

PROGRAMACIÓN MULTIMEDIA Y DISPOSITIVOS MÓVILES

- Kotlin -

PROFESOR: PATRICIA SÁNCHEZ FORMOSO

patriciasfo@educastur.org

[CURSO 2024 - 2025]

Índice

1. ¿Qué es Kotlin?
2. Ventajas de Kotlin
3. Instalando IntelliJ Idea
4. Primera aplicación en Kotlin: Hola Mundo
5. Lenguaje de programación Kotlin

1. ¿Qué es Kotlin?

- Kotlin es un lenguaje de programación creado en 2010 por JetBrains, la empresa creadora de uno de los IDE para java más famosos del mundo -> INTELLIJ IDEA
- Kotlin es una alternativa a Java, que suple varios de los problemas más habituales que los programadores nos encontramos en dicho lenguaje. Por eso y para suplir más carencias de otros lenguajes de programación kotlin fue desarrollado.

Índice

1. ¿Qué es Kotlin?
2. **Ventajas de Kotlin**
3. Instalando IntelliJ Idea
4. Primera aplicación en Kotlin: Hola Mundo
5. Lenguaje de programación Kotlin

2. Ventajas de Kotlin

1. **Seguro contra nulos:** Uno de los mayores problemas de usar java son los *NullPointerException*. Esto ocasiona una gran cantidad de problemas a la hora de desarrollar. Con Kotlin nos olvidaremos de esto pues nos obliga a tener en cuenta los posibles null.
2. **Ahorra código:** Con kotlin podrás evitar muchísimas líneas de código en comparación con otros lenguajes. Imagina hacer un POJO (*Plain Old Java Objects*) en una sola línea en vez de 50-100.
3. **Características de programación funcional:** Kotlin está desarrollado para que trabajemos tanto orientado a objetos, como funcional (e incluso mezclarlos), lo que nos dará mucha más libertad y la posibilidad de usar características como *higher-order functions*, *function types* y *lambdas*.
4. **Fácil de usar:** Al estar inspirado en lenguajes ya existentes como Java, C# o Scala, la curva de aprendizaje nos será bastante sencilla.
5. **Es el momento:** Kotlin se ha convertido oficialmente en un lenguaje de Android, por lo que ahora mismo es el boom.

Índice

1. ¿Qué es Kotlin?
2. Ventajas de Kotlin
- 3. Instalando IntelliJ Idea**
4. Primera aplicación en Kotlin: Hola Mundo
5. Lenguaje de programación Kotlin

3. Instalando IntelliJ Idea

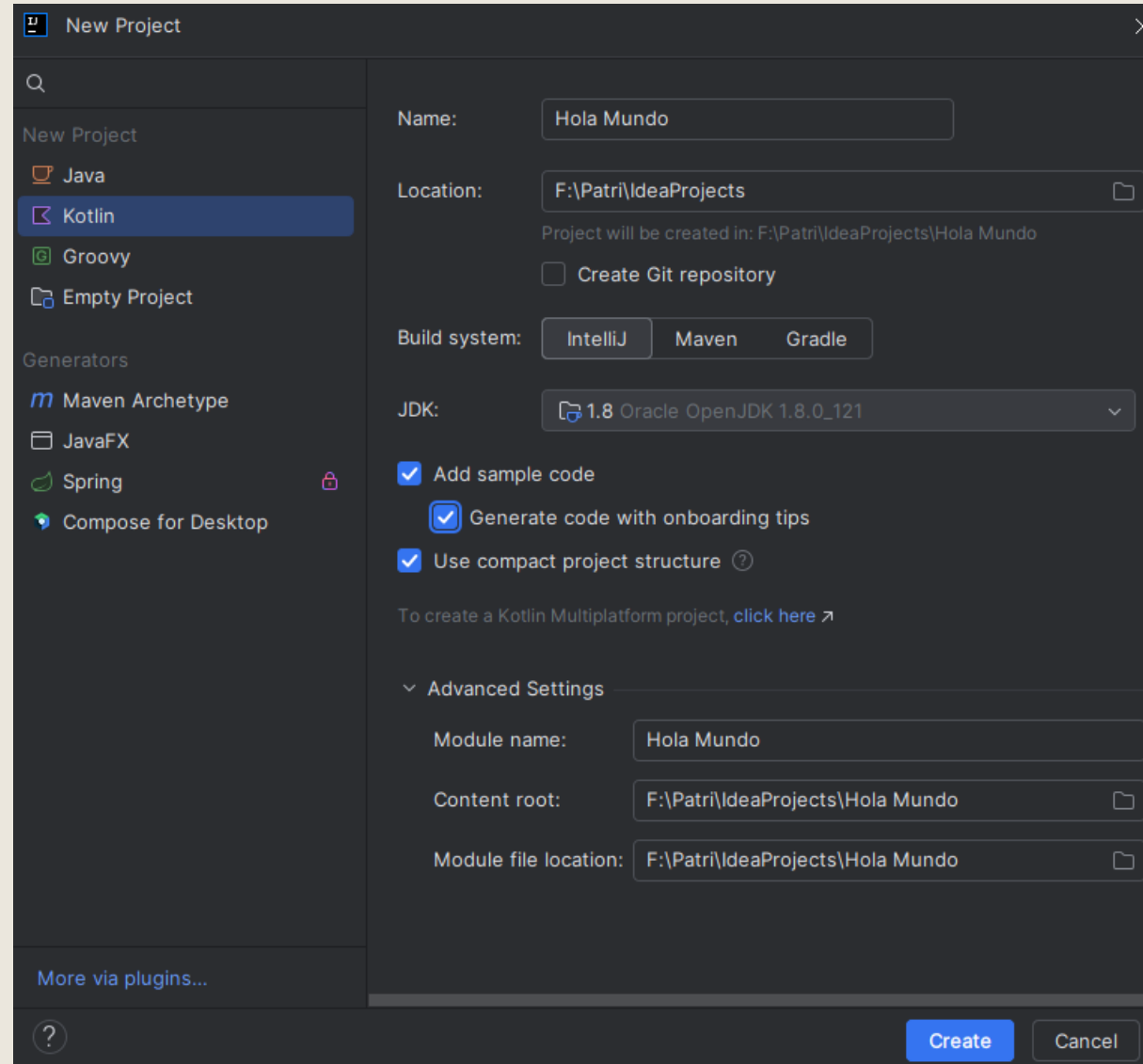
- Ir a la página oficial y descargar la última versión: <https://www.jetbrains.com/idea/>
- 2 versiones para descargar:
 - Ultimate -> de pago
 - **Community** -> gratuita y contiene lo necesario
- Una vez descargado, instalarlo:
 - En **Create associations** seleccionaremos los **.kt** que es el formato de archivo para los ficheros de kotlin
- Configurar el entorno:
 - Escoger el **theme**. El theme es básicamente la parte gráfica de nuestro IDE.
- Descargando el SDK
 - SDK (*Standard Development Kit*): básicamente es el conjunto de herramientas necesarias para poder trabajar con el IDE.

Índice

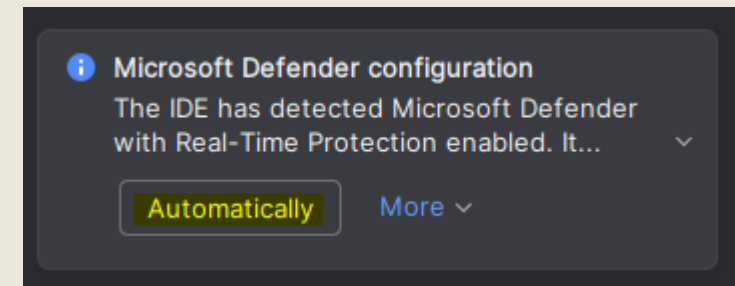
1. ¿Qué es Kotlin?
2. Ventajas de Kotlin
3. Instalando IntelliJ Idea
4. **Primera aplicación en Kotlin: Hola Mundo**
5. Lenguaje de programación Kotlin

4. Primera aplicación en Kotlin: Hola Mundo

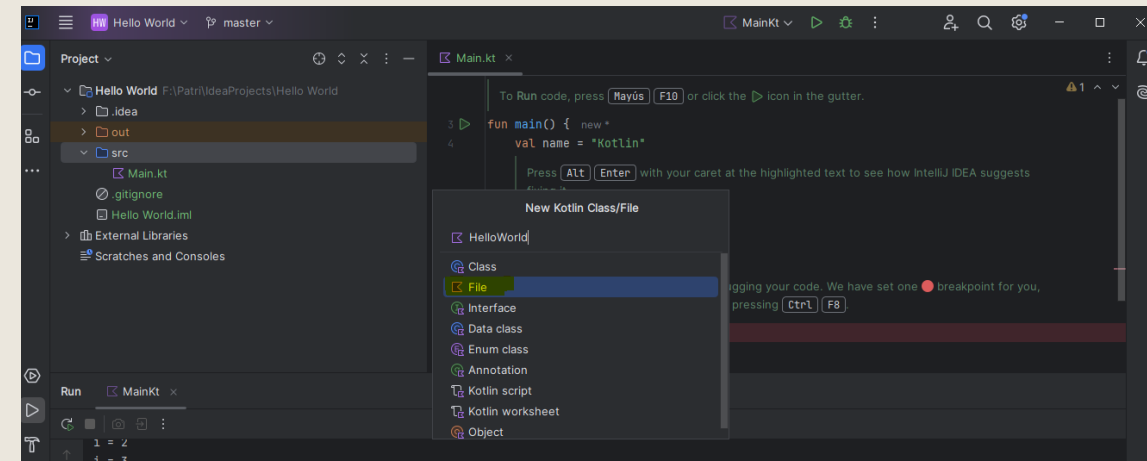
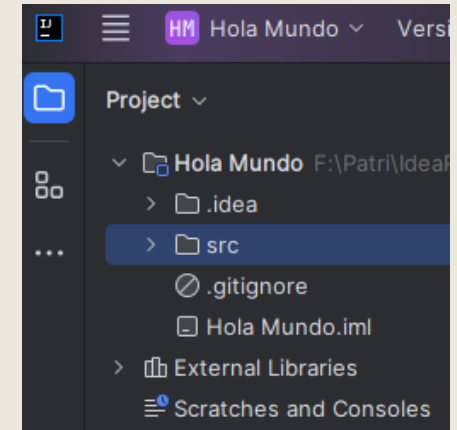
- Abrir IntelliJ Idea y le daremos a Create New Project y nos mostrará una ventana así:

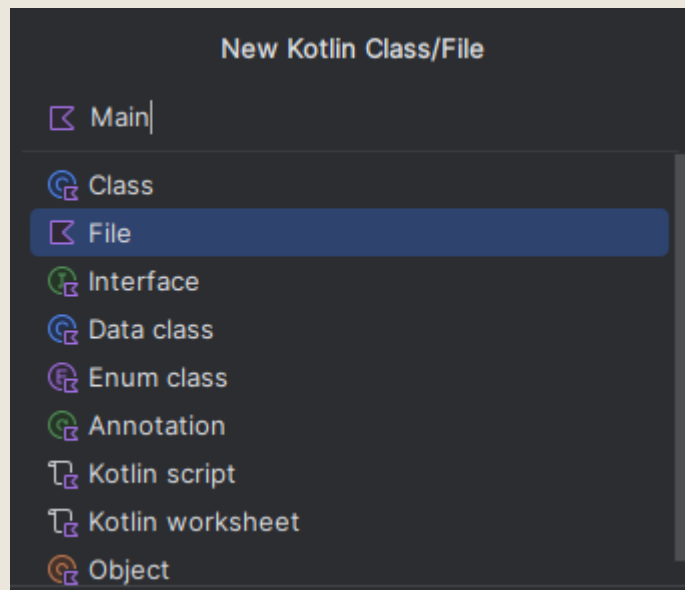


- Pulsamos en **Create** y ya tendremos nuestro proyecto creado.
- Si se muestra la siguiente ventana, pulsar **Automatically**.



- En el menú lateral izquierdo tendremos por defecto la vista «Project» que será con la que trabajaremos.
- La carpeta **.idea** y el archivo **.iml** son ficheros de configuración del IDE.
- Luego tenemos la carpeta **src** que será donde crearemos los ficheros y directorios para trabajar.
- Eliminar el archivo **Main.kt** que nos genera automáticamente.
- Carpeta **src**, botón secundario **New>Kotlin File/Class** y nos saldrá una ventana similar a esta.

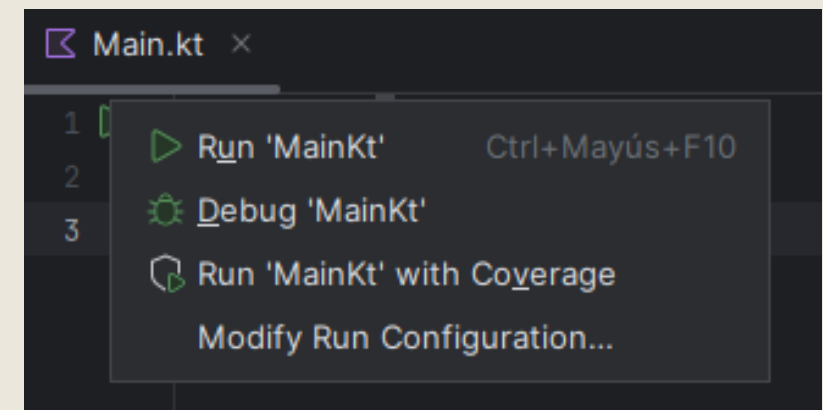




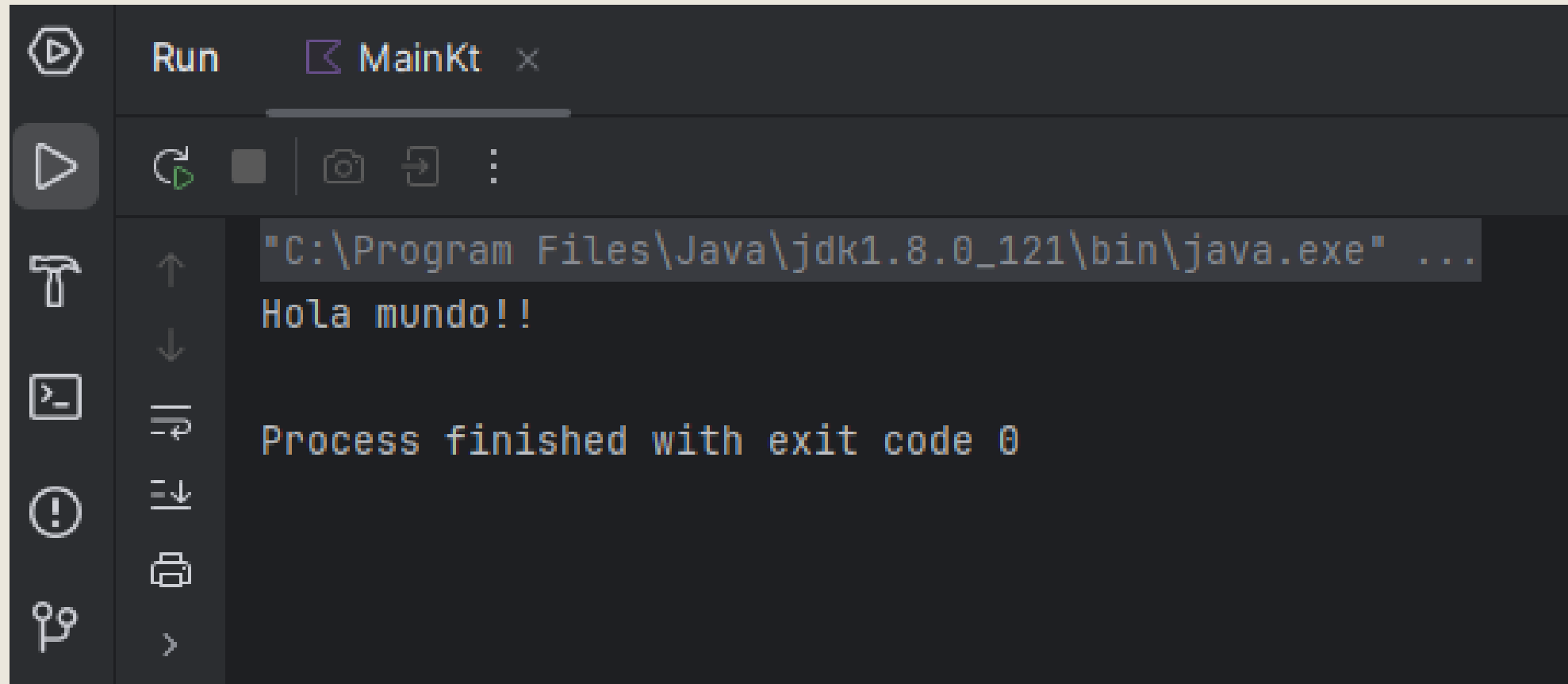
- Lo primero será crear un método main que de la posibilidad de ejecutar dicho archivo.
- Para generar dicho método, IntelliJ tiene unos templates que nos hará la vida mucho más sencilla, así que basta con escribir main y nos saldrá la opción de autocompletar. Generándonos una función igual a esta.

```
1 fun main(){
2     println("Hola mundo!!")
3 }
```

- El método **println** hace que todo lo que esté dentro del paréntesis (siempre entre comillas si es texto) lo muestre por pantalla.
- Kotlin es sensible a mayúsculas y minúsculas. Ejemplo: escribir Println (con la P mayúscula) y veréis que se os pone en rojo.
- Una vez hecho esto hacemos click en el triángulo verde y luego en Run o Debug 'MainKt'.



- El proyecto empezará a compilar y una vez termine nos mostrará por pantalla (en la parte inferior del IDE) el resultado de los cálculos hechos en la función.

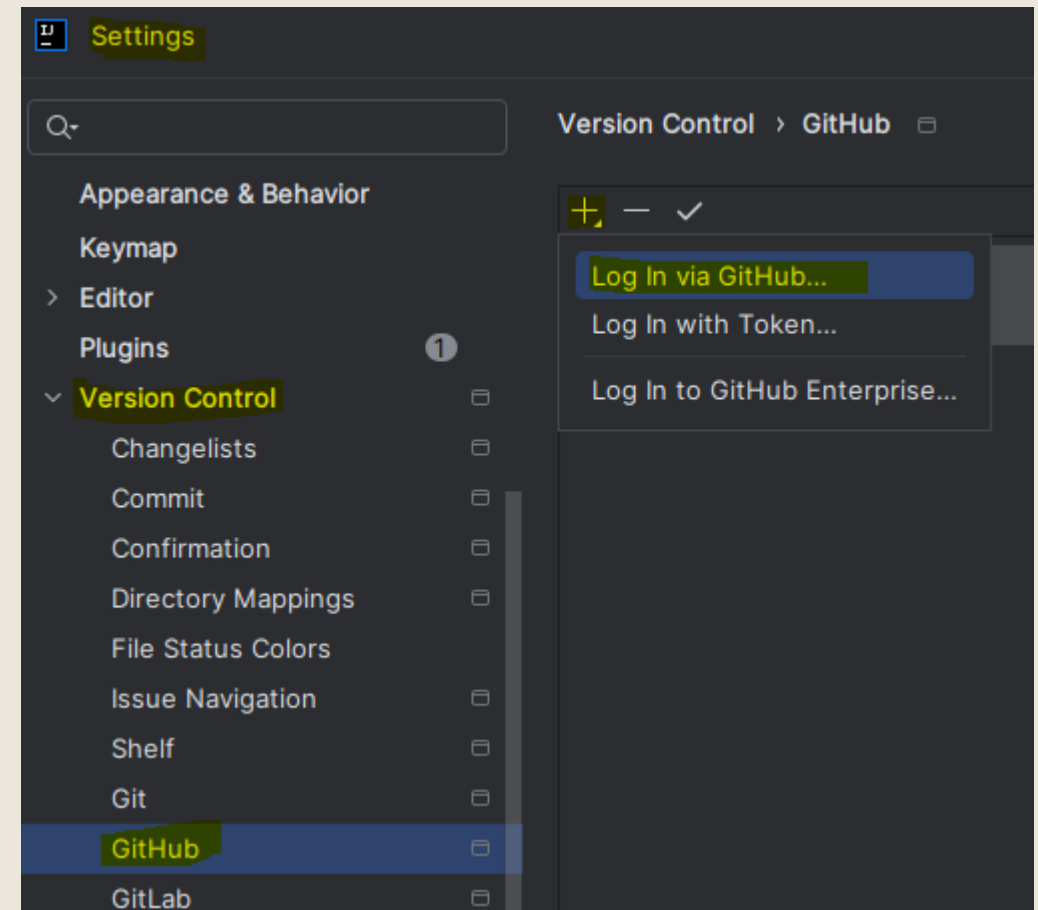
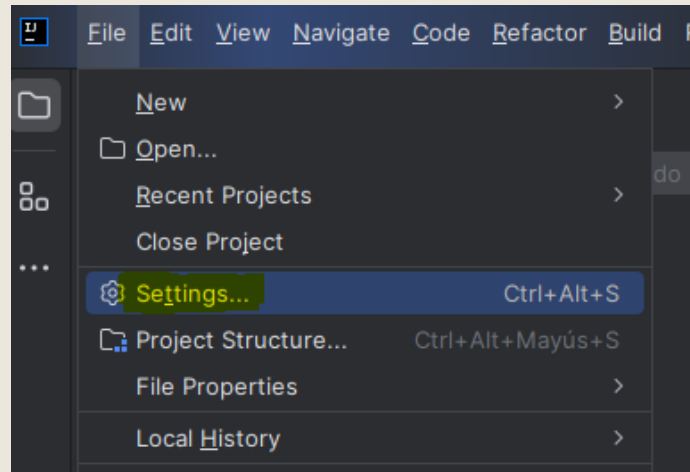


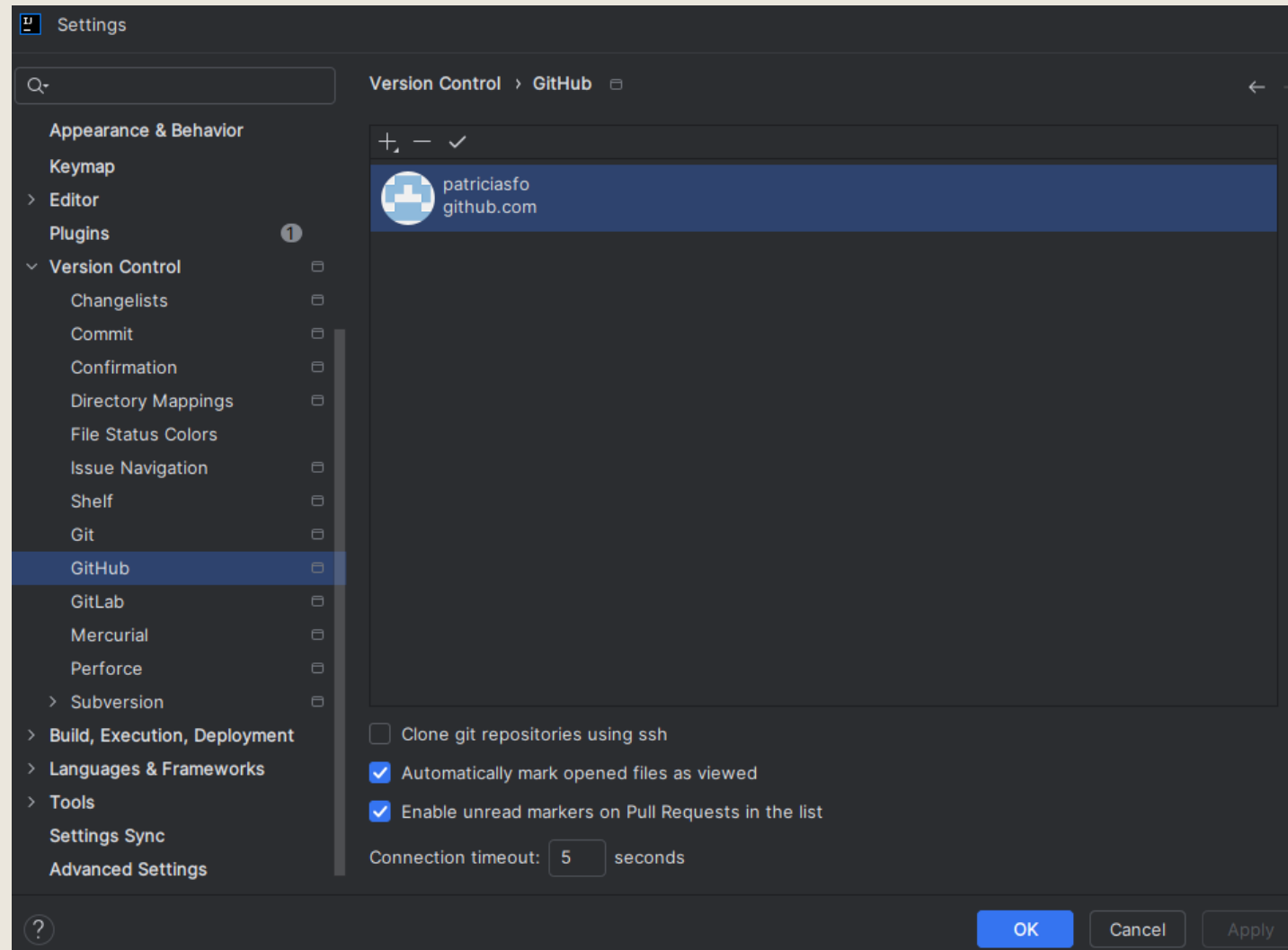
The screenshot shows the 'Run' console of an IDE. At the top, there's a 'Run' button (a play icon) and a tab labeled 'MainKt'. Below this is a toolbar with icons for running, debugging, and other actions. The main area of the console displays the following text:

```
"C:\Program Files\Java\jdk1.8.0_121\bin\java.exe" ...  
Hola mundo!!  
  
Process finished with exit code 0
```

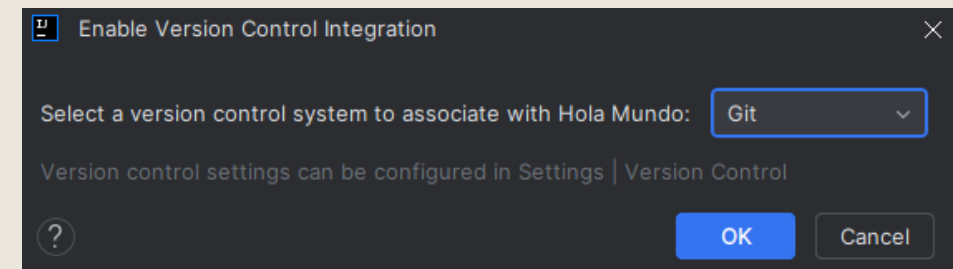
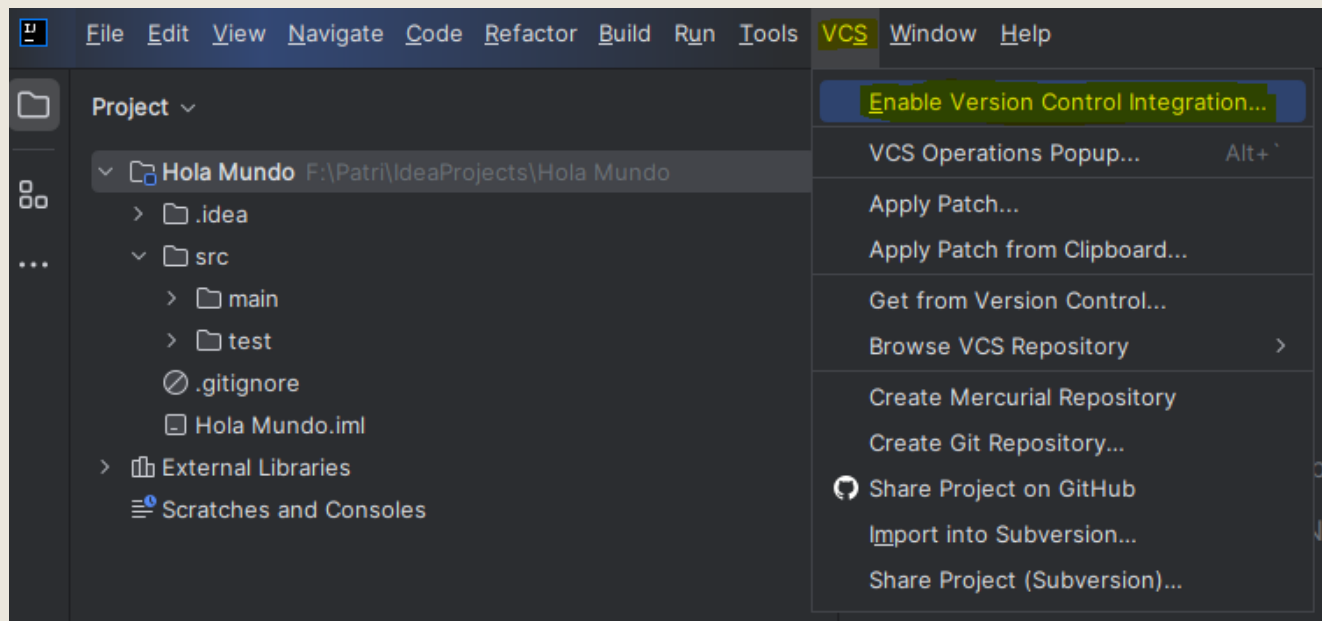
- Y con esto ya tenemos hecha nuestra primera aplicación. Muy sencillita pero ya tenemos conocimientos para ejecutarlas.

- Conviene que todos los proyectos tengan un control de versiones -> GIT
- Vamos a utilizar GitHub: repositorio, para almacenar el código de nuestros proyectos y poder gestionarlos
 1. Primero debemos registrarnos en GitHub con nuestra cuenta de educastur.
 2. Configuramos **IntelliJ Idea** para vincularlo a GitHub
 - a) **File -> Settings... -> Version Control -> GitHub** y agregar la cuenta desde la cual se va a proceder a subir sus proyectos. Pueden tener más de una cuenta.

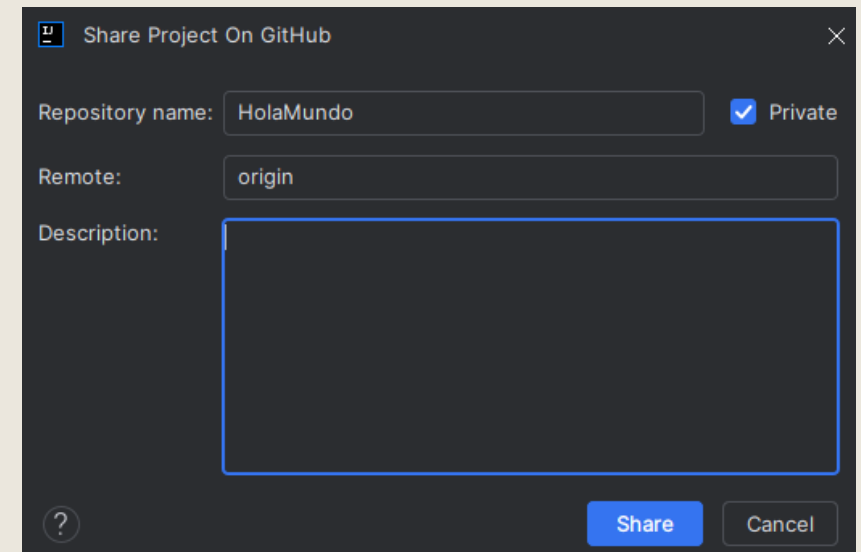
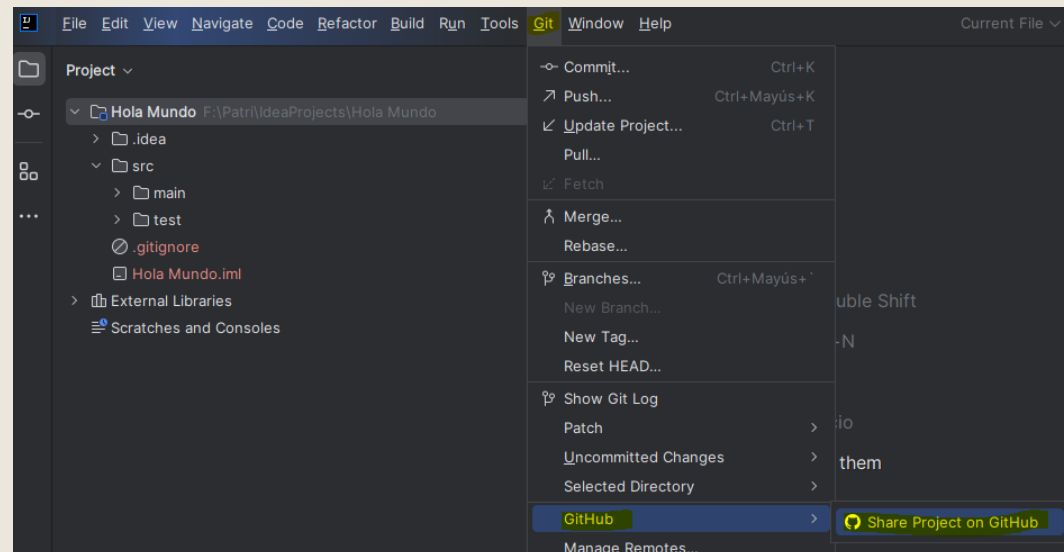




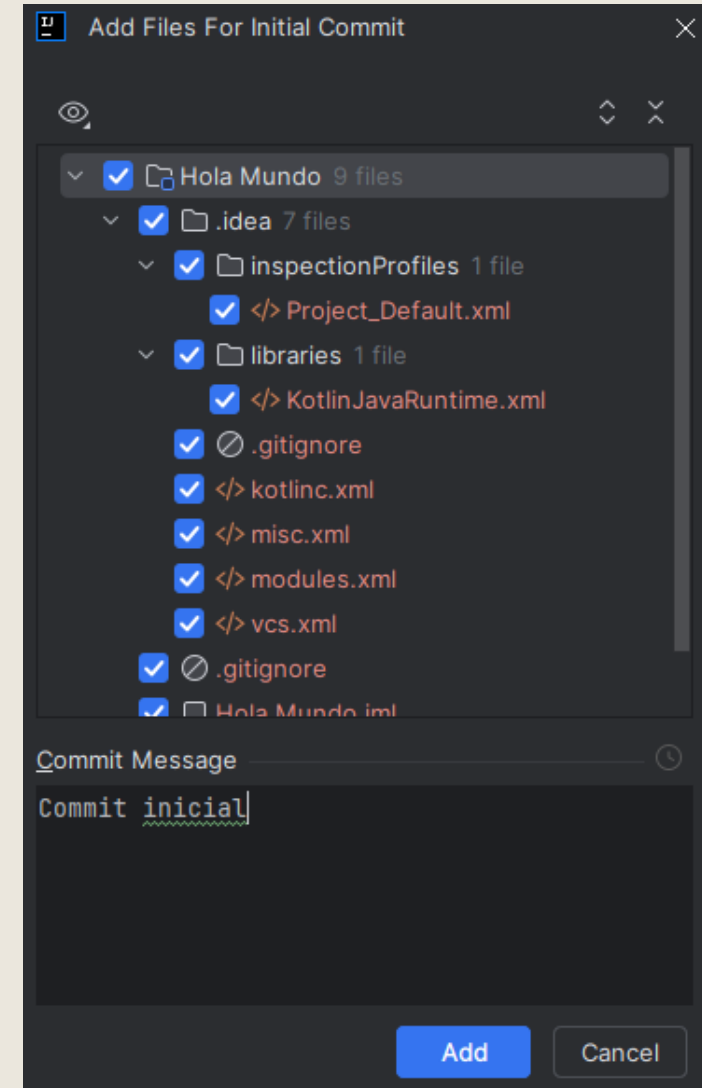
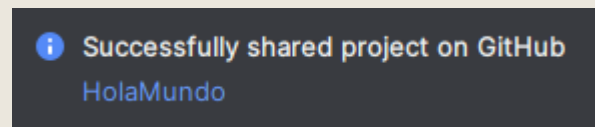
- b) Activar la integración del controlador de versiones en su proyecto a través de la opción **VCS -> *Enable Version Control Integration*** y seleccionar el controlador de versiones a utilizar. En este caso se seleccionará **Git**.

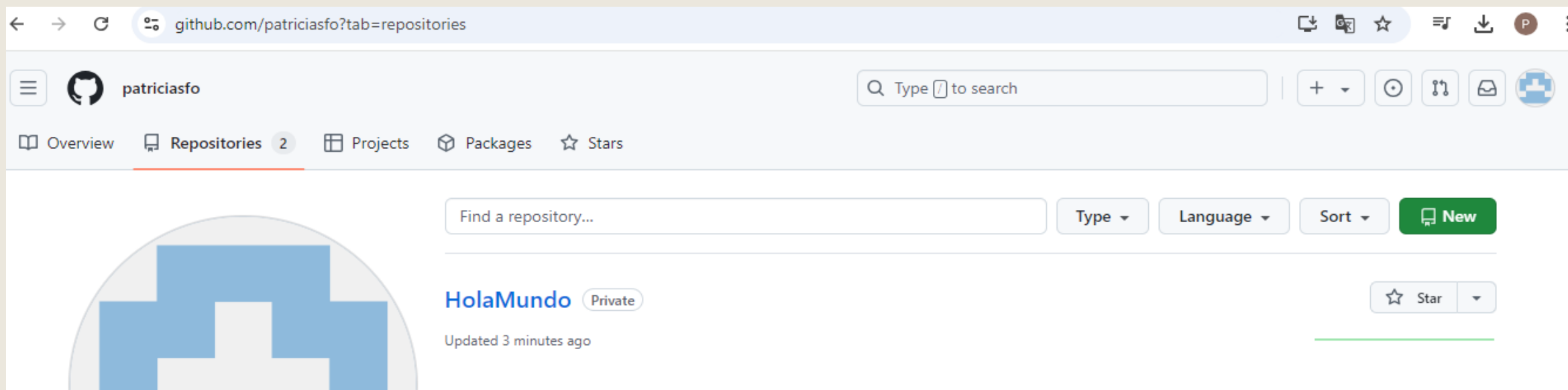
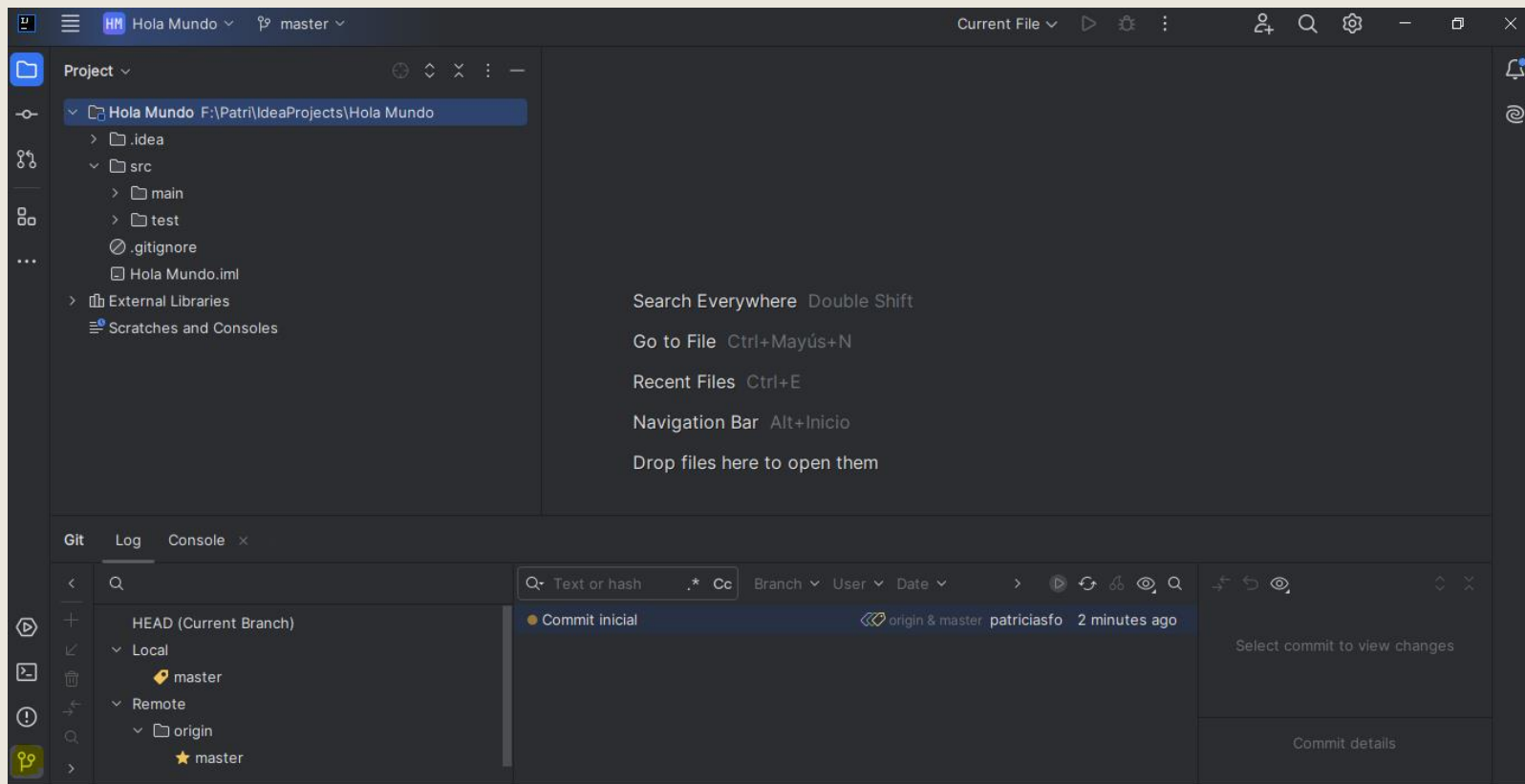


- c) Seleccionar la opción **Git -> GitHub -> Share Project on GitHub** del menú superior. Posterior a esto se mostrará una ventana para ingresar el nombre del repositorio, su descripción y el tipo repositorio. Una vez hemos terminado presionar el botón **Share**.



- d) Una vez presionado el botón Share, aparecerá una ventana donde se mostrarán todos los archivos que serán subidos al repositorio y un comentario inicial. Se tienen que verificar, marcando o desmarcando cada uno de ellos según se requiera. Y una vez listos presionar el botón **Add**.
- e) Si todo fue correcto, se podrá apreciar un mensaje de confirmación de que el proyecto ha sido subido a GitHub y se podrá apreciar el tracking del mismo con el primer commit en la vista de Git en la parte inferior.





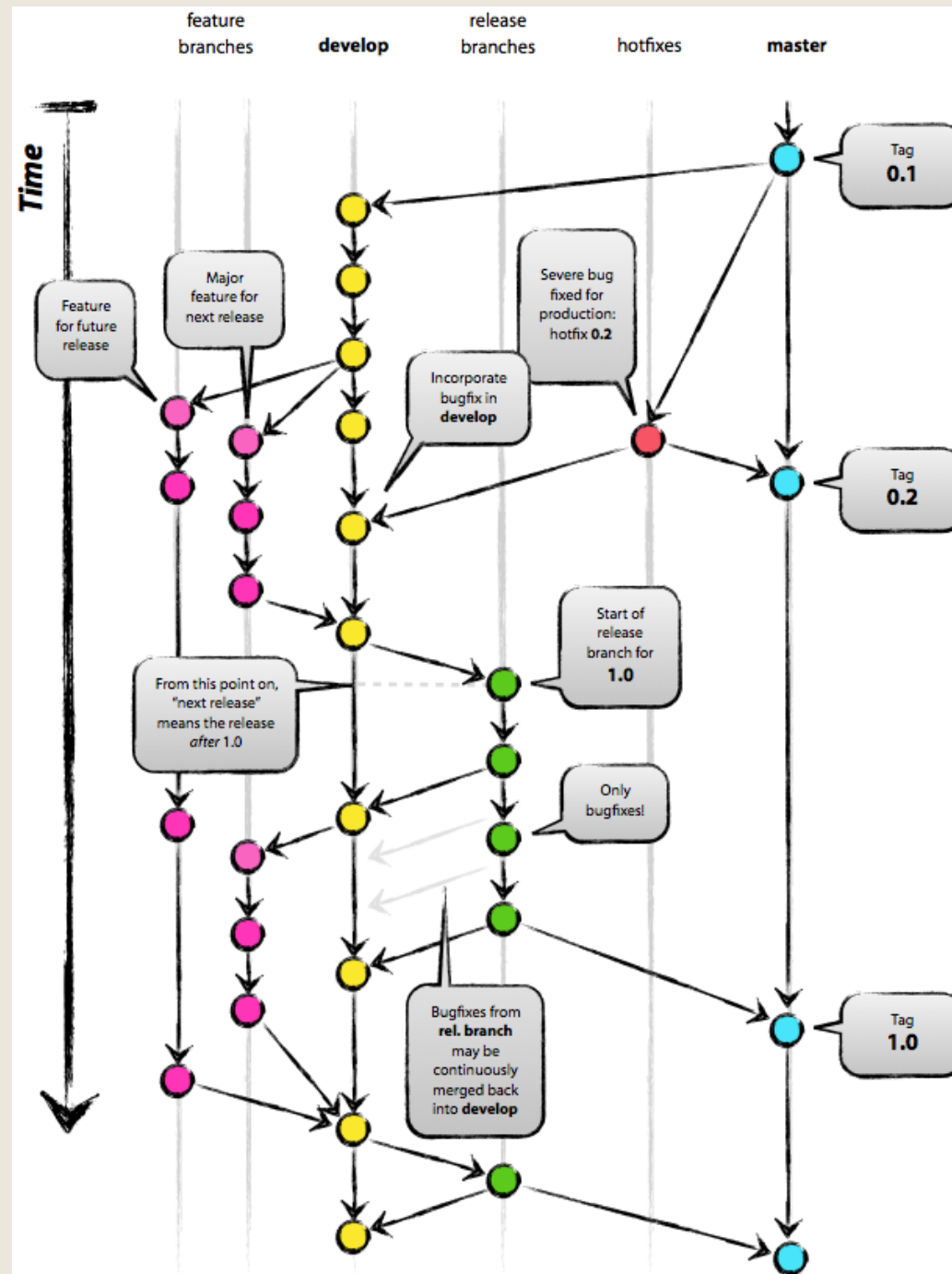
Actividad guiada con IntelliJ Idea

- Modificar el texto de la función.
- Commit
- Comprobar repositorio
- Push
- Comprobar repositorio
- Crear la Branch 'develop' y pushearla
- Modificar el texto en la rama develop
 - Commit
 - Push
- Cambiar entre las ramas y:
 - Comprobar que no existe el mismo código.
 - Ejecutar el código y ver las distintas salidas por consola.

SOURCETREE

- Otro cliente gratuito para Git que nos permite trabajar con nuestros repositorios de código de forma sencilla es **Sourcetree**.
- Descargar el programa e Instalarlo.
- Crear un nuevo proyecto de Kotlin:
 - Esta vez deja por defecto todos los valores y clases que genere.
 - Indicarle el control de versiones al igual que se hizo anteriormente.
 - Compartirlo en GitHub.
- Clonar por HTTP en Sourcetree el repositorio de GitHub.
- Iniciar el **flujo del Git**
- Ya podemos empezar a trabajar en nuestro proyecto

FLUJO DE GIT

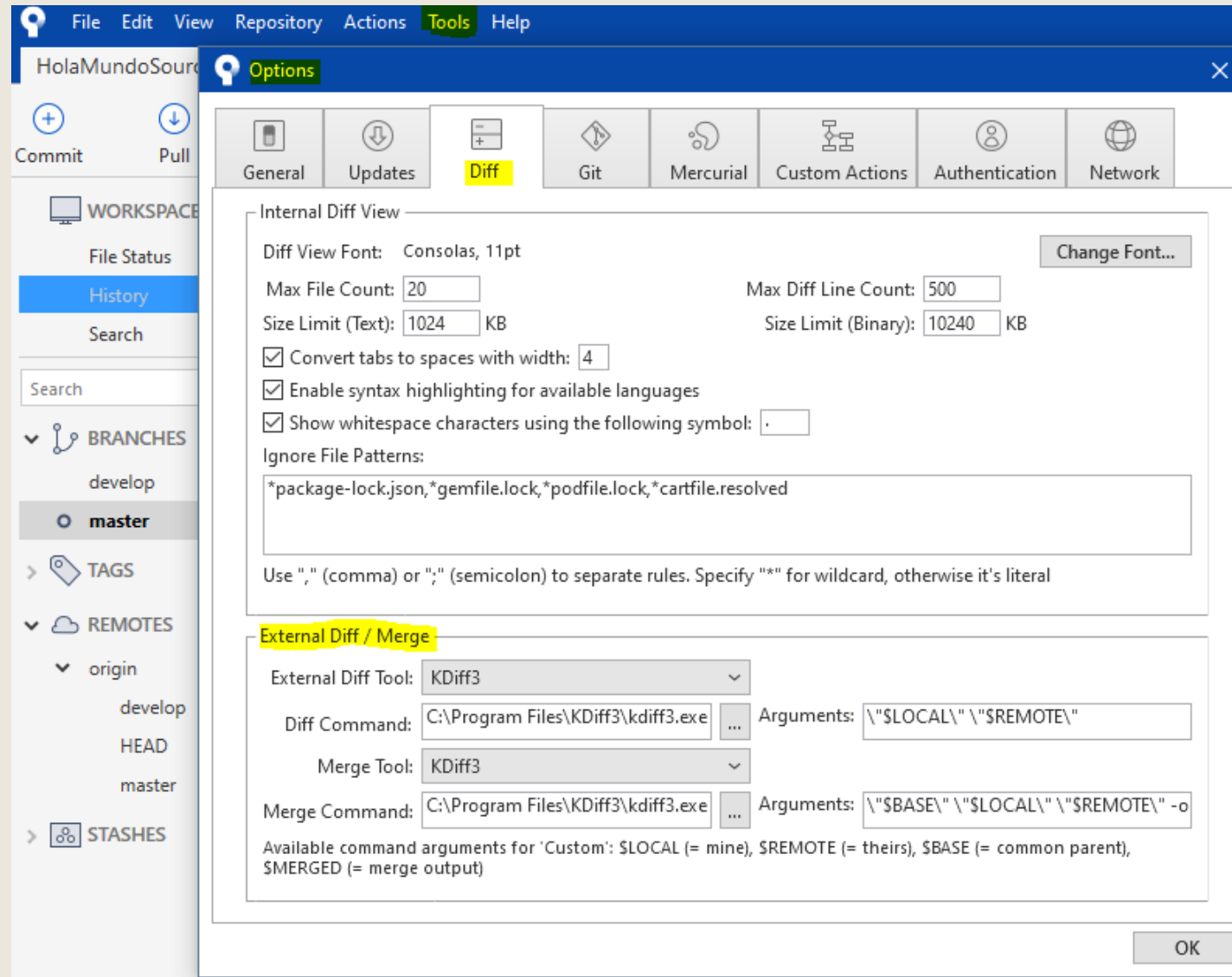


Ejercicio 1 con Sourcetree

- Cerrar el proyecto anterior y abrir el que acabamos de clonar para el sourcetree.
- Modificar el texto de la función en develop.
- Commit
- Comprobar repositorio
- Push
- Comprobar repositorio
- Cambiar entre las ramas y:
 - Comprobar que no existe el mismo código.
 - Ejecutar el código y ver las distintas salidas por consola.
- Crear feature
 - Añadir un mensaje nuevo y cuando esté probado, cerrar la feature.
- Crear release 1.0
 - Modificar alguno de los textos o añadir uno nuevo. Probarlo y cerrarla.
- Crear hotfix 1.1
 - Realizar una nueva modificación, probarla y cerrarla.

Resolución de conflictos (1)

- Se utilizará **KDiff3** como herramienta para resolver los conflictos en el código al hacer merge.
- Para asociar KDiff al **Sourcetree**:



- BRANCHES
 - master
 - new_branch_t...
- TAGS
- REMOTES
- STASHES
- SUBMODULES
- SUBTREES

- Open
- Show In Finder
- Copy Path To Clipboard
- Open In Terminal
- Quick Look

- External Diff

- Create Patch...
- Apply Patch...

- Add to index
- Unstage from index
- Remove
- Stop Tracking
- Ignore...
- Commit Selected...
- Reset...
- Reset to Commit...

- Resolve Conflicts

- Custom Actions

- Log Selected...
- Annotate Selected...

- Copy...
- Move...

- Launch External Merge Tool

- Resolve Using 'Mine'
- Resolve Using 'Theirs'

- Restart Merge
- Mark Resolved
- Mark Unresolved

A (Base): c:\Users\MiguelAngel\Documents\TestSourceTree\TestSourceTree\Models\Class1.cs.BASE.8440.cs ... B: j:\Documents\TestSourceTree\TestSourceTree\Models\Class1.cs.LOCAL.8440.cs ... C: \Documents\TestSourceTree\TestSourceTree\Models\Class1.cs.REMOTE.8440.cs ...

Top line 12 Encoding: UTF-8-BOM Line end style: DOS Top line 12 Encoding: UTF-8-BOM Line end style: DOS Top line 12 Encoding: UTF-8-BOM Line end style: DOS

```

...public class Class1
...{
...    public string Property1 { get; set; }
...    public string Property2 { get; set; }
...    public string Property3 { get; set; }
...
...}
}

...public class Class1
...{
...    public string Property1 { get; set; }
...    public string Property2 { get; set; }
...    public string Property3 { get; set; }
...    public string Property4 { get; set; }
...    public string Property5_Master2 { get; set; }
...    public string Property7 { get; set; }
...
...}
}

...public class Class1
...{
...    public string Property1 { get; set; }
...    public string Property2 { get; set; }
...    public string Property3 { get; set; }
...    public string Property4 { get; set; }
...    public string Property5 { get; set; }
...    public string Property6 { get; set; }
...    public string Property7 { get; set; }
...    public string Property8 { get; set; }
...    public string Property9 { get; set; }
...
...}
}
    
```

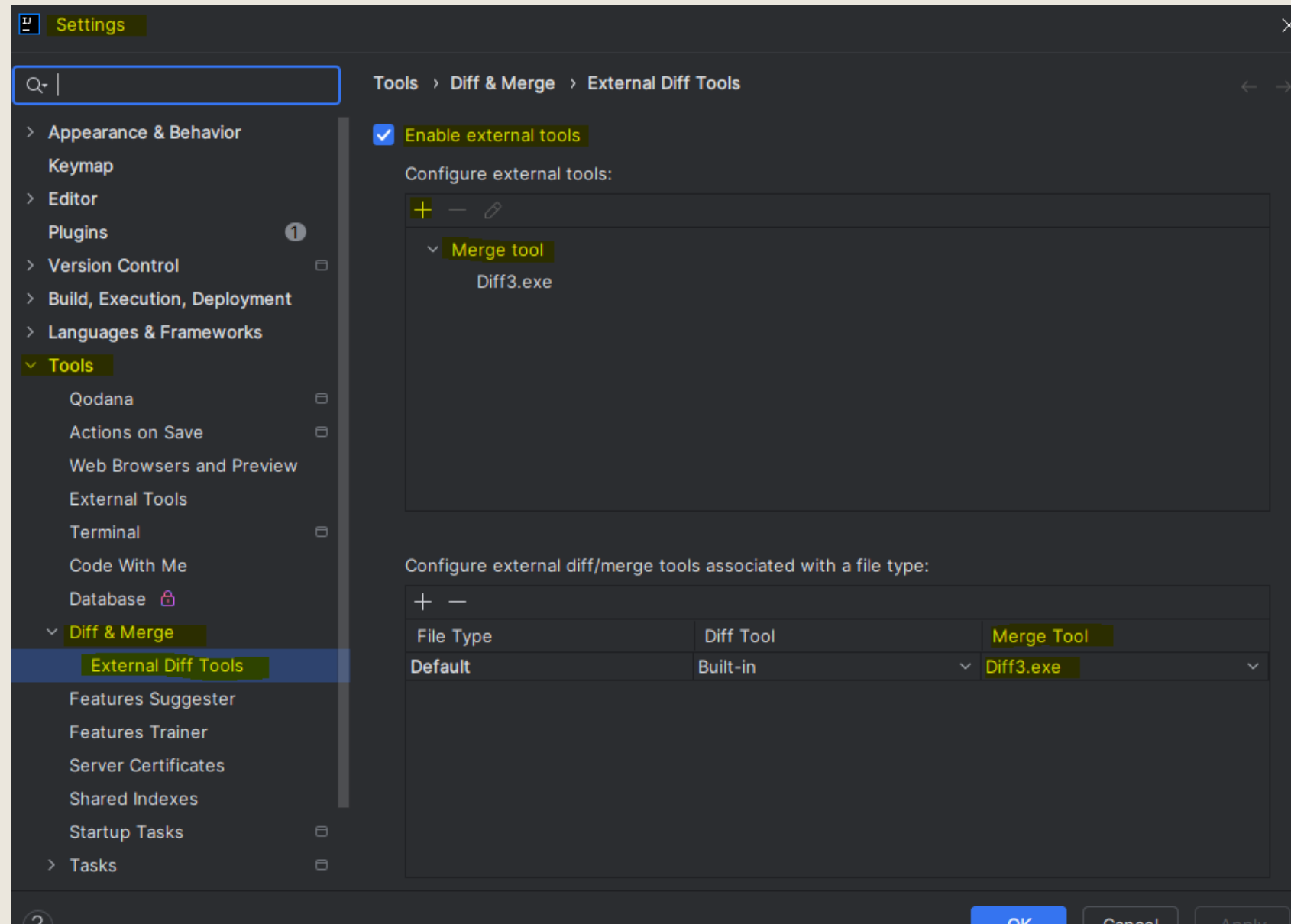
Output: C:\Users\MiguelAngel\Documents\TestSourceTree\TestSourceTree\Models\Class1.cs [Modified] Encoding for saving: Codec from C: UTF-8-BOM Line end style: DOS (A, B, C)

```

...    public string Property2 { get; set; }
...    public string Property3 { get; set; }
...    public string Property4 { get; set; }
...    public string Property5_Master2 { get; set; }
...
...    public string Property5 { get; set; }
...    public string Property6 { get; set; }
...    public string Property7 { get; set; }
...    public string Property8 { get; set; }
...    public string Property9 { get; set; }
...
...}
}
    
```


Resolución de conflictos (2)

- Para asociar KDiff a IntelliJ Idea:



Ejercicio 2 con Sourcetree en grupos para intentar resolver conflictos

- Clonar un proyecto de algún alumno
- El alumno modifica código y después commit y push
- Modificar código en local buscando generar conflicto
- Fetch y pull de los cambios del alumno
- Si hay conflicto, resolverlo con kdiff

Índice

1. ¿Qué es Kotlin?
2. Ventajas de Kotlin
3. Instalando IntelliJ Idea
4. Primera aplicación en Kotlin: Hola Mundo
5. **Lenguaje de programación Kotlin**

5. Lenguaje de programación Kotlin (1)

Documentación oficial: <https://kotlinlang.org/docs/home.html>

VARIABLES

Una variable no es más que un espacio de memoria en el que podemos guardar información. Dependiendo del tipo de información disponemos de diferentes variables, lo que nos permitirá evitar problemas como por ejemplo intentar sumar un número con una letra, ya que al ser tipos de variables diferentes no nos lo permitirá.

Las variables las declararemos del siguiente modo:

```
var numeroFavorito = 1
```

«*var*» nos dice que es una variable, a continuación le asignamos un nombre, en este caso «*numeroFavorito*» y le asignamos un valor.

Aunque no le hemos dicho el tipo de variable que es, Kotlin se lo ha tragado, esto es porque Kotlin busca el tipo de valor y le asigna un tipo por detrás, el problema de esto es que no siempre estará en lo correcto, así que debemos decirle nosotros el tipo.

```
var numeroFavorito: Int = 1
```

Esta vez le hemos puesto dos puntos y acto seguido hemos añadido el tipo de variable, en este caso *Int*. Ahora veremos con que variables podemos trabajar.

5. Lenguaje de programación Kotlin (2)

Tipos de variables:

- Variables numéricas
- Variables de texto
- Variables booleanas

Variables numéricas:

Estas variables se usan para asignar números, calcular tamaños y realizar operaciones matemáticas entre otras. Dentro de ellas se dividen en dos grupos, las variables enteras y reales.

❖ Integer:

Dentro de las enteras encontramos las variables **Int**, que es la más básica que usaremos, en la cual podremos insertar números naturales, pero hay una limitación: con una variable de tipo **Int** no podemos pasar de **-2,147,483,647** a **2,147,483,647**. Este será el número máximo y mínimo que soportará.

```
var numeroFavorito: Int = -231
```

5. Lenguaje de programación Kotlin (3)

❖ Long:

Son las variables **Long**. Básicamente es igual que Int, a diferencia de que soporta un rango mayor de números, de -9,223,372,036,854,775,807 a 9,223,372,036,854,775,807.

```
var numeroFavorito: Long = 47483647
```

❖ Float

Llegamos a las variables reales. A diferencia de las anteriores, estas pueden almacenar decimales. **Float** soporta **hasta 6 decimales**, pero también puede trabajar con números enteros. Esta variable cambia un poco respecto a las demás, pues habrá que meter una «f» al final del valor.

```
var numeroFavorito: Float = 1.93f
```

❖ Double

Terminamos las variables numéricas con los **Double**. Muy similar a float pero soporta **hasta 14 decimales**, pero también ocupa más memoria así que para un código óptimo deberemos pensar que tipo será el que más se adapte a nuestro proyecto. Tampoco habrá que añadir ningún tipo de letra al final del valor.

```
var numeroFavorito: Double= 1.932214124
```

5. Lenguaje de programación Kotlin (4)

Variables de texto:

❖ Char:

La variable **Char** nos permitirá guardar un carácter de cualquier tipo, lo único que debemos tener en cuenta es que va entre **comillas simples**.

```
var numeroFavorito: Char = '1'
```

```
var letraFavorita: Char = 'q'
```

```
var caracterFavorito: Char = '@'
```

Dentro de una variable Char podemos almacenar cualquier cosa.

❖ String:

La variable **String** será la que más usemos como norma general, nos permite almacenar cualquier tipo de caracteres pero a diferencia del Char podemos añadir la cantidad que queramos. Para ser exactos, una String no es más que una cadena de Char. Las cadenas deberán ir entre **comillas dobles**.

```
var numeroFavorito: String = "Mi número favorito es el 3"
```

```
var test: String = "Test. 12345!.$%&/"
```

5. Lenguaje de programación Kotlin (5)

Variables booleanas:

Nos queda una última variable muy sencilla, pero a la vez muy práctica. Se tratan de los **Booleanos**.

❖ Boolean:

Los Booleanos son variables que solo pueden ser verdaderas o falsas (true o false). Su uso es muy amplio, cuando trabajemos con las condiciones veremos más a fondo este tema. Para asignar un valor basta con añadir true o false sin comillas.

```
var estoyTriste: Boolean = false
```

```
var estoyFeliz: Boolean = true
```

Seguridad nula

Las variables de Kotlin no pueden retener valores nulos de manera predeterminada. Para que una variable retenga un valor nulo, debe ser de tipo *anulable*. Si deseas especificar que una variable es anulable, puedes agregar un sufijo a su tipo mediante ?.

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


5. Lenguaje de programación Kotlin (6)

Aunque hayamos visto que no es necesario definir el tipo de variable, teniendo unos conocimientos muy básicos no solo vamos a evitar errores sino también a optimizar nuestro código.

<https://play.kotlinlang.org> -> desarrollar kotlin online

5. Lenguaje de programación Kotlin (7)


Ejemplo 1. Trabajando con variables sin indicar el tipo

```
fun main() { • patriciasfo *  
    var a = 10  
    var b = 5  
    print("Suma: ")  
    println(a + b)  
    print("Resta: ")  
    println(a - b)  
    print("Multiplicación: ")  
    println(a * b)  
    print("División: ")  
    println(a / b)  
    print("El módulo (resto): ")  
    println(a % b)
```



5. Lenguaje de programación Kotlin (8)

Ejemplo 2. Trabajando con variables añadiendo el tipo a los datos

```
fun main() {  patriciasfo *  
    var a: Float = 10.5f  
    var b: Int = 5  
    print("Suma: ")  
    var resultado = a + b  
    print(resultado)  
}
```

Ejemplo 3. Añadiendo el tipo a 'resultado'

- ¿Qué tipo de dato tiene que ser?
- Si queremos que el resultado sea un número entero. ¿Qué tendremos que hacer?

5. Lenguaje de programación Kotlin (9)

- `toInt()` permite convertir una variable a *Integer*.
- `toFloat()` permite convertir una variable a *Float*.

Tenemos varios métodos `toX()` para cambiar los valores a placer. Debemos tener cuidado porque si por ejemplo, a un `String` lo intentamos pasar a un número, nos dará una excepción y el código no funcionará.

El resultado del ejercicio anterior sería:

```
fun main() {  @ patriciasfo *  
    var a: Float = 10.5f  
    var b: Int = 5  
    print("Suma: ")  
    var resultado: Int = a.toInt() + b  
    print(resultado)  
}
```

5. Lenguaje de programación Kotlin (10)

CONCATENACIÓN

Ahora imaginad que tenemos dos **String**, obviamente no los podemos sumar para mostrarlos, así que para eso está la concatenación, que no es más que a través de un atributo poder poner más de una variable.

Solo debemos añadir las variables entre comillas dobles y anteponer a cada una de ellas el símbolo \$.

```
fun main() {  @ patriciasfo *  
    val greeting = "Hola, me llamo"  
    val name = "Patricia"  
    println("$greeting $name")  
}
```

Daros cuenta que he añadido un espacio entre ambas variables, sino, aparecerá el resultado sin ese espacio.

5. Lenguaje de programación Kotlin (11)

¿Alguna otra diferencia en el código del ejemplo anterior?

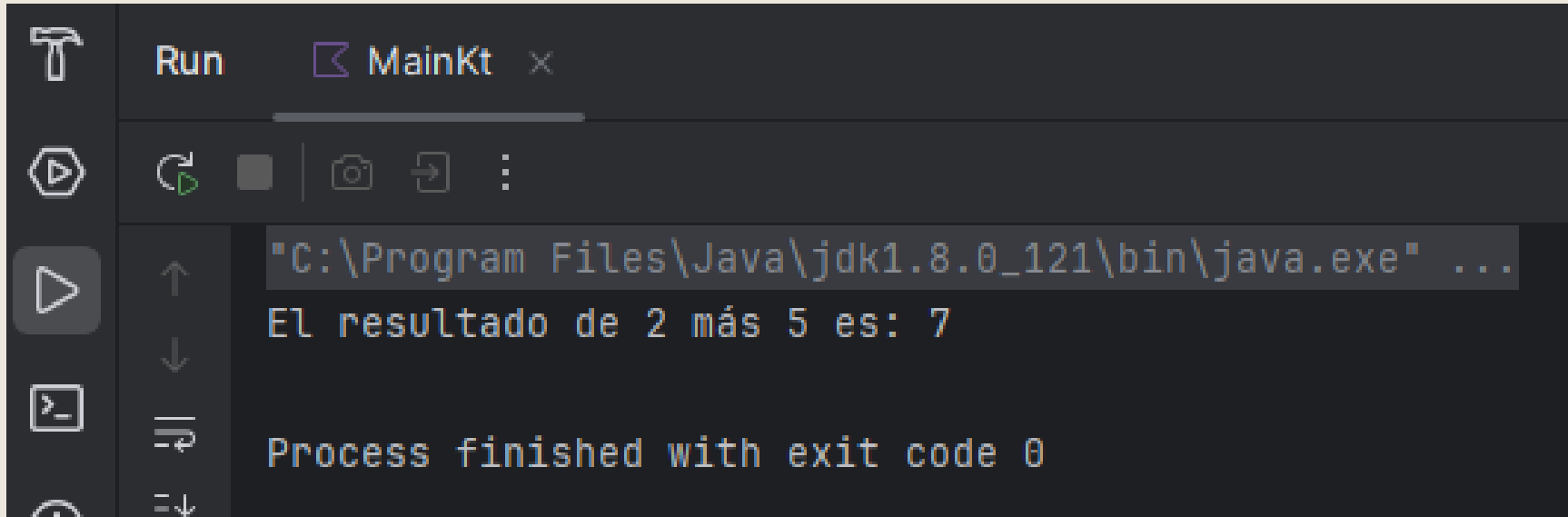
Esta vez he puesto **val** en vez de **var**. Esto se debe a que Kotlin prefiere que las variables sean inmutables, por lo que si nunca vamos a cambiar el valor de una variable, debemos poner **val** en lugar de **var**.

Con la concatenación también podemos hacer operaciones. Debemos tener cuidado, pero si lo controlamos no debería haber problema.

```
fun main() {  patriciasfo *
    val introduction = "El resultado de"
    val plus = "más"
    val firstNumber = 2
    val secondNumber = 5
    println("$introduction $firstNumber $plus $secondNumber es: ${firstNumber + secondNumber}")
}
```

5. Lenguaje de programación Kotlin (12)

Lo primero que hemos hecho es concatenar una frase, después hemos vuelto a añadir un \$ y entre llaves ({}) hemos metido la operación. El resultado sería este:



The screenshot shows the 'Run' console of an IDE. The title bar indicates the file 'MainKt'. The console output is as follows:

```
"C:\Program Files\Java\jdk1.8.0_121\bin\java.exe" ...  
El resultado de 2 más 5 es: 7  
  
Process finished with exit code 0
```

Otra solución:

```
println(introduccion + " " + firstNumber + " más " + secondNumber + " es: " + (firstNumber + secondNumber))
```

5. Lenguaje de programación Kotlin (13)

FUNCIONES

Una función no es más que un conjunto de instrucciones que realizan una determinada tarea y la podemos invocar mediante su nombre.

Declarando funciones

Las funciones se declaran usando la palabra clave **fun**, seguida del nombre del método, los paréntesis donde declararemos los valores de entrada y unas llaves que limitan la función. Por ejemplo la función main.

Ejercicio:

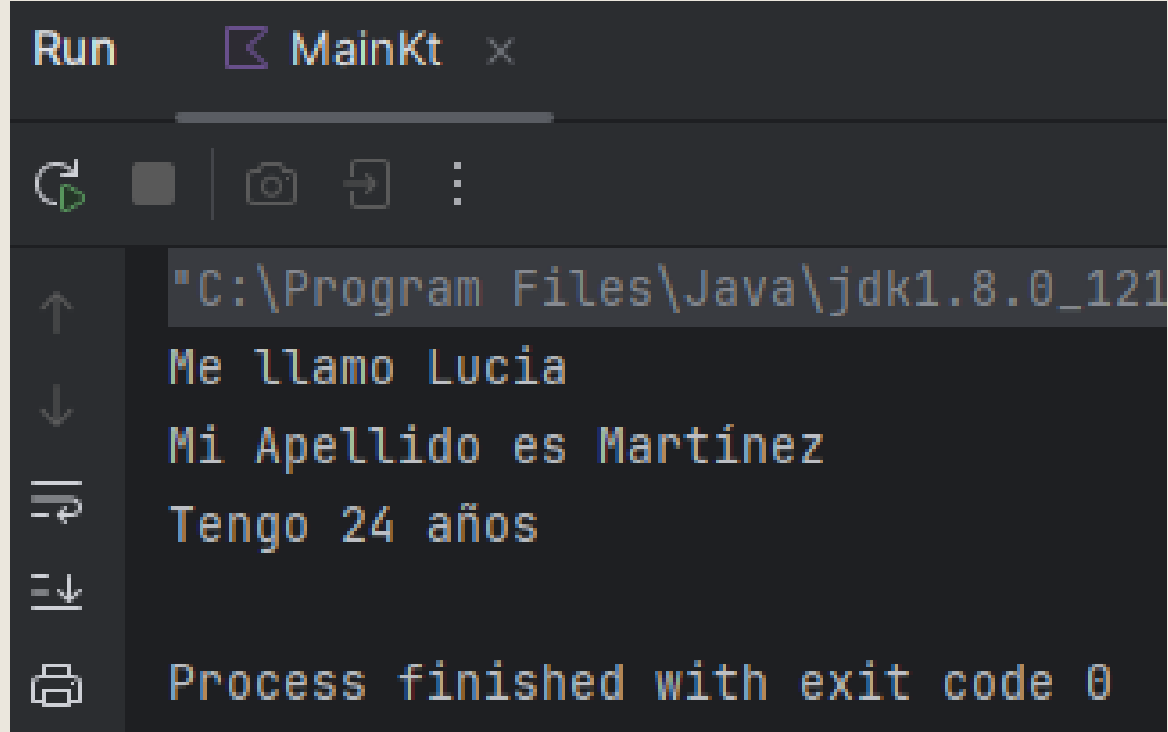
Declarar 3 funciones:

1. Una que escriba por pantalla: “Me llamo -----”
2. Otra que muestre un apellido: “Mi Apellido es -----”
3. Otra que muestre una edad: “Tengo años”

Llamarlas desde el main.

5. Lenguaje de programación Kotlin (14)

```
fun mostrarMiNombre(){ new *  
    println("Me llamo Lucia")  
}  
  
fun mostrarMiApellido(){ new *  
    println("Mi Apellido es Martínez")  
}  
  
fun mostrarMiEdad(){ new *  
    println("Tengo 24 años")  
}  
  
fun main() { new *  
    mostrarMiNombre()  
    mostrarMiApellido()  
    mostrarMiEdad()  
}
```



The screenshot shows the 'Run' window of an IDE. At the top, it says 'Run' and 'MainKt'. Below that, there are icons for running, stopping, and other actions. The output area shows the following text:

```
"C:\Program Files\Java\jdk1.8.0_121  
Me llamo Lucia  
Mi Apellido es Martínez  
Tengo 24 años  
Process finished with exit code 0
```

5. Lenguaje de programación Kotlin (15)

Si os fijáis en el código anterior, tenemos 4 métodos. 3 de ellos están destinados para una sola función (mostrar nombre, edad y apellidos) pero no se ejecutarán a no ser que sean llamados. Por ello el cuarto método, que es el que ejecutará el código, los llamará en el orden que le pongamos.

Ejercicio:

Modificar el orden de las llamadas a las funciones.

Funciones con parámetros de entrada

Ahora vamos a ver las funciones con parámetros de entrada, que son iguales, pero al llamarlas habrá que mandarle las variables que necesite.

```
fun showMyInformation(name: String, lastName: String, age: Int){  
    println("Me llamo $name $lastName y tengo $age años.")  
}
```

5. Lenguaje de programación Kotlin (16)

Como se puede observar, tiene tres parámetros de entrada, la forma de declararlos es muy fácil: el **nombre** de la variable, seguida de **dos puntos** y el **tipo de variable**, aquí si es **obligatorio** definir el tipo.

Obviamente al llamar al método podemos pasarle variables recuperadas de otros métodos y demás.

Funciones con parámetros de salida

Nos queda por ver cómo una función puede devolver un resultado o lo que haga nuestro método. La única limitación es que solo se puede devolver un parámetro, aunque para eso tenemos los métodos (ya lo veremos más tarde).

```
fun add(firsNumber: Int, secondNumber: Int) : Int{  
    return firsNumber + secondNumber  
}  
  
fun main() { new *  
    var result = add( firsNumber: 5, secondNumber: 10)  
    println(result)  
}
```

5. Lenguaje de programación Kotlin (17)

Como en el ejemplo anterior, añadimos los parámetros de entrada pero esta vez, al cerrar los paréntesis pondremos el tipo de variable que debe devolver nuestra función. Luego la función hará todo lo que tenga que hacer y cuando tenga el resultado, lo devolveremos con la palabra clave **return**.

Si el método es muy fácil, podemos evitar las llaves y simplificar la función un poco más.

```
fun add(firsNumber: Int, secondNumber: Int) : Int = firsNumber + secondNumber
```

INSTRUCCIONES CONDICIONALES

Las instrucciones condicionales permiten realizar la lógica en función del resultado de una variable o condición.

La condición if

Es de las más habituales y realizará una función o varias solo si la condición que hemos generado es verdadera.

5. Lenguaje de programación Kotlin (18)

```
fun add(firsNumber: Int, secondNumber: Int) : Int = firsNumber + secondNumber

fun main(){  @ patriciasfo *
    var result = add( firsNumber: 5,  secondNumber: 10)

    if (result > 10){
        println("El resultado es mayor que 10")
    }
}
```

Simplemente debemos añadir la condición entre paréntesis. No solo podemos usar operadores como <, >, =, !=, sino que podemos comparar Strings a través del doble igual «==».

```
fun main(){  @ patriciasfo *
    val name = "Patricia"

    if (name == "Patricia"){
        println("Se llama Patricia")
    }
}
```

5. Lenguaje de programación Kotlin (19)

If-else

Hay veces que necesitaremos más de un if, y por eso está la palabra clave else que actuará como segundo condicional.

```
fun main(){ * patriciasfo *  
    val name = "Patricia"  
  
    if (name == "Patricia"){  
        println("Se llama Patricia")  
    } else {  
        println("No se llama Patricia")  
    }  
}
```

Anidamiento

Aunque no es la práctica más correcta y no deberíamos abusar, en determinadas ocasiones necesitamos más condiciones, y aunque podríamos recurrir a otras instrucciones, lo podemos hacer con if.

5. Lenguaje de programación Kotlin (20)

```
if (animal == "dog"){  
    println("Es un perro")  
} else if(animal == "cat"){  
    println("Es un gato")  
} else if(animal == "bird"){  
    println("Es un pájaro")  
} else{  
    println("Es otro animal")  
}
```

Aquí hemos hecho varios anidamientos y aunque funciona, no es lo más correcto.

Podemos usar más de una condición a la vez gracias a los operadores **and** (&&) y **or** (||).

```
//solo entrará si cumple ambas condiciones  
if (animal == "dog" && raza == "labrador"){  
    println("Es un perro de raza labrador")  
}  
  
//Entrará si es verdadera una de las condiciones  
if (animal == "dog" || animal == "gato"){  
    println("Es un perro o un gato")  
}
```

5. Lenguaje de programación Kotlin (21)

EXPRESIÓN WHEN

Siguiendo con el control de flujo, la siguiente expresión que debemos ver es **when**. Esta nos permite realizar una o varias acciones dependiendo del resultado recibido. También se podría hacer anidando if-else, pero no sería lo correcto. La forma óptima es esta. Para los que tengan conocimientos básicos de programación en otro lenguaje, when es el sustituto del switch.

```
fun getMonth(month : Int) { new *
    when (month) {
        1 -> print("Enero")
        2 -> print("Febrero")
        3 -> print("Marzo")
        4 -> print("Abril")
        5 -> print("Mayo")
        6 -> print("Junio")
        7 -> print("Julio")
        8 -> print("Agosto")
        9 -> print("Septiembre")
        10 -> print("Octubre")
        11 -> print("Noviembre")
        12 -> print("Diciembre")
        else -> {
            print("No corresponde a ningún mes del año")
        }
    }
}
```

```
fun main(){ patriciasfo
    getMonth( month: 2)
}
```


5. Lenguaje de programación Kotlin (22)

El ejemplo es muy sencillo. La función `getMonth` recibe un *Int* que se lo mandamos al *when*, una vez ahí comprobará todos los casos disponibles (aquí tenemos de 1 a 12). Si concuerda con algún valor, automáticamente entrará por ahí y realizará la función oportuna, en este caso pintar el mes.

Si por el contrario no encuentra ningún caso igual, entrará por el *else*. Dicho ***else* no es obligatorio** así que se puede quitar, y si no entra por ningún caso pues simplemente no mostrará nada.

La expresión ***when*** no solo soporta números, sino que puede trabajar con textos y expresiones.

En este ejemplo podemos ver cómo separar varios valores a través de comas.

```
fun getMonth(month : Int) {  
    when (month) {  
        1,2,3 -> print("Primer trimestre del año")  
        4,5,6 -> print("segundo trimestre del año")  
        7,8,9 -> print("tercer trimestre del año")  
        10,11,12 -> print("cuarto trimestre del año")  
    }  
}
```

5. Lenguaje de programación Kotlin (23)

Si son rangos más altos tenemos la posibilidad de usar `in` y `!in` para trabajar con **arrays** y **rangos** (lo veremos más tarde).

```
fun getMonth(month : Int) { new *
    when (month) {
        in 1 ≤ .. ≤ 6 -> print("Primer semestre")
        in 7 ≤ .. ≤ 12 -> print("segundo semestre")
    }
}
```

Con esto podemos comprobar si está entre una cantidad de números específicos (en este caso entre 1 y 6 y 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) { new *
    when (month) {
        in 1 ≤ .. ≤ 6 -> print("Primer semestre")
        in 7 ≤ .. ≤ 12 -> print("segundo semestre")
        !in 1 ≤ .. ≤ 12 -> print("no es un mes válido")
    }
}
```

5. Lenguaje de programación Kotlin (24)

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

```
fun result(value: Any){ new *  
    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")  
    }  
}
```

Si es `Int`, sumará 1 al valor; si es un `String` lo concatenará al texto que vemos arriba y si es un `Booleano` nos pintará un resultado dependiendo si es `true` o `false`.

Para finalizar mostrarnos también que podemos guardar el resultado de un `when` automáticamente.

```
fun result(month : Int){ new *  
    val response : String = when (month) {  
        in 1 ≤ .. ≤ 6 -> "Primer semestre"  
        in 7 ≤ .. ≤ 12 -> "segundo semestre"  
        !in 1 ≤ .. ≤ 12 -> "no es un mes válido"  
        else -> "error"  
    }  
}
```

5. Lenguaje de programación Kotlin (25)

Hemos declarado la variable de tipo String, pero podríamos hacerla de tipo **Any** si no tuviéramos claro el resultado de la expresión. Aquí si es obligatorio añadir un else, aunque como podéis apreciar, podemos quitar los paréntesis de dicha condición.

Ejercicios:

- Ejercicio 1. Notificaciones móviles
- Ejercicio 2. Precio de la entrada de cine
- Ejercicio 3. Conversor de temperatura

5. Lenguaje de programación Kotlin (26)

ENTRADA ESTÁNDAR

Kotlin ofrece dos formas principales de leer desde la entrada estándar: la clase **Scanner**, similar a Java, y la función **readln()**.

■ Con Java Scanner

Normalmente se accede a la entrada estándar a través del objeto **System.in**. Es necesario importar la clase **Scanner** e inicializarla con un objeto **System.in** que represente el flujo de la entrada estándar y dicte cómo leer los datos y utilizar métodos como **.nextLine()**, **.next()** y **.nextInt()** para leer distintos tipos de datos.

Otros métodos útiles para leer la entrada con Java Scanner son **.hasNext()**, **.useDelimiter()**, y **.close()**.

Utilice siempre **.close()** cuando haya terminado de utilizar Java Scanner. Al cerrar Java Scanner, se liberan los recursos que consume y se garantiza el correcto comportamiento del programa.

5. Lenguaje de programación Kotlin (27)

```
fun main() { new *  
    // Inicializa Scanner  
    val scanner = Scanner(System.`in`)  
  
    // Lee una línea entera. Por ejemplo: "Hola, Kotlin"  
    val line = scanner.nextLine()  
    print(line)  
    // Hola, Kotlin  
  
    // Lee un string. For example: "Hola"  
    val string = scanner.next()  
    print(string)  
    // Hola  
  
    // Lee un número entero. For example: 123  
    val num = scanner.nextInt()  
    print(num)  
    // 123  
}
```

5. Lenguaje de programación Kotlin (28)

■ Con Readln()

Es la forma más sencilla de leer la entrada estándar. Esta función lee una línea de texto de la entrada estándar y la devuelve como una cadena:

```
fun main() { new *  
    // Lee un string. Por ejemplo: Patricia  
    val name = readln()  
  
    // Lee un string y lo convierte a entero. Por ejemplo: 43  
    val age = readln().toInt()  
  
    println("Hola, $name! Tienes $age años."  
    // Hola, Patricia! Tienes 43 años.  
}
```

5. Lenguaje de programación Kotlin (29)

Ejercicio 4

Modificar los ejercicios 1 a 3 de tal forma que ahora podamos introducir los datos desde la entrada estándar utilizando **readln()**.

- Ejercicio 1. El usuario debe poder indicar el número de notificaciones.
- Ejercicio 2. El usuario debe poder indicar la edad de la persona que acude al cine y si es lunes o no.
- Ejercicio 3. El usuario debe poder indicar la temperatura que quiere convertir, en qué escala está y a qué escala queremos pasarla.

5. Lenguaje de programación Kotlin (30)

ARRAYS

Los arrays son secuencias de datos, del mismo tipo e identificados por un nombre común. Para hacerlo más fácil de entender imaginemos que tenemos que almacenar los 7 días de la semana, podríamos crear 7 variables de tipo String o almacenarlas todas en un solo array.

```
val semana = arrayOf("Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo")
```

Para acceder a cada uno de los valores, lo haremos a través de la posición. Por ejemplo, imaginemos un edificio, cada valor se almacena en una planta, por lo que el primer valor estará en la posición 0, el segundo en la 1 y así con cada uno de ellos. Recordar que se empieza en la posición 0.

Entonces podremos acceder a cada uno de los valores gracias a la función **get()** que nos devolverá el valor de dicha posición.

```
println(semana .get(0))
```

Si pusiéramos en el get una posición que no tiene, por ejemplo la 7 nos daría una excepción al ejecutarse: *ArrayIndexOutOfBoundsException*; y es por ello por lo que al trabajar con arrays debemos tener bien claro el tamaño del array.

5. Lenguaje de programación Kotlin (31)

La función **size** devolverá el tamaño del array. \longrightarrow `weekDays.size`

Los arrays tienen una serie de limitaciones, entre ellas que tienen que tener un tamaño fijo, y será el número de valores que le asignemos al instanciarla. Eso significa que siempre va a tener un tamaño de 7 y no podremos añadir más datos, pero si cambiarlos a través de la función **set()**.

```
weekDays.set(0, "Horrible lunes") //Contenía Lunes
```

```
weekDays.set(4, "Por fin viernes") //Contenía Viernes
```

La función **set()** recibe dos parámetros: el primero es la posición a la que queremos acceder y el segundo el es nuevo valor a reemplazar. Hay que tener en cuenta que el valor que le mandemos debe ser del mismo tipo, por ejemplo este array es de Strings, por lo que no podemos pasar un Int.

Otra forma de definir un array sería la siguiente: imaginar que se quiere un array que contenga los números del 0 al 20 -> `val miArrayNumerico = (0 .. 20)`

5. Lenguaje de programación Kotlin (32)

Recorriendo Arrays

El **bucle for()** nos permite entre otras, recorrer un array entero, posición por posición y acceder a cada uno de los parámetros que contiene. Vamos a volver a pintar los 7 días de la semana con este método.

```
fun main() { new *
    val semana = arrayOf("Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo")

    for (posicion in semana.indices){
        println(semana.get(posicion))
    }
}
```

El **for** necesitará una variable, en este caso «*posición*» que irá teniendo el valor de cada una de las posiciones del array. Su funcionamiento es muy sencillo: cuando pasa por el for por primera vez, tendrá valor 0, comprueba el tamaño de *semana* y si es mayor, entra a la función y hace lo que le pidamos y vuelve al inicio, así hasta llegar a 6 que será la última posición de la array.

5. Lenguaje de programación Kotlin (33)

También nos permite sacar tanto el índice como el valor directamente, para ello haríamos lo siguiente.

```
fun main() { new *
    val semana = arrayOf("Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo")

    for ((posición, valor) in semana.withIndex()) {
        println("La posición $posición contiene el valor $valor")
    }
}
```

Si por el contrario solo interesa el contenido podríamos hacer directamente un **for in** sin acceder a la posición, solo al contenido

```
fun main() { new *
    val semana = arrayOf("Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo")

    for (dia in semana) {
        println(dia)
    }
}
```

5. Lenguaje de programación Kotlin (34)

LISTAS

El problema de los arrays es la limitación al definirlos, tenemos que saber de antemano su tamaño. La solución → las listas.

Tipos de listas

- **Inmutables:** no pueden variar sus datos, como los arrays.

```
val diasSemana: List<String> = listOf("Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo")
```

Funciones útiles:

- *diasSemana.size()* → muestra el tamaño de la lista
- *diasSemana.get(3)* → devuelve el valor de la posición 3
- *diasSemana.first()* → devuelve el primer valor
- *diasSemana.last()* → devuelve el último valor

Se puede trabajar fácilmente con las listas inmutables a excepción de poder añadir más elementos.

5. Lenguaje de programación Kotlin (35)

- **MutableList:** además de las características, funcionalidades, etc de las anteriores listas, nos da la posibilidad de ir rellenando la lista según lo necesitemos. El problema: es más ineficiente con la memoria.

```
var diasSemana: MutableList<String> = mutableListOf("Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado")
```

Funciones útiles:

- `println(diasSemana)` → // [Lunes, Martes, Miércoles, Jueves, Viernes, Sábado]
- `diasSemana.add("Domingo")` → Por defecto se añaden al final de la lista, en la última posición.
- `diasSemana.add(0, "Semana:")` → se añade el texto en la posición 0.

Las listas mutables pueden estar **vacías** o contener un valor **NULL**, es decir, una de las posiciones de la lista puede tener un valor nulo y si intentamos acceder a él se produce un **crash** y la aplicación se romperá.

ArrayList es una implementación de la interface **MutableList**:

```
var arrayList: ArrayList<String> = arrayListOf<String>()
```

5. Lenguaje de programación Kotlin (36)

```
var mutableList: MutableList<String> = mutableListOf()
```

Funciones para trabajar con listas mutables **de forma segura**:

- *mutableList.none()* → devuelve true si la lista está vacía
- *mutableList.firstOrNull()* → devuelve el primer campo, y si no hay, devuelve null
- *mutableList.elementAtOrNull(2)* → devuelve el elemento de la posición 2, y si no hay, devuelve null
- *mutableList.lastOrNull()* → devuelve el último campo, y si no hay, devuelve null

Función Filter

Permite filtrar en la lista a través de una o varias condiciones que pongamos.

```
val resultado = diasSemana.filter {it == "Lunes" || "Juernes"}
```

```
val resultado = diasSemana.filter {it[0] == 'M'}
```

5. Lenguaje de programación Kotlin (37)

Recorriendo listas

Al igual que en los arrays, las listas se pueden recorrer de diferentes maneras:

1. Devuelve el contenido de cada uno de los valores de la lista:

```
for (item in diasSemana) {  
    print(item)  
}
```

1. Devuelve la posición y el contenido de cada uno de los valores de la lista:

```
for ((indice, item) in diasSemana.withIndex()) {  
    println("La posición $indice contiene $item")  
}
```

1. Accede al contenido a través del iterador

```
val mutableList: MutableList<String> = mutableListOf("Lunes", "Martes", "Miércoles", "Jueves", "Viernes",  
"Sábado")  
val newListEmpty: MutableList<String> = mutableListOf()  
mutableList.forEach {  
    newListEmpty.add(it+":")  
}  
print(newListEmpty)
```


5. Lenguaje de programación Kotlin (38)

Mapas

Los mapas son colecciones de pares de datos. Cada par de datos se corresponde con una clave y un valor. Las claves son únicas y los mapas sólo pueden guardar un único valor para cada clave.

Existen mapas inmutables y mapas mutables.

Mapas inmutables:

Son mapas de sólo lectura, los valores asociados a las claves nunca cambian y no se pueden añadir ni eliminar pares de datos.

```
val map = mapOf(clave to valor, clave to valor, ...)
```

Ejemplo: `val mapa = mapOf(1 to "Pera", 2 to "Manzana")` //mapa de enteros to strings

Mapas mutables:

Son mapas sobre los que se puede leer y escribir.

```
val map = mutableMapOf(clave to valor, clave to valor, ...)
```

Para crear un **mapa vacío**:

```
val map = mapOf<String, Int>() ó val map = mutableMapOf<String, Int>()
```

5. Lenguaje de programación Kotlin (39)

Funciones:

- ***map.entries***: se accede a todas las entradas del mapa
- ***map.keys***: se accede a todas las claves del mapa
- ***map.values***: se accede a todos los valores del mapa
- ***map.size* o *map.count()***: indica el tamaño del mapa
- Para obtener un valor del mapa existen múltiples maneras:
 - ***map[clave]***
 - ***map.getValue(clave)***
 - ***map.getDefault(clave, valorPorDefecto)***: obtiene el valor de la clave, y si la clave no existe devuelve el valorPorDefecto.
 - ***map.getOrElse(clave, funcion)***: obtiene el valor de la clave, y si la clave no existe devuelve la respuesta de la función.
- ***map.containsKey(clave)***: comprueba si el mapa contiene la clave indicada.
- ***map.containsValue(valor)***: comprueba si el mapa contiene el valor indicado.

5. Lenguaje de programación Kotlin (40)

- `map.clear()`: elimina todos los elementos de un mapa
- `map.put(clave, valor)`: añade un par clave - valor al mapa
- `map.putAll(mapa)`: actualiza un mapa con todos los pares clave-valor de otro mapa
- `map.remove(clave)`: elimina del mapa el par cuya clave coincide con la indicada
- `map.remove(clave, valor)`: sólo elimina del mapa el par si coinciden la clave y el valor

Recorrer un mapa

```
val mapa = mapOf(1 to "Pera", 2 to "Manzana")  
for (fruta in mapa) println ("${fruta.key} - ${fruta.value}")
```

HashMap y LinkedHashMap

Ambas son implementaciones de la interfaz **MutableMap**.

HashMap no ofrece garantías respecto al orden de enumeración de claves, valores y colecciones de entradas.

LinkedHashMap preserva el orden de inserción de las entradas durante la iteración.

5. Lenguaje de programación Kotlin (41)

- Además de los usos del **bucle for** en listas, mapas (diccionarios), etc. también lo podemos utilizar para recorrer rangos:

```
for (x in 0 ≤ .. ≤ 10){  
    println(x)  
}
```

Se ejecuta el bucle 11 veces. ¿Y si queremos que se ejecute 10 veces?

```
for (x in 0 ≤ until < 10){  
    println(x)  
}
```

Con **until** no se tiene en cuenta el último número

```
for (x in 0 ≤ .. ≤ 10 step 2){  
    println(x)  
}
```

Para realizar incrementos distintos de 1 se utiliza **step**

```
for (x in 10 ≥ downTo ≥ 0 step 3){  
    println(x)  
}
```

Para realizar cuenta regresiva se utiliza **downTo**

5. Lenguaje de programación Kotlin (42)

BUCLE WHILE

Sirve para repetir una misma acción mientras que la condición que se valida es verdadera.

```
var x = 0
while (x < 10) {
    println(x)
    x++ // x += 2
}
```

→ Para evitar bucles infinitos

NULL SAFETY

Seguridad contra nulos. Para evitar que se produzca NullPointerException, se declaran variables con ? después del tipo. También se pueden realizar llamadas a métodos.

5. Lenguaje de programación Kotlin (43)

NULL SAFETY

```
val myString = "Prueba"  
//myString = null -> error de compilación  
println(myString)
```

```
var mySafetyString: String? = "Prueba"  
mySafetyString = null  
println(mySafetyString?.length)
```

```
mySafetyString = "Ahora con texto"  
mySafetyString?.let { // Se ejecuta cuando mySafetyString no sea nulo  
    println(it) // it es el valor de mySafetyString forzando a que no sea nulo  
} ?: run { // Se ejecuta cuando mySafetyString sea nulo.  
    println(mySafetyString)  
}
```

5. Lenguaje de programación Kotlin (44)

FUNCION LAMBDA

Una **función lambda** es un literal de función que puede ser usado como expresión. Esto quiere decir que es una función que no está ligada a un identificador y que puedes usar como valor.

Por ejemplo, si tenemos la función $f(s) = s + 2$, en Kotlin podemos expresarla como una declaración de función separada, así:

```
fun sumarDos(s: Int) = s + 2
```

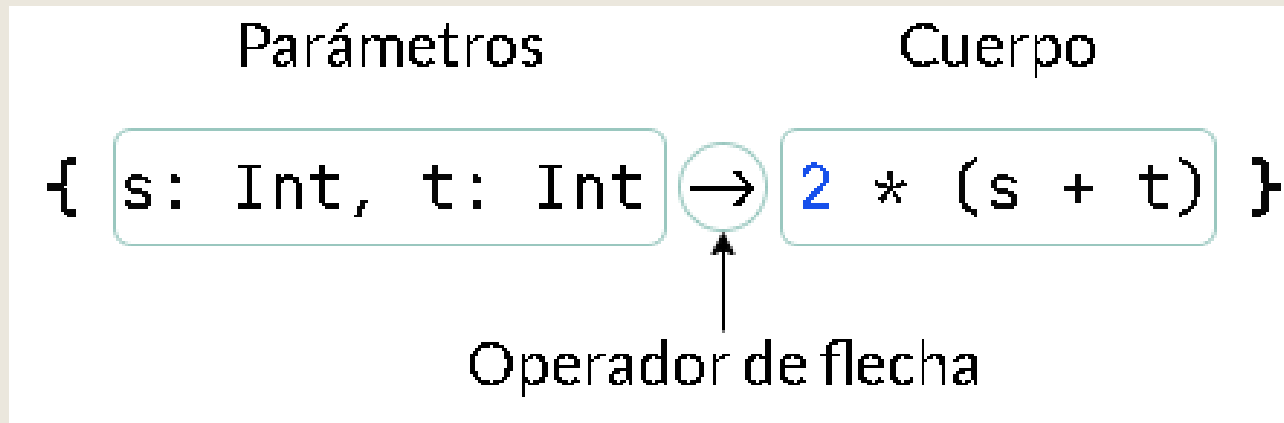
Al ser reescrita como lambda, tendrías lo siguiente:

```
{s: Int -> s + 2}
```

Definir la función de esta forma te permitirá usarla como un valor en diferentes situaciones, como pasarla como argumento de una función o almacenarla en una variable.

5. Lenguaje de programación Kotlin (45)

Sintaxis



*Para cada par de s y t
corresponde el valor
 $2*(s+t)$*

Un literal lambda va entre dos llaves `{ }` con los siguientes componentes:

- Lista de **parámetros**: cada parámetro es una declaración de variable. Esta lista es **opcional**.
- **Operador de flecha** `->`: se omite si no se usa lista de parámetros.
- **Cuerpo** de lambda: son las sentencias que van después del operador de flecha.

5. Lenguaje de programación Kotlin (46)

```
private fun lambdas(){ new *  
    val listaNumeros = arrayListOf(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
    val listaFiltrados = listaNumeros.filter {  
        it > 5 // con it accedemos a cada uno de los valores de la lista  
    }  
    println(listaFiltrados)  
}
```

← Lambda del sistema

La lambda devuelve el resultado en la última línea

```
private fun lambdas(){ new *  
    val listaNumeros = arrayListOf(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
    val listaFiltrados = listaNumeros.filter { miEntero ->  
        if (miEntero == 1){  
            return@filter true  
        }  
        miEntero > 5  
    }  
    println(listaFiltrados)  
}
```

← Lambda definida
por nosotros

5. Lenguaje de programación Kotlin (47)

Función almacenada en una constante

```
private fun lambdas(){ new *  
    var miFuncionSuma = fun (x: Int, y: Int) : Int = x + y  
    var miFuncionMultiplicar = fun (x: Int, y: Int) : Int = x * y  
    println(miFuncionOperacion(x: 5, y: 10, miFuncionSuma))  
    println(miFuncionOperacion(x: 5, y: 10, miFuncionMultiplicar))  
    println(miFuncionOperacion(x: 5, y: 10, { x, y -> x - y })))  
}  
private fun miFuncionOperacion(x: Int, y: Int, miFuncion: (Int, Int) -> Int): Int{  
    return miFuncion(x, y)  
}
```

Para definir una función dentro de otra función sin tener que pasársela como constante

Ejercicio: Realiza el ejercicio 3 del Conversor de temperatura utilizando función lambda

5. Lenguaje de programación Kotlin (48)

Otro uso habitual de las lambdas es para definir funciones asíncronas: se ejecutan pero no sabemos en el tiempo cuándo nos va a devolver un resultado.

```
private fun lambdas(){ new *
    miFuncionAsincrona( name: "Patricia", {
        println(it) //Porque podemos acceder directamente y sólo hay un parámetro
    })
    println("Primer mensaje por pantalla")
    println("Segundo mensaje por pantalla")
    println("Tercer mensaje por pantalla")
}

private fun miFuncionAsincrona(name: String, hello: (String) -> Unit) { new *
    //Tipo Unit es el tipo Vacío, el equivalente al tipo void de Java
    val miNombre = "Hello $name!"
    // Aquí convertimos a la función en asíncrona creando un hilo
    thread {
        Thread.sleep( millis: 5000) // Para simular la asincronía se duerme al hijo 5 seg.
        hello(miNombre)
    }
}
```

5. Lenguaje de programación Kotlin (49)

CLASES

- Todos los tipos vistos hasta ahora están integrados en el lenguaje de programación Kotlin. Si deseas crear tus propios tipos de datos lo puede hacer utilizando clases. Para ello utiliza la palabra reservada **class**.
- Una clase tiene **propiedades** que forman el estado del objeto: son variables (var) y/o constantes (val) que pueden incluir los métodos get y set. Para usarlas simplemente accedemos por su nombre.
 - Se recomienda que se declaren como de sólo lectura, es decir, de tipo **val**.
 - Si no se especifica lo contrario, por defecto son variables públicas -> podemos acceder desde fuera de la clase.
 - Si se desea encapsular una propiedad y que no se pueda acceder directamente a ella, sino a través de una función, ésta debe definirse como **private**.
 - Se declaran entre paréntesis después del nombre de la clase o en el cuerpo de la clase.

```
class Contacto(val identificador: Int, var email: String)
```

```
class Contacto(val identificador: Int, var email: String) {  
    val category: String = ""  
}
```

5. Lenguaje de programación Kotlin (50)

- Se pueden inicializar las propiedades con valores por defecto.

```
class Contacto(val identificador: Int, var email: String = "ejemplo@gmail.com") {  
    val category: String = "trabajo"  
}
```

- Para obtener una instancia de una clase, hay que llamar a su constructor. El constructor principal inicializa una instancia de clase y sus propiedades en el encabezado de la clase.
 - Sintaxis del constructor por defecto: **class** Persona [*private*] **constructor** (nombre: String)
 - Si no tiene anotaciones ni modificadores de visibilidad se puede **omitir** la palabra **constructor**

```
fun main() {  @ patriciasfo *  
    val contacto = Contacto( identificador: 1, email: "mary@gmail.com")  
}
```

- Si no quieres que tu clase tenga un constructor público, hay que declarar el constructor primario como privado:

```
class SinConstructor private constructor()
```

5. Lenguaje de programación Kotlin (51)

- Los métodos get y set también se pueden personalizar.
 - Sintaxis completa de una propiedad:
`var <propertyName>[: <PropertyType>] [= <property_initializer>]
[<getter>] [<setter>]`
`val <propertyName>[: <PropertyType>] [= <property_initializer>]
[<getter>] -> no se permite el setter`
 - Si se personaliza un **getter**, se llamará cada vez que acceda a la propiedad (de esta manera puede implementar una propiedad calculada).

```
class Rectangulo(val ancho: Int, val alto: Int) {  
    val area: Int new *  
    |    get() = this.ancho * this.alto  
}
```

5. Lenguaje de programación Kotlin (52)

- Si se personaliza un **setter**, se llamará cada vez que asigne un valor a la propiedad, excepto su inicialización.
- Por convención, el nombre del parámetro del setter es **value**, pero se puede elegir un nombre diferente.

```
fun main() { * patriciasfo *  
    val rect = Rectangulo( ancho: 5, alto: 5)  
    rect.contador = 20  
    println("Contador = " + rect.contador)  
}
```

```
class Rectangulo(val ancho: Int, val alto: Int) { new *  
    var contador: Int = ancho + alto new *  
    set(value) {  
        if (value >= 0)  
            field = value  
            //contador = value // ERROR recursividad  
    }  
  
    val area: Int new *  
    get() = this.ancho * this.alto  
}
```

5. Lenguaje de programación Kotlin (53)

CLASE ANIDADA O NESTED CLASS

- Es una clase que está dentro de otra, favorece el encapsulamiento y **no** puede acceder a los miembros de la clase externa.

```
fun main() {  @ patriciasfo *
    val claseAnidada = Principal.ClaseAnidada()
    val suma = claseAnidada.suma( num1: 5, num2: 8)
    println("El resultado de la suma es $suma")
}

class Principal(){  new *
    private val uno = 1

    class ClaseAnidada() {  new *
        fun suma (num1: Int, num2: Int): Int {  new
            return num1 + num2 + uno
        }
    }
}
```


5. Lenguaje de programación Kotlin (54)

CLASE INTERNA

- Es una clase que está dentro de otra, favorece el encapsulamiento y **sí** puede acceder a los miembros de la clase externa.

```
fun main() {  patriciasfo *
    val claseInterna = Principal().ClaseInterna()
    val sumarDos = claseInterna.sumarDos( num: 10)
    println("El resultado de la suma es $sumarDos")
}

class Principal{  new *
    private val uno = 1
    private fun obtenerUno(): Int{  new *
        return uno
    }
    inner class ClaseInterna{  new *
        fun sumarDos(num: Int): Int{  new *
            return num + uno + obtenerUno()
        }
    }
}
```

5. Lenguaje de programación Kotlin (55)

HERENCIA DE CLASES

- Una clase se deriva de otra de manera que **extiende** su funcionalidad o la **especializa**.
- Todas las clases extienden de la clase o tiene una superclase común: **Any**. Por lo tanto, ya heredan comportamientos:

```
class Person {  
    override fun toString(): String{  
        return super.toString()  
    }  
}
```

- Para que otras clases puedan extender de *Person*, ésta tiene que definirse con el modificador **open**. Utilizar **open** también en los métodos.
- Si la clase padre tiene propiedades, las clases hijas tiene que tenerlas también.
- En las clases hijas se puede especializar la funcionalidad de la clase padre y extender su comportamiento añadiendo nuevas propiedades y funciones.
- **Clase abstracta**: palabra reservada **abstract** tanto en la clase como en sus métodos.
- Una clase no puede heredar o extender de 2 o más clases.

5. Lenguaje de programación Kotlin (56)

```
fun main() {
    val programador = Programmer("Patricia", 25, "Kotlin")
    programador.trabajar()
    programador.imprimirLenguaje()
    val disenador = Designer("Juan", 33)
    disenador.trabajar()
}

open class Person(nombre: String, edad: Int) {
    open fun trabajar(){
        println("Esta persona está trabajando")
    }
}

class Programmer(nombre: String, edad: Int, val lenguaje: String): Person(nombre, edad){
    override fun trabajar(){
        println("Esta persona está programando")
    }
    fun imprimirLenguaje(){
        println("Este programador utiliza el lenguaje $lenguaje")
    }
}

class Designer(nombre: String, edad: Int): Person(nombre, edad){
    override fun trabajar(){
        println("Esta persona está diseñando")
        super.trabajar()
    }
}
```

5. Lenguaje de programación Kotlin (57)

```
fun main() {
    val persona = Person("Sara", 40)
    persona.irATrabajar()
    val programador = Programmer("Patricia", 25, "Kotlin")
    programador.trabajar()
    programador.imprimirLenguaje()
    programador.irATrabajar()
    val disenador = Designer("Juan", 33)
    disenador.trabajar()
}

abstract class Trabajo(){
    abstract fun irATrabajar()
}

open class Person(nombre: String, edad: Int): Trabajo() {
    open fun trabajar(){
        println("Esta persona está trabajando")
    }

    override fun irATrabajar(){
        println("Esta persona va al trabajo")
    }
}

open class Vehiculo(){
    open fun conducir(){

    }
}

class Programmer(nombre: String, edad: Int, val lenguaje: String): Person(nombre, edad), Vehiculo(){
    override fun trabajar(){
        println("Esta persona está programando")
    }
    fun imprimirLenguaje(){
        println("Este programador utiliza el lenguaje $lenguaje")
    }
    override fun irATrabajar(){
        println("Esta persona va a Google")
        //super.irATrabajar()
    }
}

class Designer(nombre: String, edad: Int): Person(nombre, edad){
    override fun trabajar(){
        println("Esta persona está diseñando")
        super.trabajar()
    }
}
```

5. Lenguaje de programación Kotlin (58)

INTERFACES

- Pueden contener **declaraciones abstractas** o implementaciones de funciones o propiedades.
- La diferencia con las clases abstractas es que las **interfaces nunca pueden almacenar estado**.

```
fun main() {  
    val jugador = Person("Patricia", 35)  
    jugador.jugar()  
    jugador.stream()  
}  
  
interface Juego {  
    val juego: String  
    fun jugar()  
    fun stream(){  
        println("Estoy haciendo stream de mi juego $juego")  
    }  
}  
  
open class Person(nombre: String, edad: Int): Trabajo(), Juego {  
    open fun trabajar(){  
        println("Esta persona está trabajando")  
    }  
    override fun irATrabajar(){  
        println("Esta persona va al trabajo")  
    }  
    // Interfaz Juego  
    override val juego: String = "Among Us"  
    /*override val juego: String  
    get() = "Among Us"*/  
    override fun jugar(){  
        println("Esta persona está jugando $juego")  
    }  
    /*override fun stream(){  
        super.stream()  
    }*/  
}  
  
abstract class Trabajo(){  
    abstract fun irATrabajar()  
}
```

5. Lenguaje de programación Kotlin (59)

ENUM CLASSES

- Las enumeraciones son una lista de constantes con nombre. En Kotlin, una enumeración tiene su propio tipo especializado, lo que indica que algo tiene una cantidad de valores posibles.
- Las enumeraciones son clases, no son meras colecciones de constantes, tienen propiedades, métodos, etc.
- Cada una de las constantes de enumeración actúa como una instancia separada de la clase y está separada por comas.
- Las enumeraciones aumentan la legibilidad del código al asignar nombres predefinidos a las constantes.
- No se puede crear una instancia de una clase enum utilizando constructores.

Definición de un tipo enumerado o enum.

```
enum class NombreEnumerado
```

5. Lenguaje de programación Kotlin (60)

Definición de un tipo enumerado o enum.

- Es un objeto en donde define cada una de sus constantes separadas por comas.

```
enum class Planetas {  
    MERCURIO, VENUS, TIERRA, MARTE, JUPITER, SATURNO, URANO, NEPTUNO  
}
```

- Dado que las constantes de enumeración son instancias de una clase Enum, las constantes se pueden inicializar pasando valores específicos al constructor. Al ser un método constructor, podemos especificar tantos parámetros como queramos:


```
enum class Planetas(val orden: Int) { new *  
    MERCURIO(1), VENUS(2), TIERRA(3), MARTE(4), JUPITER(5), SATURNO(6), URANO(7), NEPTUNO(8)  
}
```

```
enum class Planetas(val n: Int, val tamaño: Float) { new *  
    MERCURIO(n: 1, tamaño: 1f), VENUS(n: 2, tamaño: 2f), TIERRA(n: 3, tamaño: 2f), MARTE(n: 4, tamaño: 1.5f),  
    JUPITER(n: 5, tamaño: 2600f), SATURNO(n: 6, tamaño: 1500f), URANO(n: 7, tamaño: 20f), NEPTUNO(n: 8, tamaño: 20f)  
}
```

5. Lenguaje de programación Kotlin (61)

En una clase de enumerados también se pueden añadir funciones.

Usando los enum

```
fun main() {  patriciasfo *  
    val planeta = Planetas.MARTE  
  
    val colorPlaneta = when(planeta) {  
        Planetas.TIERRA -> "tierra"  
        Planetas.MARTE -> "rojo"  
        else -> "Otros colores"  
    }  
  
    println("nombre: " + Planetas.MARTE.name)  
    println("ordinal: " + Planetas.MARTE.ordinal)  
    println("orden: " + Planetas.MARTE.orden)  
    println("color: " + colorPlaneta)  
}
```

Propiedades por defecto

5. Lenguaje de programación Kotlin (62)

Usando los enum

- Obtener el enum por su nombre -> `.valueOf()`
- Para recorrer los enumerados -> `.values()`. A partir de la versión 1.9.0 usar la propiedad `.entries`.

```
EnumClass.valueOf(value: String): EnumClass  
EnumClass.entries: EnumEntries<EnumClass> // specialized List<EnumClass>
```

```
enum class RGB { RED, GREEN, BLUE }  
  
fun main() {  
    for (color in RGB.entries) println(color.toString())  
    println("The first color is: ${RGB.valueOf("RED")}")  
}
```

5. Lenguaje de programación Kotlin (63)

Enum como clases anónimas

- Podemos definir los enumerados mediante funciones y clases anónimas. Son ideales para mostrar o realizar algún cálculo:

```
enum class Interes {  
    SILVER {  
        override fun intereses() = 0.25f  
    },  
    GOLD {  
        override fun intereses() = 0.5f  
    };  
  
    abstract fun intereses(): Float  
}  
  
fun main(args: Array<String>) {  
    val cashbackPercent = Interes.SILVER.intereses()  
    print(cashbackPercent) // print 0.25  
}
```

5. Lenguaje de programación Kotlin (64)

Enum implementando interfaces

- Para definir la interfaz se utiliza la palabra reservada **interface**. Las interfaces van a definir métodos pero no van a tener su implementación.

```
interface ICardLimit {  
    fun getCreditLimit(): Int  
}
```

- Cómo un enumerado implementa una interfaz:

```
enum class CardType : ICardLimit {  
    SILVER {  
        override fun getCreditLimit() = 100000  
    },  
    GOLD {  
        override fun getCreditLimit() = 200000  
    },  
    PLATINUM {  
        override fun getCreditLimit() = 300000  
    }  
}
```

```
val creditLimit = CardType.PLATINUM.getCreditLimit()
```

5. Lenguaje de programación Kotlin (65)

MODIFICADORES DE VISIBILIDAD

- Las **clases, objetos, interfaces, constructores, funciones, propiedades** y accesos **get/set** pueden tener modificadores de visibilidad.
- Modificadores:
 - **private**: declaraciones visibles y accesibles sólo desde la clase en el que se encuentra la declaración. Si la clase es **private** es visible si está en el mismo fichero.
 - **protected**: comportamiento igual que **private**, exceptuando que el acceso no estará disponible para declaraciones de nivel superior y sí para nivel inferior (subclases).
 - **internal**: declaraciones visibles y accesibles desde cualquier parte del módulo (conjunto de ficheros kotlin compilados de forma conjunta: librería). Queremos que sólo sea visible en nuestro proyecto y no en aquellos que utilicen el nuestro.
 - **public**: modificador por defecto. Declaraciones visibles y accesibles desde cualquier parte de nuestro código.

5. Lenguaje de programación Kotlin (66)

PUBLIC

```
fun main() { ⚡ patriciasfo *
    modificadores()
}
private fun modificadores(){ new *
    val visibilidad = Visibilidad()
    visibilidad.nombre = "Patricia"
    visibilidad.diMiNombre()
}
class Visibilidad{ new *
    var nombre: String? = null
    fun diMiNombre(){ new *
        nombre?.let {
            println("Mi nombre es $it")
        } ?: run {
            println("No tengo nombre")
        }
    }
}
```

5. Lenguaje de programación Kotlin (67)

PRIVATE

```
private fun modificadores(){ new *
    val visibilidad = Visibilidad()
    visibilidad.nombre = "Patricia"
    visibilidad.diMiNombre()
}

class Visibilidad{ new *
    private var nombre: String? = null
    private fun diMiNombre(){ new *
        nombre?.let {
            println("Mi nombre es $it")
        } ?: run {
            println("No tengo nombre")
        }
    }
}
```

5. Lenguaje de programación Kotlin (68)

PROTECTED

```
private fun modificadores() { new *  
    val visibilidad2 = Visibilidad2()  
    visibilidad2.diMiNombre2()  
}  
  
open class Visibilidad2 { new *  
    protected fun diMiNombre2() { new *  
        val visibilidad = Visibilidad()  
        visibilidad.nombre = "Patricia"  
    }  
}  
  
class Visibilidad3: Visibilidad2() { new  
    fun diMiNombre3() { new *  
        diMiNombre2()  
    }  
}
```

5. Lenguaje de programación Kotlin (69)

INTERNAL

Es el que menos se utiliza

```
class Visibilidad3: Visibilidad2() {  
    internal val edad: Int? = null  
    fun diMiNombre3() { new *  
        | diMiNombre2()  
    }  
}
```


5. Lenguaje de programación Kotlin (70)

DATA CLASSES

- Son clases particularmente útiles para **almacenar datos**.
- Tienen la misma funcionalidad que las clases pero generan métodos automáticamente: mostrar una instancia por pantalla, comparar instancias, copiar instancias, etc.


Function	Description
<code>toString()</code>	Prints a readable string of the class instance and its properties.
<code>equals()</code> or <code>==</code>	Compares instances of a class.
<code>copy()</code>	Creates a class instance by copying another, potentially with some different properties.

- Para declarar este tipo de clases se usa la palabra reservada **data**:
data class Usuario(val nombre: String, val id: Int)

5. Lenguaje de programación Kotlin (71)

```
data class Trabajador(val nombre: String = "", val edad: Int = 0, val trabajo: String = ""){  
    var anteriorTrabajo: String = ""  
}
```

```
private fun dataClases() { new *  
    val trabajador = Trabajador(nombre: "Patricia", edad: 35, trabajo: "Programador")  
    trabajador.anteriorTrabajo = "Musico"  
    val sara = Trabajador()  
}
```



Por tener valores por defecto en el constructor de la clase

5. Lenguaje de programación Kotlin (72)

Imprimir como String

- Muestra por pantalla la clase y sus propiedades. Se puede utilizar *print* o *println* directamente sobre la instancia o acceder al método *.toString()*.

```
fun main() {  👤 patriciasfo *
    val user = Usuario( nombre: "Alex", id: 1)
    println("Impresión directa: " + user)
    println("Utilizando la funcion toString(): " + user.toString())
}

data class Usuario(val nombre: String, val id: Int) new *
```

Comparando instancias

- Usar el operador `==` ó `equals` afecta sólo a las propiedades del constructor primario y no las propiedades del cuerpo.

```
fun main() {  👤 patriciasfo *
    val user = Usuario( nombre: "Alex", id: 1)
    val secondUser = Usuario( nombre: "Alex", id: 1)
    val thirdUser = Usuario( nombre: "Max", id: 2)

    println("user == secondUser: ${user == secondUser}")
    println("user == thirdUser: ${user == thirdUser}")
}
```

5. Lenguaje de programación Kotlin (73)

```
data class Persona(val name: String) {  
    var edad: Int = 0  
}
```

```
fun main() {  @ patriciasfo *  
    val persona1 = Persona(name: "John")  
    val persona2 = Persona(name: "John")  
    persona1.edad = 10  
    persona2.edad = 20  
    println("persona1 == persona2: ${persona1 == persona2}")  
    println("persona1 equals persona2: ${persona1.equals(persona2)}")  
    println("persona1 with age ${persona1.edad}: ${persona1}")  
    println("persona2 with age ${persona2.edad}: ${persona2}")  
}
```

5. Lenguaje de programación Kotlin (74)

Copiando instancias

- Para realizar una copia exacta se utiliza la función `copy()`.
- Si se desea modificar alguna de las propiedades al realizar la copia, se llama a la función `copy()` y se reemplazan los valores de las propiedades como parámetros de la función.

```
fun main() {  @ patriciasfo *
    val user = Usuario( nombre: "Alex", id: 1)
    val secondUser = Usuario( nombre: "Alex", id: 1)
    val thirdUser = Usuario( nombre: "Max", id: 2)

    println(user.copy())
    println(user.copy( nombre: "Max"))
    println(user.copy(id = 3))
}
```

```
fun main() {  @ patriciasfo *
    val jack = Usuario(nombre = "Jack", id = 1)
    val olderJack = jack.copy(id = 2)
    println(jack)
    println(olderJack)
    println(jack.copy( nombre: "Max", id: 3))
}
```

5. Lenguaje de programación Kotlin (75)

Recuperación de parámetros

- A veces se necesita desestructurar un objeto en variables, es decir, almacenar todos los parámetros de un objeto de golpe.


```
fun main() { * patriciasfo *  
    val persona = Persona(name: "Mike", edad: 35)  
    val (name, age) = persona  
    println("Nombre: " + name)  
    println("Edad: " + age)  
}  
  
data class Persona(val name: String, val edad: Int)
```

- Si no se necesita alguna de las variables, se puede sustituir por `_`.
val (_, age) = persona

5. Lenguaje de programación Kotlin (76)

Recuperación de parámetros

- Por defecto, en cada data class, kotlin genera un **componentN()** para cada uno de los parámetros y en el orden en el que están declarados.

```
fun main() {  patriciasfo *  
    val persona = Persona( name: "Mike", edad: 35)  
    val name = persona.component1()  
    val age = persona.component2()  
    println("Nombre: " + name)  
    println("Edad: " + age)  
}  
  
data class Persona(val name: String, val edad: Int)
```