KOTLIN BASICS

Yeratzy Espinoza

Kotlin

Un lenguaje de programación moderno que ayuda a los desarrolladores a ser más productivos.

Expresivo y conciso

Código más seguro

Interoperable

Simultaneidad estructurada

VARIABLES

Inferencia de tipos

- El compilador puede inferir el tipo.
- Se puede declarar el tipo explícitamente si es necesario.

Variables mutables e inmutables

- La inmutabilidad no es forzada, pero se recomienda.

Kotlin es un lenguaje de tipado estático. El tipo se resuelve en tiempo de compilación y nunca cambia.

Mutable and immutable variables

- Mutable

```
var contador = 10
```

- Inmutable

```
val nombre = “Carmen”
```

DATA TYPES

Types

Integer

Long
Int
Short
Byte

Non-Integer

Double
Float
Char
Boolean

Type casting

```
val a: Int = 9
val b: Byte = a
println(b)
```

⇒ error: type mismatch: inferred type is Int but Byte was expected

```
val a: Int = 9
println(a.toByte())
```

⇒ 9

Numeric operator methods

Kotlin mantiene los números como primitivos, pero le permite llamar a métodos sobre números como si fueran objetos.

```
2.times(3)
⇒ Kotlin.Int = 6
```

```
2.4.div(2)
⇒ Kotlin.Double = 1.2
```

```
3.5.plus(4)
⇒ Kotlin.Double = 7.5
```

Strings

$variable –> Se llama **interpolación variable**

Concatenación

```
val numFrutas = 4
val numVerduras = 3

println("Tengo $numFrutas frutas" + "y $numVerduras verduras")
```

⇒ Tengo 4 frutas y 3 verduras

Strings templates

La expresión template comienza con un signo de pesos ($) y puede ser un valor simple:

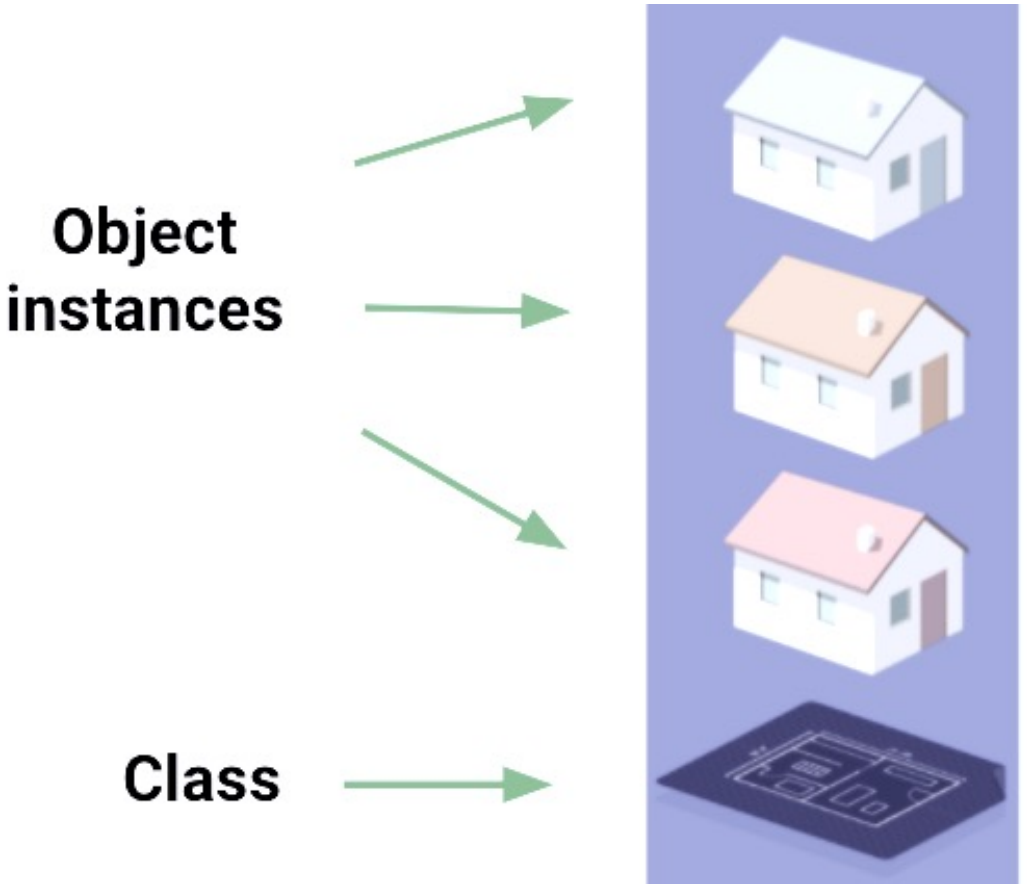
```
val i = 10
println("i = $i")
⇒ i = 10
```

O como una expresión dentro de llaves:

```
val x = "hola"
println("$x.length es ${x.length}")
⇒ hola.length es 4
```

CLASSES

- Las clases son planos de objetos
- Las clases definen métodos que operan en sus instancias de objetos



Define and use a class

```
class House {
  val color: String = "white"
  val numberOfWindows: Int = 2
  val isForSale: Boolean = false

  fun updateColor(newColor: String){...}
  ...
}
```

Object Instance

```
val myHouse = House()
println(myHouse)
```

Constructors

Cuando se define un constructor en el encabezado de la clase, puede contener:

- Sin parámetros

  class A

- Parámetros
  - No marcado con var o val → la copia existe solo dentro del alcance del constructor

    class B(x: Int)

  - Marcados con var o val → la copia existe en todas las instancias de la clase
    - class B(val y: Int)

Default parameters

Las instancias de clase pueden tener valores predeterminados.

- Utilice valores predeterminados para reducir la cantidad de constructores necesarios.
- Los parámetros predeterminados se pueden mezclar con los parámetros requeridos.
- Más conciso (no es necesario tener varias versiones de constructor)

```
class Box(val length: Int, val width:Int = 20, val height:Int = 40)

val box1 = Box(100, 20, 40)
val box2 = Box(length = 100)
val box3 = Box(length = 100, width = 20, height = 40)
```

Primary constructor

Declare el constructor principal dentro del encabezado de la clase.

```
class Circle(i: Int) {
   init {
        ...
   }
}
```

Es equivalente a:

```
class Circle {
    constructor(i: Int) {
        ...
    }
}
```

Initializer block

- Cualquier código de inicialización requerido se ejecuta en un bloque de inicio especial.
- Se permiten varios bloques de inicio.
- Los bloques de inicio se convierten en el cuerpo del constructor principal.

Initializer block example

```
class Cuadrado(val lado: Int) {
    init {
        println(lado * 2)
    }
}

val c = Cuadrado(10)

=> 20
```

DATA CLASS

- Clase especial que existe solo para almacenar un conjunto de datos.
- Marque la clase con la palabra clave **data**.
- Genera getters para cada propiedad (también setters para vars)
- Genera métodos toString (), equals (), hashCode (), copy () y operadores de desestructuración
- Formato: data class <NameOfClass> (parameterList)

Data class example

Definir data class:

```
data class Jugador(val nombre: String, val puntos: Int)
```

Usar el data class:

```
val primerJugador = Jugador(”Manuel", 10)
println(primerJugador)
```

=> Player(name=Manuel, score=10)

COMPANION OBJECTS

Object/singleton

- A veces solo quieres que exista una instancia de una clase.
- Se utiliza la palabra clave **object** en lugar de la palabra clave **clase.**
- Se accede con NameOfObject. <Función o variable>

Companion objects

- Permite que todas las instancias de una clase compartan una única instancia de un conjunto de variables o funciones.
- Usar palabra clave **companion**.
- Referenciado a través de ClassName.PropertyOrFunction

```
class sistemaFisica {
    companion object Constantes {
        val gravedad = 9.8
        val unidad = "metric"
        fun calcularFuerza(masa: Double, aceleracion: Double) : Double {
            return masa * aceleracion
        }
    }
}

println(sistemaFisica. Constantes.gravedad)
println(sistemaFisica. Constantes. calcularFuerza(10.0, 10.0))
⇒9.8
⇒100.0
```

ENUM CLASS

Tipo de datos definido por el usuario para un conjunto de valores con nombre.

Se usa para requerir que las instancias sean de uno o varios valores constantes.

El valor constante es, por defecto, no visible para nosotros.

Use **enum** antes de la palabra clave **class.**

Definir un enum con colores rojo, verde y azul.

```
enum class Color(val r: Int, val v: Int, val a: Int) {
   ROJO(255, 0, 0),
   VERDE(0, 255, 0),
   AZUL(0, 0, 255)
}

println("" + Color. ROJO.r + " " + Color. ROJO.v + " " + Color. ROJO.a)
=> 255 0 0
```

FUNCIONES

Unit returning functions

Se declaran usando la palabra clave **fun**

Si una función no devuelve ningún valor útil, su tipo de retorno es **Unit**.

```
fun imprimirHola(nombre: String): Unit {
    println("Hola $nombre!")
}
```

La declaración del tipo de retorno Unit es opcional.

Function arguments

Las funciones pueden tener:

- Parámetros por default
- Parámetros requeridos
- Argumentos nombrados

Default parameters

Los valores predeterminados proporcionan un respaldo si no se pasa ningún valor de parámetro.

```
fun conducir(velocidad: String = ”rapido”) {
   println(”conduciendo $ velocidad”)
}
```

conducir() ⇒ conduciendo rapido
conducir(”lento") ⇒ conduciendo lento
conducir(velocidad=”como tortuga") ⇒ conduciendo como tortuga

Required parameters

Si no se especifica ningún valor predeterminado para un parámetro, se requiere el argumento correspondiente.

```
fun tempHoy(dia: String, temp: Int) {
    println(”Hoy es $dia y estamos a $temp grados.")
}
```

Compact functions

Compact functions o single-expression functions, hacen que el código sea más conciso y legible.

```
fun doblar(x: Int): Int {
    x * 2
}
```

Versión completa

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

Versión compacta

CONDITIONALS

Control Flow

Kotlin presenta varias formas de implementar la lógica condicional:

- Declaraciones If / Else
- Declaraciones When
- For loops
- While loops

Declaraciones if/else

```
val numVasos = 10
val numPlatos = 8

If (numVasos > numPlatos) {
        println(“Demasiados vasos”)
} else {
         println(“Sin vasos suficientes.”)
}
```

⇒ Demasiados vasos

Rangos

Tipo de datos que contiene un intervalo de valores comparables.

Ejemplo: Números enteros del 1 al 100

Los rangos están delimitados.

Los objetos dentro de un rango pueden ser mutables o inmutables

Ranges in if/else statements

```
val numEstudiantes = 50

if (numEstudiantes in 1..100) {
    println(numEstudiantes)
}
```

⇒ 50

When

```
when (resultado) {
    0  -> println(”Sin resultados")
    in 1..39 -> println(”Se obtienen resultados”)
    else -> println(”Son muchos resultados”)
}
```

for loops

```
val mascotas = arrayOf( ”gato", ”perro", ”pajaro")

for (element in mascotas) {
    print(element + " ")
}
```

⇒ gato perro pajaro

No es necesario definir una variable iteradora e incrementarla para cada iteración.

for loops: elements and indexes

```
for ((index, element) in mascotas.withIndex()) {
    println("Mascota en la posición $index es $element")
}
```

⇒ Mascota en la posición 0 es gato
Mascota en la posición 1 es perro
Mascota en la posición 2 es pajaro

for loops: step sizes and ranges

```
for (i in 1..5) print(i)
```

⇒ 12345

```
for (i in 3..6 step 2) print(i)
```

⇒ 35

```
for (i in 5 downTo 1) print(i)
```

⇒ 54321

```
for (i in 'd'..'g') print(i)
```

⇒ defg

Returns and jumps

Kotlin tiene tres expresiones de salto estructural:

- **return** por defecto retornos de la función adjunta más cercana o función anónima
- **break** termina el enclosing loop más cercano.
- **continue** pasa al siguiente paso del bucle enclosing loop más cercano.

while loops

```
var nFrutas = 0

while (nFrutas < 50) {
    nFrutas ++
}
println("$nFrutas frutas ")
```

⇒ 50 frutas

```
do {
    nFrutas--
} while (nFrutas > 50)
println("$nFrutas frutas ")
```

⇒ 49 frutas

repeat loops

```
repeat(2) {
    print(”¡Hola!")
}
```

⇒ ¡Hola! ¡Hola!

LISTS

- Las listas son colecciones ordenadas de elementos.
- Se puede acceder a los elementos de la lista mediante sus índices.
- Los elementos se pueden repetir más de una vez en una lista

Immutable list using listOf()

Declarar una lista usando listOf()

```
val instrumentos = listOf(”guitarra", "piano", "violin")
println(instrumentos)
```

⇒ [guitarra, piano, violin]

Las listas creadas con listOf() no pueden ser cambiadas, son inmutables.

Mutable list using mutableListOf()

Las listas se pueden cambiar usando mutableListOf ()

```
val instrumentos = mutableListOf(”guitarra", "piano", "violin")
 instrumentos.remove("violin")
```

Extension functions

Añadir funciones a una clase existente que no se puede modificar directamente.

- No modifica realmente la clase existente.
- No se puede acceder a las variables de instancia privadas.

Formato: fun ClassName.functionName(params) { body }

Why use extension functions?

Agregue funcionalidad a las clases que no están abiertas.

Agrega funcionalidad a las clases que no te pertenecen.

Separe la API central de los métodos auxiliares para las clases que te pertenecen.

Agregar esImpar() en la clase **Int** :

```
fun Int. esImpar(): Boolean { return this % 2 == 1 }
```

Llamar esImpar() en un entero:

```
3. esImpar()
```

List filters

Forma parte de una lista basada en alguna condición.

Si una función literal tiene solo un parámetro, puede omitir su declaración y el "->".

El parámetro se declara implícitamente con el nombre it.

```
val nums = listOf(1, 2, 3)
nums.filter { it > 0 }
```

Filters itera a través de una colección, donde es el valor del elemento durante la iteración. Esto es equivalente a:

```
nums.filter { n: Int -> n > 0 }
```

ó

```
nums.filter { n -> n > 0 }
```

Si la expresión devuelve verdadero, se incluye el elemento.

Map

Realiza la misma transformación en cada elemento y devuelve la lista.

```
val numeros = listOf(1, 2, 3)
println(numeros.map { it * 3 })
=> [3, 6, 9]
```

Lists setOf

Son colecciones de datos que no permiten tener valores repetidos.

```
val set = setOf(1, 1, 2, 3, 4, 5, 5, 1)
println(set)
=> [1, 2, 3, 4, 5]
```

ARRAYS

- Los arrays almacenan varios elementos.
- Se puede acceder a los elementos de la lista mediante sus índices.
- Los elementos son mutables.
- El tamaño de los arrays es fijo.

Array using arrayOf()

- Se crea usando arrayOf ()

```
val mascotas = arrayOf( ”gato", ”perro", ”pajaro")
println(java.util.Arrays.toString(mascotas))
```

⇒ [gato, perro, pajaro]

Arrays with mixed or single types

Un array puede contener diferentes tipos de datos.

```
val mezcla = arrayOf(”nueve", 8)
```

También puede contener un solo tipo de dato.

```
val numeros = intArrayOf(1, 2, 3)
```

Combining arrays

Usar el operador **+**.

```
val numeros1 = intArrayOf(1, 2, 3)
val numeros2 = intArrayOf(4, 5, 6)
val combinacion = numeros2 + numeros1
println(Arrays.toString(combinacion))
```

⇒ [4, 5, 6, 1, 2, 3]

NULL SAFETY

Null safety

- Las variables no pueden ser nulas por defecto.
- Se puede asignar explícitamente una variable nula utilizando el operador de llamada segura.
- Permite excepciones null-pointer utilizando el operador `!!`
- Se pueden probar los nulos utilizando el elvis(`?:`) operator

Las variables no pueden ser nulas

Las variables nulas no están permitidas por defecto.

- Declara una variable Int y asignale null:

```
var numberOfSomething: Int = null
```

⇒ error: null can not be a value of a non-null type Int

Safe call operator

El operador de llamada segura (?), después del tipo indica que la variable puede ser nula.

Declara un Int? as nullable

```
var numberOfSomething : Int? = null
```

En general, no establezca una variable en nula, ya que puede tener consecuencias no deseadas.

Testing for null

- Comprobar si la variable numberOfSomething no es nula. Luego decrementa esa variable.

```
var numberOfSomething = 6
  if (numberOfSomething != null) {
       numberOfSomething = numberOfSomething.dec()
  }
```

- En Kotlin se puede escribir usando el operador de llamada segura.

```
var numberOfSomething = 6
   numberOfSomething = numberOfSomething?.dec()
```

The !! operator

- Si está seguro de que una variable no será nula, use !! para forzar la variable a un tipo no nulo.
- Entonces puedes llamar a métodos/propiedades en él.

```
val len = s!!.length
        => Lanza NullPointerException si es nulo
```

Advertencia: !! arrojará una excepción, solo debe usarse cuando sea excepcional mantener un valor nulo.

Elvis operator

- Realiza pruebas nulas con el operador ?:

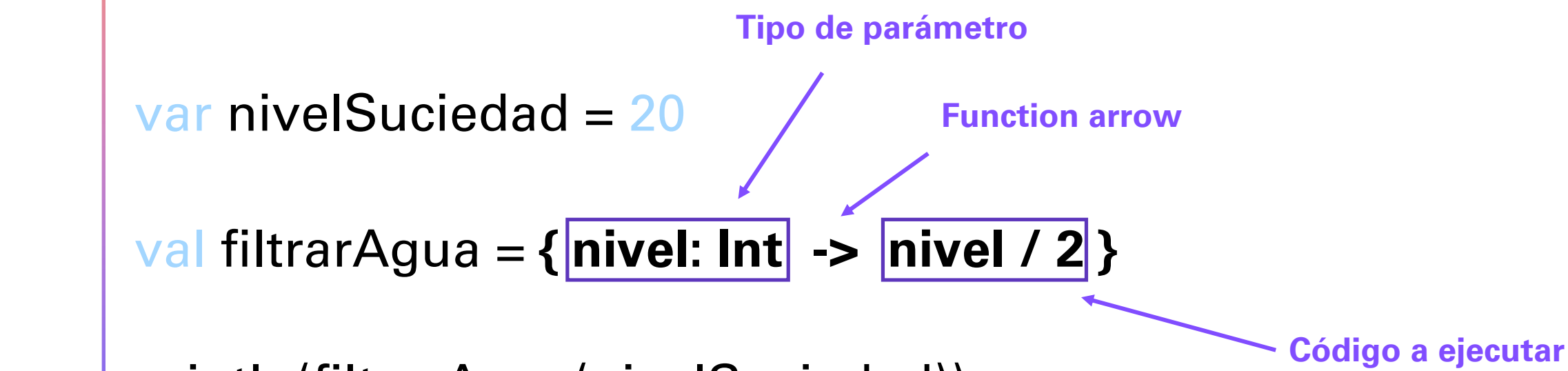
```
numberOfSomething = numberOfSomething?.dec() ?: 0
```

LAMBDAS AND HIGHER-ORDER FUNCTIONS

- Las funciones de Kotlin se pueden almacenar en variables y estructuras de datos.
- Pueden pasarse como argumentos y devolverse desde otras funciones de orden superior.
- Puede utilizar funciones de orden superior para crear nuevas funciones "integradas"

Lambda functions

Una lambda es una expresión que crea una función que no tiene nombre.



Higher-order functions

Toman funciones como parámetros o devuelven una función.

```
fun encodeMsg(msg: String, encode: (String) -> String): String {
    return encode(msg)
}
```

El cuerpo del código llama a la función que se pasó como segundo argumento y le pasa el primer argumento.

Last parameter call syntax

Kotlin prefiere que cualquier parámetro que tome una función sea el último parámetro.

```
encodeMessage("acronym", { input -> input.toUpperCase() })
```

Se puede pasar una lambda como parámetro de función sin ponerla entre paréntesis

```
encodeMsg("acronym”) { input -> input.toUpperCase() }
```