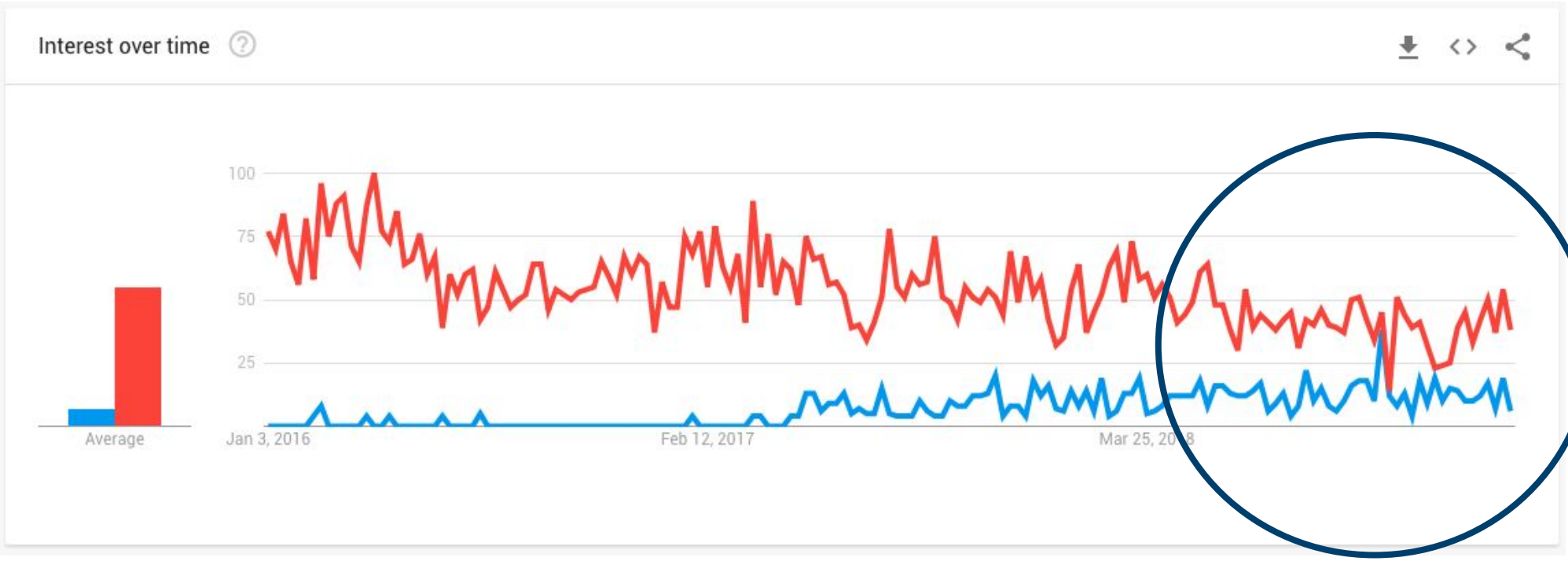
2019

Java


Interest over time
100
75
50
25
Average
Jan 3, 2016
Feb 12, 2017
Mar 25, 2018

Kotlin

Java



Kotlin

Conciso
Reduce código

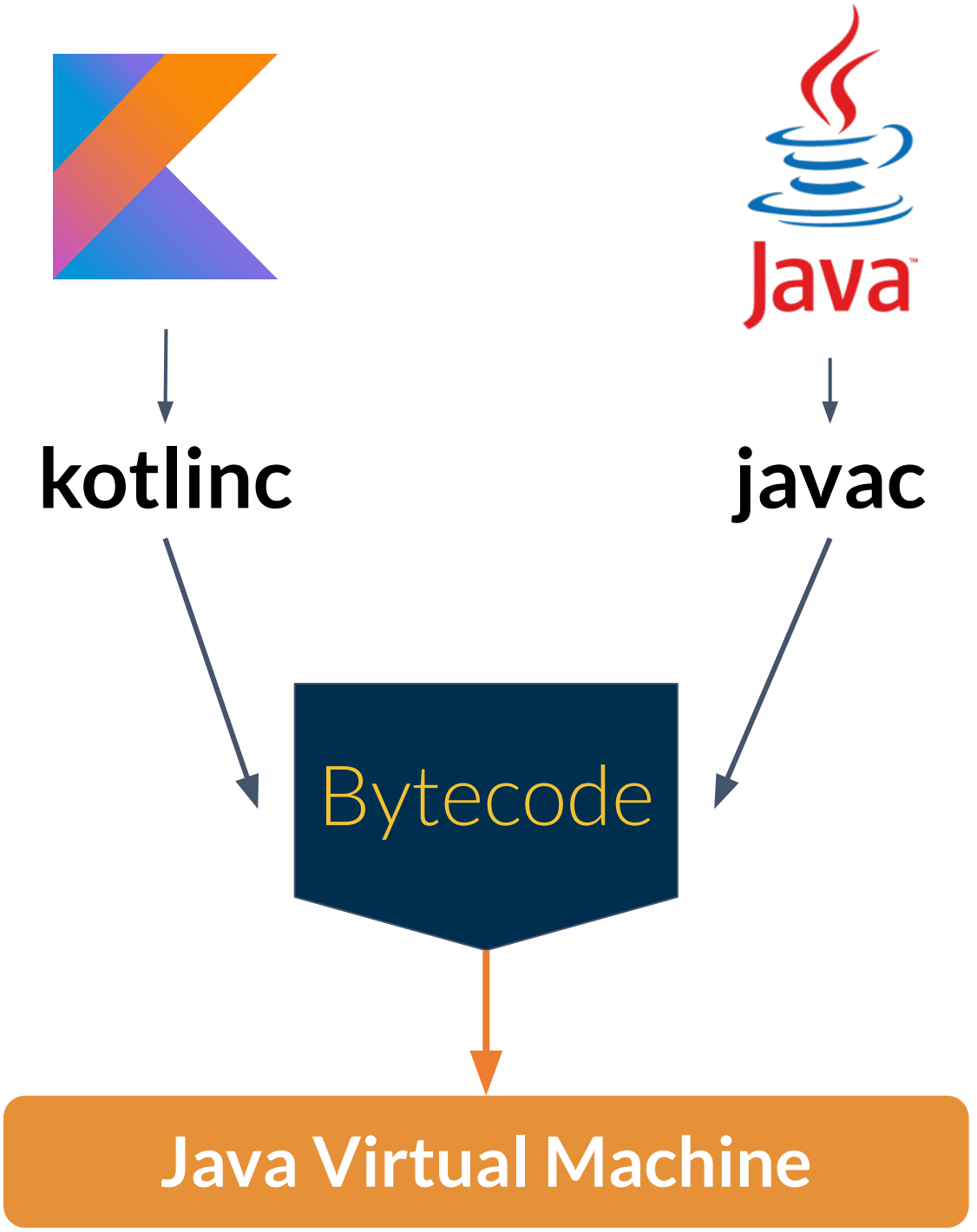
Conciso
Reduce código

Seguro
NullPointerException

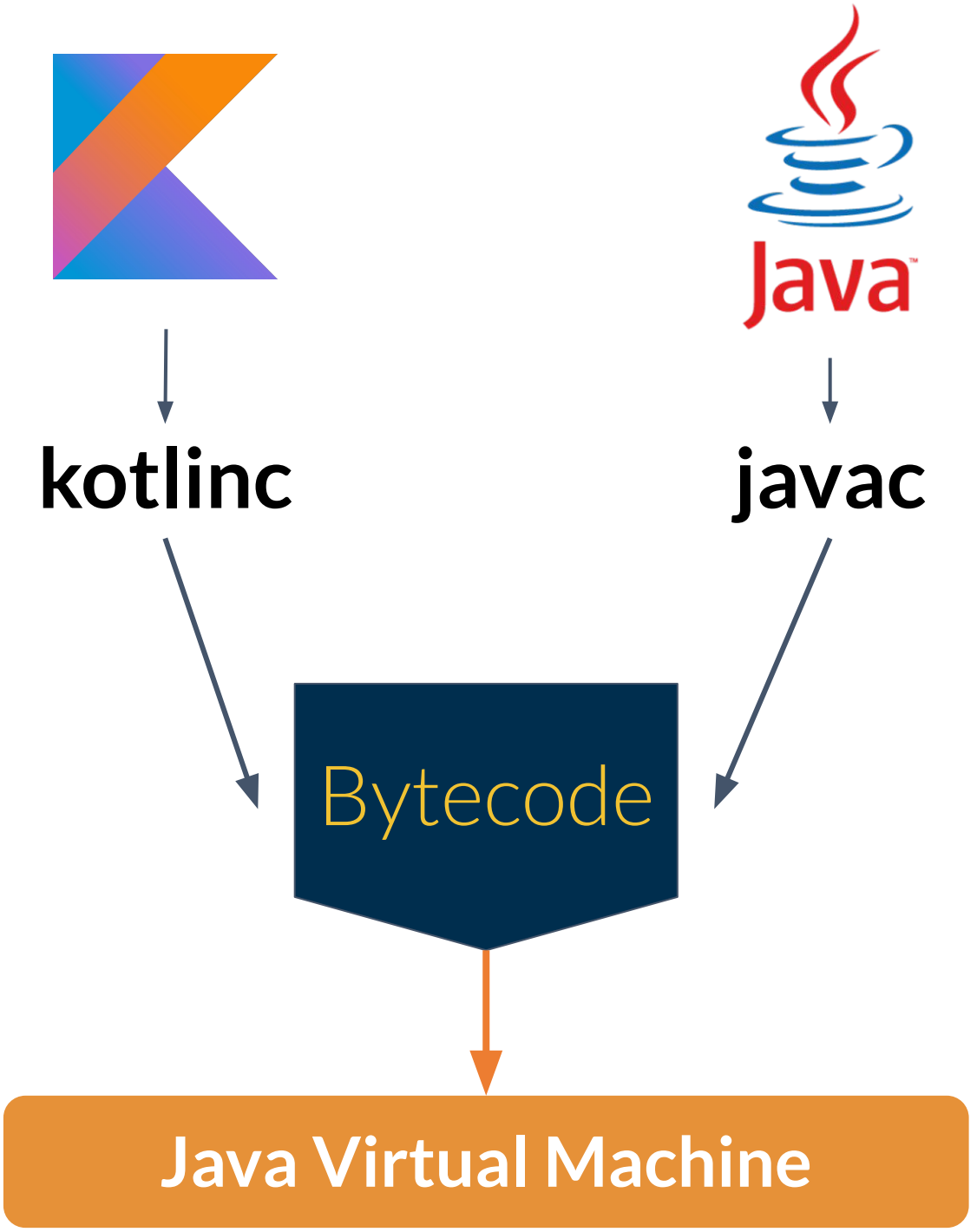
Conciso
Reduce código

Seguro
NullPointerException

Interoperable
Podemos usarlo con Java

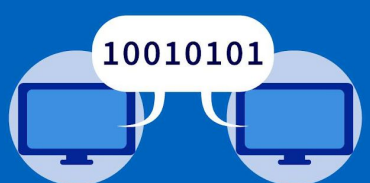

Java
kotlinc
javac
Bytecode
Java Virtual Machine

.jar



¿CÓMO FUNCIONA LA MÁQUINA VIRTUAL DE JAVA?

Existen lenguajes de alto y bajo nivel



Java es un lenguaje de alto nivel y para comunicarse con el hardware usa un código intermediario: el **Byte Code**

La **Máquina Virtual de Java** lee el Byte Code y lo traduce al lenguaje máquina del Sistema Operativo donde estás trabajando.

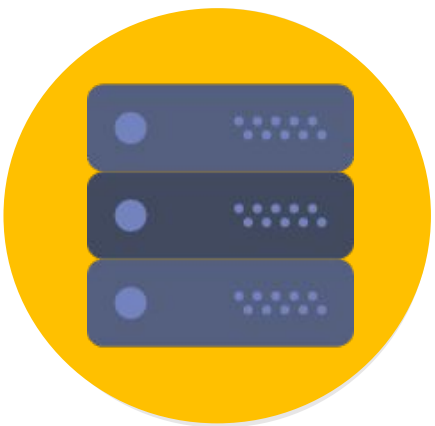
Conciso
Reduce código

Seguro
NullPointerException

Interoperable
Podemos usarlo con Java

Versátil
Diferentes tipos de aplicaciones

Kotlin



Kotlin Server Side
Ktor

Web
KotlinJS

Mobile
Android

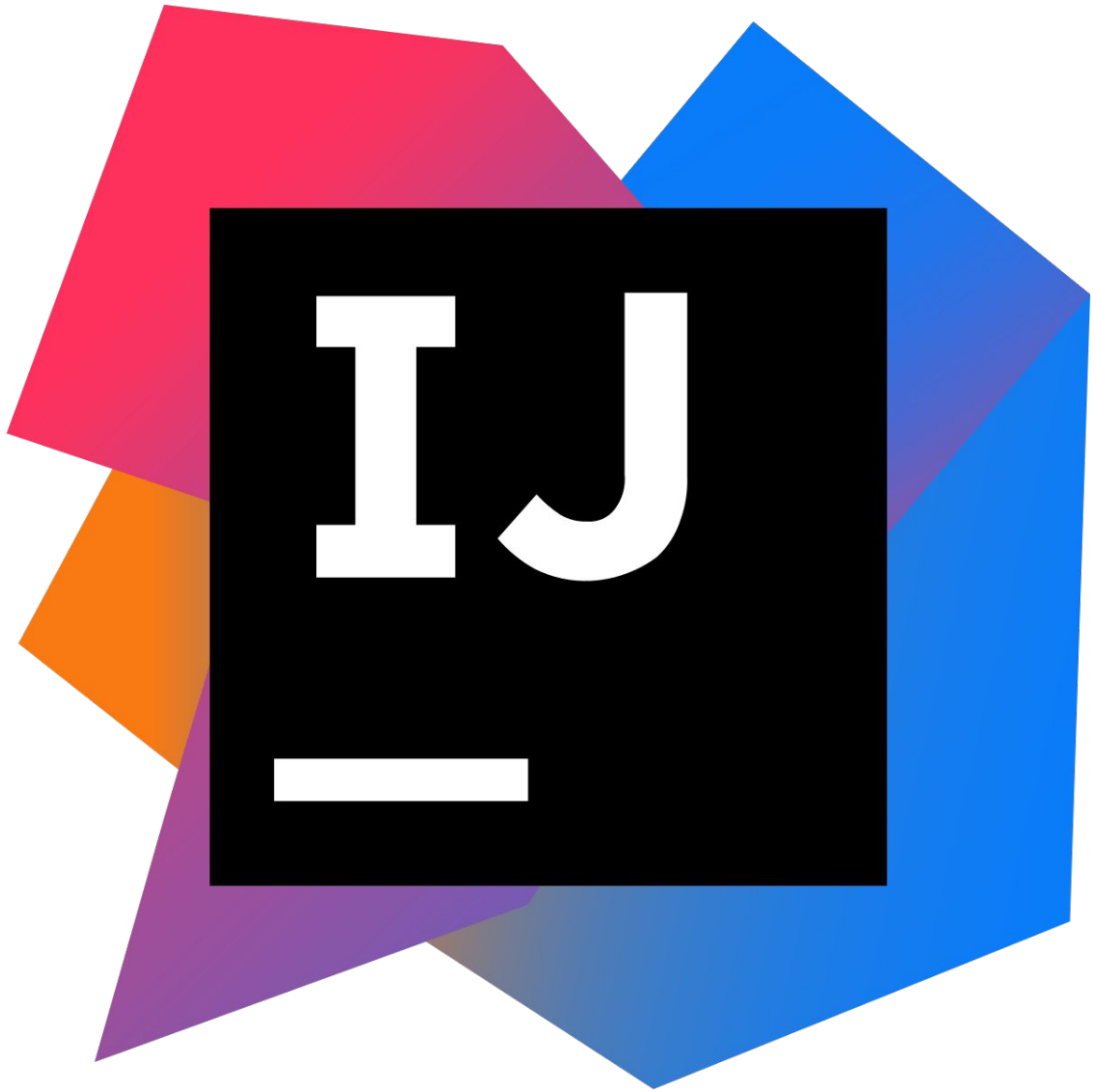
Platzi

Curso de Kotlin

Anahí Salgado
@anncode

Entorno de desarrollo

JDK de Java


IJ

.kt

HolaMundo.kt

Variables
vs.
Objetos

En Kotlin todo
es un
objeto

Tenemos tipos primitivos y objetos

No debemos declarar valores primitivos en Kotlin

Pero son definidos cuando no los usamos como objetos

```
var i = 10
i = i * 2
println(i)
```

En este modo Kotlin lo toma como un primitivo

Kotlin utiliza wrappers para los números esto se llama

boxing

Operadores

Expresión	Función	Se traduce a
a + b	plus	a.plus(b)
a - b	minus	a.minus(b)
a * b	times	a.times(b)
a / b	div	a.div(b)
a % b	mod	a.mod(b)

Expresión	Función	Se traduce a
a +=b	a = a + b	a.plusAssign(b)
a -= b	a = a - b	a.minusAssign(b)
a *= b	a = a * b	a.timesAssign(b)
a /= b	a = a / b	a.divAssign(b)
a %= b	a = a % b	a.modAssign(b)

Operador	Significado	Expresión	Se traduce a
+	Suma unaria	+a	a.unaryPlus()
-	Resta Unaria	-a	a.unaryMinus()
!	Negación	!a	a.not()
++	Incremento	++a	a.inc()
--	Decremento	--a	a.dec()

Operación	Significado	Expresión	Se traduce a
>	Mayor que	a > b	a.compareTo(b) > 0
<	Menor que	a < b	a.compareTo(b) < 0
>=	Mayor igual que	a >= b	a.compareTo(b) >= 0
<=	Menor igual que	a < = b	a.compareTo(b) <= 0
==	Es igual a	a == b	a?.equals(b) ?: (b === null)
!=	No es igual a	a != b	!(a?.equals(b) ?: (b === null))

Tipos de variables
var val const

Hay dos tipos de variables en Kotlin, changeables y unchangeables

var (changeables)
val (unchangeables)

const

El valor se determina en tiempo de compilación

val

El valor se puede determinar en tiempo de ejecución

```
const val foo = getDataAPI()

val fooVal = getDataAPI()

const val bar = "Hola Kotlin"
```

Strings

Valores Nulos

¿La nada existe?

¿Cómo puedes dividir un número de cosas entre nada?

Nada es un concepto ligado al lenguaje

En Kotlin, el pariente más cercano es nulo.

Una buena práctica de programación es comenzar con variables **no nulas.**

Kotlin es Null Safety

Kotlin evita que una excepción sea lanzada porque provoca vulnerabilidades

En Kotlin **por defecto ningún valor** puede ser **nulo**. Nos marcará un error.

En Kotlin **por defecto ningún valor** puede ser **nulo**. Nos marcará un error.

Is a safer language

```
if (loggedInUser != null) {
 if (userToSend != null) {
   if (validator != null) {
     …

    validator.canUserSendMessage(loggedInUser,
userToSend)

     ...
   } //end if (loggedInUser != null)
 } //end if (userToSend != null)
} //end if (validator != null)
```

```
val variable : Int?

variable?.metodo()
```

```
if (loggedInUser != null) {
 if (userToSend != null) {
   if (validator != null) {
     …

    validator.canUserSendMessage(loggedInUser,
userToSend)

     ...
   } //end if (loggedInUser != null)
 } //end if (userToSend != null)
} //end if (validator != null)
```

```
val filePath =
arguments?.getString(ARGUMENTS_PATH)
```

Double bang

!!

```
var msg: String?
msg = null
println(msg!!.length)
```

El operador **Double Bang** se utiliza pocas veces en Kotlin y es preferible **no usarlo**

!!

:(

Operador Elvis

?:

```
if (list != null) return list.size
else return 0
```

```
if (list != null) return list.size
else return 0


it.listFiles()?.size ?: 0
```

Arrays

Declaración

Dimensiones mirarlo como un json

Métodos útiles

Expresiones

En Kotlin **todo** es una **expresión**

En Kotlin **siempre** se devuelve un **valor**

Expresiones.

Se componen de variables y operadores que devuelven un valor

```
${} -> poner expresiones

$ -> poner valores
```

Algunas excepciones que no pueden ser una expresión son:

Bucles for y while

Funciones

Dos tipos de funciones

Funciones provistas por Kotlin

Funciones declaradas por nosotros

En Kotlin **siempre** se devuelve un **valor**

Unit es la forma de decir que no devuelve **nada**.

En las funciones podemos declarar valores por defecto a los parámetros, si estos no reciben un valor al usar la función, este toma el valor definido por defecto.

Algunas excepciones que no pueden ser una expresión son:

Tratar de asignar un bucle for o while a una variable

Kotlin one-line functions

Kotlinizar una función significa mejorarla

Las funciones one-line son expresiones

Filters

Es una librería funcional para listas en Kotlin

Lambdas

En otros lenguajes se conocen como Funciones anónimas, functions literals o funciones literales

Anónimo.
Declaramos una función que
no tiene nombre

Una lambda es una **expresión** que hace una función.

```
{println(“hola mundo”)}()

var hola = {println(“hola mundo”)}()

hola()
```

```
{println("Hola")}()

val w = {d: Int, c: Int -> d+c}

println(w(2,3))

println({d: Int, c: Int -> d+c}(4,4))
```

```
val random1 = random()

val random2 = {random()}
```

Paso por referencia ::variable

Programación Orientada a Objetos en Kotlin

Clases

Palabra reservada

class

Mínima declaración

class Shoe

Tiene Properties

var size
var color
var model

En Kotlin todos los valores son públicos por defecto

public todo acceso

private acceso solo dentro de la clase

protected acceso solo dentro de la clase y las clases que hereden

internal acceso entre módulos

- Modificadores de acceso
  public, private, protected, internal

```
private var size
```

Getters

Setters

1. Provistos por el programador

2. Acceso directo desde Kotlin

- **Getter y Setter** implícitos en la declaración de la propiedad

```
var size
var color
var model
```

- Añadiendo comportamiento a **Getter y Setter**

```
var model
    get(){}
    set(){}
```

Constructor

Constructor Primario

Forma clásica de inicializar una clase

Constructor Secundario

Cuando ponemos una inicialización

Cuando queremos **sobrecargar** un constructor usamos la palabra reservada constructor

```
constructor(n : Int): this() {}
```

init vs constructor

Al final es mejor usar un solo constructor e init

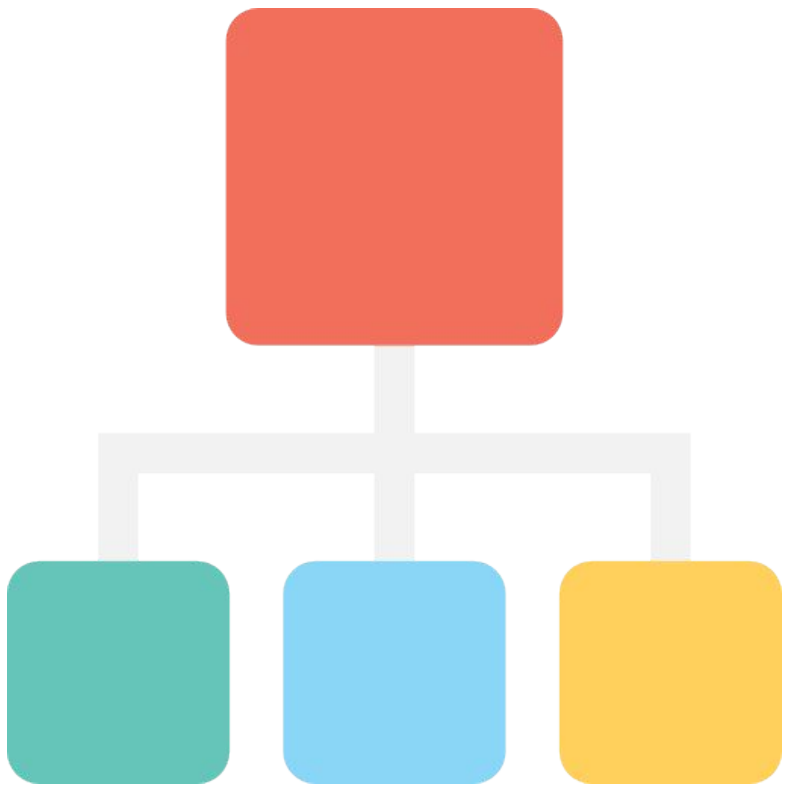
Herencia

Herencia

crearemos nuevas clases a partir de otras

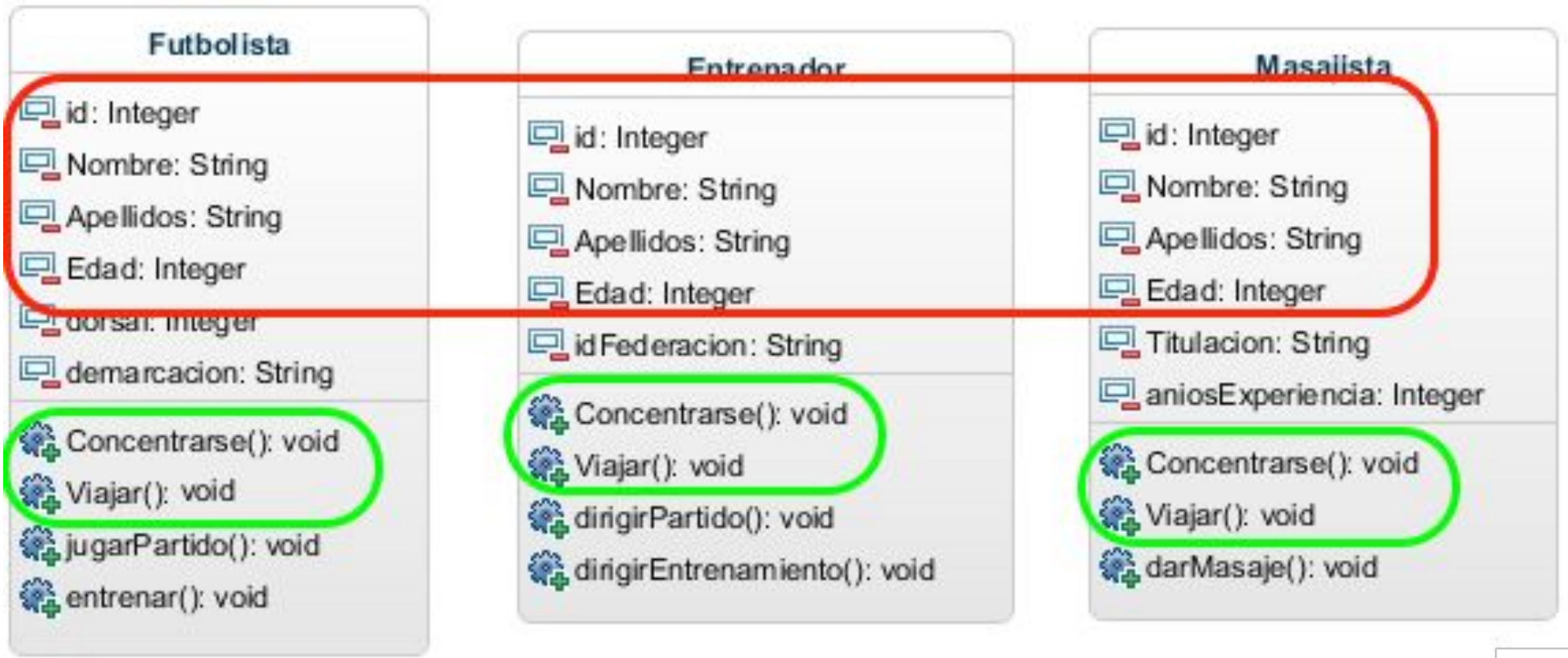
Herencia

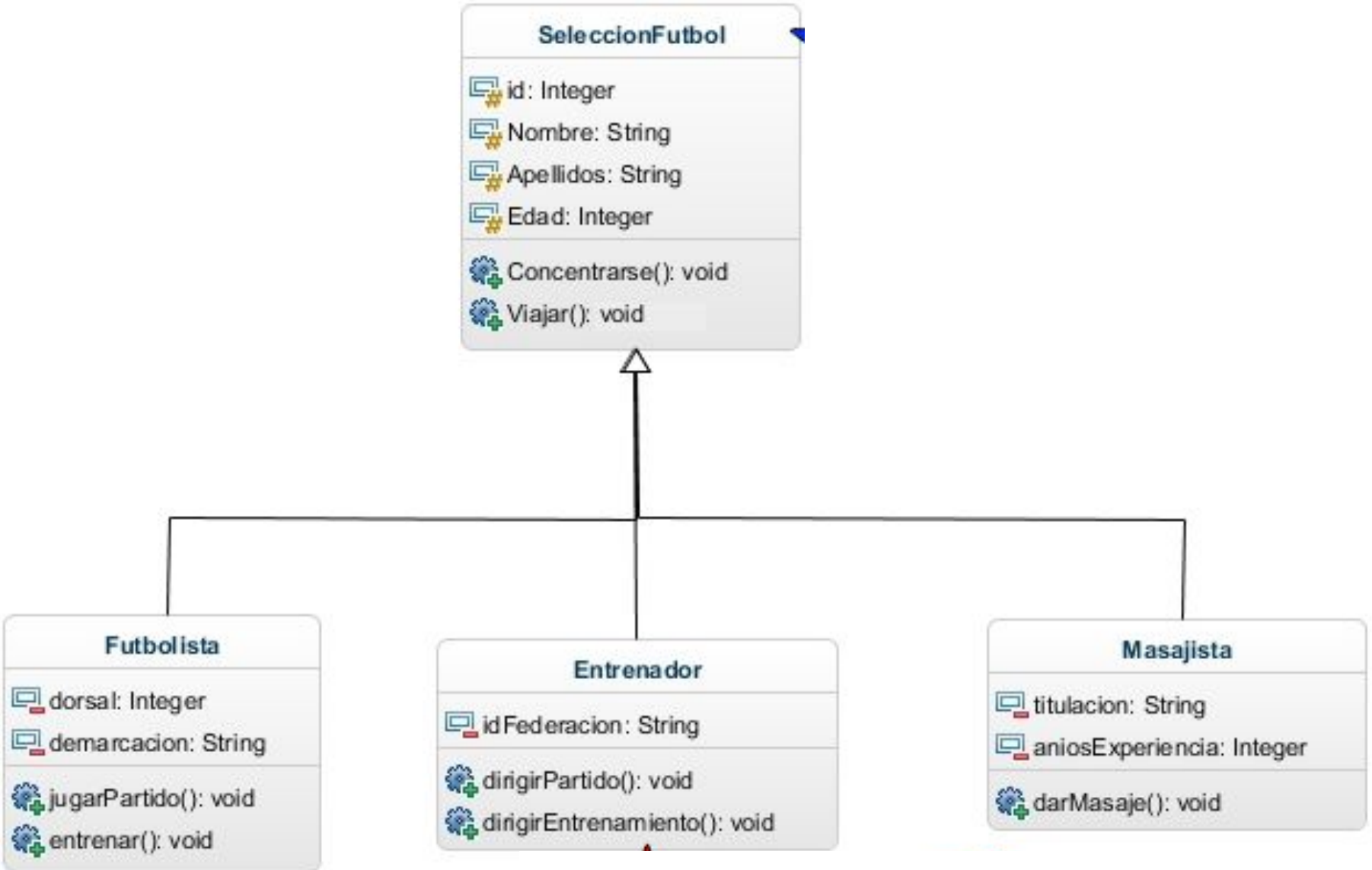
- Se establece una relación **padre e hijo**



Superclase

Subclase


Futbolista
id: Integer
Nombre: String
Apellidos: String
Edad: Integer
demarcacion: String
Concentrarse(): void
Viajar(): void
jugarPartido(): void
entrenar(): void
id: Integer
Nombre: String
Apellidos: String
Edad: Integer
idFederacion: String
Concentrarse(): void
Viajar(): void
dirigirPartido(): void
dirigirEntrenamiento(): void
Masajista
id: Integer
Nombre: String
Apellidos: String
Edad: Integer
Titulacion: String
aniosExperiencia: Integer
Concentrarse(): void
Viajar(): void
darMasaje(): void


SeleccionFutbol
id: Integer
Nombre: String
Apellidos: String
Edad: Integer
Concentrarse(): void
Viajar(): void
Futbolista
dorsal: Integer
demarcacion: String
jugarPartido(): void
entrenar(): void
Entrenador
idFederacion: String
dirigirPartido(): void
dirigirEntrenamiento(): void
Masajista
titulacion: String
aniosExperiencia: Integer
darMasaje(): void

Any

Todas las clases

No se puede heredar de una clase en Kotlin por **defecto**, están **cerradas**

```
open class Product
```

Clases Abstractas

- Son clases que **no tienen implementación**

  abstract

- **abstract**

```
abstract class Product
```

- En la **Herencia** se **deben implementar todos** los métodos sólo abstractos.

- **No** se pueden **crear instancias** de una clase Abstracta

Interfaces

Acciones

Funcionalidades

Métodos

\+ Interfaces

Atributos

Propiedades

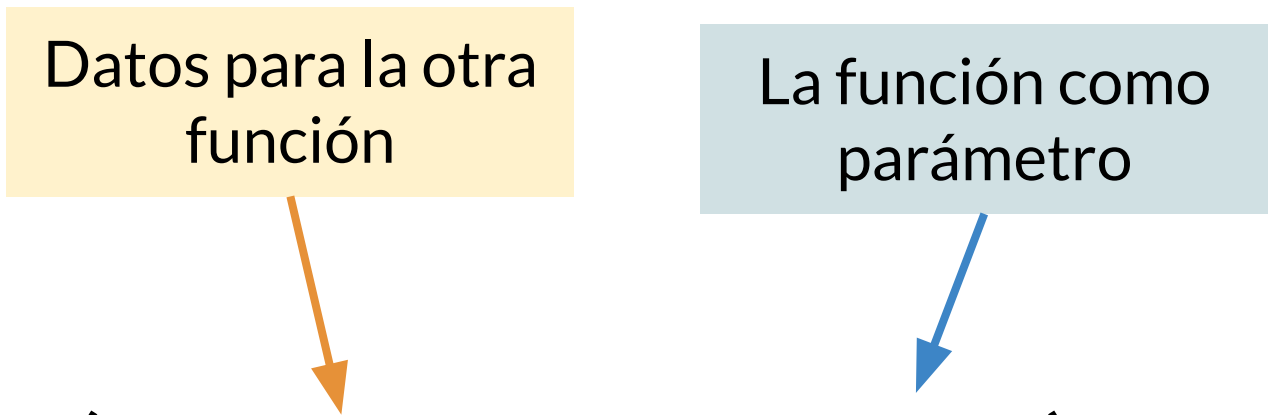
+ Clases abstractas

Descomposición

```
val (name) = d2
println(name)
```

Funciones de Orden superior

Poner una **función** como **parámetro** de otra




```
fun miFuncion( variables, funcion): ret {
    return funcion(variables)
}
```

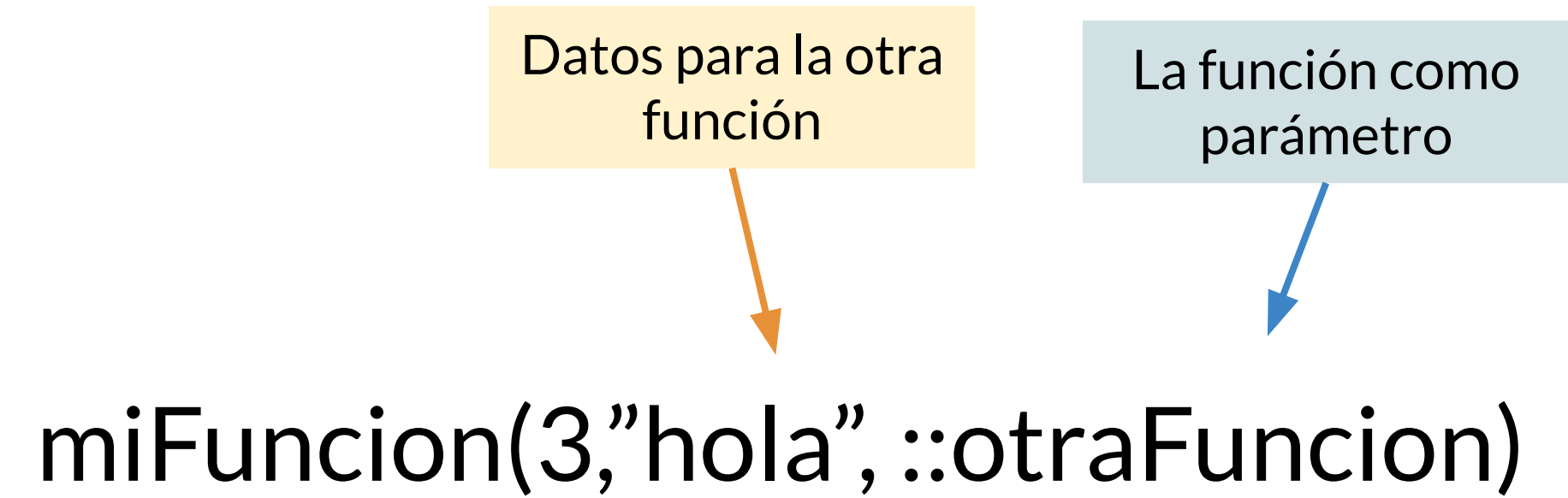
Utilizaremos **lambdas** para definir la **función**

Lambda

{... -> ... }

Lambda como parámetro

```
fun miFuncion( variables, {... -> ... }): ret{
    return funcion(variables)
}
```


Datos para la otra función
La función como parámetro
miFuncion(3,”hola”, ::otraFuncion)

```
view.setOnClickListener ({ view ->
    //do stuff
})


fun someRequest(callback: (Data, Int) -> Unit)
{

}
```

Recursividad

Recursividad

Es otro tipo de iteraciones más avanzadas.

filter, map, reduce

Recursividad

Iteración clásica

listas

Iteración Recursiva

listas, árboles, grafos

Lambdas
Inlines

Las lambdas son objetos

Una expresión lambda es una instancia de una función de una interface

Sin inline un objeto es creado cada vez que llamamos a la función

Esto no es bueno en memoria

Por eso ponemos la palabra reservada inline antes de la función