



# Lección 1: Fundamentos de Kotlin



# Introducción

## Lo básico de Kotlin

- [Cómo empezar un proyecto.](#)
- [Uso de operadores.](#)
- [Tipos de datos.](#)
- [Variables.](#)
- [Condicionales y control de flujo.](#)
- [Listas y arrays.](#)
- [Seguridad frente a valores nulos \(null safety\).](#)

# Empezando con Kotlin

<https://kotlinlang.org/docs/basic-syntax.html>

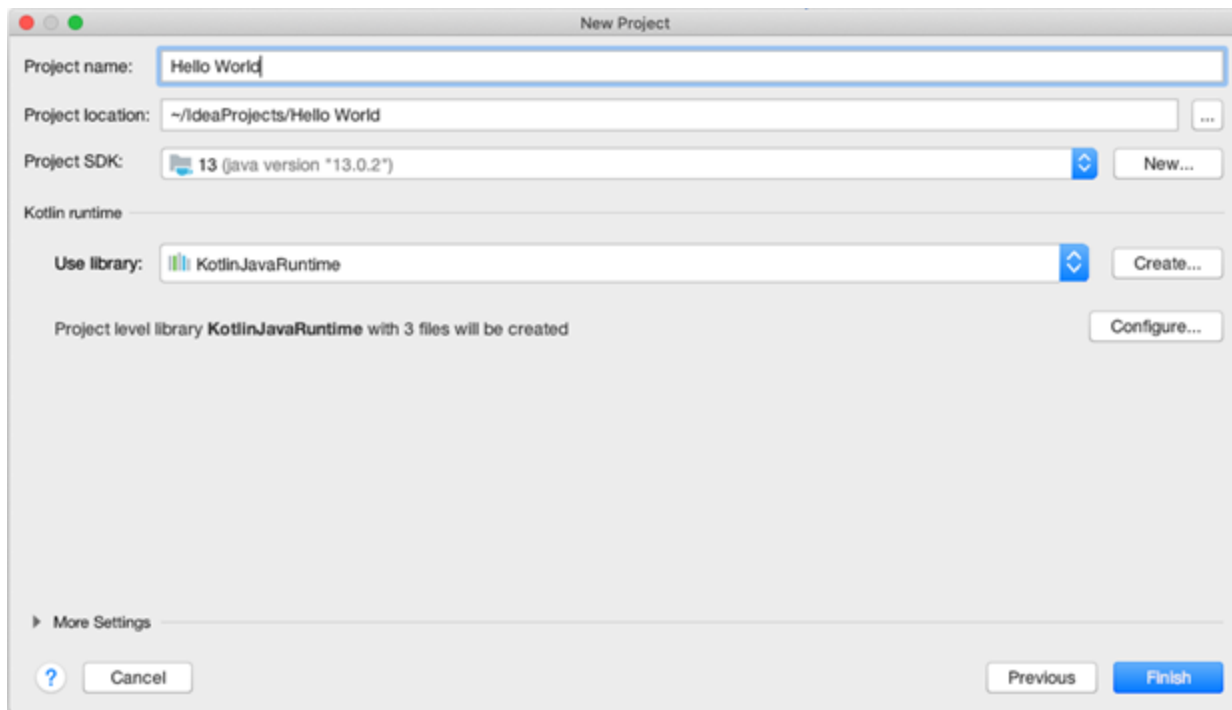
# Abre IntelliJ IDEA



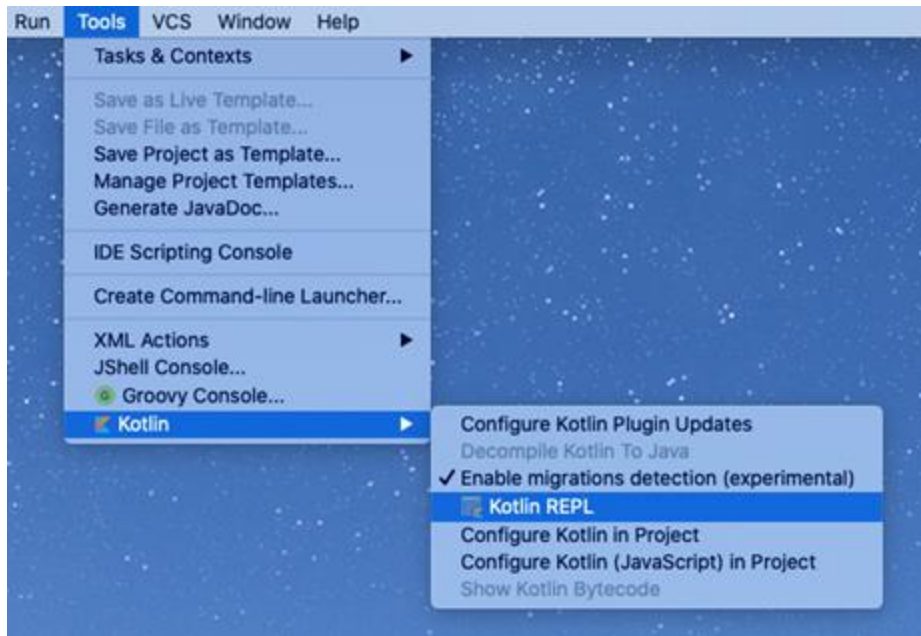
# Crea un nuevo proyecto



# Ponle un nombre al proyecto



# Abre la REPL (*Read-Eval-Print-Loop*), donde puedes escribir y ejecutar código línea por línea



# Crea la función printHello()

```
Run: Kotlin REPL (in module HelloKotlin) x
Welcome to Kotlin version 1.3.41 (JRE 11.0.2+9-LTS)
Type :help for help, :quit for quit

fun printHello() {
    println("Hello World")
}

printHello()
Hello World

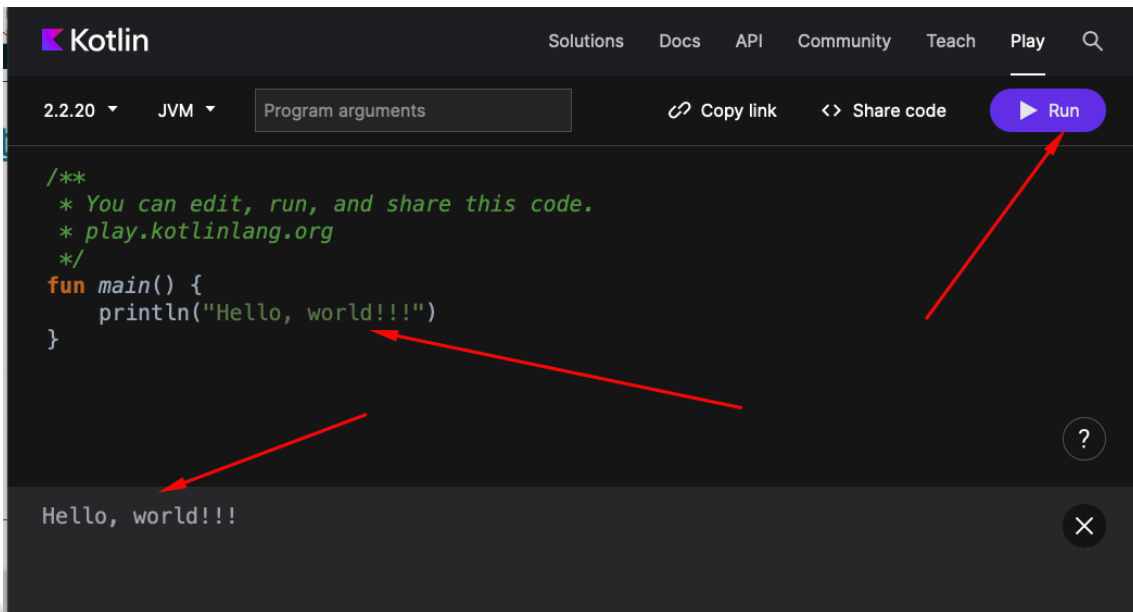
|<=>> to execute
```

Press **Control+Enter**  
(**Command+Enter** on  
a Mac) to execute.



# O bien en play.kotlin.org

<https://play.kotlinlang.org/>



# Operadores

# Operadores

Los operadores numéricos, al igual que en otros lenguajes, Kotlin utiliza `+`, `-`, `*`, `/` y `%` para suma, resta, multiplicación, división y módulo (o resto). También existen los operadores de incremento y decremento, los operadores de comparación, el operador de asignación y los operadores de igualdad.

# Operators

- Operadores matemáticos
- Incremento y decremento
- Comparación
- Operadores de asignación
- Operadores de igualdad

+ - \* / %

++ --

< <= > >=

=

== !=

# Operadores matemáticos con enteros

1 + 1      =>      2

53 - 3      =>      50

50 / 10      =>      5

9 % 3      =>      0

```
fun main() {  
    //Operadores matemáticos con enteros  
    println(1 + 1)    // 2  
    println(53 - 3)   // 50  
    println(50 / 10)  // 5  
    println(9 % 3)    // 0  
}
```

# Operadores matemáticos con doubles

1.0 / 2.0 => 0.5

2.0 \* 3.5 => 7.0

```
//Ejemplo para doubles  
println(1.0 / 2.0) // 0.5  
println(2.0 * 3.5) // 7.0
```

- Kotlin admite diferentes tipos numéricos, como `Int`, `Long`, `Double` y `Float`. Observa que todos comienzan con mayúscula.
- Aunque en la representación interna no sean objetos, sí lo son en el sentido de que podemos invocar funciones y propiedades sobre ellos, y Kotlin representa los objetos usando mayúsculas iniciales

# Resultados al jugar con enteros y doubles

1+1

⇒ `kotlin.Int` = 2

1.0/2.0

⇒ `kotlin.Double` = 0.5

53-3

⇒ `kotlin.Int` = 50

2.0\*3.5

⇒ `kotlin.Double` = 7.0

50/10

⇒ `kotlin.Int` = 5



# Métodos de operadores numéricos

En Kotlin los números son primitivos, pero se comportan como objeto:

`2.times(3)`

⇒ `kotlin.Int = 6`

`3.5.plus(4)`

⇒ `kotlin.Double = 7.5`

`2.4.div(2)`

⇒ `kotlin.Double = 1.2`

```
//Métodos de operadores numéricos
println(2.4.div(other = 2))    // 1.2
println(3.5.plus(other = 4))  // 7.5
println(2.times(other = 3))   // 6
```

# Tipos de datos

<https://kotlinlang.org/docs/basic-types.html>

# Tipo de dato entero

Tipo	Bits	Notas
Long	64	From $-2^{63}$ to $2^{63}-1$
Int	32	From $-2^{31}$ to $2^{31}-1$
Short	16	From -32768 to 32767
Byte	8	From -128 to 127

# Tipos de coma flotante y otros tipos numéricos

Tipo	Bits	Notas
Double	64	16 - 17 Cifras significativas
Float	32	6 - 7 Cifras significativas
Char	16	16-bit Character Unicode
Boolean	8	Verdadero o falso. Las operaciones incluyen: <ul style="list-style-type: none"><li>• <code>  </code> → OR lógico</li><li>• <code>&amp;&amp;</code> → AND lógico</li><li>• <code>!</code> → negación</li></ul>

# Tipos de operandos

Los resultados de las operaciones mantienen los tipos de los operandos

`6*50`

`⇒ kotlin.Int = 300`

`1/2`

`⇒ kotlin.Int = 0`

`6.0*50.0`

`⇒ kotlin.Double = 300.0`

`1.0*2.0`

`⇒ kotlin.Double = 0.5`

`6.0*50`

`⇒ kotlin.Double = 300.0`

```
//Tipos de operandos
```

```
println(6 * 50)      // 300    -> Int, porque ambos son Int
println(6.0 * 50.0)  // 300.0  -> Double, porque ambos son Double
println(6.0 * 50)    // 300.0  -> Double, porque uno es Double
println(1 / 2)       // 0      -> Int, división entera
println(1.0 / 2.0)   // 0.5    -> Double
```

# Tipo de casting


- Kotlin **no convierte implícitamente entre tipos numéricos**, por lo que no puedes asignar directamente un valor de tipo Short a una variable de tipo Long, ni un Byte a un Int.
- La conversión implícita de números es una fuente común de errores en los programas, pero puedes evitarlo **asignando valores de distintos tipos mediante casting**.
- Aquí creamos una variable y mostramos primero qué ocurre si intentas reasignarla con un tipo incompatible. Luego usamos **toByte()** para convertirla y así imprimirla sin errores

# Tipo de casting

Asigna un Int a un Byte

```
val i: Int = 6
val b: Byte = i
println(b)
```

```
//Casting
val i: Int = 6
val b: Byte = i
```



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

Convierte un Int a un Byte con un casting

```
val i: Int = 6
println(i.toByteArray())
```

⇒ 6

```
//Casting
val i: Int = 6
val b: Byte = i.toByteArray()
```

# Guiones bajos para números largos

Usa guiones bajos para hacer que las constantes numéricas largas sean más legibles

```
val unMillon = 1_000_000
```

```
val idNumero = 999_99_9999L
```

```
val valorHexadecimal = 0xFF_EC_DE_5E
```

```
val numeroBinario = 0b11010010_01101001_10010100_10010010
```

```
//Guiones bajos para números largos
val unMillon = 1_000_000_000 // 1000000000
val idNumero = 1234_5678_9012_3456L
val valorHex = 0xAB_CD_EF_12
val valorBinario = 0b1010_1100_1111_0001

println(unMillon) // 1000000000
println(idNumero) // 1234567890123456
println(valorHex) // 2882400018
println(valorBinario) // 44273
```



# Strings

Las cadenas (Strings) son cualquier secuencia de caracteres encerrada entre comillas dobles

```
val s1 = "Hola Mundo"
```

Los literales de cadena pueden contener caracteres de escape

```
val s2 = " Hola Mundo!\n"
```

O cualquier texto arbitrario delimitado por comillas triples (""")

```
val text = """  
    var bicicletas = 50  
    """
```

# Caracteres de escape

Secuencia	Significado	Ejemplo en Kotlin	Resultado
<code>\n</code>	Nueva línea	<code>"Hola\nMundo"</code>	Hola Mundo
<code>\t</code>	Tabulación	<code>"Uno\tDos"</code>	Uno Dos
<code>\"</code>	Comillas dobles	<code>"Ella dijo: \"Hola\""</code>	Ella dijo: "Hola"
<code>\'</code>	Comilla simple	<code>"Es un \'char\'"</code>	Es un 'char'
<code>\\</code>	Barra invertida (\)	<code>"Ruta: C:\\Users\\"</code>	Ruta: C:\Users\
<code>\r</code>	Retorno de carro (carriage return)	<code>"Hola\rMundo"</code>	Mundo
<code>\b</code>	Retroceso (backspace)	<code>"AB\bC"</code>	AC
<code>\uXXXX</code>	Carácter Unicode (hex)	<code>"\u03A9"</code>	Ω

# Concatenación de cadenas

```
val numeroDePerros = 3
```

```
val numeroDeGatos = 2
```

Interpolación de variables

```
"Tengo $numeroDePerros perros" + " y $numeroDeGatos gatos"
```

=> Tengo 3 perros y 2 gatos

```
val numeroDePerros = 3
val numeroDeGatos = 2
println("Tengo $numeroDePerros perros"+" y $numeroDeGatos de gatos")
```

# Plantillas de cadena

Una expresión de plantilla comienza con un signo de dólar (\$) y puede ser un valor simple:

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

La diferencia es que:

\$variable → inserta directamente el valor de la variable.  
\${expresion} → evalúa la expresión y luego inserta el resultado.

O una expresión dentro de llaves ({ }):

```
val s = "abc"  
println("$s.length es ${s.length}")  
  
=> abc.length is 3
```

# Expresiones de plantilla de cadena

```
val camisas = 10
```

```
val pantalones = 5
```

```
"Tengo ${camisas + pantalones} prendas de ropa"
```

=> Tengo 15 prendas de ropa

```
val camisas = 10
val pantalones = 5
println("Tengo ${camisas + pantalones} prendas de ropa")
```

# Variables

<https://kotlinlang.org/docs/basic-syntax.html#variables>

# Variables

- **Potente inferencia de tipos**
  - Deja que el compilador infiera el tipo
  - Puedes declarar el tipo explícitamente si lo necesitas
- Variables mutables e inmutables
  - La inmutabilidad no es obligatoria, pero sí recomendada

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

# Especificar el tipo de la variable

## Notación de dos puntos

```
var ancho: Int = 12
```

```
var largo: Double = 2.5
```

**Importante:** Una vez que un tipo ha sido asignado por ti o por el compilador, no puedes cambiar el tipo o recibirás un error



- El tipo que almacenas en una variable se infiere cuando el compilador puede deducirlo por el contexto. Si lo deseas, también puedes especificar el tipo de una variable explícitamente usando la notación de dos puntos.
- Algunas cosas a tener en cuenta sobre la notación de dos puntos:
  - El tipo de dato se coloca **después** del nombre de la variable.
  - Siempre coloca un espacio después de :

# Variables mutables e inmutables

- Mutables (su valor puede cambiar)

```
var contador = 10
```

- Inmutables (su valor no puede cambiar después de ser asignado)

```
val nombre = "Ana"
```

Aunque **no es obligatorio** usar val, se recomienda hacerlo siempre que sea posible para favorecer la inmutabilidad y evitar errores accidentales

# var y val

```
var contador = 1
```

```
contador = 2
```

```
val tamaño = 1
```

```
tamaño = 2
```

=> Error: val cannot be reassigned

Hay que tener en cuenta que en Kotlin, por defecto, **las variables no pueden ser nulas**.

Hablaremos de la *seguridad frente a nulos* más adelante


```
// Inferencia de tipos
var edad = 25 // El compilador infiere Int
val nombre = "Ana" // El compilador infiere String
println("$edad $nombre")
```

```
// Declaración explícita
var altura: Double = 1.75
val esEstudiante: Boolean = true
println("$altura $esEstudiante")
```

// Variables mutables (var) e inmutables (val)

```
var contador = 1
contador = 2 // permitido
```

```
val pais = "España"
pais = "México" // Error: val no puede reasignarse
```



# Condicionales y bucles

<https://kotlinlang.org/docs/control-flow.html>

# Control de flujo

Kotlin ofrece varias formas de implementarlo:

- Sentencias f/Else
- Sentencias When
- Bucles For
- Bucles While

# Sentencias if/else

```
val copas = 30
```

```
val tazas = 50
```

```
if (copas > tazas) {  
    println("¡Demasiadas tazas!")  
} else {  
    println("¡No hay suficientes tazas!")  
}
```

=> ¡No hay suficientes tazas!

```
//Sentencias if/else
```

```
val copas = 30
```

```
val tazas = 50
```

```
if (copas > tazas) {  
    println("¡Demasiadas tazas!")  
} else {  
    println("¡No hay suficientes tazas!")  
}
```

# Sentencia if con multiples casos

```
val invitados = 30
if (invitados == 0) {
    println("No hay invitados")
} else if (invitados < 20) {
    println("Grupo pequeño de personas")
} else {
    println("Grupo grande de personas!")
}
⇒ ¡Grupo grande de personas!
```

```
//Sentencia if con múltiples casos
val invitados = 30

if (invitados == 0) {
    println("No hay invitados")
} else if (invitados < 20) {
    println("Grupo pequeño de personas")
} else {
    println("¡Grupo grande de personas!")
}
```



# Rangos

- Tipo de dato que contiene un intervalo de valores comparables (por ejemplo, enteros del 1 al 100 inclusive)
- Los rangos son acotados.
- Los objetos dentro de un rango pueden ser mutables o inmutables

- **Un rango (o *intervalo*)** define los **límites inclusivos** de una secuencia continua de valores de algún tipo comparable, como un `IntRange` (por ejemplo, los enteros del **1 al 100 inclusive**). El primer número es el punto inicial y el segundo número es el punto final.
- Todos los rangos son **acotados**, y el lado **izquierdo** del rango siempre es  $\leq$  que el lado **derecho** del rango
- Aunque la implementación en sí misma es **inmutable**, no existe ninguna restricción de que los objetos almacenados también deban ser inmutables. **Si se almacenan objetos mutables**, entonces el **rango se vuelve efectivamente mutable**

```
// Rango de enteros del 1 al 100
val rango = 1 ≤ .. ≤ 100
println(50 in rango)    // true
println(150 in rango)   // false

// Rango descendente
val descendente = 10 ≥ downTo ≥ 1
println(descendente.toList()) // [10, 9, 8, ..., 1]

// Rango con paso
val pares = 0 ≤ .. ≤ 10 step 2
println(pares.toList()) // [0, 2, 4, 6, 8, 10]

// Rango de caracteres
val letras = 'a' ≤ .. ≤ 'f'
println(letras.toList()) // [a, b, c, d, e, f]
```

# Rangos en sentencias if/else

```
val numeroDeEstudiantes = 50
if (numeroDeEstudiantes in 1..100) {
    println(numeroDeEstudiantes)
}
=> 50
```

```
val numeroDeEstudiantes = 50
if (numeroDeEstudiantes in 1 ≤ .. ≤ 100) {
    println(numeroDeEstudiantes)
}
```

**Nota:** No hay espacios alrededor del operador de rango ..

```
val numeroDeEstudiantes2 = 85

if (numeroDeEstudiantes2 in 1 ≤ .. ≤ 30) {
    println("Grupo pequeño de estudiantes")
} else if (numeroDeEstudiantes2 in 31 ≤ .. ≤ 60) {
    println("Grupo mediano de estudiantes")
} else if (numeroDeEstudiantes2 in 61 ≤ .. ≤ 100) {
    println("Grupo grande de estudiantes")
} else {
    println("Número de estudiantes fuera de rango")
}
```

- En Kotlin, la condición que evalúes también puede usar rangos. Los rangos te permiten **especificar un subconjunto de un grupo más grande**; por ejemplo, aquí solo nos interesan los enteros entre 1 y 100 (es decir, un rango de valores de tipo `Int`, o `IntRange`).
- Existen clases equivalentes a `IntRange` para otros tipos, como **`CharRange`** y **`LongRange`**.
- Opcional: También puedes definir un tamaño de paso (*step*) entre los límites del rango. Por ejemplo, en `1..8`, si definimos un paso de 2, nuestro rango incluirá los elementos 1, 3, 5, 7.

```
for (i in 1..8 step 2) {  
    print(i)  
}  
  
// Salida: 1357
```

```
// IntRange  
val numeros = 1..5  
println(numeros.toList()) // [1, 2, 3, 4, 5]  
  
// CharRange  
val letras2 = 'a'..'f'  
println(letras2.toList()) // [a, b, c, d, e, f]  
  
// LongRange  
val largos = 10000000000L..100000000005L  
println(largos.toList()) // [10000000000, 100000000001, ..., 100000000005]
```

# Sentencia when

```
when (resultado) {  
    0 -> println("Sin resultados")  
    in 1..39 -> println("¡Algunos resultados!")  
    else -> println("¡Muchos resultados!")  
}
```

⇒ That's a lot of results!

```
val resultado = 45
```

```
when (resultado) {  
    0 -> println("Sin resultados")  
    in 1..39 -> println("¡Algunos resultados!")  
    else -> println("¡Muchos resultados!")  
}
```

Además de una sentencia **when**, también puedes definir una expresión **when** que proporcione un valor de retorno



```
val dia = 3
```

```
when (dia) {
```

```
    1 -> println("Lunes")
```

```
    2 -> println("Martes")
```

```
    3 -> println("Miércoles")
```

```
    else -> println("Otro día")
```

```
}
```

```
val nombreDelDia = when (dia) {
```

```
    1 -> "Lunes"
```

```
    2 -> "Martes"
```

```
    3 -> "Miércoles"
```

```
    else -> "Otro día"
```

```
}
```

```
println("Hoy es $nombreDelDia")
```

- Esto hace que when sea muy flexible y legible, ya que puede reemplazar tanto a switch como a expresiones condicionales más largas

# Bucles for

```
val mascotas = arrayOf("perro", "gato", "canario")  
for (masc in mascotas) {  
    print(masc + " ")  
}
```

⇒ perro gato canario

En Kotlin, los bucles for se usan para recorrer **colecciones, arrays, rangos o secuencias de caracteres**

```
val mascotas = arrayOf("perro", "gato", "canario")  
for (masc in mascotas) {  
    print(masc + " ")  
}
```

No necesitas definir una variable iteradora ni incrementarla en cada pasada

# Bucles for: elementos e índice

```
for ((indice, masc) in mascotas.withIndex()) {  
    println("Elemento en $indice es $masc\n")  
}
```

⇒ Elemento en 0 es dog

Elemento en 1 es cat

Elemento en 2 es canary

```
for ((indice, masc) in mascotas.withIndex()) {  
    println("Elemento en $indice es $masc\n")  
}
```

# Bucles for: tamaños de paso y rangos

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

⇒ 12345

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

⇒ 54321

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

⇒ 35

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

⇒ defg

En Kotlin, los bucles for con rangos permiten controlar **el orden** (ascendente o descendente) y el **tamaño del paso** (step).

```
// Rango ascendente
for (i in 1 ≤ .. ≤ 5) print(i) // Salida: 12345

// Rango descendente
for (i in 5 ≥ downTo ≥ 1) print(i) // Salida: 54321

// Rango con paso
for (i in 3 ≤ .. ≤ 6 step 2) print(i) // Salida: 35

// Rango de caracteres
for (c in 'd' ≤ .. ≤ 'g') print(c) // Salida: defg
```

# Bucles while

```
var bicis = 0
while (bicis < 50) {
    bicis++
}
```

println("\$bicis Bicis en el parking\n")  
⇒ 50 bicycles in the bicycle rack

```
do {
    bicis--
} while (bicis > 50)
```

println("\$bicis Bicis en el parking\n")  
⇒ 49 bicycles in the bicycle rack

En Kotlin, un bucle while ejecuta su bloque de código **mientras la condición sea verdadera**. También existe do..while, que **se ejecuta al menos una vez** antes de comprobar la condición

```
var bicis = 0
while (bicis < 50) {
    bicis++
}
println("$bicis Bicis en el parking\n")

do {
    bicis--
} while (bicis > 50)
println("$bicis Bicis en el parking\n")
```

```
var bicicletas = 0
```

```
while (bicicletas < 5) {  
    println("Bicicletas en el estacionamiento: $bicicletas")  
    bicicletas++  
}
```

```
// Salida:  
// Bicicletas en el estacionamiento: 0  
// Bicicletas en el estacionamiento: 1  
// Bicicletas en el estacionamiento: 2  
// Bicicletas en el estacionamiento: 3  
// Bicicletas en el estacionamiento: 4
```

```
bicicletas = 5
```

```
do {  
    println("Bicicletas en el estacionamiento: $bicicletas")  
    bicicletas--  
} while (bicicletas > 0)
```

```
// Salida:  
// Bicicletas en el estacionamiento: 5  
// Bicicletas en el estacionamiento: 4  
// Bicicletas en el estacionamiento: 3  
// Bicicletas en el estacionamiento: 2  
// Bicicletas en el estacionamiento: 1
```

# Bucles repeat

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

⇒ ¡Hola! ¡Hola!

En Kotlin, el bucle `repeat(n)` ejecuta un bloque de código **exactamente n veces**. No necesita condición ni contador explícito: el compilador se encarga de todo.

Bucle	Sintaxis básica	Se ejecuta mientras...	Ejemplo	Salida
<b>for</b>	for (i in 1..5)	Se recorre un rango, colección o array	for (i in 1..3) print(i)	123
<b>while</b>	while (condición)	La condición sea verdadera	while (x > 0) { x-- }	Depende del valor inicial
<b>do..while</b>	do { ... } while (condición)	Igual que while, pero garantiza <b>una ejecución mínima</b>	do { println(x) } while (x > 0)	Al menos una vez
<b>repeat</b>	repeat(n) { ... }	Número fijo de veces (n)	repeat(3) { print("K") }	KKK



# Listas y arrays

<https://kotlinlang.org/docs/arrays.html#work-with-arrays>

# Listas vs Arrays

## Listas

- **Inmutables (listOf)** → no puedes agregar ni quitar elementos.
- **Mutables (mutableListOf)** → puedes añadir, eliminar o modificar elementos.
- **Tamaño dinámico** → crece o se reduce según agregues o elimines.

## Arrays

- **Tamaño fijo** → no se puede agrandar ni achicar después de crearlo.
- **Contenido mutable** → sí puedes cambiar los valores de cada posición.
- **Más “bajo nivel”** que las listas, parecido a Java.

# Listas

- Las listas son colecciones ordenadas de elementos.
- Los elementos de una lista pueden accederse de forma programática a través de sus índices
- Los elementos pueden repetirse más de una vez en una lista

Un ejemplo de una lista es una oración: es un grupo de palabras, donde el orden es importante y pueden repetirse

# Lista inmutable usando listOf()

Declara una lista usando listOf() e imprímela.

```
val instrumentos = listOf("trompeta", "piano", "violín")  
println(instrumentos)  
⇒ [trompeta, piano, violín]
```

```
val instrumentos = listOf("trompeta", "piano", "violín")  
println(instrumentos)    // [trompeta, piano, violín]
```

En Kotlin, una lista creada con listOf() es **inmutable**, lo que significa que **no puedes añadir ni eliminar elementos después de su creación**

# Lista mutable usando mutableListOf()

Lists can be changed using `mutableListOf()`

```
val miLista = mutableListOf("trompeta", "piano", "violín")
```

```
myList.remove("violin")
```

```
⇒ kotlin.Boolean = true
```

En Kotlin, una lista creada con `mutableListOf()` **sí permite** agregar, eliminar o modificar elementos después de su creación

Con una lista definida con **val**, no puedes cambiar a qué lista se refiere la variable, pero **sí puedes cambiar el contenido de la lista**

```
val frutas = mutableListOf("manzana", "banana")
```

```
// ✅ Puedes cambiar el contenido
```

```
frutas.add("cereza")
```

```
frutas.remove(element = "banana")
```

```
println(frutas) // [manzana, cereza]
```

```
// ❌ No puedes hacer que 'frutas' apunte a otra lista
```

```
// frutas = mutableListOf("pera", "uva") // Error: Val cannot be reassigned
```

# Arrays

- Los arrays almacenan **múltiples elementos**.
- Los elementos de un array pueden **accederse** de forma programática a través de sus **índices**.
- Los elementos de un array **son mutables**.
- El tamaño del **array es fijo**

```
// Crear un array
val mascotas2 = arrayOf("perro", "gato", "canario")

// Acceder por índice
println(mascotas2[0]) // perro
println(mascotas2[2]) // canario

// Modificar un elemento
mascotas2[1] = "conejo"
println(mascotas2.joinToString()) // perro, conejo, canario

// ❌ El tamaño es fijo: no puedes agregar o quitar elementos
// mascotas.add("tortuga") // Error
```



# Array usando arrayOf()

Un array de cadenas puede crearse usando `arrayOf()`

```
val mascotas = arrayOf("perro", "gato", "canario")  
println(java.util.Arrays.toString(mascotas))  
⇒ [perro, gato, canario]
```

```
val mascotas3 = arrayOf("perro", "gato", "canario")  
println(java.util.Arrays.toString(a = mascotas3))
```

Con un array definido con `val`, no puedes cambiar a qué array se refiere la variable, pero sí puedes cambiar el contenido del array

```
val numeros2 = arrayOf(1, 2, 3)
```

```
// ✅ Puedes modificar el contenido
```

```
numeros2[0] = 10
```

```
numeros2[2] = 30
```

```
println(numeros2.joinToString()) // 10, 2, 30
```

```
// ❌ No puedes reasignar la referencia
```

```
// numeros = arrayOf(4, 5, 6) // Error: Val cannot be reassigned
```

val → fija la **referencia** (no puedes cambiar el array completo).

var → permite cambiar tanto el **contenido** como la **referencia**.

```
var numeros3 = arrayOf(1, 2, 3)
```

```
// ✅ Modificar el contenido
```

```
numeros3[1] = 20
```

```
println(numeros3.joinToString()) // 1, 20, 3
```

```
// ✅ Cambiar la referencia a un nuevo array
```

```
numeros3 = arrayOf(4, 5, 6)
```

```
println(numeros3.joinToString()) // 4, 5, 6
```

# Arrays con tipos mezclados o de un solo tipo

Un array puede contener diferentes tipos.

```
val mix = arrayOf("sombrero", 2, true)
```

Un array también puede contener solo un tipo (enteros en este caso).

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

En Kotlin, un `arrayOf()` puede contener **elementos de distintos tipos**, aunque lo más recomendable es que todos sean del mismo tipo para mayor seguridad y legibilidad.

Constructor	Tipo resultante	Ejemplo	Salida
intArrayOf()	IntArray	val numeros = intArrayOf(1, 2, 3)	1, 2, 3
doubleArrayOf()	DoubleArray	val decimales = doubleArrayOf(1.1, 2.2)	1.1, 2.2
floatArrayOf()	FloatArray	val flotantes = floatArrayOf(1.5f, 2.5f)	1.5, 2.5
longArrayOf()	LongArray	val largos = longArrayOf(100L, 200L)	100, 200
charArrayOf()	CharArray	val letras = charArrayOf('a','b','c')	a, b, c
booleanArrayOf()	BooleanArray	val banderas = booleanArrayOf(true,false)	true, false
byteArrayOf()	ByteArray	val bytes = byteArrayOf(1, 2, 3)	1, 2, 3
shortArrayOf()	ShortArray	val cortos = shortArrayOf(10, 20)	10, 20

# Combinando arrays

Usa el operador +

```
val num1 = intArrayOf(1,2,3)
val num2 = intArrayOf(4,5,6)
val combinado = num1 + num2
println(Arrays.toString(combinado))
```

=> [4, 5, 6, 1, 2, 3]

En Kotlin, puedes **combinar dos arrays** usando el operador +.

El resultado es un **nuevo array** que contiene todos los elementos de ambos.

```
val num1 = intArrayOf(1,2,3)
val num2 = intArrayOf(4,5,6)
val combinado = num1 + num2
println(Arrays.toString(a = combinado))
```

# Seguridad frente a valores nulos

# Null Safety

<https://kotlinlang.org/docs/null-safety.html#collections-of-a-nullable-type>

# Null safety

- En Kotlin, las **variables** no pueden ser nulas por defecto.
- Puedes asignar **explícitamente una variable a null** usando el operador de llamada segura (?).
- Puedes permitir **excepciones de puntero nulo** usando el operador !!.
- Puedes **comprobar si es null** usando el operador Elvis (?:).

- Una característica distintiva de Kotlin es su **intento de eliminar el peligro** de las referencias nulas en el código.
- Las referencias nulas ocurren cuando el programa intenta acceder a una variable que no apunta a nada, lo que puede detener la ejecución y causar una **NullPointerException**



# Las variables no pueden ser nulas

En Kotlin, las variables nulas no están permitidas por defecto

Declara un `Int` y asígnale `null`.

```
var numero: Int = null
```

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

```
var numero: Int = null /Users/damiansualdea/IdeaProjects/Leccion001/src/Main.kt:291:23
Kotlin: Null cannot be a value of a non-null type 'Int'.
```

# Operador de llamada segura

El operador de llamada segura (?), colocado después del tipo, indica que una variable puede ser `null`.

Declara un `Int?` como nulo

```
var numero: Int? = null
```

**Cuando trabajas con tipos de datos complejos, como una lista:**

- Puedes permitir que los elementos de la lista sean nulos.
- Puedes permitir que la lista sea nula, pero si no lo es, sus elementos no pueden ser nulos.
- Puedes permitir tanto que la lista como sus elementos sean nulos.

En general, no establezcas una variable en `null`, ya que puede tener consecuencias no deseadas.

# Prueba de nulidad (Testing for null)

Comprueba si la variable `numeroDeLibros` no es `null`. Luego, decrementa esa variable

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

Ahora observa la forma de Kotlin de escribirlo, usando el operador de llamada segura

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

# El operador !!

Si estás seguro de que una variable no será nula, usa !! para forzarla a un tipo no nulo. Luego podrás llamar métodos o propiedades sobre ella

```
val len = s!!.length
```



*throws NullPointerException if s is null*

**Advertencia:** Como !! lanzará una excepción, solo debe usarse cuando realmente sería excepcional que una variable tuviera un valor nulo.

En la práctica, se recomienda **evitar !! siempre que sea posible** y usar en su lugar:

- ?. → llamada segura.
- ?: → operador Elvis con valor alternativo.

# Operador Elvis

Encadena comprobaciones de nulidad con el operador `? :`

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

Si `numeroDeLibros` no es nulo, decrementa y úsalo; de lo contrario, usa el valor que está después de `?:`, que en este caso es 0

El operador Elvis (`?:`) permite asignar un valor alternativo cuando una expresión resulta ser null

El operador `?:` a veces se llama el operador Elvis, porque se parece a una carita sonriente de lado con un copete, como el peinado característico de Elvis Presley.

# Ruta de aprendizaje

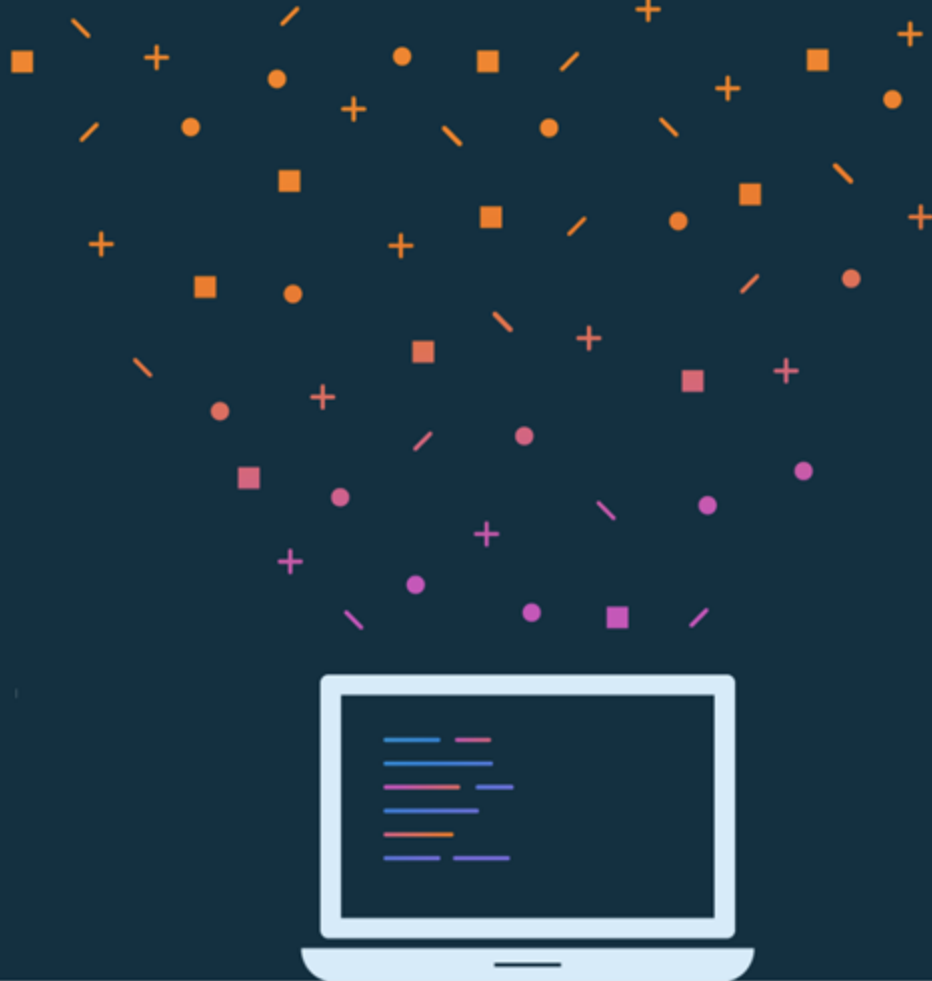
**Practica lo que has aprendido completando la ruta de ejercicios**

[Lesson 1: Kotlin basics](#)





# Lección 2: Funciones





# Introducción

## Lección 2: Funciones

- [Programas en Kotlin](#)
- [\(casi\) Todo tiene un valor](#)
- [Funciones en Kotlin](#)
- [Funciones compactas](#)
- [Lambdas y funciones de orden superior](#)
- [Filtros de listas](#)

# Programas en Kotlin

# Configuración inicial

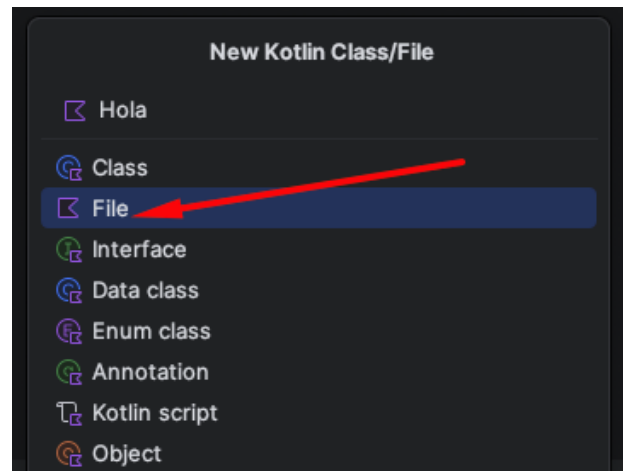
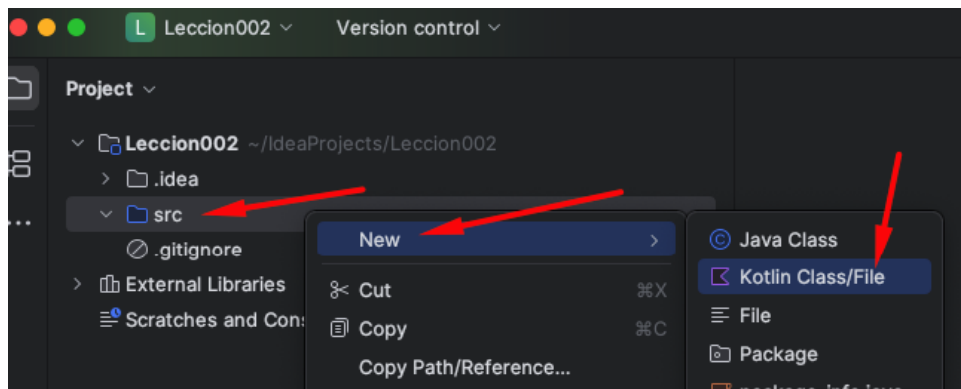
Antes de poder escribir código y ejecutar programas en Kotlin, necesitas preparar el entorno:

- Crea un archivo en tu proyecto
- Crea una función `main()`
- Pasa argumentos al `main()` (Opcional)
- Usa los argumentos pasados en llamadas a funciones (Optional)
- Ejecuta el programa

# Crea un nuevo archivo Kotlin

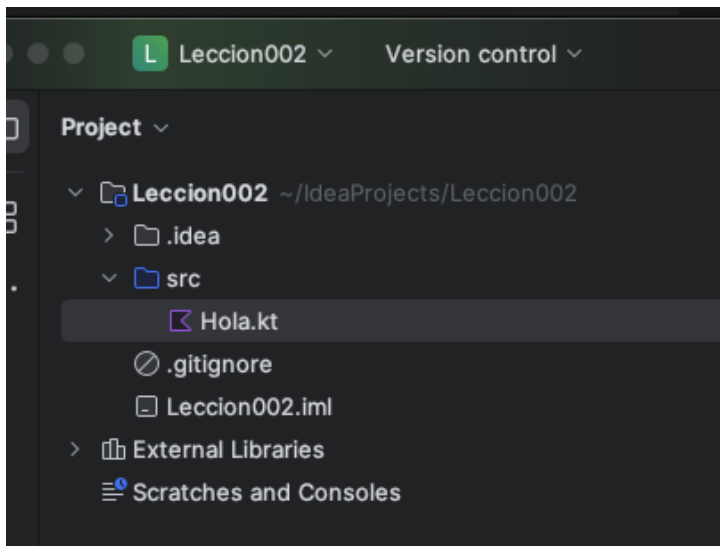
En el panel de Project en IntelliJ, haz clic derecho en la carpeta `src`.

- Selecciona **New > Kotlin File/Class**.
- Selecciona **File**, nombre del fichero `Hola`, y pulsa **Enter**.



# Crea un archivo en Kotlin

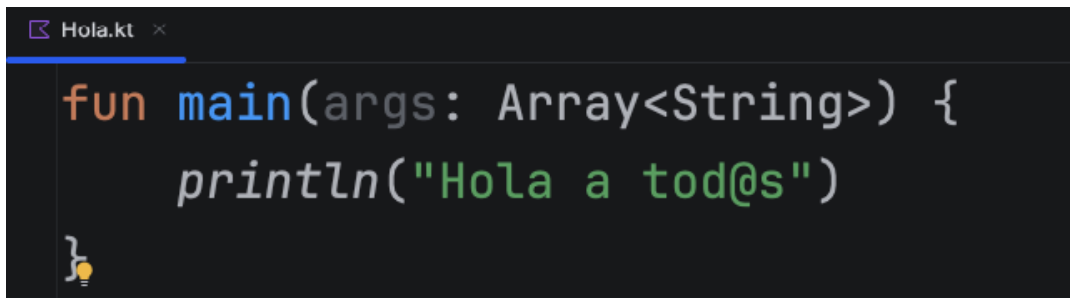
Ahora deberías de ver un archivo en la carpeta `src` llamado `Hola.kt`.



# Crea una función `main()`

`main()` es el punto de entrada para la ejecución de un programa en Kotlin.

En el archive `Hola.kt`:

A screenshot of a code editor window titled 'Hola.kt'. The code inside is a Kotlin function definition for 'main'. The function takes an 'Array<String>' parameter named 'args' and prints 'Hola a tod@s' to the console. The code is color-coded: 'fun' is orange, 'main' is blue, 'args' is grey, 'Array<String>' is grey, '{' is grey, 'println' is green, and the string 'Hola a tod@s' is green. A lightbulb icon is visible in the bottom left corner of the code editor.

```
fun main(args: Array<String>) {  
    println("Hola a tod@s")  
}
```

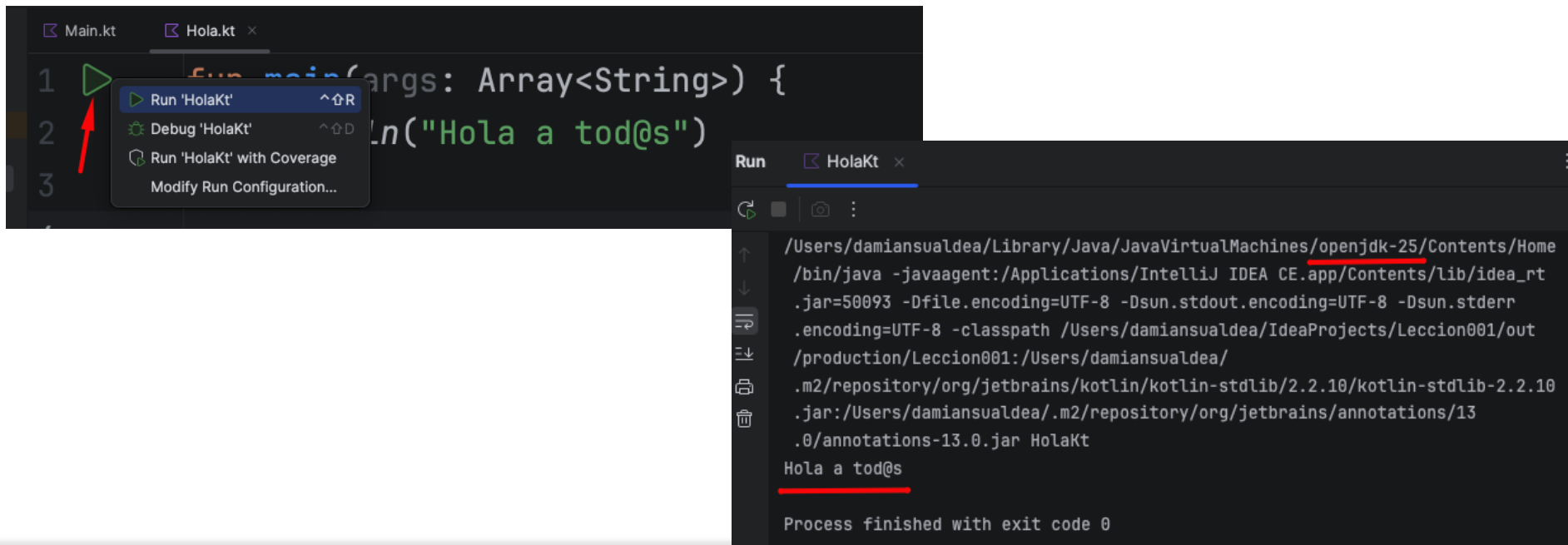
Los argumentos en la función `main()` son opcionales

- Al igual que en otros lenguajes, la **función main()** de Kotlin especifica **el punto de entrada para la ejecución**.
- Cualquier argumento de línea de comandos se pasa como un Array de cadenas (Array<String>).
- Esta función **no tiene una sentencia return**.
- En Kotlin, **toda función devuelve algo**, incluso cuando no se especifica nada explícitamente.
- Así, una función como main() devuelve el tipo **kotlin.Unit**, que es la forma de Kotlin de decir *sin valor*.

Nota: Cuando una función devuelve kotlin.Unit, **no es necesario especificarlo explícitamente**. Esto es diferente de algunos otros lenguajes, donde debes indicar explícitamente que no se devuelve nada

# Ejecuta tu programa Kotlin

Para ejecutar el programa, clic en el icono run (▶) a la izquierda de la función `main()`.





# (casi) Todo tiene un valor

# (casi) Todo es una expresión

En Kotlin, casi todo es una expresión y tiene un valor. Incluso una expresión `if` tiene un valor

```
val temperatura = 20
```

```
val haceCalor = if (temperatura > 40) true else false
```

```
println(haceCalor)
```

⇒ `false`

Nota: Los **bucles son una excepción** a la regla de que ‘todo tiene un valor’. No existe un valor lógico que devolver en un bucle `for` o `while`, por lo tanto no tienen valor. Si intentas asignar el valor de un bucle a algo, el compilador dará un error

# Valores de expresión

En Kotlin, casi todo es una **expresión**, lo que significa que **devuelve un valor**. Incluso si no devuelve un valor útil, lo que se devuelve es el tipo especial Unit

```
val esUnit = println("Esto es una expresión")  
println(esUnit)
```

⇒ Esto es una expresión  
kotlin.Unit

Algunos otros lenguajes tienen *sentencias*, que son líneas de código que no devuelven ningún valor. En Kotlin, casi todo es una expresión y tiene un valor, incluso si ese valor es kotlin.Unit. (Unit en Kotlin es equivalente a void en Java).

# Funciones en Kotlin

# Sobre las funciones

- Un bloque de código que realiza una tarea específica.
- Divide un programa grande en partes más pequeñas y modulares.
- Se declara usando la palabra clave fun.
- Puede recibir argumentos, ya sea con nombre o con valores por defecto

# Parts of a function

```
fun imprimeHola() {  
    println("Hola a tod@s")  
}
```

```
imprimeHola()
```

# Funciones que devuelven Unit

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

```
fun imprimeHola(name: String?): Unit {  
    println("Hola desde aquí!")  
}
```

`Unit` es un tipo que solo devuelve un valor:  
`Unit`.

# Funciones que devuelven Unit

La declaración del tipo de retorno `Unit` es opcional.

```
fun imprimeHola(name: String?): Unit {  
    println("Hola desde aquí!")  
}
```

Es equivalente a:

```
fun imprimeHola(name: String?) {  
    println("Hola desde aquí!")  
}
```



# Argumentos de funciones

Las funciones pueden recibir:

- Parámetros con valores por defecto
- Parámetros requeridos
- Argumentos nombrados

# Parámetros por defecto

Los valores por defecto proporcionan una alternativa si no se pasa ningún valor al parámetro.

```
fun conducir(valocidad: String = "rápido") {  
    println("Conduciendo $speed")  
}
```

Utiliza "=" después del tipo,  
para definir valores por  
defecto

drive() ⇒ conduciendo rápido


drive("lento") ⇒ conduciendo lento

drive(speed = "lento como una tortuga") ⇒ conduciendo lento como una tortuga

# Parámetros requeridos

En Kotlin, si un parámetro **no tiene un valor por defecto**, es **obligatorio** pasarlo al llamar la función

Parámetros requeridos




```
fun tempHoy(dia: String, temp: Int) {  
    println("Hoy es $day y la temperatura es $temp grados.")  
}
```

# Parámetros por defecto vs requeridos

Las funciones pueden tener una combinación de parámetros por defecto y parámetros requeridos

```
fun reformat(text: String,  
             dividirPorCamelCase: Boolean,  
             separador: Char,  
             normalizar: Boolean = true){
```

 Tiene un valor por defecto

Posibles llamadas a la función:

```
reformat("Hoy es un grand día", false, '_')
```

```
reformat("Hoy es un grand día", false, '_', false)
```

# Argumentos nombrados

Para mejorar la legibilidad, usa argumentos nombrados en los parámetros requeridos

```
reformat(str, dividirPorCamelCase = false, separador = '_')
```

Usar **argumentos nombrados**:

- Hace que el código sea más **fácil de leer y entender**.
- Evita confusiones si los parámetros son del mismo tipo (Boolean, Char, etc.).

Se considera una **buena práctica** colocar los argumentos con valores por defecto después de los argumentos posicionales, de modo que quien **llama** a la función **solo tenga que especificar los parámetros requeridos**

# Funciones compactas single-expressions functions

# Funciones de una sola expresión

Las funciones compactas, o funciones de una sola expresión, hacen tu código más conciso y legible

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



**Forma clásica**

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



**Forma compacta**

# Lambdas y funciones de orden superior



# Las funciones en Kotlin son ciudadanos de primera clase

- Las funciones en Kotlin pueden **almacenarse** en **variables** y **estructuras de datos**.
- Pueden pasarse como **argumentos** a otras funciones de orden superior, y también pueden ser devueltas por ellas.
- Puedes usar funciones de orden superior para crear nuevas funciones ‘integradas’

# Funciones lambda

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

**Parámetros y tipos**

**Función flecha**

```
var nivelSuciedad = 20
val filtroDeAgua = {nivel: Int -> nivel / 2}
println(filtroDeAgua(nivelSuciedad))
⇒ 10
```

**Código a ejecutar**

Una lambda es una expresión que crea una función. Pero en lugar de declarar una función con nombre, declaras una función que no tiene nombre.

Lo que hace útil a esto es que la expresión lambda puede pasarse ahora como dato.

En otros lenguajes, las lambdas se llaman *funciones anónimas*, *funciones literales* u otros nombres similares.

Al igual que las funciones con nombre, las lambdas pueden tener parámetros.

En las lambdas, los parámetros (y sus tipos, si es necesario) van a la izquierda de lo que se llama una *flecha de función* (->).

El código a ejecutar va a la derecha de la flecha de función.

Una vez que la lambda se asigna a una variable, puedes llamarla igual que a una función.

# Sintaxis para tipos de función

La sintaxis de Kotlin para los tipos de función está estrechamente relacionada con su sintaxis para las lambdas. Declara una variable que contenga una función.

```
val filtroDAgua: (Int) -> Int = {nivel -> nivel / 2}
```

**Nombre de la  
variable**

**Tipo de dato de la variable  
(tipo función)**

**Función**

# Ruta de aprendizaje

**Practica lo que has aprendido  
completando la ruta de ejercicios:**

[Lesson 2: Functions](#)





# Lección 3: Clases y objetos



# Introducción

## Lección 3: Clases y objetos

- [Clases](#)
- [Herencia](#)
- [Funciones de extensión](#)
- [Clases especiales](#)
- [Organizando el código](#)

# Classes



# Clases

- Las clases son planos para los objetos.
- Las clases definen métodos que operan sobre sus instancias de objeto

**Objetos  
instancias**

**Clase**



# Clases vs Instancias

## Clase Casa

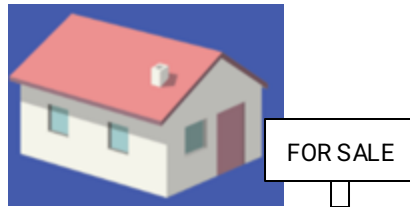
### Datos

- `color (String)`
- `numeroVentanas(Int)`
- `enVenta (Boolean)`

### Comportamiento

- `actualizaColor()`
- `ponerEnVenta()`

## Instancias de objetos



# Definir y usar una clase

Definición de clase

```
class Casa {  
    val color: String = "blanco"  
    val numeroVentas: Int = 2  
    val enVenta: Boolean = false  
  
    fun actualizaColor(nuevoCol: String){...}  
    ...  
}
```

Instancia la clase: objeto

```
val miCasa = House()  
println(miCasa)
```

# Constructores

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

- Ningún parámetro

```
class A
```

- Parámetros

- Sin `var` o `val` → La copia existe **solo dentro del alcance del constructor**

```
class B(x: Int)
```

- Con `var` o `val` → a copia existe en **todas las instancias de la clase**

```
class C(val y: Int)
```

**x: Int → parámetro local**

**val x: Int o var x: Int → propiedad de la clase**

```
class B (x: Int) {  
    init {  
        println("El valor de x es $x")  
    }  
}
```

```
class C (val y: Int)
```

```
fun main () {  
    val obj = B(10)  
        println(obj.y)  
        //Error  
}
```

```
fun main () {  
    val obj = C(20) {  
        println(obj.y)  
    }  
}
```

# Ejemplos de constructores

```
class A
```

```
val aa = A()
```

```
class B(x: Int)
```

```
val bb = B(12)
println(bb.x)
=> compiler error unresolved
reference
```

```
class C(val y: Int)
```

```
val cc = C(42)
println(cc.y)
=> 42
```

- El constructor de la clase A no tiene parámetros.
- El constructor de la clase B tiene un parámetro de entrada: x, que es un Int. Como el parámetro no está marcado con var o val, la variable x no existe fuera del alcance del constructor. Por lo tanto, si creamos una instancia de objeto llamada bb e intentamos acceder a la propiedad x, obtendremos un error de compilación. La propiedad x no existe en el objeto.
- En el tercer caso, tenemos un constructor para la clase C con un parámetro de entrada: un val llamado y. Si creas una instancia de objeto llamada cc, puedes acceder a la propiedad y, que en este caso tiene el valor 42.
- En resumen, puedes definir las propiedades directamente dentro del constructor usando var o val (como se ve en el tercer ejemplo).

# Parámetros por defecto

Las instancias de clase pueden tener valores por defecto.

- Usa valores por defecto para reducir la cantidad de constructores necesarios.
- Los parámetros por defecto se pueden combinar con parámetros requeridos.
- Más conciso (no es necesario tener múltiples versiones de constructores)

```
class Caja(val largo: Int, val ancho:Int = 20, val alto:Int = 40)
```

```
val c1 = Box(100, 20, 40)
```

```
val c2 = Box(largo = 100)
```

```
val c3 = Box(largo = 100, ancho = 20, alto = 40)
```



# Constructor primario

Declara el constructor primario dentro del encabezado de la clase

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

Esto es técnicamente equivalente a:

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

- En todos los ejemplos que hemos visto hasta ahora, el constructor está dentro del encabezado de la clase. Esto se llama el *constructor primario*.
- Esta sintaxis hace que Kotlin sea más conciso a la hora de definir clases. Técnicamente, el primer fragmento de código es equivalente al segundo (que es más verboso...).
- El segundo fragmento de código se parece más a cómo definirías una clase en otro lenguaje, donde el constructor se declara explícitamente por separado. Pero en Kotlin, escribe tu código usando el primer fragmento.

¿Notas el bloque init? Vamos a echarle un vistazo

# Bloque de inicialización

- Cualquier código de inicialización requerido se ejecuta en un bloque especial `init`.
- Se permiten múltiples bloques `init`.
- Los bloques `init` se convierten en el cuerpo del constructor primario.

Puedes tener **varios bloques `init`**, y se ejecutan en el **orden en que aparecen**. El compilador los trata como si fueran parte del **cuerpo del constructor primario**.

# Ejemplo de bloque de inicialización

Use the `init` keyword:

```
class Square(val side: Int) {  
    init {  
        println(side * 2)  
    }  
}
```

```
val s = Square(10)  
=> 20
```

# Multiple constructors

- Use the `constructor` keyword to define secondary constructors
- Secondary constructors must call:
  - The primary constructor using `this` keyword
  - OR
  - Another secondary constructor that calls the primary constructor
- Secondary constructor body is not required

# Multiple constructors example

```
class Circle(val radius:Double) {  
    constructor(name:String) : this(1.0)  
    constructor(diameter:Int) : this(diameter / 2.0) {  
        println("in diameter constructor")  
    }  
    init {  
        println("Area: ${Math.PI * radius * radius}")  
    }  
}  
  
val c = Circle(3)
```

# Properties

- Define properties in a class using `val` or `var`
- Access these properties using dot `.` notation with property name
- Set these properties using dot `.` notation with property name (only if declared a `var`)

# Person class with name property

```
class Person(var name: String)

fun main() {
    val person = Person("Alex")

    println(person.name) ← Access with .<property name>
    person.name = "Joey" ← Set with .<property name>
    println(person.name)
}
```



# Custom getters and setters

If you don't want the default `get/set` behavior:

- Override `get()` for a property
- Override `set()` for a property (if defined as a `var`)

**Format:** `var` propertyName: DataType = initialValue  
    `get()` = ...  
    `set(value)` {  
        ...  
    }

# Custom getter

```
class Person(val firstName: String, val lastName:String) {  
    val fullName:String  
        get() {  
            return "$firstName $lastName"  
        }  
}
```

```
val person = Person("John", "Doe")  
println(person.fullName)  
=> John Doe
```

# Custom setter

```
var fullName:String = ""  
get() = "$firstName $lastName"  
set(value) {  
    val components = value.split(" ")  
    firstName = components[0]  
    lastName = components[1]  
    field = value  
}
```

```
person.fullName = "Jane Smith"
```

# Member functions

- Classes can also contain functions
- Declare functions as shown in *Functions* in Lesson 2
  - `fun` keyword
  - Can have default or required parameters
  - Specify return type (if not `Unit`)

# Inheritance

# Inheritance

- Kotlin has single-parent class inheritance
- Each class has exactly one parent class, called a superclass
- Each subclass inherits all members of its superclass including ones that the superclass itself has inherited

If you don't want to be limited by only inheriting a single class, you can define an interface since you can implement as many of those as you want.

# Interfaces

- Provide a contract all implementing classes must adhere to
- Can contain method signatures and property names
- Can derive from other interfaces

**Format:** `interface` NameOfInterface { interfaceBody }

# Interface example

```
interface Shape {  
    fun computeArea() : Double  
}  
  
class Circle(val radius:Double) : Shape {  
    override fun computeArea() = Math.PI * radius * radius  
}  
  
val c = Circle(3.0)  
println(c.computeArea())  
=> 28.274333882308138
```



# Extending classes

To extend a class:

- Create a new class that uses an existing class as its core (subclass)
- Add functionality to a class without creating a new one (extension functions)

# Creating a new class

- Kotlin classes by default are not subclassable
- Use `open` keyword to allow subclassing
- Properties and functions are redefined with the `override` keyword

# Classes are final by default

Declare a class

```
class A
```

Try to subclass A

```
class B : A
```

=>Error: A is final and cannot be inherited from

# Use open keyword

Use `open` to declare a class so that it can be subclassed.

Declare a class

```
open class C
```

Subclass from C

```
class D : C()
```

# Overriding

- Must use `open` for properties and methods that can be overridden (otherwise you get compiler error)
- Must use `override` when overriding properties and methods
- Something marked `override` can be overridden in subclasses (unless marked `final`)

# Abstract classes

- Class is marked as `abstract`
- Cannot be instantiated, must be subclassed
- Similar to an interface with the added the ability to store state
- Properties and functions marked with `abstract` must be overridden
- Can include non-abstract properties and functions

# Example abstract classes

```
abstract class Food {  
    abstract val kcal : Int  
    abstract val name : String  
    fun consume() = println("I'm eating ${name}")  
}  
class Pizza() : Food() {  
    override val kcal = 600  
    override val name = "Pizza"  
}  
fun main() {  
    Pizza().consume()    // "I'm eating Pizza"  
}
```

# When to use each

- Defining a broad spectrum of behavior or types? Consider an interface.
- Will the behavior be specific to that type? Consider a class.
- Need to inherit from multiple classes? Consider refactoring code to see if some behavior can be isolated into an interface.
- Want to leave some properties / methods abstract to be defined by subclasses? Consider an abstract class.
- You can extend only one class, but implement one or more interfaces.



# Extension functions

# Extension functions

Add functions to an existing class that you cannot modify directly.

- Appears as if the implementer added it
- Not actually modifying the existing class
- Cannot access private instance variables

**Format:** `fun` ClassName.functionName( params ) { body }

# Why use extension functions?

- Add functionality to classes that are not open
- Add functionality to classes you don't own
- Separate out core API from helper methods for classes you own

Define extension functions in an easily discoverable place such as in the same file as the class, or a well-named function.

# Extension function example

Add `isOdd()` to `Int` class:

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

Call `isOdd()` on an `Int`:

```
3.isOdd()
```

Extension functions are very powerful in Kotlin!

# Special classes

# Data class

- Special class that exists just to store a set of data
- Mark the class with the `data` keyword
- Generates getters for each property (and setters for vars too)
- Generates `toString()`, `equals()`, `hashCode()`, `copy()` methods, and destructuring operators

**Format:** `data class` <NameOfClass>( parameterList )

# Data class example

Define the data class:

```
data class Player(val name: String, val score: Int)
```

Use the data class:

```
val firstPlayer = Player("Lauren", 10)  
println(firstPlayer)  
=> Player(name=Lauren, score=10)
```

Data classes make your code much more concise!

# Pair and Triple

- `Pair` and `Triple` are predefined data classes that store 2 or 3 pieces of data respectively
- Access variables with `.first`, `.second`, `.third` respectively
- Usually named `data` classes are a better option (more meaningful names for your use case)



# Pair and Triple examples

```
val bookAuthor = Pair("Harry Potter", "J.K. Rowling")  
println(bookAuthor)  
=> (Harry Potter, J.K. Rowling)
```

```
val bookAuthorYear = Triple("Harry Potter", "J.K. Rowling", 1997)  
println(bookAuthorYear)  
println(bookAuthorYear.third)  
=> (Harry Potter, J.K. Rowling, 1997)  
1997
```

# Pair to

`Pair`'s special `to` variant lets you omit parentheses and periods (infix function).

It allows for more readable code

```
val bookAuth1 = "Harry Potter".to("J. K. Rowling")
```

```
val bookAuth2 = "Harry Potter" to "J. K. Rowling"
```

```
=> bookAuth1 and bookAuth2 are Pair (Harry Potter, J. K. Rowling)
```

Also used in collections like `Map` and `HashMap`

```
val map = mapOf(1 to "x", 2 to "y", 3 to "zz")
```

```
=> map of Int to String {1=x, 2=y, 3=zz}
```

# Enum class

User-defined data type for a set of named values

- Use `this` to require instances be one of several constant values
- The constant value is, by default, not visible to you
- Use `enum` before the `class` keyword

**Format:** `enum class EnumName { NAME1, NAME2, ... NAMEn }`

Referenced via `EnumName.<ConstantName>`

# Enum class example

Define an `enum` with red, green, and blue colors.

```
enum class Color(val r: Int, val g: Int, val b: Int) {  
    RED(255, 0, 0), GREEN(0, 255, 0), BLUE(0, 0, 255)  
}
```

```
println("" + Color.RED.r + " " + Color.RED.g + " " + Color.RED.b)  
=> 255 0 0
```

# Object/singleton

- Sometimes you only want one instance of a class to ever exist
- Use the `object` keyword instead of the `class` keyword
- Accessed with `NameOfObject.<function or variable>`

# Object/singleton example

```
object Calculator {  
    fun add(n1: Int, n2: Int): Int {  
        return n1 + n2  
    }  
}
```

```
println(Calculator.add(2,4))  
=> 6
```

# Companion objects

- Lets all instances of a class share a single instance of a set of variables or functions
- Use `companion` keyword
- Referenced via `ClassName.PropertyOrFunction`

# Companion object example

```
class PhysicsSystem {  
    companion object WorldConstants {  
        val gravity = 9.8  
        val unit = "metric"  
        fun computeForce(mass: Double, accel: Double): Double {  
            return mass * accel  
        }  
    }  
}  
  
println(PhysicsSystem.WorldConstants.gravity)  
println(PhysicsSystem.WorldConstants.computeForce(10.0, 10.0))  
=> 9.8100.0
```



# Organizing your code

# Single file, multiple entities

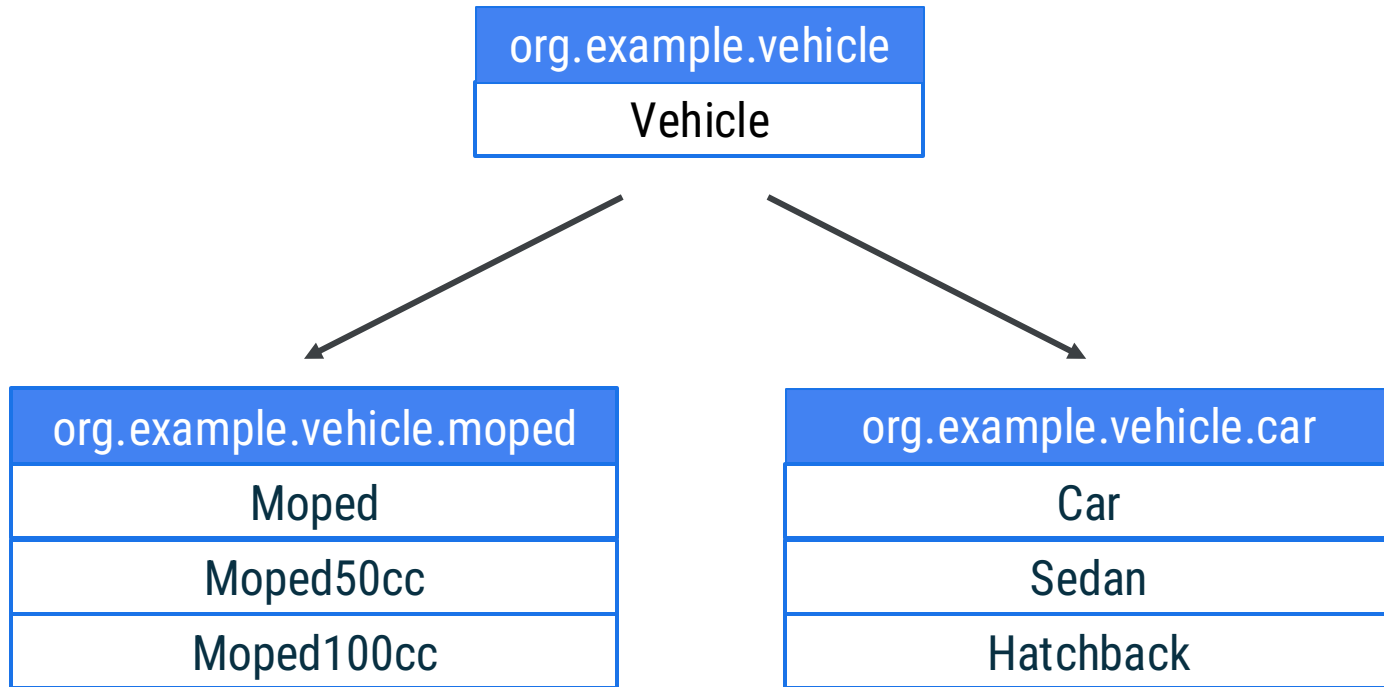
- Kotlin DOES NOT enforce a single entity (class/interface) per file convention
- You can and should group related structures in the same file
- Be mindful of file length and clutter

# Packages

- Provide means for organization
- Identifiers are generally lower case words separated by periods
- Declared in the first non-comment line of code in a file following the `package` keyword

```
package org.example.game
```

# Example class hierarchy



# Visibility modifiers

Use visibility modifiers to limit what information you expose.

- `public` means visible outside the class. Everything is public by default, including variables and methods of the class.
- `private` means it will only be visible in that class (or source file if you are working with functions).
- `protected` is the same as `private`, but it will also be visible to any subclasses.

# Summary

# Summary

In Lesson 3, you learned about:

- Classes, constructors, and getters and setters
- Inheritance, interfaces, and how to extend classes
- Extension functions
- Special classes: data classes, enums, object/singletons, companion objects
- Packages
- Visibility modifiers

# Pathway

Practice what you've learned by completing the pathway:

[Lesson 3: Classes and objects](#)

