



# UD2.1 – Fundamentos de Kotlin

**2º CFGS**  
**Desarrollo de Aplicaciones Multiplataforma**  
**2023-24**

# Importante de cara a todo el curso

Los nombres de variables, funciones, métodos, clases... deben escribirse en **inglés**.

Los nombres de los archivos también debe ser en **inglés**.

Se debe utilizar comentarios [KDoc](#) (como JavaDoc) al principio de cada archivo al menos con las etiquetas `@author` y `@versión`.

También se deben utilizar comentarios KDoc al principio de cada clase y de cada método.

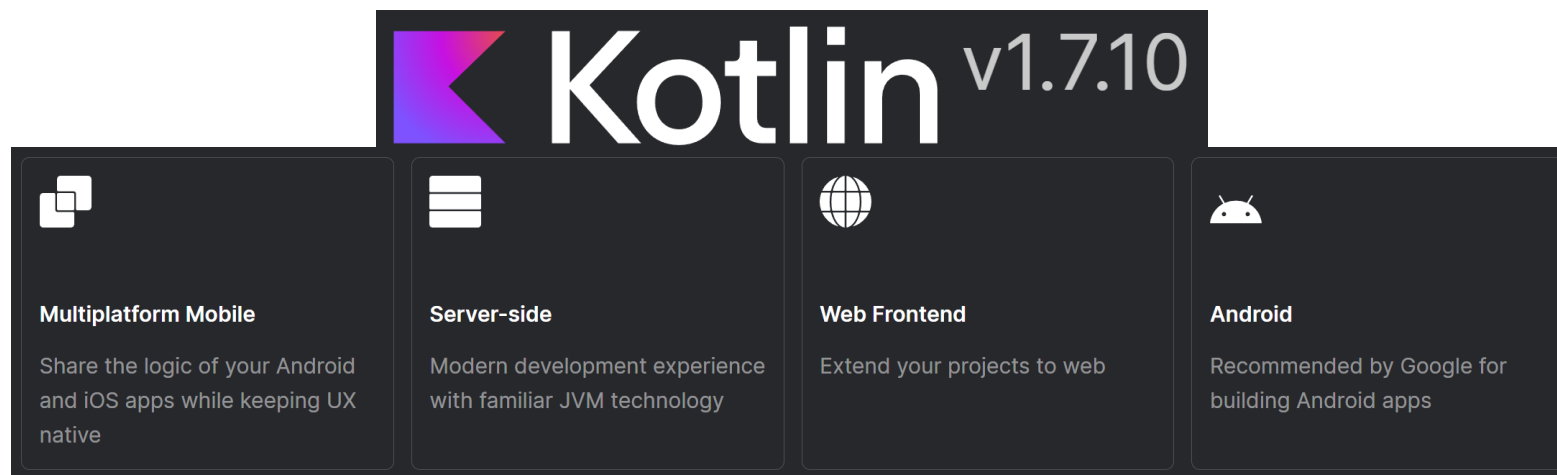
Evidentemente el código debe ser auto explicativo y en caso de haber elementos o algoritmos complejos en el código también se deben comentar.

# 1.- Introducción

En un principio **Java** fue el **lenguaje de programación para Android**.

En **2017** Google nombró a **Kotlin** como **lenguaje oficial para Android** equiparándolo con Java.

Kotlin tiene soporte oficial de Google y se incorpora en Android Studio.



# 1.- Introducción

## Algunas características de Kotlin:

- Lenguaje **multiplataforma** y **multipropósito**.
- Se **compila sobre JVM** por lo que es totalmente compatible con Java y sus librerías (llamadas a Java desde Kotlin y viceversa).
- Lenguaje orientado a objetos (**POO**).
- Es un **lenguaje conciso**, evita código innecesario (hasta un 40% menos).
- Es **Null Safety**, gestiona los nulos de forma segura y evitando errores NullPointerException.
- Extraoficialmente se puede decir que es como Java al estilo de Python (sus ventajas).

# 1.- Introducción

En la **documentación oficial** se puede encontrar todo lo necesario sobre el lenguaje Kotlin: <https://kotlinlang.org/docs/basic-syntax.html>

También está disponible una **plataforma para aprender Kotlin** mediante ejemplos: <https://play.kotlinlang.org/byExample/overview>

# 1.- Introducción

En el curso se usará **Kotlin** como lenguaje de programación para Android.

Al ser Kotlin un lenguaje multipropósito se pueden desarrollar aplicaciones Android, de escritorio, web, para consola...

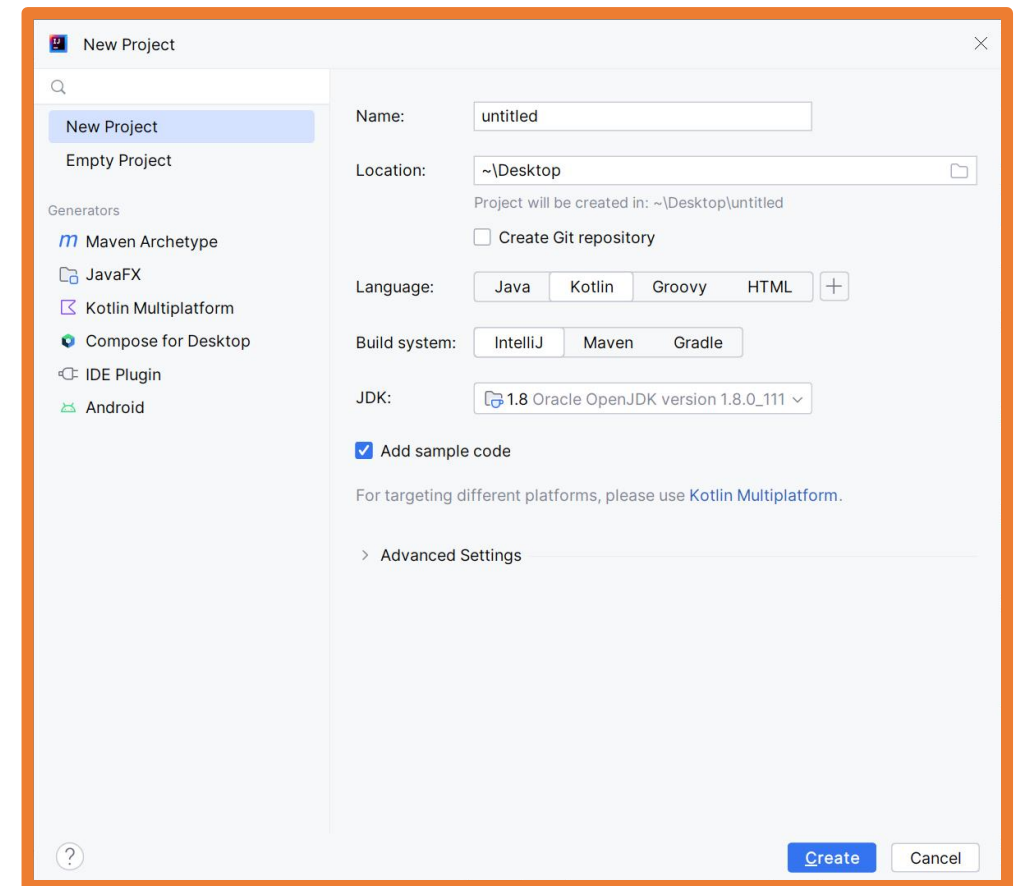
En esta unidad se verán los **fundamentos y la sintaxis de Kotlin** para más adelante **poder usar el lenguaje dentro de Android Studio** y desarrollar aplicaciones móviles Android.

Así, si se quieren crear **scripts** Kotlin y probarlos se debe tener un **JDK** de Java instalado y usar el entorno de desarrollo **IntelliJ IDEA**.

## 2.- Ejecutar código Kotlin

Teniendo un JDK de Java instalado en el ordenador, en **IntelliJ IDEA** se debe crear un proyecto con las siguientes opciones:

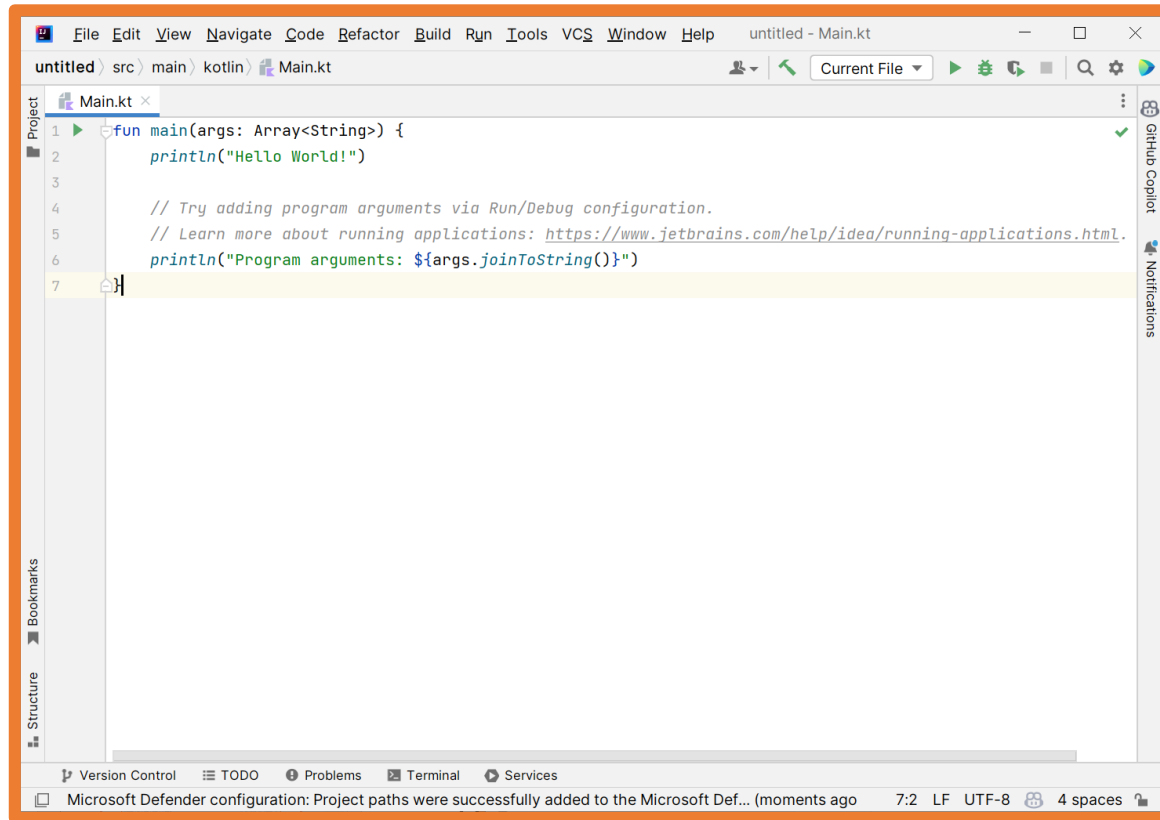
- Language: **Kotlin**
- Build system: **IntelliJ**
- **JDK**: la que se tenga instalada
- Add sample code: **marcada**



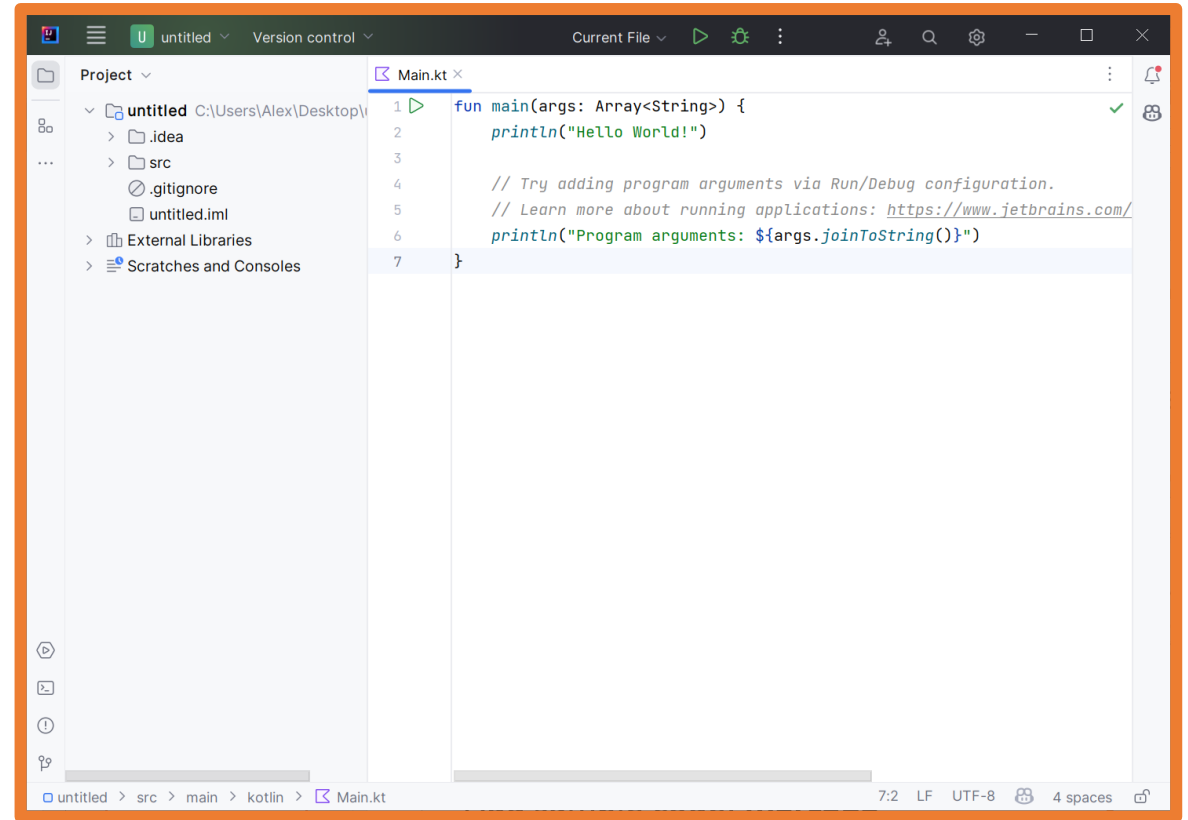
## 2.- Ejecutar código Kotlin

IntelliJ IDEA ha añadido una nueva interfaz, se puede usar cualquiera:

Antigua:



Nueva:

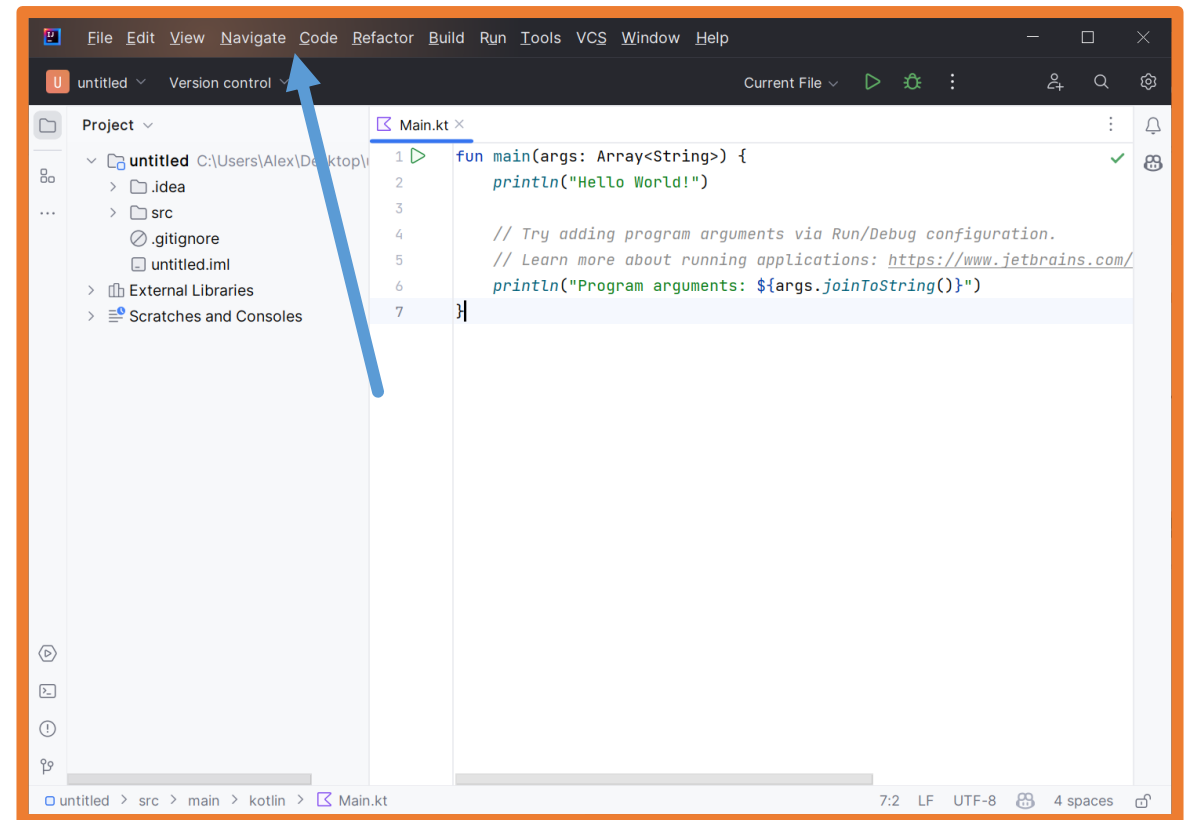
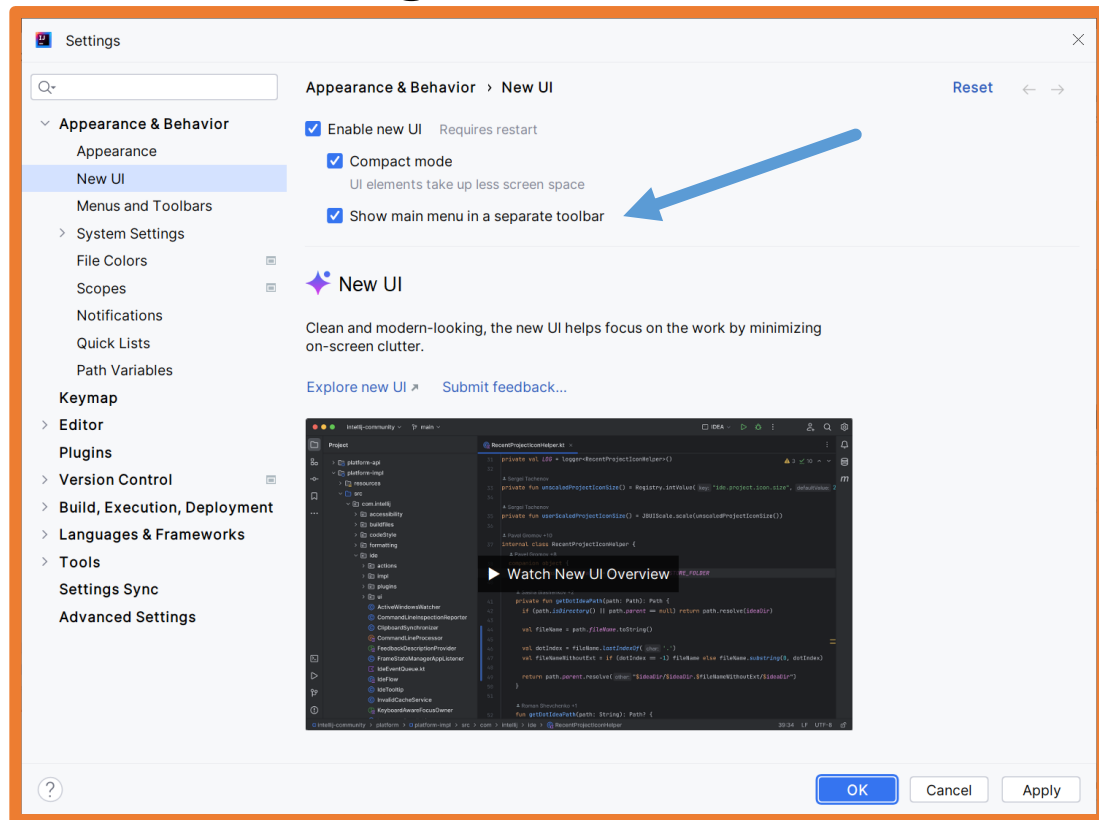




## 2.- Ejecutar código Kotlin

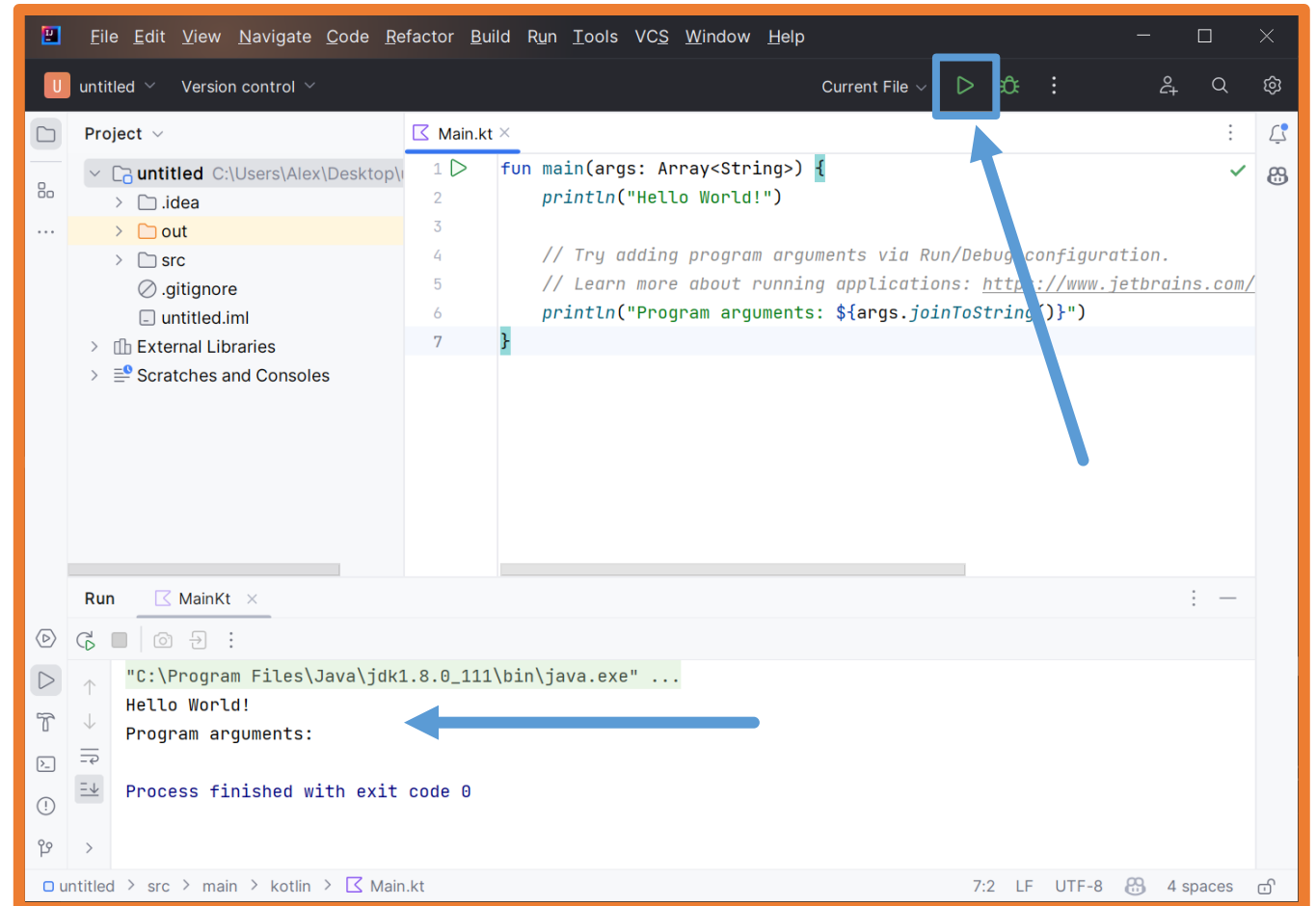
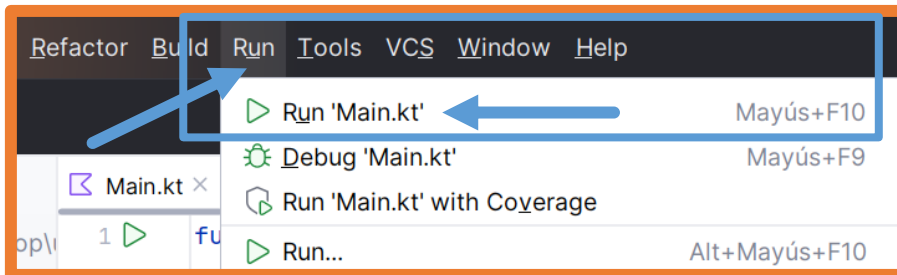
Si se quiere tener siempre visible la barra de menú en la nueva interfaz:

File → Settings



## 2.- Ejecutar código Kotlin

Para ejecutar un programa en IntelliJ IDEA hay varias formas de hacerlo.



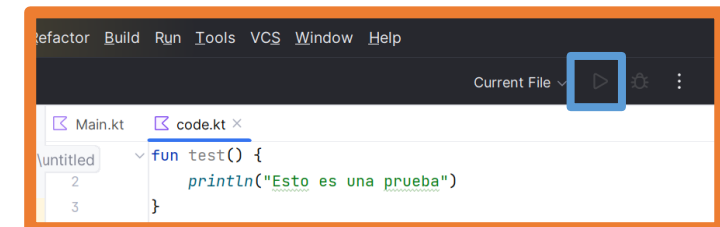
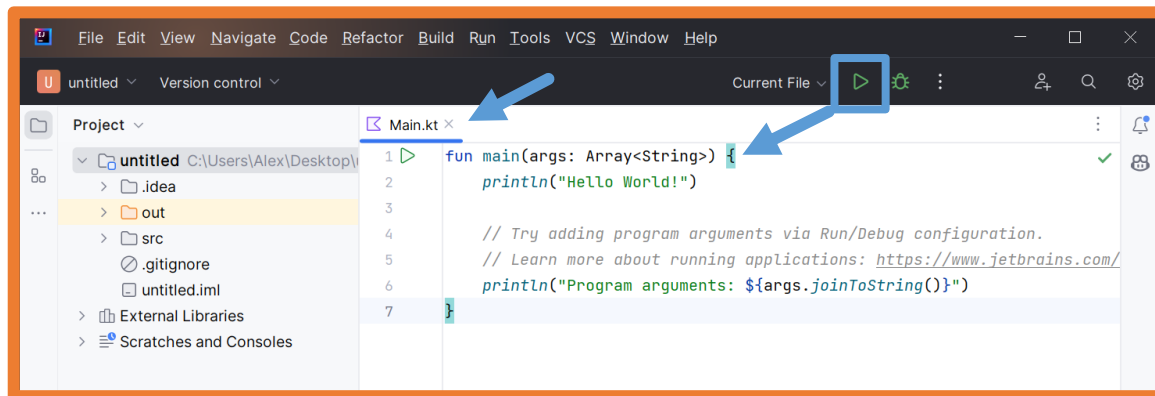
## 2.- Ejecutar código Kotlin

Al ejecutar un proyecto en Kotlin se ejecuta la función **main** del archivo que está activo.

Si hubiera más archivos con funciones con el nombre **main** solo se ejecutará la del archivo activo.

Si en un archivo no hay función **main** cuando esté activo el archivo no estará activo el icono **Run**.

En el ejemplo se ejecutará la función **main** del archivo **Main.kt**.



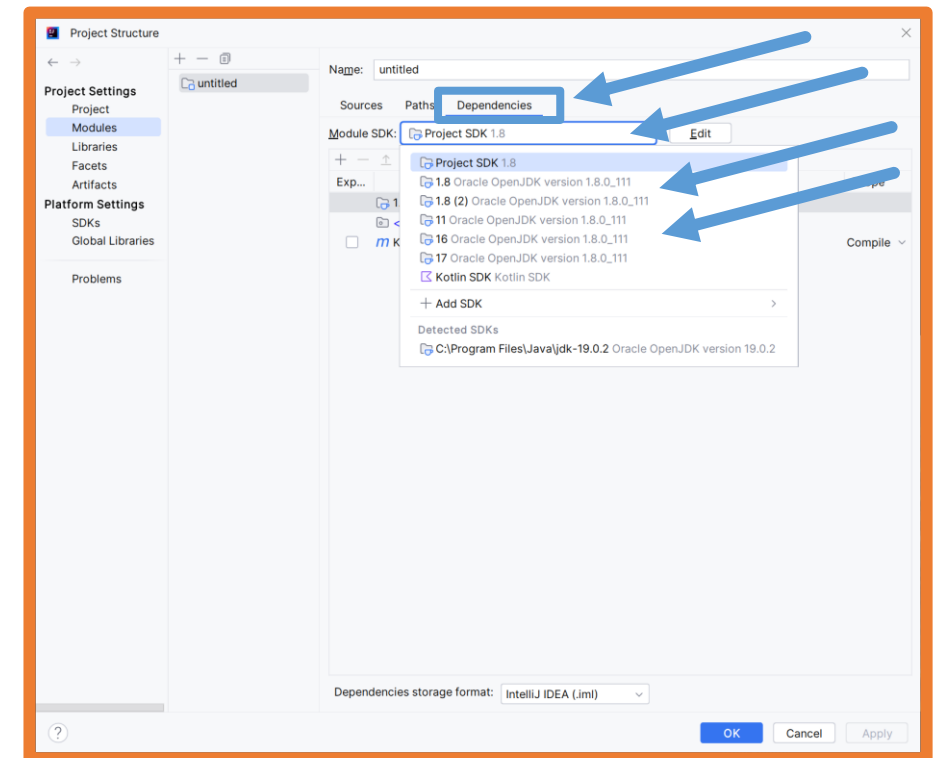
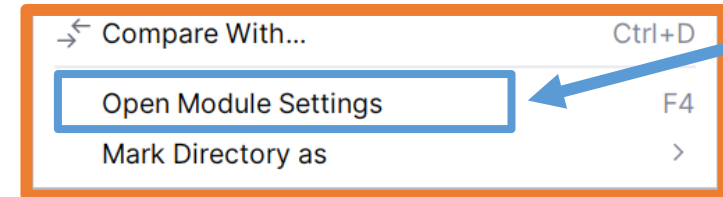
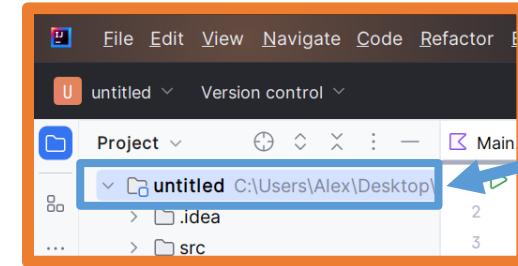
## 2.- Ejecutar código Kotlin

En ocasiones la opción **run** está desactivada, apareciendo el icono **play** de color gris.

Para solucionar esto se deben seguir los siguientes pasos:

- Encima de la carpeta del proyecto hacer clic derecho.
- Seleccionar la opción **Open Module Settings**.
- Seleccionar la pestaña **Dependencies**.
- En el apartado **Module SDK** cambiar la SDK seleccionada.

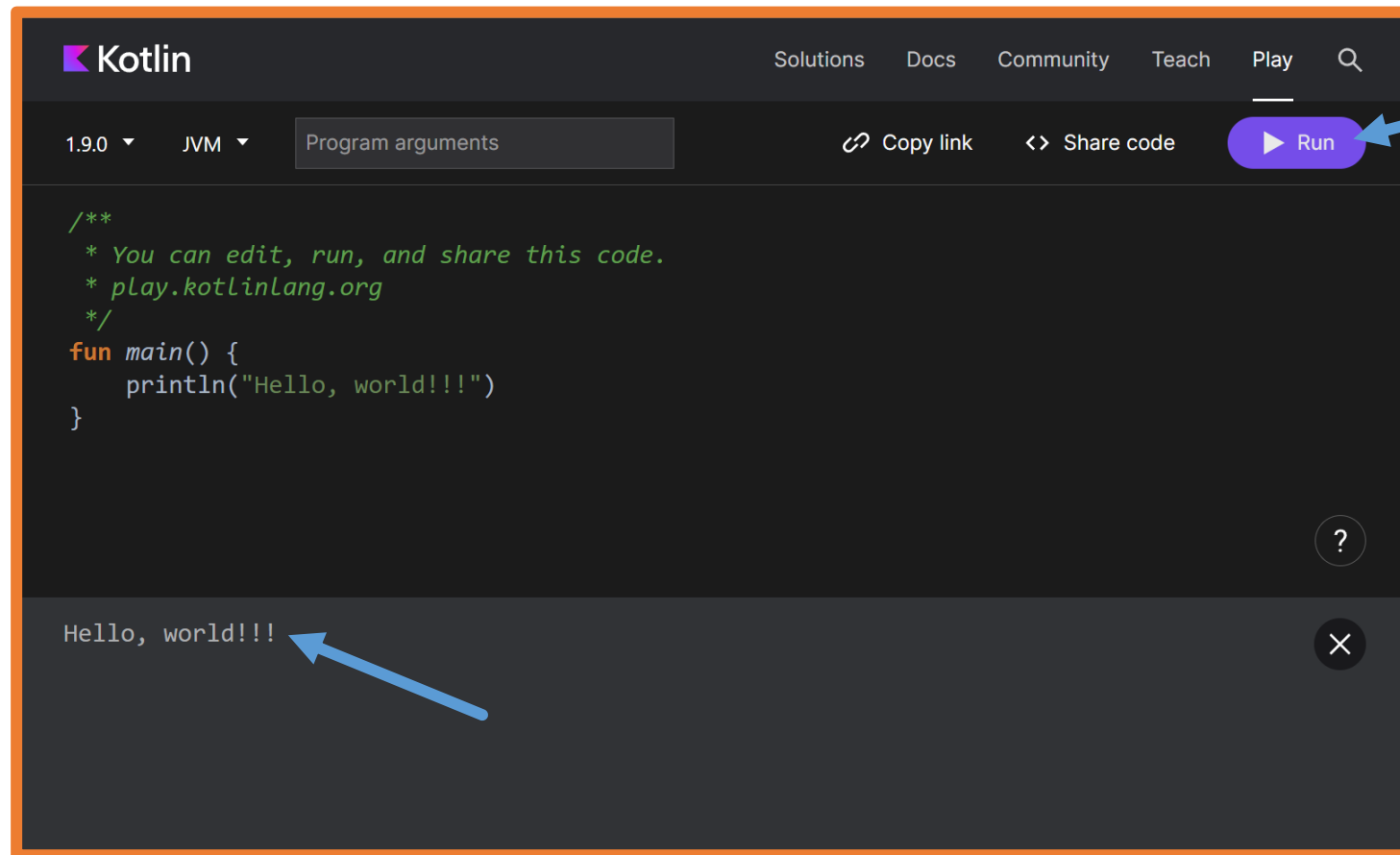
Ahora el icono **run** debería aparecer activado (verde).



## 2.- Ejecutar código Kotlin

También se pueden probar scripts de Kotlin en el enlace oficial:

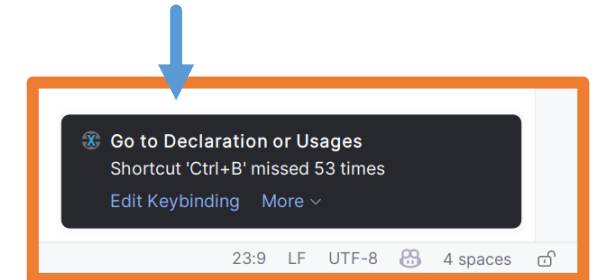
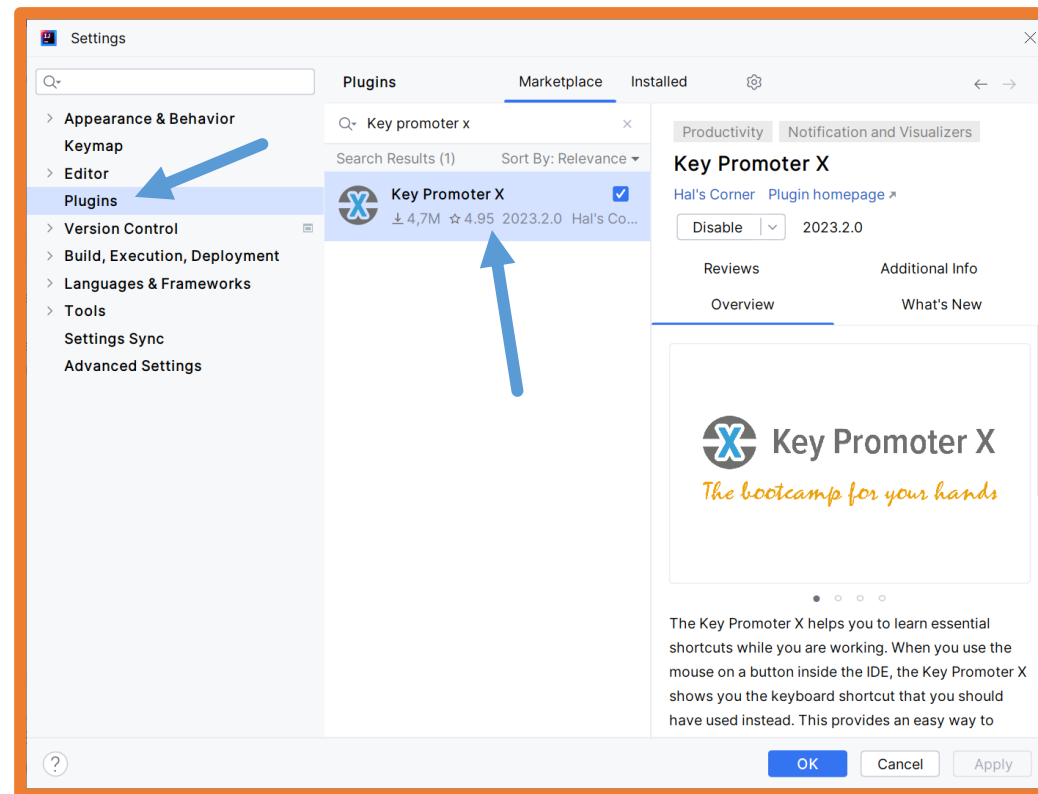
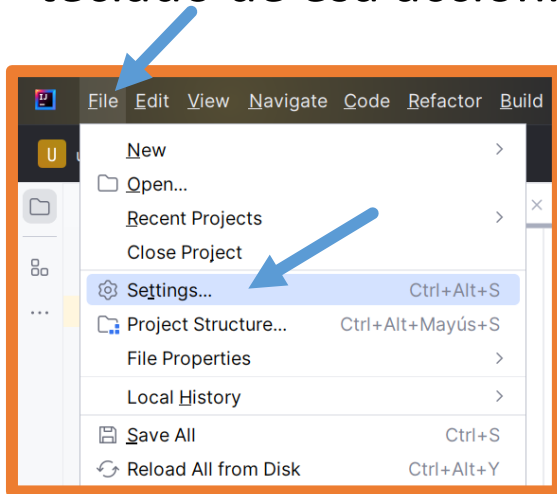
<https://play.kotlinlang.org/>



## 2.- Ejecutar código Kotlin

IntelliJ IDEA dispone de **plugins** para ayudar a los programadores y mejorar la productividad.

Por ejemplo, **Key Promoter X** cuando se usa el ratón muestra abajo a la derecha el atajo de teclado de esa acción.



# ¡Advertencia!

El curso pasado se estudió el módulo Programación.

Por esa razón **muchos conceptos se contemplarán como conocidos** en los ejemplos antes incluso de ver cómo se usan en Kotlin.

Entre ellos pueden estar:

- control de flujo (if, for...).
- funciones.
- programación orientada a objetos.
- los accesos a propiedades de objetos.
- las llamadas a funciones/métodos.
- excepciones.
- ...

### 3.- Guía de estilo Kotlin

En todo lenguaje de programación es importante que **el código sea legible**.

Las plataformas oficiales suelen crear guías de estilo para que los desarrolladores las sigan.

Es conveniente revisar dicha guía para desarrollar código correctamente.

También se puede usar la opción **Code → Reformat Code (Ctrl+Alt+L)** del IDE para así que se formatee el código de manera ajustada a las guías de estilo (siempre y cuando no se haya cambiado la configuración en el IDE).



## 4.- Saludo en Kotlin

Los archivos en Kotlin tienen la extensión **.kt**.

Al contrario que en Java **no requiere que todo sean clases**.

La función principal **main** no es necesario que se encuentre dentro de una clase.

**main** puede recibir argumentos pero no es necesario indicarlo.

**No es necesario indicar la visibilidad** public/static en la función.

La función **println** está disponible sin necesidad de usar System.out.

La función **readln** captura como cadena lo introducido en la consola por teclado.

El ; al final de cada instrucción es **opcional, por convención no debe usarse**.

Los **comentarios** tanto de línea como de bloque son como en Java: **//** y **/\* \*/**.

Greeting.kt

```
// Programa en Kotlin  
  
fun main() {  
    println("Hello there!")  
}
```

## 5.- Paquetes e importaciones

Como en Java, el paquete al que pertenece el archivo y las importaciones que se realicen se colocan en la parte superior del archivo.

```
package com.alextorres.hellothere

import android.support.v7.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

## 6.- Tipos de datos en Kotlin

### Numéricos

Se indican con las palabras reservadas **Int**, **Float**, **Long**, **Short**, **Byte** y **Double**

Decimales: **Double** (64 bits), **Float** (32 bits)

Enteros: **Long** (64 bits), **Int** (32 bits), **Short** (18 bits), **Byte** (8 bits)

Ejemplos de **literales**:

Tipo de número		Ejemplo
Entero	Int	154
	Long	154L
	Hexadecimal	0x8F
	Binario	0b00010110
Decimal	Double	154.8 1.548e2 (158x10 <sup>2</sup> )
	Float	154.8f 154.8F

## 6.- Tipos de datos en Kotlin

### Numéricos

Kotlin permite el uso del **carácter subrayado** (guion bajo) **en los literales numéricos** para facilitar la lectura de los mismos.

Ejemplos de **literales**:

2\_000\_000

44\_294\_051

6135\_8442\_0103\_56100



## 6.- Tipos de datos en Kotlin

### Caracteres

Se indican con la palabra reservada **Char**.

Se utiliza la codificación UTF-16 por lo que los literales se pueden indicar con el carácter o con su representación UNICODE entre comillas simples.

Ejemplos de **literales**:

'N'      →      '\u004E'  
'7'      →      '"\u0037'

No se pueden tratar como números.

## 6.- Tipos de datos en Kotlin

### Caracteres

Existen una serie de caracteres de escape

<code>\t</code>	Tabulación
<code>\b</code>	Retroceso
<code>\r</code>	Retorno de carro
<code>\n</code>	Salto de línea
<code>\'</code>	Apostrofe
<code>\"</code>	Comilla doble
<code>\\</code>	Backslash
<code>\\$</code>	Símbolo de dólar
<code>\u+XXXX</code>	Símbolo Unicode (4 dígitos hexadecimales)

## 6.- Tipos de datos en Kotlin

### Booleanos

Se indican con la palabra reservada **Boolean**.

Los valores posibles son verdadero y falso y sus **literales**:

true

false

## 6.- Tipos de datos en Kotlin

### Cadenas

Se indican con la palabra reservada **String**.

Los literales de cadenas **se indican con comillas dobles**:

"Hello there!"

"Rick Sanchez"



## 6.- Tipos de datos en Kotlin

### Cadenas

**¡IMPORTANTE!**

Con las **triples comillas dobles** la cadena admite saltos de línea, se conocen como **raw strings** (cadenas en crudo) en las que cuentan todos los caracteres englobados entre las comillas.

```
"""A veces no soy yo  
Busco un disfraz mejor  
Bailando hasta el apagón  
Disculpad mi osadía"""
```

## 6.- Tipos de datos en Kotlin

### Cadenas

**¡IMPORTANTE!**

Se pueden eliminar los espacios en blanco del principio y del final y los márgenes de la izquierda.

```
"" | A veces no soy yo  
  | Busco un disfraz mejor  
  | Bailando hasta el apagón  
  | Disculpad mi osadía"".trimMargin()
```

Por defecto el carácter | es el delimitador del margen pero se puede cambiar: `trimMargin(">")`

## 6.- Tipos de datos en Kotlin

### Cadenas

**¡IMPORTANTE!**

Los **string templates** (plantillas de cadenas) permiten integrar las variables dentro de los strings, así usando el símbolo **\$** seguido del nombre de una variable, Kotlin lo sustituirá por el valor almacenado en la variable.

Si la variable es un objeto o se quiere operar con ella se debe envolver en llaves **{ }**.

```
"El descuento es un $discount %"
```

```
"El precio es ${product.price} €"
```

```
""" | Libreta de cuadros $notebookPrice €  
    | Bolígrafo azul $penPrice €  
    | Total: ${notebookPrice + penPrice}  
""".trimMargin()
```

## 6.- Tipos de datos en Kotlin

### Cadenas

**¡IMPORTANTE!**

Se **recomienda** siempre el uso de **string templates** y de **raw strings**.

Se pueden concatenar cadenas con el carácter **+** pero se debe evitar su uso.

Se debe evitar el uso de varias instrucciones seguidas con `println`.

## 6.- Tipos de datos en Kotlin

### Arrays

Los arrays son **estructuras de datos de longitud fija** que permiten almacenar varios valores del mismo tipo (números, caracteres, strings, booleanos, objetos).

1 2 3 4 5

"Rick", "Morty", "Summer"

## 7.- Declaración de variables

Mediante la palabra reservada **var** se declaran variables.

Las variables son como cajas que **almacenan valores** y estos valores pueden cambiar durante la ejecución del programa.

Kotlin es un lenguaje de **tipado estático**, esto significa que cuando una variable es de un tipo, ese tipo no puede cambiar (como Java).

**Al declarar una variable se puede indicar o no el tipo de dato** que se almacenara.

Si no se indica, Kotlin lo **deducirá** (inferred) dependiendo del tipo de dato que se asigne.

## 7.- Declaración de variables

Cuando se declara una variable obligatoriamente se debe realizar una de estas dos opciones:

- **Indicar el tipo de dato** que se almacenará.
- **Indicar el valor** que se almacenará (Kotlin deducirá el tipo de dato).

También se puede declarar una variable realizando las dos acciones:  
indicar el tipo de dato y el valor que almacenará.

## 7.- Declaración de variables

### Declaración de variables de tipos "simples"

```
var name: String = "Rick Sanchez"  
var dimension = "c-137" // el tipo de dato se deduce  
var gender: Char = 'h'  
age: Int  
age = 70  
var isAlive: Boolean = true
```



## 7.- Declaración de variables

### Declaración de arrays

Son objetos y se crean mediante la **clase Array**, por ello disponen de los métodos get y set pero por convenciones está disponible también el uso de corchetes [ ].

```
// se indica el tipo de dato del array
var numbers: Array<Int> = arrayOf(1, 2, 3, 4, 5)

// se deduce el tipo de dato del array
var names = arrayOf("Rick", "Morty", "Summer")

println(numbers[1])      // numbers.get(1)
numbers[3] = 12         // numbers.set(3, 12)

println("Hay ${names.size} nombres");
```

## 7.- Declaración de variables

### Declaración de arrays

Se puede usar el constructor de la clase Array para indicar el tamaño e incluso una **función lambda** para rellenar los elementos.

```
var array = Array( size: 10) { }           // array de 10 elementos vacíos (Unit)

var zeros = Array( size: 5) { 0 }          // array de 5 enteros todos 0

var emptys = Array( size: 5) { "" }        // array de 5 cadenas todas ""

var serie = Array( size: 100) { it }        // array de 100 enteros: 0, 1, 2 .. 99

var reverse = Array( size: 50) { 50-it }    // array de 50 enteros: 50, 49, 48 .. 0
```

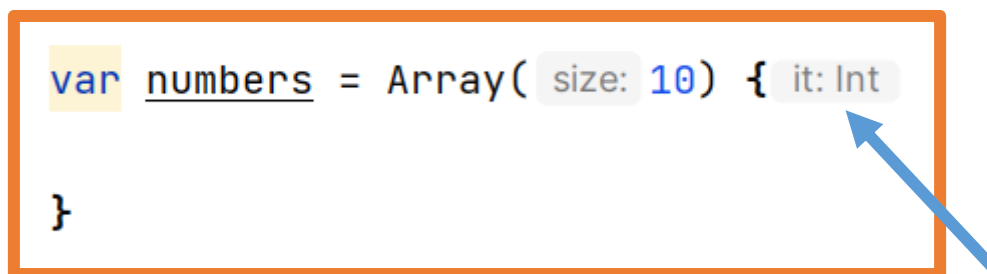
## 7.- Declaración de variables

### Declaración de arrays

El código escrito entre llaves es una **función lambda** (anónima), que se explicarán más adelante.

En esta función lambda se dispone de la variable **it** que representa el número de iteración comenzando por cero.

Como IntelliJ IDEA por defecto tiene activadas **las pistas en el código** (hints), si se deja un salto de línea entre las llaves se mostrará esta información.



```
var numbers = Array(size: 10) {  
    it: Int  
}
```

## 7.- Declaración de variables

### Declaración de arrays multidimensionales

Los **arrays multidimensionales** son arrays en los que cada elemento es a su vez un array.

Los más comunes son las matrices que se pueden representar como una tabla.

Hay varias maneras de declarar un array multidimensional en Kotlin:

```
// Matriz 3x4 de enteros ya inicializados
val matrix = arrayOf(
    arrayOf(5, 3, 2, 0),
    arrayOf(4, 9, 4, 5),
    arrayOf(8, 1, 7, 6)
)

// Matriz de 3x3 enteros inicializados a 0
var matrixOfInts = Array( size: 3) { Array( size: 3) { 0 } }

// Matriz de 6x5 cadenas inicializadas a ""
var matrixOfStrings = Array( size: 6) { Array( size: 5) { "" } }
```

5	3	2	0
4	9	4	5
8	1	7	6

## 7.- Declaración de variables

El **acceso** a los elementos de la matriz se realiza **con los índices de la fila y la columna**.

```
val matrix = arrayOf(  
    arrayOf(5, 3, 2, 0),  
    arrayOf(4, 9, 4, 5),  
    arrayOf(8, 1, 7, 6)  
)  
  
matrix[1][3] = 8  
  
for (row in matrix) {  
    for (column in row) {  
        print("$column ")  
    }  
    println()  
}
```

## 7.- Declaración de variables

Todos los tipos de variables son clases y en la jerarquía de Kotlin tienen una superclase que es **Any** (en Java sería Object).

Se puede declarar una variable indicando el tipo Any y luego asignar cualquier tipo de dato. Esta acción puede tener utilidad en algunos casos como se verá más adelante.

```
var number: Any = 6  
var name: Any = "Rick Sanchez"  
var isAlive: Any = true  
var numbers = arrayOf(1, 2, 3)
```

## 7.- Declaración de variables

Para poder usar una variable esta debe almacenar un valor.

Existe un tipo de dato que permite dejar vacía una variable **Unit** (en Java sería void).

Se puede declarar una variable indicando el tipo Any y luego asignar cualquier tipo de dato. Esta acción puede tener utilidad en algunos casos como se verá más adelante.

```
val variable1: Any
// Produce error al no haberse inicializado la variable
println("valor de variable: $variable1")

val variable2: Any = Unit
// Funciona correctamente
println("valor de variable: $variable2")
```

## 7.- Declaración de variables

### Ámbito de las variables

El ámbito de una variable es el lugar donde se puede utilizar.

Como norma general una variable **se podrá utilizar dentro de todo el bloque de código en el que se declare** que se delimita por las llaves { }.

Una variable se podrá usar dentro de todo ese bloque, incluidos otros bloques que se contengan dentro.

Si una variable **se declara fuera de un bloque { }** se podrá utilizar **en todos los archivos del proyecto.**



## 8.- Conversión de tipos

Si se quiere realizar una conversión de tipo de dato siempre **se deberá usar un método de la clase del tipo de dato a convertir.**

Por ejemplo, para convertir un Double a un Int, se deberá usar sobre la variable decima el método toInt de la clase Double.

Esta acción además truncará el valor y se perderá la parte decimal.

```
var decimalNumber: Double = 12.34  
var intNumber: Int  
intNumber = decimalNumber.toInt()
```

## 8.- Conversión de tipos

Los métodos para la conversión de tipos suelen ir precedidos de **to**:

toDouble()

toFloat()

toLong()

toInt()

toChar()

toShort()

toByte()

toString()

Cuando se pide un dato por teclado este siempre es una cadena, así que si se pide un número se debe convertir a entero o decimal según convenga:

```
var number = readln().toInt()
```

## 9.- Declaración de constantes

Se usa la palabra reservada **val** para declarar constantes.

Su uso es similar al de las variables declaradas con `val` con la excepción de que una vez asignado un valor ya no se podrá cambiar durante la ejecución.

```
val age: Int = 23  
age = 24
```

Aunque sí que se pueden asignar durante la ejecución del programa por ejemplo tras pedir un dato al usuario.

```
val age: Int  
age = 24
```

```
val age: Int  
age = readln().toInt()
```

## 9.- Declaración de constantes

Los array y todos los objetos se pueden declarar como val y posteriormente cambiar los valores que almacenan.

Esto es debido a que son objetos y como val no se podrá cambiar la instancia que almacena, aunque sí se podrán cambiar los valores.

```
val numbers = arrayOf(1, 2, 3, 4, 0)
// Se puede cambiar el contenido de los elementos
numbers[4] = 5

// Al declararse como var no se puede modificar el objeto
var numbers = arrayOf(1, 2)
```

## 9.- Declaración de constantes

Se puede usar la palabra reservada **const** previamente a la declaración con **val** para declarar constantes de valores conocidos.

Estas constantes no podrán ser asignadas en tiempo de ejecución por eso deben de estar fuera de las funciones.

Las constantes con **const** sí se pueden declarar dentro de objetos declarados como **singleton** (clase con una única instancia que se verán más adelante).

```
const val PI = 3.1416

fun main() {
    println(PI)
}
```

```
fun main() {
    const val PI = 3.1416
    println(PI)
}
```

```
object Constants {
    const val PI = 3.1416
}

fun main() {
    println(Constants.PI)
}
```

# 10.- Operadores

Permiten crear expresiones. Son similares a los de Java

De signo:        +   -

Aritméticos:    +   -   \*   /   %

Asignación:     =   +=   -=   \*=   /=   %=

Incremento:    ++   --   (puede ser pre-incremento o post-incremento)

Relacionales:   ==   !=   <   >   <=   >=

Lógicos:        &&   ||   !

A nivel de bit:   and   or   xor   inv   shl (desplazamiento ←)   shr (desplazamiento →)

# Práctica

## Actividad 1

Primeros programas en Kotlin

## 11.- Null Safety

Uno de los errores más comunes y temidos que puede producirse cuando se programa es el odiado **NullPointerException**.

Este error ocurre cuando el programador no inicializa un objeto o por alguna razón no se ha inicializado.

Por defecto Kotlin no permite asignar **null** a las variables para evitar posibles errores.

```
var name: String = null
```


Pero Kotlin sí que tiene soporte para el valor **null** de forma nativa.



# 11.- Null Safety

Para **declarar variables que puedan almacenar null** en la declaración se debe usar el símbolo ? detrás del tipo de dato de la variable.

```
var name: String = "Rick Sanchez"  
name = null  
  
var nullableName: String? = "Morty Smith"  
nullableName = null
```

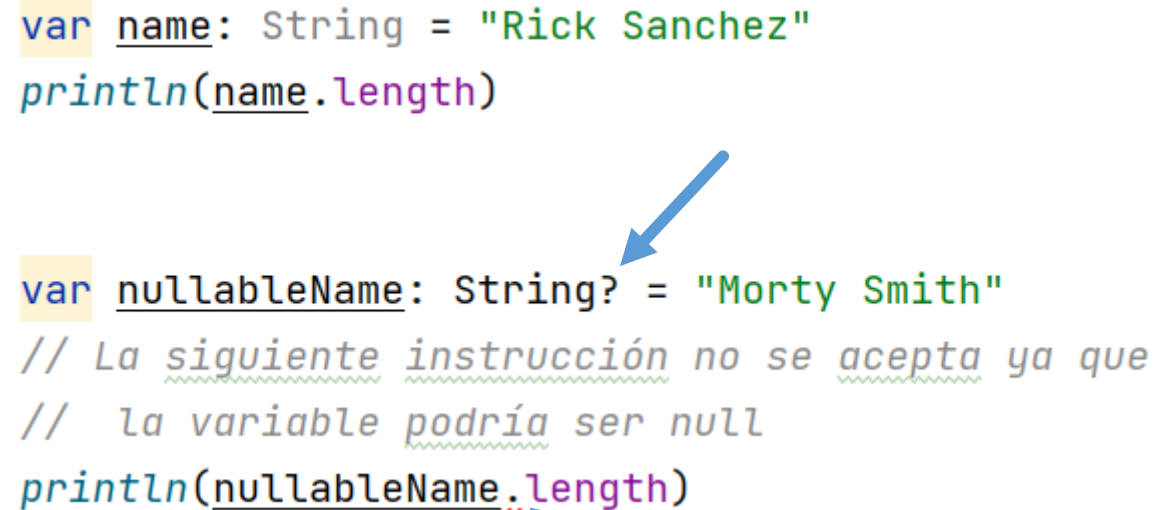


# 11.- Null Safety

Kotlin evitará realizar acciones que puedan dar error si la variable es null.

```
var name: String = "Rick Sanchez"
println(name.length)

var nullableName: String? = "Morty Smith"
// La siguiente instrucción no se acepta ya que
// la variable podría ser null
println(nullableName.length)
```



# 11.- Null Safety

Como en Java, para las variables que pueden almacenar null se puede hacer una comprobación previa antes de intentar acceder a ella.

```
var nullableName: String? = "Morty Smith"
if (nullableName != null) {
    println("Hola $nullableName!")
} else {
    println("Hola invitado!")
}
```

Con el operador ? también permite realizar las comprobaciones anteriores de una manera más sencilla y reduciendo código.

```
var nullableName: String? = "Morty Smith"
println("${nullableName?.length}")
```

Este operador permite comprobaciones simultaneas.

```
val city: String? = user?.address?.city
```

## 12.- Operador Elvis ?:

El operador Elvis ?: permite ofrecer un valor por defecto cuando una variable almacene el valor nulo.

```
var nullableName: String? = "Morty Smith"
println(nullableName ?: "desconocido")
nullableName = null
println(nullableName ?: "desconocido")
```



```
"C:\Program Files\Java\jdk1.8.0_111\bin\java.exe" ...
Morty Smith
desconocido

Process finished with exit code 0
```

El uso más común es conseguir un valor distinto de null cuando una propiedad es null o un método devuelve null.

```
val upperName = name?.uppercase() ?: "Desconocido"
val city = user?.address?.city ?: "Desconocida"
```

## 12.- Operador Elvis ?:

Este operador también se puede usar para **lanzar** una excepción.

```
var nullableName: String? = "Morty Smith"
val name = nullableName ?: throw IllegalArgumentException("El nombre es nulo")
```

Si se necesita ejecutar varias instrucciones tras el operador Elvis, estas se deberán incluir en un bloque de instrucciones tipo **run { }** (se explicará con detalle más adelante).

Se debe tener en cuenta que la última instrucción del bloque debe ser el valor a asignar en el caso de que la variable sea null.

```
val a = b ?: run {
    val valueWhenBIsNull = c.notNullValue()
    storeValue(valueWhenBIsNull)
    valueWhenBIsNull ^run // Este es el valor por defecto del operador Elvis
}
```

## 13.- El operador !!

Cuando se esté muy seguro de que una variable **que puede almacenar null no lo almacena**, se puede utilizar el operador **!!** para evitar la comprobación de Kotlin.

En este caso si la variable almacena null se lanzará la **excepción NullPointerException**.

```
var nullableName: String? = "Morty Smith"  
println(nullableName!!.uppercase())
```

Este operador suele aparecer en migraciones de Java a Kotlin.

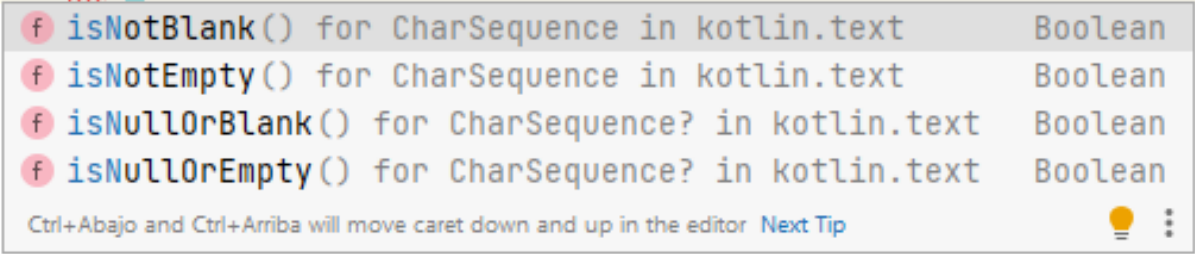
La filosofía de kotlin **es contraria a su uso**.

## 14.- Expresiones regulares

En todo lenguaje de programación es necesario comprobar si los valores introducidos por el usuario cumplen unas reglas específicas.

Una acción muy típica es comprobar si el usuario ha introducido un valor y si ese valor es una cadena vacía o no.

```
var name = readln()
if (name.isN)
```



f	isNotBlank()	for CharSequence in kotlin.text	Boolean
f	isEmpty()	for CharSequence in kotlin.text	Boolean
f	isNullOrBlank()	for CharSequence? in kotlin.text	Boolean
f	isNullOrEmpty()	for CharSequence? in kotlin.text	Boolean

Ctrl+Abajo and Ctrl+Arriba will move caret down and up in the editor [Next Tip](#)

Estas comprobaciones en ocasiones pueden no ser suficientes para los requisitos del programa, en ese caso se deben usar **expresiones regulares**.

## 14.- Expresiones regulares

Las expresiones regulares son **patrones de caracteres** que permiten **comprobar** si **una cadena de caracteres** se ajusta al patrón o no.

Ejemplos típicos de comprobaciones con expresiones regulares:

- Que un nombre no tenga números.
- Que tenga una longitud concreta.
- Que se siga un orden en los caracteres (DNI → 8 números y 1 letra).
- Que sea un email.
- ...

Las expresiones regulares existen en muchos lenguajes de programación así que es importante conocer su funcionamiento.



## 14.- Expresiones regulares

Para crear una expresión regular en Kotlin se utiliza la clase **Regex**:

```
val checkDNI = Regex("expresión_regular")
```

Se permite indicar [modificadores](#) a la expresión regular:

```
val check = Regex("expresión_regular", RegexOption.IGNORE_CASE)
```

El diseño de expresiones regulares es una disciplina cuyo nivel de dificultad aumenta conforme se quieren comprobar patrones más complejos.

A continuación, se verá una pequeña guía del uso de las expresiones regulares que servirá para aprender a utilizarlas de una manera básica.

# 14.- Expresiones regulares

## Agrupación

- `[ ]`      `[abc]`    contiene cualquier carácter de entre los indicados.
- `[^ ]`      `[^abc]`    contiene cualquier carácter que no sea de los indicados.
- `[ - ]`      `[0-9]`    contiene cualquier carácter que se encuentre en el rango.
- `[^ - ]`    `[^A-B]`    contiene cualquier carácter que no esté en el rango.
- `( | )`      `(x|y)`    contiene uno de los caracteres (separador `|`).

## Cantidad de caracteres

- `{ }`      `a{3}`      contiene exactamente 3 'a' seguidas.
- `{ , }`    `a{3,}`    contiene 3 o más 'a' seguidas.
- `{ , }`    `a{3,5}`    contiene 3, 4 o 5 'a' seguidas.
- `*`      `a*`      contiene 0 o más 'a'.      Similar: `a{0, }`
- `+`      `a+`      contiene 1 o más 'a'.      Similar: `a{1, }`
- `?`      `a?`      contiene 0 o 1 'a'.      Similar: `a{0,1}`

# 14.- Expresiones regulares

## Inicio – fin

- `^`      `^hola`      empieza con "hola".
- `$`      `hola$`      acaba con "hola".
- `^ $`      `^hola$`      exactamente "hola".

## Otros

- `\\s`      el carácter espacio en blanco.
- `\\S`      cualquier carácter que no sea espacio en blanco.
- `\\w`      cualquier carácter menos símbolos.
- `\\W`      solo símbolos.
- `\\C`      no es una letra.
- `\\d`      un dígito.      Similar: `[0-9]`
- `\\D`      no es un dígito.      Similar: `[^0-9]`

# 14.- Expresiones regulares

## Ejemplos de expresiones regulares en Kotlin

```
// tiene 4 minúsculas  
"[a-z]{4}"
```

```
// tiene 8 caracteres: letras y/o números  
"[a-zA-Z0-9]{8}"
```

```
// 7 u 8 dígitos seguidos de 1 letra (DNI)  
"^\\d{7,8}\\w{1}$"
```

```
// conjunto de letras, seguidas de arroba seguida, seguidas de un  
// conjunto de letras, seguidas de un punto y seguido de 2 o 3  
// letras → patrón simple para un email: rick_sanchez@mail.com  
"^\\[a-z_\\.\\]+@[a-z]+\\.\\[a-z]{2,3}$"
```

## 14.- Expresiones regulares

### Comprobar si una variable cumple una expresión regular

Para comprobar si una variable cumple un patrón establecido en una expresión regular se pueden utilizar tanto métodos de la clase String como métodos de la clase Regex:

```
val name = "Rick"  
// Empieza con mayúscula y le siguen al menos dos minúsculas  
val checkName = Regex( pattern: "[A-Z][a-z]{2,}$")  
// Comprobación con String  
println(name.contains(checkName))  
// Comprobación con Regex  
println(checkName.matches(name))
```

La clase Regex tiene más métodos con diferentes funcionalidades, entre ellos:

replace

find

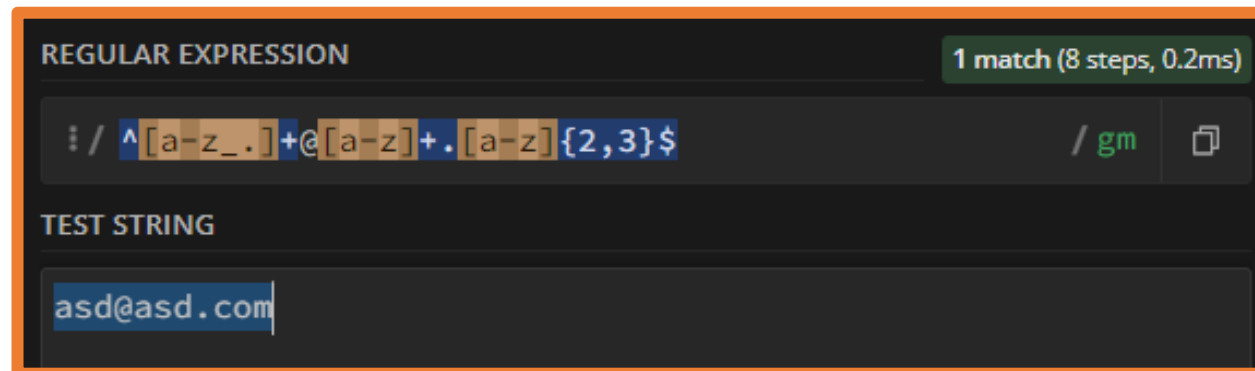
## 14.- Expresiones regulares

Para crear expresiones regulares se pueden consultar Cheat Sheets:

<https://quickref.me/regex>

También existen herramientas para comprobar el funcionamiento de una expresión regular:

<https://regex101.com/>



## 15.- Excepciones

Las excepciones se capturan con bloques **try – catch** de la misma manera que en Java.

```
try {  
    var name: Int = readln().toInt()  
} catch (e: Exception) {  
    println("Error: ${e.message}")  
}
```

Igual que en Java se pueden lanzar excepciones si se desea:

```
val number: Int  
try {  
    println("Introduce un número entre 1 y 100:")  
    number = readln().toInt()  
    if(number<1 || number>100) throw Exception("Número fuera de rango")  
} catch (e: NumberFormatException) {  
    println("Error: No se ha introducido un número")  
} catch (e: Exception) {  
    println("Error: ${e.message}")  
}
```