# Introducción A Kotlin

En este tutorial te mostraremos una introducción a Kotlin. Verás en breve: que es el lenguaje Kotlin, a que plataformas apunta, los beneficios de codificar con él y en que IDEs podemos crear programas con este lenguaje.

* [¿Qué Es Kotlin ?](https://www.develou.com/introduccion-a-kotlin/#Que_Es_Kotlin)
* [Plataformas Objetivo Soportadas](https://www.develou.com/introduccion-a-kotlin/#Plataformas_Objetivo_Soportadas)
* [Interoperabilidad Con Java](https://www.develou.com/introduccion-a-kotlin/#Interoperabilidad_Con_Java)
* [¿Por Qué Usar Kotlin?](https://www.develou.com/introduccion-a-kotlin/#Por_Que_Usar_Kotlin)
* [IDEs Que Soportan Kotlin](https://www.develou.com/introduccion-a-kotlin/#IDEs_Que_Soportan_Kotlin)
* [¿Qué Sigue?](https://www.develou.com/introduccion-a-kotlin/#Que_Sigue)

## ¿Qué Es Kotlin ?

[Kotlin](https://kotlinlang.org/) es un lenguaje de propósito general que soporta tanto programación funcional como orientada a objetos.

A su vez es un proyecto de código abierto desarrollado por la compañía [JetBrains](https://www.jetbrains.com/es-es/" \t "_blank) , la cual es su principal conductor, con la ayuda de la [comunidad](https://discuss.kotlinlang.org/).

Usa un tipado estático como Java, pero también puede omitir los tipos pareciéndose a los [lenguajes tipados dinámicamente](https://docs.oracle.com/cd/E57471_01/bigData.100/extensions_bdd/src/cext_transform_typing.html).

Nace con la necesidad de mejorar la legibilidad y expresiones que Java usa.

## Plataformas Objetivo Soportadas

Kotlin permite compilar aplicativos para:

* **JVM** (Android y aplicaciones servidor)
* **JavaScript**
* **Nativo** (iOS).

En esta serie de tutoriales usaremos JVM para ejecutar los ejemplos propuestos al aprender los temas centrales de Kotlin.

Sin embargo, cuando pases a codificar [apps Android](https://www.develou.com/android/) será fácil ya que usarás los mismos fundamentos vistos en esta guía de Kotlin.

## Interoperabilidad Con Java

Kotlin es un lenguaje de programación que corre sobre **la máquina virtual de Java**. Esto facilita la comunicación entre ambos lenguajes.

La interoperabilidad posibilita que Kotlin pueda usar todas los frameworks y librerías de Java.

O incluso mezclar ambos códigos en un mismo proyecto.

El código fuente de Kotlin es compilado a bytecodes de Java por la máquina virtual, por lo que generará archivos .class.

## ¿Por Qué Usar Kotlin?

Usando Kotlin te beneficiarás de:

* **Menos código**: JetBrains estima que puedes reducir hasta un 40% de líneas de código en comparación a Java.
* **Código más seguro**: Reduce la aparición de excepciones de nulos al usar tipos no anulables.
* **Interoperabilidad**: Kotlin es 100% interoperable con Java. Lo que significa que puedes llamar código Java desde Kotlin y viceversa.
* **Corrutinas**: Las corrutinas de Kotlin simplifican la ejecución de tareas asíncronas
* **Desarrollo Multiplataforma**: Con Kotlin puedes desarrollar aplicaciones Android, iOS y aplicaciones web. Con el fin de compartir código en común entre ellas.

## IDEs Que Soportan Kotlin

Para programar en Kotlin puedes elegir entre:

* [IntelliJ IDEA](https://www.jetbrains.com/es-es/idea/)
* [Android Studio](https://developer.android.com/studio)
* [Eclipse](https://www.eclipse.org/downloads/)
* [NetBeans](https://netbeans.apache.org/download/index.html).

También puedes compilar programas Kotlin con su propio [compilador de línea de comandos](https://kotlinlang.org/docs/tutorials/command-line.html).

# Variables En Kotlin

En este tutorial aprenderás a declarar y usar variables en Kotlin para almacenar datos en memoria. Verás que existen variables de solo lectura (read-only) y mutables, además de la inferencia que el compilador de Kotlin realiza.

* [¿Qué Es Una Variable?](https://www.develou.com/variables-en-kotlin/#Que_Es_Una_Variable)
* [Variables De Solo Lectura](https://www.develou.com/variables-en-kotlin/#Variables_De_Solo_Lectura)
* [Variables Mutables](https://www.develou.com/variables-en-kotlin/#Variables_Mutables)
* [Inferencia De Tipos](https://www.develou.com/variables-en-kotlin/#Inferencia_De_Tipos)
* [¿Qué Sigue?](https://www.develou.com/variables-en-kotlin/#Que_Sigue)

## ¿Qué Es Una Variable?

Una variable es una posición en memoria que contiene un valor asociado a un nombre. Su contenido se especifica por el tipo de dato que le especifiques al compilador.

En [Kotlin](https://kotlinlang.org/docs/reference/basic-syntax.html" \l "defining-variables" \t "_blank) el cambio del valor de una variable a lo largo del flujo del programa, depende del identificador usado como verás a continuación.

## Variables De Solo Lectura

Una variable de solo lectura (read-only) es una variable que no puede ser reasignada.

Para declararlas, usa la palabra reservada val. y especifica su tipo de dato a su derecha con dos puntos (:).

O declárala con su tipo y asígnala en una línea futura.

val xPos: Int = 1 // Asignación junto a declaración

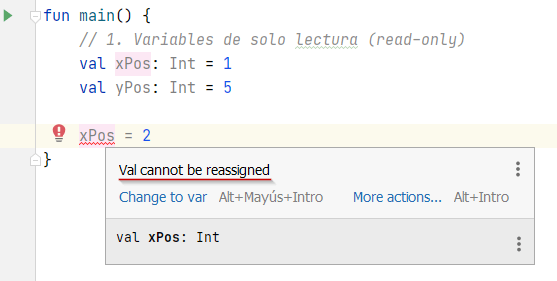
val yPos: Int // Declaración

yPos = 5 // Asignación

El valor que les asignes en su declaración no podrás cambiarlo posteriormente. Si lo intentas el compilador te mostrará este error.

Val cannot be reassigned

En [IntelliJ IDEA](https://www.develou.com/hola-mundo-en-kotlin/) el compilador te lo advertirá justo en la escritura:



Puedes interpretar las variables de solo lectura como la correspondencia al modificador final de Java.

Si no defines el tipo ni asignas un valor a la variable en la declaración, obtendrás este error:

This variable must either have a type annotation or be initialized

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

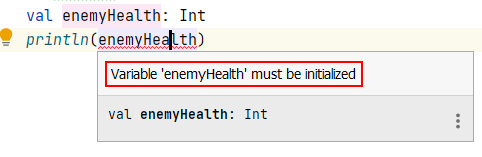
En el caso de que cuentes con declaración con tipo, pero intentes usarla en una sentencia se te reclamará la inicialización.

val enemyHealth: Int

println(enemyHealth)

El error que te arrojará sería este:

Variable 'enemyHealth' must be initialized



## Variables Mutables

Este tipo de variables las declaras con la palabra reservada var. Te permiten modificar su valor cuando lo desees.

Si haces que la variable del ejemplo de la sección anterior sea mutable, ya no tendrás el error y se asignara el nuevo valor.

fun main() {

// 2. Variables mutables

var xPos: Int = 1

val yPos: Int = 5

xPos = 2

println("Coordenada actual del jugador: ($xPos, $yPos)")

}

Si corres la aplicación obtendrás como salida:

Coordenada actual del jugador: (2, 5)

**Nota**: El operador $ que viste al interior del string de println(), permite insertar el valor de la variable referenciada. Veremos esto más en detalle en el tutorial de Strings.

## Inferencia De Tipos

En Kotlin puedes omitir los tipos de las variables en su declaración, ya que el compilador puede inferirlos de acuerdo al valor de su asignación.

val playerName = "Oliver" // Se infiere :String

val playerHealth = 75 // Se infiere :Int

val playerLucky = 0.2 // Se infiere :Double

Pero que el compilador te apoye con la asignación del tipo, no significa que nunca los especifiques o que no respetes el contenido.

Por ejemplo, intentar asignar un tipo de naturaleza distinta.

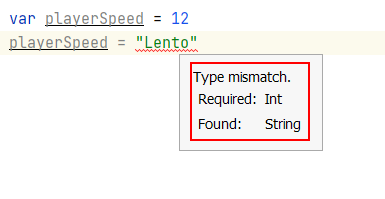
var playerSpeed = 12

playerSpeed = "Lento" // Error

El compilador arrojará el siguiente error, ya que estas asignando una cadena a lo que el compilador había inferido como entero:

Type mismatch: inferred type is String but Int was expected

O en caso de IntelliJ IDEA lo verás de forma emergente.



En la medida que puedas, busca usar val en vez de var, ya que facilitará el enfoque funcional en tu aplicación, el cual es más fácil de entender y soportar.

## El Tipo String

La [clase String](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-string/) es la que representa a los strings Kotlin con el fin de tratar texto en nuestros programas.

class String : Comparable<String>, CharSequence

Cada literal String de a forma "cadena" es creado como una instancia de la clase. Por lo que solo basta con usar las comillas dobles para encerrar múltiples caracteres que representen el String deseado:

val customerName:String = "Claudia Martinez"

val customerName = "Claudia Martinez" // Puedes omitir el tipo

## Caracteres De Un String

Debido a que String implementa la interfaz CharSecuence, es posible interpretar el conjunto como una secuencia de elementos Char.

Esto te permitirá leer los caracteres en sus posiciones a través de get() o su operador equivalente de acceso en corchetes string[indice]:

println(customerName[5])

println(customerName.get(5))

También es posible recorrerlos a través del [bucle for](https://www.develou.com/for-en-kotlin/):

for(char in customerName){

print(char)

}

Incluso usar la propiedad de extensión indices para obtener el rango de índices que sirvan de referencia en el bucle for:

for (index in customerName.indices) {

println(customerName[index])

}

Si quieres saber el tamaño de la secuencia de caracteres, accede a la propiedad miembro length del string.

println(customerName.length) // 16

## Concatenar Strings En Kotlin

Para crear la combinación de varios valores en un solo String, usa el operador de adición + como herramienta de concatenación.

Este te permite generar una nueva cadena entre un String y otro tipo de la siguiente forma:

// Concatenación

val a = "José" + " Monteria"

val b = "Cantidad de ahorros: " + 536.4

val c = "Resultado: " + (2 + 4)

Puedes usar a combinación de variables o literales si así lo deseas. Incluso puedes usar el operador de asignación y adición en su forma simplificada:

var d = "Nueva concatenación"

d += '!'// "Nueva concatenación!"

## Strings Con Múltiples Líneas

En ocasiones es necesario crear literales de String que posean múltiples líneas y que sean interpretadas en su forma plana (raw strings).

Esto puedes lograrlo usando la sintaxis de triple doble comillas (""") para encerrar el conjunto de caracteres.

fun main() {

// raw string

val welcomeText = """

¡Bienvenido a la guía de Kotlin de develou.com!

En esta serie de tutoriales aprenderás

los conceptos básicos sobre el lenguaje

y las herramientas necesarias para probarlo

"""

println(welcomeText)

}

Salida:

¡Bienvenido a la guía de Kotlin de develou.com!

En esta serie de tutoriales aprenderás

los conceptos básicos sobre el lenguaje

y las herramientas necesarias para probarlo

Como ves, la construcción del String con múltiples líneas conserva la sangría y espacios de origen.

Para eliminarlos usa la función trimMargin(), la cual usa como referencia el prefijo '|' como delimitador de las márgenes:

fun main() {

// raw string

val welcomeText = """

|¡Bienvenido a la guía de Kotlin de develou.com!

|En esta serie de tutoriales aprenderás

|los conceptos básicos sobre el lenguaje

|y las herramientas necesarias para probarlo

""".trimMargin()

println(welcomeText)

}

**Salida:**

¡Bienvenido a la guía de Kotlin de develou.com!

En esta serie de tutoriales aprenderás

los conceptos básicos sobre el lenguaje

y las herramientas necesarias para probarlo

## Plantillas De Strings En Kotlin

En Kotlin es posible crear expresiones interpoladas sobre Strings o también conocidas como la sintaxis de plantillas de Strings ([String Templates](https://kotlinlang.org/docs/basic-types.html" \l "string-templates" \t "_blank)).

La interpolación te permite concatenar literales junto a fragmentos de expresiones antecedidas por el símbolo '$'. Hay dos formas de hacerlo:

* $id, donde id es un identificador simple
* ${e}, donde e es una expresión valida en Kotlin

Veamos un ejemplo de un identificador simple:

fun main() {

val playerName = "NKiller"

println("Tu nuevo nombre es $playerName")

}

**Salida:**

Tu nuevo nombre es NKiller

Como ves, el fragmento interpolado $playerName es convertido a String y el resultado final es la concatenación de todos los fragmentos.

En el caso de las expresiones, debes usar llaves ({}) para encerrar la expresión. De esa forma el resultado sigue el mismo de conversión a String y es aplicado a la concatenación final:

fun main() {

println("Tamaño del nick:${playerName.length}")

}

**Salida:**

Tamaño del nick:7

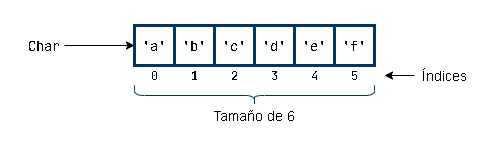
La expresión final de acceso ${playerName.length} es resuelta primero accediendo al entero 7 y luego se convierte a través del método toString().

Este tipo de plantillas te permitirá incluir todo tipo de expresiones como operaciones aritméticas, acceso a métodos, ejecución de funciones, etc.

## Crear Arrays

Un [arreglo](https://kotlinlang.org/docs/basic-types.html#arrays) es una estructura con valores de datos, que están almacenados de forma contigua en memoria. Todos los elementos son referenciados por un mismo nombre y tienen el mismo tipo de dato.

Los elementos estarán indexados tomando como base el 0 y el tamaño declarado del arreglo será fijo.



Kotlin usa la clase genérica Array<T> para representar arreglos. Crear instancias con un tipo parametrizado usa los siguientes métodos:

* arrayOf(vararg elements:T): recibe un argumento variables con elementos de tipo T y retorna el arreglo que los contiene.
* arrayOfNulls(size:Int): crea un arreglo de tamaño size con elementos de tipo T e inicializa los valores con null
* emptyArray(): crear un arreglo vacío con el tipo T

**Por ejemplo:**

Considera que necesitas almacenar el valor de los ingresos de 12 meses del usuario. En vez de crear 12 variables para cada mes (januaryIncome, febrauryIncome, etc.) optas por crear un arreglo con estos elementos:

val income = arrayOf<Double>(

0.5, 2.5, 4.0, 5.0,

4.5, 6.0, 7.6, 8.0,

5.0, 6.4, 4.0, 9.1

)

Al igual que cualquier declaración de instancias, puedes omitir el tipo parametrizado <Double> porque el compilador de Kotlin puede inferirlo.

### Usar Constructor Array()

Si deseas crear un array con un tamaño especifico y calcular todos sus elementos a partir de una función, entonces usa el constructor Array(size, init).

Por ejemplo, creemos un array que contenga los número del 1 al 10 con signo negativo:

fun main() {

val negativeNumbers = Array(10) { -(it + 1) }

println(negativeNumbers.joinToString())

}

**Salida:**

-1, -2, -3, -4, -5, -6, -7, -8, -9, -10

El segundo parámetro init de Array() es del [tipo función](https://www.develou.com/tipos-funcion-en-kotlin/) Int -> T, donde el parámetro entero es el índice y el cuerpo la expresión que calcula el valor del elemento. Por esto pasamos la función lambda { -(it + 1) }.

La función joinToString() permite crear un String legible para imprimir. Aunque también puedes usar contentToString().

println(negativeNumbers.contentToString())

// [-1, -2, -3, -4, -5, -6, -7, -8, -9, -10]

## Acceso A Elementos

La clase Array te provee operadores set() y get() para asignar y obtener valores en los índices de tus arreglos. Sin embargo, es preferible usar el operador de acceso indexado [ ] para referirte a los elementos.

fun main() {

val planets = arrayOfNulls<String>(8)

planets[0] = "Mercurio"

planets[1] = "Venus"

planets[2] = "Tierra"

planets[3] = "Júpiter"

planets[4] = "Saturno"

planets[5] = "Urano"

planets[6] = "Neptuno"

planets[7] = "Plutón"

println(planets.contentToString())

}

**Salida:**

[Mercurio, Venus, Tierra, Júpiter, Saturno, Urano, Neptuno, Plutón]

Como ves, la asignación a cada posición del array de planetas se realiza con los índices en corchetes. La tierra como está en la posición 3 sería asignada con [2].

## Recorrer Arrays

Una de las formas más convencionales de recorrer arreglos es a través del [bucle for](https://www.develou.com/for-en-kotlin/) a través del operador in junto a la propiedad de extensión indices. Esta contiene el rango válido de los índices del array.

Retomemos el ejemplo anterior de planetas. Si quisiéramos imprimir cada línea con la posición y el bucle para recorrerlo se vería así:

fun main() {

val planets = arrayOfNulls<String>(8)

planets[0] = "Mercurio"

planets[1] = "Venus"

planets[2] = "Tierra"

planets[3] = "Júpiter"

planets[4] = "Saturno"

planets[5] = "Urano"

planets[6] = "Neptuno"

planets[7] = "Plutón"

for(i in planets.indices){

println("${planets[i]} está en la posición ${i+1}")

}

}

**Salida:**

Mercurio está en la posición 1

Venus está en la posición 2

Tierra está en la posición 3

Júpiter está en la posición 4

Saturno está en la posición 5

Urano está en la posición 6

Neptuno está en la posición 7

Plutón está en la posición 8

Ahora, también es posible usar el atributo size del mismo array:

for (i in 0 until planets.size) {

println("${planets[i]} está en la posición ${i + 1}")

}

Otra opción es usar la función de extensión forEachIndexed(), la cual recibe como parámetros el índice y el valor, y te permite imprimir en el cuerpo:

planets.forEachIndexed { i, v ->

println("$v está en la posición ${i+1}")

}

# Propiedades lateinit En Kotlin

En este tutorial hablaremos sobre las propiedades lateinit en Kotlin y cómo pueden proveerte un mecanismo para inicialización tardía

* [El Modificador lateinit](https://www.develou.com/propiedades-lateinit-en-kotlin/#El_Modificador_lateinit)
* [Restricciones De Propiedades lateinit](https://www.develou.com/propiedades-lateinit-en-kotlin/#Restricciones_De_Propiedades_lateinit)
* [Ejemplo De Propiedades lateinit En Kotlin](https://www.develou.com/propiedades-lateinit-en-kotlin/#Ejemplo_De_Propiedades_lateinit_En_Kotlin)
  + [Solución Con Propiedad Anulable](https://www.develou.com/propiedades-lateinit-en-kotlin/#Solucion_Con_Propiedad_Anulable)
  + [Solución Con Propiedad lateinit](https://www.develou.com/propiedades-lateinit-en-kotlin/#Solucion_Con_Propiedad_lateinit)
* [Comprobar Inicialización Con isInitialized](https://www.develou.com/propiedades-lateinit-en-kotlin/#Comprobar_Inicializacion_Con_isInitialized)

## El Modificador lateinit

El modificador [lateinit](https://kotlinlang.org/docs/reference/properties.html" \l "late-initialized-properties-and-variables" \t "_blank) te permite inicializar una propiedad [no anulable](https://www.develou.com/anulabilidad-en-kotlin/) por fuera del constructor.

Este mecanismo te ayuda cuando deseas asignar el valor de una propiedad después y no deseas usar comprobaciones de nulos (expresiones if, operador de acceso seguro o aserciones) una vez inicializada.

lateinit var propiedad:String

## Restricciones De Propiedades lateinit

Ten en cuenta las siguientes restricciones a la hora de definir una propiedad lateinit:

* Deben ser propiedades mutables var (es evidente, ya que necesitas cambiar el valor fuera del constructor)
* Debe declararse en el cuerpo de la clase, no el [constructor primario](https://www.develou.com/constructores-en-kotlin/)
* No deben tener accesores (get y set) personalizados
* No pueden declararse con [tipos primitivos](https://www.develou.com/tipos-primitivos-en-kotlin/)
* No pueden ser anulables

## Ejemplo De Propiedades lateinit En Kotlin

Supongamos que hemos creado un pequeño framework para videojuegos compuesto de las clases Game y VideoController.

Game representa a la entidad maestra que controla las reglas del mundo y la creación de las abstraciones como VideoController.

Y VideoController es el controlador asociado a la creación de escenas y renderizado.

### Solución Con Propiedad Anulable

Proveeremos a Game de dos métodos, init() para inicializar el componente de video y createScene() para generar la escena preliminar.

class VideoController(var screenWitdh: Int, var screenHeight: Int)

class Game {

private var videoController: VideoController? = null

fun init(witdh:Int, height:Int) {

videoController = VideoController(witdh, height)

}

fun createScene() {

print("Resolución de ${videoController?.screenWitdh}x" +

"${videoController?.screenHeight}")

}

}

Como no podemos dejar a videoController sin valor hasta init(), entonces lo declaramos como anulable para asignarle null.

En createScene() usaremos el operador de acceso seguro .? para acceder a screenWitdh y screenHeight.

Como puedes notar, si en el futuro tuviésemos más usos del controlador de video dentro de Game, tendríamos grandes cantidades de comprobaciones de nulos.

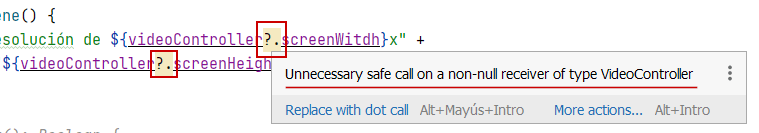
Resolvamoslo con el modificador lateinit.

### Solución Con Propiedad lateinit

Usa el modificador lateinit en la declaración de videoController.

private lateinit var videoController: VideoController

Inmediatemente IntelliJ IDEA te sugerirá que ya no es necesario el uso del operador ?.:



Así que retiralos:

fun createScene() {

print("Resolución de ${videoController.screenWitdh}x" +

"${videoController.screenHeight}")

}

Ahora prueba su funcionamiento creando una instancia de Game, inicializa los componentes y luego crea la escena:

fun main() {

val game = Game()

game.init(800,600)

game.createScene()

}

En la salida tendrás:

Resolución de 800x600

## Comprobar Inicialización Con isInitialized

Si intentas comentar la línea game.init(800,600) de la ejecución anterior, obtendrás una excepción del tipo UninitializedPropertyAccessException:

Exception in thread "main" kotlin.UninitializedPropertyAccessException:

lateinit property videoController has not been initialized

Y es algo que se esperaba, el valor no tiene un valor de inicialización, ya que no se respetó la secuencia de tu framework.

Para determinar si la propiedad fue o no inicializada, llama a isInitialized en la [referencia de la propiedad](https://www.develou.com/reflexion-en-kotlin/#referencias-de-propiedades).

Por ejemplo, en createScene() puedes verificar la propiedad isInitialized antes de acceder al controlador de video. Si no ha sido inicializada crea un controlador por defecto.

class Game {

//...

fun createScene() {

if (!this::videoController.isInitialized) {

videoController = VideoController(300, 300)

}

print(

"Resolución de ${videoController.screenWitdh}x" +

"${videoController.screenHeight}"

)

}

}

Al ejecutar el programa sin llamar a init() tendrás:

Resolución de 300x300

## ¿Qué Es Un Constructor?

Un constructor es una función especial que se usa para inicializar el contenido de las nuevas instancias de una [clase](https://www.develou.com/clases-en-kotlin/).

A diferencia de otros lenguajes, no usas la palabra new para crear el objeto, si no que llamas al constructor como una función normal.

class Player

val player1 = Player()

Como ves, usas el nombre de la clase para realizar la llamada.

Si no provees un constructor, entonces el compilador de Kotlin genera uno sin parámetros por defecto.

Los constructores de una clase pueden escribirse como **primarios** y **secundarios**. Observa de qué se trata.

## Constructor Primario

El constructor primario o principal, hace parte de la cabecera de la clase. Este recibe como argumentos, aquellos datos que necesitas explicitamente para inicializar las propiedades al crear el objeto.

En su sintaxis concisa, si declara con val o var antes del parámetro de un constructor primario, creas una propiedad automáticamente en la clase.

Usa la palabra reservada constructor luego del nombre de la clase:

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

Si no tienes anotaciones o modificadores de visibilidad puedes omitirla:

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

### Ejemplo De Constructor Primario

Supón que estamos creando un juego RPG y necesitamos modelar la clase para las armas que usarán los personajes de los jugadores. Se necesitarán los atributos para el daño de ataque y la velocidad de golpe.

La definición de la clase junto con constructor primario se vería así:

class Weapon(val attack: Int, val speed: Double)

Al crear la instancia de una nueva arma, llamamos al constructor como una función y pasamos los argumentos para las propiedades requeridas:

fun main() {

val weapon1 = Weapon(3, 0.5)

println("Arma 1 (ataque:${weapon1.attack}, velocidad: ${weapon1.speed})")

}

Al imprimir el contenido del objeto creado se reflejarían las propiedades establecidas para weapon1.

Arma 1 (ataque:3, velocidad: 0.5)

### Bloques De Inicialización

Es posible expandir la inicialización de las propiedades usando la sección init en la clase para el constructor primario.

En ella incluyes como cuerpo la lógica de asignación.

Por ejemplo, la clase Weapon puede ser escrita en forma expandida de la siguiente forma:

class Weapon(attack: Int, speed: Double) {

val attack: Int

val speed: Double

init {

this.attack = attack

this.speed = speed

}

}

Con la sintaxis completa, las propiedades son declaradas al interior de la clase y los parámetros pasan a solo ser parámetros de constructor sin val ni var.

Si los nombres de los parámetros del constructor se llaman igual a las propiedades, usa la expresión this para acceder a la propiedad de la instancia que intentas generar con el constructor.

### Visibilidad Del Constructor

Cambia la visibilidad de tus constructores en Kotlin usando los [modificadores](https://www.develou.com/visibilidad-en-kotlin/) existentes (public, internal, protected y private).

La sintaxis consiste en usar la palabra reservada constructor y anteponer el modificador.

class Item internal constructor(name: String)

## Constructores Secundarios

Si la lista de argumentos del constructor primario no satisface la creación de tu objeto en alguna circunstancia, entonces puedes crear un constructor secundario a la medida.

Su declaración se realiza a través de constructor al interior de la clase.

Si tienes un constructor primario es obligatorio usar la expresión this para delegarle los parámetros que requiera.

Luego escribe la lógica de inicialización en el bloque.

class Clase{

constructor(parametro1:Tipo, parametro2:Tipo /\*,..\*/): this(/\*parametros\*/){

/\* Cuerpo de constructor \*/

}

}

### Ejemplo De Constructor Primario Y Secundario

Se requiere crear una clase de contactos que contenga los atributos Id y Nombre. Se pueden crear contactos con tan solo el nombre y el Id autogenerado. O también con ambos valores.

Esto puedes solucionarlo así:

class Contact(var name: String) {

var id: String

init {

id = UUID.randomUUID().toString()

}

constructor(id:String, name: String) : this(name){

this.id = id

}

}

Declaras un constructor primario con el nombre y la autogeneración del Id en el bloque de inicialización.

Luego en el constructor secundario recibes ambos parámetros. Delegas hacia el constructor primario el nombre y el id lo reemplazas.

De esta forma puedes crear los dos tipos de contactos:

Contact("Erika")

Contact("C-1", "Mauricio")

### Ejemplo Con Dos Constructores Secundarios

Sin embargo, la delegación del ejemplo anterior no es necesaria, ya que no usas un valor autogenerado en el Id en la llamada del constructor con dos parámetros.

En cambio sí reescribes con dos constructores secundarios la clase, evitarías esa ejecución extra:

class Contact {

var id: String

var name: String

constructor(name: String) {

this.name = name

id = UUID.randomUUID().toString()

}

constructor(id: String, name: String){

this.id = id

this.name = name

}

}

Y al instanciar sería exactamente igual:

val contacto1 = Contact("Erika")

val contacto2 = Contact("C-1", "Mau

## Declaración De Una Clase

Conceptualmente, las clases son el componente medular del paradigma de [POO](https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos) (Programación Orientada A Objetos). En donde representan entidades o conceptos, que hacen parte de un modelo predefinido.

En Kotlin, una clase es una agrupación de propiedades (atributos) y funciones (métodos) miembro, que actúan como una plantilla para la instanciación de objetos.

La declaración de una clase se compone de:

* Nombre de la clase
* Cabecera
* Cuerpo

Para definirla, usa la palabra clave class seguido de su nombre.

class Supervisor {

/\*\*Cuerpo de clase\*\*/

}

La anterior declaración está en una forma reducida. Sin embargo, la declaración de la cabecera puede llevar más elementos como:

* Modificadores
* Declaración de parámetros
* Constructor primario
* Herencia
* Restricciones de tipo

En la gramática de Kotlin, esos elementos de la cabecera y el bloque de código para el cuerpo son opcionales. Por lo que incluso podrías reducir la declaración a:

class Supervisor

## Ejemplo De Declaración De Clase

A continuación verás un ejemplo simple donde puedes aplicar los conceptos básicos de POO y la definición de clases en Kotlin.

### Problema

Supón que estamos creando un juego, donde un avión se mueve en dos dimensiones a lo largo de un límite de [300,300]. Su posición inicial debe ser [0,0].

El objetivo es modelar dicho avión, junto a su información de coordenadas y permitir modificarlas dependiendo de la dirección en que se desplaza.

### Solución

Al analizar el enunciado, sabrás que la clase Avion (Plane) se estructura así

**Atributos**

* Posición en eje x
* Posición en eje y

**Métodos**

* **Movimientos hacia la izquierda** — Restar la unidad al eje x. Evitar el decremento menor a 0
* **Movimiento hacia la derecha** — Sumar la unidad al eje x. Evitar el incremento superior a 300
* **Movimiento hacia arriba** — Restar la unidad al eje y. Evitar decrementos menores a 0
* **Movimiento hacia abajo** — Sumar la unidad al eje y. Evitar incrementos mayores a 300

La siguiente es una de las posibles soluciones en código, al anterior análisis:

class Plane {

var x: Int = 0

private set

var y: Int = 0

private set

fun moveLeft() {

x -= if (x == 0) 0 else 1

}

fun moveRight() {

x += if (x == 300) 0 else 1

}

fun moveUp() {

y -= if (y == 0) 0 else 1

}

fun moveDown() {

y += if (y == 300) 0 else 1

}

}

Los atributos x e y se definen como propiedades mutables con [var](https://www.develou.com/variables-en-kotlin/" \t "_blank) y de tipo Int.

Cada método es una función miembro sin retorno que altera los ejes, a través de una [expresión if](https://www.develou.com/expresion-if-en-kotlin/) que bifurca las condiciones del diseño.

## Crear Instancias De Clases

Para crear una instancia de una clase, usa una llamada regular de funciones para el constructor.

val plane = Plane()

Puedes acceder a la propiedad miembro con la notación instancia.propiedad.

plane.x

Por ejemplo:

Simular el desplazamiento **Derecha > Derecha > Abajo > Arriba > Izquierda** de un avión. Luego imprimir su posición final.

fun main() {

val plane = Plane()

plane.moveRight()

plane.moveRight()

plane.moveDown()

plane.moveUp()

plane.moveLeft()

println("Posición actual del avión: (${plane.x},${plane.y})")

}

La salida en consola será:

Posición actual del avión: (1,0)

## ¿Qué Es Una Función?

Kotlin es un lenguaje que te permite aplicar estilos de [programación funcional](https://kotlinlang.org/docs/reference/faq.html#is-kotlin-an-object-oriented-language-or-a-functional-one) en tus aplicaciones. Por ello la **función** es una herramienta indispensable.

Una función es un conjunto de instrucciones que realizan una tarea específica, empaquetadas como unidad.

Esta puede ser llamada en el lugar donde sea que la necesites para ejecutar las sentencias.

## Sintaxis De Declaración

El siguiente ejemplo muestra la sintaxis de declaración de una función en Kotlin:

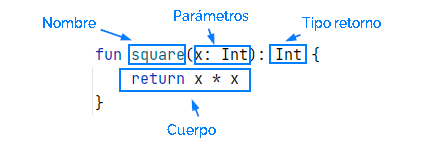
fun square(x: Int): Int {

return x \* x

}

La anterior función eleva al cuadrado un parámetro entero.

La declaración comienza con la palabra clave fun y se conforma de:



* **Nombre de la función** − Es el nombre que eliges para la función con el fin de esclarecer su propósito
* **Lista de parámetros** − Datos de entrada para la función. Defínelos como nombre:tipo y sepáralos por comas.
* **Tipo de retorno** − Tipo de dato de salida de la función. Una función con un cuerpo de bloque de código, siempre debe tener su tipo definido.
* **Cuerpo de la función** – Son todas las sentencias que realizan la tarea para llegar al resultado final de retorno. Usa la expresión return para devolver el valor.

## Llamar Una Función

La forma de invocar a una función es escribir el nombre en el punto del flujo donde requieras ejecutar las instrucciones y pasarle los argumentos con que se declaró.

Por ejemplo, llama la anterior función así:

fun square(x: Int): Int {

return x \* x

}

fun main() {

print(square(2))

}

Al ejecutar el programa la salida será 4.

### Funciones Con Cuerpo De Expresión

Si tu función retorna tan solo una expresión, puedes reducir su sintaxis a una función con cuerpo de expresión. Donde la expresión es ubicada luego del signo = en línea con la firma de la función.

**Por ejemplo:**

Determinar el resultado del modelo matemático f(x,y,z) = 3x-2y+z2 en el punto A=(1,0,5):

fun model(x: Int, y: Int, z: Int) = 3 \* x - 2 \* y + z \* z

fun main() {

println("f(1,0,5)= ${model(1, 0, 5)}")

}

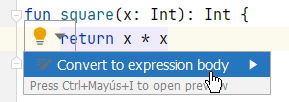
El modelo matemático comprende tres variables de entrada, por lo que esos serán el número de parámetros. En la misma línea, construimos la ecuación con [operadores aritméticos](https://www.develou.com/operadores-en-kotlin/) y así llamarla desde main() con los valores propuestos.

El resultado sería:

f(1,0,5)= 28

La declaración con cuerpo de expresión única, permite que el compilador infiera el tipo y así poder omitirlo de la función.

Por otro lado, IntelliJ IDEA puede hacer la **conversión a single-expression body** por ti, si despliegas las acciones emergentes en la función.



### Retorno Tipo Unit

Unit es la palabra reservada análoga a void en Java. Esto determina que una función no retorna valor.

**Ejemplo**: Crear una función que imprima si un carácter es una vocal o no.

fun isVowel(c: Char): Unit {

when (c) {

'a', 'e', 'i', 'o', 'u' -> println("$c es una vocal")

else -> println("$c no es una vocal")

}

}

fun main() {

isVowel('v')

}

La solución propuesta, usa una [expresión when](https://www.develou.com/expresion-when-en-kotlin/) para comprobar si el parámetro de tipo Char coincide con las vocales.

Sin embargo, como no deseamos retornar un valor significativo hacia el exterior, usamos como tipo de retorno Unit.

La salida del programa sería:

v no es una vocal

**Omitir tipo Unit** − Aunque el compilador de Kotlin no puede realizar inferencias de tipos con funciones con cuerpo de bloques de código, el tipo Unit es una excepción, es posible omitirlo de la declaración debido a su naturaleza.

La anterior función la puedes reescribir sin él:

fun isVowel2(c: Char) {

when (c) {

'a', 'e', 'i', 'o', 'u' -> println("$c es una vocal")

else -> println("$c no es una vocal")

}

}

## Lanzar Excepciones En Kotlin

Para lanzar lanzar [objetos de excepción](https://kotlinlang.org/docs/exceptions.html) en Kotlin, usa la expresión throw seguido de la instanciación del tipo. Al igual que en Java, la declaración del tipo debe tener como supertipo a la interfaz Throwable.

Considera un programa sencillo que lee por teclado un texto escrito por el usuario, donde se desea contar la cantidad de dígitos que el texto tenga.

En el caso de que el texto que introduzca sea nulo o un espacio en blanco, entonces lanzaremos una excepción que indique el incumplimiento de esta precondición.

fun main() {

print("Escribe el texto: ")

val userInput = readLine()

println("Cantidad de digitos: ${countDigits(userInput)}")

}

fun countDigits(userInput: String?): Int {

if (userInput.isNullOrBlank()) {

throw IllegalArgumentException("Entrada inválida, la palabra debe tener al menos un carácter")

}

return userInput.count(Char::isDigit)

}

**Salida al no incluir texto:**

Escribe el texto:

Exception in thread "main" java.lang.IllegalArgumentException: Entrada inválida, la palabra debe tener al menos un carácter

Como habíamos especificado, la construcción throw es una expresión. Esta característica te habilita su uso en otras expresiones.

Por ejemplo, countDigits() puede ser refactorizada como una [función con cuerpo de una sola expresión](https://www.develou.com/funciones-en-kotlin/#cuerpo-de-expresion) al indicar a throw como resultado de la [expresión if](https://www.develou.com/expresion-if-en-kotlin/).

fun countDigits(userInput: String?) =

if (userInput.isNullOrBlank())

throw IllegalArgumentException("Entrada inválida, la palabra debe tener al menos un carácter")

else

userInput.count(Char::isDigit)

## Manejar Excepciones En Kotlin

Al igual que en Java, Kotlin te permite capturar excepciones con la expresión try..catch..finally.

Pon en el bloque try el código que es propenso a lanzar excepciones y luego añade bloques catch que verifiquen la aplicabilidad de un subtipo de excepción.

El bloque finally se ejecuta luego de que se aplique o no algún bloque catch. Normalmente aquí liberas los recursos que has tomado del sistema y limpias referencias para evitar fugas de memoria.

Tomemos como ejemplo la conversión de un String a Double. Supongamos que deseamos crear una [función de extensión](https://www.develou.com/funciones-de-extension-en-kotlin/) que envuelva a la función String.toDouble() para que retorne un valor por defecto en caso de que se lance una excepción del tipo NumberFormatException.

fun main() {

println("5.3".toDoubleOrDefault(1.0))

println("5.".toDoubleOrDefault(1.0))

println(".3".toDoubleOrDefault(1.0))

println("dos".toDoubleOrDefault(1.0))

}

fun String.toDoubleOrDefault(defaultValue: Double): Double {

return try {

toDouble()

} catch (e: NumberFormatException) {

defaultValue

}

}

**Salida:**

5.3

5.0

0.3

1.0

Como ves, el bloque catch al atrapar un error de parsing del string debido a un formato incorrecto, retorna el valor por defecto que entra como argumento de toDoubleOrDefault().

Por supuesto esta función también puede ser escrita con cuerpo de expresión:

fun String.toDoubleOrDefault(defaultValue: Double) = try {

toDouble()

} catch (e: NumberFormatException) {

defaultValue

}

## Excepciones Marcadas

Kotlin no diferencia entre excepciones marcadas y no marcadas (checked y unchecked). Las funciones no deben especificar explícitamente que tipo de excepción lanzaran (como el uso de la cláusula throws en la firmas de métodos en Java), ni tampoco se te obliga a atrapar excepciones.

Esto quiere decir que cuando escribes sentencias que invoquen operaciones que lancen excepciones marcadas desde el SDK de Java, el compilador de Kotlin no te resaltará su uso sin bloque try..catch. Por ejemplo, como intentar abrir un archivo que no existe.

fun main() {

val stream = FileInputStream("archivo inválido")

}

Aunque este constructor hace explicito que lanzará una excepción marcada del tipo FileNotFoundException, no recibimos como exigencia controlarla. Obviamente al ejecutar esta sentencia tendremos el aviso esperado:

Exception in thread "main" java.io.FileNotFoundException: archivo inválido (El sistema no puede encontrar el archivo especificado)

Esto no significa que las ignoraremos, el manejo de errores debe seguir siendo parte de tu esquema. Significa que nos permite evitar la propagación de cambios en cadena entre diferentes firmas de funciones, cuando hay modificaciones de excepciones.

## Expresión continue Simple

La expresión continue un una [expresión de salto](https://kotlinlang.org/docs/reference/returns.html) que solo se permite dentro de los cuerpos de los bucles. Su funcionalidad es saltar las sentencias de una iteración y pasar a la siguiente.

Se especifica con la palabra reservada continue para saltar la iteración actual del bucle más interno, si existe anidamiento.

while (loopExpression) {

/\*...\*/

if (ifExpression) {

continue

}

/\* Sentencias saltadas\*/

}

**Ejemplo**:

Imprimir los múltiplos de 4 que existen en el conjunto del 1 al 20.

fun main() {

for (i in 1..20) {

if (i % 4 != 0) {

continue

}

println(i)

}

}

Para diferenciar las iteraciones innecesarias, asociadas a aquellos números no divisibles por cuatro, usaremos una [expresión if](https://www.develou.com/expresion-if-en-kotlin/) que ejecute un salto con continue.

4

8

12

16

20

De esta manera solo se ejecutará la impresión de las iteraciones restantes.

## Expresión continue Etiquetada

Al igual que [break](https://www.develou.com/break-en-kotlin/), la expresión continue puede ser usada en una forma etiquetada denotada con continue@etiqueta.

Donde etiqueta es el identificador del bucle etiquetado, al que deseas generar el salto de iteración.

Loop@ for(i in 1..5){

for (j in 1..7) {

if (ifExpression) {

continue@Loop

}

/\*Sentencias saltadas\*/

}

/\*Sentencias saltadas\*/

}

**Ejemplo:**

Imprimir las posibles parejas entre los caracteres de "Develou" y "Kotlin", solo si la suma de sus enteros equivalentes en [ASCII](https://es.wikipedia.org/wiki/ASCII) es menor o igual a 180.

fun main() {

Loop@ for (d in "Develou") {

for (k in "Kotlin") {

if (d.toByte() + k.toByte() > 180) {

continue@Loop

}

print("($k,$d)")

}

println()

}

}

La solución proveída usa un bucle for dentro de otro con el fin de comparar carácter a carácter.

El problema nos pide descartar las iteraciones irrelevantes de ambos bucles, es decir, cuando ambos caracteres convertidos a número, suman 180.

De esta forma, al anotar el bucle para los caracteres de "Develou" con Loop, podemos saltar tanto el print() de la pareja como el println() exterior.

El resultado en la salida sería:

(K,D)(o,D)(K,e)(K,e)

# La Expresión break En Kotlin

En este tutorial te mostraremos el uso de la expresión break en Kotlin, para que termines tus bucles en una situación precisa. También veras el uso de etiquetas con break y varios ejemplos.

## break Simple En Kotlin

La expresión break es una [expresión de salto](https://kotlinlang.org/docs/reference/returns.html) que se permite solo al interior de los bucles. Su objetivo es pasar el control al siguiente punto del programa luego del bucle, es decir, finalizar el bucle.

En su forma simple, se especifica la palabra reservada break, la cual provoca un salto de ruptura en la sentencia de bucle más interna del alcance actual.

**Ejemplo**: Imprimir la cantidad de caracteres de cada palabra que el usuario ingrese por teclado. Terminar la aplicación si escribe "salir".

fun main() {

while (true) {

print("Escribe una palabra:")

val word = readLine()!!

if (word == "salir") break

println("Caracteres:${word.length}")

}

}

El ejemplo muestra un [bucle while](https://www.develou.com/while-en-kotlin/) que se ejecuta infinitamente. Precisamente usamos una sentencia break simple para terminar con la condición de que sea el string «salir».

Veamos otro ejemplo para un **bucle dentro de otro**.

**Ejemplo**: Comprobar que todos los valores de una matriz sean iguales a uno. De lo contrario especificar invalidez en la primera aparición encontrada de la fila.

fun main() {

val matrix = arrayOf(

intArrayOf(1, 1, 1),

intArrayOf(0, 1, 0),

intArrayOf(1, 1, 0),

)

for (row in matrix.indices) {

for (col in matrix[row].indices) {

if (matrix[row][col] == 0) {

println("Matriz inválida en [$col,$row]")

break

}

}

}

}

El ejemplo muestra una matriz con tres posiciones inválidas: [0,1], [0,2] y [2,2]. Al recorrerla con un doble [bucle for](https://www.develou.com/for-en-kotlin/), se detecta la inconsistencia del primer cero y son impresos solo dos resultados.

Matriz inválida en [0,1]

Matriz inválida en [2,2]

La expresión break nos permitió terminar el bucle de filas prematuramente, asegurándonos de que nos es necesario perder tiempo en más comprobaciones.

## break Etiquetado En Kotlin

Una expresión etiquetada break se define con la sintaxis break@etiqueta, donde etiqueta es el identificador de una sentencia de bucle etiquetada.

etiqueta@ for (i in 1..5) {

/\*...\*/

}

El bucle se etiqueta con el símbolo @ como sufijo al identificador. Al declarar un break que apunte a la etiqueta, este se evaluará como un salto de ruptura de dicho bucle.

etiqueta@ for (i in 1..5) {

for (j in 1..7) {

if (...) break@etiqueta

}

}

**Por ejemplo**: Modifiquemos el ejercicio de la matriz en la sección anterior. Ahora, al encontrar el primer cero se terminará la búsqueda.

fun main() {

val matrix = arrayOf(

intArrayOf(1, 1, 1),

intArrayOf(0, 1, 0),

intArrayOf(1, 1, 0),

)

Loop@ for (row in matrix.indices) {

for (col in matrix[row].indices) {

if (matrix[row][col] == 0) {

println("Matriz inválida en [$col,$row]")

break@Loop

}

}

}

}

Hemos puesto en el bucle de las filas la etiqueta Loop@ e introducido el mecanismo de finalización con break@Loop. Al ejecutar la aplicación, la salida será:

Matriz inválida en [0,1]

## Sintaxis Del Bucle while

Las sentencias del [bucle while](https://kotlinlang.org/docs/reference/control-flow.html#while-loops) se componen de una expresión condicional (subtipo de kotlin.Boolean) y un cuerpo de cero o más sentencias.

while(condition){

/\*...\*/

}

Este bucle repetirá su cuerpo mientras la condición sea true o alguna expresión de salto sea evaluada.

**Ejemplo**: Realizar la sumatoria desde 1 hasta el número 10.

fun main() {

var sum = 0

var i = 1

while (i <= 10) {

sum += i++

}

println("Sumatoria:$sum")

}

El while anterior, usa como condición que el contador i sea menor o igual a 10. Mientras no lo sea, se acumulará en sum el valor del contador en cada iteración.

En la salida del programa tendrás:

Sumatoria:55

Adicionalmente, puedes reemplazar el bloque de código ({}) por sentencias sueltas.

while (i <= 10) sum += i++

## El Bucle do while

El bucle do while tiene en común el uso de la expresión de condición del while, sin embargo su sintaxis es diferente, ya que la condición es evaluada luego de ejecutar las sentencias del cuerpo.

do {

/\*..\*/

}while (condicion)

Esto te asegura que el cuerpo sea ejecutado al menos una vez sin importar que suceda, lo que hace al bucle do while perfecto para la anteposición de una acción.

**Ejemplo**: Preguntarle al usuario cuál es la capital de Colombia y presentarle 4 respuestas alternativas. Si responde mal, presentar de nuevo el mensaje hasta que lo logre.

fun main() {

do {

println("¿Cuál es la capital de Colombia?:")

println("a. La Paz")

println("b. Buenos Aires")

println("c. La Habana")

println("d. Bogotá")

print("Respuesta:")

val option = readLine()!!

val again = option != "d"

if(again){

println("¡Incorrecto!\n")

}else{

println("¡Felicitaciones!\n")

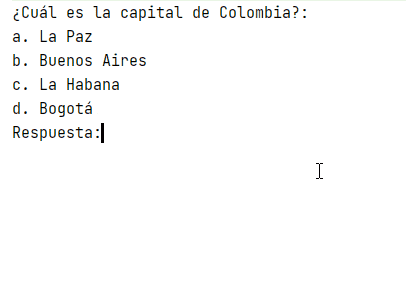
}

} while (again)

}

En el do..while anterior, se muestra como la primer pregunta viene sin tener que valuar la condición. Luego se repite el proceso las iteraciones necesarias hasta que again sea false.

La salida del programa se verías así:



# Bucle for En Kotlin

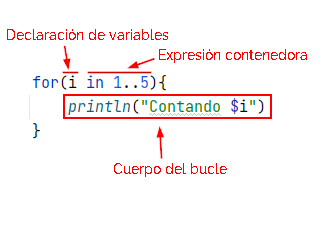
En este tutorial aprenderás sobre la sintaxis del bucle for en Kotlin para iterar sobre colecciones de elementos. Luego verás ejemplos recorrer estructuras de datos como rangos, arreglos y strings.

* [Sintaxis Del Bucle for](https://www.develou.com/for-en-kotlin/#Sintaxis_Del_Bucle_for)
* [Recorrer Un Rango](https://www.develou.com/for-en-kotlin/#Recorrer_Un_Rango)
* [Recorrer Un Array](https://www.develou.com/for-en-kotlin/#Recorrer_Un_Array)
* [Recorrer Un String](https://www.develou.com/for-en-kotlin/#Recorrer_Un_String)
* [¿Qué Sigue?](https://www.develou.com/for-en-kotlin/#Que_Sigue)

## Sintaxis Del Bucle for

El bucle [for](https://kotlinlang.org/docs/reference/control-flow.html" \l "for-loops" \t "_blank) en Kotlin se asemeja a las sentencias foreach de otros lenguajes, donde se provee un iterador (iterator()) para recorrer los elementos de datos estructurados.

La sentencia se compone de una **declaración de variables**, una **expresión contenedora**, compuesta por el operador in y los datos estructurados (arreglos, listas, etc.), y el **cuerpo del bucle**.



**Ejemplo**: Imprimir la cuenta del uno al cinco.

fun main(){

for(i in 1..5){

println("Contando $i")

}

}

Como ves, la variable declarada es i y la estructura de datos es un rango del 1 al 5. Lo que significa que el cuerpo se ejecutará cinco veces.

El valor de i es accesible desde el cuerpo, por lo que puedes [imprimir su contenido](https://www.develou.com/flujo-de-entrada-y-salida-en-kotlin/) con println():

Contando 1

Contando 2

Contando 3

Contando 4

Contando 5

Aunque usamos el cuerpo como un bloque de código, al ser una sola sentencia, puedes omitir las llaves.

for (i in 1..5) println("Contando $i")

A continuación, verás algunos ejemplos de iteraciones sobre colecciones.

## Recorrer Un Rango

El siguiente es un ejemplo que recorre el [rango](https://www.develou.com/rangos-en-kotlin/) de caracteres [‘a’,’f’]. Verás varios bucles for usando progresiones (por escribir) con el fin de manipular el ritmo y orden de las iteraciones.

fun main() {

// iteración regular

for (char in 'a'..'f') print(char)

// iteración con avance de 2

println()

for (char in 'a'..'f' step 2) print(char)

println()

// iteración en reversa

for (char in 'f' downTo 'a') print(char)

// iteración excluyendo el límite superior

println()

for (char in 'a' until 'f') print(char)

}

La salida del programa será:

abcdef

ace

fedcba

abcde

## Recorrer Un Array

Para iterar a lo largo de un arreglo con el bucle for, debemos usar como base los índices de sus elementos.

Esto significa, que usamos en la declaración de variable el índice y luego expresamos su existencia sobre la propiedad rango entero indices.

**Ejemplo**: Crear un arreglo con cinco nombres de campeones e imprimir su índice y valor.

fun main() {

val champions = arrayOf("Teemo", "Ahri", "Vayne", "Brand", "Kindred")

for (i in champions.indices) {

println("[$i, ${champions[i]}]")

}

}

La salida será:

[0, Teemo]

[1, Ahri]

[2, Vayne]

[3, Brand]

[4, Kindred]

Adicionalmente, Kotlin te permite reescribir la sentencia for anterior con el método withIndex(), el cual retorna en elementos IndexedValue, que contienen el índice y el valor.

Dichos valores pueden ser recibidos en un [declaración de desestructuración](https://www.develou.com/desestructuracion-en-kotlin/) de la forma (indice, valor). La cual es accesible desde el cuerpo del bucle.

fun main() {

val champions = arrayOf("Teemo", "Ahri", "Vayne", "Brand", "Kindred")

// Desestructuración de elementos de Array

for ((i, v) in champions.withIndex()) {

println("[$i, $v]")

}

}

## Recorrer Un String

También es posible iterar sobre un string con el bucle for. La sentencia interpretará la posición y valor de cada carácter.

**Ejemplo**: Imprimir el string "Develou" en forma vertical.

fun main(){

for(c in "Develou"){

println(c)

}

}

En la salida de la aplicación tendrás:

D

e

v

e

l

o

u

# La Expresión When En Kotlin

En este tutorial aprenderás a cómo usar la expresión when en Kotlin, con el fin de manipular el flujo de tus programas, a partir de varias entradas con sus respectivas condiciones.

* [when Como Sentencia](https://www.develou.com/expresion-when-en-kotlin/#when_Como_Sentencia)
* [Múltiples Valores En Una Entrada](https://www.develou.com/expresion-when-en-kotlin/#Multiples_Valores_En_Una_Entrada)
* [Usar Expresiones Como Condición](https://www.develou.com/expresion-when-en-kotlin/#Usar_Expresiones_Como_Condicion)
* [Rangos En When](https://www.develou.com/expresion-when-en-kotlin/#Rangos_En_When)
* [Comparar Tipos Con when](https://www.develou.com/expresion-when-en-kotlin/#Comparar_Tipos_Con_when)
* [Usar When Como If](https://www.develou.com/expresion-when-en-kotlin/#Usar_When_Como_If)
* [Declaración De Variable En El Sujeto](https://www.develou.com/expresion-when-en-kotlin/#Declaracion_De_Variable_En_El_Sujeto)
* [When Como Expresión](https://www.develou.com/expresion-when-en-kotlin/#When_Como_Expresion)
* [¿Qué Sigue?](https://www.develou.com/expresion-when-en-kotlin/#Que_Sigue)

## when Como Sentencia

La [expresión condicional when](https://kotlinlang.org/docs/reference/control-flow.html#when-expression) te permite comparar el valor de un argumento (sujeto) contra una lista de entradas.

Las entradas tienen condiciones asociadas al cuerpo que se ejecutará (condición -> cuerpo). Dichas condiciones pueden ser:

* Expresiones
* Comprobaciones de rangos
* Comprobaciones de tipos

Al haber coincidencia, se ejecuta el código en el cuerpo de la entrada (caso/rama) correspondiente.

**Ejemplo**: Tomar la entrada del usuario y determinar si es el carácter y para continuar, n para cerrar. De lo contrario asimilar como entrada inválida.

fun main() {

val input = 'y'

when (input) {

'y' -> print("Continuando...")

'n' -> print("Cerrando...")

else -> print("Entrada inválida")

}

}

El resultado de la evaluación anterior es:

Continuando...

Al igual que la [expresión if](https://www.develou.com/expresion-if-en-kotlin/), se usa else en caso de que ninguna de las entradas sea satisfecha.

when se podría considerar análoga a la sentencia switch de Java. Solo que when no requiere sentencias break para determinar la terminación de las ramas de condición.

## Múltiples Valores En Una Entrada

Si quieres comprobar múltiples valores en una entrada, pasa la lista separada por comas a la condición para el when.

**Ejemplo**: Determinar que turno le toca al operario según la balota que obtuvo en el sorteo.

fun main() {

val input = 2

when (input) {

1, 2, 3 -> print("Te toca turno nocturno")

4, 5, 6 -> print("Te toca turno diurno")

}

}

El resultado en consola es:

Te toca turno nocturno

## Usar Expresiones Como Condición

Como viste al inicio, las entradas reciben expresiones que puedan ser emparejadas con el sujeto.

Además de literales, también es posible usar expresiones variadas como conjunciones, comparaciones, operaciones, instanciaciones, etc.

**Ejemplo**: Determinar si la entrada coincide con los límites del tipo Char.

fun main() {

val input = 100

when (input) {

Char.MAX\_VALUE.toInt() -> print("Límite superior")

Char.MIN\_VALUE.toInt() -> print("Límite inferior")

else -> print("No es ninguno de los límites")

}

}

Como salida tendrás:

No es ninguno de los límites

## Rangos En When

Puedes usar rangos como condiciones de las entradas del when. Denota la coincidencia con el operador in o !in.

**Ejemplo**: Determinar si número que el usuario introduce por teclado pertenece a los rangos [0,49] o [50,99].

fun main() {

print("Ingresa el número:")

val input = readLine()!!.toInt()

when (input) {

in 1..49 -> print("$input pertenece a [1..49]")

in 50..99 -> print("$input pertence a [50..99]")

else -> print("Fuera de los rangos contemplados")

}

}

Si ingresas el número 40, tendrás:

40 pertenece a [1..49]

## Comparar Tipos Con when

when comprueba si una variable es de un tipo para determinar las acciones a seguir. La comparación es realizada con el operador is.

**Ejemplo**: Comprobar si la respuesta de un formulario hipotético se tomó como entera o cadena.

fun main() {

val formResponse: Any = 12

when (formResponse) {

is Int -> print("Respuesta Entera")

is String -> print("Respuesta String")

}

}

A diferencia de Java, no es necesario castear la variable para determinar si es el tipo correcto. A esto se le llama **Smart Cast** en Kotlin.

**Por ejemplo**: Usar el código anterior y multiplicar los resultados por 100, sin importar el tipo.

fun main() {

val formResponse: Any = "12"

when (formResponse) {

is Int -> {

val times = formResponse \* 100

print("Respuesta Entera:$times")

}

is String -> {

val times = formResponse.toInt() \* 100

print("Respuesta String:$times")

}

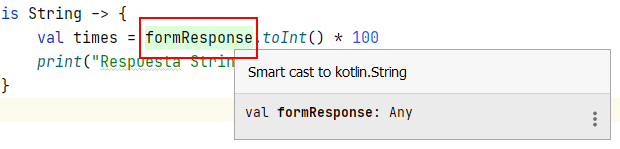
}

}

Al ejecutar tendrás:

Respuesta String:1200

Nota que IntelliJ IDEA sobresalta el valor interpretado por el Smart Cast.



## Usar When Como If

Por otro lado, si deseas comprobar expresiones booleanas sin tener un valor de comparación, entonces el when puede escribirse sin argumento.

fun main() {

val a = -5

when {

a > 0 -> print("Es positivo")

a == 0 -> print("Es cero")

else -> print("Es negativo")

}

}

## Declaración De Variable En El Sujeto

Si deseas pasar una variable en el argumento del when, debes crear una expresión de asignación a una variable de solo lectura.

**Ejemplo**: Calcular el daño recibido por el jugador dependiendo de la suma de su suerte y bonos adicionales.

fun main() {

val playerLucky = 0.2

val bonus = 0.3

when (val damage: Double = playerLucky + bonus) {

in 0.0..0.3 -> print("Daño recibido:${damage \* 10}")

in 0.3..0.6 -> print("Daño recibido:${damage \* 20}")

in 0.6..1.0 -> print("Daño recibido:${damage \* 30}")

}

}

Como ves, la variable damage es declarada e inicializada en el sujeto del when, lo que permite usarla en su bloque de estructura.

El programa imprime el siguiente valor:

Daño recibido:10.0

## When Como Expresión

when puede ser usado como expresión en retornos y asignaciones de variables como la expresión if.

Es obligatorio usar la rama else, a menos que se garantice que las ramas definidas cubren todas las posibles opciones (Exhaustive when expression).

**Por ejemplo**: Imprimir el texto equivalente al número de una calificación que va del 1 al 5.

fun main() {

val markNumber = 4

val markText = when (markNumber) {

1 -> "Insuficiente"

2 -> "Deficiente"

3 -> "Aceptable"

4 -> "Sobresaliente"

5 -> "Excelente"

else -> "No permitido"

}

print("La calificación es: $markText")

}

Al número 4 le correspondería el resultado de:

La calificación es: Sobresali

# La Expresión if En Kotlin

Este tutorial te ayudará a usar la expresión if en Kotlin para que puedas bifurcar el flujo de tus aplicaciones, al evaluar condiciones.

* [If Como Sentencia](https://www.develou.com/expresion-if-en-kotlin/#If_Como_Sentencia)
* [If Como Expresión](https://www.develou.com/expresion-if-en-kotlin/#If_Como_Expresion)
  + [If Y Else En Expresión](https://www.develou.com/expresion-if-en-kotlin/#If_Y_Else_En_Expresion)
  + [Usar Ramas Como Bloques](https://www.develou.com/expresion-if-en-kotlin/#Usar_Ramas_Como_Bloques)
* [¿Qué Sigue?](https://www.develou.com/expresion-if-en-kotlin/#Que_Sigue)

## If Como Sentencia

La [estructura de control if](https://kotlinlang.org/docs/reference/control-flow.html#if-expression), te ayuda a conducir el flujo de tu aplicación, basada en la evaluación de una expresión que produzca un resultado booleano.

if(expresion){

// Acciones si se cumple

}else if (expresion2){

// Acciones si se cumple

}else{

// Acciones si nada se cumple

}

Procesa el resultado adverso de la expresión con else y agrega más evaluaciones alternas con else if.

**Ejemplo**: Determinar si el jugador puede aprender una habilidad que requiere el nivel 45.

fun main() {

val level = 12

if (level >= 45) {

println("Aprendiste Acrobacia")

} else {

println("Se requiere nivel 45. Nivel actual $level")

}

}

El resultado sería:

Se requiere nivel 45. Nivel actual 12

El flujo de ejecución ejecuto las sentencias del cuerpo de else, porque obviamente 12 es menor a 45.

## If Como Expresión

A diferencia de Java, el condicional if es una expresión primaria en Kotlin.

Esto significa que al evaluar una condición produce un valor, que puede ser usado en el retorno de una función o en la asignación a una variable.

Veamos el contraste con un ejemplo.

**Ejemplo**: Evaluar la vida del enemigo, si es menor o igual a 15, establecer el ataque de la carta en 9, de lo contrario en 6.

fun main() {

val enemyHeroHealth = 16

val cardAttack: Int

if (enemyHeroHealth <= 15) {

cardAttack = 9

} else {

cardAttack = 6

}

println("Ataque:$cardAttack")

}

El anterior uso del if como sentencia puede ser reescrito con la siguiente expresión:

val cardAttack = if (enemyHeroHealth <= 15) 9 else 6

De esta forma mejoras la legibilidad para un if cuyo propósito es muy simple para la sintaxis tradicional.

fun main() {

val enemyHeroHealth = 16

val cardAttack = if (enemyHeroHealth <= 15) 9 else 6

println("Ataque:$cardAttack")

}

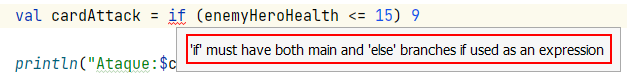
El resultado sería:

Ataque:6

### If Y Else En Expresión

Es obligatorio usar else cuando usas if como expresión. De lo contrario el compilador te marcará el error.

'if' must have both main and 'else' branches if used as an expression



### Usar Ramas Como Bloques

La rama if y else pueden ser convertidas en bloques si usas llaves. Esto quiere decir que puedes tener múltiples líneas en el cuerpo de cada una.

La última expresión debe ser el valor resultante del bloque.

Por ejemplo, modifiquemos el código anterior para agregar [impresiones](https://www.develou.com/flujo-de-entrada-y-salida-en-kotlin/) correspondientes al valor establecido.

fun main() {

val enemyHeroHealth = 16

val cardAttack = if (enemyHeroHealth <= 15) {

println("Grrrrrrrr!!")

9

} else {

println("...")

6

}

println("Ataque:$cardAttack")

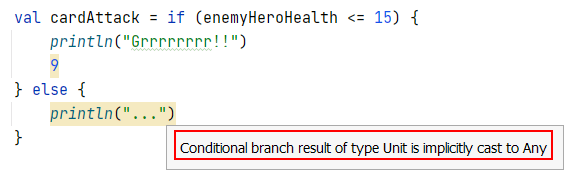
}

La salida del programa será:

...

Ataque:6

Si remueves el valor 6 de la rama else, verás que el compilador infiere el tipo como Any, ya que println() retorna en Unit, y se realiza un casting.



Al ejecutar la aplicación así, la variable cardAttack habrá tomado una referencia Unit.

...

Ataque:kotlin.Unit

Si quieres asegurar que el valor sea siempre entero, entonces especifica Int en la declaración de la variable.

De esta forma el compilador no inferirá otros tipos y se te presentará el error guía.

Texto

Descripción generada automáticamente

# Operadores En Kotlin

En este tutorial, aprenderás acerca de los [operadores en Kotlin](https://kotlinlang.org/docs/reference/keyword-reference.html#operators-and-special-symbols) y cómo usarlos para crear expresiones con [variables](https://www.develou.com/variables-en-kotlin/).

* [Operadores De Signo](https://www.develou.com/operadores-en-kotlin/#Operadores_De_Signo)
* [Operadores Aritméticos](https://www.develou.com/operadores-en-kotlin/#Operadores_Aritmeticos)
* [Operadores De Asignación Compuesta](https://www.develou.com/operadores-en-kotlin/#Operadores_De_Asignacion_Compuesta)
* [Operadores De Incremento Y Decremento](https://www.develou.com/operadores-en-kotlin/#Operadores_De_Incremento_Y_Decremento)
* [Operadores Relacionales](https://www.develou.com/operadores-en-kotlin/#Operadores_Relacionales)
* [Operadores Lógicos](https://www.develou.com/operadores-en-kotlin/#Operadores_Logicos)
* [Operadores A Nivel De Bits](https://www.develou.com/operadores-en-kotlin/#Operadores_A_Nivel_De_Bits)
* [¿Qué Sigue?](https://www.develou.com/operadores-en-kotlin/#Que_Sigue)

## Operadores De Signo

Comenzamos con los operadores unarios que representan la propiedad de un número de ser negativo, representado por -, o positivo, representado por +.

| **Operador** | **Expresión** | **Función equivalente** |
| --- | --- | --- |
| + | +a | a.unaryPlus() |
| - | -a | a.unaryMinus() |

Ejemplo: Negar una variable con el valor de -3

fun main() {

val a = -3

println("a=${-a}")

}

Por ley de signos la salida será:

a=3

## Operadores Aritméticos

Te permiten expresar operaciones aritméticas entre dos operandos.

| **Operador** | **Operación** | **Expresión** | **Función Equivalente** |
| --- | --- | --- | --- |
| + | Suma | a+b | a.plus(b) |
| - | Resta | a-b | a.minus(b) |
| \* | Multiplicación | a\*b | a.times(b) |
| / | División | a/b | a.div(b) |
| % | Residuo | a%b | a.rem(b) |

**Ejemplo**: Imprimir el resultado de todas las operaciones aritméticas con los números 10 y 20.

fun main() {

val a = 10

val b = 20

println("($a + $b)= ${a + b}")

println("($a - $b)= ${a - b}")

println("($a x $b)= ${a \* b}")

println("($a / $b)= ${a / b}")

println("($a % $b)= ${a % b}")

}

El resultado sería:

(10 + 20)= 30

(10 - 20)= -10

(10 x 20)= 200

(10 / 20)= 0

(10 % 20)= 10

Como ves, **al dividir dos enteros**, se conserva la parte entera como resultado y se descartan lo decimales.

## Operadores De Asignación Compuesta

Estos operadores son la combinación entre el operador de asignación y los operadores aritméticos, con el fin de usar como operando la variable de resultado.

**Por ejemplo**: si quieres acumular el valor de b en a, la abreviación para el operador compuesto de suma es la siguiente:

a += b // a = a + b

De esta manera, la siguiente tabla resume a los demás:

| **Operador** | **Expresión simplificada** | **Expresión Completa** | **Función Equivalente** |
| --- | --- | --- | --- |
| += | 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.remAssign(b) |

Ejemplo: Aplicar todos los operadores de asignación compuesta con un operando de 5 sobre un valor de 100. Al final imprimir el resultado.

fun main() {

var a = 100

val b = 5

a += b

println("+= $a")

a -= b

println("-= $a")

a \*= b

println("\*= $a")

a /= b

println("/= $a")

a %= b

println("%= $a")

}

La salida de la ejecución será:

+= 105

-= 100

\*= 500

/= 100

%= 0

## Operadores De Incremento Y Decremento

El operador de incremento, representado por dos signos de suma (++), incrementa en la unidad al operando.

monstersKilled++

Análogamente, el operador de decremento, doble signo menos (--), disminuye en la unidad al operando.

hearts--

El incremento depende de si el operador es prefijo o sufijo del operando:

* **Prefijo**: Se realiza el incremento sobre la variable y luego es usada en la expresión que la contiene.
* **Sufijo**: Se usa la variable en la expresión y luego si se aplica el incremento.

Ejemplo: Declarar una variable entera con el valor de 2 e imprimir su valor luego de incrementar con prefijo y decrementar con sufijo.

fun main() {

var a = 2

println("De $a a ${++a}")

println("De $a a ${a--}")

println("Valor final > $a")

}

El resultado será:

De 2 a 3

De 3 a 3

Valor final > 2

Aunque el valor es disminuido en la segunda impresión, este no se hace efectivo sino hasta que se termina la línea. Por eso el resultado final es 2 en la tercera impresión.

## Operadores Relacionales

Los operadores relacionales te permiten verificar enunciados de igualdad y desigualdad entre dos valores. El tipo de dato resultante de la expresión es Boolean, indicando la veracidad del enunciado expresado.

Aquí hay una la tabla de estos operadores:

| **Operador** | **Enunciado** | **Expresión** | **Función Equivalente** |
| --- | --- | --- | --- |
| == | a es igual a b | a==b | a?.equals(b) ?: (b === null) |
| != | a es diferente de b | a!=b | !(a?.equals(b) ?: (b === null)) |
| < | a es menor que b | a<b | a.compareTo(b)<0 |
| > | a es mayor que b | a>b | a.compareTo(b)>0 |
| <= | a es menor ó igual que b | a<=b | a.compareTo(b)<=0 |
| >= | a es mayor o igual que b | a>=b | a.compareTo(b)>=0 |

**Ejemplo**: Usar los operadores relacionales entre los números 17 y 20.

fun main() {

val a = 17

val b = 20

println("$a es igual a $b: ${a == b}")

println("$a es diferente a $b: ${a != b}")

println("$a es menor que $b: ${a < b}")

println("$a es mayor que $b: ${a > b}")

println("$a es menor o igual que $b: ${a <= b}")

println("$a es mayor o igual que $b: ${a >= b}")

}

La salida será:

17 es igual a 20: false

17 es diferente a 20: true

17 es menor que 20: true

17 es mayor que 20: false

17 es menor o igual que 20: true

17 es mayor o igual que 20: false

## Operadores Lógicos

Los operadores lógicos te permiten crear expresiones de lógica proposicional como lo son conjunción, disyunción y negación.

| **Operador** | **Descripción** | **Expresión** |
| --- | --- | --- |
| && | Conjunción (and): el resultado es true si a y b son true | a&&b |
| || | Disyunción (or): el resultado es true si a o b son true | a||b |
| ! | Negación (not): el resultado es false si a es true, o viceversa | !a |

**Ejemplo**: Usar las proposiciones «5 mayor que 0» y «5 es par» para comprobar el funcionamiento de los operadores lógicos.

fun main() {

val input = 5

var res: Boolean

val greaterThanZero = input > 0

val isEven = input % 2 == 0

res = greaterThanZero && isEven

println("Es mayor que cero y par:$res")

res = greaterThanZero || isEven

println("Es mayor que cero o par:$res")

res = greaterThanZero && !isEven

println("Es mayor que cero e impar:$res")

}

## Operadores A Nivel De Bits

Kotlin provee funciones para los tipos primitivos enteros, que actúan como operadores a nivel de bits.

| **Función** | **Descripción** | **Expresión** |
| --- | --- | --- |
| and() | **and** bit a bit | a and b |
| or() | **or** bit a bit | a or b |
| xor() | **xor** bit a bit | a xor b |
| inv() | **not** bit a bit | a.inv() |
| shl() | Desplazamiento de bits a la izquierda | a shl b |
| shr() | Desplazamiento de bits a la derecha | a shr b |
| ushr() | Desplazamiento de bits a la derecha sin signo | a ushr b |

**Ejemplo**: Operar a nivel de bits los números enteros 5 y 7.

fun main() {

val a = 5

val b = 7

println("a and b: ${a and b}")

println("a or b: ${a or b}")

println("a xor b: ${a xor b}")

println("a.inv(): ${a.inv()}")

println("a shl b: ${a shl b}")

println("a shr b: ${a shr b}")

println("a ushr b: ${a ushr b}")

}

El resultado de las operaciones a nivel de bits sería:

a and b: 5

a or b: 7

a xor b: 2

a.inv(): -6

a shl b: 640

a shr b: 0

a ushr b: 0

## Casting Inteligente En Kotlin

El compilador de Kotlin tiene la capacidad de realizar en una sola operación la verificación de tipo y casting. Permitiéndote usar inmediatamente el resultado del casting en el contexto aplicado.

A esta capacidad se le conoce como [**Smart cast**](https://kotlinlang.org/docs/typecasts.html#smart-casts) o casting inteligente. Cuando usas el operador is, el compilador castea explícitamente el valor y hace consciente a las siguientes sentencias del resultado.

Por ejemplo:

Considera la declaración del tipo Movable, el cual permite responder a peticiones a los objetos que se muevan. Esta interfaz contendrá una operación para calcular la rapidez del objeto.

interface Movable {

fun calculateSpeed(): Double

}

Ahora crearemos tres clases para representar bicicletas, autos y personas. Cada uno tendrá una propiedad base que permitirá deducir la rapidez. En las bicicletas influirá el tamaño de las llantas, para los autos los caballos de fuerza del motor y en las personas su peso corporal:

class Bike(val tireSize: Int) : Movable {

override fun calculateSpeed() = tireSize \* 5.0

}

class Car(val horsepower: Int) : Movable {

override fun calculateSpeed() = horsepower \* 0.62

}

class Person(val weigth: Double) : Movable {

override fun calculateSpeed() = weigth \* 0.72

}

Comprobemos el uso del smart cast creando una función que verifique los datos de un tipo Movable e imprima la rapidez que alcanza y su propiedad base:

fun main() {

verify(Bike(22))

verify(Car(300))

verify(Person(60.0))

}

fun verify(movable: Movable) {

if (movable is Bike) {

print("Bicicleta corre a: ${movable.calculateSpeed()}km/h Tamaño de llanta: ${movable.tireSize}mm")

return

}

if (movable is Car) {

print("Auto corre a: ${movable.calculateSpeed()}km/h Caballos de fuerza: ${movable.horsepower}hp")

return

}

if (movable is Person) {

print("Persona corre a: ${movable.calculateSpeed()}km/h Peso: ${movable.weigth}kg")

return

}

}

**Salida:**

Bicicleta corre a: 110.0km/h Tamaño de llanta: 22mm

Auto corre a: 186.0km/h Caballos de fuerza: 300hp

Persona corre a: 43.199999999999996km/h Peso: 60.0kg

Como ves, el parámetro movable en ningún momento se castea, ya que el uso del operador is en cada expresión if lo hace automáticamente. Por esta razón podemos acceder a las variables particulares de cada clase en el bloque de código.

### Smart Cast En Expresión When

El compilador extiende esta capacidad a la [expresión when](https://www.develou.com/expresion-when-en-kotlin/) cuando realizas verificaciones de tipo en sus ramas. Por ejemplo, si refactorizamos la función anterior verás:

fun verify(movable: Movable){

when(movable){

is Bike -> println("Bicicleta corre a: ${movable.calculateSpeed()}km/h" +

" Tamaño de llanta: ${movable.tireSize}mm")

is Car -> println("Auto corre a: ${movable.calculateSpeed()}km/h " +

"Caballos de fuerza: ${movable.horsepower}hp")

is Person->println("Persona corre a: ${movable.calculateSpeed()}km/h " +

"Peso: ${movable.weigth}kg")

else -> throw IllegalArgumentException("Tipo desconocido")

}

}

El uso de is en cada caso del when realiza el casting explícitamente para que el bloque de la rama use directamente a movable como el tipo verificado.

### Reglas Para Smart Cast

Cuando el compilador encuentra una verificación de tipo e intenta realizar el casting, se asegura de que el valor no cambiará en el momento de usarse. Por lo cual, este proceso funciona solo en:

* **Variables locales de solo lectura**: Siempre, excepto las propiedades locales delegadas
* **Propiedades de solo lectura**: Solo si la propiedad es private, protected o si la verificación de tipo se realiza en el mismo módulo. No se aplica si la propiedad está marcada como open o si tiene getter personalizado.
* **Variables mutables locales**: Solo si la variable no ha sido modificada entre la verificación y el uso, si no ha sido modificada por una función lambda y si no es una propiedad local delegada
* **Propiedades mutables**: Nunca

## Operador De Casting Inseguro

Para realizar un casting se usa el operador de infijo as. Con el expresas el enunciado «tomar operador izquierdo como tipo derecho».

Por ejemplo:

fun main() {

showBike(Bike(30))

}

fun showBike(movable: Movable){

val bike = movable as Bike

print(bike.calculateSpeed())

}

**Salida:**

150.0

¿Que pasaría si usaramos la versión anulable de Movable?

fun main() {

showBike(null)

}

fun showBike(movable: Movable?){

val bike = movable as Bike

print(bike.calculateSpeed())

}

**Salida:**

Exception in thread "main" java.lang.NullPointerException: null cannot be cast to non-null type Bike

Aunque el compilador no marca un aviso al usar as con un tipo anulable, en la ejecución se lanzará un NPE al pasar null en showBike().

La forma de evitar esta excepción, es declarar a bike como Bike?:

fun main() {

showBike(null)

}

fun showBike(movable: Movable?){

val bike = movable as Bike?

print(bike?.calculateSpeed())

}

**Salida:**

null

## Operador De Casting Seguro

El casting entre tipos puede ser cubierto con el operador de acceso seguro junto al operador de casting, es decir, la construcción as?. Este evita excepciones de casting si los tipos son diferentes.

Toma como ejemplo la expresión movable as? Bike, si movable es de tipo Bike, entonces se consume el casting movable as Bike.

Si movable no es tipo Bike, entonces el resultado será null. Esto podemos verlo cuando pasamos una instancia de Person() en showBike():

fun main() {

showBike(Person(54.0))

}

fun showBike(movable: Movable){

val bike = movable as? Bike

print(bike?.calculateSpeed())

}

El resultado será null al usar el operador de casting seguro as?, lo que nos evita una excepción del tipo ClassCastException.

## Conversión Explícita

Las [conversiones explicitas](https://kotlinlang.org/docs/reference/basic-types.html#explicit-conversions) son exigidas por el compilador cuando intentas asignar un tipo de dato numérico que difiere de sus familiares.

**Por ejemplo**: Intentar inicializar una variable Int con un tipo Short:

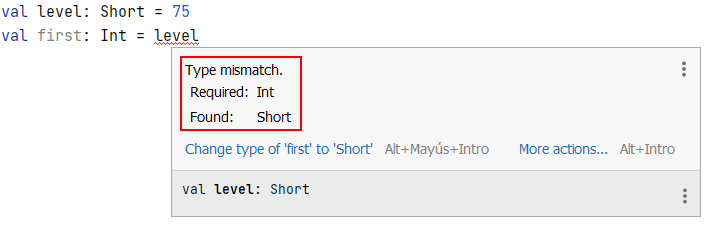
val level: Short = 75

val first: Int = level

Del ejemplo anterior esperarías una conversión implícita de tipo, ya que Short es un entero más pequeño que Int. Sin embargo no es el caso.

En Kotlin los tipos pequeños no son subtipos de los grandes. Por ello el compilador de arrojará el siguiente error.

Type mismatch: inferred type is Short but Int was expected

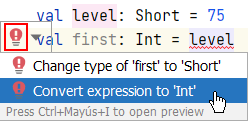


Para realizar la conversión explicita en Kotlin del escenario anterior, usa el método toInt().

val level: Short = 75

val first: Int = level.toInt()

Adicionalmente, IntelliJ IDEA te ayuda a encontrar la solución si haces click en la bombilla roja, y seleccionas **Convert expression to ‘Int’**.



## Métodos De Conversión

Si exploras la clase Number de Kotlin, verás que cada tipo primitivo que herede de ella, tiene los siguientes métodos de conversión.

* toDouble()
* toFloat()
* toLong()
* toInt()
* toChar()
* toShort()
* toByte()

Debido a la diferencia de tamaño de bits en cada tipo, las conversiones aplicaran **truncados** o **redondeos** necesarios, cuando el tamaño difiera.

**Ejemplo**: Imprimir el resultado de la conversión explícita de un número entero que exceda el rango de Short:

fun main() {

// Redondeo y truncado en conversiones explícitas

val measure = 4005215

println("toShort():${measure.toShort()}")

}

La salida de la aplicación será:

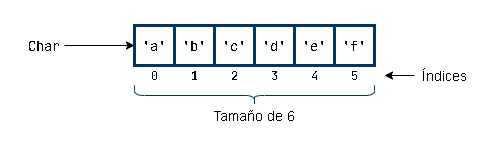
toShort():7519

Al no pertenecer al rango Short.MIN\_VALUE y Short.MAX\_VALUE, se usa el truncado con los 16 bits menos significativos del entero.

## Crear Arrays

Un [arreglo](https://kotlinlang.org/docs/basic-types.html#arrays) es una estructura con valores de datos, que están almacenados de forma contigua en memoria. Todos los elementos son referenciados por un mismo nombre y tienen el mismo tipo de dato.

Los elementos estarán indexados tomando como base el 0 y el tamaño declarado del arreglo será fijo.



Kotlin usa la clase genérica Array<T> para representar arreglos. Crear instancias con un tipo parametrizado usa los siguientes métodos:

* arrayOf(vararg elements:T): recibe un argumento variables con elementos de tipo T y retorna el arreglo que los contiene.
* arrayOfNulls(size:Int): crea un arreglo de tamaño size con elementos de tipo T e inicializa los valores con null
* emptyArray(): crear un arreglo vacío con el tipo T

**Por ejemplo:**

Considera que necesitas almacenar el valor de los ingresos de 12 meses del usuario. En vez de crear 12 variables para cada mes (januaryIncome, febrauryIncome, etc.) optas por crear un arreglo con estos elementos:

val income = arrayOf<Double>(

0.5, 2.5, 4.0, 5.0,

4.5, 6.0, 7.6, 8.0,

5.0, 6.4, 4.0, 9.1

)

Al igual que cualquier declaración de instancias, puedes omitir el tipo parametrizado <Double> porque el compilador de Kotlin puede inferirlo.

### Usar Constructor Array()

Si deseas crear un array con un tamaño especifico y calcular todos sus elementos a partir de una función, entonces usa el constructor Array(size, init).

Por ejemplo, creemos un array que contenga los número del 1 al 10 con signo negativo:

fun main() {

val negativeNumbers = Array(10) { -(it + 1) }

println(negativeNumbers.joinToString())

}

**Salida:**

-1, -2, -3, -4, -5, -6, -7, -8, -9, -10

El segundo parámetro init de Array() es del [tipo función](https://www.develou.com/tipos-funcion-en-kotlin/) Int -> T, donde el parámetro entero es el índice y el cuerpo la expresión que calcula el valor del elemento. Por esto pasamos la función lambda { -(it + 1) }.

La función joinToString() permite crear un String legible para imprimir. Aunque también puedes usar contentToString().

println(negativeNumbers.contentToString())

// [-1, -2, -3, -4, -5, -6, -7, -8, -9, -10]

## Acceso A Elementos

La clase Array te provee operadores set() y get() para asignar y obtener valores en los índices de tus arreglos. Sin embargo, es preferible usar el operador de acceso indexado [ ] para referirte a los elementos.

fun main() {

val planets = arrayOfNulls<String>(8)

planets[0] = "Mercurio"

planets[1] = "Venus"

planets[2] = "Tierra"

planets[3] = "Júpiter"

planets[4] = "Saturno"

planets[5] = "Urano"

planets[6] = "Neptuno"

planets[7] = "Plutón"

println(planets.contentToString())

}

**Salida:**

[Mercurio, Venus, Tierra, Júpiter, Saturno, Urano, Neptuno, Plutón]

Como ves, la asignación a cada posición del array de planetas se realiza con los índices en corchetes. La tierra como está en la posición 3 sería asignada con [2].

## Recorrer Arrays

Una de las formas más convencionales de recorrer arreglos es a través del [bucle for](https://www.develou.com/for-en-kotlin/) a través del operador in junto a la propiedad de extensión indices. Esta contiene el rango válido de los índices del array.

Retomemos el ejemplo anterior de planetas. Si quisiéramos imprimir cada línea con la posición y el bucle para recorrerlo se vería así:

fun main() {

val planets = arrayOfNulls<String>(8)

planets[0] = "Mercurio"

planets[1] = "Venus"

planets[2] = "Tierra"

planets[3] = "Júpiter"

planets[4] = "Saturno"

planets[5] = "Urano"

planets[6] = "Neptuno"

planets[7] = "Plutón"

for(i in planets.indices){

println("${planets[i]} está en la posición ${i+1}")

}

}

**Salida:**

Mercurio está en la posición 1

Venus está en la posición 2

Tierra está en la posición 3

Júpiter está en la posición 4

Saturno está en la posición 5

Urano está en la posición 6

Neptuno está en la posición 7

Plutón está en la posición 8

Ahora, también es posible usar el atributo size del mismo array:

for (i in 0 until planets.size) {

println("${planets[i]} está en la posición ${i + 1}")

}

Otra opción es usar la función de extensión forEachIndexed(), la cual recibe como parámetros el índice y el valor, y te permite imprimir en el cuerpo:

planets.forEachIndexed { i, v ->

println("$v está en la posición ${i+1}")

}

<https://www.develou.com/guia-de-kotlin/>

<https://play.kotlinlang.org/#eyJ2ZXJzaW9uIjoiMS43LjEwIiwicGxhdGZvcm0iOiJqYXZhIiwiYXJncyI6IiIsIm5vbmVNYXJrZXJzIjp0cnVlLCJ0aGVtZSI6ImlkZWEiLCJjb2RlIjoiLyoqXG4gKiBZb3UgY2FuIGVkaXQsIHJ1biwgYW5kIHNoYXJlIHRoaXMgY29kZS5cbiAqIHBsYXkua290bGlubGFuZy5vcmdcbiAqL1xuZnVuIG1haW4oKSB7XG4gICAgdmFsIHhwb3M6IEludCA9IDJcbiAgICBwcmludGxuKHhwb3MpXG59In0=>