http://cursoKotlin.com

Tutoriales en PDF

Bienvenidos a este primer capítulo de un curso, en el que sin duda, aprenderemos ambos sobre Kotlin. En esta serie de entradas, mi intención no es copiar y pegar, me gustaría debatir y aprender recíprocamente.

En este capítulo veremos qué es Kotlin y porqué estamos en el momento perfecto para empezar.

¿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.

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.

¿Por qué usar Kotlin?

Me imagino que será una de las preguntas que más te estarás haciendo a medida que lees este artículo, y la verdad que no hay una respuesta mágica que te diga que esto será un éxito o no, así que te voy a hablar por qué desde mi punto de vista vale la pena.

- **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.
- **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.
- **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*.

- **Fácil de usar**: Al estar inspirado en lenguajes ya existentes como Java, C# o Scala, la curva de aprendizaje nos será bastante sencilla.
- **Es el momento:** Hace escasos días en la Google I/O 2017, Kotlin se ha convertido oficialmente en un lenguaje de Android, por lo que ahora mismo es el boom. Si eres rápido, si trabajas y te implicas puedes dedicarte profesionalmente a Kotlin, y más ahora que empresas muy importantes están empezando a usar el lenguaje (Pinterest, Gradle, Evernote, Uber, etc).

Conclusión

Si hay buenos momentos para subirse a la ola, este es uno estupendo. Kotlin ha entrado con muchísima fuerza, no solo para plantar cara a Java, sino para ganarle. De aquí a un año todo apunta a que será un perfil muy demandado por reclutadores en el Linkedin, y debido a la novedad del tema actualmente hay muy pocos programadores que controlen Kotlin. Si buscas mejorar, actualizarte y no quedarte desfasado y poder llegar a ser un referente del sector únete conmigo a este curso que iré desarrollando y conviértete en un profesional. Ahora o nunca.

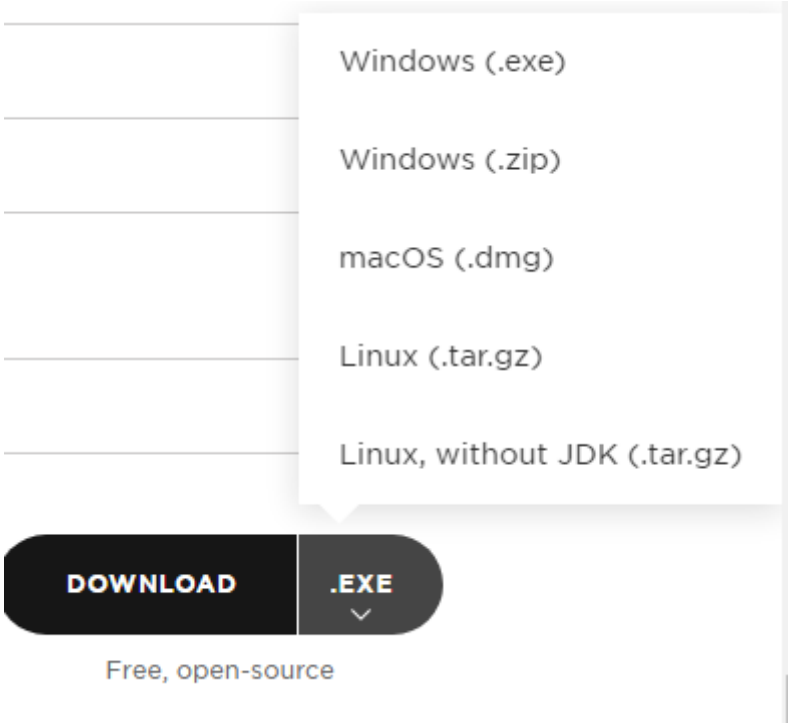
Una vez nos hemos decidido a empezar con Kotlin, debemos bajarnos un IDE con el que trabajar. Obviamente tiraremos por IntelliJ, que es de los creadores de este lenguaje.

Instalando Intellij Idea

Lo primero que debemos hacer es ir a la página oficial y bajarnos la última versión. Podéis ir directamente desde aquí. Ahora le damos a descargar y nos llevará a la parte inferior de la web donde tendremos 2 versiones para descargar, **Ultimate** y **Community**. si miramos la tabla veremos la diferencia entre ambas, como es lógico una versión es de pago (la más completa) y otra gratuita. Con la versión **Comminity** tendremos más que necesario.

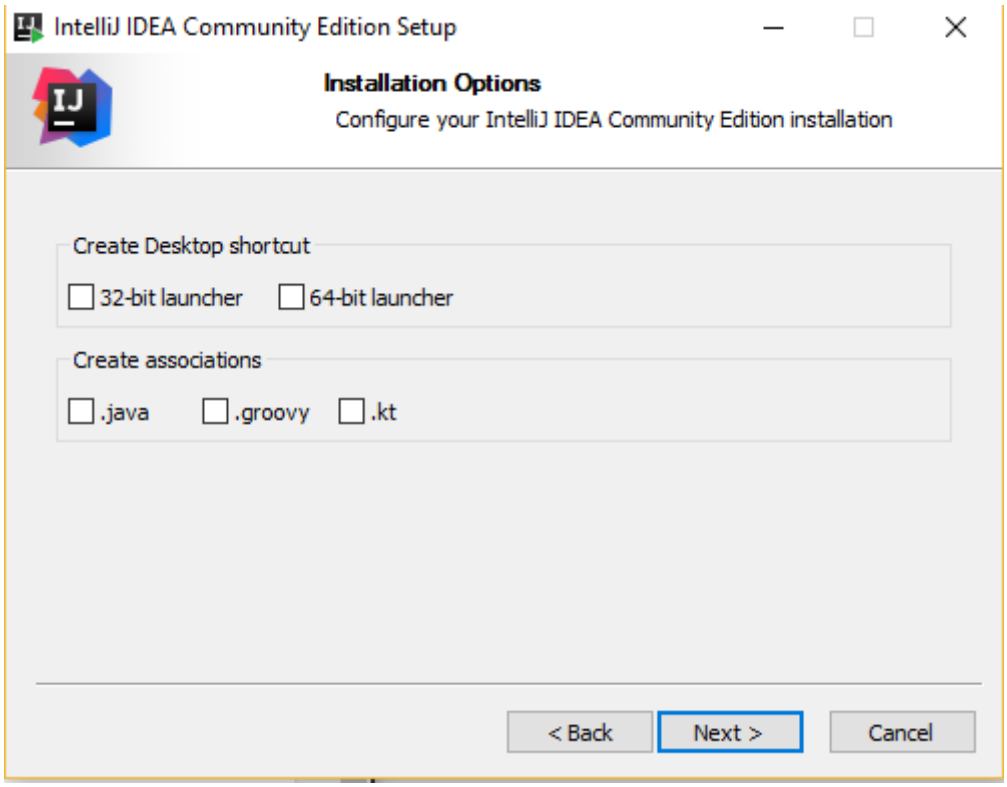
	Ultimate For web and enterprise development	**Community** For JVM and Android development
License	Commercial	Open-source, Apache 2.0
Java, Kotlin, Groovy, Scala	✓	✓
Android	✓	✓
Maven, Gradle, SBT	✓	✓
Git, SVN, Mercurial, CVS	✓	✓
Detecting Duplicates	✓	
Perforce, ClearCase, TFS	✓	
JavaScript, TypeScript	✓	
Java EE, Spring, GWT, Vaadin, Play, Grails, Other Frameworks	✓	
Database Tools, SQL	✓	

Descargamos necesaria para nuestro sistema operativo, podemos seleccionarla clickando en el botón que tenemos a la derecha de *download.*



Una vez descargado lo instalamos. En esta caso lo haré con windows, pero si hiciese falta puedo actualizar la entrada con Linux o Mac OS.

Una vez ejecutado le damos a siguiente y seleccionamos la ruta de la instalación, en la siguiente vista veremos una pantalla así.



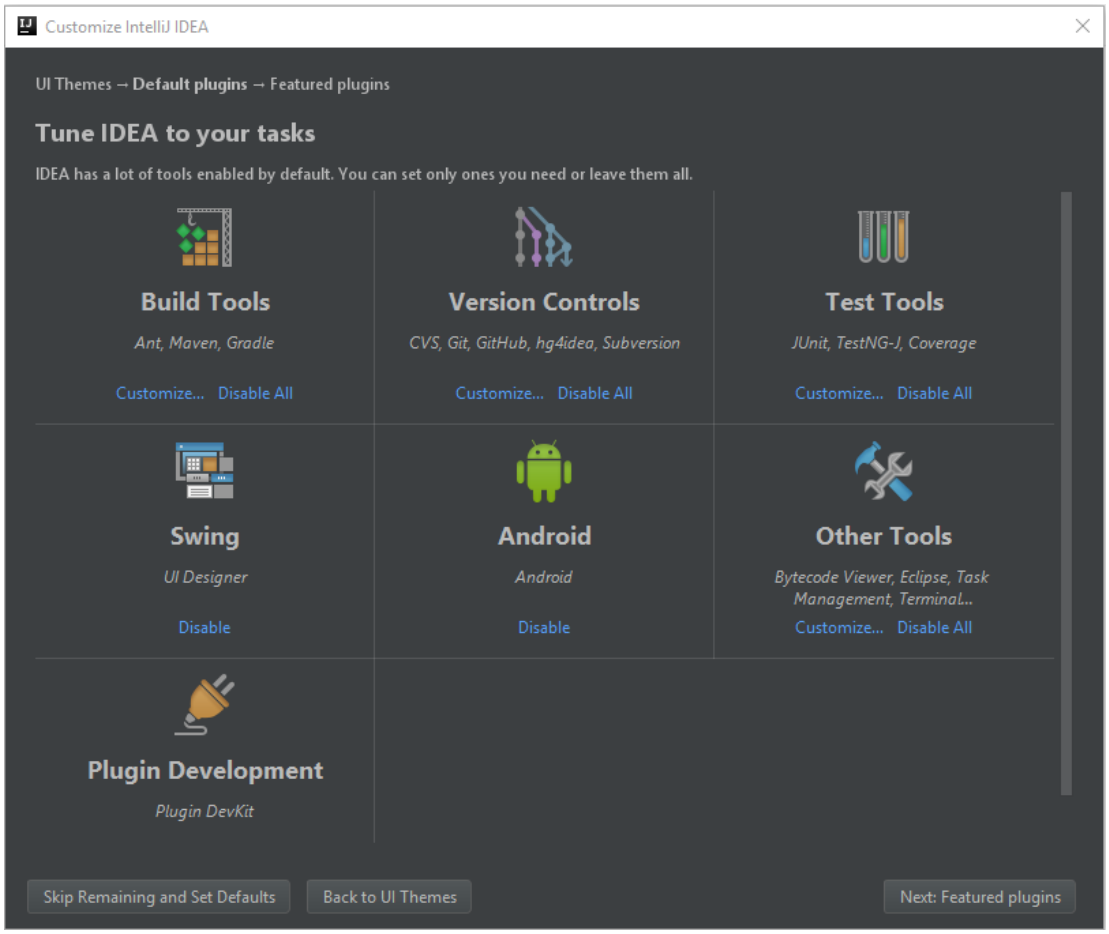
Como este curso está enfocado a Kotlin, en *Create associations* seleccionaremos los **.kt** que es el formato de archivo para los ficheros de kotlin, pero si también queréis trabajar con **.java** o **.groovy** los podéis seleccionar, aunque para este curso no harán falta.

Pulsamos en siguiente y luego en instalar. Una vez hecho lo abriremos por primera vez y nos preguntará si queremos importar algún archivo de configuración (por ejemplo si estamos instalando esto en un segundo equipo nuestro y queremos mantener la configuración como estaba en el otro ordenador). Pero esta vez le damos a no.

Configurando el entorno

Aceptamos los términos y condiciones y nos saldrá una vista para seleccionar el theme. El theme es básicamente la parte gráfica de nuestro IDE, por defecto es blanca, pero no la recomiendo si trabajáis muchas horas porque cansa más la vista (al menos a mi) así que yo seleccionaré la versión **Darcula**.

La siguiente vista nos mostrará los plugins por defecto que podremos activar, desactivar e incluso modificar.



Para este curso no necesitaremos varias cosas de aquí pero si hay varias que recomiendo (aunque si las desactivamos ahora las podemos activar más tarde y viceversa así que no hay problema). **Test Tools, Build Tools, Version Controls y Other Tools** las dejaremos activadas. Android sería una muy buena opción (me gustaría enfocar el curso más adelante a ello una vez controlemos lo básico) pero para eso tiraremos de Android Studio en su debido momento.

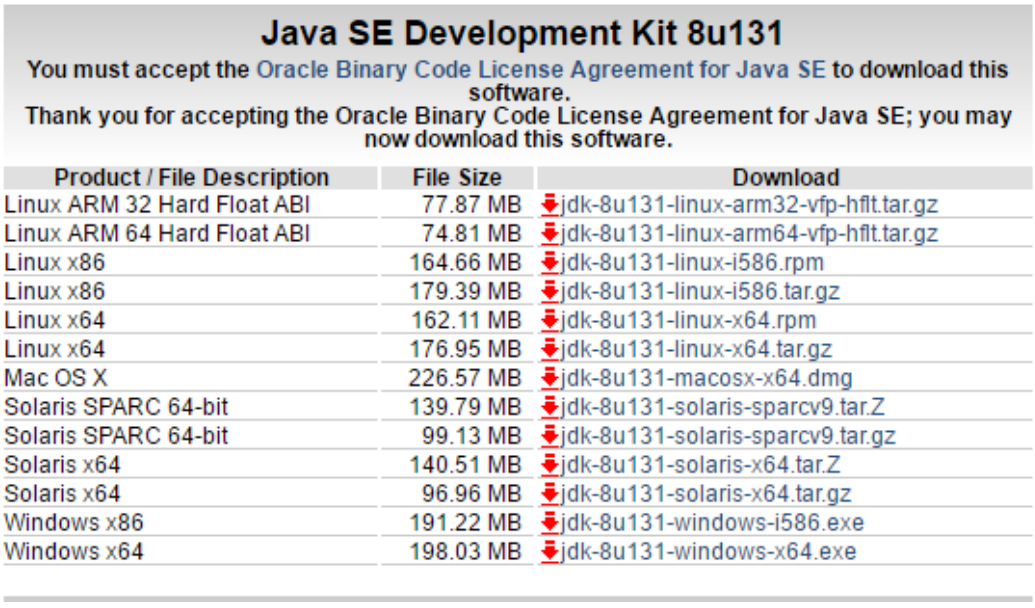
Así que pulsamos en Next y nos iremos a las *Featured plugins*, pero en esta pantalla no instalaremos nada, así que pulsamos directamente en **Start using IntelliJ IDEA**.

Descargando el SDK

El siguiente paso será bajarnos el SDK, que es básicamente un conjunto de herramientas necesarias para poder trabajar con el IDE.

Vamos a la página oficial y hacemos click en **Java Platform (JDK) 8u131** o la versión más actual en el momento.

Una vez le hemos dado nos llevará a un listado de versiones disponibles dependiendo del sistema operativo. Así que aceptamos las condiciones y nos bajamos la versión que necesite nuestro equipo.



Java SE Development Kit 8u131

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.
Thank you for accepting the Oracle Binary Code License Agreement for Java SE; you may now download this software.

Product / File Description	File Size	Download
Linux ARM 32 Hard Float ABI	77.87 MB	jdk-8u131-linux-arm32-vfp-hflt.tar.gz
Linux ARM 64 Hard Float ABI	74.81 MB	jdk-8u131-linux-arm64-vfp-hflt.tar.gz
Linux x86	164.66 MB	jdk-8u131-linux-i586.rpm
Linux x86	179.39 MB	jdk-8u131-linux-i586.tar.gz
Linux x64	162.11 MB	jdk-8u131-linux-x64.rpm
Linux x64	176.95 MB	jdk-8u131-linux-x64.tar.gz
Mac OS X	226.57 MB	jdk-8u131-macosx-x64.dmg
Solaris SPARC 64-bit	139.79 MB	jdk-8u131-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	99.13 MB	jdk-8u131-solaris-sparcv9.tar.gz
Solaris x64	140.51 MB	jdk-8u131-solaris-x64.tar.Z
Solaris x64	96.96 MB	jdk-8u131-solaris-x64.tar.gz
Windows x86	191.22 MB	jdk-8u131-windows-i586.exe
Windows x64	198.03 MB	jdk-8u131-windows-x64.exe

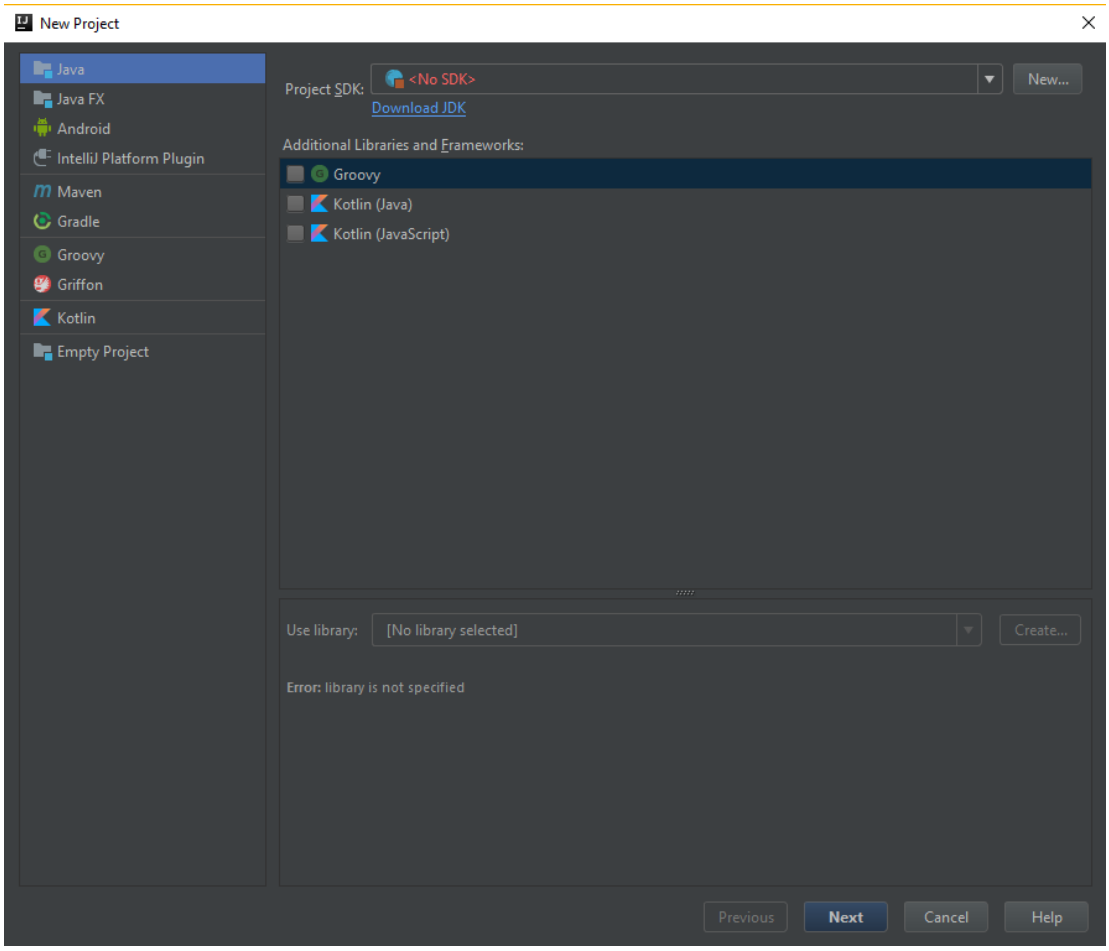
En mi caso descargaré **jdk-8u131-windows-x64.exe**

Una vez descargado lo ejecutamos y lo instalamos, no tiene ninguna complejidad. Recomiendo dejar la ruta de instalación por defecto para que nuestro IDE lo encuentre, de todos modos si no lo encuentra solo habría que seleccionarlo dentro del IntelliJ.

Con esto ya tenemos configurado nuestro entorno para programar en Kotlin.

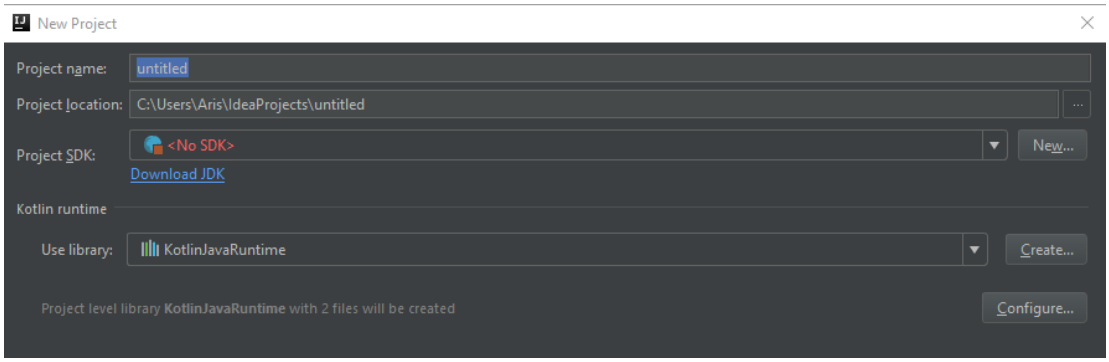
Una vez el entorno ha sido configurado, es el momento de mostraros como crear un nuevo proyecto en Kotlin y haremos el famoso “Hello World” que no es más que escribir nuestro primer programa que mostrará por pantalla dicha frase.

Lo primero que haremos será abrir IntelliJ y le daremos a *Create New Project* y nos mostrará una ventana así



Seleccionamos Kotlin (en el menú lateral) y una vez dentro seleccionamos **Kotlin (JVM)** y le damos a siguiente.

Nos aparecerá una ventana así que posiblemente tenga el mismo error.

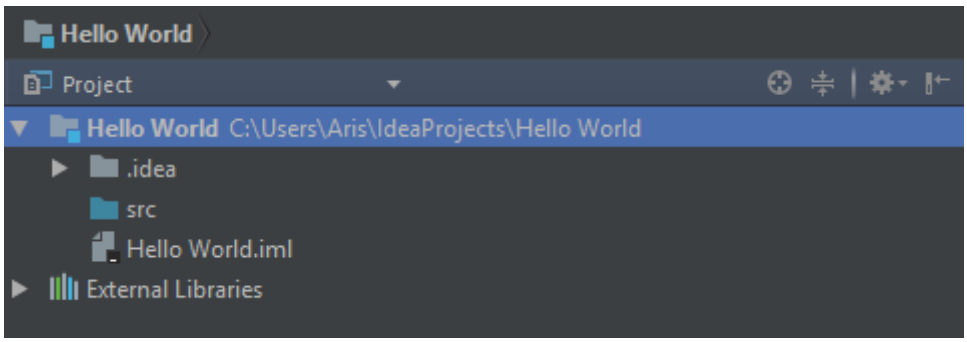


Eso significa que IntelliJ no está encontrando nuestro SDK (el que instalamos en el capítulo anterior) así que le daremos a *New* y buscaremos la ruta donde lo instalamos. Por defecto sería *C:\Program Files\Java* y el directorio de nuestro SDK.

Le ponemos un nombre a nuestro proyecto, en este caso Hello World y nos quedaría algo así.

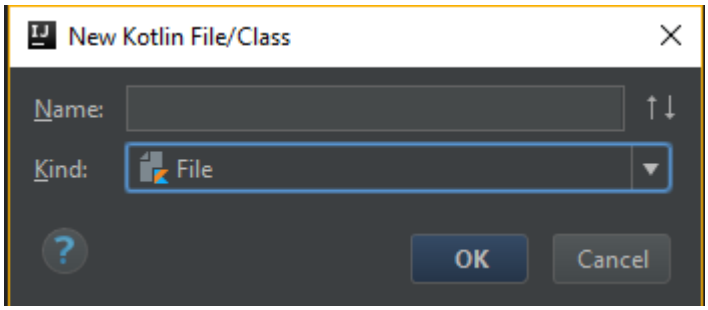
Pulsamos en **Finish** y ya tendremos nuestro proyecto creado.

En el menú lateral izquierdo tendremos por defecto la vista "Project" que será con la que trabajaremos. Si hacéis click ahí podréis ver diferentes organizaciones de los archivos, pero solo será visualmente, puesto que la estructura real del proyecto no cambiará por mucho que lo cambiemos ahí. Tendremos los siguientes archivos y directorios.



No hay mucho que contar aquí, tenemos la carpeta **.idea** y el archivo **.iml** que son ficheros de configuración del IDE. Luego tenemos la carpeta **src** que será donde crearemos los ficheros y directorios para trabajar.

Así que vamos a la carpeta src, botón secundario *New>Kotlin File/Class* y nos saldrá una ventana similar a esta.



Lo llamaremos HelloWorld (todo junto) y en kind lo dejaremos en *File* por ahora. Ya tenemos nuestro primer fichero con el que trabajar. Lo primero será crear un método *main* que de la posibilidad de ejecutar dicho archivo. En capítulos siguientes veremos que son las funciones y clases pero ahora no hace falta explicarlo.

Para generar dicho método, IntelliJ tiene unos template que nos hará la vida mucho más sencillo así que basta con escribir main y nos saldrá la opción de autocompletar. Generándonos una función igual a esta.

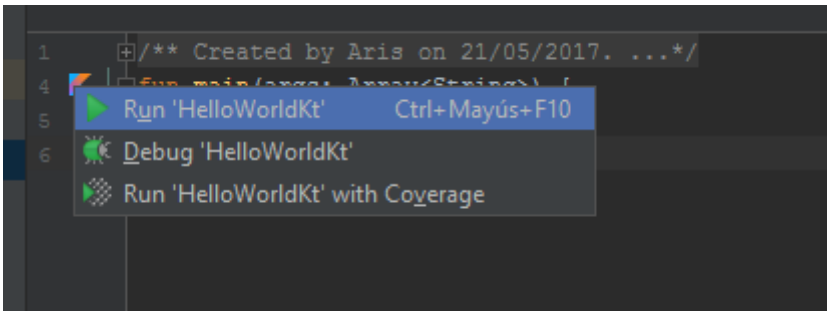
```
1 fun main(args: Array<String>) {
2
3 }
```

Esto nos permitirá poder ejecutar el archivo. Ahora dentro del método añadiremos la línea de código para pintar en pantalla nuestro Hello World. La función completa quedará así.

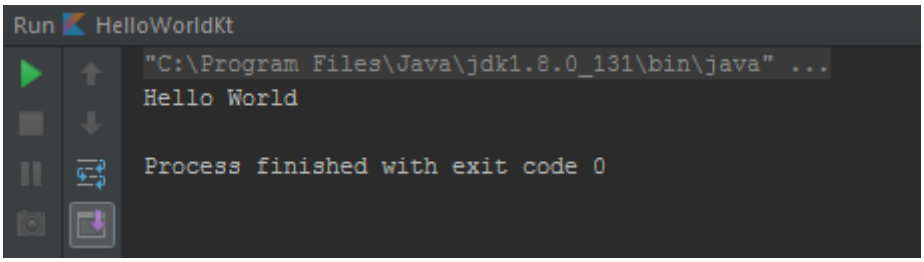
```
1 fun main(args: Array<String>) {
2     println("Hello World")
3 }
```

Como veis método println hace que todo lo que esté dentro del paréntesis (siempre entre comillas si es texto) lo muestre por pantalla. También hay que destacar que Kotlin es sensible a mayúsculas y minúsculas. Si lo queréis probar escribid Println (con la P mayúscula) y veréis que se os pone en rojo.

Una vez hecho esto hacemos click en la K que está a la izquierda de la función y pulsamos en run 'HelloWorldKT'



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.



Y con esto ya tenemos hecha nuestra primera aplicación. Muy sencillita pero ya tenemos conocimientos para generar nuestras clases, ficheros, interfaces, etc y ejecutarlas.

Como este curso está enfocado para todo tipos de usuarios, lo primero que haré será explicar que es una variable. A continuación veremos los tipos y como implementarlos en java.

¿Qué es una variable?

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

```
1 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.

Como podéis ver, 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.

```
1 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.

Tipos de variables

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.

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

- 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**.

```
1 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.

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

- Double

Terminamos con 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.

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

VARIABLES Alfanuméricas

Aunque los números sean muy útiles, hay veces que necesitaremos guardar cadenas de texto, o una mezcla de caracteres. Para ello tenemos las variables alfanuméricas.

- 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.

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

var letraFavorita: Char = 'q' var numeroFavorito: Char = '@'

Como podéis ver 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.

```
1 var numeroFavorito: String = "Mi número favorito es el 3"
2 var test: String = "Test. 12345!·$%&/"
```

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.

```
1  var estoyTriste: Boolean = false
2  var estoyFeliz: Boolean = true
```

Conclusión

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.

En el siguiente capítulo empezaremos a trabajar con las variables.

En el capítulo 4 vimos los principales tipos de variables, ahora empezaremos a trabajar con ellos para conseguir un poco de fluidez.

Operaciones aritméticas

En el ejemplo que veremos a continuación os mostraré los tipos de operaciones básicas que podemos realizar.

```
fun main(args: Array<String>) {
    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)
}
```

Como podéis ver son operaciones muy básicas en las que no hay ninguna complicación. Ahora imaginemos que tenemos un Float y un Int. Tampoco tendríamos para realizar cualquiera de las operaciones siguientes a no ser que queramos almacenar el resultado en una nueva variable.

```
fun main(args: Array<String>) {
    var a: Float = 10.5f
    var b: Int = 5

    print("Suma: ")
    var resultado = a + b

   print(resultado)
}
```

Esto funcionaría, pero solo porque no le hemos asignado un tipo a la variable resultado, así que automáticamente se le asignará Float. Pero ¿qué pasa si generamos la variable resultado siendo Int? Pues que no funcionará porque no podemos operar con variables de tipos diferentes. Para ello haremos uso del *toInt()* que nos permite convertir una variable a Integer.

```
fun main(args: Array<String>) {
    var a: Float = 10.5f
    var b: Int = 5
    var resultado: Int

    //Esto no funciona
    print("Suma: ")
    resultado =  a + b

    //Esto sí
    print("Suma: ")
    resultado =  a.toInt() + b

   print(resultado)
}
```

También podríamos haber tenido la variable resultado en Float y pasar b con *toFloat()*.

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

Concatenación

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

```
fun main(args: Array<String>) {
    val greeting = "Hola, me llamo"
    val name = "aris"

    println("$greeting $name")
}
```

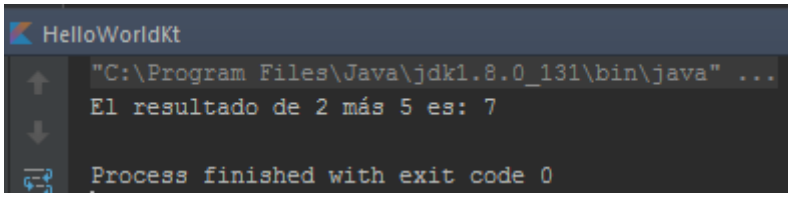
Solo debemos añadir las variables entre comillas dobles y anteponer a cada una de ellas el símbolo *$*. Daros cuenta que he añadido un espacio entre ambas variables, sino, aparecerá el resultado sin ese espacio.

> *NOTA: 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.*

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

```
1      val introduction = "El resultado de"
2      val plus = "más"
3      val firstNumber = 2
4      val secondNumber = 5
5
6      println("$introduction $firstNumber $plus $secondNumber es: ${firstNu
```

En este último ejemplo he intentado hacerlo un poco más complejo. Lo primero que hemos hecho es concatenar una frase, después hemos vuelto a añadir un $ y entre llaver ({}) hemos metido la operación. El resultado sería este.



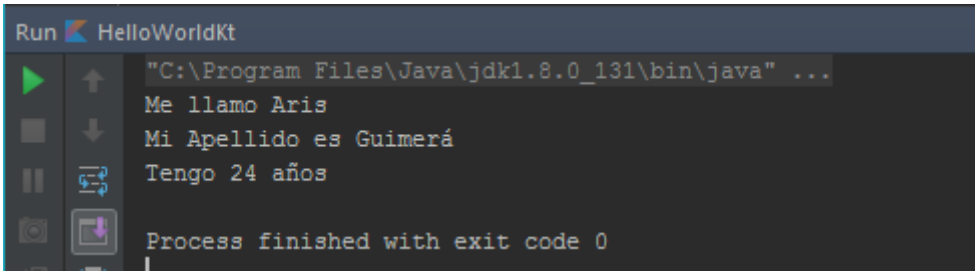
En este tema hablaremos de las funciones en kotlin, también llamadas métodos en otros lenguajes como por ejemplo java. Una función no es más que un conjunto de instrucciones que realizan una determinada tarea y la podemos invocar mediante su nombre.

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.

```
1  fun main(args: Array<String>) {
2      showMyName()
3      showMyLastName()
4      showMyAge()
5  }
6  fun showMyName(){
7      println("Me llamo Aris")
8  }
9  fun showMyLastName(){
10     println("Mi Apellido es Guimerá")
11 }
12 fun showMyAge(){
13     println("Tengo 24 años")
14 }
```

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 se ejecutar el código, los llamará en el orden que le pongamos. Dándonos un resultado así.



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 main(args: Array<String>) {
    showMyInformation("Aris", "Guimerá", 24)
}
fun showMyInformation(name: String, lastName: String, age: Int){
    println("Me llamo $name $lastName y tengo $age años.")
}
```

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 como 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 main(args: Array<String>) {
    var result = add(5, 10)
    println(resul)
}
fun add(firsNumber: Int, secondNumber: Int) : Int{
    return firsNumber + secondNumber
}
```

Como 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 + secondNu
```

Las instrucciones condicionales nos permiten realizar lógica en función del resultado de una variable o condición, en este primer apartado veremos las condiciones if-else.

La confición if

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

```
1 fun main(args: Array<String>) {
2     var result = add(5, 10)
3 
4     if(result > 10){
5         println("El resultado es mayor que 10")
6     }
7 }
8 fun add(firsNumber: Int, secondNumber: Int) : Int = firsNumber + secondNu
```

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

```
1     var name = "Aris"
2 
3     if(name == ("Aris")){
4         println("Se llama Aris")
5     }
```

If-Else

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

```
1   var name = "Aris"
2 
3     if(name == ("Aris")){
4         println("Se llama Aris")
5     }else{
6         println("No se llama Aris")
7     }
```

El funcionamiento está muy claro, si no pasa la condición estipulada, irá directa al else, así por ejemplo no tenemos que hacer 2 if, uno comprobando si el nombre es igual, y otro comprobando si es diferente.

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.

```
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.

Para poder 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")
    }
```

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 con el tutorial anterior (if-else en Kotlin) 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.

When

```
fun main(args: Array<String>) {
    getMonth(2)
}
fun getMonth(month : Int){
    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")
        }
    }
}
```

El ejemplo es muy sencillo. La función *getMonth* recibe un Int que se lo mandamos a el 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 como 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")
    }
}
```

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

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

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

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

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

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

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

```
fun result(month : Int){
    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"
    }
}
```

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.

Las arrays (o arreglos) 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 Strings o almacenarlas todas en una sola array.

```
1 fun main(args: Array<String>) {
2     val weekDays = arrayOf("Lunes", "Martes", "Miércoles", "Jueves", "Vie
3 }
```

Ahora la variable **weekDays** contiene todos los días de la semana. Ahora 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.

```
1  fun main(args: Array<String>) {
2   val weekDays = arrayOf("Lunes", "Martes", "Miércoles", "Jueves", "Viern
3
4       println(weekDays.get(0))
5       println(weekDays.get(1))
6       println(weekDays.get(2))
7       println(weekDays.get(3))
8       println(weekDays.get(4))
9       println(weekDays.get(5))
10      println(weekDays.get(6))
11 }
```

Esto nos pintará los días de la semana (posiciones de la 0 a la 6). Si pusiéramos en el get una posición que no tiene, por ejemplo la 7 nos daría una excepción al ejecutarse la aplicación *ArrayIndexOutOfBoundsException,* y es por ello por lo que al trabajar con arrays debemos tener bien claro el tamaño de la array.

Para evitar ese tipo de problemas podemos usar la función size, que nos devolverá el tamaño de dicha array.

```
fun main(args: Array<String>) {
    val weekDays = arrayOf("Lunes", "Martes", "Miércoles", "Jueves", "Vi

    println(weekDays.get(0))
    println(weekDays.get(1))
    println(weekDays.get(2))
    println(weekDays.get(3))
    println(weekDays.get(4))
    println(weekDays.get(5))
    println(weekDays.get(6))

    if(weekDays.size >= 8){
        println(weekDays.get(7))
    }else{
        println("no tiene más parámetros la array")
    }
}
```

Si ejecutamos dicho código no entrará en el if porque el tamaño de la array es de 7 (No es lo mismo tamaño que posición), por lo que nos pintará “no tiene más parámetros en la array”.

Como había dicho al principio del post, las array 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 esta array son de Strings, por lo que no podemos pasar un Int.

Recorriendo Arrays

Ahora que ya conocemos un poco más que es una array y como trabajar con ellas, vamos a ver como recorrerlas.

Aunque podríamos hacerlo como hicimos arriba, hay modos más rápidos y óptimos, el bucle *for()*. Este bucle nos permite entre otras, recorrer la array entera, 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(args: Array<String>) {
    val weekDays = arrayOf("Lunes", "Martes", "Miércoles", "Jueves", "Vie

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

Mucho menos código ¿verdad?. Vamos a ver que es lo que hemos hecho.

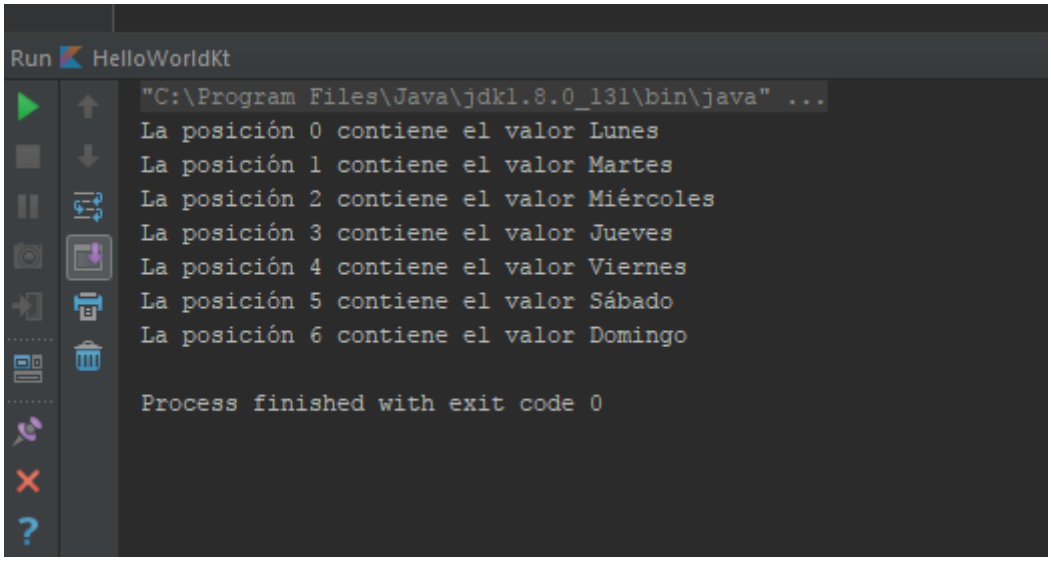
El for necesitará una variable, en este caso “posición” que irá teniendo el valor de cada una de las posiciones de la array. Su funcionamiento es muy sencillo, cuando pasa por el for por primera vez, tendrá valor 0, entonces comprueba el tamaño de *weekDays* y si es mayor, entra a

la función, hace lo que le pidamos (en este caso pintar en la posición de la variable) y vuelve al inicio, así hasta llegar a 6 que será la última posición de la array.

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

```
1 fun main(args: Array<String>) {
2     val weekDays = arrayOf("Lunes", "Martes", "Miércoles", "Jueves", "Vie
3
4     for ((posicion, valor) in weekDays.withIndex()) {
5         println("La posición $posicion contiene el valor $valor")
6     }
7 }
```

Y dicho for nos devolvería lo siguiente



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

```
1   for (weekDay in weekDays) {
2         println(weekDay)
3     }
```

Devolviéndonos la lista de días de la semana.

Listas en Kotlin

En el capítulo anterior hablamos sobre los arrays o arreglos y vimos todo el potencial que tenía, pero uno de los mayores inconvenientes era la limitación al definirlos, puesto que teníamos que saber de ante mano el tamaño de dicho array.

Tipos de listas

Las colecciones se pueden clasificar en dos grandes grupos, las mutables e inmutables. Es decir, las que se pueden editar (mutables) y las que son solo de lectura (inmutable).

Listas inmutables

La declaración de una lista inmutable sería así.

```
1 val readOnly: List<String> = listOf("Lunes", "Martes", "Miércoles", "Juev
```

Existen varias funciones útiles para trabajar con ellas.

```
1     val readOnly: List<String> = listOf("Lunes", "Martes", "Miércoles", "
2
3     readOnly.size //Muestra el tamaño de la lista
4     readOnly.get(3) //Devuelve el valor de la posición 3
5     readOnly.first() //Devuelve el primer valor
6     readOnly.last() //Devuelve el último valor
7     println(readOnly) //[Lunes, Martes, Miércoles, Jueves, Viernes, Sábad
```

Como pueden ver podemos trabajar completamente con ellas, a excepción de añadir más elementos.

También podemos filtrar usando el patrón iterator, que es un mecanismo para acceder a los elementos de la lista, por ejemplo *.first()* o *.last()*

```
1     val a = readOnly.filter { it == "Lunes" || it == "Juernes" }
```

El *.filter* nos permite filtrar en la lista a través de una o varias condiciones que pongamos. Para ello llamamos a it (iterator) y buscaremos en la lista, si contiene la palabra "**Lunes**" o "**Juernes**". En este caso solo pintará "**Lunes**".

Listas mutables

Ahora nos toca hablar de las más interesantes, las listas mutables, que poseen todo lo anterior, pero también nos da la posibilidad de ir rellenando la lista a medida que lo necesitemos, el único inconveniente es que más ineficiente con la memoria.

```
1 var mutableList: MutableList<String> = mutableListOf("Lunes", "Martes", "
2
3     println(readOnly) //[Lunes, Martes, Miércoles, Jueves, Viernes, Sábad
4
5     mutableList.add("domingo")
```

La definición es muy similar, a través de la función *mutableListOf.* Ahora si os fijáis he añadido de lunes a sábado que será lo que pintará, luego, usando *mutableList.add* he añadido el domingo.

Por defecto los valores se irán añadiendo en la última posición, pero podemos añadir el índice en el que queramos escribir nuestro valor.

```
1     mutableList.add(0, "Semana: ")
```

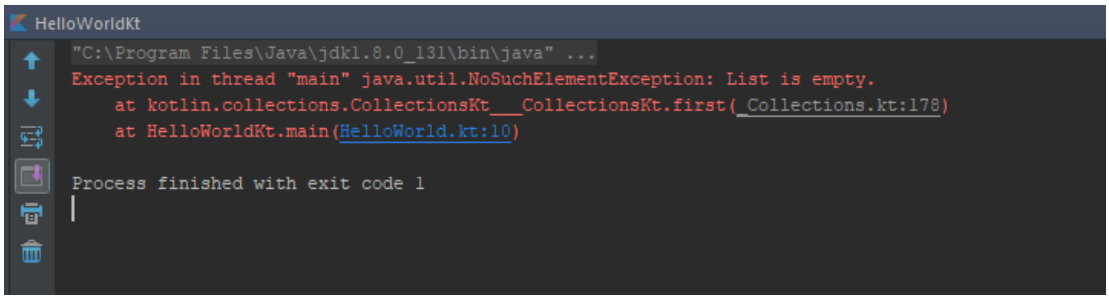
Con esto, si pintásemos la lista nos mostraría lo siguiente.

```
1 [Semana: , Lunes, Martes, Miércoles, Jueves, Viernes, Sábado, domingo]
```

Ahora tenemos que tener cuidado con una lista mutable por el simple hecho de que podría estar vacía o contener un *null*. Un null es un valor nulo en una de sus posiciones, que, si acedemos a él e intentamos operar, la aplicación se romperá, nos saltará un *crash*. Para ello, tenemos algunas funciones que nos permitirá trabajar como una lista inmutable, pero con seguridad.

```
1  var mutableList: MutableList<String> = mutableListOf()
2
3     mutableList.none() //Nos devuelve un true si está vacía la lista
4     mutableList.firstOrNull() //Nos devolverá el primer campo, y si no ha
5     mutableList.elementAtOrNull(2) //El elemento del índice 2, si no hay,
6     mutableList.lastOrNull() //Último valor de la lista o null
```

En este ejemplo he instanciado una lista sin valor alguno, por lo que está vacía. Con todos esos métodos recuperaremos un null (menos el primero que dará *true*) y la aplicación seguirá corriendo, si por le contrario hubiera puesto un .first() nos hubiese pasado lo siguiente.



Debemos de tener mucho cuidado a la hora de trabajar con datos que puedan mutar.

Recorriendo listas

Igual que hicimos en el capítulo de las arrays, aquí también tenemos formas (muy similares) para recorrer las listas.

Esta sería la forma más sencilla, y nos devolvería el contenido de cada uno de los valores de la lista.

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

Si necesitamos saber también la posición de cada uno de los valores podemos usar la función *.withIndex* que nos permite generar 2 variables, la primera será la posición y la segunda el contenido.

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

Para el último ejemplo, usaremos *.forEach,* una función que hará que por cada posición haga una cosa, por ejemplo pintar el valor como el for anterior. A diferencia de ellos, no tenemos una variable con el contenido *(véase índice e item)* sino que accedemos a él con el iterator, en este caso simplemente habría que poner *it*.

En este ejemplo imaginemos que queremos añadir a los días de la semana dos puntos ":" así que vamos a crear una nueva lista (mutable) e iremos rellenando la lista a través del *forEach.*

```
val mutableList: MutableList<String> = mutableListOf("Lunes", "Martes",
  val newListEmpty: MutableList<String> = mutableListOf()

  mutableList.forEach {
      newListEmpty.add(it+":")
  }

  print(newListEmpty)
```

Después de hacer 10 capítulos sobre introducción al Kotlin, creo que es el momento de empezar con Android, que era la idea principal cuando creé el blog. El curso de introducción se irá desarrollando también, pero a un ritmo más lento.

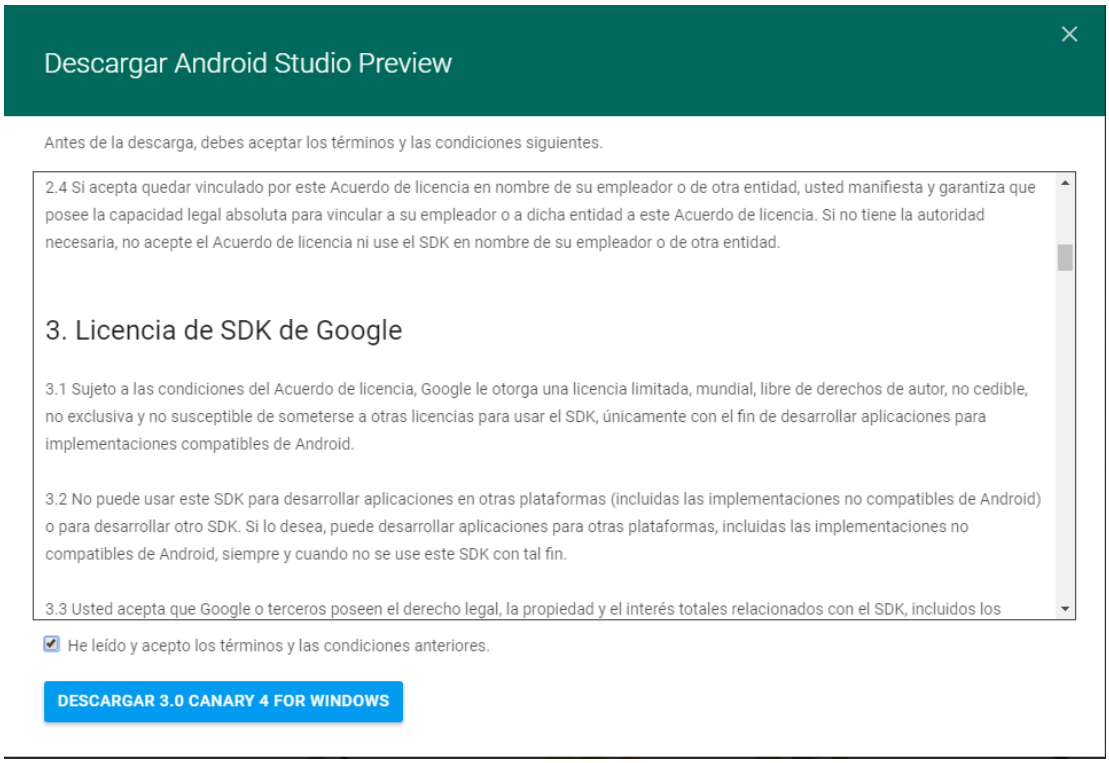
Configurando Android Studio

Ahora que empezamos con Android, vamos a cambiar de IDE. Será de la misma empresa (JetBrains) pero enfocado a Android.

Android Studio es un IDE muy completo, pero su versión estable todavía no soporta kotlin a no ser que hagamos una serie de pasos, así que he optado por empezar con la versión canary. **Android Studio Canary** es una versión especial que va recibiendo actualizaciones antes que la versión estable, por lo que puede tener algún fallo, pero accedemos a las novedades antes. Se pueden tener las 2 versiones a la vez, la canary y la estable.

Descargando e instalando la versión canary

Para acceder a esta versión solo tenemos que ir a este link y bajarnos la última versión.



Aceptamos los términos y empezará la descarga.

En Windows

Una vez haya terminado debemos descomprimir el archivo y renombrar la carpeta a un nombre que nos sea fácil de recordar, por ejemplo “android-canary” y guardarlo en una ruta fácil de recordar. Yo recomiendo dejarlo en la ruta de aplicaciones por defecto, sea cual sea tu sistema operativo.

Luego para abrir la aplicación entramos en *android-canary\bin*, e inicia studio64.exe (o studio.exe si tu arquitectura es de 32 bits).

En Mac

- Descarga la aplicación
- Una vez terminado descomprímela y haz doble click en ella
- Arrastra el archivo a la carpeta Aplicaciones y ejecútala.

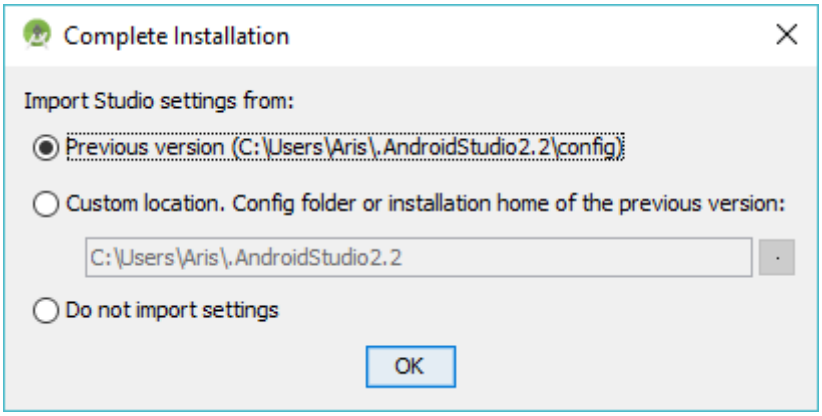
En Linux

- Descarga la aplicación
- Descomprime el .ZIP
- Renombra la carpeta
- Abre un terminal *carpetaAndroid\bin* y ejecuta studio.sh

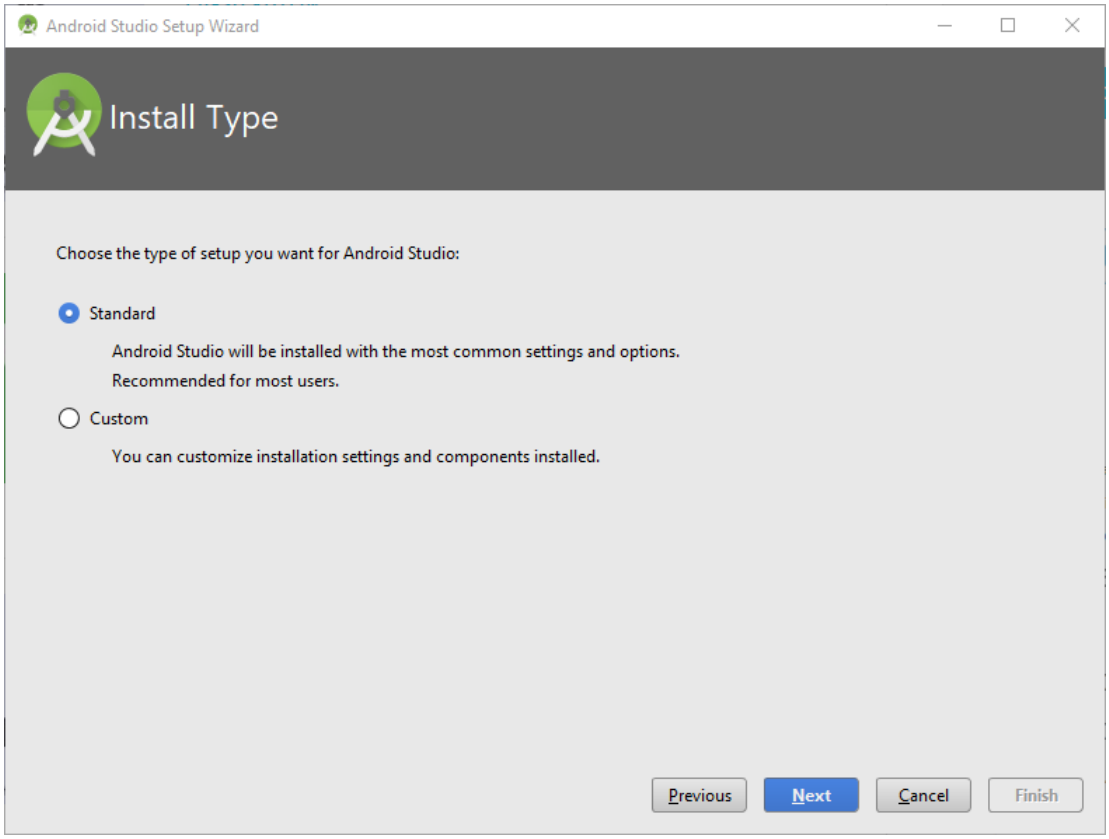
Configurando Android Studio Canary version

Una vez descargado y abierto por primera vez nos saldrá un aviso de que si queremos importar la configuración de android studio. Si tenéis la versión estable instalada y trabajáis con ella os recomiendo importarla para que todo siga igual. Yo seleccionaré “**Do not import settings**” para

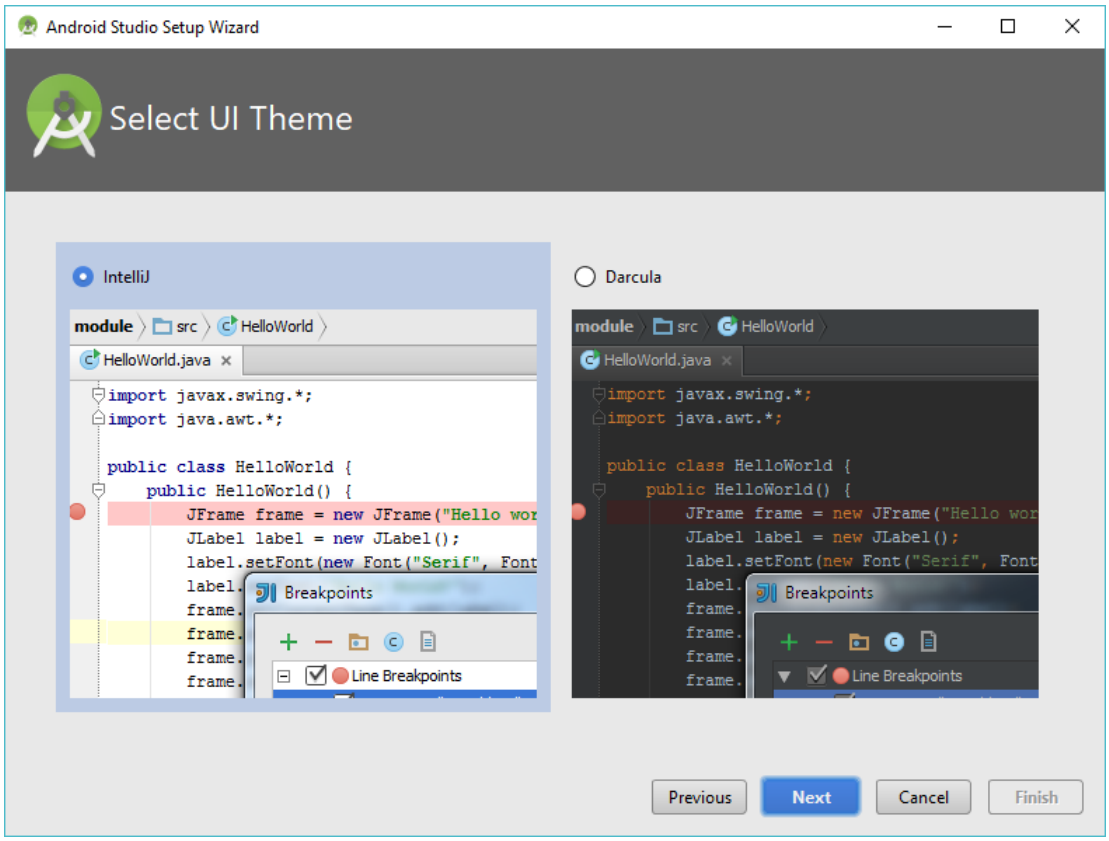
mostraros todo el proceso.



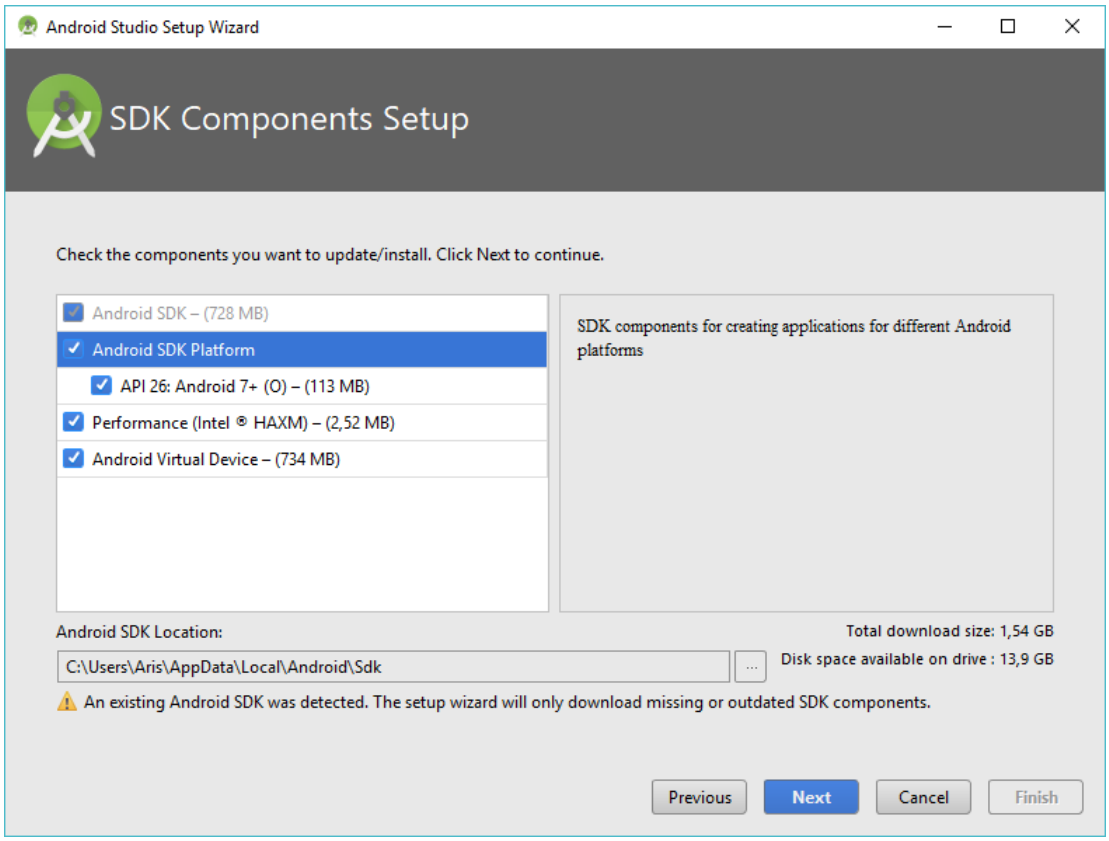
Pulsamos **OK** y nos saldrá una pantalla que nos da la bienvenida, pulsamos en **Next**.



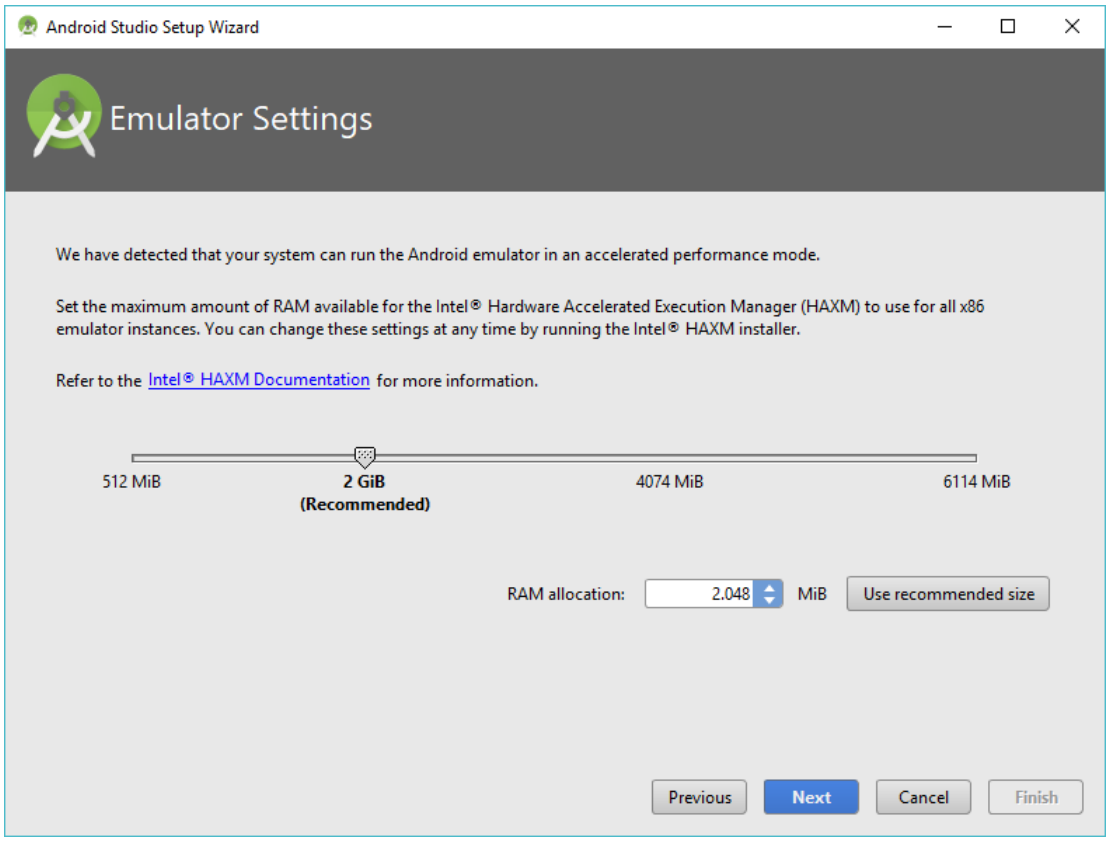
Ahora debemos seleccionar si queremos la configuración **standar** o no, en este curso seguiré la **custom** para recomendaros lo que a mi punto de vista hay que poner y que no.



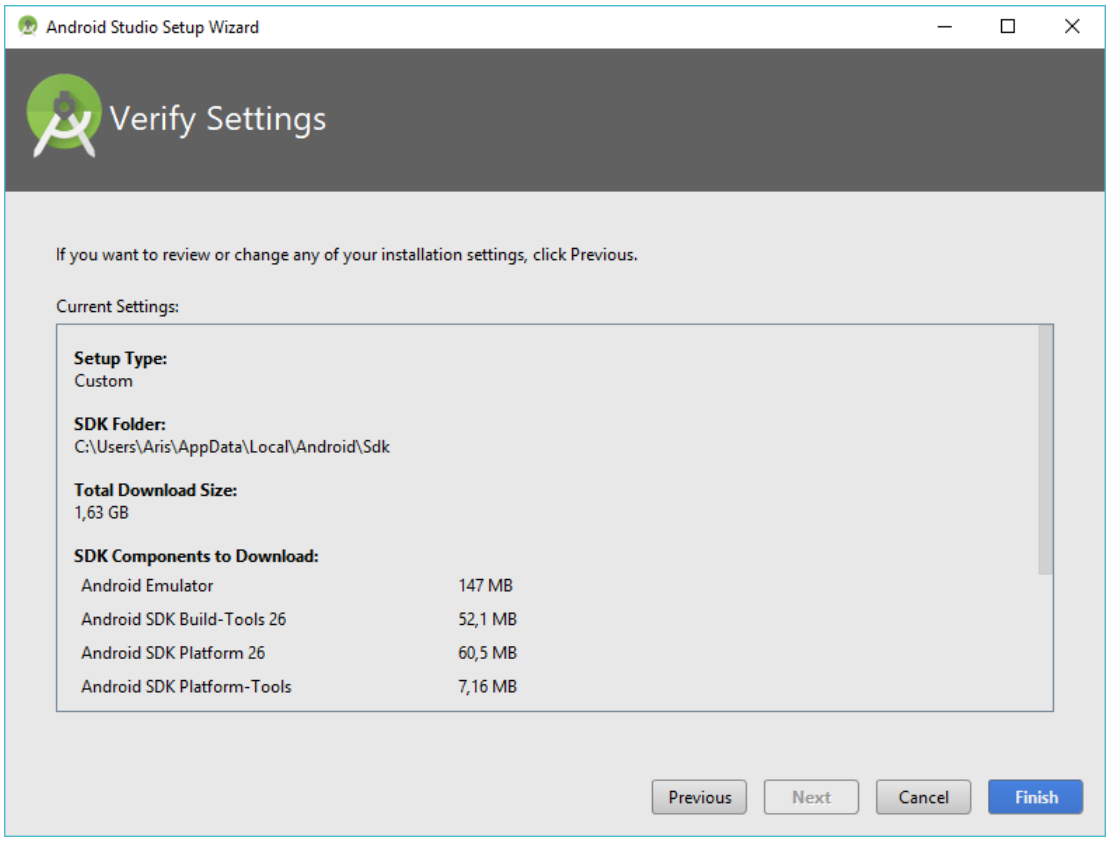
Lo primero que nos mostrará será la opción de elegir diseño, como defecto pone el "**IntelliJ**". Yo trabajo 8 horas al día con este programa y os recomiendo muchísimo la opción "**Darcula**" si vais a dedicarle muchas horas, pues el fondo blanco acaba cansando mucho la vista.



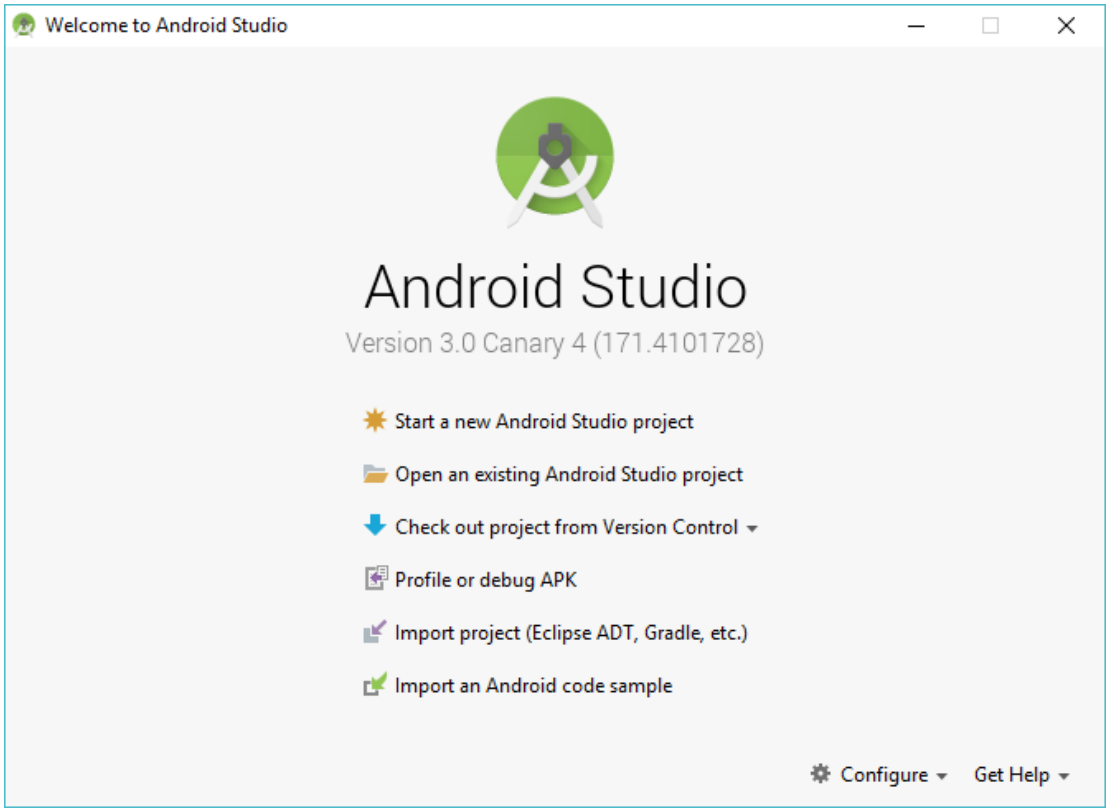
El siguiente paso será elegir los componentes del SDK, si vais a usar el emulador para las aplicaciones tenéis que seleccionarlo, sino podéis obviarlo. Yo como tengo la otra versión, solo me sale el API 26, si no tenéis ninguna por lo menos poned desde **android 4.4** hasta **android 7**.



Si hemos seleccionado la opción del emulador, aquí podemos elegir la cantidad de ram que le vamos a permitir usar. Como norma general te recomiendan 2GB, pero a mi nunca me ha ido bien así. R**ecomiendo al menos 4GB pero NUNCA más de la mitad de la ram del equipo**.



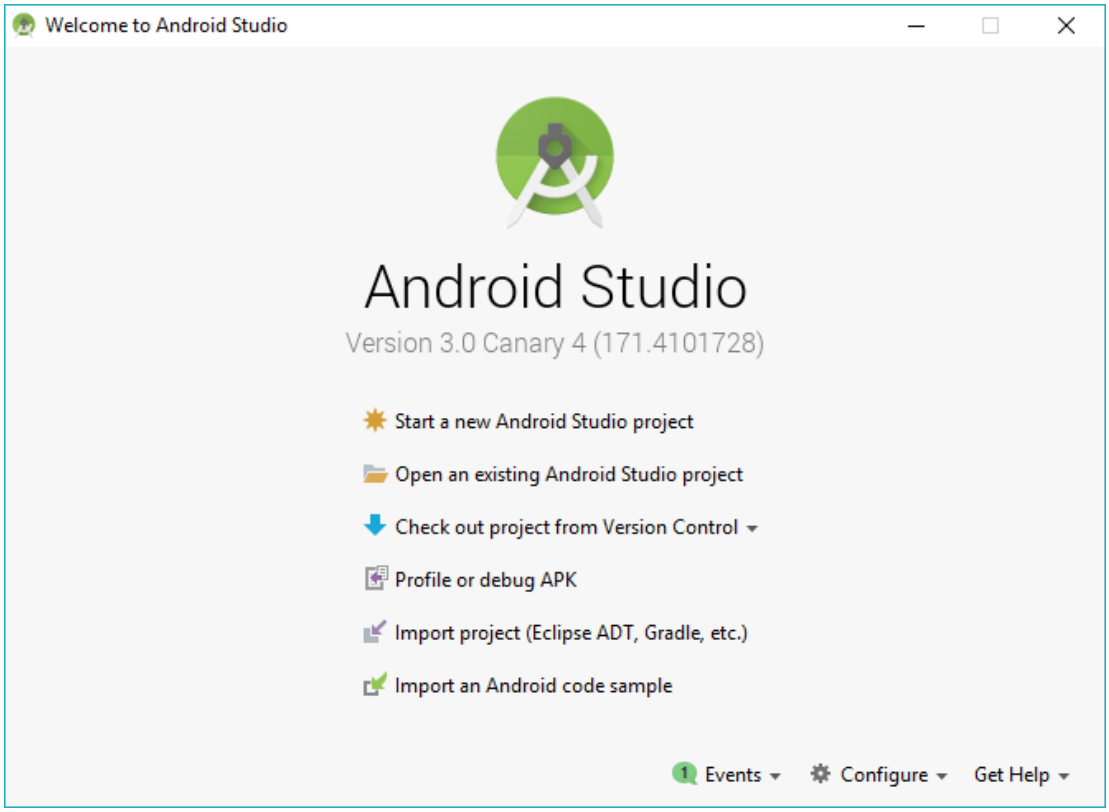
Pulsamos en **Finish** y empezará a descargar todos los componentes necesarios.



Una vez finalice ya tendremos nuestro Android Studio listo para trabajar con Kotlin.

Nuestra primera App

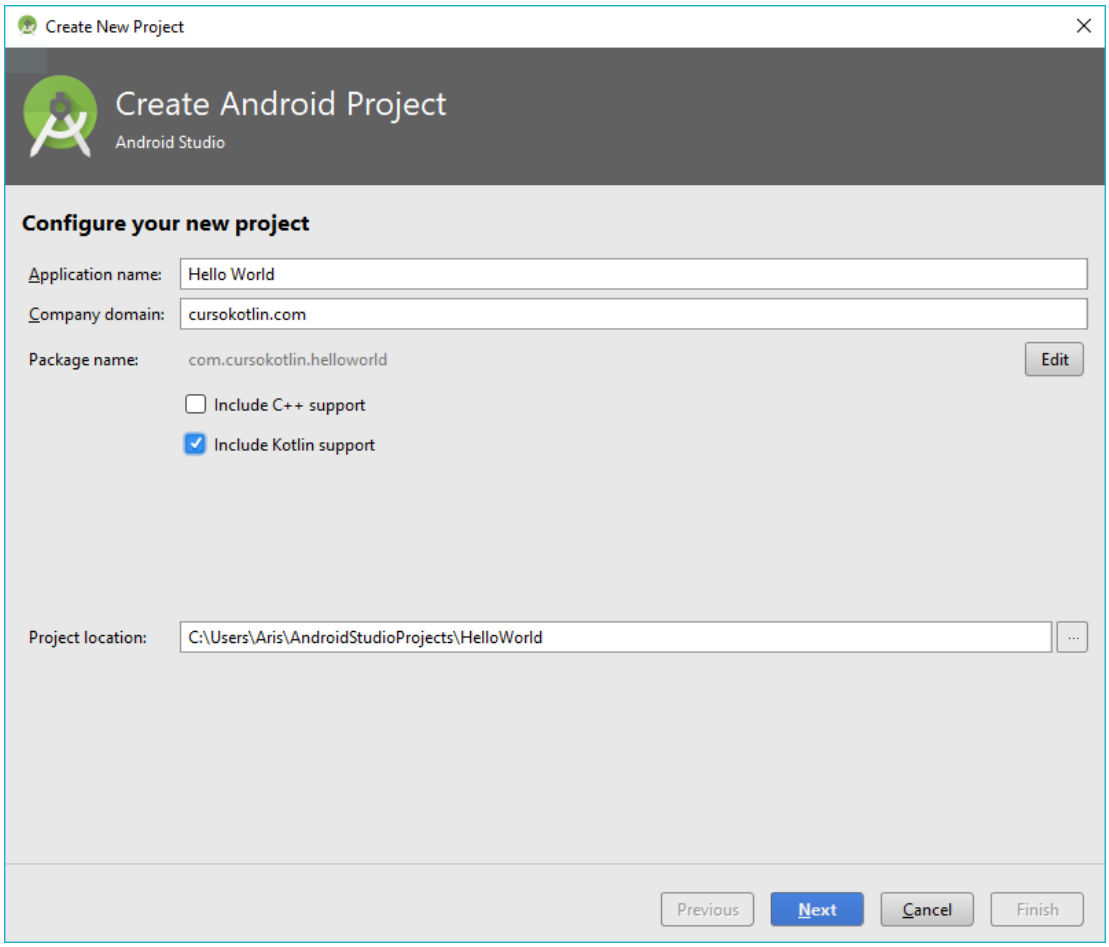
Una vez configurado nuestro entorno, lo que vamos a hacer será un proyecto nuevo, para ello, abriremos Android Studio y haremos click en "**Start a new Android Studio project**"



Cuando lo hagamos, la siguiente pantalla nos pedirá una serie de datos, el primero será el nombre de la aplicación, que deberá ser claro y conciso. Como es tradición en el mundillo, nuestra primera app se llamará "**Hello World**".

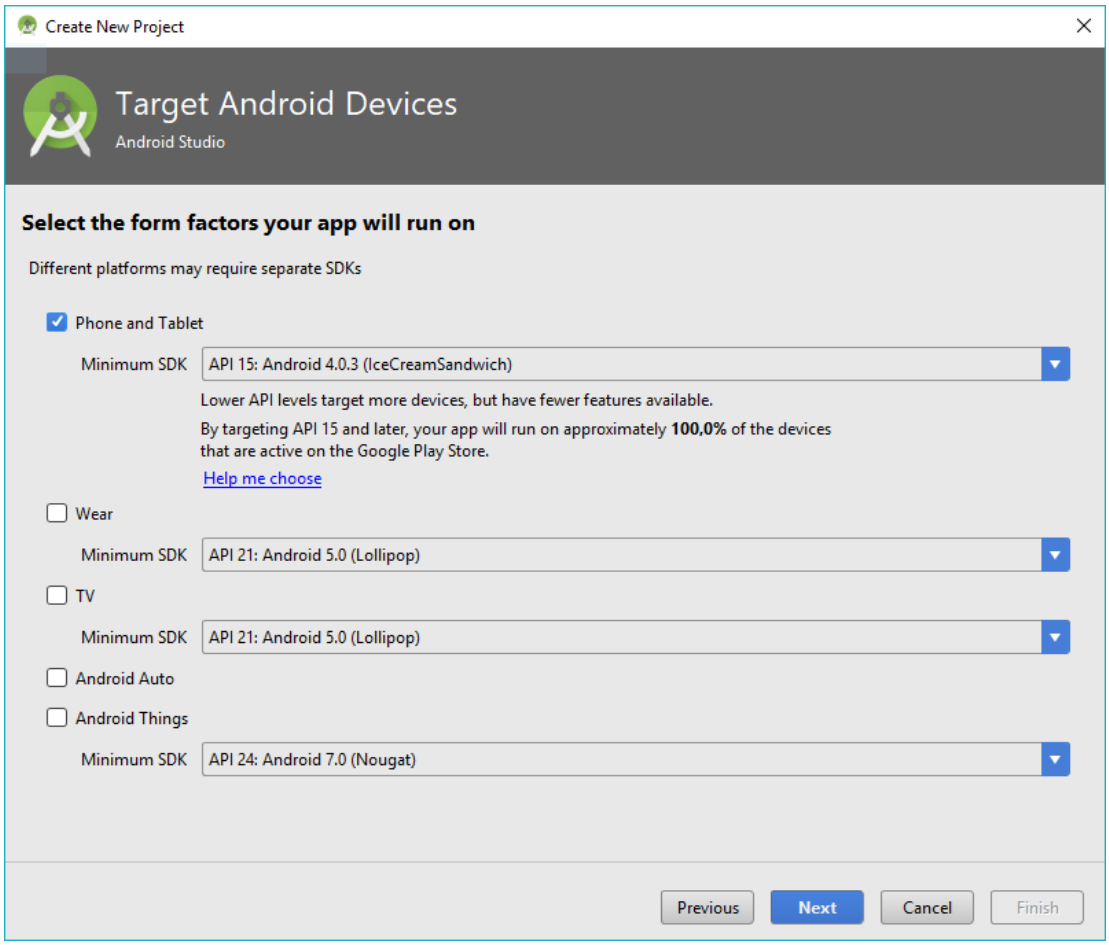
A continuación pondremos el "Company domain", este campo se usa para generar un paquete, este es un identificador único para cuando publiquemos nuestra app en la store de Google Play. Para el dominio, la estructura básica que se suele usar es com.tunombre o com.nombredetucompañía, para que, junto a el nombre de la app terminen de generar nuestro

identificador. Como podéis ver en el nombre del paquete, está todo en minúsculas, esto es así para evitar conflictos con clases e interfaces. Luego, seleccionamos el directorio donde guardaremos la aplicación, y para finalizar es imperativo marcar la opción "**include Kotlin support**".



Al pulsar sobre next, nos saldrán unas nuevas opciones, que nos permitirán seleccionar para que dispositivo queremos desarrollar. En este curso os enseñaré para a programar para móviles y tablets, pero una vez lo hayáis terminado, tendréis conocimientos más que de sobra para hacer frente a las otras plataformas.

Para elegir en SDK mínimo (mínima versión en la que nuestra app funcionará), debemos pensar a cuanto público queremos llegar. Una app que se desarrolle en Android 6, solo podrá acceder a un 4'7% del sector total (para verlo haz click en "Help me choose"). Aunque cada uno usa sus truquitos, yo suelo empezar por la API 15, que corresponde a más del 97% de los móviles que están funcionando a día de hoy, ya luego si por cualquier petición del cliente, hay que añadir una funcionalidad nueva no permitida en dicha versión, la subo.



A continuamos nos mostrará una pantalla con varias pantallas a elegir. Seleccionamos "Empty Activity" y continuamos a la última pantalla en la que podemos seleccionar los nombres de la Activity y el Layout, esto lo explicaré a continuación, así que por ahora dejaremos el nombre por defecto, con las casillas marcadas y pulsamos en "**finish**".

¿Qué es una Activity?

Una vez generado el proyecto, estaremos delante de nuestra primera activity, llamada MainActivity. Una Activity es cada una de las pantallas que ve el usuario, en ellas se declaran los métodos (funciones) que se pueden hacer. Por ejemplo, imaginemos que la aplicación es de cocina, pues una activity sería cortar las verduras, hervir, freír, etc. Las activities tienen dos partes, la lógica y la visual (los layouts que veremos a continuación).

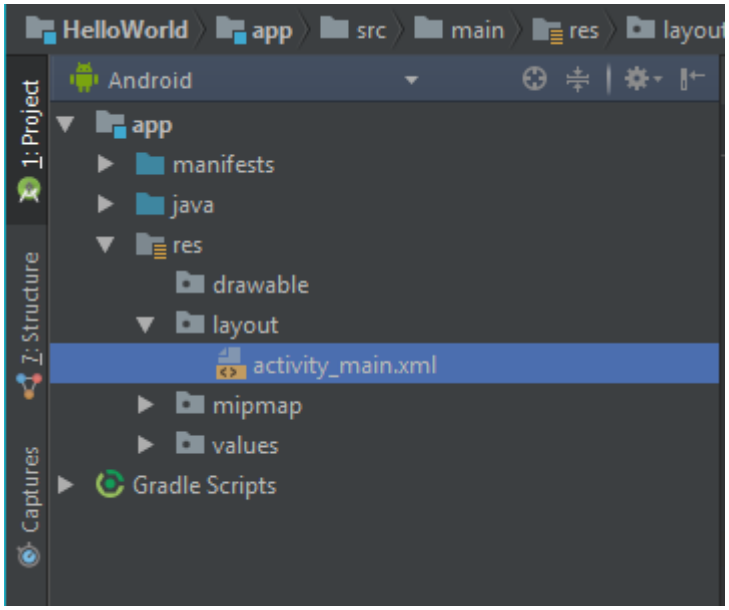
En la parte lógica, que la desarrollaremos en Kotlin, daremos respuesta a cada unas de las interacciones del usuario. Si seguimos con el ejemplo anterior, imaginemos que el usuario pulsa el botón de freír, este evento llegaría a nuestra clase y ahí ejecutamos la acción. No quiero entrar más en detalle, porque lo veremos más tarde.

La activity es básicamente una clase kotlin (.kt) que **extiende de AppCompatActivity**, que tiene los métodos necesarios para poder comunicarnos con Android. Así que por defecto siempre tendremos la función "*OnCreate*" que será el encargado de crear nuestra actividad y asignarle un layout.

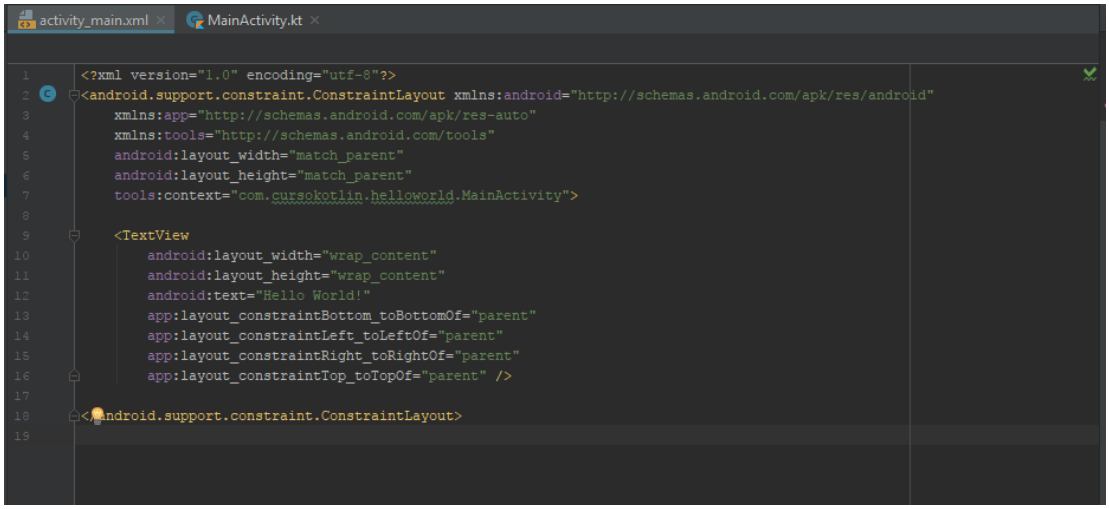
¿Qué es un Layout?

El layout es la segunda parte de la actividad, la interfaz. En ella se agregarán los componentes como los botones, imágenes y demás. Estos archivos son muy simples y se pueden completar arrastrando los componentes o picándolos en XML.

Para acceder a él, debemos ir a la estructura del proyecto, que está en el lado izquierdo.



Con hacer doble click sobre el archivo se nos abrirá y veremos algo así.



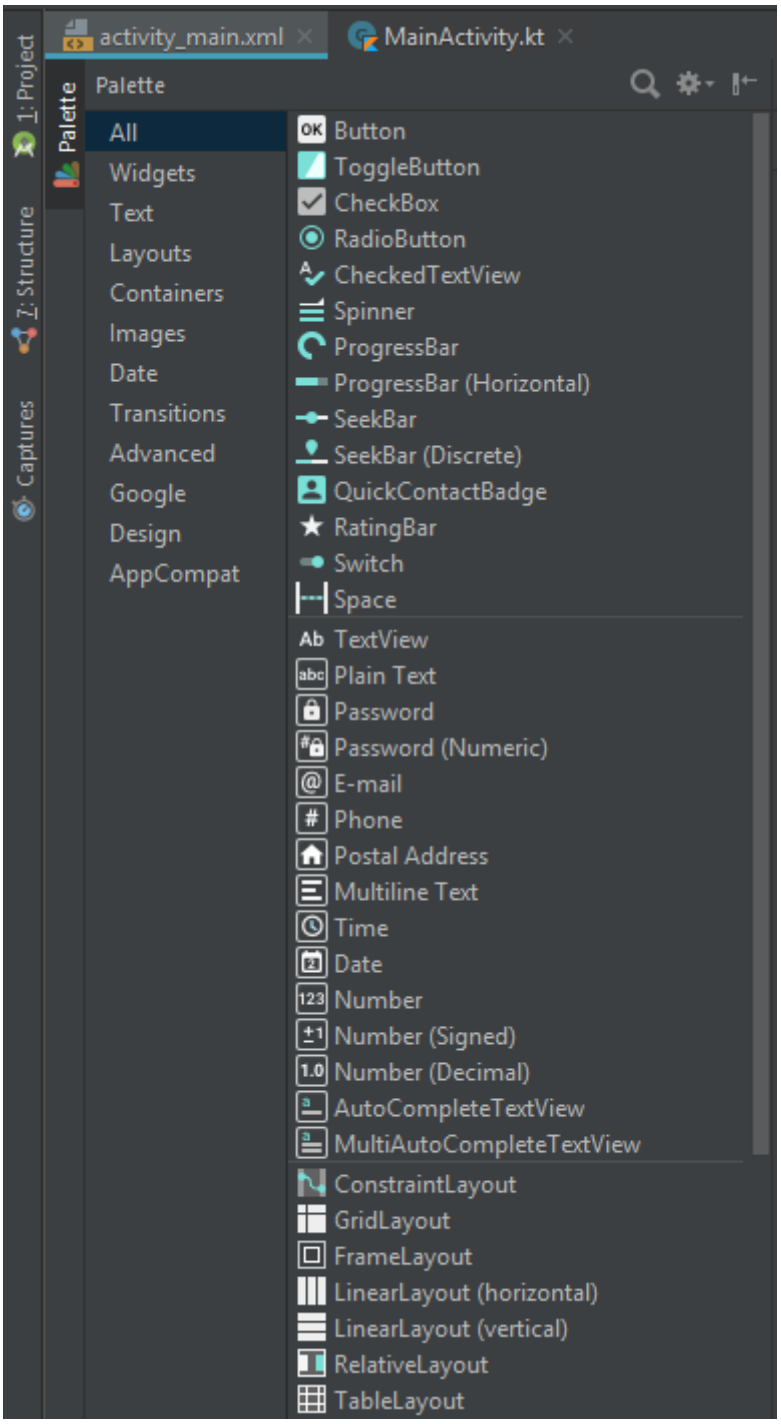
Parece un poquito complicado al principio, pero una vez se explique se irá toda la dificultad. Lo primero que tenemos es el código en XML que nos genera por defecto, como podemos ver, está anidado. Tenemos un padre "*android.support.constraint.ConstraintLayout*" y dentro un "*TextView*" que es un componente muy básico para mostrar texto no editable en la pantalla. A la derecha, en las pestañas del lateral, hay una llamada "preview", que si la pulsamos nos mostrará a tiempo real el diseño de la vista.

Antes de continuar, vamos a cambiar a el padre para usar un layout más sencillo, así que simplemente cambiamos "*android.support.constraint.ConstraintLayout*" por "*RelativeLayout*". Como podéis ver, la etiqueta que cierra la anidación también ha cambiado, porque hace referencia al RelativeLayout. Así que nuestra vista quedaría así (he quitado varias líneas que hacían referencia al constrainLayout, en un futuro lo veremos más a fondo).

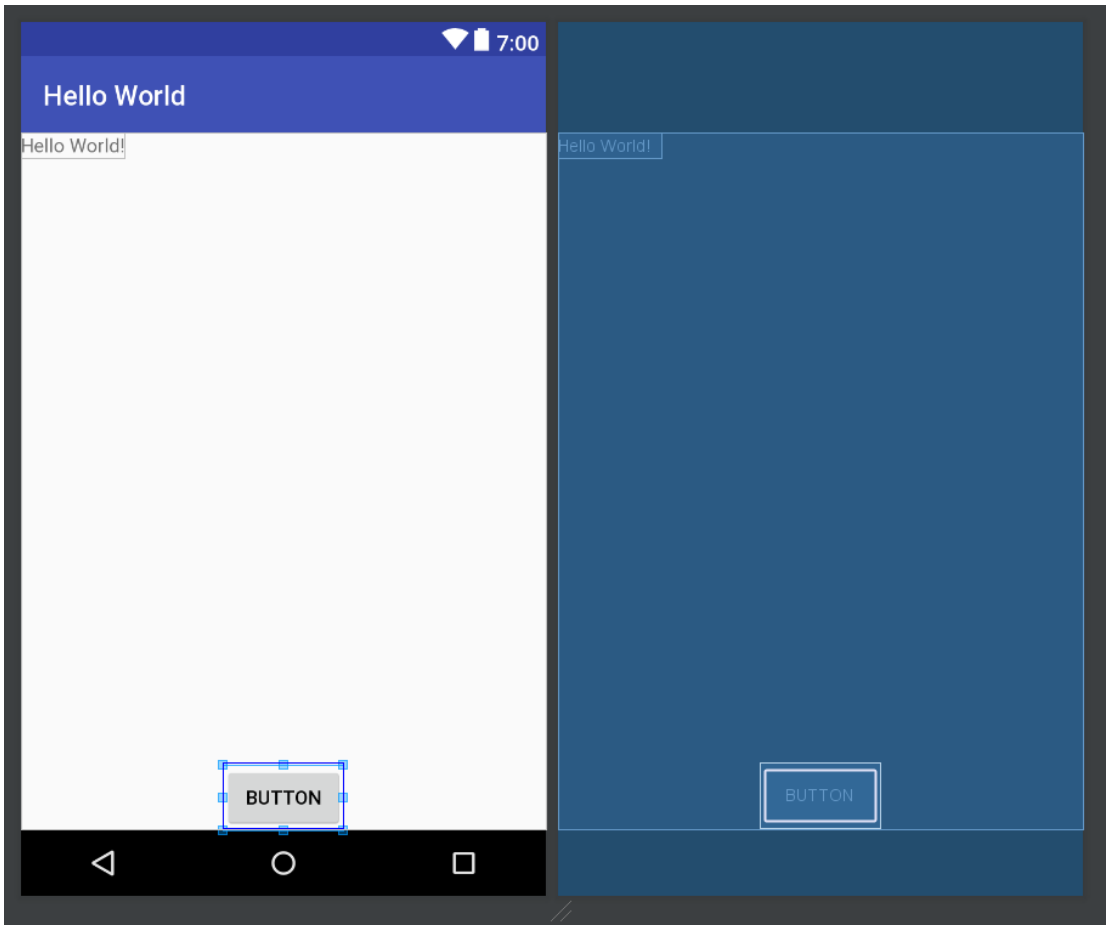
```xml
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.cursokotlin.helloworld.MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</RelativeLayout>
```

Ahora vamos a ver la parte más visual del layout, para ello, en la parte inferior izquierda, veremos dos pestañas "**Design**" y "**Text**", desde aquí podemos cambiar la forma de trabajar con las vistas. Si nos vamos a **Design**, veremos que el código cambia por la vista y han aparecido nuevas columnas.

Desde aquí, si podemos manipular los componentes que están dentro de la vista, e incluso añadir nuevos. Para ello vamos a coger un botón de la caja de componentes que está en la esquina superior izquierda.



Simplemente pulsamos en el botón y lo arrastramos a la parte de la vista que queramos, por ejemplo abajo del todo.



Una vez lo hagamos, y mientras lo tengamos fijado, nos aparecerán una serie de campos en el lateral derecho, estos, son los atributos del botón. Los podemos editar desde aquí o desde el XML. Como esta es la primera práctica, lo haremos desde aquí, pero en las siguientes lo haremos desde el código, pues es la forma más óptima para dejar nuestras vistas impolutas.

Los tres atributos básicos que debemos añadir:

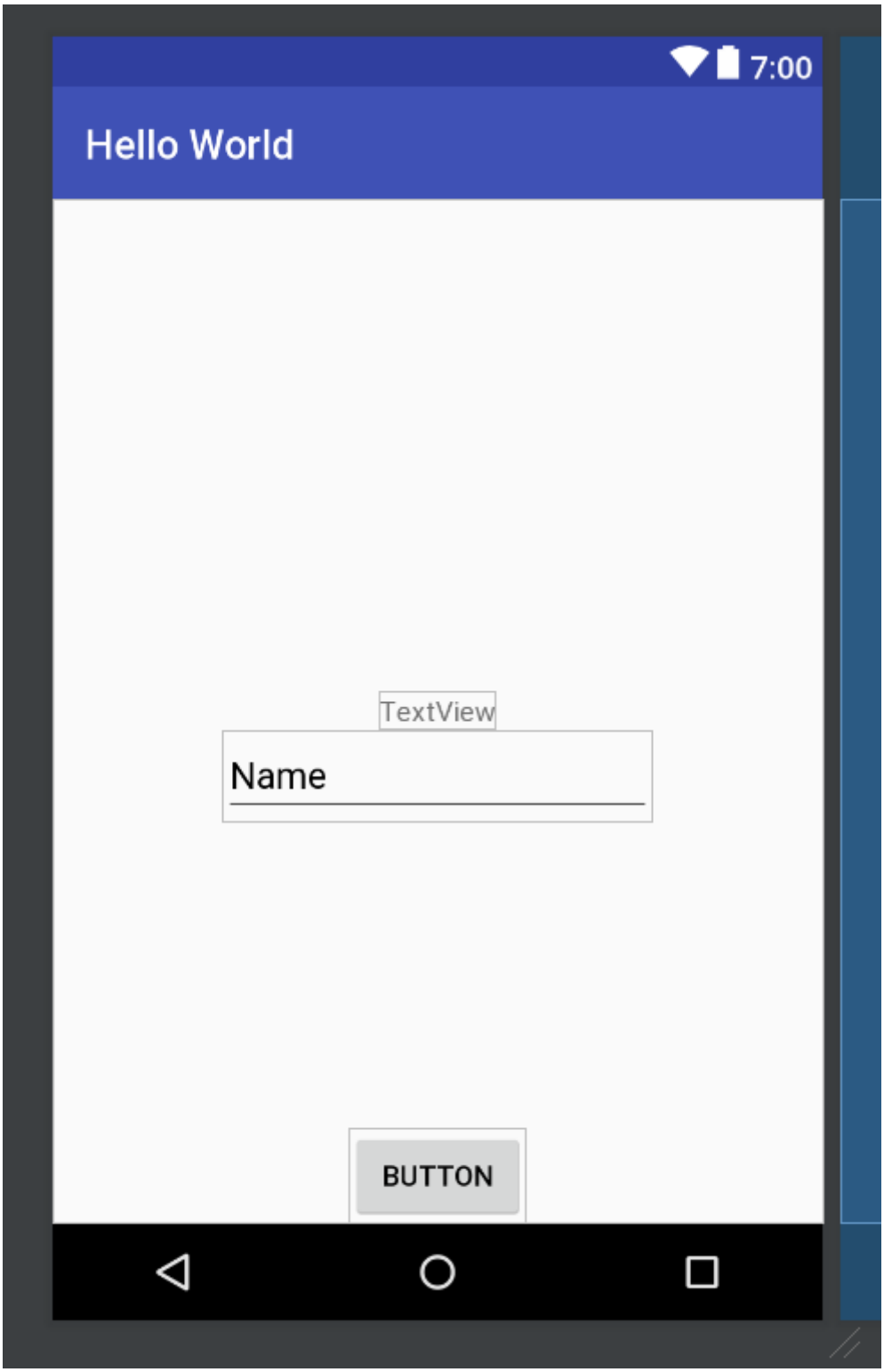
- **ID**: Es el identificador único del componente, con él, podremos definir en nuestra activity dicho componente para poder trabajar con él. Como es un botón que nos hará cambiar de activity, lo llamaré "*btnChangeActivity*".
- **layout_width**: Este campo hace referencia a la anchura del componente, con este campo (que es obligatorio), podemos fijar la anchura a través de dos opciones. "*Wrap_content*" que añadirá una anchura lo suficientemente grande para incluir todo el contenido (en este caso el texto "BUTTON") y "*Mach_parent*" que cogerá todo el espacio que su padre (en este caso el RelativeLayout) le permita.
- **layout_height**: Permite asignarle la altura a cada componente. Sigue el mismo comportamiento que el "*layout_width*".

Para este ejercicio dejaremos los dos con "*wrap_content*" pero os animo a probar a añadirle el "*match_parent*" a algunos de los componentes para que vayáis entendiendo el comportamiento.

Componentes básicos

Ahora que ya sabemos lo básico, vamos a añadir 2 componentes más. Un TextView y un EditText. No deberíais tener problema en hacerlo. La idea es un TextView que te pregunte el nombre y un EditText donde poder escribirlo. Simplemente debemos arrastar los dos

componentes al centro de la pantalla (uno debajo de otro) y le asignaremos un ID. Los componentes se llaman “TextView” y “Plain Text” en la columna. Deberíamos tener algo así al acabar.



Un poquillo feo ¿No? Lo arreglaremos yendo a el XML y asignando algunos atributos.

Lo primero que haremos será añadir los atributos a los dos nuevos componentes, pero esta vez desde el código, empezaremos por el TextView. Para ello simplemente hay que añadir la siguiente línea:

```
1 android:id="@+id/tvName"
```

La explicación es muy simple, lo primero que hacemos es llamar a Android, accediendo a la propiedad ID. Acto seguido, con el @+id estamos creando una referencia del componente en el archivo R.java de android (Lo expliquaré en el siguiente punto) y para acabar le damos el nombre que queramos, en este caso, al TextView que nos preguntará el nombre, lo llamaremos tvName. Aunque se puede añadir en cualquier parte del XML, la línea del ID como norma general en las prácticas del buen uso, suele ir la primera.

Seguramente cuando cambiaste el nombre del id, el TextView se fue para arriba del todo y se marcó una línea roja en el otro componente ¿Verdad? en la línea "*android:layout_below="@+id/textView"*" esto se debe a que al estar en un RelativeLayout, debemos indicarle a cada componente donde va a estar. El **layout_below** lo que hace es decir que dicho componente estará justo debajo del que tenga la ID llamada "textView", pero como ahora acabamos de renombrar el TextView a tvName, el EditText no encuentra ningún id con dicha referencia, así que cambiaremos *android:layout_below="@+id/textView"* por *android:layout_below="@+id/tvName"* y todo volverá a su sitio.

Ahora habrá que cambiar el texto que trae por defecto por el que queremos. Así que localizamos la línea "*android:text="TextView*" y cambiamos el texto a "¿Cómo te llamas?".

Ahora ya lo tendríamos listo, pero en comparación con el EditText, se ve bastante pequeño, así que accederemos a un atributo más.

```
1 android:textAppearance="@style/TextAppearance.AppCompat.Title
```

Es muy parecida a las anteriores, *textAppearance* lo único que hace es decirle al componente que su apariencia será controlada a través de un estilo. Un style o estilo es simplemente una serie de atributos reutilizables, imaginemos que tenemos una App que tiene varios textos iguales (fondo de un color, un tamaño de texto predefinido, etc), podemos añadir los mismos atributos a cada uno de ellos, con los riesgos que conlleva (si luego hay que cambiar el color del texto, habría que ir a todas las pantallas que llevasen textos y cambiarlos uno a uno) o podemos asignarles un mismo estilo al que todos accederán, así si se cambia algo, automáticamente todos los textos lo harían.

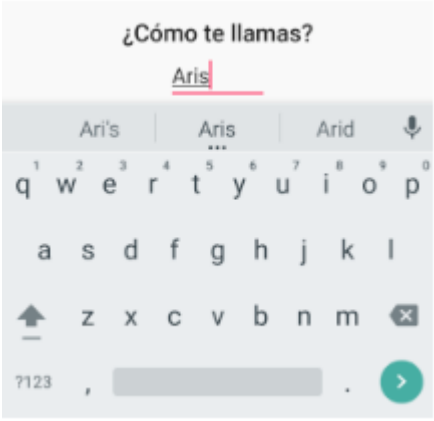
Así que lo único que hacemos es referencia al archivo style con el "@style" y luego el nombre de dicho estilo. Nos habrá quedado algo así.

```
1 <TextView
2         android:id="@+id/tvName"
3         android:layout_width="wrap_content"
4         android:layout_height="wrap_content"
5         android:layout_centerHorizontal="true"
6         android:layout_centerVertical="true"
7         android:textAppearance="@style/TextAppearance.AppCompat.Title"
8         android:text="¿Cómo te llamas?" />
```

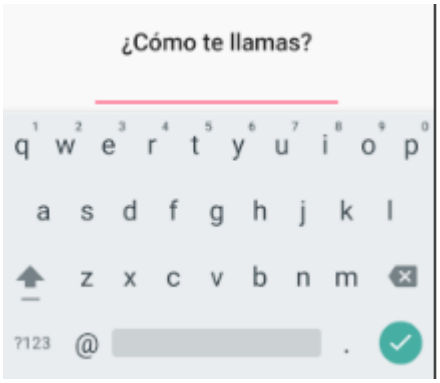
Ahora nos toca el EditText que sería hacer básicamente lo mismo, añadir un ID y listo. Este se llamará etName. Y de paso comentaré un par de atributos que seguramente se nos habrán generado automáticamente.

- **android:layout_below**: Explicado anteriormente, cabe destacar que este atributo es solo para los RelativeLayout y los que hereden de él. Además del below tenemos algunos muy parecidos como layout_above, que haría lo mismo pero en vez de ponerse debajo, se pondría encima.

- **android:layout_centerHorizontal="true"**: Nos pondrá siempre el componente en el centro de la vista en el eje horizontal. Así aunque el móvil sea muy grande o muy pequeño siempre se mantendrá centrado.
- **android:ems="10"**: Añade una anchura al EditText suficiente para que se vean X caracteres en la pantalla. Por ejemplo ahora tenemos puesto 10, si se añaden más, irán desapareciendo de la vista, probad a cambiar el número para que no os quede duda alguna.
- **android:inputType="textPersonName"**: El atributo inputType nos permite que una vez hagamos click en el editText, el teclado que se muestre, esté adecuado al tipo de información que veremos. Os pongo un pequeño ejemplo para entenderlo más fácil. Con textPersonName el teclado que se mostraría sería así.



Pero si ahora cambiamos el valor a *textWebEmailAddress* se vería así.



La diferencia es casi ínfima, aquí se vería el @ y en el anterior no. Aunque sea un cambio tan pequeño, estos pequeños detalles hacen que el usuario se sienta cómodo con nuestra app, así que hay que cuidarlos.

Por último tenemos un text en el EditText. El problema de dicho atributo en este tipo de componentes es que cuando hagamos click en él, ese texto tendremos que borrarlo para añadir nuestro nombre por ejemplo, por eso los chicos de Google crearon un atributo muy cómodo, *android:hint="Name"* que hace básicamente lo mismo, pero cuando el usuario escriba, el texto desaparecerá y no habrá que borrarlo primero. Así que nuestro XML una vez terminado se verá así.

```xml
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.cursokotlin.helloworld.MainActivity">

    <Button
        android:id="@+id/btnChangeActivity"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:layout_centerHorizontal="true"
        android:text="Enviar" />

    <TextView
        android:id="@+id/tvName"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        android:textAppearance="@style/TextAppearance.AppCompat.Title"
        android:text="¿Cómo te llamas?" />

    <EditText
        android:id="@+id/etName"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/tvName"
        android:layout_centerHorizontal="true"
        android:ems="10"
        android:inputType="textWebEmailAddress"
        android:hint="Name" />

</RelativeLayout>
```

En la primera parte de este artículo terminamos de explicar un poquillo por encima las activities y los layouts y terminamos de maquetar nuestra primera pantalla. Ahora es el momento de conectar dicha interfaz con nuestra clase.

Conectando la vista al código

Lo primero que haremos será referencias nuestros componentes (en este caso el editText y el botón) en el código. Para ello, Kotlin lo hace de un modo muy sencillo, simplemente tenemos que usar la id que le asignamos a cada uno de los componentes en el capítulo anterior.

```
1     override fun onCreate(savedInstanceState: Bundle?) {
2         super.onCreate(savedInstanceState)
3         setContentView(R.layout.activity_main)
4         btnChangeActivity.setOnClickListener { checkValue() }
5     }
6
7     fun checkValue(){
8         if(etName.text.toString().isEmpty()){
9             Toast.makeText(this, "El nombre no puede estar vacío", Toast
10        }else{
11            //Iremos a otra pantalla
12        }
13    }
```

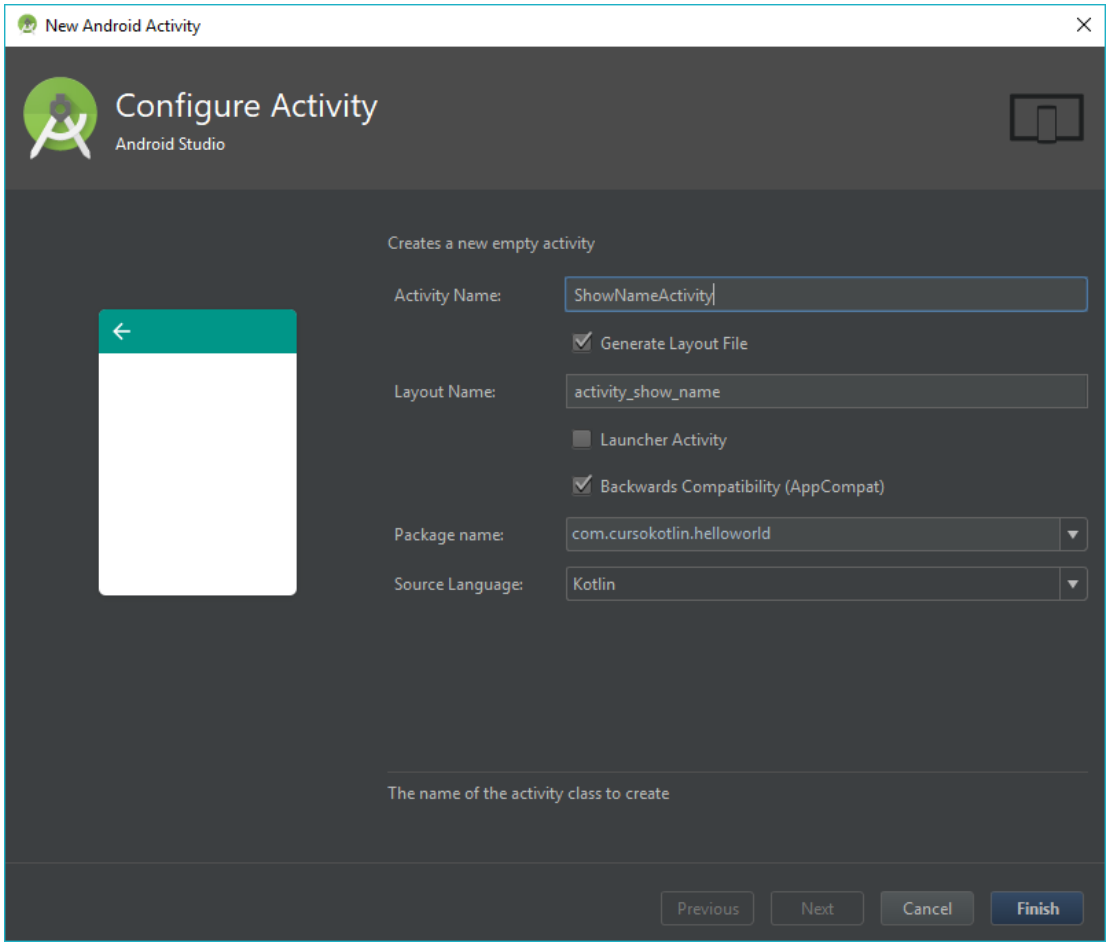
Lo primero que hemos hecho ha sido llamar al botón, y diciéndole que cuando se produzca el evento del pulsado sobre él que lance el método *checkValue()*. Esto lo hemos hecho con la función *setOnClickListener* que es el encargado de estar "escuchando" hasta que el botón haya sido pulsado.

El método *checkValue()* será el encargado de comprobar si el campo nombre está vacío. Si lo está, lanzará un toast diciendo que no debe estar vació, sino iremos a la nueva pantalla, pasándole el nombre, pero primero crearemos dicha pantalla.

Generando nuestra segunda activity

Ahora vamos a generar una segunda pantalla que será la que se abrirá cuando hayamos escrito un nombre. Podríamos hacerlo en la misma pantalla, pero quiero hablaros de como pasar información entre activities así que manos a la obra.

Para ello vamos a la ruta app/nombreDelPaquete/ y hacemos click derecho sobre él. New/Activity/Empty Activity.



Nos saldrá una pantalla igual a la imagen y la dejaremos así, le hemos cambiado el nombre, tenemos marcado que nos genere el layout y seleccionando el lenguaje Kotlin. Pulsamos en **Finish**.

Una vez creada vamos a empezar maquetando la pantalla. Esta será muy sencilla para no alargar mucho la entrada.

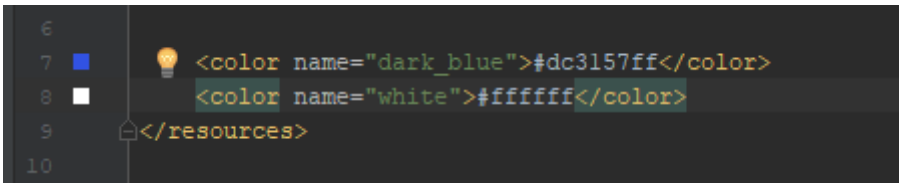
Lo primero que vamos a hacer será cambiar el ConstrantLayout por un RelativeLayout como hicimos con la otra vista, le pondremos un fondo y en el centro habrá un texto dándonos la bienvenida. En la parte inferior un botón para volver a la otra pantalla.

```xml
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.cursokotlin.helloworld.ShowNameActivity"
    android:background="@color/dark_blue">

    <TextView
        android:id="@+id/tvGreeting"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textColor="@color/white"
        android:text="@string/welcome"
        android:layout_centerInParent="true"
        android:textSize="30sp"/>

    <Button
        android:id="@+id/btnBack"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/back"
        android:layout_centerHorizontal="true"
        android:layout_alignParentBottom="true"/>

</RelativeLayout>
```

Aunque sea muy sencillo, he añadido algunas cositas que quiero comentar. Para empezar, el padre tiene un atributo *android:background="@Color/dark_blue"* que si lo ponéis os va a dar error. Eso se debe a que el color "dark_blue" lo he creado en el archivo colors, y será lo que tengáis que hacer ahora, porque aunque se pueda meter el color ahí directamente (en código hexadecimal) lo correcto es hacerlo en dicho fichero por si un día tenemos que cambiar el color base de una app no tengamos que hacerlo en cada una de las pantallas, sino cambiando solo el color desde ahí. Así que vamos a *app/res/values/colors* y añadimos los dos colores que he creado.

```xml
<color name="dark_blue">#dc3157ff</color>
<color name="white">#ffffff</color>
```

Para poder cambiar el color solo hay que cambiar el código hexadecimal, la forma más fácil es haciendo click en el cuadradito del lateral izquierdo que contiene cada recurso, y desde ahí seleccionar el nuevo color.



Y al igual que ha pasado con los colores los textos están referenciados en el archivo String, de esto hablaremos más adelante, pero voy a explicar lo básico.

Vamos a *app/res/values/strings.xml* y tendremos que añadir cada uno de los textos que usemos con una etiqueta que será la referencia. Quedaría así.

```xml
<string name="welcome">welcome %s</string>
<string name="back">volver</string>
```

Os estaréis preguntando que significa el **%s** que hay en la string "welcome". Ese parámetro vale para que Android entienda que el %s será sustituido por un texto que todavía desconocemos. Así que cuando estemos en la segunda pantalla y se esté creando la vista, se lo tenemos que pasar.

Comunicación entre pantallas

Ahora volvamos a la clase MainActivity.kt

En el método *checkValue(),* en la condición else, iremos a nuestra nueva actividad y para ello usaremos un **Intent**. Un Intent es un objeto que contiene instrucciones con las cuales android se comunica.

```
1  fun checkValue(){
2          if(etName.text.toString().isEmpty()){
3              Toast.makeText(this, "El nombre no puede estar vacío", Toast.
4          }else{
5              val intent = Intent(this, ShowNameActivity::class.java)
6              intent.putExtra("name", etName.text.toString())
7              startActivity(intent)
8          }
9      }
```

Vamos a explicar un poquito más a fondo que hemos

```
1 val intent = Intent(this, ShowNameActivity::class.java)
```

Aquí creamos el intent, le tenemos que pasar el contexto (this) y luego la pantalla a la que queremos ir, en este caso **ShowNameActivity**.

```
1 intent.putExtra("name", etName.text.toString())
```

El intent tiene la posibilidad de añadirle "extras", que no son más que datos para recuperar en otra parte de la aplicación. Para poder recuperarlos hay que ponerles una clave a dichos datos. En este caso estoy pasando el nombre que el usuario escribió (*etName.text.toString()*) y lo estoy pasando con la clave **name**. con esto hacemos que cuando llegue el intent a la otra pantalla, solo tenga que buscar si hay algún valor con la clave **name**, y nos devolverá el nombre.

Para finalizar llamamos a la función *starActivity(intent)* que nos lanzará la actividad declarada en el intent que le pasamos.

Recuperando valores del intent

Ya tenemos casi acabada nuestra aplicación, ahora debemos ir a **ShowNameActivity.kt**, recuperar el nombre y pintarlo.

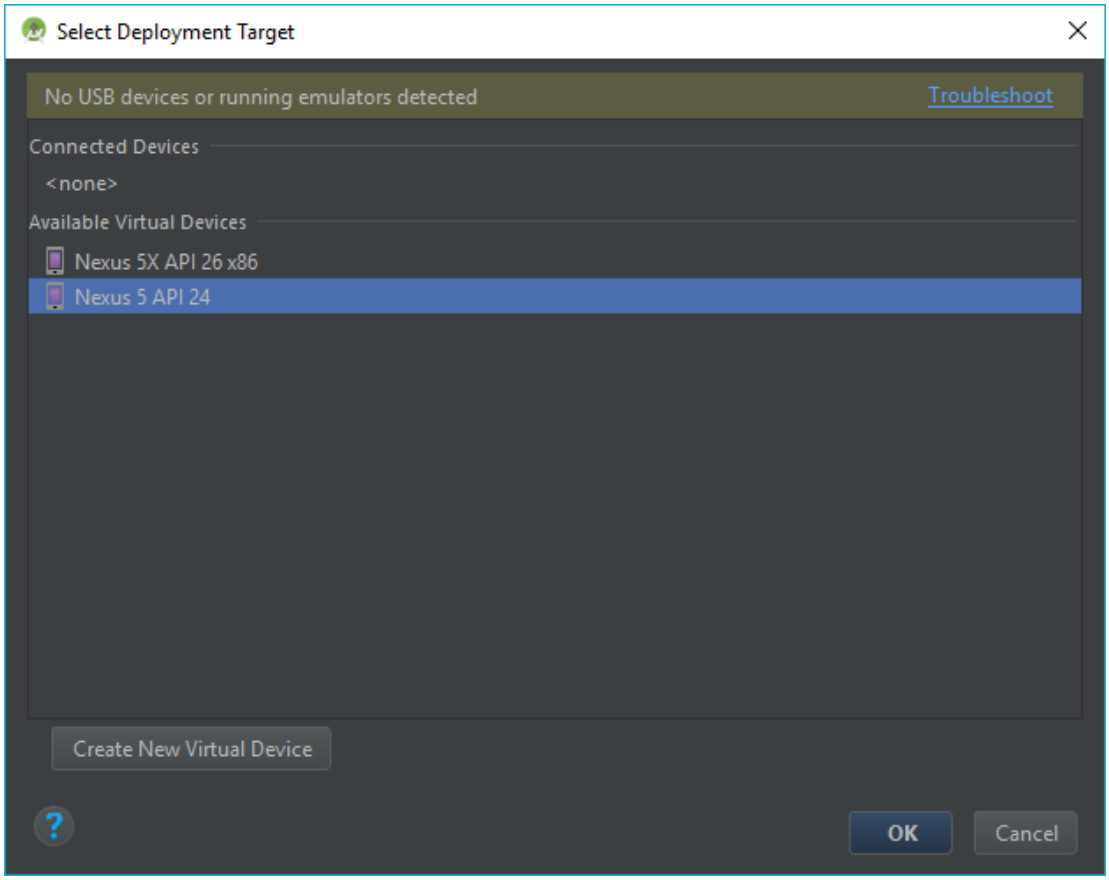
```
class ShowNameActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_show_name)
        getAndShowName()
        btnBack.setOnClickListener { onBackPressed() }
    }

    fun getAndShowName(){
        val bundle = intent.extras
        val name = bundle.get("name")
        tvGreeting.text = getString(R.string.welcome, name)
    }
}
```

Cuando se lanza esta actividad hará dos cosas, la primera será llamar al método que hemos creado *getAndshowName()* que sacará del bundle (un objeto dentro del intent donde almacena la información) el nombre a través de la clave "**name**".

Ahora que tenemos el nombre, accedemos al TextView y le ponemos el texto que deseamos, en este caso, como es una String necesitaremos la función *getString(R.string.nombreDeNuestraString)* y recordad que tenía el **%s** por lo que añadimos una coma y el valor por el que queremos sustituirlo, en este caso el nombre.

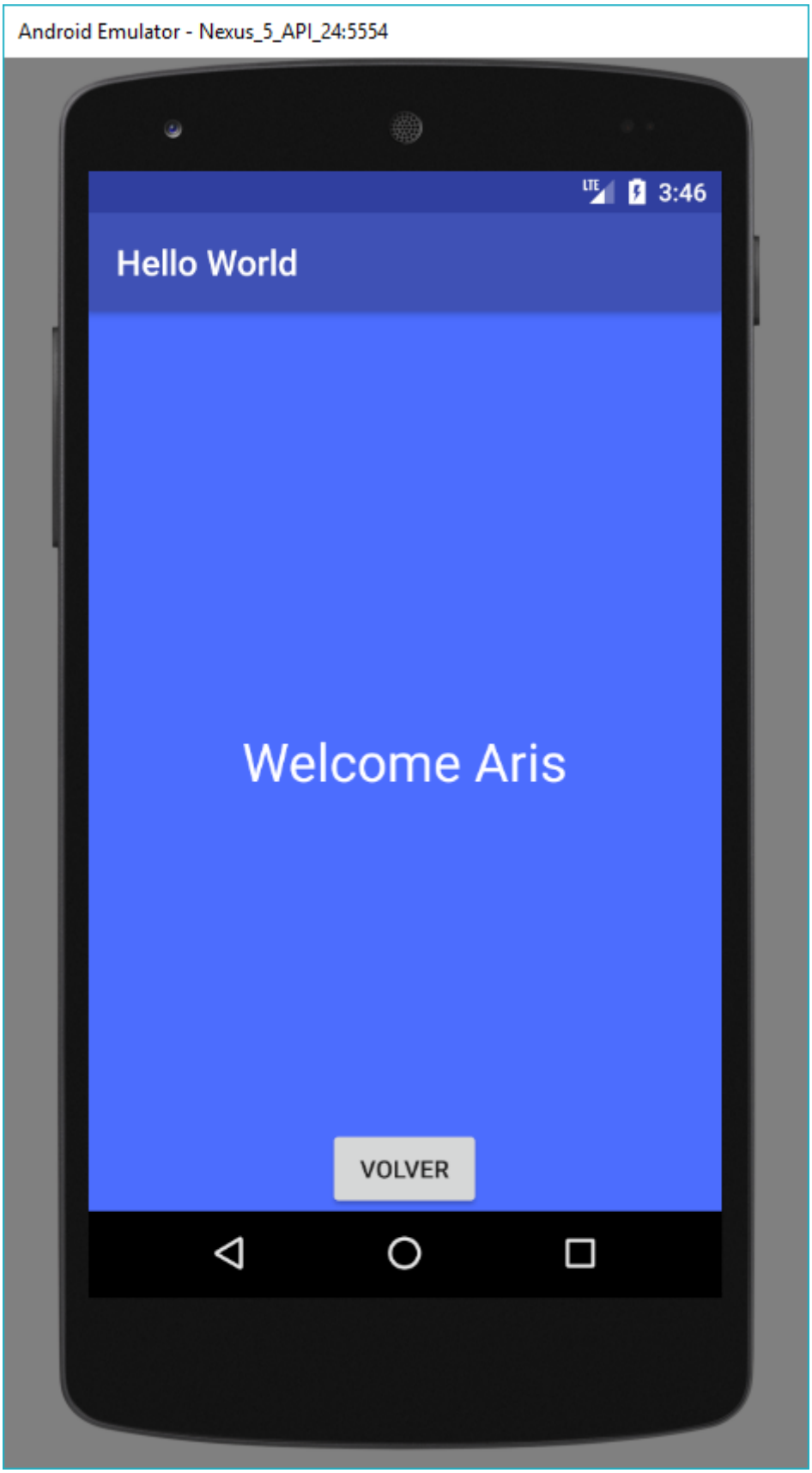
Para finalizar al botón le hemos añadido otro *listener* que ejecutará la función *onBackPresses()* cuando se pulse, que lo que hace es volver a la actividad anterior.

Ahora basta con ir a *run/run 'app'* para lanzar nuestra aplicación. Nos saldrá una imagen similar a esta, donde podremos seleccionar un emulador (trae alguno por defecto, sino lo crearemos) o algún móvil si lo tenemos configurado y conectado.



Seleccionamos el que queremos y ya tendremos nuestra primera aplicación lista.


Android Emulator - Nexus_5_API_24:5554
3:47
Hello World
¿Cómo te llamas?
Name
El nombre no puede estar vacío
ENVIAR


Android Emulator - Nexus_5_API_24:5554
3:46
Hello World
Welcome Aris
VOLVER

Data Classes

Una data class es una clase que contiene solamente atributos que quedemos guardar, por ejemplo datos de un superhéroe. Con esto conseguimos por ejemplo no tener varias variables "nombre1", "nombre2" para almacenar todos los superhéroes con los que vamos a trabajar.

Generando nuestra Data Class

Vamos a crear un proyecto nuevo llamado "**SuperHeroList**" y dentro crearemos una nueva clase de kotlin (No una activity) llamada **Superhero**.

```
1 data class Superhero(
2         var superhero:String,
3         var publisher:String,
4         var realName:String,
5         var photo:String
6 )
```

A simple vista podemos ver varias cosas, la primera que a diferencia de una clase normal esta no lleva llaves y la clase va precedida por "data". Esto le dice a Kotlin que esta clase va a tener por defecto una serie de funciones (aunque no vamos a tener que generarlas nosotros, lo hará el propio lenguaje por detrás) que podremos usar para rellenar dichos campos. La idea es tener un objeto *Superhero* por cada superhéroe que tengamos. Un objeto es simplemente la instancia de una clase.

Creando objetos *Superhero*

Aunque la clase parezca muy sencilla, por detrás nos ha generado funciones para poder recuperar el valor de cada uno de los atributos, sustituirlos, compararlos...

Lo primero que haremos será crear nuestro primer objeto. Para generarlo usaremos el constructor por defecto que trae la clase. Un constructor es una función que se llama automáticamente cuando instanciamos el objeto, y su única función es asignar un valor a cada uno de los parámetros.

```
1 val batman:Superhero = Superhero("Batman", "DC", "Bruce Wayne", "https://
```

Hemos creado un objeto de la clase *Superhero* llamado batman. Para instanciarlo, tenemos que pasar entre paréntesis cada uno de los valores para cada atributo.

Trabajando con Batman

Ahora que tenemos el objeto batman, podemos acceder a cualquiera de sus atributos para recuperar el valor o cambiarlo.

```
1 val batman:Superhero = Superhero("Batman", "DC", "Bruce Wayne", "https://
2 val batmanRealName = batman.realName //Recuperamos "Bruce Wayne"
3 batman.realName = "Soy Batman" //Cambiamos "Brece Wayne" por "Soy Batman"
4 batman.toString() //Devolverá todos los atributos con su valor
```

Objetos inmutables

Supongamos que no queremos que esos objetos puedan ser cambiados, es información verídica que no requiere de cambio alguno. Para ello debemos ir a la data class y cambiar **var** por **val**.

```
1 data class Superhero(
2         val superhero:String,
3         val publisher:String,
4         val realName:String,
5         var photo:String
6 )
```

Ahora una vez creemos el objeto solo se podrá cambiar la foto. Si pasáramos todos los campos a val tendríamos un objeto inmutable por lo que no podríamos usar por ejemplo el *batman.realName = "Nuevo nombre"*.

Si quisiéramos modificar a batman podríamos usar la función *Copy()* para crear un nuevo objeto con algún atributo distinto.

```
1 val batman:Superhero = Superhero("Batman", "DC", "Bruce Wayne", "https://
2 val superBatman:Superhero = batman.copy(superhero = "SuperBatman")
```

Ahora si hacemos un *toString()* veremos que todos los campos son iguales excepto el atributo **superhero**.

Recuperación de parámetros

Por defecto, en cada data class, kotlin nos genera un *componentN()* para cada uno de los parámetros.

```
1         batman.component1() //Batman
2         batman.component2() //DC
3         batman.component3() //Bruce Wayne
4         batman.component4() //https://cursokotlin.com/wp-content/uploads/
```

También podemos almacenar todos los parámetros de un objeto de golpe.

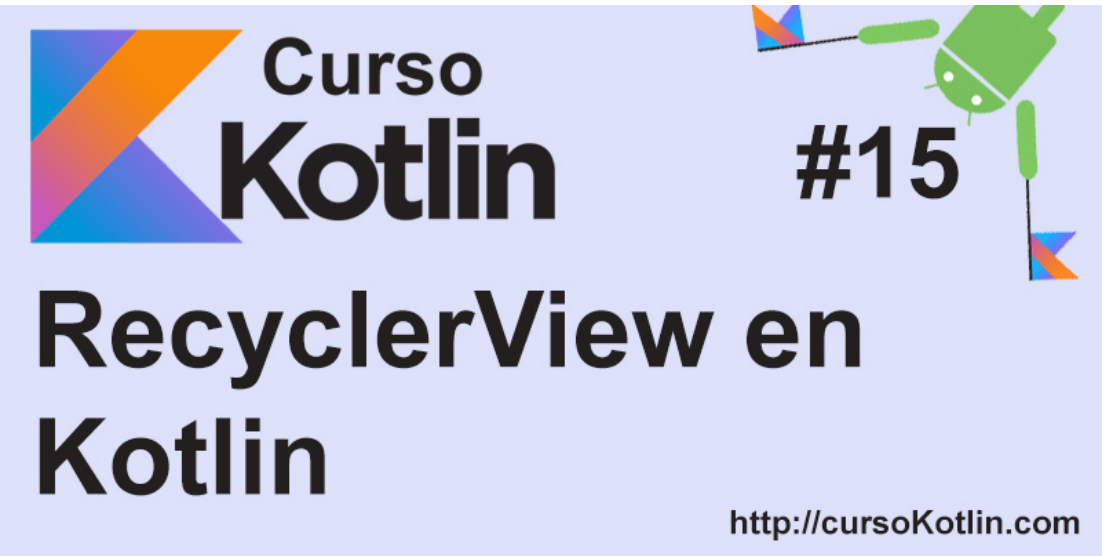
```
1  val (superheroFav, publisherFav, realNameFav, photoFav) = batman
2         superheroFav //Batman
3         publisherFav //DC
4         realNameFav //Bruce Wayne
5         photoFav //https://cursokotlin.com/wp-content/uploads/2017/07/bat
```

Lista de superhéroes

Para acabar vamos a generar una lista de superheroes, que la usaremos en el siguiente capítulo para nuestra primera lista.

```
var superheros:MutableList<Superhero> = mutableListOf()
        superheros.add(Superhero("Spiderman", "Marvel", "Peter Parker", "
        superheros.add(Superhero("Daredevil", "Marvel", "Matthew Michael
        superheros.add(Superhero("Wolverine", "Marvel", "James Howlett",
        superheros.add(Superhero("Batman", "DC", "Bruce Wayne", "https://
        superheros.add(Superhero("Thor", "Marvel", "Thor Odinson", "https
        superheros.add(Superhero("Flash", "DC", "Jay Garrick", "https://c
        superheros.add(Superhero("Green Lantern", "DC", "Alan Scott", "ht
        superheros.add(Superhero("Wonder Woman", "DC", "Princess Diana",
```

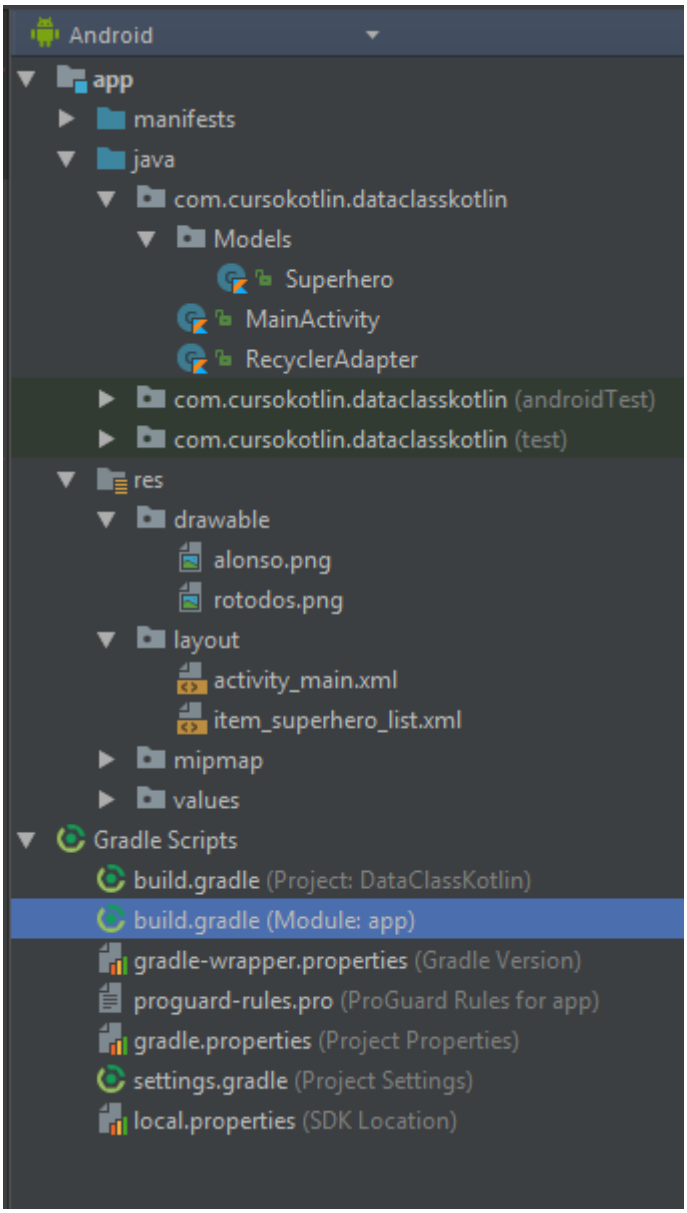
Con esto tenemos una lista de superhéroes, en la que podríamos filtrar por ejemplo por compañía o ir añadiendo más en el futuro.



Llegamos a algo imprescindible en el desarrollo de aplicaciones móviles, las listas. Antaño se usaban ListView en android, pero con el tiempo Google sacó las RecyclerView, listas más customizables, potentes y eficientes.

Configurando nuestro proyecto

Lo primero que haremos será crear un nuevo proyecto o podemos usar el del capítulo anterior, pues reutilizaremos código de ahí.



Una vez creado iremos al *build.gradle* del módulo app e implementaremos una dependencia. Una dependencia es básicamente código externo del proyecto el cual podemos implementar para aprovecharlo, en este caso implementaremos el Recyclerview que por defecto no lo trae.

Así que vamos a la parte de las *dependencies{}* y añadimos

```
1     implementation 'com.android.support:recyclerview-v7:25.0.0'
```

Y también añadiremos una librería llamada **Picasso** que la usaremos para la gestión de imágenes.

```
1     implementation 'com.squareup.picasso:picasso:2.5.2'
```

Nos aparecerá en la parte superior una barra que nos dirá que es necesario sincronizar el gradle, así que le damos. Si todo sale bien ya tendremos esta parte configurada, si te da error de sdk posiblemente es que tengas varias versiones diferentes (CompileSDKVersion, buildToolsVersion...) todas a 25. Debería quedar así.

```
apply plugin: 'com.android.application'

apply plugin: 'kotlin-android'

apply plugin: 'kotlin-android-extensions'

android {
    compileSdkVersion 25
    buildToolsVersion "25.0.0"
    defaultConfig {
        applicationId "com.cursokotlin.dataclasskotlin" //Nombre de nues
        minSdkVersion 15
        targetSdkVersion 25
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJU
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt')
        }
    }
}

dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    androidTestImplementation ('com.android.support.test.espresso:espres
        exclude group: 'com.android.support', module: 'support-annotatio
    })
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jre7:$kotlin_vers
    implementation 'com.android.support:appcompat-v7:25.4.0'
    testImplementation 'junit:junit:4.12'
    implementation 'com.android.support.constraint:constraint-layout:1.0
    implementation 'com.android.support:recyclerview-v7:25.0.0'
    implementation 'com.squareup.picasso:picasso:2.5.2'
}
```

Editando nuestra layout y creando una celda

El layout principal será el contenedor del RecyclerView, pero luego para inflarlo tendremos que crear un adapter (hablaremos de él un poco más abajo) y una celda que será la que muestre cada una de las filas de la lista.

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/androi
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.cursokotlin.dataclasskotlin.MainActivity">

    <android.support.v7.widget.RecyclerView
        android:id="@+id/rvSuperheroList"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

</RelativeLayout>
```

En este ejemplo seguiremos con los objetos superhéroe del capítulo anterior así que la celda que hagamos llevará un **ImageView** para el logo del superhéroe y tres textos, uno para el nombre real, el nombre de superhéroe y el de la compañía (DC o Marvel). Vamos a *res/layout* y

creamos un archivo llamado **item_superhero_list.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/androi
    xmlns:tools="http://schemas.android.com/tools"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="10dp">

    <ImageView
        android:id="@+id/ivAvatar"
        android:layout_width="100dp"
        android:layout_height="100dp"
        android:layout_marginEnd="10dp"
        android:layout_marginRight="10dp"
        android:scaleType="centerCrop" />

    <TextView
        android:id="@+id/tvSuperhero"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textStyle="bold"
        android:textSize="20sp"
        android:layout_alignParentTop="true"
        android:layout_toRightOf="@+id/ivAvatar"
        android:layout_toEndOf="@+id/ivAvatar"
        tools:text= "dadawdawdwd" />

    <TextView
        android:id="@+id/tvRealName"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="15sp"
        android:layout_below="@+id/tvSuperhero"
        android:layout_toRightOf="@+id/ivAvatar"
        android:layout_toEndOf="@+id/ivAvatar"
        tools:text = "wdawd"/>

    <TextView
        android:id="@+id/tvPublisher"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textStyle="italic"
        android:layout_alignParentBottom="true"
        android:layout_alignParentRight="true"
        android:layout_alignParentEnd="true"
        tools:text = "DC"/>
</RelativeLayout>
```

Si no me equivoco todos los atributos de arriba ya los hemos visto excepto los “tools:text” la función tools lo que hace es ayudarnos a ver algo de un modo más sencillo. Aquí por ejemplo está poniendo un texto para poder ir maquetando mejor la vista, pero cuando compile la aplicación, ese texto no estará, así que podríamos decir que es un molde. Si tenéis alguna otra duda podéis dejar un comentario.

Ahora vamos a crear un modelo superhéroe del cual haremos varias listas para cargar en el recyclerview, como esto ya está hecho en el capítulo anterior solo os voy a dejar la clase aquí para que la copien.

```
data class Superhero(
        var superhero:String,
        var publisher:String,
        var realName:String,
        var photo:String
)
```

Creando nuestros objetos superhéroes

Volvemos a **MainActivity.kt** y vamos a crear dos funciones, una que configurará nuestro recyclerview con el adapter y otra que generará la lista de objetos superhéroes.

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        setUpRecyclerView()
    }

    fun setUpRecyclerView(){
        ...
    }

    fun getSuperheros(): MutableList<Superhero>{
        var superheros:MutableList<Superhero> = ArrayList()
        superheros.add(Superhero("Spiderman", "Marvel", "Peter Parker",
        superheros.add(Superhero("Daredevil", "Marvel", "Matthew Michael
        superheros.add(Superhero("Wolverine", "Marvel", "James Howlett",
        superheros.add(Superhero("Batman", "DC", "Bruce Wayne", "https:/
        superheros.add(Superhero("Thor", "Marvel", "Thor Odinson", "http
        superheros.add(Superhero("Flash", "DC", "Jay Garrick", "https://
        superheros.add(Superhero("Green Lantern", "DC", "Alan Scott", "h
        superheros.add(Superhero("Wonder Woman", "DC", "Princess Diana",
        return superheros
    }
}
```

Aunque la clase no está terminada, quería que vieran el modelo de superhéroe creado, para que ahora entiendan mejor que vamos a hacer en el adapter.

Creando el adapter

Ahora vamos a crear el adapter del Recyclerview. Un adapter es la clase que hace de puente entre la vista (el recyclerview) y los datos. Así que creamos una nueva clase llamada **RecyclerAdapter.kt**

La clase RecyclerAdapter se encargará de recorrer la lista de superhéroes que le pasaremos más tarde, y llamando a otra clase interna que tendrá, este rellenará los campos.

Vamos a ir viendo método por método, así que hasta el final de la explicación la clase tendrá fallos.

```
class RecyclerAdapter : RecyclerView.Adapter<RecyclerAdapter.ViewHolder>(

    var superheros: MutableList<Superhero>  = ArrayList()
    lateinit var context:Context

    ...
}
```

Lo primero que hacemos es decirle a la clase que tendrá un **RecyclerAdapter** y un **ViewHolder**. Después creamos las dos variables que necesitamos para el adapter, una lista de superhéroes (que rellenaremos en la clase principal y se la mandaremos aquí y el contexto de la mainAcitivity para poder trabajar con la librería **Picasso**.

```
fun RecyclerAdapter(superheros : MutableList<Superhero>, context: Conte
        this.superheros = superheros
        this.context = context
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        val item = superheros.get(position)
        holder.bind(item, context)
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): V
        val layoutInflater = LayoutInflater.from(parent.context)
        return ViewHolder(layoutInflater.inflate(R.layout.item_superhero
    }

    override fun getItemCount(): Int {
        return superheros.size
    }
```

Lo primero es un simple constructor (tiene el mismo nombre que la clase y lo único que hará será recibir la lista y el contexto que le pasamos desde la clase principal).

Los siguientes tres métodos tienen delante la palabra reservada **override**, esto es porque son métodos obligatorios que se implementan de la clase RecyclerView. *onBindViewHolder()* se encarga de coger cada una de las posiciones de la lista de superhéroes y pasarlas a la clase ViewHolder(todavía no está hecha) para que esta pinte todos los valores.

Después tenemos *onCreateViewHolder()* que como su nombre indica lo que hará será devolvernos un objeto ViewHolder al cual le pasamos la celda que hemos creado.

Y para finalizar el método *getItemCount()* nos devuelve el tamaño de la lista, que lo necesita el RecyclerView.

Para finalizar el adapter queda hacer la clase ViewHolder de la que tanto hemos hablado. No es necesaria hacerla dentro del adapter, pero como van tan ligadas la una a la otra creo que es lo mejor.

```
class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {
        val superheroName = view.findViewById(R.id.tvSuperhero) as TextV
        val realName = view.findViewById(R.id.tvRealName) as TextView
        val publisher = view.findViewById(R.id.tvPublisher) as TextView
        val avatar = view.findViewById(R.id.ivAvatar) as ImageView

        fun bind(superhero:Superhero, context: Context){
            superheroName.text = superhero.superhero
            realName.text = superhero.realName
            publisher.text = superhero.publisher
            itemView.setOnClickListener(View.OnClickListener { Toast.mak
            avatar.loadUrl(superhero.photo)
        }
        fun ImageView.loadUrl(url: String) {
            Picasso.with(context).load(url).into(this)
        }
    }
```

Lo primero que hace esta clase es hacer referencia a los items de la celda, el *view.findViewByID()* busca los items a través de la id que le ponemos, y luego añadimos el **as X** donde X es el tipo del componente (ImageView, TextView...).

Dentro tiene el método *bind(superhero:Superhero, context: Context)* que dicho método lo llamamos desde el *onBindViewHolder()* para que rellene los datos. Después de añadir todos los textos que queremos, hacemos 2 cositas más. La primera es que llamamos a **itemView** que es la vista (celda) que estamos rellenando y le ponemos un **setOnClickListener** que pintará en pantalla el nombre del superhéroe en el que hemos hecho click. Y para finalizar seleccionamos el ImageView y con el método que hemos creado le pasamos la URL de la imagen. Esta url la usará Picasso para descargarla de internet y pintarla en nuestra lista. Pero para que Picasso pueda hacer eso, debemos ponerle permiso de internet a nuestra app.

Vamos a AndroidManifest.xml y añadimos el permiso después de abrir la etiqueta manifest.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.cursokotlin.dataclasskotlin">

    <uses-permission android:name="android.permission.INTERNET" /> //Ped

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Y nuestro adapter completo sería así.

```
class RecyclerAdapter : RecyclerView.Adapter<RecyclerAdapter.ViewHolder>

    var superheros: MutableList<Superhero>  = ArrayList()
    lateinit var context:Context

    fun RecyclerAdapter(superheros : MutableList<Superhero>, context: Co
        this.superheros = superheros
        this.context = context
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        val item = superheros.get(position)
        holder.bind(item, context)
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): V
        val layoutInflater = LayoutInflater.from(parent.context)
        return ViewHolder(layoutInflater.inflate(R.layout.item_superhero
    }

    override fun getItemCount(): Int {
        return superheros.size
    }

    class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {
        val superheroName = view.findViewById(R.id.tvSuperhero) as TextV
        val realName = view.findViewById(R.id.tvRealName) as TextView
        val publisher = view.findViewById(R.id.tvPublisher) as TextView
        val avatar = view.findViewById(R.id.ivAvatar) as ImageView

        fun bind(superhero:Superhero, context: Context){
            superheroName.text = superhero.superhero
            realName.text = superhero.realName
            publisher.text = superhero.publisher
            avatar.loadUrl(superhero.photo)
            itemView.setOnClickListener(View.OnClickListener { Toast.mak
        }
        fun ImageView.loadUrl(url: String) {
            Picasso.with(context).load(url).into(this)
        }
    }
}
```

Ahora volvemos al **MainActivity.kt** para finalizar el método que nos queda. Primero crearemos dos variables, la del RecyclerView y la del adapter que acabamos de terminar.

```
    lateinit var mRecyclerView : RecyclerView
    val mAdapter : RecyclerAdapter = RecyclerAdapter()
```

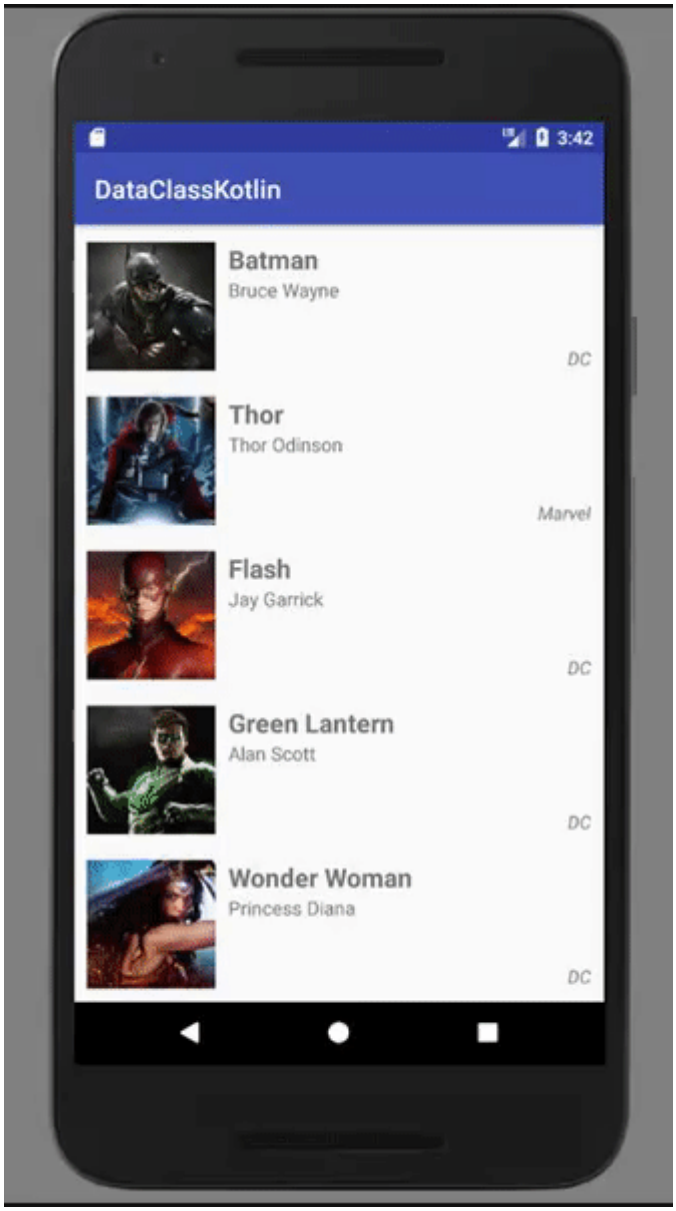
El recyclerview lo tenemos que instanciar en el método *setUpRecyclerView()* por lo que tenemos que ponerle la propiedad lateinit a la variable, indicándole a Kotlin que la instanciaremos más tarde.

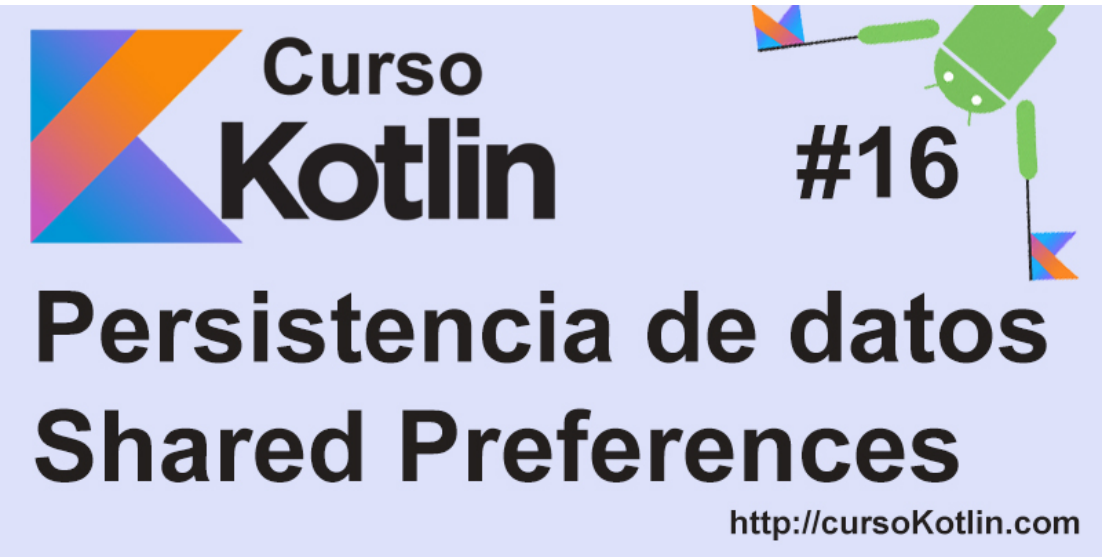
```
    fun setUpRecyclerView(){
        mRecyclerView = findViewById(R.id.rvSuperheroList) as RecyclerVie
        mRecyclerView.setHasFixedSize(true)
        mRecyclerView.layoutManager = LinearLayoutManager(this)
        mAdapter.RecyclerAdapter(getSuperheros(), this)
        mRecyclerView.adapter = mAdapter
    }
```

Para finalizar creamos el método, lo primero que hará será buscar en el layout el Recycler y asignarselo a la variable que creamos. El siguiente método se usa para evitar problemas cuando el padre que contiene la lista cambia y puede estropear el recycler.

Luego llamamos a nuestro adapter y le pasamos los campos que necesite, en este caso getSuperheros (que devuelve una lista de superhéroes) y this (el contexto). Y para finalizar se lo asignamos a el recyclerview.

Ahora solo necesitamos compilar para ver el resultado de nuestra aplicación.





Un requisito básico para la mayoría de aplicaciones móviles a nivel profesional suele ser la persistencia de datos, con ella conseguimos no solo persistir información del usuario, sino que podemos almacenar contenido que ayuden a un funcionamiento mas óptimo, por ejemplo cacheando contenido que recuperamos de internet para no repetir las consultas cada vez que pasamos por dicha activity.

Las tres formas de almacenar información son a través de las Shared Preferences, ficheros o bases de datos. Como norma general no se suelen usar ficheros así que los descartaremos por ahora y en este capítulo veremos las **Shared Preferences**.

¿Que son las Shared Preferences?

Las Shared están destinadas a almacenar pequeñas cantidades de información a través de clave-valor. Debe ser información no comprometida puesto que Android generará un fichero XML donde almacenará toda esta información sin cifrar. Estos ficheros pueden ser compartidos o privados.

En este proyecto vamos a crear una aplicación muy sencilla en la cual si no hay ningún nombre guardado nos pedirá que lo pongamos, si por el contrario ya disponemos de uno nos saldrá en pantalla junto a un botón de borrar. Otra parte importante es que haremos nuestro primer **patrón de diseño**, el Singleton.

Lo primero que haremos será crear un nuevo proyecto y haremos una maquetación muy simple del layout.

```xml
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schem
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.cursokotlin.sharedpreferences.MainActivity">

    <TextView
        android:id="@+id/tvName"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <EditText
        android:id="@+id/etName"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:hint="Escribe tu nombre"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <Button
        android:id="@+id/btnDeleteValue"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Borrar"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"/>

    <Button
        android:id="@+id/btnSaveValue"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Guardar"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"/>

</android.support.constraint.ConstraintLayout>
```

Veréis que hay componentes que se sobreponen unos sobre otros, esto es porque así de paso os voy a enseñar los posibles casos que puede tener una vista.

Lo siguiente que haremos será crear una clase donde definiremos todo lo necesario para trabajar con shared, la llamaremos **Prefs**. Esta clase recibirá un contexto, en este caso el de la aplicación, para poder instanciarlo una sola vez al iniciar la aplicación y tener un objeto pref.

```
class Prefs (context: Context) {
    val PREFS_NAME = "com.cursokotlin.sharedpreferences"
    val SHARED_NAME = "shared_name"
    val prefs: SharedPreferences = context.getSharedPreferences(PREFS_NAM

    var name: String
        get() = prefs.getString(SHARED_NAME, "")
        set(value) = prefs.edit().putString(SHARED_NAME, value).apply()
}
```

Lo primero que hemos hecho ha sido definir dos constantes, **PREFS_NAME** y **SHARED_NAME** la primera será la clave del objeto pref que crearemos más adelante y la segunda la clave del nombre que almacenaremos. Recordad que las shared preferences se almacenan con clave-valor, lo que significa que para pedir el valor de "*name*" necesitamos pedirlo a través de la clave **SHARED_NAME**.

Observad también que hemos definido una variable name que será donde almacenemos el nombre como dije antes, pero que hemos sobreescrito el método get y set, así que cuando pidamos el valor de *name,* este accederá a el objeto prefs y pedirá dicho valor que corresponde la clave **SHARED_NAME**. Lo mismo con el set, que a través de *prefs.edit().putString(SHARED_NAME, value).apply()* almacenará el valor que le digamos. Obviamente si fuera otro tipo de variable, por ejemplo un **Int**, cambiaríamos el *putString()* por un *putInt()* y así con cada tipo de variable.

Ahora vamos a crear una clase algo diferente. Esta clase va a extender de *Application()* y eso significa que será lo primero en ejecutarse al abrirse la aplicación.

Para extender de dicha clase en Kotlin es muy sencillo, simplemente debemos crear una clase como siempre y a continuación del nombre después de dos puntos, pondremos la clase en cuestión.

```
1  class SharedApp : Application() {
2      companion object {
3          lateinit var prefs: Prefs
4      }
5
6      override fun onCreate() {
7          super.onCreate()
8          prefs = Prefs(applicationContext)
9      }
10 }
```

Aunque la clase sea bastante pequeña vamos a comentar unas cosas. Para empezar hemos declarado un companion object que será el objeto de nuestra clase Prefs que usaremos en todo el proyecto, así que para que lo entendáis el companion object es una forma de tener un objeto disponible en todo el proyecto (un objeto estático para los que conozcan java). Y delante lleva un **lateinit** que quiere decir que será instanciado más tarde, en este ejemplo en el método onCreate() de esta clase.

Recordad que para que esta clase se lance al abrir la app debemos ir al **AndroidManifest.xml** y añadir *android:name=".SharedApp "* dentro de la etiqueta *<Application>*

Ahora iremos a nuestro **MainActivity** a desarrollar un poco de lógica que falta.

La idea es crear dos métodos, uno mostrará una vista para invitados y el otro la vista del perfil, la diferencia entre ellos será que si eres invitada te mostrará un EditText y un botón de guardar y por el contrario si ya hay un nombre guardado en persistencia de datos pues te saludará y tendrá un botón para borrar dicho campo de memoria.

```
    fun showProfile(){
          tvName.visibility = View.VISIBLE
          tvName.text = "Hola ${SharedApp.prefs.name}"
          btnDeleteValue.visibility = View.VISIBLE
          etName.visibility = View.INVISIBLE
          btnSaveValue.visibility = View.INVISIBLE
      }

      fun showGuest(){
          tvName.visibility = View.INVISIBLE
          btnDeleteValue.visibility = View.INVISIBLE
          etName.visibility = View.VISIBLE
          btnSaveValue.visibility = View.VISIBLE
      }
```

Como ya había dicho al comienzo del post, os quiero hablar sobre la visibilidad. Un componente puede estar en tres estados visible, invisible y gone.

- **Visible**: El componente se ve en la pantalla, por defecto viene esta opción activada.
- **Invisible**: El componente no se ve pero sigue estando en la pantalla, por lo que se puede seguir trabajando con él, por ejemplo poner algo a la derecha de un componente invisible.
- **Gone**: El componente NO está en la pantalla por lo que no hay interacción posible.

Fijaros que el método showProfile asigna un valor a tvName, lo que significa que está accediendo a las shared preferences. Para ello simplemente llamamos a la clase SharedApp (la que contiene el companion object), el objeto del cual estamos hablando y el atributo name que como vimos al principio hemos modificado para que cuando hagamos un get (sacar la información que almacena) le pida nuestras shared preference el valor de **SHARED_NAME**.

Ahora necesitamos un método que compruebe si hay información en *name* y así comprobar si el usuario ha guardado su nombre.

```
fun configView(){
        if(isSavedName()) showProfile() else showGuest()
    }

    fun isSavedName():Boolean{
        val myName = SharedApp.prefs.name
        return myName != EMPTY_VALUE
    }
```

Para finalizar solo debemos guardar o borrar la información del usuario cada vez que pulsemos el botón correspondiente y acto seguido volver a llamar a *configView()* para que muestre la vista oportuna.

```
        btnSaveValue.setOnClickListener {
            SharedApp.prefs.name = etName.text.toString()
            configView() }
        btnDeleteValue.setOnClickListener {
            SharedApp.prefs.name = EMPTY_VALUE
            configView()}
```

La clase completa quedaría así.

```

```

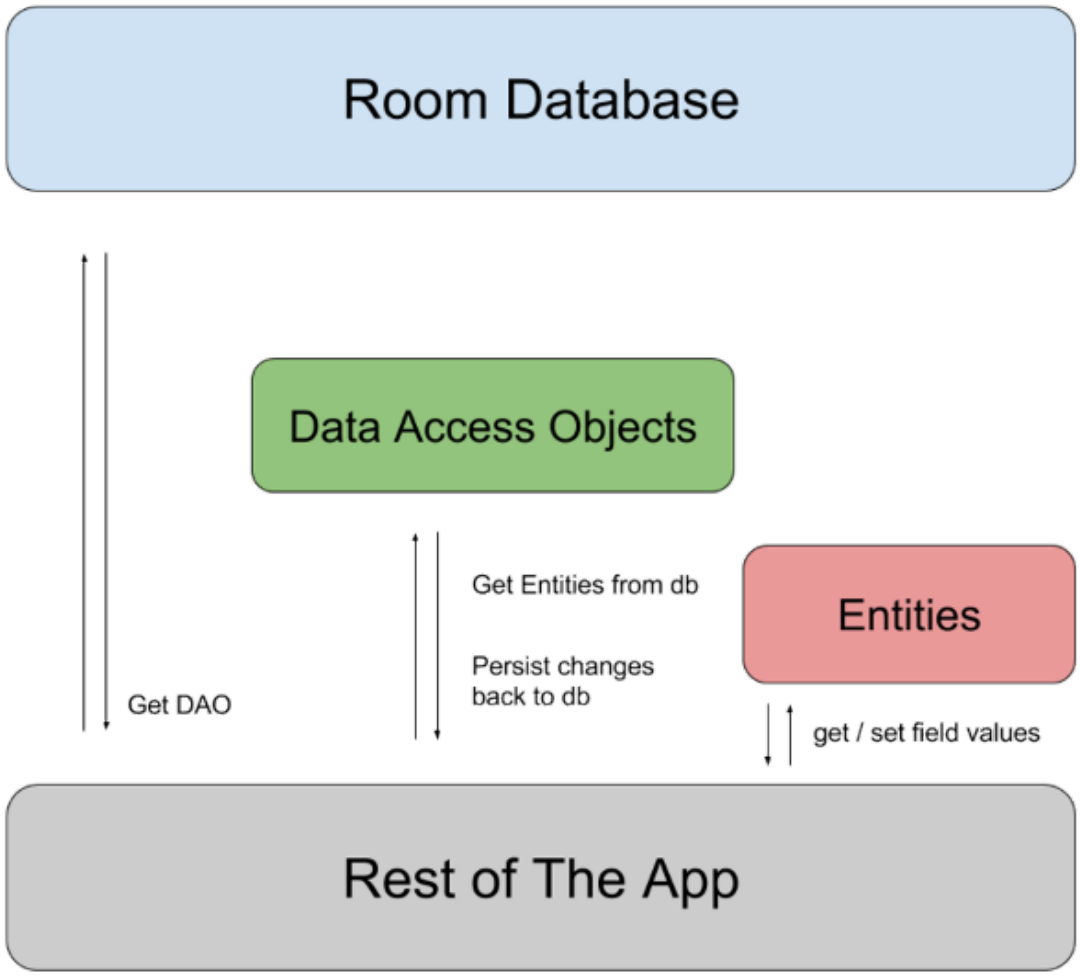
```kotlin
class MainActivity : AppCompatActivity() {

    val EMPTY_VALUE = ""

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        configView()

        btnSaveValue.setOnClickListener {
            SharedApp.prefs.name = etName.text.toString()
            configView() }
        btnDeleteValue.setOnClickListener {
            SharedApp.prefs.name = EMPTY_VALUE
            configView()}
    }


    fun showProfile(){
        tvName.visibility = View.VISIBLE
        tvName.text = "Hola ${SharedApp.prefs.name}"
        btnDeleteValue.visibility = View.VISIBLE
        etName.visibility = View.INVISIBLE
        btnSaveValue.visibility = View.INVISIBLE
    }

    fun showGuest(){
        tvName.visibility = View.INVISIBLE
        btnDeleteValue.visibility = View.INVISIBLE
        etName.visibility = View.VISIBLE
        btnSaveValue.visibility = View.VISIBLE
    }

    fun configView(){
        if(isSavedName()) showProfile() else showGuest()
    }

    fun isSavedName():Boolean{
        val myName = SharedApp.prefs.name
        return myName != EMPTY_VALUE
    }
}
```

Con esto ya tendríamos lista nuestra primera app con persistencia de datos. Este ha sido un ejemplo muy básico pero prefiero ir poco a poco con este tema pues puede ser muy abundante si no vamos poco a poco.

En el capítulo pasado, empezamos a conocer la persistencia de datos y su utilidad, el problema es que como ya comenté las **shared preferences** están destinadas a persistir pequeños datos de preferencia del usuario, por lo que si necesitamos guardar más información necesitamos una base de datos. Aunque hay disponible muchísimas opciones muy completas, en este curso trabajaremos siempre con Room, puesto que es la alternativa que nos propone google.

¿Qué es Room?

Room es un ORM (*Object-Relational mapping*) que nos permitirá trabajar de una forma más sencilla con bases de datos SQL.



La imagen anterior nos muestra el funcionamiento de dicha herramienta que, aunque parezca complicado al principio, es muy fácil de entender cuando nos pongamos a ello.

La idea es clara, tendremos una base de datos que le devolverá a nuestra app los **Data Access Objects** (DAO) estos son los encargados de persistir la información en la base de datos y de devolvernos las **entities**, que serán las encargadas de devolvernos la información que hemos ido almacenando.

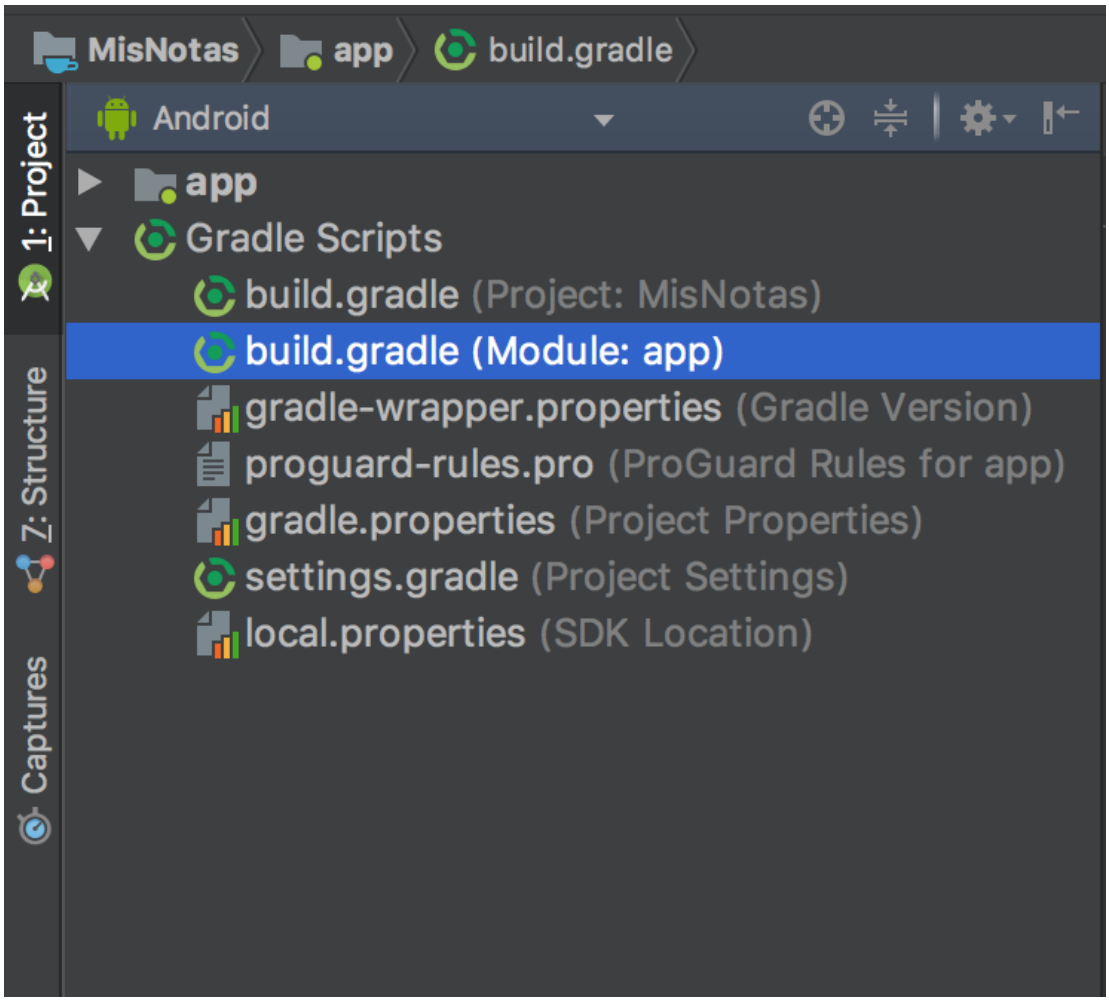
Una vez entendamos un poco como funciona es el momento de ponerse manos a la obra. Lo primero que haremos será crear nuestro primer proyecto como siempre.

MisNotas

Así se llamará nuestra app y consistirá en una lista de tareas las cuales podemos marcar como hechas o no y tendremos la posibilidad de borrarlas. Al final del artículo lo tendréis disponible para descargar a través de GitHub (ya haré un artículo sobre git/gitflow y demás).

Lo primero que haremos será añadir todas las dependencias necesarias. Como ya he comentado anteriormente las dependencias son pequeñas llamadas que hace nuestro fichero Gradle para implementar funciones que por defecto nuestro proyecto no tiene. Por ejemplo, ahora necesitamos usar Room y aunque sea oficial de Google no viene por defecto en nuestro proyecto y es muy fácil de entender. Si vinieran todas las dependencias ya implementadas al abrir un proyecto, nuestra app pesaría muchísimo.

Así que vamos a ir al gradle del módulo app (por defecto tendremos un gradle por módulo y por aplicación, de eso ya hablaré en otros artículos). Para localizarlo, en tan sencillo como ir a **Gradle Scripts** si tenemos la vista "android".



Y meteremos todas las dependencias que necesitamos. En este caso vamos a meter cuatro más, ya luego a medida que vayamos a usarlas las iré explicando.

```
1     implementation 'com.android.support:design:26.1.0'
2     implementation 'com.android.support:recyclerview-v7:26.1.0'
3     implementation 'org.jetbrains.anko:anko-common:0.9'
4     implementation "android.arch.persistence.room:runtime:1.0.0-rc1"
5     kapt "android.arch.persistence.room:compiler:1.0.0-rc1"
```

Las meteremos donde están las demás, dentro de *dependencies {}*. Sincronizamos el gradle.

Creando nuestra base de datos

Vamos a crear un directorio nuevo llamado database dentro de nuestro proyecto y ahí crearemos todo lo necesario siguiente el esquema que vimos antes. Necesitaremos crear 3 clases, empezaremos creando nuestra entidad que será el modelo a persistir en la base de datos.

TaskEntity

La aplicación va a ser una lista de tareas, así que el modelo se llamará **TaskEntity** y contendrá 3 valores, una ID para localizar el objeto, un nombre (el de la tarea a realizar) y un Booleano que será el que usemos para saber si la tarea está hecha o no. Así que creamos nuestra clase y la

dejamos así.

```
@Entity(tableName = "task_entity")
data class TaskEntity (
        @PrimaryKey(autoGenerate = true)
        var id:Int = 0,
        var name:String = "",
        var isDone:Boolean = false
)
```

Aunque es bastante pequeñita quiero recalcar algunas cosillas:

- La anotación @Entity la utilizamos para añadirle un nombre a nuestra entidad como tabla de la base de datos. Cada base de datos puede contener una o varias tablas y cada una persiste un modelo diferente.
- La anotación @PrimaryKey (autoGenerate = true) está diciendo que la variable id es un valor que se autogenera al crear un objeto de esta clase y que no podrá repetirse. Es un valor único con el cual podremos localizar un objeto concreto.

TaskDao

TaskDao será la interfaz que contendrá las consultas a la base de datos. Aquí distinguiremos cuatro tipos de consultas.

- **@Query**: Se hacen consultas directamente a la base de datos, usaremos SQL para hacerlas. En este ejemplo haremos dos muy sencillitas, pero se pueden hacer cosas impresionantes.
- **@Insert**: Se usará para insertar entidades a la base de datos, a diferencia de las **@Query** no hay que hacer ningún tipo de consulta, sino pasar el objeto a insertar.
- **@Update**: Actualizan una entidad ya insertada. Solo tendremos que pasar ese objeto modificado y ya se encarga de actualizarlo. ¿Cómo sabe que objeto hay que modificar? Pues por nuestro id (recordad que es la **PrimaryKey**).
- **@Delete:** Como su propio nombre indica borra de la tabla un objeto que le pasemos.

```
@Dao
interface TaskDao {
    @Query("SELECT * FROM task_entity")
    fun getAllTasks(): MutableList<TaskEntity>
}
```

Por ahora solo meteremos una query, que lo que hará será seleccionar todas las **TaskEntity** que tengamos en la base de datos. Fijaros que es una interfaz en lugar de una clase, y que contiene la anotación **@Dao**.

TaskDatabase

Una vez tengamos nuestro Dao y nuestra entidad, vamos a crear la base de datos que los contendrá, en este caso se llamará **TaskDatabase**.

```
@Database(entities = arrayOf(TaskEntity::class), version = 1)
abstract class TasksDatabase : RoomDatabase() {
    abstract fun taskDao(): TaskDao
}
```

Lo primero que debemos fijarnos es en la anotación @Database, que especifica que la entidad será una lista de **TaskEntity** (entidad que ya hemos creado) y que la versión es 1. Las versiones se usan para la posible migración de datos al actualizar la App. Imaginemos que sacamos una segunda versión de la app y en vez de 3 parámetros almacenamos 4, no podemos cambiar nuestra entidad de golpe pues habría problemas. Para eso se usa la versión, junto a un fichero de migración que le dice al programa que deberá hacer para pasar de la versión 1 a la 2, 3 o la que sea.

También debemos fijarnos en que nuestra clase extienda de **RoomDatabase()** que es una clase que tenemos gracias a importar la dependencia de Room en nuestro gradle. Para finalizar tiene una sola función que hace referencia al Dao que hemos creado anteriormente, si tuviésemos más Dao's pues habría que implementarlos ahí también.

Con esto ya tenemos nuestra base de datos preparada, ahora debemos instanciarla al inicio de la aplicación. Como hicimos en el capítulo anterior con las **Shared Preferences** crearemos una clase application para poder acceder a nuestra información en cualquier clase.

MisNotasApp

```
1  class MisNotasApp: Application() {
2
3      companion object {
4          lateinit var database: TasksDatabase
5      }
6
7      override fun onCreate() {
8          super.onCreate()
9          MisNotasApp.database =  Room.databaseBuilder(this, TasksDatabase
10     }
11 }
```

Como ya he comentado, esta clase es básicamente igual a la del capítulo anterior por lo que no hay nada que explicar salvo que la instancia de database necesitará tres parámetros, el contexto (this), la clase de nuestra base de datos (TasksDatabase) y el nombre que le pondremos, en este caso la he llamado "trasks-db".

Recordad que para que esta clase se lance al abrir la app debemos ir al **AndroidManifest.xml** y añadir *android:name=".MisNotasApp"* dentro de la etiqueta *<Application>*

activity_main

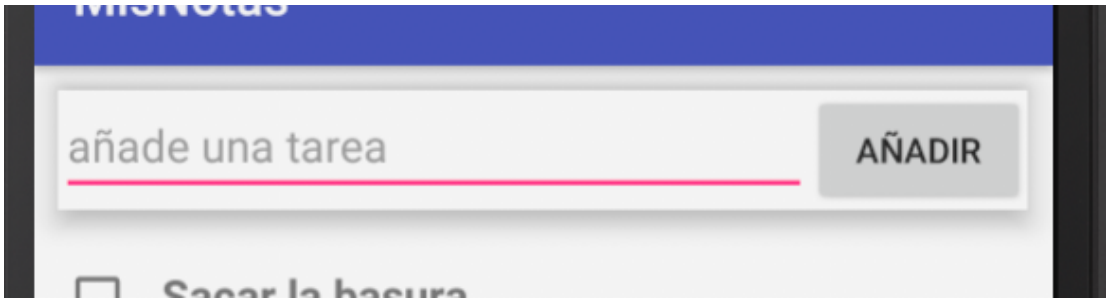
Aunque tengamos varias clases por detrás, nuestra app solo tendrá un layout, una sola vista. La idea era crear algo sencillo y usable, por lo que me decanté por una barra superior donde añadir las tareas y luego un RecyclerView donde se muestren todas. El XML es muy sencillito, solo he usado algún atributo nuevo que comentaré.

```
```

```xml
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/androi
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@android:color/background_light"
    tools:context="com.cursokotlin.misnotas.UI.MainActivity">

    <android.support.v7.widget.RecyclerView
        android:id="@+id/rvTask"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_below="@+id/rlAddTask"/>


    <RelativeLayout
        android:id="@+id/rlAddTask"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:elevation="10dp"
        android:layout_margin="10dp"
        android:background="@android:color/white">

        <EditText
            android:id="@+id/etTask"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:hint="añade una tarea"
            android:layout_alignParentLeft="true"
            android:layout_toLeftOf="@+id/btnAddTask"
            />

        <Button
            android:id="@+id/btnAddTask"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignParentRight="true"
            android:text="Añadir"/>

    </RelativeLayout>

</RelativeLayout>
```

Debajo del RecyclerView he metido la barra superior. La he puesto debajo porque en los XML de android, si imaginamos que va todo por capas, la parte inferior es la capa más visible. Por ejemplo si pusiéramos dos fotos de tamaño completo a la pantalla, se vería la que esté en la parte más abajo de nuestro archivo.

A parte de eso conocemos todos los atributos de la vista excepto *android:elevation* que lo que hace es dar una elevación a nuestro componente añadiéndole una sombra por debajo. El efecto es el siguiente.



MainActivity

Ya tenemos nuestra vista preparada, es el momento de empezar a generar la lógica. Empezamos creando las variables necesarias.

```
    lateinit var recyclerView: RecyclerView
    lateinit var tasks: MutableList<TaskEntity>
```

Ahora nos vamos al OnCreate de nuestra actividad, lo primero que haremos será instanciar *tasks* como una arrayList y acto seguido llamaremos a la función *getTasks()* que vamos a crear. Esta función será la encargada de acceder a nuestro DAO para hacer la primera consulta, recuperar todas las entidades que el usuario tenga guardadas. Así que por ahora dejamos el OnCreate así.

```
override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        tasks = ArrayList()
        getTasks()
}
```

Ahora antes de seguir tengo que hacer un poco de hincapié en los hilos. Los hilos son los que permiten la multitarea de un dispositivo, por ejemplo en un hilo puedes ir guardando información asíncronamente (por detrás) mientras el usuario sigue haciendo cosas. En Android, el **hilo principal es el encargado de la parte** visual de la aplicación, por lo que no nos deja acceder a la base de datos desde ahí, por ello crearemos un hilo secundario que de modo asíncrono hará la petición a la base de datos y recuperará la información que necesitemos.

Como este capítulo no va de hilos, no solo lo he resumido mucho sino que lo voy a hacer de la manera más sencilla (en un futuro haré un artículo dedicado a ello). Para este fin vamos a usar **Anko**, una librería muy completa para Kotlin que nos hará más sencillas muchas tareas del día a día. **Anko** ya lo implementamos al principio del capítulo en las dependencias.

```
  fun getTasks() {
        doAsync {
            tasks = MisNotasApp.database.taskDao().getAllTasks()
            uiThread {
                setUpRecyclerView(tasks)
            }
        }
    }
```

Gracias a Anko la función queda muy sencilla. Todo lo que tengamos que hacer en el segundo hilo asíncrono, lo meteremos dentro de *doAsync{}* y una vez haya acabado, usando *uiThead{}* podemos decir que haga algo en el hilo principal, el de la vista. Nosotros hemos asignado a *tasks* los valores que recuperamos de nuestra base de datos, y cuando los completa, llamamos al método *setUpRecyclerView(tasks)* que será el que configure nuestro RecyclerView.

Pero antes de mostraros el método anterior vamos a crear nuestro adapter, que a diferencia del primero que vimos hace unos capítulos, este tendrá diferentes eventos para capturar los clicks del usuario.

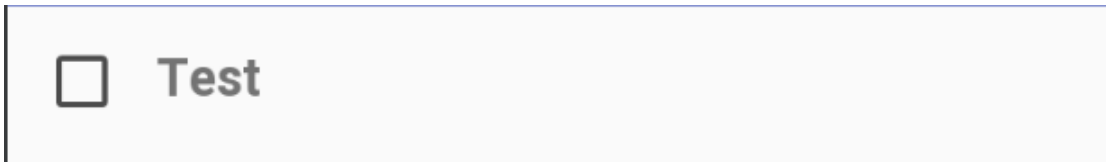
TasksAdapter

A diferencia de nuestro último adapter que tenía un constructor para pasar los parámetros, en este caso usaremos la propia clase para hacerlo. Se puede hacer de ambas formas pero así vais viendo formas diferentes que más se acomoden a vuestro modo de trabajo.

Antes de empezar a trabajar con el layout de la celda. Vamos a crear algo muy sencillito así que creamos un nuevo layout llamado **item_task.xml**.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    xmlns:tools="http://schemas.android.com/tools"
    android:orientation="horizontal"
    android:layout_margin="10dp">

    <CheckBox
        android:id="@+id/cbIsDone"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginRight="10dp"
        android:layout_marginEnd="10dp" />

    <TextView
        android:id="@+id/tvTask"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textStyle="bold"
        android:textSize="18sp"
        tools:text="Test"/>

</LinearLayout>
```

Nos quedará una celda muy sencilla. La idea es que cuando marquemos el Checkbox se actualice en la base de datos el objeto y si hacemos click en cualquier otra parte de la vista se borre de base de datos.



Así que esta vez le pasaremos 3 parámetros, la lista de tareas que tenemos almacenadas en nuestra base de datos y funciones. Estas funciones nos permitirán recuperar el evento del click en cada una de las celdas, ya sea la vista completa o un componente concreto.

```
class TasksAdapter(
        val tasks: List<TaskEntity>,
        val checkTask: (TaskEntity) -> Unit,
        val deleteTask: (TaskEntity) -> Unit) : RecyclerView.Adapter<Task
```

Ahora en el *onBindViewHolder* haríamos lo mismo de siempre pero añadiéndole el *setOnClickListener* a cada uno de los componentes que nos interese, en este caso yo se lo he puesto al checkbox y como quiero controlar el click en cualquier otra parte de la vista podemos acceder a *itemView* que nos devuelve la celda completa.

```
override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        val item = tasks[position]
        holder.bind(item, checkTask, deleteTask)
    }
```

Y ya completamos TaskAdapter con los dos métodos que nos faltan *onCreateViewHolder* y *getItemCount*. Estos métodos los dejaremos por defecto así que no voy a explicar nada ya que lo he hecho en capítulso anteriores.

```
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): Vi
        val layoutInflater = LayoutInflater.from(parent.context)
        return ViewHolder(layoutInflater.inflate(R.layout.item_task, pare
    }

    override fun getItemCount(): Int {
        return tasks.size
    }
```

Para finalizar la clase ViewHolder que solo tendrá de novedad en la función bind, a través de *.isChecked* podemos iniciar la vista con el checkbox marcado o no, así que comprobaremos si está a true nuestra entidad y si es así pues lo marcamos.

Una vez configurada la celda, le añadimos *.setOnClickListener* a nuestro checkBox y al *itemView* que es el componente completo pasando el propio objeto.

```
   class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {
        val tvTask = view.findViewById<TextView>(R.id.tvTask)
        val cbIsDone = view.findViewById<CheckBox>(R.id.cbIsDone)

       fun bind(task: TaskEntity, checkTask: (TaskEntity) -> Unit, delet
            tvTask.text = task.name
            cbIsDone.isChecked = task.isDone
            cbIsDone.setOnClickListener{checkTask(task)}
            itemView.setOnClickListener { deleteTask(task) }
        }
    }
```

Puedes ver la clase completa haciendo click aquí.

Volvemos al MainActivity

Con nuestro adapter completo es el paso de configurarlo desde MainActivity con nuestra función *setUpRecyclerView()* a la cual le pasaremos la lista de tareas que hemos recuperado.

```
fun setUpRecyclerView(tasks: List<TaskEntity>) {
        adapter = TasksAdapter(tasks, { updateTask(it) }, {deleteTask(it)
        recyclerView = findViewById(R.id.rvTask)
        recyclerView.setHasFixedSize(true)
        recyclerView.layoutManager = LinearLayoutManager(this)
        recyclerView.adapter = adapter
    }
```

Debemos fijarnos que al instanciar el adapter le pasamos tres parámetros, la lista de tareas, *updateTask(it)* y *deleteTask(it).* Estos no son parámetros sino métodos que tendremos en el **MainActivity**, que serán llamados automáticamente cuando se ejecute el evento del click que configuramos en el adapter.

Antes de mostraron esos dos métodos, vamos a configurar el botón de añadir tareas, que lo que hará será crear un objeto Task, almacenarlo en base de datos y luego añadirlo a la lista que tiene el adapter. Lo primero que haremos será ir a nuestro DAO y añadir una nueva función de insertar.

```
@Insert
fun addTask(taskEntity : TaskEntity):Long
```

Simplemente recibirá un objeto **TaskEntity** y lo añadirá a la base de datos. Fijaros que devuelve un Long, eso es porque nos dará automáticamente la **ID** del item añadido.

Nos vamos a nuestro *onCreate* del **MainAcitivity** y añadimos lo siguiente.

```
1  btnAddTask.setOnClickListener {
2              addTask(TaskEntity(name = etTask.text.toString()))}
```

Simplemente le hemos asignado al evento del click un método llamado *addTask()* al cual le pasamos un objeto nuevo con el texto de la celda. Dicha función añadirá a base de datos la tarea, luego recuperemos dicha tarea y la añadiremos a la lista del adapter. Hay varias formas de hacer esto mejor, pero he ido poniendo varias formas en cada una de las peticiones a la base de datos para poder observarlas.

```
1  fun addTask(task:TaskEntity){
2          doAsync {
3              val id = MisNotasApp.database.taskDao().addTask(task)
4              val recoveryTask = MisNotasApp.database.taskDao().getTaskByI
5              uiThread {
6                  tasks.add(recoveryTask)
7                  adapter.notifyItemInserted(tasks.size)
8                  clearFocus()
9                  hideKeyboard()
10             }
11         }
12     }
```

Así que lo que estamos haciendo es añadir la tarea y luego recuperamos dicho objeto a través de *getTaskById* pasándole la **ID** que nos devuelve *addTask*. Obviamente debemos añadir a nuestro **DAO** la función de recuperar el item.

```
1 @Query("SELECT * FROM task_entity where id like :arg0")
2 fun getTaskById(id: Long): TaskEntity
```

Cuando hemos acabado de realizar esto, en el hilo principal añadimos el objeto recuperado a la lista, le decimos al adapter que hemos añadido un objeto nuevo a través de *adapter.notifyItemInserted* (hay que pasarle pa posición, como es el último objeto añadido, podemos saber cual es recuperando el tamaño de la lista) y luego los métodos *clearFocus()* y *hideKeyboard()* simplemente nos quitarán el texto del editText y bajarán el teclado.

```
1 fun clearFocus(){
2    etTask.setText("")
3 }
4
5 fun Context.hideKeyboard() {
6     val inputMethodManager = getSystemService(Activity.INPUT_METHOD_SERVI
7     inputMethodManager.hideSoftInputFromWindow(currentFocus.windowToken,
8 }
```

Con esto ya podemos insertar tareas en nuestra app.

Una vez tengamos tareas, pasamos a actualizarlas con *updateTask()* que será el encargado de cambiar en la base de datos el **booleano** que marca si la tarea está hecha o no, así que nos volvemos a nuestro **DAO** y añadimos un update.

```
1 @Update
2 fun updateTask(taskEntity: TaskEntity):Int
```

Y su respectivo método en la actividad principal es muy sencillo, simplemente recibe el objeto, nosotros cambiamos el booleano por su opuesto (si es true, lo ponemos a false) y se lo mandamos al **DAO**.

```
1   fun updateTask(task: TaskEntity) {
2           doAsync {
3               task.isDone = !task.isDone
4               MisNotasApp.database.taskDao().updateTask(task)
5           }
6       }
```

Recordamos que con la exclamación delante de un booleano nos da el valor contrario.

Para finalizar nuestra app nos falta la opción de borrar tareas tocando cualquier parte de la celda menos el checkBox que ya hace una función de actualizar. Así que volvemos al **DAO** y hacemos una función @Delete.

```
1  @Delete
2  fun deleteTask(taskEntity: TaskEntity):Int
```

En el método *deleteTask()* debemos hacer algo más, para empezar, buscaremos en nuestra lista **tasks** la posición del item que vamos a borrar para tener una referencia. Para ello usaremos la función de las listas *indexOf(item)* que nos devolverá dicha posición y la almacenamos en una variable, luego borraremos el objeto de la base de datos y de nuestra lista y acabaremos en el hilo principal avisando al adapter que hemos removido un objeto, pasándole la posición.

```
1   fun deleteTask(task: TaskEntity){
2           doAsync {
3               val position = tasks.indexOf(task)
4               MisNotasApp.database.taskDao().deleteTask(task)
5               tasks.remove(task)
6               uiThread {
7                   adapter.notifyItemRemoved(position)
8               }
9           }
10      }
```

Puedes ver la clase completa haciendo click aquí.

Este artículo ha sido un poquito más “intenso” pero ya empezamos a entrar en verdadera materia. Os recuerdo que podéis descargar el proyecto completo desde mi GitHub. Y que si tenéis alguna duda podéis dejarla en los comentarios.