

# Kotlin

[Kotlin](#) es un lenguaje de programación que aparece en 2016, y fue diseñado por la empresa JetBrains<sup>1</sup> y desarrolladores de código abierto, como un lenguaje totalmente interoperable con Java, de ahí que se diga que Kotlin es una evolución de Java.

Kotlin se usa principalmente para desarrollar aplicaciones backend, crear aplicaciones nativas Android ya que en 2019 Google decidió que fuera el lenguaje de referencia, también se utiliza para realizar librerías multiplataforma, aplicaciones web, aplicaciones de escritorio, e incluso aplicaciones IOS usando KMM - Kotlin Multiplatform, donde se utiliza una misma base de código para Android e IOS en la parte servidor y luego la parte gráfica es propia de cada plataforma.

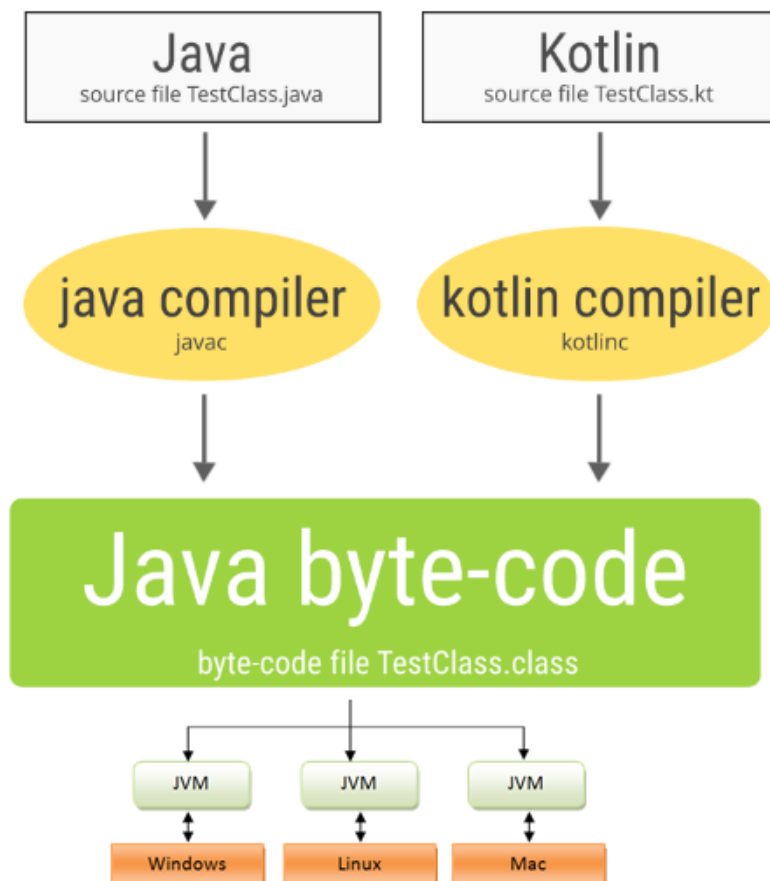
Kotlin es un lenguaje de programación [multiparadigma](#), es decir, integra dos o más maneras de programar. Algunos de los paradigmas de Kotlin son: lenguaje **estructurado**, como tal, hace uso de variables, sentencias condicionales, bucles, funciones ..., Kotlin admite la programación **funcional** pudiéndose definir expresiones lambda<sup>2</sup>, Kotlin es también un **lenguaje de programación orientado a objetos** y, por consiguiente, permite definir clases con sus métodos correspondientes y crear instancias de esas clases.

---

<sup>1</sup> JetBrains es la compañía creadora del IDE IntelliJ IDEA

<sup>2</sup> Una función/expresión lambda es una función anónima, es decir, una subrutina que no está ligada a un identificador. Suelen ser sintácticamente más simples que una función nominal y son características de los lenguajes funcionales.

Además, Kotlin es un lenguaje **multiplataforma**, es decir, permite la ejecución de un mismo programa en múltiples sistemas operativos. Esto es posible, ya que los ficheros fuentes de Kotlin (.kt), se compilan a *bytecode* (.class), que es independiente del sistema operativo. El único requisito, es que tenga instalada una máquina virtual de java (JVM).

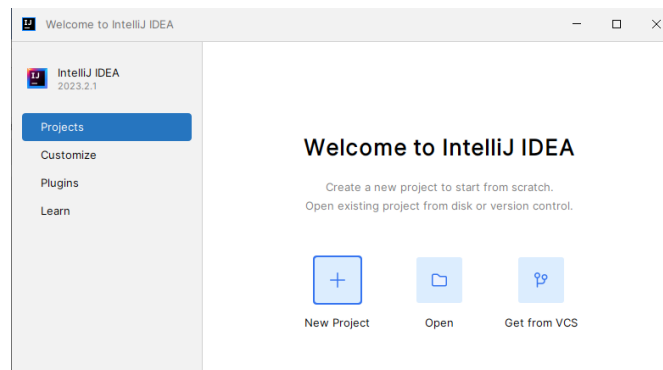


Cualquier editor simple es suficiente para escribir código en Kotlin aunque se recomienda el uso del IDE [IntelliJ IDEA](#) en su versión gratuita **Community Edition**, el cual ya que tienen algunas características que facilitan mucho la programación como el chequeo de errores mientras se escribe, el autocompletado de nombres de variables y funciones y mucho más. Este IDE te permite crear programas en Java y Kotlin, ya que incluye las herramientas necesarias para poder ejecutar dichos programas, es decir, incluye la JVM. En el caso del desarrollo de aplicaciones Android se recomienda el uso del IDE [Android Studio](#).

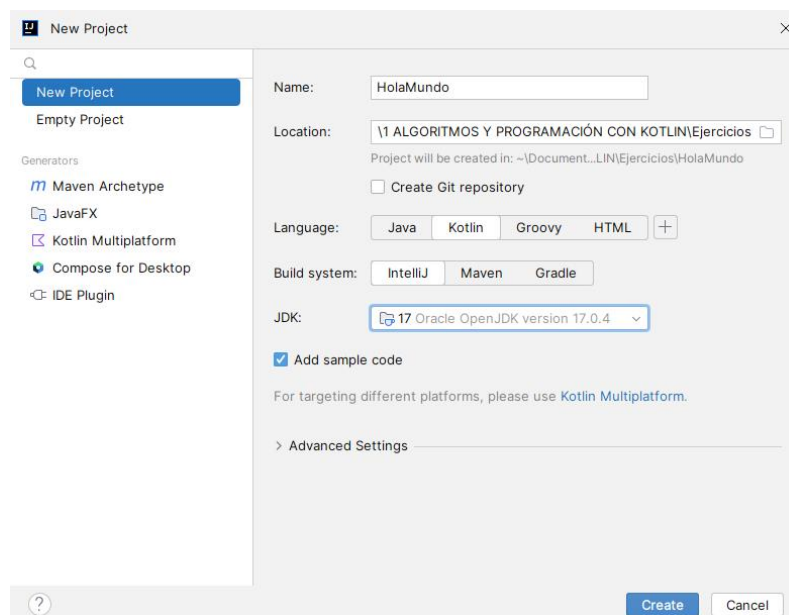
## ¡Hola mundo! - Mi primer programa

El primer programa que aprende a hacer cualquier aspirante a programador es un **Hola mundo**. Es seguramente el programa más sencillo que se puede escribir. Se trata de un programa que muestra por consola el mensaje “Hola mundo”.

Lo primero que vamos a hacer, es abrir **IntelliJ IDEA**, Cuando aparezca la ventana de bienvenida, ve a la sección **Projects** y presiona **New Project**:



En la ventana de **New Project**, configúralo tal y como se muestra a continuación, y después pulsa el botón “**Create**”:



Como dejamos marcada la opción “Add sample Code”, vemos como se nos ha creado nuestro primero programa, cuyo objetivo es mostrar por consola el texto “Hello World!”.



A partir de Kotlin 1.3, puedes declarar main() sin argumentos, permitiéndote acortar su sintaxis si sabes que no los usarás. Por lo que podemos modificarlo y dejarlo como se muestra a continuación:

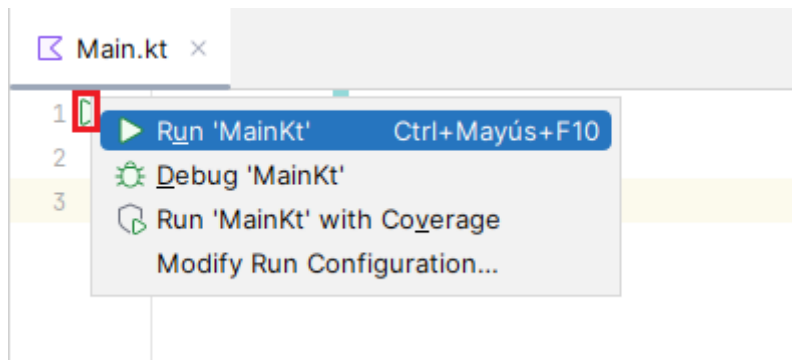


Las líneas del código anterior tienen el siguiente significado:

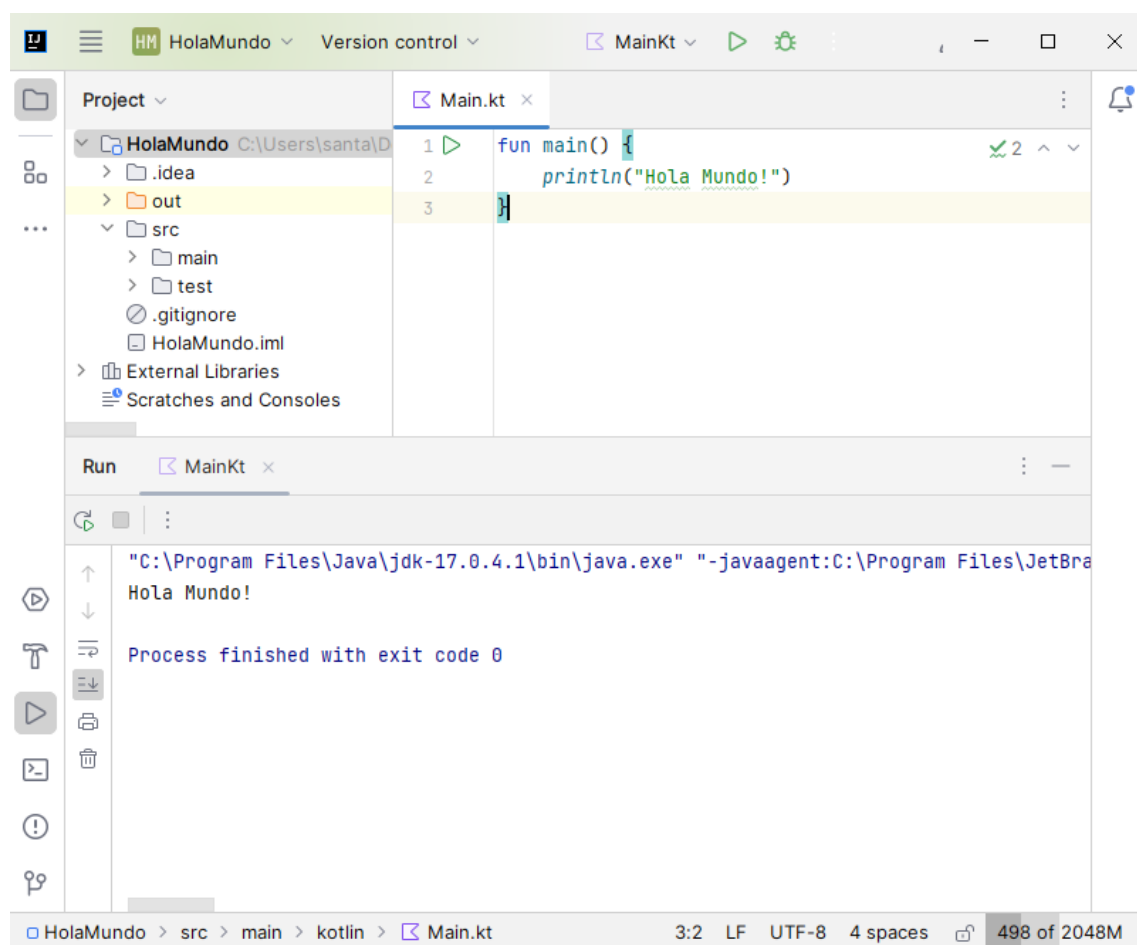
- La función `main()` es el punto de entrada de una aplicación Kotlin, ejecutando todas las instrucciones que pongas en su interior. Toma como parámetros los argumentos de línea de comandos en un array de strings.
- Las llaves, de apertura `{` y de cierre `}` especifican el inicio y el final del cuerpo de `main()`
- La función `println()` imprime en consola el texto pasado como argumento, en este caso "Hola Mundo!".
- Para escribir un comentario podemos usar `//` si es la misma línea o `/* */` si ocupa varias líneas.

A diferencia de Java, un archivo Kotlin como **Main.kt**, no necesita una clase declarada explícitamente, para definir código.

Solo nos queda ejecutar tu primer “Hola Mundo” en Kotlin y ver el resultado. Presiona el botón verde en la parte izquierda de main() y luego selecciona **Run «Main.kt»**.



Al ejecutarse, verás en la ventana Run, como te aparece el mensaje: **Hola mundo**



## Comentarios

Se pueden introducir comentarios en una única línea, mediante el uso de la doble barra (//). Todo lo que precede a //, hasta el fin de la línea, no será tomado en cuenta al interpretar el algoritmo.

También se pueden crear comentarios de mas de una línea, usando /\* para indicar el principio y \*/ para el final.

```
// This is an end-of-line comment

/* This is a block comment
   on multiple lines. */
```

## Variables y Tipos de datos

Una variable es una “celda” de la memoria destinada a almacenar un dato, que es referenciada por un nombre o identificador. Su valor puede ser modificado en cualquier momento del algoritmo, para que pueda ser incrementada, decrementado o simplemente reutilizada, es decir, es “variable”.

Antes de poder utilizar una variable, ésta se debe **declarar**. En Kotlin, una variable se declara utilizando la palabra reservada **var** seguida a continuación del nombre o identificador de la variable.

Por convención, para los **identificadores de variables**, se usa el estilo lower camel case, es decir, **comienzan en minúsculas** y si tiene varias palabras, no se usa subrayado y la inicial de cada palabra va en mayúsculas,

Ejemplos: *valorEnEuros*, *contador*, *isValid*

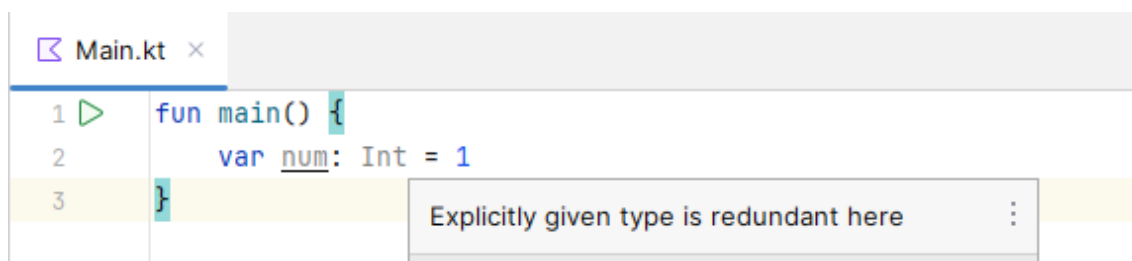
Si se declara una variable, es obligatorio, definir el tipo de datos de la variable, o bien asignarle un valor:

```
var num: Int
```

```
var num = 1
```

- En el primer caso, estamos definiendo una variable cuyo nombre es ‘num’ que contendrá datos de tipo entero. Usamos los dos puntos “:” para separar el identificador del tipo de datos.
- En el segundo caso, no se está definiendo el tipo, directamente se está asignando un valor de tipo entero, de esta manera, se infiere el tipo, indicando igualmente, que la variable ‘num’ contiene datos de tipo entero.

Si declaramos el tipo de la variable y la inicializamos a la vez, nuestro IDE nos dirá que lo que estamos haciendo es redundante, ya que, al inicializar la variable, ya estamos indicando el tipo.



```

1 fun main() {
2     var num: Int = 1
3 }

```

Explicitly given type is redundant here

Por tanto 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, es lo que se conoce como **inferencia de tipos**:

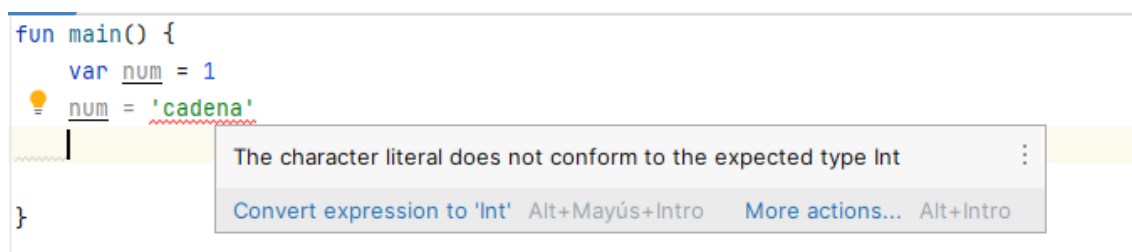
```

var num = 1 //se infiere entero
var cadena = "hola" //se infiere cadena de caracteres
var decimal = 2.5 //se infiere entero con decimales

```

Kotlin, es un **lenguaje fuertemente tipado**, esto quiere decir, que el tipo de datos con el que se crea la variable, es el tipo de datos que va a tener siempre, por tanto, el tipo de datos de una variable en Kotlin no se puede cambiar.

Por ejemplo, si a una variable que inicialmente fue declarada o inicializada con un tipo, luego le asignamos un dato de un tipo diferente, el compilador dará error:



```

fun main() {
    var num = 1
    num = 'cadena'
}

```

The character literal does not conform to the expected type Int

Convert expression to 'Int' Alt+Mayús+Intro    More actions... Alt+Intro



## Asignación de valores a variables

La sentencia de asignación se utiliza para dar un valor a una variable. En Kotlin (y en la mayoría de lenguajes de programación) se utiliza el símbolo igual ( = ) para este cometido.

Es importante recalcar que una asignación no es una ecuación.

Por ejemplo:

$$x = 7 + 1$$

es una asignación en la cual se evalúa la parte derecha  $7 + 1$ , y el resultado de esa evaluación se almacena en la variable que se coloque a la izquierda del igual, es decir, en la  $x$ , o lo que es lo mismo, el número 8 se almacena en  $x$ .

La sentencia  $x + 1 = 23 * 2$  no es una asignación válida ya que en el lado izquierdo debemos tener únicamente un nombre de variable.

```
fun main() {  
    var x = 2  
    var y = 9  
  
    var sum = x + y  
    println("La suma de mis variables es $sum") //11  
  
    var multi = x * y  
    println("La multiplicación de mis variables es $multi") //18  
}
```

## **Tipos de Datos**

Los tipos de datos más usados en Kotlin son:

- Int → Representa tipos de datos numéricos enteros
- Double → Datos numéricos decimales
- Char → Carácter
- String → Cadenas de caracteres
- Boolean → true para representar verdadero / false para representar falso (solo dos valores posibles)

¿Qué tipos de datos utilizarías para almacenar los siguientes datos?

- El nombre de una ciudad → String
- El precio de un kilo de naranjas → Double
- El número de ruedas de un coche → Int
- Si es o no un usuario vip → Boolean
- Letra → Char

En Kotlin, debemos tener en cuenta que todos los tipos son objetos.

## Valores numéricos

Para los **números enteros**, existen cuatro tipos con diferentes rangos de valores:

Tipo	Tamaño (bits)	valor mínimo	valor máximo
Byte	8	-128	127
Short	dieciséis	-32768	32767
Int	32	-2.147.483.648 ( $-2^{31}$ )	2.147.483.647 ( $2^{31} - 1$ )
Long	64	-9.223.372.036.854.775.808 ( $-2^{63}$ )	9.223.372.036.854.775.807 ( $2^{63} - 1$ )

Cuando se inicializa una variable entera sin una especificación de tipo explícita, el compilador infiere automáticamente el tipo **Int**. Si el valor supera el rango de Int, entonces será de tipo **Long**. Si deseas usar un tipo de tamaño distinto, debes definirlo explícitamente.

```
var one = 1 // Int
var threeBillion = 3000000000 // Long
var oneLong = 1L // Long
var oneByte: Byte = 1 //Byte
```

Para los **números reales**, existen los tipos **Float** y **Double**, precisión simple y doble:

Tipo	Tamaño (bits)	Bits significativos	bits exponentes	Dígitos decimales
Float	32	24	8	6-7
Double	64	53	11	15-16

El compilador de Kotlin inferirá el tipo **Double** si no especificas el tipo en una variable numerica con decimales. Para usar Float usa los literales constantes f o F como sufijo para la declaración o inferencia de tipo.

```
var dodgeChance = 0.2 //Double
var attackSpeed = 0.5f //Float
```

## Caracteres

Un carácter es el almacenamiento de un valor Unicode en una variable tipo **Char**. Su asignación se realiza poniendo cualquier símbolo entre comillas simples ( ' ).

```
var response = 'Y'
```

## Caracteres De Escape

En una variable de tipo **Char**, puedes contener una marca de escape representada por el backslash \. permitiéndote acceder por ejemplo, a los siguientes caracteres de escape.

- \t: Tabulación
- \r: Retorno de carro
- \n: Salto de línea
- \': Apostrofe
- \": Comilla doble
- \\: Backslash
- \\$: Símbolo de dólar

Por ejemplo, podemos usar saltos de línea en el siguiente mensaje:

```
print("Uno\nDos\nTres")
```

Mostraría por consola:

```
Uno  
Dos  
Tres
```

## Cadenas de Caracteres

Las cadenas en Kotlin están representadas por el tipo **String**. Generalmente, un valor de cadena es una secuencia de caracteres entre comillas dobles ( " )

```
var str = "abcd 123"
```

Las cadenas son inmutables, esto quiere decir, que una vez que se inicializa una cadena, no puede cambiar su valor. Todas las operaciones que transforman cadenas devuelven sus resultados en un nuevo objeto **String**, dejando la cadena original sin cambios.

```
var str = "abcd"  
println(str.uppercase()) // ABCD  
println(str) //abcd
```

Para concatenar cadenas, podemos utilizar el operador +. Esto también funciona para concatenar cadenas y variables de otros tipos, siempre que el primer elemento de la expresión sea una cadena:

```
var s = "abc"+1 // abc1  
println(s+"def") // abc1def
```

También se pueden interpolar datos en las cadenas de texto usando el carácter \$ delante del nombre de la variable, dentro de una cadena de caracteres:

```
var myString = "hola"  
println("Este mensaje es "+myString) // concatenación de cadenas  
println("Este mensaje es $myString") // interpolación de datos en las cadenas de texto
```

## Valores nulos en Kotlin

Kotlin es **Null Safety**, es decir, que gestiona los nulos de forma segura, de modo que puedes garantizar que tu código no va a producir `KotlinNullPointerException`.

En Kotlin los objetos por defecto no aceptan valores nulos, para que le podamos asignar un null tendremos que indicárselo específicamente. De esta forma vamos a poder garantizar que no se producirá un `NullPointerException` en tiempo de ejecución sin necesidad de llenar todo el código de comprobaciones `if (a != null)` o `if (b == null)` ya que las variables no podrán ser null, pero también nos da una vía de escape para que podamos tener los queridos `NullPointerException`

Como ya hemos dicho, por defecto los objetos en Kotlin son not-null y si intentamos asignarle un null a una variable directamente nos va a dar error.

```
var nombre: String  
nombre = null
```

Null can not be a value of a non-null type String

Add 'toString()' call Alt+Mayús+Intro More actions... Alt+Intro

Si queremos que los objetos sí que sean nullable, tenemos que añadir **?** después del nombre del tipo.

```
var nombre: String?  
nombre = null
```

Pero, a partir de ese momento, el compilador te obligará a comprobar el nulo antes de hacer nada con la variable. De esta forma, se asegura que no se produce un `NullPointerException`.

Por ejemplo:

```
val y = x.toDouble()
```

No compilará, si no compruebas primero si es nulo:

```
if (x != null) {
    val y = x.toDouble()
}
```

Kotlin, nos da dos opciones, para no tener que estar comprobando continuamente si un objeto es nulo:

```
var nombre: String?
nombre = null

var longitud: Int = nombre.length
```

Only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable receiver of type String?  
 Replace with safe (?.) call Alt+Mayús+Intro More actions... Alt+Intro

En el caso anterior, Kotlin no nos deja pedir la longitud de la cadena porque puede ser nulo, y nos dice que tenemos 2 opciones:

- ?. si queremos hacerlo de forma segura, en este caso, si la variable no es nula ejecutará la operación, y en caso contrario, la variable tendrá valor null.
- !!. si no queremos que el compilador compruebe si puede ser null. Solo deberíamos usarlo si estamos completamente seguros que no será nulo, ya que en el caso que sea nulo, se producirá un NullPointerException.

Ejemplo, de cómo se haría de forma segura. Si usamos ?. estamos propagando a los siguientes objetos o funciones la necesidad de soportar nulos. Por ejemplo, vemos como al pedir la longitud, debemos indicar que la variable “longitud” también debe admitir nulos.

```
var nombre: String? = null

var longitud: Int? = nombre?.length
println("Longitud: ${longitud}") // Longitud: null
```

Para evitar esto, podemos usar el operador Elvis `?:` que nos permite definir un valor alternativo si se encuentra un null, por ejemplo, para la longitud podríamos usar 0.

```
fun main() {
    var nombre: String?
    nombre = null

    var longitud: Int = nombre.length
}
```

Replace with safe (?.) call  
Add non-null asserted (!) call  
Inspection 'Constant conditions' options  
Press Ctrl+Q to toggle preview

5 var longitud: Int = nombre?.length ?: 0

```
var nombre: String? = null

var longitud: Int = nombre?.length ?: 0
println("Longitud: ${longitud}") // Longitud: 0
```

Ejemplo, de cómo se haría de forma no segura, desactivando la comprobación de nulos y por lo tanto compilará sin problemas, eso sí, el resultado de la ejecución puede no ser bueno... En Kotlin no tiene sentido utilizar este modo, pero bueno, es la opción si echamos de menos los null pointers...

```
var nombre: String?
nombre = null

var longitud: Int = nombre.length
```

Replace with safe (?.) call  
Add non-null asserted (!) call  
Inspection 'Constant conditions' options  
Press Ctrl+Q to toggle preview

5 var longitud: Int = nombre!!.length



## Alcance de las variables

El alcance de las variables se refiere al ámbito en que éstas pueden ser utilizadas. Como regla general hay que considerar que una variable podrá ser utilizado sólo dentro del bloque de código donde ha sido declarada y nunca antes de hacerlo.

Una variable existe desde su declaración en un bloque, y deja de existir cuando la ejecución abandona el bloque donde se declaró.

Vamos a hablar de variables globales y locales para explicar el alcance de éstas.

- Las variables locales son aquellas que se declaran en algún bloque del programa y perduran solamente dentro de ese bloque. Podrán ser, por tanto, utilizadas solamente dentro del bloque donde fueron declaradas.
- Las variables globales son aquellas que se declaran a nivel clase, al inicio del programa y perduran durante toda su ejecución. Pueden ser utilizadas desde cualquier parte del mismo.

En Kotlin, se pueden declarar variables locales sin inicializar, pero no así si se trata de una variable global cuyo alcance es infinito, por lo tanto, el compilador no tiene forma de garantizar que el campo se inicialice antes de acceder a él. Para evitar que el programa falle al utilizar una variable no inicializada, el compilador genera un error y esto no lo permite.

```
class Hola {  
  
    var prueba: String // esto no está permitido por el compilador  
  
    fun mensaje(){  
        var abc: String // como variable local si está permitido  
  
        abc = "hola"  
        println(abc)  
    }  
}
```

También debemos tener en cuenta, que dos variables no podrán llamarse de la misma manera, dentro del mismo ámbito. Una excepción a esta regla, es que las variables declaradas a nivel clase –**variables globales**– pueden coexistir con variables declaradas a nivel de método –**variables locales**–. Cuando se referencia el nombre de la variable, se utilizará la local si existe una con el mismo nombre en ese ámbito; en caso contrario se estará usando la variable de la clase.

```
Hola.kt x
1  class Hola {
2
3      var abc = "adios"
4
5      fun mensaje(){
6          var abc = "hola"
7          println(abc)
8      }
9  }
```

```
Main.kt x
1  fun main(){
2
3      var hola = Hola()
4      hola.mensaje() // hola
5
6  }
```

## Constantes

Las constantes son estructuras que se definen con un **valor fijo que no se puede modificar**. Se utilizan para evitar tener que escribir números literales a lo largo del código, cuyos valores pueden cambiar en un futuro (nunca en la ejecución del algoritmo), o que se repiten varias veces.

De esa manera, colocaremos el nombre de la constante que defina dicho valor y en el momento en que ésta cambie, sólo tendremos que cambiar el valor en el lugar donde fue definida.

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 NUMERO_PAGINAS: Int = 10 // Asignación junto a declaración
val LONGITUD: Int // Declaración
LONGITUD = 5 // Asignación
```

Se usa **val**, para definir variables que serán de solo lectura, es decir, se asigna valor una sola vez y no se podrá cambiar, es similar a usar la palabra reservada final en Java.

En Kotlin, también se usa la palabra reservada **const**, para variables cuyo valor se conocen en tiempo de compilación. Declarar una variable **const** es muy parecido a usar la palabra clave **static** en Java.

```
const val WEBSITE_NAME = "Baeldung"
```

Hay que tener en cuenta que Kotlin recomienda utilizar una notación diferente para los identificadores de las constantes. Se recomienda utilizar sólo mayúsculas y si el nombre contiene más de una palabra, se deben de separar con el carácter subrayado.

## Operadores

**Aritméticos:** Permiten realizar operaciones aritméticas entre dos o más variables o expresiones. Son operadores aritméticos, los ya conocidos (+, -, \*, /) y además el resto que se escribe %. El resultado de estos operadores variará según uses enteros o reales.

Operador	Operación	Expresión
+	Suma	<code>a+b</code>
-	Resta	<code>a-b</code>
*	Multiplicación	<code>a*b</code>
/	División	<code>a/b</code>
%	Residuo	<code>a%b</code>

### Ejemplo:

```
// al dividir dos enteros,  
// se conserva la parte entera como resultado y se descartan los decimales  
var resultado = 14 / 5  
println(resultado) // 2  
  
// para que el resultado sea decimal  
// se han de dividir dos números decimales  
var a = 14.0  
var b = 5.0  
var resultado2 = a/b  
println(resultado2) // 2.8
```

**Relacionales:** El resultado de una expresión relacional siempre es booleano, si se cumple será verdadero (true), si no se cumple será falso (false).

Los operadores relacionales o de comparación son:

<, >, >=, <=, != (distinto) , == (igual)

Operador	Enunciado	Expresión
==	<i>a</i> es igual a <i>b</i>	<code>a==b</code>
!=	<i>a</i> es diferente de <i>b</i>	<code>a!=b</code>
<	<i>a</i> es menor que <i>b</i>	<code>a&lt;b</code>
>	<i>a</i> es mayor que <i>b</i>	<code>a&gt;b</code>
<=	<i>a</i> es menor ó igual que <i>b</i>	<code>a&lt;=b</code>
>=	<i>a</i> es mayor o igual que <i>b</i>	<code>a&gt;=b</code>

### Ejemplo:

```
fun main() {
    val a = 17
    val b = 20

    println("$a es igual a $b: ${a == b}") //17 es igual a 20: false
    println("$a es diferente a $b: ${a != b}") //17 es diferente a 20: true
    println("$a es menor que $b: ${a < b}") //17 es menor que 20: true
    println("$a es mayor que $b: ${a > b}") //17 es mayor que 20: false
    println("$a es menor o igual que $b: ${a <= b}") //17 es menor o igual que 20: true
    println("$a es mayor o igual que $b: ${a >= b}") //17 es mayor o igual que 20: false
}
```

**Lógicos:** && (es el AND), || (es el OR), ! (es el NOT)

El operador **!** permite lo que se conoce como “negar” el valor de la variable. Si la variable que lo acompaña es de tipo booleano, devolverá el valor contrario (true si la variable es false, y viceversa)

Operador	Descripción	Expresión
&&	Conjunción (and): el resultado es <code>true</code> si <code>a</code> y <code>b</code> son <code>true</code>	<code>a&amp;&amp;b</code>
	Disyunción (or): el resultado es <code>true</code> si <code>a</code> o <code>b</code> son <code>true</code>	<code>a  b</code>
!	Negación (not): el resultado es <code>false</code> si <code>a</code> es <code>true</code> , o viceversa	<code>!a</code>

**Ejemplo:**

```
fun main() {
    val input = 5
    var res: Boolean

    res = (input > 0) && ((input % 2) == 0)
    println("Es mayor que cero y par: $res") //Es mayor que cero y par: false

    res = (input > 0) || ((input % 2) == 0)
    println("Es mayor que cero o par: $res") //Es mayor que cero o par: true

    res = (input > 0) && !((input % 2) == 0)
    println("Es mayor que cero e impar: $res") //Es mayor que cero e impar: true
}
```

### Operadores Unarios y de Asignación Compuesta:

Los operadores unarios de incremento `++` y decremento `--`, actúan sobre una única variable:

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

`num++;`      *// es equivalente a `num = num + 1;`*

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

`num--;`      *// es equivalente a `num = num - 1;`*

Los operadores de asignación compuesta son la combinación entre el operador de asignación y los operadores aritméticos, con el fin de usar como operando la variable de resultado: `+=n`, `-=n`, `*=`, etc.

Operador	Expresión simplificada	Expresión Completa
<code>+=</code>	<code>a+=b</code>	<code>a=a+b</code>
<code>-=</code>	<code>a-=b</code>	<code>a=a-b</code>
<code>*=</code>	<code>a*=b</code>	<code>a=a*b</code>
<code>/=</code>	<code>a/=b</code>	<code>a=a/b</code>
<code>%=</code>	<code>a%=b</code>	<code>a=a%b</code>

### Ejemplos:

Operador de asignación multiplicación \*= Multiplica el valor de la variable del lado izquierdo por el valor de la expresión del lado derecho, y le asigna el resultado

```
var x = 10  
x *= 2 + 3 // x = x * (2 + 3)  
println(x) // 50;
```

Operador de asignación división /= divide el valor de la variable del lado izquierdo por el valor de la expresión del lado derecho, y le asigna el resultado

```
var x = 100  
x /= 8 + 2 // x = x / (8 + 2)  
println(x) // 10;
```

Operador de asignación módulo %= calcula el módulo de la variable con la expresión del lado derecho, y le asigna el resultado

```
var x = 9  
x %= 3 - 1 // x = x % (3 - 1)  
println(x) // 1;
```



**Orden de evaluación de los operadores:** Es importante tener en cuenta el orden de evaluación de los operadores ya que, en algunos casos, definen el resultado de la operación. Pongamos algún ejemplo:

```
var x = 3 + 2 * 5  
println("Cuanto vale x? = $x");
```

Si se quiere alterar la precedencia de algunos operadores o el orden de evaluación de las expresiones, se pueden hacer uso de los paréntesis para agrupar expresiones que queremos que se evalúen antes que otras.

Hay que tener en cuenta que Kotlin, al igual que la mayoría de los lenguajes de programación **siempre evaluará primero las expresiones más internas**. Pero, en aquellas expresiones en igualdad de condiciones, las operaciones **multiplicación y división** (el módulo también) tienen precedencia frente a la suma y la resta.

1. Paréntesis.
2. Exponente.
3. Multiplicación.
4. División.
5. Módulo.
6. Suma.
7. Resta.

Además, los operadores unarios tienen siempre precedencia sobre el resto de operadores.

```
var x = (3 + 2) * 5  
println("Cuanto vale x = $x") //25  
  
x = (3 + 2) * 5 + (3 * 2 * (3 + 2))  
println("Y ahora: $x") //55
```

## Escritura y Lectura por consola

### Escritura por consola

Para escribir cualquier tipo de expresión en la consola se utilizan los métodos **println** (con salto de línea), **print** (sin salto de línea) del paquete [kotlin.io](https://kotlin.io), para imprimir el mensaje que desees pasar como parámetro.

- Para escribir con salto de línea, usaremos **println**:

```
println(3<2);    // Escribe false
println("Mensaje");    // Escribe Mensaje
println(4*4);    // Escribe 16
```

- Para escribir sin salto de línea, usaremos **print**:

```
print("Mensaje en una ");
print("sola linea");
// Mensaje en una sola linea
```

### Lectura por consola

Lee datos desde el teclado del usuario con la función **readLine()**, la cual monitorea indefinidamente el flujo de entrada, hasta que se confirme la escritura de bytes con la tecla **ENTRAR**.

El resultado retornado es un String anulable, es decir la función `readLine` devuelve `String?`.

Por ejemplo:

```
println("Introduce tu nombre")
val nombre = readLine()

println("Introduce tu edad")
val edad = readLine()

println("Tu nombre es: $nombre, y tu edad: $edad")
```

Nota: Si lo ejecutas, en la ventana **Run** de IntelliJ IDEA, la aplicación estará esperando por tu entrada y confirmación con la tecla ENTER.

## Estructuras de control

### Sentencia condicional simple: IF

La sentencia **if** permite ejecutar una serie de acciones en base a una condición. Dicha condición puede ser simple (comparar dos expresiones) o bien compleja, pudiéndose comparar múltiples expresiones unidas por cualquier operador lógico.

Si el resultado final de la condición es verdadero (la condición se cumple), se ejecutará el bloque de código asociado a la instrucción **if**, y a continuación, el programa seguirá inmediatamente después. En caso contrario, no se ejecutará ese código y el programa seguirá inmediatamente después.

```
if (expresión-lógica) {  
    instrucción 1;  
    ...  
}
```

### Ejemplo:

```
println("Introduce un numero")  
val numero: Int = readln().toInt()  
  
if ( numero > 0){  
    println("$numero es un numero positivo")  
}
```

### Sentencia condicional doble: IF/ELSE

Es igual que la anterior, sólo que se añade un apartado **else** que contiene las acciones que se ejecutarán si la **expresión lógica** evaluada no se cumple.

#### Ejemplo:

```
println("Introduce un numero")
val numero: Int = readln().toInt()

if ( (numero % 2) == 0){
    println("$numero es un numero par")
}
else {
    println("$numero es un numero impar")
}
```

### Sentencia if/else if

Este es el caso más completo de la sentencia **if**. Realmente es posible evaluar tantas condiciones como se deseen y asociar un bloque de código al cumplimiento de cada una de esas condiciones, teniendo en cuenta que, por cada sentencia **if** sólo se acabará ejecutando un sólo bloque de instrucciones.

Se ejecutará siempre el bloque asociado a la condición que antes se cumpla (en orden de arriba a abajo) o el bloque asociado al caso **else** si no se cumple ninguna condición (siempre que se haya incluido caso else). Si no se incluye ningún else, es posible que, aun incluyendo una sentencia **if**, puede que nunca se ejecute ninguna sentencia.

#### Ejemplo:

```
println("Introduce un numero")
val x: Int = readln().toInt()

if (x > 10) {
    println("$x es mayor que 10");
}
else if (x < 10) {
    println("$x es menor que 10");
}
else {
    println("$x es igual a 10");
}
```

## Operador condicional

Los operadores condicionales permiten evaluar varias expresiones como si se trataran de una sola condición.

- Operador AND: Representado por los caracteres **&&**, se utiliza para combinar varias expresiones y que éstas sean evaluadas como una sola condición verdadera cuando todas ellas lo sean. Si alguna de ellas es falsa, el resultado siempre será falso independientemente del valor de todas las demás.
- Operador OR: Representado por los caracteres **||**, se utiliza para combinar varias expresiones y que éstas sean evaluadas como una sola condición verdadero simplemente con que una de ellas lo sea. Si todas ellas son falsas, el resultado de la condición compuesta será falso.
- Operador NOT: Representado por el caracter **!**, se utiliza para invertir el resultado de la expresión que queda a su derecha.

### Ejemplo:

```
println("Introduce un numero")
val x: Int = readln().toInt()

if ((x > 10) && (x < 20)) {
    println("El valor de $x está entre 10 y 20");
}
if ((x == 10) || (x == 20)) {
    println("El valor de $x es 10 o 20");
}
if (!(x == 10)) {
    println("$x es distinto que 10");
}
```

Nota: Es posible combinar expresiones con distintas operaciones: &&, ||, !

```
if ( ((x > 10) && (x < 20) && !(x == 15)) || (x > 30) ) {
    println("El valor de $x está entre 10 y 20, pero distinto de 15, o bien es mayor de 30");
}
```

## Sentencia condicional múltiple: WHEN

La expresión condicional **when**, te permite comparar el valor de un argumento contra una lista de entradas.

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

- Expresiones
- Comprobaciones de rango
- Comprobaciones de tipo

Al haber coincidencia, se ejecutará el código del caso en cuestión. Al igual que la expresión **if**, se usa **else** en caso de que ninguno de los casos coincida.

Sirve como sustituta de algunas expresiones de tipo **if-else**, considerándose análoga a la sentencia **switch** de Java.



```
switch (...) {  
    case :  
        ....  
    Case :  
        ....  
    default :  
        ....  
}
```

```
when (...) {  
    condition ->  
        ....  
    condition ->  
        ....  
    else ->  
        ....  
}
```

### Ejemplo 1: El argumento a analizar es una variable de tipo entero

Se pide al usuario introducir un número, el número introducido por el usuario se guarda en la variable `x`, con la instrucción `when(x)`, se evalúa el valor almacenado en la variable `x`, si se cumple la condición que `x` sea igual a 1, se ejecuta la parte derecha de esa condición, es decir, se escribirá por consola "x es 1", si es 2 se escribe "x es 2", si no es ninguno de los casos definidos, imprimirá "x no es ni 1 ni 2".

En este ejemplo, se está evaluando el valor de una expresión, concretamente el valor de una variable.

```
println("Introduce un número")
var x: Int = readln().toInt()

when(x) {
    1 -> print("x es 1")
    2 -> print("x es 2")
    else -> {
        print("x no es ni 1 ni 2")
    }
}
```

### Ejemplo 2:

```
println("Identifica el día de la semana")
println("Introduce un número del 1 al 5")
var dia = readln().toInt()
when (dia) {
    1 -> println("es lunes... (sniff)")
    2 -> println("Es martes")
    3 -> println("Es Miercoles!")
    4 -> println("Es Jueves!!")
    5 -> println("ES VIERNES!!!")
    else -> println("** NO HAY CLASE **")
}
```

### Ejemplo 3: El argumento a analizar es una variable de tipo cadena

```
println("Escribe el nombre de un mes")
val mes = readln()
when (mes) {
    "Enero" -> println("Frío")
    "Febrero" -> println("Más frío")
    "Marzo" -> println("Saca la cabeza el lagarto")
    "Abril" -> println("Aguas mil")
    else -> {
        println("El mes introducido no es correcto")
    }
}
```

### Ejemplo 4: Múltiples valores en una condición

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

```
var input = 2

when (input) {
    1, 2, 3 -> print("Te toca turno nocturno")
    4, 5, 6 -> print("Te toca turno diurno")
}
```

### Ejemplo 5: Uso de expresiones en la condición

```
val input = 100
when (input) {
    Int.MAX_VALUE -> print("Límite superior")
    Int.MIN_VALUE -> print("Límite inferior")
    else -> print("No es ninguno de los límites")
}
```



### Ejemplo 6: Uso de rangos de valores en la condición

En este ejemplo, podemos comprobar si está entre un rango de valores específicos (en este caso entre 1 y 6 ó 7 y 12), o si por el contrario no está en un rango específico (de 1 a 12) poniendo una exclamación al principio de la expresión *in*.

```
fun getMonth(month : Int){  
    when (month) {  
        in 1 ≤ .. ≤ 6 -> print("Primer semestre")  
        in 7 ≤ .. ≤ 12 -> print("segundo semestre")  
        !in 1 ≤ .. ≤ 12 -> print("no es un mes válido")  
    }  
}
```

### Ejemplo 7: Comprobación de tipos en la condición

También podemos usar la expresión *is* para comprobar el tipo de variable que es.

```
fun evaluaTipos(value: Any){  
    when (value){  
        is Int -> print(value + 1)  
        is String -> print("El texto es $value")  
        is Boolean -> if (value) print("es verdadero") else print("es falso")  
    }  
}
```

### Ejemplo 8: Usar When como un if

```
var a = -5  
when {  
    a > 0 -> print("Es positivo")  
    a == 0 -> print("Es cero")  
    else -> print("Es negativo")  
}
```

## Estructuras de repetición

Las estructuras de bucle o repetición permiten ejecutar de forma repetida estructuras de código. Existen diferentes tipos de bucle de forma que, como programador, podrás elegir la que más se adecue a cada situación.

Debe existir una condición de salida, que hace que el flujo del programa abandone el bucle y continúe justo en la siguiente sentencia. Si no existe condición de salida o si esta condición no se cumple nunca, se produciría lo que se llama un bucle infinito y el programa no terminaría nunca.

### Sentencia while

La sentencia **while** permite crear bucles. Un bucle es un conjunto de sentencias que se repiten **mientras se cumpla una determinada condición**. Por tanto, NO se sale de un bucle while hasta que la condición que se evalúa sea falsa.

Esta condición **se evalúa antes** de entrar dentro el bucle, por tanto, el cuerpo del bucle podrá no ejecutarse nunca si la primera vez la condición es falsa.

No nos importa el número de veces que se tenga que ejecutar el bucle, sino que éste permanecerá mientras que ocurra algo.

```
while (expresión lógica) {  
    instrucciones;  
}
```

#### Ejemplo 1:

// Escribe los números del 1 al 99

```
var i = 0  
while (i < 100) {  
    println(i)  
    i++  
}
```

#### Ejemplo2:

// Bucle infinito, escribe “...” continuamente

```
while (true) {  
    println("...")  
}
```

## Sentencia do-while

La única diferencia con respecto a la sentencia **while**, es que la expresión lógica **se evalúa después** de haber ejecutado las instrucciones del bucle. Es decir, **el bucle se ejecuta al menos una vez**.

Esta sentencia es útil para la **validación** de datos de entrada.

```
do {  
    instrucciones;  
}while (expresión lógica);
```

**Ejemplo 1:** lee un número por teclado y lo convierte a entero, hasta que deje de cumplirse la condición, es decir, hasta que el número sea mayor que cero.

```
var num: Int  
do {  
    print("Teclea número mayor de 0")  
    num = readln().toInt()  
} while (num <= 0)
```

## Ejemplo 2:

- 1) se escribe el contenido de cont que es 10 (no se evalúa la condición).
- 2) se decrementa en 1 la variable cont.
- 3) ahora se evalúa la condición; como  $(9 \geq 0)$  es verdadero, se vuelve a realizar el bucle hasta que deja de cumplirse la condición.

```
var cont = 10  
do {  
    println(cont)  
    cont--  
} while (cont >= 0)
```

## Sentencia for

El uso en Kotlin del bucle **for** es uno de los más habituales, ya que continuamente estamos recorriendo colecciones de objetos, desde el principio hasta el fin. El bucle **for** en Kotlin, se asemeja a las sentencias **foreach** de otros lenguajes.

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

### Ejemplo 1: Imprimir la cuenta del uno al cinco

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.

Declaración de variables

Expresión contenedora

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

Cuerpo del bucle

### Ejemplo 2: Recorrer un rango

Este ejemplo recorre el rango de caracteres de la 'a' a la 'f' de diferentes formas:

```
// iteración regular
for (char in 'a' .. 'f') print(char) //abcdef

// iteración con avance de 2
println()
for (char in 'a' .. 'f' step 2) print(char) //ace

println()
// iteración inversa
for (char in 'f' downTo 'a') print(char) //fedcba

// iteración excluyendo el límite superior
println()
for (char in 'a' until 'f') print(char) //abcde
```

### Ejemplo 3: Recorrer una cadena de caracteres

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

```
for(c in "Ejemplo"){
    print(c)
}
```

#### Ejemplo 4: Recorrer un array

Para iterar a lo largo de un `array` con el bucle `for`, debemos usar como base los índices de sus elementos, que devuelven la posición de cada elemento contenido en el array.

```
var colores = arrayOf("Blanco", "Negro", "Rojo", "Verde", "Azul")

for (i in colores.indices) {
    println("En la posición $i, se encuentra el color: ${colores[i]}")
}
```

La salida será:

```
En la posición 0, se encuentra el color: Blanco
En la posición 1, se encuentra el color: Negro
En la posición 2, se encuentra el color: Rojo
En la posición 3, se encuentra el color: Verde
En la posición 4, se encuentra el color: Azul
```

**Ejemplo 5:** Recorrer un array usando la función `withIndex()` que devuelve el par (índice, valor)

```
var colores = arrayOf("Blanco", "Negro", "Rojo", "Verde", "Azul")

for ((index, value) in colores.withIndex()) {
    println("El elemento en la posición $index es $value")
}
```

La salida será:

```
El elemento en la posición 0 es Blanco
El elemento en la posición 1 es Negro
El elemento en la posición 2 es Rojo
El elemento en la posición 3 es Verde
El elemento en la posición 4 es Azul
```

## Interrupción de una estructura de bucle

Todas las estructuras de bucle pueden ser interrumpidas utilizando la instrucción **break**. Cuando se encuentra dicha instrucción, se corta la ejecución completa de todo el bucle (la iteración actual y todas las que pudieran quedar) y el código continua su ejecución inmediatamente después del final de la estructura de bucle.

```
while (condicion1) {  
    sentencia1;  
    sentencia2;  
  
    // Si se cumple condicion2, se rompe el bucle  
    // y la ejecución del programa seguirá en sentencia4  
    if (condicion2)  
        break;  
  
    sentencia3;  
}  
  
sentencia4;
```

La instrucción **continue** corta la ejecución de la iteración actual y vuelve al inicio del bucle para continuar su ejecución justo en la iteración siguiente a la interrumpida.

```
do {  
    sentencia1;  
    sentencia2;  
  
    // Si se cumple la condición no se ejecutarán  
    // sentencia3 y sentencia4 para esa iteración  
    // Pero el bucle continuará con la siguiente iteración  
    if (condicion2)  
        continue;  
  
    sentencia3;  
    sentencia4;  
} while (condicion1)
```

## Funciones en Kotlin

Las funciones se declaran usando la palabra clave **fun**, seguida del nombre, y entre paréntesis declararemos los valores de entrada (también llamados parámetros de entrada), pero también podemos tener funciones sin parámetros de entrada. Además, las funciones pueden devolver algún valor.

### Ejemplo de funciones sin parámetros

En el siguiente ejemplo, tenemos una función **showMyName** cuyo objetivo es mostrar el nombre, por otra parte, también tenemos la función **showMyLastName**, cuyo objetivo es mostrar el apellido, pero estas funciones no se van a ejecutar, a menos que sean llamados.

Existe un método especial llamado **main** que, al incluirlo en un fichero de Kotlin, hará que ese fichero sea ejecutable. Este método **main**, será el invocado cuando se ejecute el fichero. Por eso, si en el método **main**, llamamos a los métodos anteriores, haremos que se ejecute su código.

```
fun main() {  
    showMyName()  
    showMyLastName()  
}  
fun showMyName(){  
    println("Me llamo Luna")  
}  
fun showMyLastName(){  
    println("Mi Apellido es García")  
}
```

NOTA: Una aplicación deberá tener una clase llamada **Main.kt** con un método **main**, que será el punto de entrada de la aplicación.



### Ejemplo de funciones con parámetros de entrada

Ahora vemos una función llamada `showMyInformation` con parámetros de entrada. Como podemos ver, tiene dos parámetros de entrada, la forma de declararlos es muy fácil, se pone el nombre del parámetro seguido de dos puntos y el tipo de datos de ese parámetro. Una función puede tener tantos parámetros como sean precisos.

Para llamar a esta función, habrá que pasarle las variables o valores por cada parámetro que tenga definidos.

```
fun main() {  
    showMyInformation( name: "Luna",  lastName: "García")  
}  
fun showMyInformation(name: String, lastName: String){  
    println("Me llamo $name con apellidos $lastName")  
}
```

### Ejemplo de funciones que devuelven un valor

Nos queda por ver como una función puede devolver un resultado calculado en la función. La única limitación, es que solo se puede devolver un valor.

Las funciones que devuelven un valor, deben indicar el tipo de datos del valor devuelto en la definición de la función, justo después de definir los parámetros de entrada, por lo que después del paréntesis de cierre, se pondrán dos puntos **:** seguidos del tipo de datos del valor a devolver.

Para que la función devuelva el valor calculado se debe utilizar la palabra clave `return` seguido de la variable o valor que se quiere devolver.

```
fun main() {  
    var resultado = suma(1, 2)  
    println("El resultado de la suma es $resultado")  
}  
fun suma(firstNumber: Int, secondNumber: Int): Int{  
    return firstNumber + secondNumber  
}
```

## Clases

El nombre de una clase debe comenzar por Mayúsculas y utilizar el estilo de escritura camel case. Cada clase en Kotlin estará escrita en un fichero fuente con el **mismo nombre de la clase** y extensión .kt .

Para definir una clase, usa la palabra clave **class** seguido de su nombre.

Los contenidos de una clase deben ir en el siguiente orden:

1. Declaraciones de propiedades y bloques inicializadores.
2. Constructores secundarios
3. Declaraciones de funciones

No tenemos que ordenar las declaraciones de las funciones alfabéticamente ni por visibilidad, etc. En su lugar, junta las cosas que estén relacionadas, de modo que alguien que lea la clase de arriba a abajo pueda seguir la lógica de lo que está sucediendo.

**Ejemplo:**

```
class Complejo{  
    var re: Int = 0  
    var im: Int = 0  
  
    fun suma(v: Complejo) {  
        re = re + v.re  
        im = im + v.im  
    }  
}
```

**Modificador de visibilidad:** Para indicar la visibilidad, se deben usar los modificadores de visibilidad, los cuales se pueden indicar, o no, también se les conoce como modificadores de acceso (o scope).

Las clases, interfaces, constructores y funciones, así como las propiedades y sus definidores, pueden tener modificadores de visibilidad.

Existen los siguientes modificadores de visibilidad en Kotlin:

- **public:** Por lo general se indicará una visibilidad completa a través de la palabra reservada `public`. En Kotlin, `public` es la visibilidad por defecto, tanto para las clases, interfaces, funciones y propiedades, lo que significa que son visibles para todo el proyecto. Puedes usar explícitamente `public` para mayor claridad, pero no es necesario.

```
public class MiClasePublica {  
    public val miAtributo = 1  
}
```

```
class MiClasePublica {  
    val miAtributo = 1  
}
```

- **private:** El modificador `private` hace que el atributo sea visible solo para la propia clase. Esto es igual que en Java.

```
class MiClasePrivada {  
    private val miAtributo = 2  
}
```

- **protected:** En Kotlin, el modificador `protected` hace que los atributos sean visibles para la propia clase y las clases que hereden de ella, pero no es visible para otras clases en el mismo paquete.

```
class MiClaseProtegida {  
    protected val miAtributo = 3  
}
```



- **internal**: En Kotlin, el modificador `internal` significa que estará disponible solo en el mismo módulo. Por módulo en Kotlin, nos referimos a un grupo de archivos que se compilan juntos. Los módulos pueden ser: proyectos maven, conjuntos gradle, archivos generados a partir de una tarea Ant o un módulo IntelliJ IDEA. Este modificador no existe en Java.

```
class MiClaseInternal {  
    internal val miAtributo = 4  
}
```

## Objetos, miembros y referencias

Un objeto es una instancia (ejemplar) de una clase. La clase es la definición general y el objeto es la materialización concreta (en la memoria del ordenador) de una clase.

El fenómeno de crear objetos de una clase se llama instanciación.

Los objetos se manipulan con referencias. Una referencia es una variable que apunta a un objeto que se encuentra en la memoria. Las referencias se declaran igual que las variables de tipos primitivos.

Ejemplo:

```
Punto.kt x
1  class Punto {
2      var x: Int = 0
3      var y: Int = 0
4
5  }
```

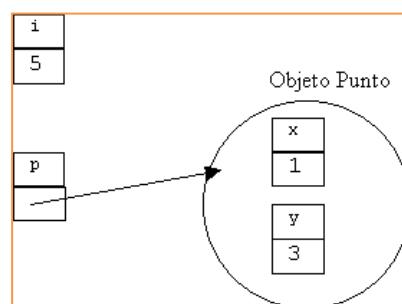
```
var p: Punto
var p = Punto()
```

La primera línea del ejemplo declara una referencia (p) que es de tipo Punto, en este caso la referencia no apunta a ningún sitio.

En la segunda línea se crea un objeto de tipo Punto y se hace que la referencia (p) apunte a él. Se puede hacer ambas operaciones en la misma expresión.

### Ejemplo:

```
var p = Punto ()  
  
p.x = 1;  
p.y = 3;  
  
var i = 5;
```



Es importante señalar que, en el ejemplo, p no es el objeto. Es una referencia que apunta al objeto.

Los métodos miembros se declaran dentro de la declaración de la clase.

Para acceder a los métodos o atributos miembro, lo haremos a través de la referencia.

### Ejemplo:

```
Circulo.kt x
1 class Circulo {
2     var centro = Punto()
3     var radio: Int = 0
4
5     fun superficie() : Double {
6         return 3.14 * radio * radio
7     }
8 }
```

**Los métodos accesorios (getters y setters):** Los *setters* permiten acceder a los atributos de una clase para modificarlos, mientras que los *getters* permiten acceder a los mismos para leerlos.

En Kotlin, los *getters* y *setters* se generan automáticamente.



¿Cómo funcionan los getters y los setters? Lo primero, es saber que el siguiente código:

```
class Person {  
    var name: String = "noname"  
}
```

Es equivalente a:

```
class Person {  
    var name: String = "noname"  
    // getter  
    get() {  
        return field  
    }  
    // setter  
    set(value) {  
        field = value  
    }  
}
```

Nota: Como puedes ver, dentro de los *getters* y *setters* se utiliza *field*. Esta variable *field* es especial y se refiere/apunta al campo. Se usa *field*, para evitar recursividades, ya que si pusiera *this.name* se llamaría al método *get/set*.

	<p style="text-align: center;"><b>CFGS DAM</b></p> <p style="text-align: center;"><b>Tema 0: Introducción a Kotlin</b></p>	
---	--	---

Cuando creamos una instancia del objeto de la clase *Person* e inicializamos la propiedad *name*, internamente se llama al método *set* estableciendo el valor en el campo, debido al código: *field = value*

```
val p = Person()
p.name = "jack"
println("${p.name}")
```

Ahora, cuando accedemos a la propiedad *name* del objeto, internamente se llama al método *get*, obteniendo el campo debido al código: *return field*

```
println("${p.name}")
```

Por tanto, para acceder a los datos miembros, podemos llamarlos directamente, gracias a la existencia por defecto de los métodos *getters* y *setters* (aunque no los veamos).



### Ejemplo:

```
var c = Circulo()  
c.centro.x = 2  
c.centro.y = 3  
c.radio = 5  
  
var s = c.superficie()
```

Es interesante observar en el ejemplo:

- Los datos miembros pueden ser tanto primitivos como referencias. La clase *Circulo* contiene un dato miembro de tipo *Punto* (que es el *centro* del círculo).
- El acceso al dato miembro *centro* del tipo *Punto*, se hace encadenando el operador “.” en la expresión *c.centro.x* que se podría leer como 'el miembro *x* del objeto (*Punto*) centro del objeto (*Circulo*) *c*'.
- Aunque el método *superficie* no recibe ningún argumento los paréntesis son obligatorios (distinguen los datos de los métodos).
- Existe un Objeto *Punto* para cada instancia de la clase *Circulo* (que se crea cuando se crea el objeto *Circulo*).

También, debemos saber que los *getters* y *setters* en Kotlin heredan el mismo acceso que las propiedades. Así que, si defines una propiedad como privada, el *getter* y *setter* también serán privados y esto no puede ser cambiado.

Tenemos que tener en cuenta, que estos métodos *getters* y *setters* los podemos modificar, podemos “sobreescribirlos”, para así acceder/modificar los datos como necesitemos, pero siempre teniendo en cuenta que los métodos *getters* son para recuperar y *setters* para modificar.

```
class Person {  
    var name: String = "noname"  
    // getter  
    get() {  
        return field  
    }  
    // setter  
    set(value) {  
        field = value  
    }  
    var age: Int = 0  
    // getter  
    get() {  
        return field  
    }  
    // setter  
    set(value) {  
        if (value < 18) field = 18  
        else if (value >= 18 && value <= 30) {  
            field = value  
        } else field = value-3  
    }  
}
```

## Conceptos básicos. Resumen

- Una Clase es una definición de un nuevo tipo, al que se da un nombre.
- Una Clase contiene Datos Miembro y Métodos Miembro que configuran el estado y las operaciones que puede realizar.
- Un Objeto es la materialización (instanciación) de una clase. Puede haber tantos Objetos de una Clase como resulte necesario.
- Los Objetos se crean (se les asigna memoria).
- Los Objetos se manipulan con Referencias.
- Una Referencia es una Variable que apunta a un Objeto.
- El acceso a los elementos de un Objeto (Datos o métodos) se hace con el operador. (punto): *nombre\_referencia.miembro*

## Clases - Constructores

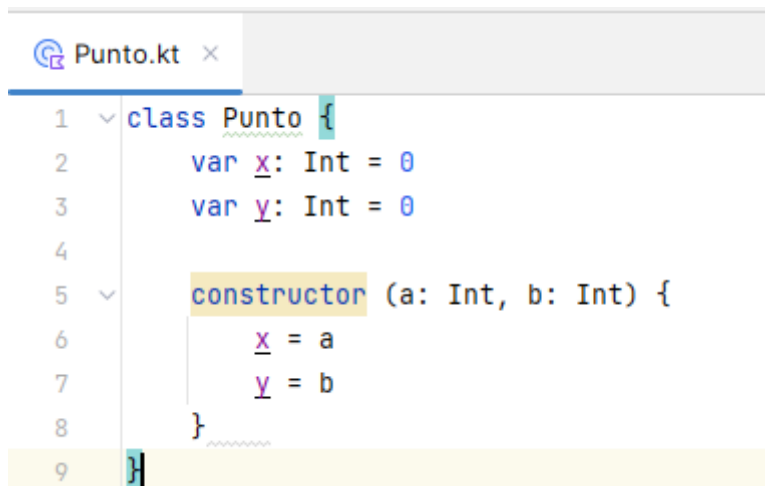
### Noción de constructor

Cuando se crea un objeto (se instancia una clase) es posible definir un proceso de inicialización que prepare el objeto para ser usado. Esta inicialización se lleva a cabo invocando un método especial denominado constructor. Los constructores tienen algunas características especiales:

- El nombre del constructor tiene que ser “constructor”.
- Puede recibir cualquier número de argumentos de cualquier tipo, como cualquier otro método.
- No devuelve ningún valor.

**El constructor no es un miembro más de una clase. Sólo es invocado cuando se crea el objeto. No puede invocarse explícitamente en ningún otro momento.**

Continuando con los ejemplos se podría escribir un constructor para la clase Punto, de la siguiente forma:



```
1 class Punto {  
2     var x: Int = 0  
3     var y: Int = 0  
4  
5     constructor(a: Int, b: Int) {  
6         x = a  
7         y = b  
8     }  
9 }
```

Con este constructor se crearía un objeto de la clase Punto de la siguiente forma. A diferencia de otros lenguajes, no usas la palabra *new* para crear el objeto, sino que llamas al constructor como una función normal. Como puede verse, se usa el nombre de la clase para realizar la llamada:

```
var p = Punto (1, 2)
```

### Constructor no-args.

Si una clase no declara ningún constructor, Kotlin incorpora un constructor por defecto (denominado constructor no-args) que no recibe ningún argumento y no hace nada. En Java se les denomina constructor vacío.

Si se declara algún constructor, entonces ya no se puede usar el constructor no-args. Es necesario usar el constructor declarado en la clase.

En el ejemplo el constructor no-args sería:

```
Punto.kt x
1  class Punto {
2      var x: Int
3      var y: Int
4
5      constructor () {
6          x = 0
7          y = 0
8      }
9  }
```

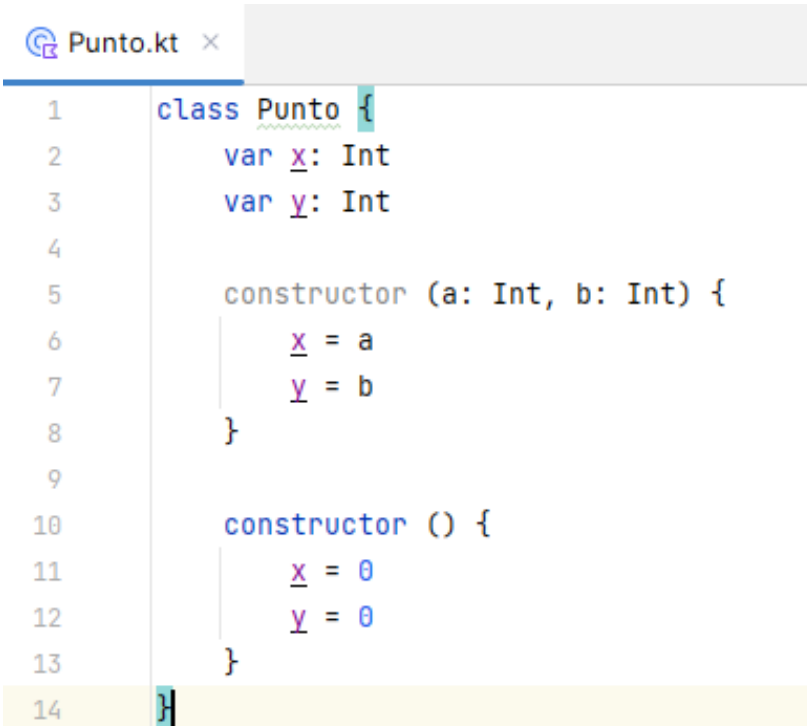
### Sobrecarga de constructores.

Una clase puede definir varios constructores (un objeto puede inicializarse de varias formas). Para cada instanciación se usa el que coincide en número y tipo de argumentos. Si no hay ninguno coincidente se produce un error en tiempo de compilación.

Cuando se declaran varios constructores para una misma clase estos deben distinguirse en la lista de argumentos, bien en el número, o bien en el tipo.

Esta característica de definir métodos con el mismo nombre se denomina sobrecarga y es aplicable a cualquier método miembro.

### Ejemplo:



```
1  class Punto {  
2      var x: Int  
3      var y: Int  
4  
5      constructor (a: Int, b: Int) {  
6          x = a  
7          y = b  
8      }  
9  
10     constructor () {  
11         x = 0  
12         y = 0  
13     }  
14 }
```

En el ejemplo se definen dos constructores. El citado en el ejemplo anterior y un segundo que no recibe argumentos e inicializa las variables miembro a 0.

En Kotlin, puedes usar la palabra reservada `this`. Es una referencia implícita que tienen todos los objetos y que apunta a sí mismo. Puedes ver el uso más habitual de `this`, en el siguiente ejemplo:

```
Punto.kt x
1  class Punto {
2      var x: Int = 0
3      var y: Int = 0
4
5      constructor (x: Int, y: Int) {
6          this.x = x
7          this.y = y
8      }
9  }
10
```

Usando `this` como prefijo, podemos referirnos a las propiedades de la clase, tal que podemos usar `this` para resolver ambigüedades con variables locales que tienen el mismo nombre que una propiedad de la clase.

En Kotlin, un constructor puede delegar en otro, usando `this`

```
Punto.kt x
1  class Punto {
2      var x: Int = 0
3      var y: Int = 0
4
5      constructor (x: Int, y: Int) {
6          this.x = x
7          this.y = y
8      }
9
10     constructor(): this(x: 0, y: 0)
11 }
12
```

### *Constructor Primario y Secundario*

El constructor primario es el constructor principal de una clase y se define dentro de los paréntesis en la declaración de la clase. Puede tener parámetros y estos parámetros serán a su vez atributos de la clase, usándose el constructor primario para inicializar dichos atributos.

Los constructores secundarios son otros constructores que pueden tener diferentes parámetros y se utilizan para proporcionar diferentes maneras de crear una instancia de una clase. Si existe un constructor primario, cada constructor secundario **debe** llamar al constructor primario usando la palabra clave `"this"`.

**Ejemplo** de una clase con un constructor primario y un constructor secundario:

```
Person.kt x
1 class Person(var name: String, var age: Int) { //primario
2     constructor(name: String) : this(name, age: 0) { //secundario
3     }
4 }
```

La clase "Person" tiene un constructor primario con dos parámetros "name" y "age". También tiene un constructor secundario con un solo parámetro "name", que llama al constructor primario y proporciona un valor predeterminado para la edad.

### *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. Se usa para inicializar las propiedades de la clase y suelen escribirse en la clase, antes que los constructores secundarios.

```
Person.kt x
1 class Person(name: String) {
2     var name:String
3     var age:Int
4
5     init {
6         this.name = name
7         this.age = 0
8     }
9
10    constructor(name:String, age:Int):this(name){
11        this.age = age
12    }
13 }
```



### *Propiedades de lateinit en Kotlin*

El modificador `lateinit` te permite inicializar una propiedad **no anulable** 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.

#### **Ejemplo:**

```
lateinit var propiedad:String
```

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
- No deben tener accesores (`get` y `set`) personalizados
- No pueden declararse con tipos primitivos en Kotlin: enteros, flotantes, booleanos y caracteres.
- No pueden ser anulables

Para determinar si la propiedad fue o no inicializada, llama a `isInitialized` en la referencia de la propiedad sobre la que se usó `lateinit`.

### ***Propiedades lazy en Kotlin***

El modificador **lazy** te permite posponer la lógica de los accesorios de una propiedad.

Una *propiedad lazy* o *perezosa*, es aquella que su valor es calculado, a través de la función lazy.

Cuando usas el operador de acceso punto para obtener el valor de la propiedad, se ejecutará la lambda de inicialización y se establecerá su valor.

Al delegar el contenido de la propiedad con la función lazy {}, la primera llamada recordará el valor inicial, por lo que en la llamada 2 no habrá una segunda ejecución, y ya que solo tomará un valor al ejecutarse la lambda, debes usar val para la declaración.

Cuando la idea es inicializar solo una vez un elemento, declararlas como propiedades lazy es una buena opción.

#### **Ejemplo:**

```
val propiedadLazy by lazy{  
    /* lógica de accesor  
}
```

## Clases - miembros estáticos - object class

### Variables estáticas

Las variables estáticas son un tipo especial de variables que se definen dentro de un bloque *companion object* (en Java sería usando la palabra clave *static* delante del tipo y del nombre).

Cuando se define una variable estática, **solo** existe **una** copia del atributo para toda la clase y no una para cada objeto, tal que, si se cambia el valor desde un objeto, se cambia para todos, es decir, una vez que esta variable es creada, es compartida por todos los objetos o instancias de clase que se hayan creado. Esto es útil cuando se quiere llevar la cuenta global de algún parámetro.

### Ejemplo:

```
Punto.kt x
1 class Punto {
2     var x: Int = 0
3     var y: Int = 0
4
5     constructor(x: Int, y: Int) {
6         this.x = x
7         this.y = y
8         numPuntos++
9     }
10
11     constructor(): this(x: 0, y: 0)
12
13     //como seria en java -> static int numPuntos = 0;
14     companion object {
15         var numPuntos: Int = 0;
16
17         fun getEjemplo(): String {
18             return "función estática";
19         }
20     }
21
22     fun getNumPuntos(): Int {
23         return numPuntos
24     }
25 }
```

En el ejemplo, *numPuntos* es un contador que se incrementa cada vez que se crea una instancia de la clase *Punto*, y es compartida por todos los objetos de *Punto*.

El acceso a las variables estáticas desde fuera de la clase donde se definen se realiza a través del nombre de la clase y no del nombre del objeto como sucede con las variables miembro normales (no estática). En el ejemplo anterior puede escribirse:

```
var x: Int = Punto.numPuntos
```

No obstante, también es posible acceder a las variables estáticas a través de una referencia a un objeto de la clase y un método. Por ejemplo:

```
var p = Punto()  
x = p.getNumPuntos()
```

Las variables estáticas de una clase existen, se inicializan y pueden usarse antes de que se cree ningún objeto de la clase.

### Métodos estáticos

Para los métodos, la idea es la misma que para los datos: los métodos estáticos se asocian a una clase, no a una instancia, tal que para acceder a él no hay que crear un objeto y se accede usando el nombre de la clase, seguido de "." y a continuación, el nombre de la función estática. Para que una función sea estática, ha de crearse dentro del bloque *companion object*.

Los métodos que sean estáticos, nunca podrán acceder a atributos o invocar a otros métodos de la clase que no sean también estáticos, por eso deberán recibir como parámetros de entrada todos los datos necesarios para su funcionamiento.

```
companion object {  
    var numPuntos: Int = 0;  
  
    fun getEjemplo():String{  
        return "función estática";  
    }  
}
```

Accedería a la función, de la siguiente manera: `Punto.getEjemplo()`

Podemos usar los métodos estáticos para crear instancias de una clase determinada (uso de “factorías”):

**Ejemplo:**

```
enum class Modelo {  
    SINMODELO,  
    SEAT,  
    FERRARI,  
    VOLVO;  
}  
  
enum class Tipo {  
    SINDEFINIR,  
    BARCO,  
    COCHE,  
    BICICLETA;  
}  
  
class Vehiculo {  
    var tipo: Tipo = Tipo.SINDEFINIR  
    var modelo = Modelo.SINMODELO  
    var velocidad : Int = 0  
  
    constructor(tipo:Tipo, modelo:Modelo, velocidad:Int){  
        this.tipo = tipo  
        this.modelo = modelo  
        this.velocidad = velocidad  
    }  
  
    override fun toString(): String {  
        return "$tipo(modelo=$modelo, velocidad=$velocidad)"  
    }  
}
```

```
import kotlin.random.Random

class Factoria {
    //Una clase factoria se encarga de crear instancias de una clase determinada.
    //Es un patron de diseño
    companion object {

        fun devolverVehiculo():Vehiculo{
            var i = Random.nextInt( from: 1, until: 4)
            return when (i){
                1 -> {crearBarco()}
                2 -> {crearCoche()}
                3 -> {crearBicicletas()}
                else-> {crearIndeterminado()}
            }
        }

        fun crearBarco(): Vehiculo {
            var tipo=Tipo.BARCO
            var modelo= Modelo.entries.toTypedArray().random()
            var velocidad: Int = Random.nextInt( from: 1, until: 100)

            return Vehiculo (tipo,modelo,velocidad)
        }

        fun crearCoche(): Vehiculo {
            var tipo=Tipo.COCHES
            var modelo= Modelo.entries.toTypedArray().random()
            var velocidad: Int = Random.nextInt( from: 1, until: 100)

            return Vehiculo (tipo,modelo,velocidad)
        }

        fun crearBicicletas(): Vehiculo {
            var tipo=Tipo.BICICLETA
            var modelo= Modelo.entries.toTypedArray().random()
            var velocidad: Int = Random.nextInt( from: 1, until: 100)

            return Vehiculo (tipo,modelo,velocidad)
        }

        fun crearIndeterminado(): Vehiculo {
            var tipo=Tipo.SINDEFINIR
            var modelo= Modelo.SINMODELO
            var velocidad: Int = 0

            return Vehiculo (tipo,modelo,velocidad)
        }
    }
}

fun main(args: Array<String>) {
    //create 10 vehiculos aleatoriamente
    for (i in 1 .. 10){
        var v = Factoria.devolverVehiculo()
        println(v)
    }
}
```

## Object class

El patrón [Singleton](#) puede ser útil en varios casos, y Kotlin facilita la declaración de singletons.

En Kotlin, **los objetos se utilizan para crear instancias singleton o para agrupar funciones relacionadas**. Podemos acceder y utilizar sin esfuerzo los objetos de la instancia singleton sin necesidad de patrones singleton explícitos ni declaraciones estáticas.

Es útil utilizar objetos para mantener un estado global, proporcionar funcionalidad centralizada o gestionar operaciones de toda la aplicación.

Veamos un ejemplo:

```
PersonManager.kt x
1  object PersonManager {
2      private val personList = mutableListOf<Person>()
3
4      fun addPerson(person: Person) {
5          personList.add(person)
6      }
7
8      fun removePerson(person: Person) {
9          personList.remove(person)
10     }
11
12     fun getAllPersons(): List<Person> {
13         return personList.toList()
14     }
15 }
```

Declaramos un objeto para administrar una lista de personas, y ahora podemos administrar una lista de personas con un método estático:

```
fun main(){  
    val person1 = Person(name: "Ada", age: 31)  
    val person2 = Person(name: "Chris", age: 30)  
  
    PersonManager.addPerson(person1)  
    PersonManager.addPerson(person2)  
}
```



## Excepciones en Kotlin

La principal diferencia entre los mecanismos de excepciones de Kotlin y Java es que **todas las excepciones son no revisadas en Kotlin**. En otras palabras, no hay que declararlas de manera explícita en las firmas de función, como si se hace en Java.

En Kotlin, si se produce una excepción, por defecto es lanzada y se mostrará la traza del error por consola, y el código parará, sin necesidad de indicarlo en la firma a través de throws, tal que tampoco estás obligado a capturar las excepciones.

Esto no significa que las ignoremos, el manejo de errores debe seguir siendo parte de tu objetivo.

```
fun editFile(file: File, text: String) {  
    file.parentFile.mkdirs()  
    val fileOutputStream = FileOutputStream(file)  
    val writer = BufferedWriter(OutputStreamWriter(fileOutputStream))  
    try {  
        writer.write(text)  
        writer.flush()  
        fileOutputStream.fd.sync()  
    } finally {  
        writer.close()  
    }  
}
```

Hemos convertido el método editFile() a una función Kotlin. Puedes ver que la función no tiene la declaración throws IOException en su declaración de función, throws no es siquiera una palabra clave en Kotlin.

También, podemos llamar a esta función sin implementar el bloque try...catch y el compilador no genera un error, solo se usa si queremos hacer un tratamiento de la excepción, pero esto no es obligatorio para el compilador Kotlin.

```
try {  
    editFile(File(""), "text 123")  
} catch (e: IOException) {  
    e.printStackTrace()  
    throw e  
}
```

Un tratamiento que se puede hacer es convertir esta excepción en una propia, lo cual veremos cómo se hace un poco más adelante.

## Bloque try...catch

El constructor try con cláusulas catch y finally en Kotlin es similar al de Java.

```
fun foo() {  
    try {  
        throw Exception("Exception message")  
    } catch (e: Exception) {  
        println("Exception handled")  
        throw e  
    } finally {  
        println("inside finally block")  
    }  
}
```

Estamos creando un objeto Exception dentro del bloque try. Manejamos todas las subclases de tipo Exception en el bloque catch.

El bloque opcional finally siempre se ejecuta, es aquí donde típicamente cerramos cualquier recurso o conexiones que fueron abiertas previamente para prevenir fugas de recurso. Por ejemplo, si abres un archivo o te conectas a una base de datos o conexión de red en un bloque try, deberías cerrarlo o liberarlo en un bloque finally.

En Kotlin la cláusula **throw** es una expresión y puede combinarse con otras expresiones.

```
val letter = 'c'  
val result =  
    if (letter in 'a'..'z')  
        letter  
    else  
        throw IllegalArgumentException("A letter must be between a to z")
```

También, la cláusula **try** puede ser usado como una expresión.

```
fun foo(number: Int) {  
    val result = try {  
        if (number != 1) {  
            throw IllegalArgumentException()  
        }  
        true  
    } catch (e: IllegalArgumentException) {  
        false  
    }  
    println(result)  
}
```

Asignamos el valor devuelto desde el bloque try...catch a la variable result. Si el número no es 1, este lanza `IllegalArgumentException` y el bloque catch la captura, por lo que false será asignado a la variable result. Si el número es 1 en su lugar, entonces el valor de expresión true será asignado a la variable result.

Aunque sería mas correcto, implementar el código como se muestra a continuación:

```
fun foo(number: Int) {  
    val result = if (number != 1) {  
        false  
    }  
    true  
  
    println(result)  
}
```

## Java Interop

Las excepciones en Kotlin se comportan de manera normal en Java, existe una notación útil llamada **@Throws** en Kotlin. Debido a que todas las excepciones en Kotlin son no revisadas, los desarrolladores que utilizan tu código Kotlin desde Java podrían no ser conscientes de que tus funciones lanzan excepciones. Se puede alertar con la anotación **@Throw** que necesitan manejar la excepción. Veamos un ejemplo práctico de esta anotación.

```
fun addNumberToTwo(a: Any): Int {  
    if (a !is Int) {  
        throw IllegalArgumentException("Number must be an integer")  
    }  
    return 2 + a  
}
```

Definimos una función Kotlin que puede crear una excepción `IllegalArgumentException` solo si el tipo pasado a la función no es de tipo `Int`.

Nosotros llamamos a esta función de nivel superior `addNumberToTwo()` directamente desde Java de la siguiente manera:

```
public void myJavaMethod() {  
    Integer result = FunctionsKt.addNumberToTwo(5);  
    System.out.println(result); // 7  
}
```

Esto funciona bien; el compilador no detecta la posible excepción. Sin embargo, si queremos comunicar en Java que la función de nivel superior `addNumberToTwo()` crea una excepción, simplemente agregamos la anotación **@Throws** a la creación de la función.

```
@Throws(IllegalArgumentException::class)  
fun addNumberToTwo(a: Any): Int {  
    if (a !is Int) {  
        throw IllegalArgumentException("Number must be an integer")  
    }  
    return 2 + a  
}
```

Esta anotación **@Trows** puede aceptar una lista separada por comas de argumentos de excepción de clases. En el código de arriba, solo incluimos una clase de excepción (`IllegalArgumentException`).

Ahora tenemos que actualizar nuestro código Java para manejar la excepción.

```
public void myJavaMethod() throws IllegalArgumentException {  
    Integer result = FunctionsKt.addNumberToTwo(5);  
    System.out.println(result);  
}
```

## Creación de excepciones propias o personalizadas en Kotlin

Una excepción personalizada es una clase que extiende la clase `Exception` o una de sus subclases (por ejemplo, `RuntimeException`). Estas clases se utilizan para representar situaciones excepcionales específicas que pueden ocurrir en tu aplicación.

Definir excepciones personalizadas en Kotlin te permite gestionar de manera más efectiva situaciones excepcionales específicas en tu código, proporcionando información detallada sobre los errores y permitiendo un manejo más específico de las excepciones. Esto contribuye a una programación más robusta y mantenible

- Proporcionan información detallada sobre el error, lo que facilita la depuración.
- Permiten capturar y manejar errores de manera específica en lugar de usar excepciones genéricas.
- Ayudan a mejorar la legibilidad del código al identificar claramente las situaciones excepcionales

Para crear una excepción personalizada, sigue estos pasos:

1. Define una clase que herede de `Exception` o una de sus subclases.
2. Puedes agregar propiedades y métodos personalizados para proporcionar información adicional sobre la excepción.
3. Debes proporcionar un constructor que llame al constructor de la superclase y pase un mensaje descriptivo como parámetro.

```
class MiExcepcion : Exception("Se generó un cero")
```

Otra manera de crear tu propia excepción sería:

```
class MiExcepcion2 (message:String="Se generó un uno") : Exception(message)
```

Puedes lanzar una excepción personalizada utilizando la palabra clave `throw` seguida de una instancia de tu excepción personalizada.

```
fun miMetodo(): Int {  
    val random = (0..2).random() // Genera un número aleatorio entre 0 y 2  
    if (random == 0) {  
        throw MiExcepcion() // Lanza la excepción si es cero  
    }  
    else {  
        throw MiExcepcion2() // Lanza la excepción2 si es uno  
    }  
    return random // Devuelve el numero generado si no es 1 ni 0  
}
```

Para capturar una excepción personalizada, puedes utilizar un bloque **try-catch** y especificar el tipo de excepción que deseas capturar, igual que con cualquier excepción.

```
fun main() {  
    val miObjeto = MiClase()  
    try {  
        val resultado = miObjeto.miMetodo()  
        println("Resultado: $resultado")  
    } catch (e: MiExcepcion) {  
        println("Se produjo una excepción: ${e.message}")  
        throw e  
    } catch (e: MiExcepcion2) {  
        println("Se produjo una excepción: ${e.message}")  
        throw e  
    }  
}
```

En el ejemplo anterior, vemos como se pueden capturar más de una excepción, para lo cual, se crea un bloque catch por cada excepción.

Otra opción es la que se muestra a continuación:

```
fun main() {  
    val miObjeto = MiClase()  
    try {  
        val resultado = miObjeto.miMetodo()  
        println("Resultado: $resultado")  
    } catch (e: Exception) {  
        when(e){  
            is MiExcepcion -> {  
                e.printStackTrace()  
                throw e  
            }  
            is MiExcepcion2 -> {  
                e.printStackTrace()  
            }  
        }  
    }  
}
```