v1.1.2

Elementos básicos

Kotlin

Sentencias, bloques y comentarios

Sentencias

- En Kotlin, las sentencias se escriben una en cada línea
- No es necesario incluir un `;` al final

Sentencias y ;

```
val rocket = "🚀"; println(rocket)
```

Bloques

- Agrupan instrucciones
- Definen el ámbito de las variables
- En Kotlin se utilizan las llaves { y } para delimitarlos

Comentarios

- De una línea, `//`
- De múltiples líneas, `/* */`
- Se pueden anidar, si están balanceados `/* /* */ */`

Variables y constantes

Declaración

```
// Asignar una vez
val a: Int = 1  // immediate assignment
val b = 2       // `Int` type is inferred
val c: Int      // Type required when no initializer is provided
c = 3           // deferred assignment

// Mutables
var x = 5 // `Int` type is inferred
x += 1
```

Recomendación

- Por defecto, declarar las variables con `val`
- Cambiarlo a `var` cuando sea necesario modificar el valor
- Ayuda a evitar errores

Inferencia de tipos

```
val b = 2
```

Anotaciones de tipo

```
val c: Int

c = 3
```

Salida por consola

```
println("¡Hola mundo!")

val hora = 9
print("Son las ${hora} en punto")

println(", ¡a cenar!")
```

Valores numéricos y lógicos

Tipos de datos numéricos

Tipo	Bits	Descripción
`Int`	32	Valor numérico entero
`Long`	64	Valor numérico entero "largo"
`Float`	32	Valor numérico de precisión simple
`Double`	64	Valor numérico de precisión doble
`Short`	16	Valor numérico "corto"
`Byte`	8	Byte

Literales numéricos

	Literal	Notación	Ejemplo
Enteros	Decimal		`1_000_000`
	Long	`L`	`123L`
	Binario	`0b`	`0b1011_1100`
	Hexadecimal	`0x`	`0xF3A`
Coma flotante	Double	`e`	`1.25e4`
	Float	`f o F`	`123.5f`

Conversiones de tipo

```
val b: Byte = 1 // OK, literals are checked statically
val i: Int = b // ERROR

val i: Int = b.toInt() // OK: explicitly widened
```

https://kotlinlang.org/docs/reference/basic-types.html#explicit-conversions

Conversiones de tipo

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

- Todos los tipos disponen de los métodos de conversión

Valores lógicos

```
val orangesAreOrange = true

val turnipsAreDelicious: Boolean = false
```

Cadenas de texto y caracteres

String

```
// String
val s = "Hello, world!\n"

// Raw string
val text = """
    for (c in "foo")
        print(c)
"""
```

Características de los String

- Son inmutables
- Se pueden recorrer los caracteres individuales con un bucle `for`
- Se pueden comparar directamente con el operador `==`

Plantillas de Strings

```
// Nombre simple
val i = 10
val s = "i = $i" // evaluates to "i = 10"

// Cualquier expresión
val s = "abc"
val str = "$s.length is ${s.length}" // evaluates to "abc.length is 3"
```

Caracteres

```
var letra: Char = 'A'
```

Operadores: asignación y aritméticos

Operador de asignación

- Copia el contenido de la parte derecha en la parque izquierda
- No devuelve valor
- Hay versiones compuestas, como +=

Operadores aritméticos

Operador	Operación
`+`	Suma
`-`	Resta
`*`	Multiplicación
`/`	División
`%`	Resto de la división
`-i`	Menos unario (cambio de signo)
`+i`	Más unario (no afecta al valor)

Entrada y salida de datos

Salida por consola

```
println("¡Hola mundo!")

val hora = 9
print("Son las ${hora} en punto")

println(", ¡a cenar!")
```

Lectura desde teclado

```java
// Java
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

String texto = br.readLine();
```

```kotlin
// Kotlin
val br = BufferedReader(InputStreamReader(System.`in`))

val texto = br.readLine()
```

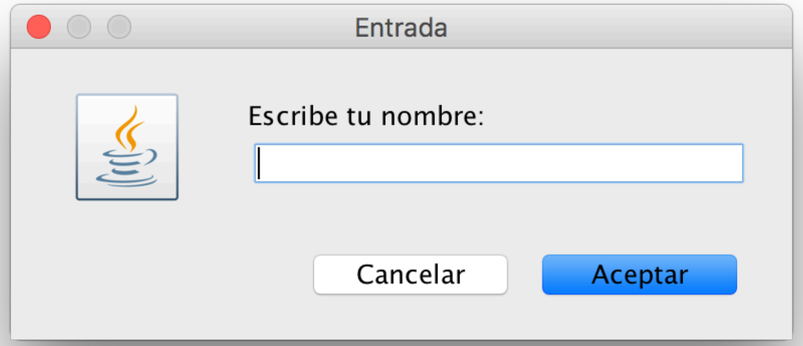
Salida con cuadros de diálogo

```
JOptionPane.showMessageDialog(null, "¡Hola kotlin!")
```

https://docs.oracle.com/javase/tutorial/uiswing/components/dialog.html

Entrada con cuadros de diálogo

```
val texto = JOptionPane.showInputDialog("Escribe tu nombre:")
```



https://docs.oracle.com/javase/tutorial/uiswing/components/dialog.html

Tipos de datos anulables

Tipos de datos anulables

- En Kotlin, el sistema de tipos distingue entre referencias que pueden almacenar `null` y las que no
- Este mecanismo hace explícito el hecho de que una variable puede ser nula
- Permite al compilador detectar el uso de referencias nulas
- Se declaran poniendo `?` detrás del nombre del tipo

Variables anulables

```
var a: String = "abc"
a = null // compilation error

var b: String? = "abc"
b = null // ok

val l = a.length // ok

val l = b.length // error: variable 'b' can be null
```

Acceso a variables anulables

Modo	Sintaxis	Notas
Comprobación de `null`	`if (b != null)`	Sólo se puede usar con variables inmutables
Operador de llamada segura	`val l: Int? = b?.length`	Encadenable `bob?.department?.head?.name`
Operador Elvis	`val l = b?.length ?: -1`	Si es `null`, valor por defecto
Operador !!	`val l = b!!.length` ⚠️	Puede producir una excepción NPE en tiempo de ejecución

https://en.wikipedia.org/wiki/Elvis_operator

Conversión de tipos

Consulta de tipo

```
if (obj is String) {
    print(obj.length)
}

if (obj !is String) { // same as !(obj is String)
    print("Not a String")
}
else {
    print(obj.length)
}
```

Conversiones

	Sintaxis	Notas
~~Implícitas~~		No se hacen conversiones automáticas
Explícitas	`.toInt()`	Disponibles para los tipos predeterminados
Smart cast	`if(x is String) print(x)`	Si el compilador lo puede deducir, no hace falta la explícita
Unsafe cast	`as`	Excepción si no se puede convertir

Conversión explícita (casting)

```
val π = 3.14159265359  // Double

val pi:Int = π  // error

val pi:Int = π.toInt()  // ok
```

Smart cast

```
fun demo(x: Any) {
    if (x is String) {
        print(x.length) // x is automatically cast to String
    }
}
```

Operador de cast

```
val x: String = y as String
    // "unsafe" -> si "y" es null, excepción

val x: String? = y as String?
    // admite null -> no hay excepción

val x: String? = y as? String
    // "safe" -> si "y" es null, no excepción
```