# ¿Qué es un algoritmo?

La palabra **algoritmo**proviene del sobrenombre de un matemático árabe del siglo IX, **Al-Khwarizmi**, que fue reconocido por enunciar paso a paso las reglas para las operaciones matemáticas básicas con decimales (suma, resta, multiplicación y división).

Una definición de algoritmo, es un grupo de órdenes consecutivas que presentan una solución a un problema o tarea. Algunos ejemplos de algoritmos los podemos encontrar en las matemáticas (como el algoritmo para resolver una multiplicación) y en los manuales de usuario de un aparato (como una lavadora o una impresora).

Sin embargo, hoy en día se relaciona la palabra algoritmo con el mundo de la informática, más concretamente en la programación; los conocidos como **algoritmos informáticos**.

Un algoritmo informático es una secuencia de instrucciones finitas que llevan a cabo una serie de procesos para dar respuesta a determinados problemas. Es decir, un algoritmo informático resuelve cualquier problema a través de unas instrucciones y reglas concisas, mostrando el resultado obtenido.

Los algoritmos son muy importantes en el mundo de la informática ya que permiten al programador resolver el problema antes de escribirlo en un lenguaje de programación que entienda la máquina u ordenador. Antes de escribir el código de un programa hay que resolver con un algoritmo el problema que se nos plantea.

Entendemos por un **programa informático** a una serie compleja de algoritmos escritos en un lenguaje de programación que pueden ser ejecutados en un ordenador.

La actividad de expresar un algoritmo en forma de programa se le denomina **Programación** o **Codificación**, y es parte fundamental en el **Desarrollo de aplicaciones**.

La razón principal para que las personas aprendan a programar, en general, y los lenguajes de programación en particular, es utilizar el ordenador como **herramienta para la resolución de problemas**.

En general, cualquier desarrollo lleva implícitas las siguientes fases que provienen del modelo de Programación Estructurada:

**Análisis**:

* **Definición** del problema
* Especificación de entradas
* Especificación de **salidas**

**Diseño de Algoritmos**:

* Diseño descendente: dividir en módulos o **subprogramas**
* **Refinamientos sucesivos**
* Uso de herramientas: **Diagrama de Flujo** y **Pseudocódigo**

**Resolución con ordenador**:

* **Codificación** (escritura en un lenguaje de programación)
* Ejecución
* Pruebas

|  |  |
| --- | --- |
|  | **Enseñemos a los niños a codificar | Mitch Resnick (16 min)**  <https://www.ted.com/talks/mitch_resnick_let_s_teach_kids_to_code?language=es&subtitle=es> |

Mitch dice que la codificación no es sólo para genios informáticos, sino que es de utilidad para todas las personas. En una charla divertida y llena de demostraciones, Resnick describe los beneficios de enseñar a los niños a codificar, para que puedan hacer algo más que "leer" las nuevas tecnologías, sino también crearlas.

# ¿Cómo hacer un algoritmo?

Todo algoritmo consta de tres partes:

* **Entrada**: en la entrada o input del algoritmo será donde se introduzcan todos aquellos datos que el algoritmo necesite para operar.
* **Procesamiento**: con lo recibido en la entrada, el algoritmo realizará una serie de cálculos lógicos para resolver el problema.
* **Salida**: los resultados obtenidos en el procesamiento se mostrarán en la salida o output del algoritmo.

Para hacer un algoritmo hay que tener en cuenta sus características:

* **secuenciales**, se procesan uno después del otro;
* **precisos**, deben ser objetivos al resolver el problema;
* **ordenados**, deben ser leídos y ejecutados de forma precisa;
* **finitos**, deben tener un número determinado de pasos;
* **concretos**, deben mostrar un resultado al problema resuelto;
* **definidos**, ante las mismas entradas siempre deben obtenerse los mismos resultados.

En resumen, un algoritmo debe ser suficiente para resolver el problema y ante varios algoritmos que resuelvan el mismo problema, siempre será preferible el que tenga un camino más corto.

# Kotlin

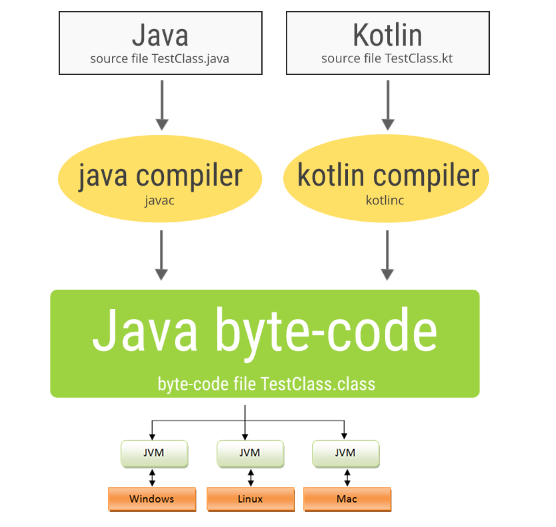
De una forma muy simplificada podríamos decir que un lenguaje de programación es un idioma que entiende el ordenador. Cualquier aplicación – procesador de textos, navegador, programa de retoque fotográfico, etc. - está compuesta de una serie de instrucciones convenientemente empaquetadas en ficheros que le dicen al ordenador de una manera muy precisa qué tiene que hacer en cada momento.

[Kotlin](https://kotlinlang.org/) es un lenguaje de programación que aparece en 2016, y fue diseñado por la empresa JetBrains[[1]](#footnote-1) 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 utilizar 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[[2]](#footnote-2), 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.

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).

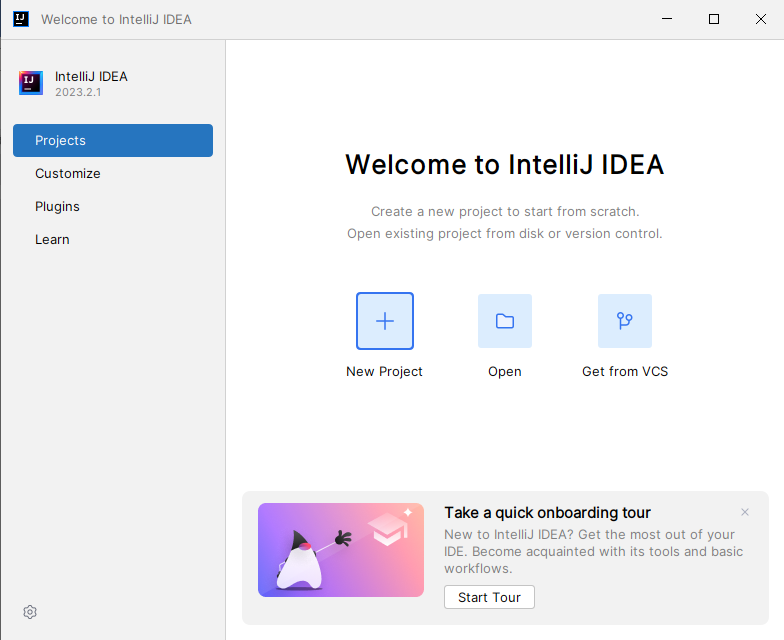


Cualquier editor simple es suficiente para escribir código en Kotlin aunque se recomienda el uso de un IDE, en concreto usaremos [**IntelliJ IDEA**](https://www.jetbrains.com/idea/download) en su versión gratuita **Community Edition**, el cual ya que tienen algunas características que facilitan mucho 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.

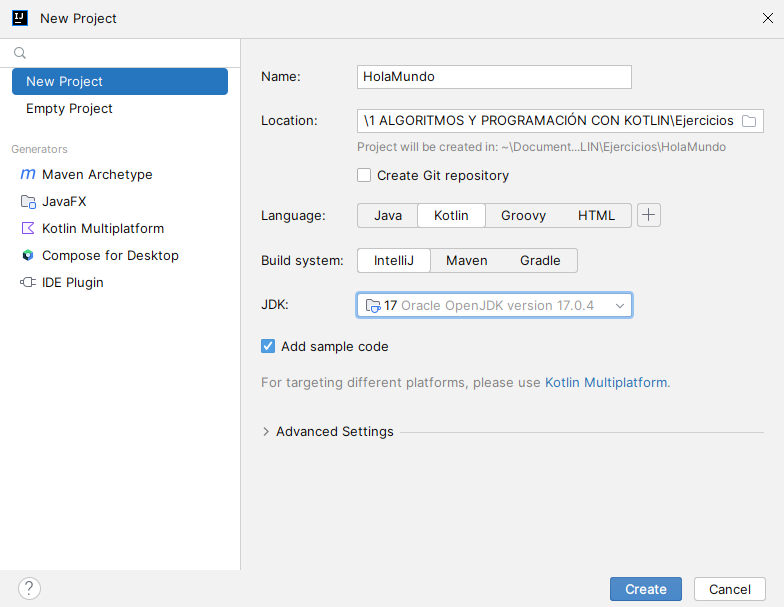
# ¡Hola mundo! - Mi primer programa

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:**



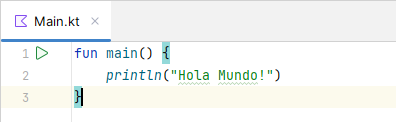
En la ventana de **New Project**, configúralo tal y como se muestra a continuación, y después pulsa el botón “**Create**”:



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:

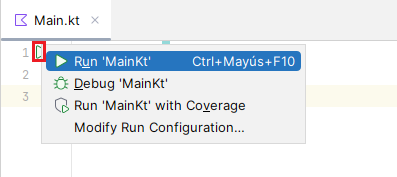


Las líneas del código anterior tienen el siguiente significado:

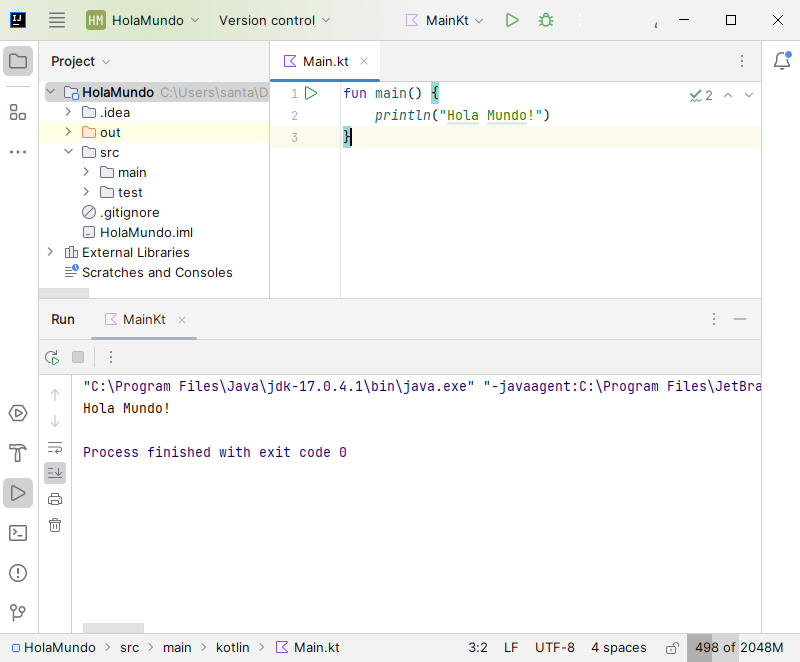
* La función main() es el punto de entrada de una aplicación Kotlin, ejecutando todas las instrucciones que pongas en su interior. Toma como parámetros los argumentos de línea de comandos en un array de strings.
* Las llaves, de apertura { y de cierre } especifican el inicio y el final del cuerpo de 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»**.



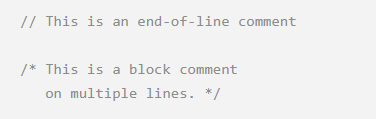
Al ejecutarse, verás en la ventana Run, como te aparece el mensaje: **Hola mundo**



# 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.

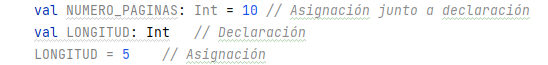


# Constantes

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.



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.

# 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](https://es.wikipedia.org/wiki/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*

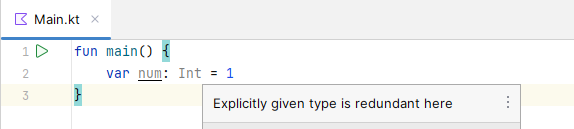
Si se declara una variable, es obligatorio, definir el tipo de datos de la variable, o bien asignarle un valor:

**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.



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**:

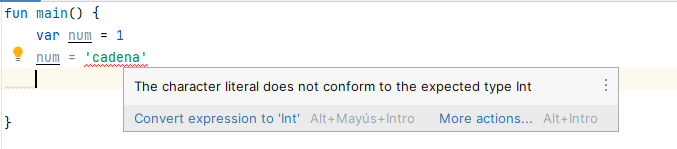
**var** num= 1 //se infiere entero

**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:



**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.

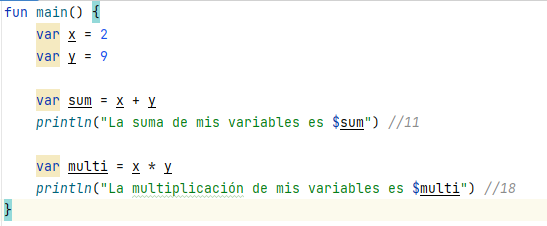
Es importante recalcar que una asignación no es una ecuación.

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.



**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 almacenan los siguientes datos?

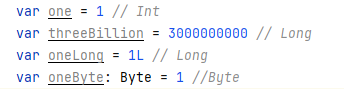
* El nombre de una ciudad 🡪 String
* El precio de un kilo de naranjas 🡪 Double
* El número de ruedas de un coche 🡪 Int
* Si es o no un usuario vip 🡪 Boolean
* Letra 🡪 Char

**Valores numéricos**

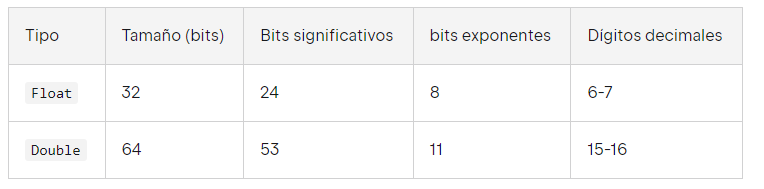
Para los **números enteros**, existen cuatro tipos con diferentes rangos de valores:



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.



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



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.



**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 ( ' ).



**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:



Mostraría por consola:

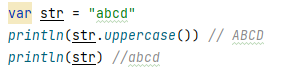


**Cadenas de Caracteres**

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



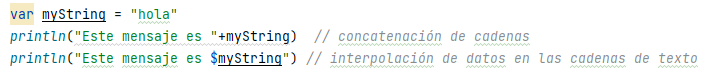
Las cadenas son inmutables, esto quiere decir, que una vez que se inicializa una cadena, no puede cambiar su valor. Todas las operaciones que transforman cadenas devuelven sus resultados en un nuevo objeto **String**, dejando la cadena original sin cambios.



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:

****

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:

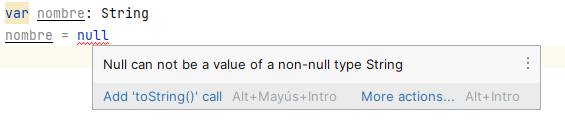


**Valores nulos en Kotlin**

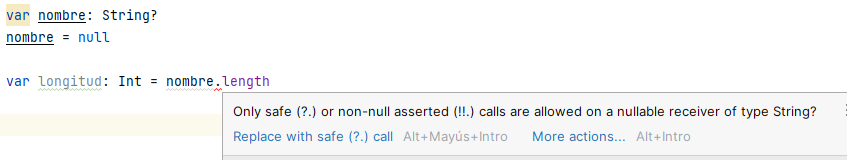
En Kotlin los objetos por defecto no aceptan valores nulos, para que le podamos asignar un null tendremos que indicar que ese objeto realmente puede ser null. De esta forma vamos a poder garantizar que no se no producirá un NullPointerException en tiempo de ejecución sin necesidad de llenar todo el código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

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.

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.



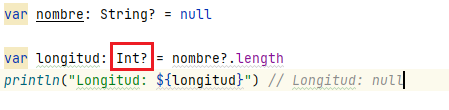
Si queremos que los objetos sí que sean nullables, para esto tenemos que añadir **?** después del nombre del tipo.



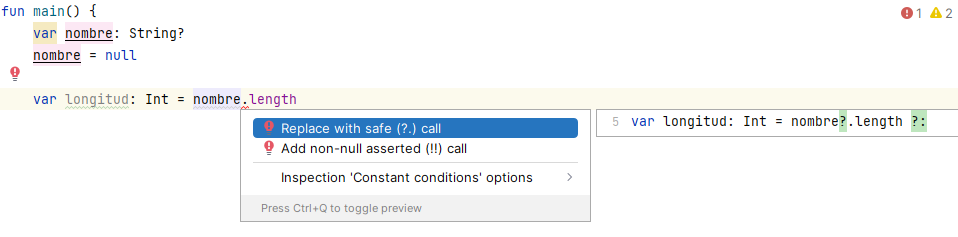
Pero Kotlin no nos deja pedir la longitud de la cadena porque puede ser nulo, y nos dice que tenemos 2 opciones:

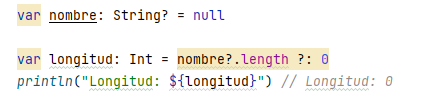
* ?. si queremos hacerlo de forma segura
* !!. si no queremos que el compilador compruebe si puede ser null.

Ejemplo, de cómo se haría de forma segura. Pero, si solo usamos **?.** siempre vamos a poder recibir un null y por lo tanto estamos propagando a los siguientes objetos o funciones la necesidad de soportar nulos:

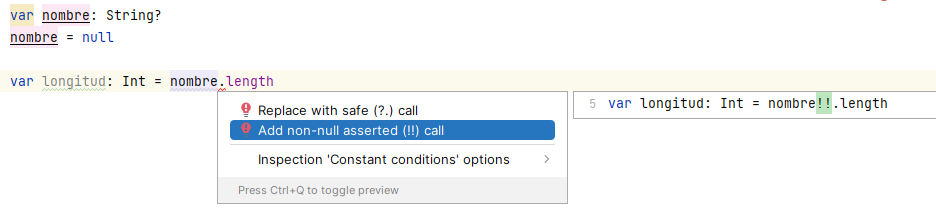


Para evitar esto, podemos usar el operador Elvis **?:** que nos permite definir un valor alternativo si se encuentra un null, por ejemplo para la longitud podríamos usar 0.





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…



**Alcance de las variables**

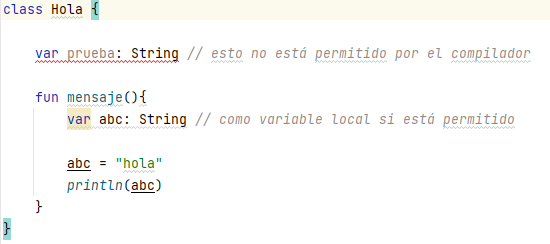
El alcance de las variables se refiere al ámbito en que éstas pueden ser utilizadas. Como regla general hay que considerar que una variable podrá ser utilizado sólo dentro del bloque de código donde ha sido declarada y nunca antes de hacerlo.

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

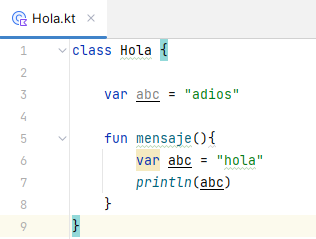
Vamos a hablar de variables globales y locales para explicar el alcance de éstas.

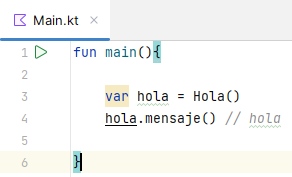
* Las variables locales son aquellas que se declaran en algún bloque del programa y perduran solamente dentro de ese bloque. Podrán ser, por tanto, utilizadas solamente dentro del bloque donde fueron declaradas.
* Las variables globales son aquellas que se declaran a nivel clase, al inicio del programa 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.



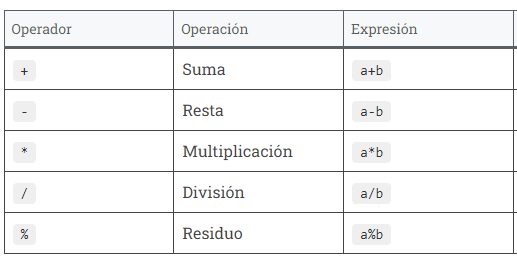
También debemos tener en cuenta, que dos variables no podrán llamarse de la misma manera, dentro del mismo ámbito. Una excepción a esta regla, es que las variables declaradas a nivel clase –**variables globales**– pueden coexistir con variables declaradas a nivel de método –**variables locales**–. Cuando 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.



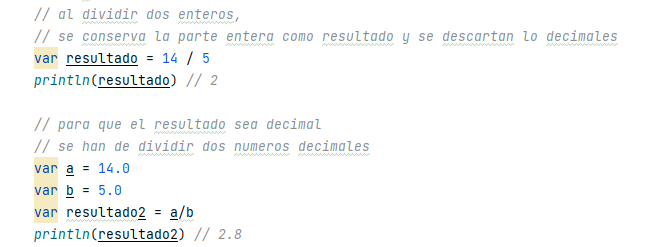


# Operadores

**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.

****

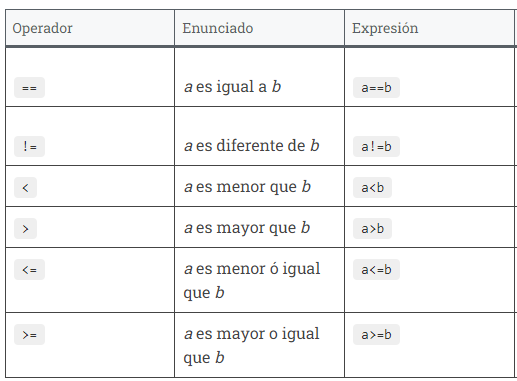
**Ejemplo:**

****

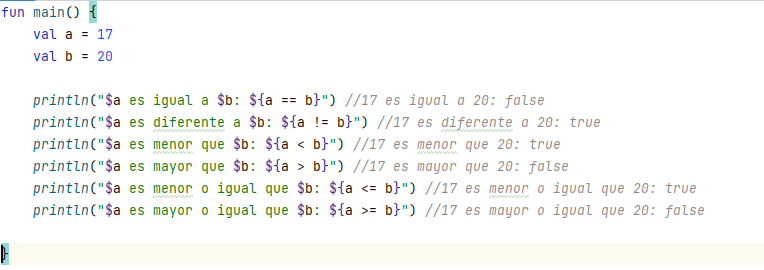
**Relacionales**: El resultado de una expresión relacional siempre es booleano, si se cumple será verdadero (true), si no se cumple será falso (false).

Los operadores relacionales o de comparación son:

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

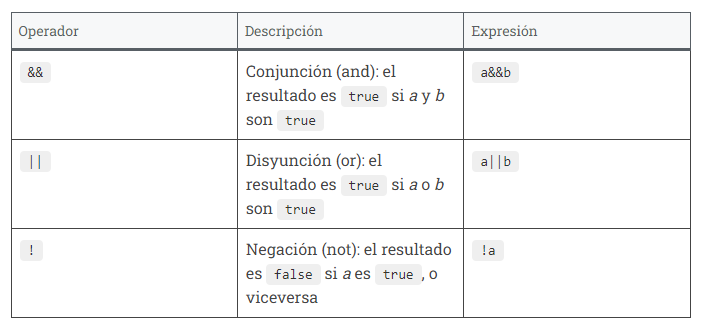


**Ejemplo:**

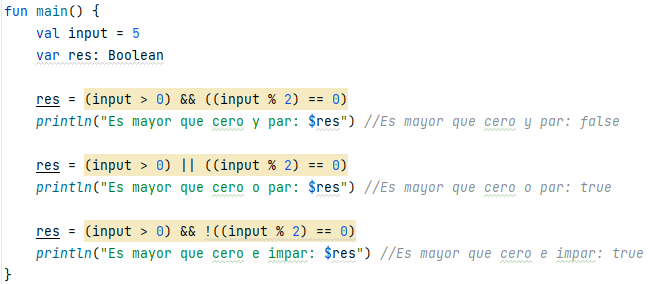


**Lógicos**: **&&** (es el AND), **||** (es el OR), **!** (es el NOT)

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)



**Ejemplo:**



**Operadores Unarios y de Asignación Compuesta**:

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

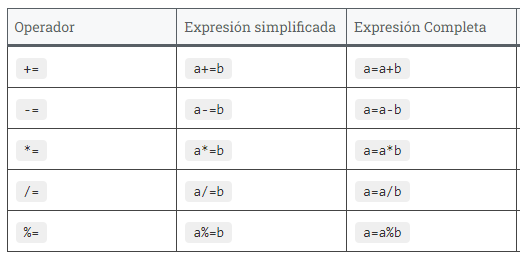
El operador de incremento, representado por dos signos de suma (++), incrementa en la unidad al operando.

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

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

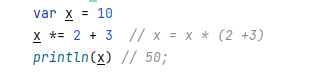
num**--**; // es equivalente a num = num - 1;

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.

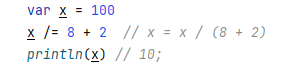


**Ejemplos:**

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



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



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

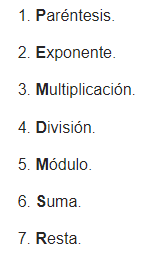


**Orden de evaluación de los operadores:** Es importante tener en cuenta el orden de evaluación de los operadores ya que, en algunos casos, definen el resultado de la operación. Pongamos algún ejemplo:

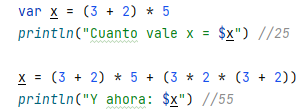
****

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.



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

****

# 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 linea), **print** (sin salto de linea) del paquete [kotlin.io](https://kotlinlang.org/api/latest/jvm/stdlib/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");

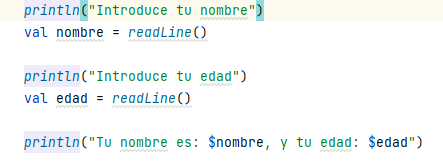
// Mensaje en una sola linea

**Lectura por consola**

Lee datos desde el teclado del usuario con la función **readLine(), l**a 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**:



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.

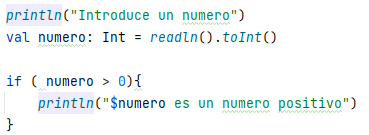
**if** (expresión-lógica) **{**

instrucción 1;

...

**}**

**Ejemplo:**

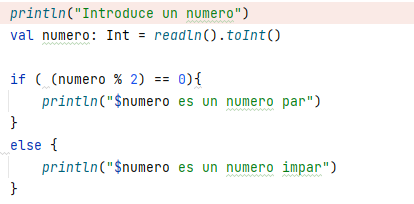


**Estructuras de control.**

**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:**

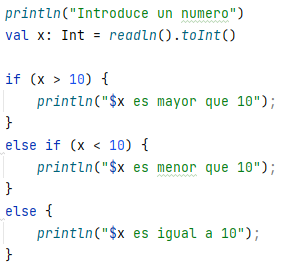
****

**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:**

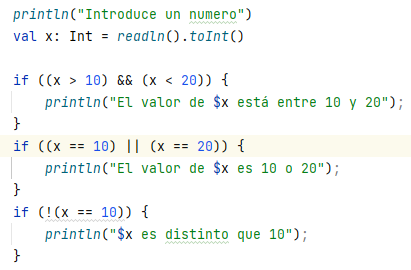
****

**Operador condicional**

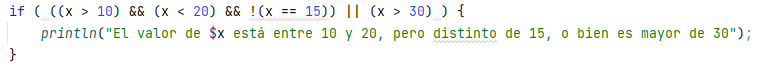
Los operadores condicionales permiten evaluar varias expresiones como si se trataran de una sola condición.

* 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**:



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



**Sentencia condicional múltiple: WHEN**

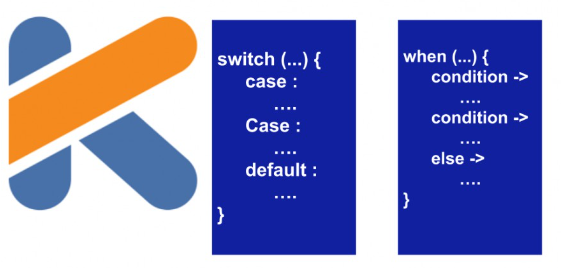
La expresión condicional **when**, te permite comparar el valor de un argumento contra una lista de entradas.

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.

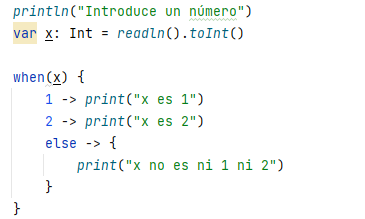
Sirve como sustituta de algunas expresiones de tipo if-else, considerándose análoga a la sentencia *switch* de Java.



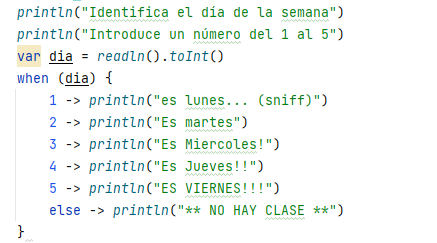
**Ejemplo 1: El argumento a analizar es una variable de tipo entero**

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”.

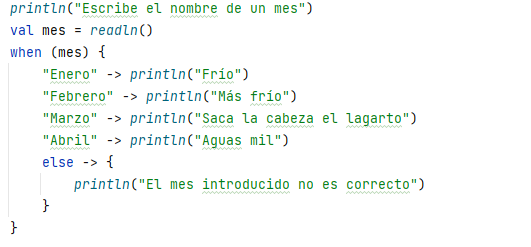
En este ejemplo, se está evaluando el valor de una expresión, concretamente el valor de una variable.



**Ejemplo 2:**

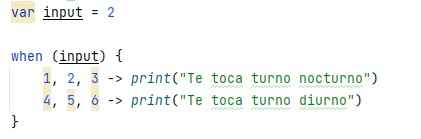
****

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

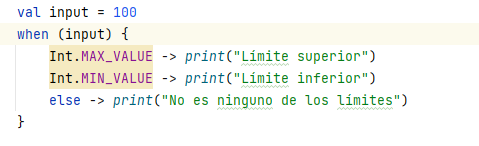


**Ejemplo 4: Múltiples valores en una condición**

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

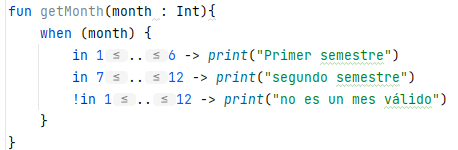
****

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

****

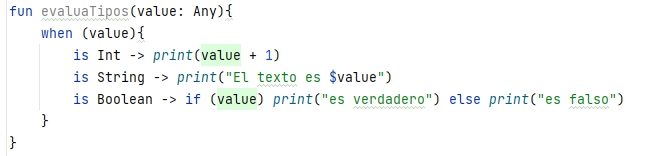
**Ejemplo 6: Uso de rangos de valores en la condición**

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*.

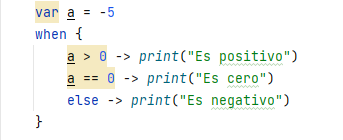
****

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

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

****

**Ejemplo 8: Usar When como un if**

****

# 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.

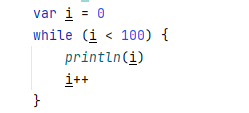
**while (**expresión lógica) **{**

instrucciones;

**}**

**Ejemplo 1: Ejemplo2:**

// Escribe los números del 1 al 99 // Bucle infinito, escribe “...” continuamente

****

**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**.

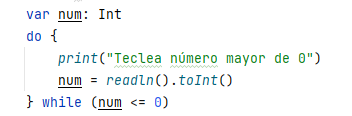
Esta sentencia es útil para la **validación** de datos de entrada.

**do {**

instrucciones;

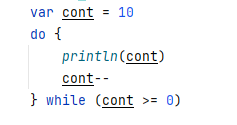
**}while (**expresión lógica);

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



**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 >= 0) es verdadero, se vuelve a realizar el bucle hasta que deja de cumplirse la condición.



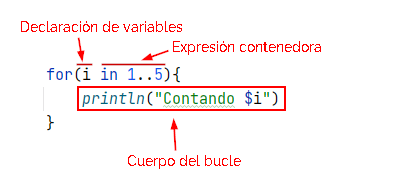
**Sentencia for**

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.

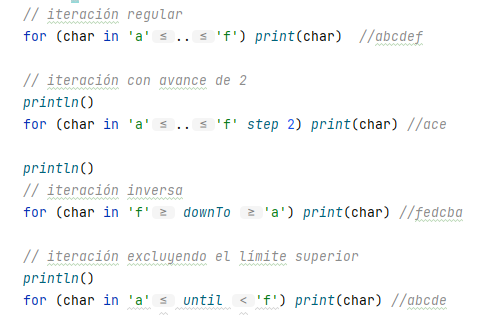
**Ejemplo 1**: Imprimir la cuenta del uno al cinco

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.



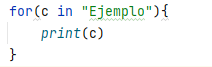
**Ejemplo 2**: Recorrer un rango

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



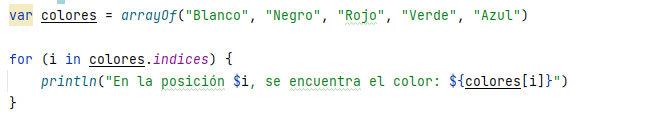
**Ejemplo 3**: Recorrer una cadena de caracteres

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

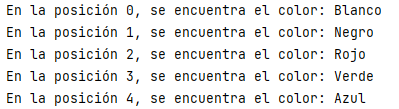


**Ejemplo 4**: Recorrer un array

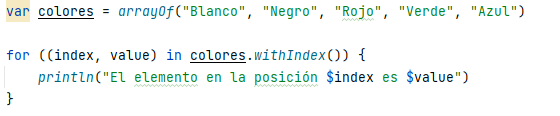
Para iterar a lo largo de un array con el bucle for , debemos usar como base los índices de sus elementos, que devuelven la posición de cada elemento contenido en el array.



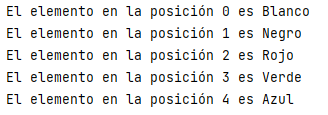
La salida será:



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



La salida será:



**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)

# Variables según su propósito

## Contadores

Un contador es una variable que se utiliza para **llevar la cuenta de cualquier suceso que se necesite** dentro de un programa.

En general se suele contar de forma natural desde 0 y de 1 en 1, aunque se pueden realizar otros tipos de cuenta, según lo requerido en el proceso.

Se utilizan realizando sobre ellos dos operaciones básicas:

* **Inicialización**. Todo contador se inicializa a 0 si realiza cuenta natural o a un valor inicial X si se desea realizar otro tipo de cuenta.

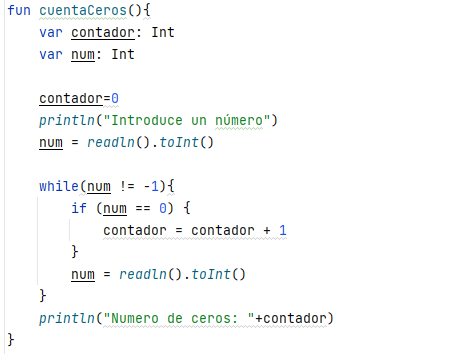
**contador** = ***0***

* **Contabilización o incremento**. Cada vez que aparece el evento a contar se ha de incrementar el contador en 1 si se realiza cuenta natural o en Y (Incremento) si se realiza otro tipo de cuenta.

**contador** = **contador *+ constante***

**Ejemplo:** Algoritmo que, dada una secuencia de números leída por teclado, que acabe con -1, y escriba el número de ceros introducidos.

Si el usuario introduce: 1 0 3 0 5 0 -1, Número de ceros será igual a 3



## Variables según su propósito: Acumuladores

Los acumuladores son variables que se utilizan en un programa para **acumular resultados parciales de cálculos con una misma operación**. En general se utilizan para calcular sumas y productos, sin descartar otros posibles tipos de acumulación.

Se utilizan realizando sobre ellos dos operaciones básicas:

* **Inicialización**. Todo acumulador necesita ser inicializado con el **valor neutro de la operación** que va a acumular, que en el caso de la suma es 0 y en el del producto es 1.
* **Acumulación**. Se utiliza una asignación:

Con la suma sería:

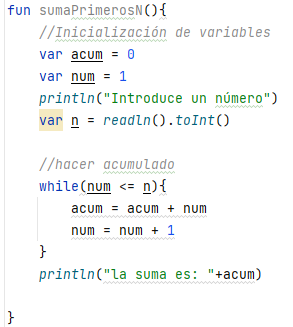
**acum** = **acum *+ variable***

Con el producto sería:

**acum** = **acum *\* variable***

**Ejemplo:** Algoritmo que suma los N primeros números naturales. Tal que, N es leído por teclado.

Si n=5, el resultado de la suma será igual a 15 (1 + 2 + 3 + 4 +5)



#### **Variables según su propósito: Interruptores (switches)**

Los interruptores son variables que se utilizan en un programa y que sólo pueden tomar dos valores (Verdadero (1) y Falso (0).

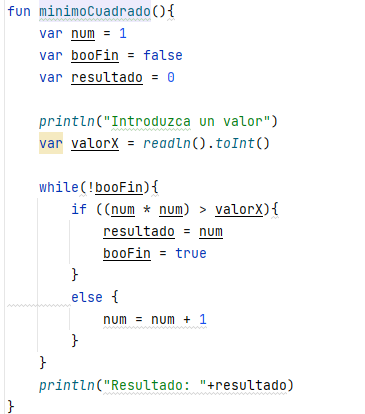
Se utilizan **inicializándolos con un valor** y en los puntos que corresponda, se **cambian al valor opuesto**, de tal forma que examinando su valor posteriormente, es posible detectar qué situación se ha dado con anterioridad.

Se suelen utilizar para finalizar bucles, habilitar/deshabilitar condiciones etc…

**Ejemplos:**

1. Algoritmo que calcule el número entero positivo cuyo cuadrado es superior a un valorX dado.

Si valorX=10, el resultado es 4, ya que 4\*4>10

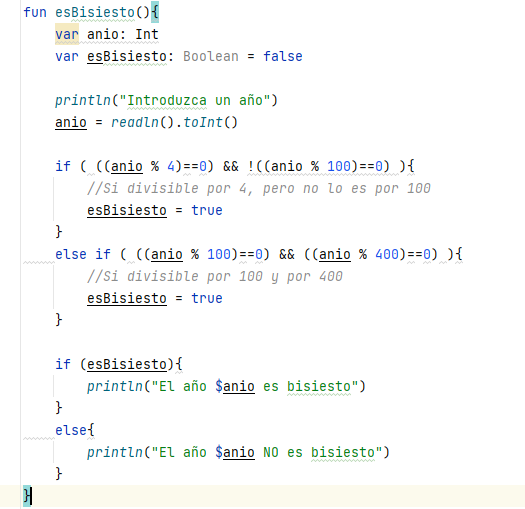
****

1. Algoritmo que, dado un año, nos dice si es bisiesto.

Utiliza una variable "interruptor" para anotar el estado y escribir el resultado al final.

El año 2012 es un año bisiesto

El año 2021 NO es un año bisiesto

****

# Funciones en Kotlin

En la POO, una **función** es un fragmento de código que realiza una determinada operación, y la podemos invocar mediante su nombre. Se define dentro de una clase. Es lo que se denomina en la programación estructurada **subprograma** o **método**.

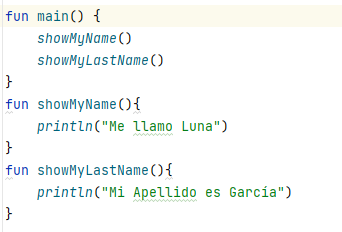
Las funciones se declaran usando la palabra clave **fun**, seguida del nombre, y entre paréntesis declararemos los valores de entrada (también llamados parámetros de entrada), pero también podemos tener funciones sin parámetros de entrada. Además, las funciones pueden devolver algún valor.

**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 es necesario incluir en una clase llamada **Main.kt** que implementa una aplicación, es decir, es una clase ejecutable. Este método **main**, será el invocado cuando se ejecute la clase.

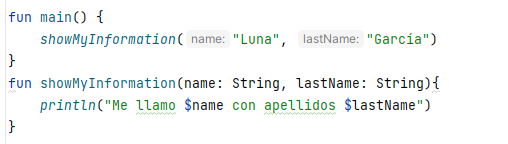
Por eso, si en el método **main**, llamamos a los métodos anteriores, haremos que se ejecute su código.



**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.

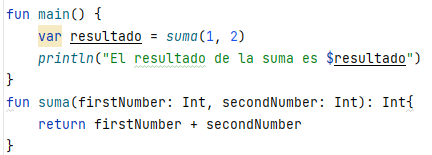


**Ejemplo de funciones que devuelven un valor**

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 **:** seguido 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.



1. JetBrains es la compañía creadora del IDE IntelliJ IDEA [↑](#footnote-ref-1)
2. 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. [↑](#footnote-ref-2)