

Kotlin

**Índice**

Variables y tipos de datos ……………………………………. 4

Funciones ……………………………………………………… 5

Lista (*Array*) …………………………………………………... 7

Diccionario (*Map*) ………………………………………………. 8

Bucles: *for*, *while* ……………………………………………. 9

NullSafety ………………………………………………………. 10

Clases …………………………………………………………… 12

Valores por defecto y SafetyNull ………………………………… 13

Constructor ………………………………………………………… 14

Enumerados (*enum*) ………………………………………………. 15

Clases anidadas …………………………………………………… 17

Herencia ……………………………………………………………. 18

Clase abstracta ……………………………………………………. 20

Interface ……………………………………………………………. 21

Modificadores de visibilidad ……………………………………… 22

Data Class …………………………………………………………. 23

Alias (Type Aliases) ………………………………………………. 24

Declaraciones desestructuradas …………………………….. 26

Extensiones ……………………………………………………. 28

Lambdas ……………………………………………………….. 29

Jetpack Compose ……………………………………………... 32

Composable (funciones que permiten modificar el layout) ………… 33

Preview (vista previa del layout) ………………………………… 34

Modifier (características de los elementos del layout) ………… 34

Material (elementos del layout) ………………………………….. 35

Theme ………………………………… 37

LazyColumn (listas y scroll vertical) …………………………………… 38

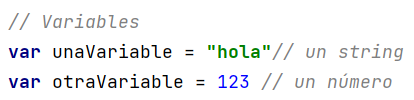
Animaciones ………………………………………………………...

Navegación entre pantallas ……………………………………….

Interacción entre vistas ………………………………………..

**Variables y tipos de datos**

Cada variable se inicia con la palabra reservada **var**, el tipo de dato se definirá al asignar el primer dato a la variable.



Una constante se declara con la palabra reservada **val**. No puede modificarse su valor y el tipo de dato se define al asignar un dato.





Se puede forzar el tipo de dato de una variable:

En resumen, existen los siguientes tipos de datos primitivos

* Enteros: Byte, Short, Int, Long
* Decimales: Float, Double
* Booleanos: Boolean
* Textos: String

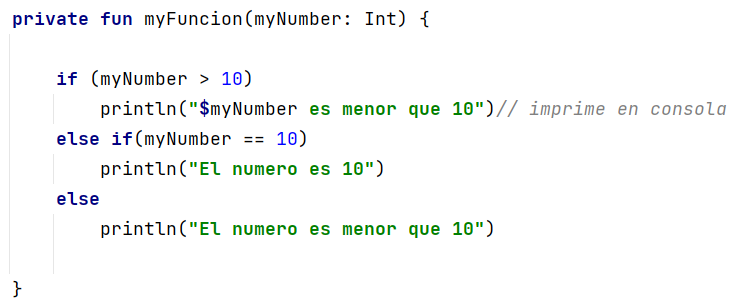
**Funciones**

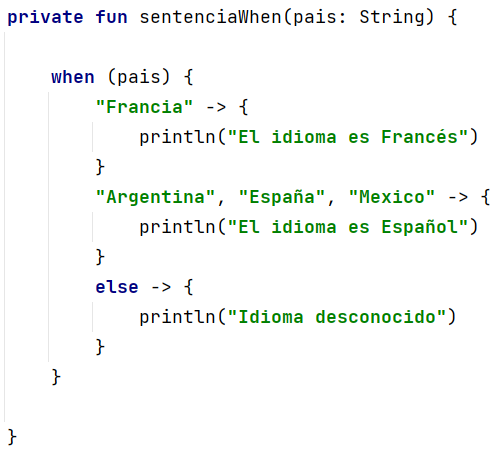
Estructuras condicionales

Las estructuras condicionales de tipo **if** permiten establecer si una característica es verdadera o falsa.

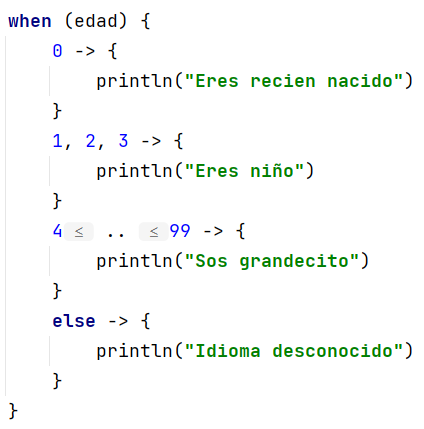
Nombre de la variable (parámetro)

Tipo de dato del parámetro



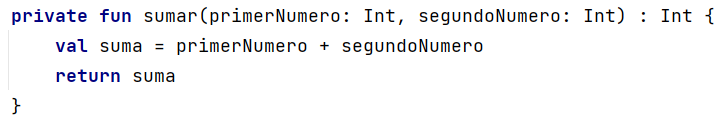
Sentencia de control de flujo

La sentencia WHEN (o switch) permite controlar grandes cantidades de condicionales.



Funciones con valores de retorno

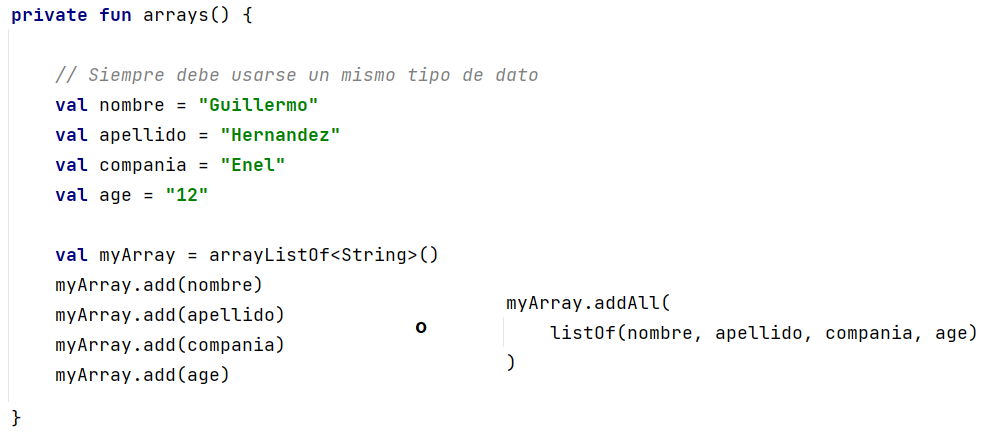
Para que una función devuelva un dato, se debe escribir de la siguiente manera:



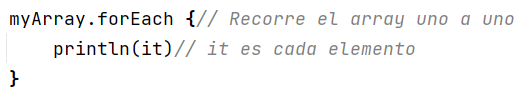
La palabra reservada **Unit** será para indicar que no habrá retorno (como un **void**)

**Array / Arreglo**

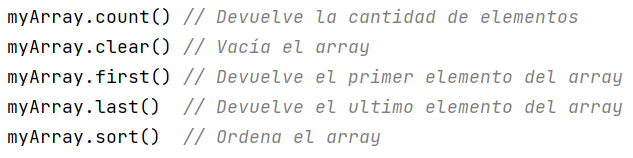
Un conjunto ordenado de diversos datos del mismo tipo.



Para eliminar un elemento en específico se utiliza:



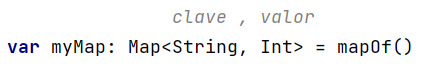
Para recorrer un array, se utiliza **forEach**



**Map / Diccionario**

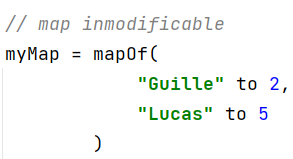
Un Map (Diccionario) es un tipo de Colección no ordenada.

Se estructura en Clave-Valor, y **no pueden existir dos claves iguales**.



Map inmodificable

Es aquel diccionario que no puede modificarse tras crearse y definirse.

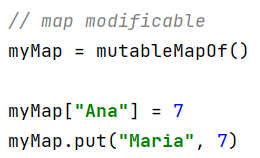


valor

clave

Map modificable

Diccionario que puede ser modificado luego de crearse.



valor

clave

Si se desea editar el valor asignado a una clave, se utiliza el mismo método **put**

Para acceder a un dato, se utiliza:



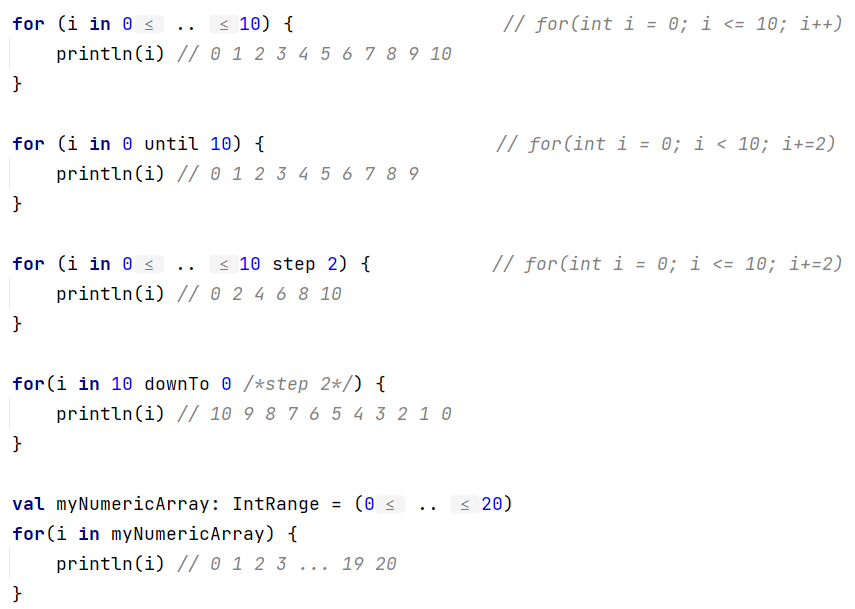
Para remover una clave se utiliza:

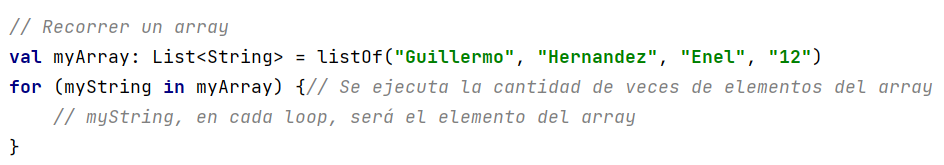
clave

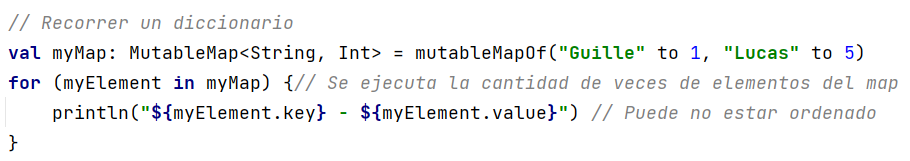
**Bucles**

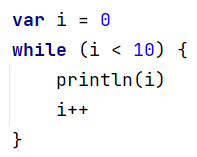
Permite recorrer estructuras que almacenan datos, como rangos numéricos, arrays o maps.

Bucles de tipo **FOR**







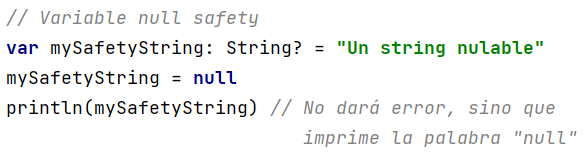


Bucles de tipo **WHILE**

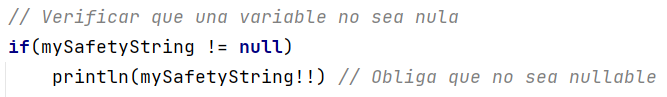
Ejecuta un bucle mientras la condición sea verdadera.

**Null safety**

La seguridad contra nulos permite prevenir errores de tipo NullPointerException debido a la existencia de un **null** en alguna variable. Esto se realiza agregando un signo de pregunta al tipo de dato.

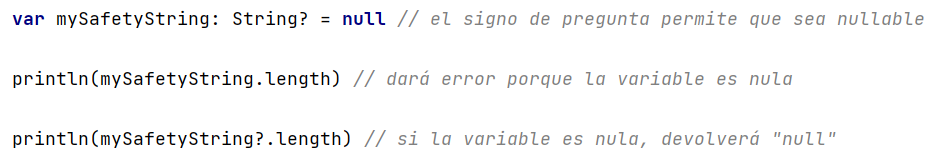


Se puede obligar a que una variable no sea nula con un doble signo de exclamación (como si el tipo de dato no tuviera signo de pregunta), pero para ello es necesario verificar que la variable efectivamente no es nula ya que en caso contrario dará error.

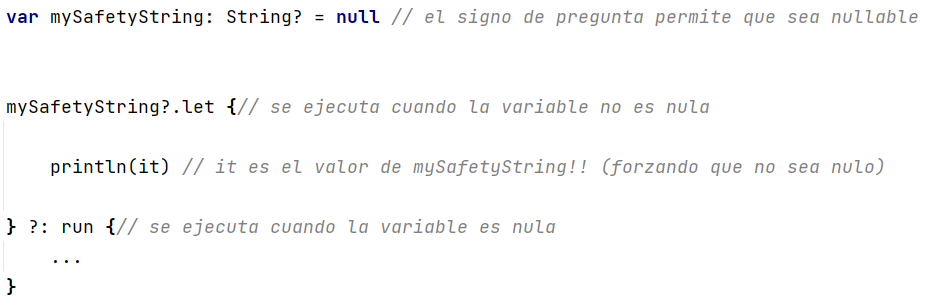


Safe call

Para evitar verificaciones en variables nullables, se puede utilizar el signo de pregunta nuevamente de la siguiente manera:



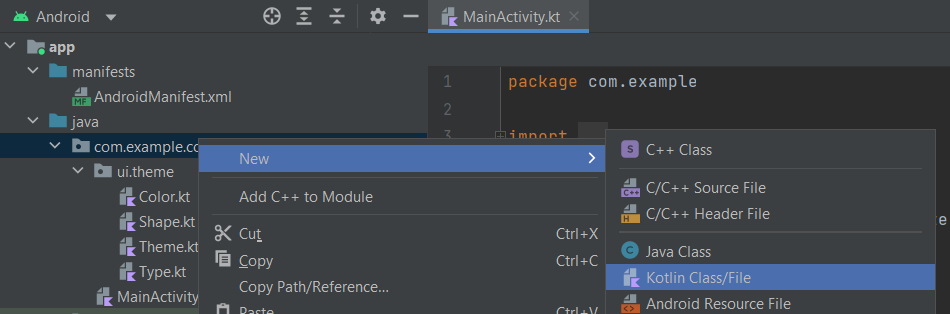
Para ejecutar un bloque de código u otro según si la variable es nula o no se utiliza el método **LET** y el método **RUN**



**Clases**

Objetos (construcciones flexibles) de uso general, para definir nuevos tipos de datos. Pueden contener en su interior atributos y métodos.

Para crear una class, se procede de la siguiente manera:



Una vez creado el archivo donde se alojará nuestra class, se escriben sus atributos de las siguientes maneras, dependiendo de si se quiere que sean atributos públicos o privados (sólo accesibles desde la misma class)

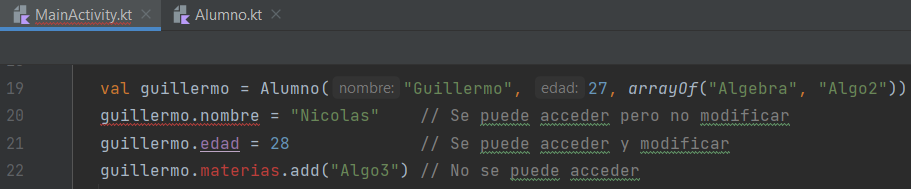
Listado de strings



público público privado

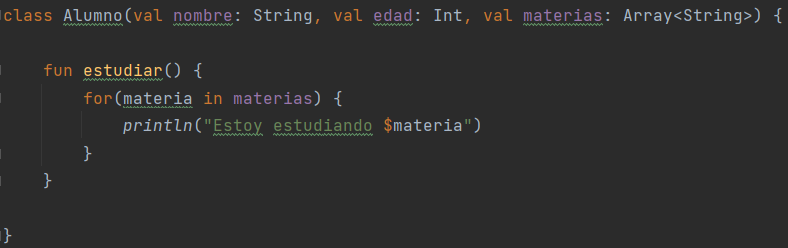
constante variable (sólo accesible desde la misma class)

La forma de instanciar un objeto de clase es la siguiente:



Métodos

Se pueden agregar métodos (funciones de la class):

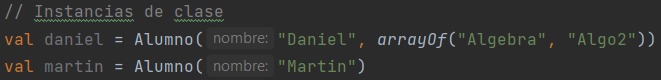


Valores por defecto

Se permite el agregado de valores por defecto a los parámetros de la siguiente manera:



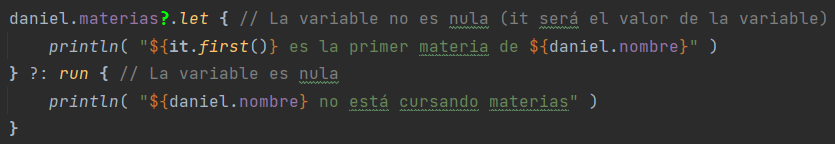
Es importante utilizar el **NullSafety** para los casos en que puedan existir parámetros nulos.

Cuando existen valores por defecto, se puede instanciar un objeto de la clase sin definirlos:

Al momento de utilizar los parámetros que pueden ser nulos (marcados con el signo de pregunta), es necesario utilizar el **SafeCall**.



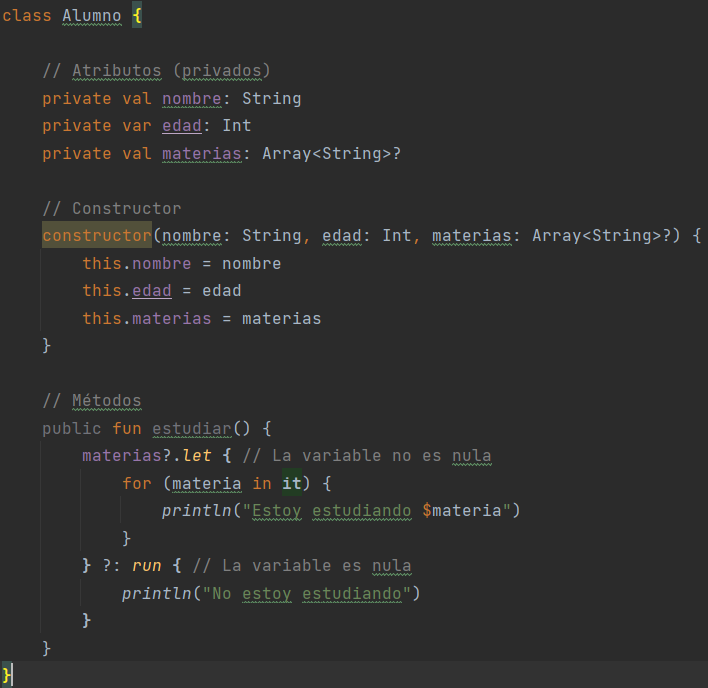
O bien, utilizando los métodos Let-Run:



Constructor

Una class genérica posee un constructor, atributos de clase y métodos.

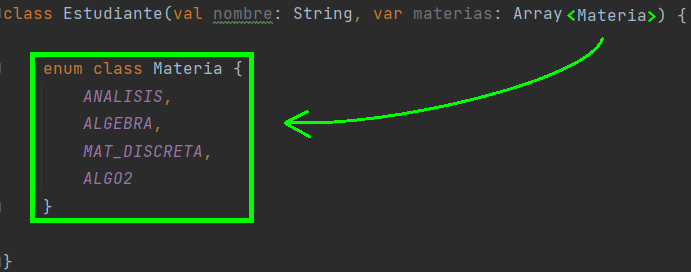
El constructor existe por defecto con los parámetros establecidos en la firma de la class. Sin embargo, puede explicitarse con la palabra reservada **constructor**.



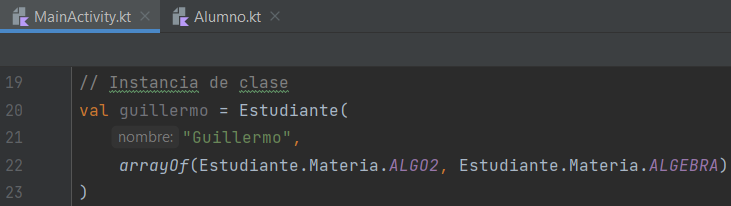
Enumerados

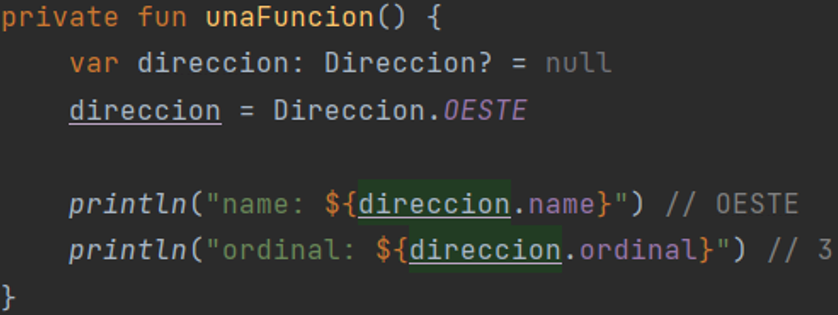
Si se desea trabajar con una lista enumerada (class de enumeración), se utilizan los ***enum***.

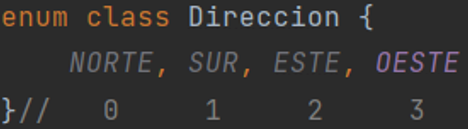
Permite implementar enumeraciones ***type-safe*** (tipado seguro), es decir, la class creada tomará valores que siempre serán conocidos con antelación.



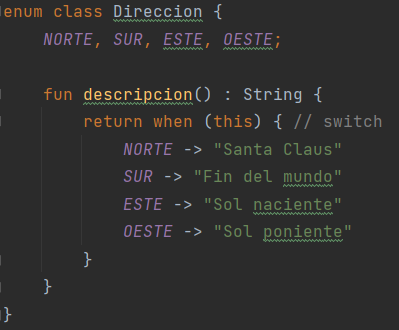
De esta manera, en la lista de materias sólo se podrá escribir uno de los elementos enumerados y definidos con antelación, no existe posibilidad de equivocarse:



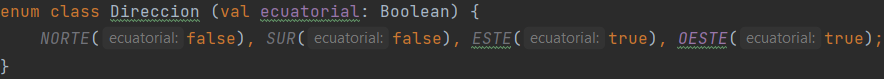
* Propiedades: ***name*** *y* ***ordinal***



* Métodos (***this*** hará referencia al elemento enumerado seleccionado)



* Parámetros (propiedades extras)

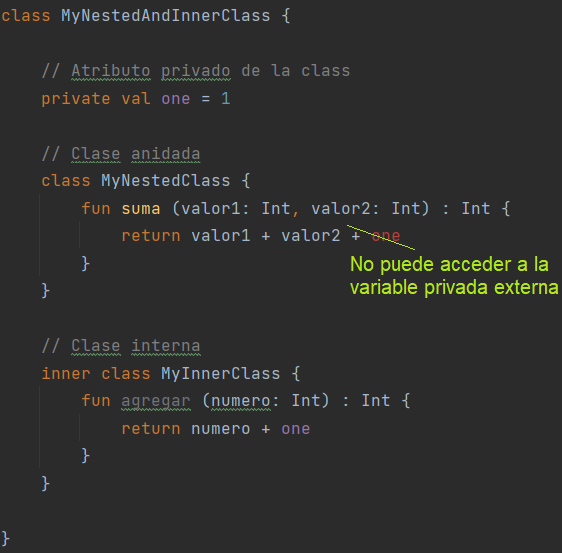


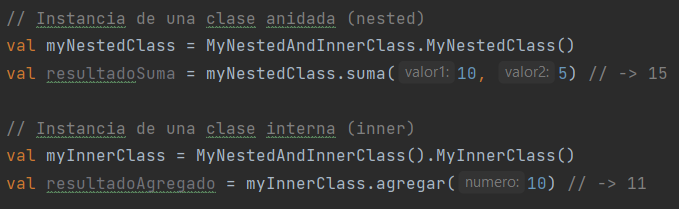


Clases anidadas (Nested and Inner Classes)

Son aquellas clases que se encuentran contenidas dentro de otra class. Favorecen el encapsulamiento.

La **clase anidada** (nested) **no puede** acceder a los miembros de la clase externa, mientras que la **clase interna** (inner) **sí puede**.



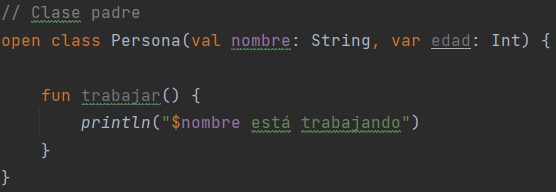


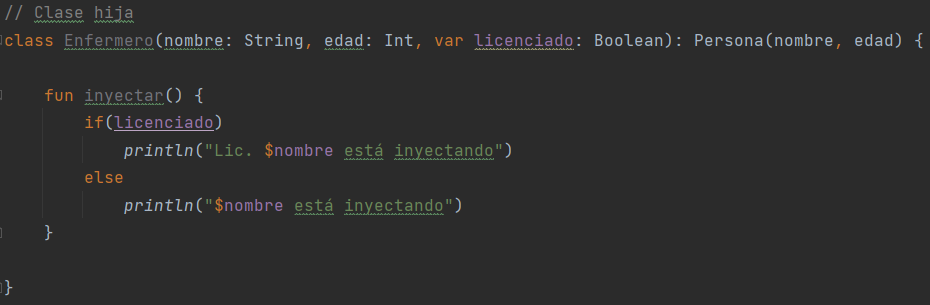
Herencia (inheritance)

En Kotlin, toda class es heredada de la “súper clase” (clase padre) común llamada **Any**.

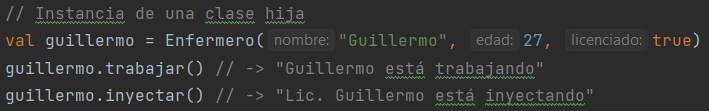
Además, toda class se crea de manera “final”, es decir, por defecto no pueden tener herencia.

Para que una clase pueda tener herencia (y convertirse en *clase padre*), debe tener la palabra reservada **open**. Además, al crear la clase hija, será necesario redefinir los parámetros de la clase padre.



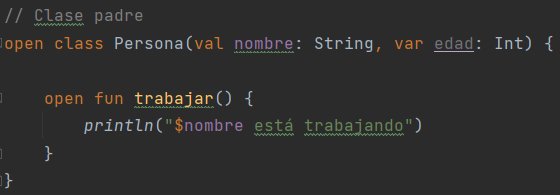


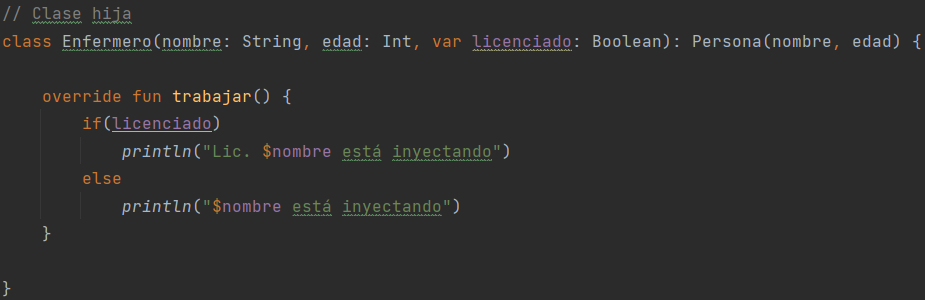
Finalmente, para instanciarlo, se realiza desde la clase hija, la cual podrá utilizar los métodos y propiedades de su clase padre.



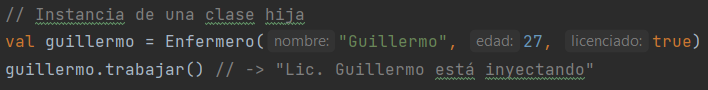
* Sobreescritura (override)

Para poder modificar un método definido en la clase padre, en la clase hija, será necesario agregar la palabra reservada **open** en el método original y **override** en el método de la clase hija.



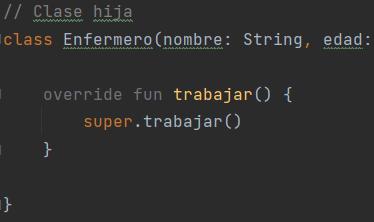


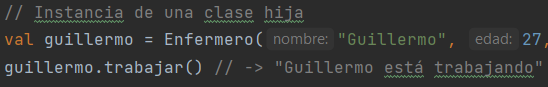
De esta manera, se puede utilizar el método “trabajar” desde la clase hija para ejecutar un método propio y ya no el de la clase padre.



* **super**

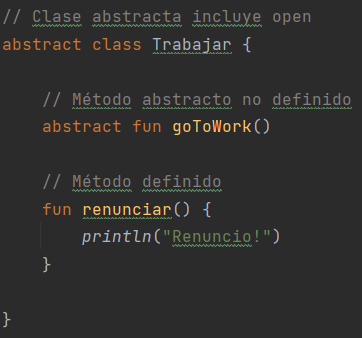
La palabra reservada súper permite ejecutar el método original de la clase padre desde la clase hija, incluso si éste ha sido sobrescrito.

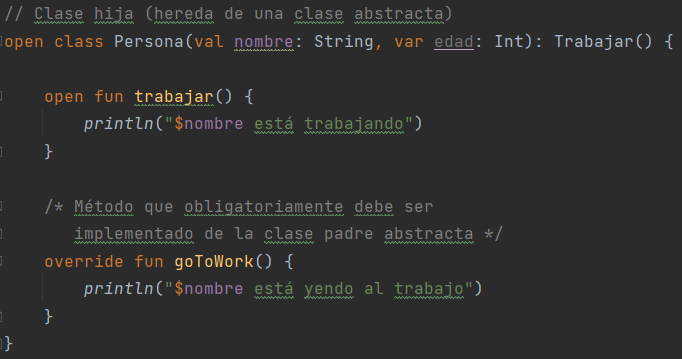




Clase abstracta

No define una implementación, sino un comportamiento.

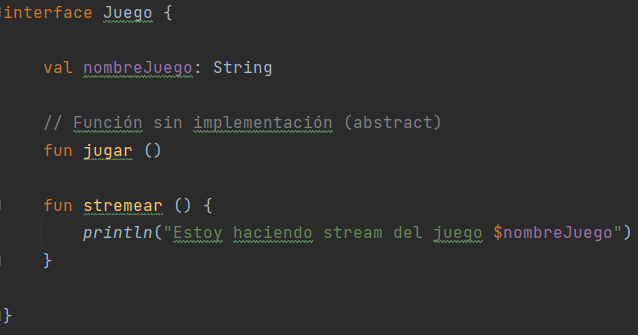
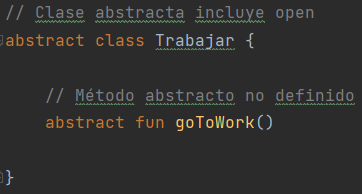


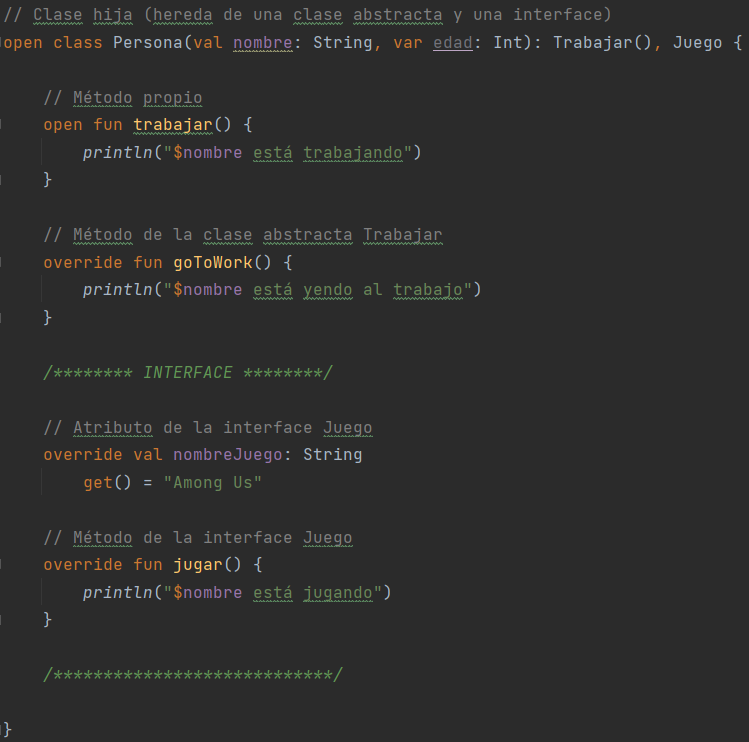


Interfaces

Pueden contener declaraciones abstractas o implementaciones de funciones o propiedades.

La diferencia con las clases abstractas es que las interfaces **no pueden almacenar estados**, y por lo tanto **no pueden instanciarse ni tener constructores**.





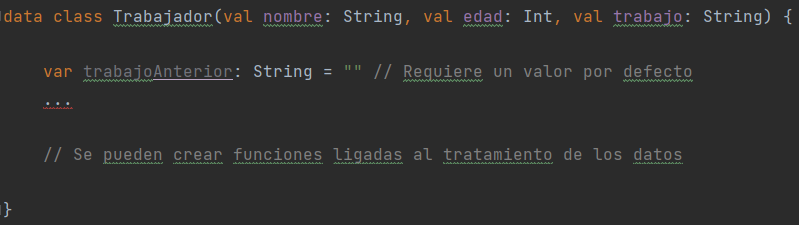
Modificadores de visibilidad

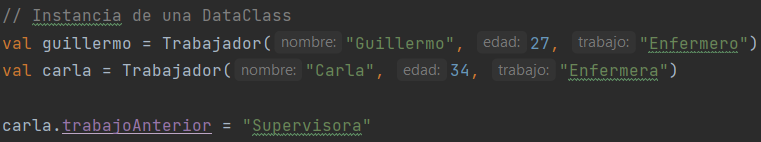
Limita el acceso a métodos y atributos de las clases.

* **Public** (*por defecto*): será accesible desde cualquier parte del código.
* **Private**: sólo será accesible desde la class donde se encuentra.
* **Protected**: accesible desde clases de nivel inferior (subclases o clases hijas)
* **Internal**: accesible desde cualquier parte del módulo o fichero de kotlin.

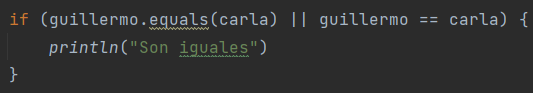
Data class

Permite almacenar datos (atributos) y operar con ellos con varios métodos que existen por defecto.

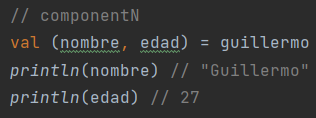




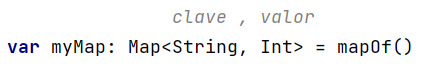
Algunos de los métodos por defecto de una DataClass son:



* equals Permite comparar si una DataClass es igual a otra.
* toString Transforma en texto los datos almacenados en el constructor principal.
* copy Permite copiar una DataClass en otra, como parámetro se puede alterar alguno de los atributos que contenga.



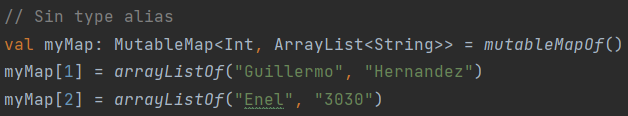
* componentN Se trata de una función de desestructuración de datos. Utiliza una Tupla para obtener atributos del DataClass directamente.

**Type Aliases**

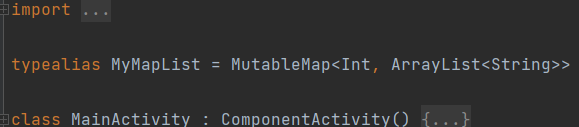
Permite acortar nomenclaturas de tipos de datos, funciones y clases anidadas.

Se utiliza la palabra reservada **typealias** al inicio de la class donde se utilizará.

Map

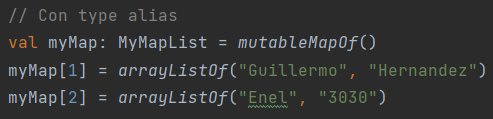


Requiere escribir todo el tipo de dato del Map



Sin embargo, si definimos el tipo de dato en un type alias:

Ya no será necesario escribirlo cada vez que utilicemos ese mismo tipo de dato para un map.



Clave Valor

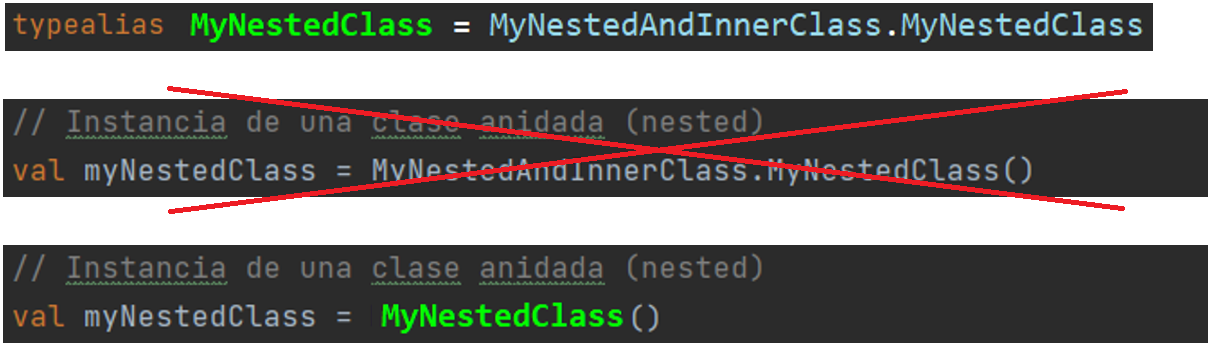
Funciones

Se puede definir toda la firma de la función y su retorno en un type alias.



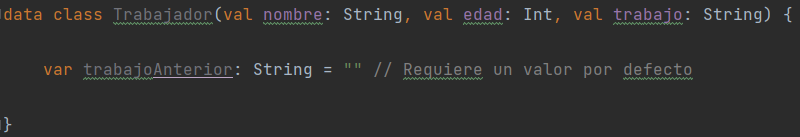
Clases anidadas

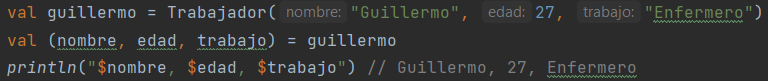
Se puede resumir el llamado de una clase anidada mediante un type alias.



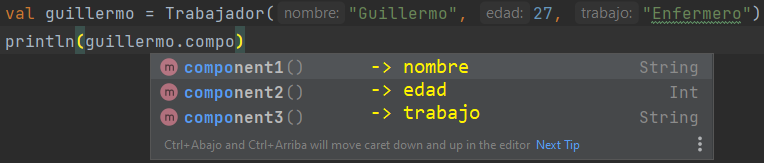
**Declaraciones desestructuradas**

Se trata de desestructurar en varios fragmentos un elemento más grande. Crea varias variables de una sola vez para almacenar, en cada una, los datos de un elemento mayor.

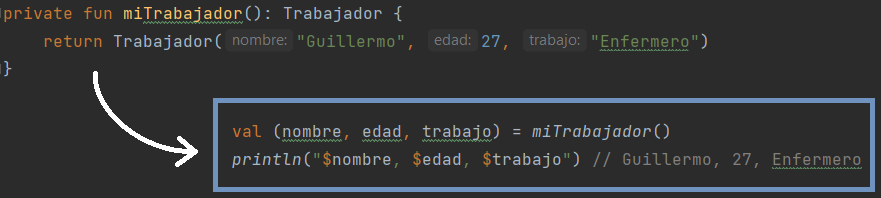
Se pueden desestructurar DataClasses creando las variables de su constructor.

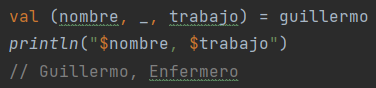


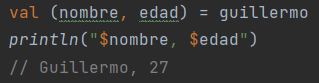
O bien, accediendo a dichos parámetros mediante la palabra “component”



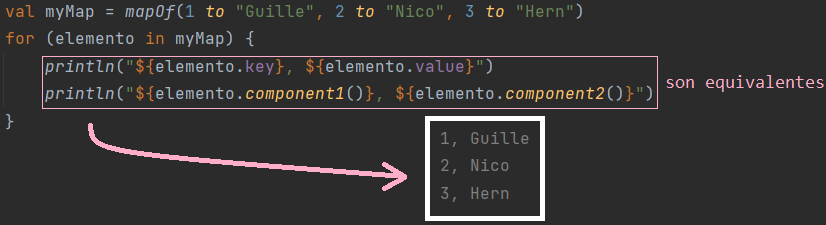
Si tenemos una función que retorna en una DataClass, se puede desestructurar de la misma manera.

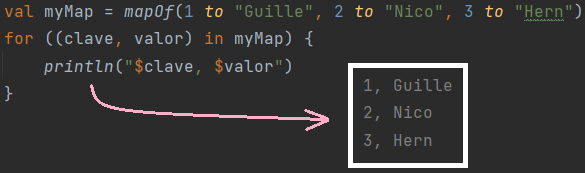


No necesariamente se transforma en variable todos los parámetros que contiene el DataClass.



Se pueden desestructurar **Maps**



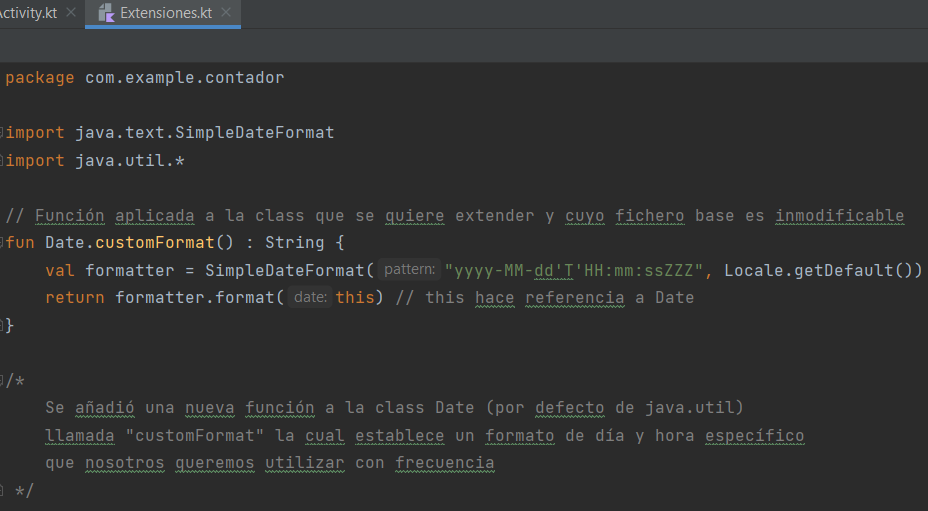
Construyendo variables que almacenen los “component”, puede quedar de la siguiente manera:

**Extensiones**

Posibilita ampliar la funcionalidad de una class sin la necesidad de utilizar la herencia.

Se suele utilizar para añadir funciones o propiedades a una class previamente creada por un tercero.

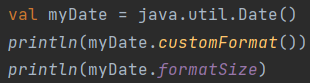
Para realizar esto, es necesario crear un archivo kotlin con el siguiente formato:



Luego, podremos utilizar una función personalizada extendida de una class preexistente:

Es de utilidad tener en cuenta que el objeto instanciado puede ser nulo **(this** == **null)** y, para trabajarlo, será necesario agregar el interrogante:

Por otro lado, se puede crear un atributo / variable / propiedad a una class preexistente:



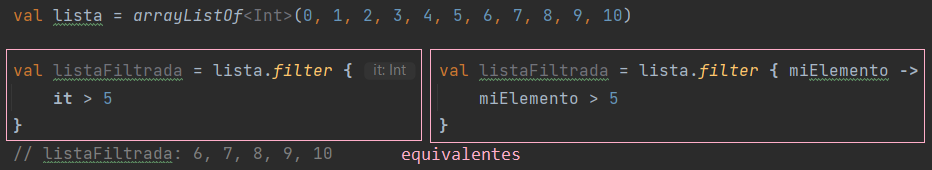
Si customFormat pudiera ser nulo, se puede definir así:

**Lambdas**

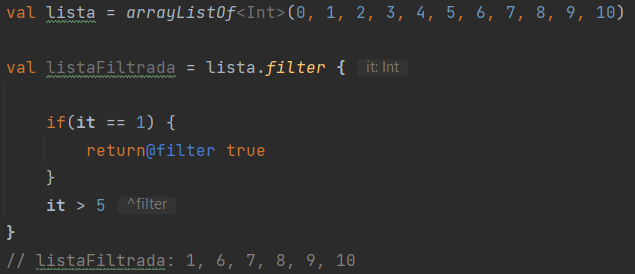
Funciones de orden superior que permite definir variables o funciones que pueden trabajar con otras funciones.

* Filter

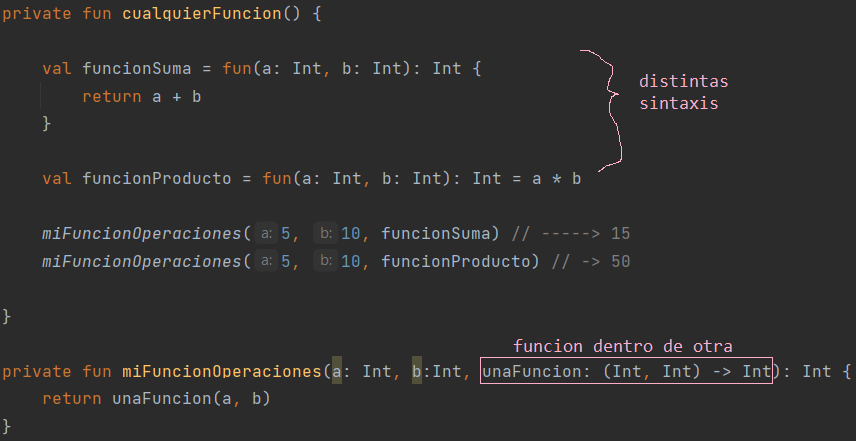
Es una función que puede utilizarse dentro de otra función para filtrar elementos de un array.



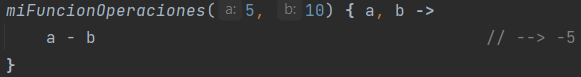
Se pueden agregar excepciones mediante ***return@filter***



Una forma de crear funciones lambda personalizadas es la siguiente:



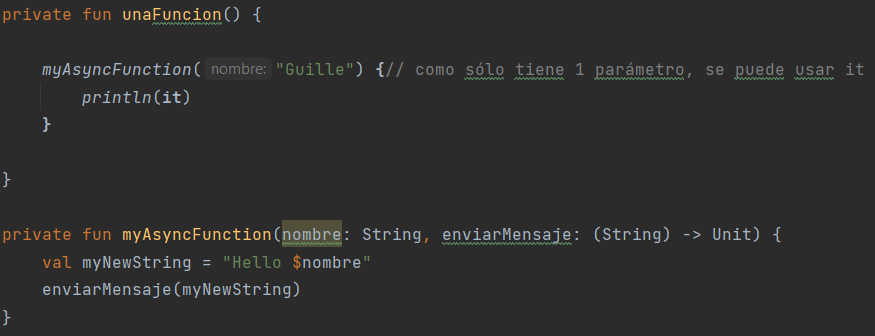
También se puede definir la función dentro de la otra de forma directa, en el llamado de la función principal.



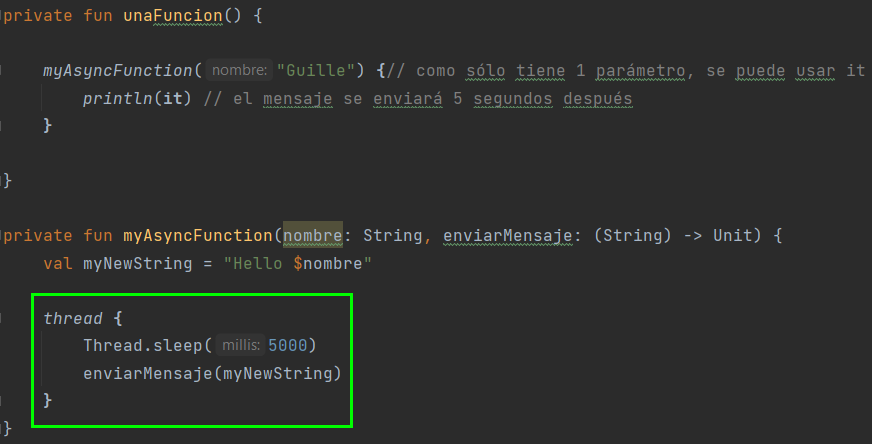
Funciones callback (asíncrona)

Una función asíncrona es aquella que se ejecuta en segundo plano, no línea a línea como cualquier otra función. Es de especial utilidad cuando se realizan llamados a servidores externos.

La forma de crear un lambda con capacidad asíncrona es la siguiente:



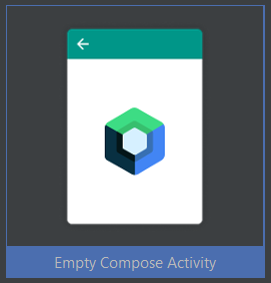
Hasta ahora la función no es asíncrona, para ello se utiliza **thread**.



**Jetpack Compose**

Escrito en Kotlin, introduce un paradigma de programación declarativa para crear vistas dinámicas (reemplazando la programación imperativa de las vistas en xml).



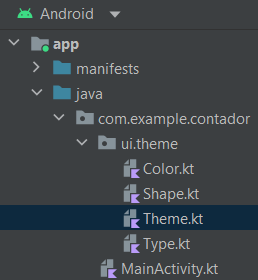
Para comenzar a utilizar Jetpack Compose, es necesario crear una nueva activity y seleccionar la que incluya la palabra “Compose”.

Es necesario que el lenguaje de programación a utilizar sea Kotlin.

El SDK mínimo deberá ser Lollipop (API 21 – Android 5.0)

Una vez creada la activity, se debe empezar a crear una “vista” en **onCreate**, dentro de un bloque de código denominado **setContent**.



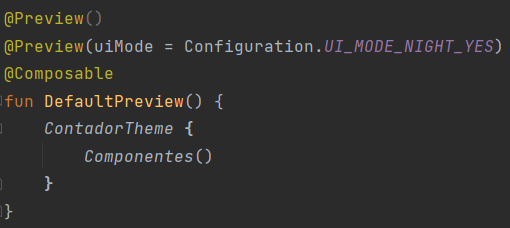


Donde el Theme está predefinido dentro de Theme.kt

Función preview composable

Una función composable con el tag “preview” permite la visualización dentro de la interfaz. Para ello, es necesario especificar qué se va a mostrar y luego compilar el programa.

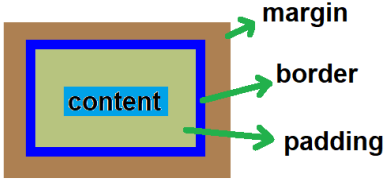


Es posible, además, visualizar cómo se vería en modo claro y modo oscuro al mismo tiempo de la siguiente manera:

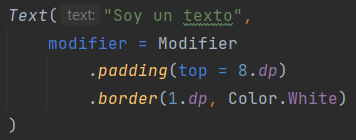
Si se utiliza **showSystemUi = true**, se visualizará en modo pantalla real.

Modifier

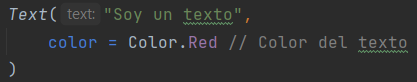
Es un método que poseen casi todos los elementos composables que permite modificar sus características, como el tamaño, padding, color, etc.

Los nombres de dichas características son las mismas que se utilizan en los archivos xml.

Por ejemplo, en un texto:

Además, tiene la propiedad de poder concatenarse, ejecutándose los modificadores en el orden en que se escriben (lo cual puede provocar efectos diferentes)

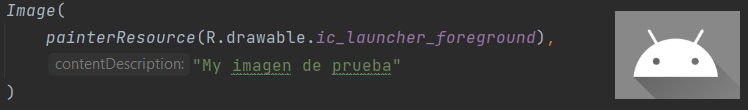
Material

Los elementos composables (material) son todos aquellos que pueden utilizarse como vistas y layout.

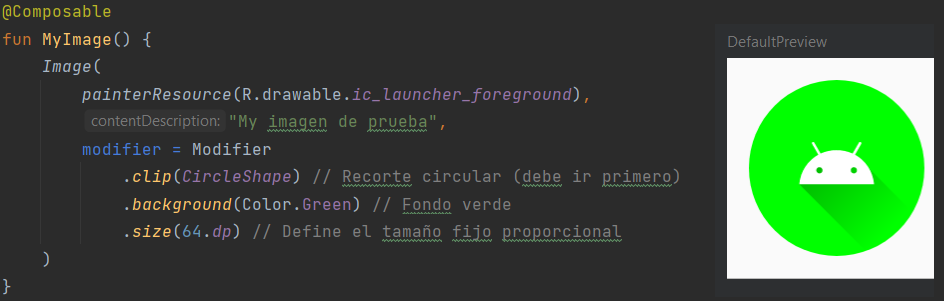
* Text: permite escribir un texto (*TextView*)
* Spacer: permite colocar un separador



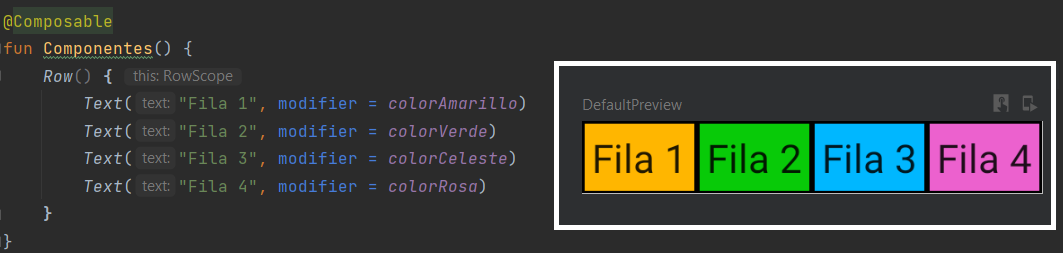
* Image: permite agregar una imagen



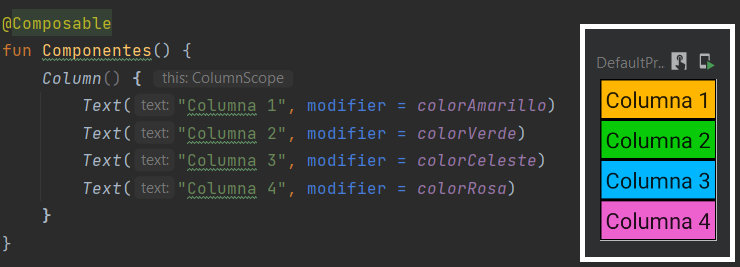
La cual puede ser modificada con ciertos Modificadores:



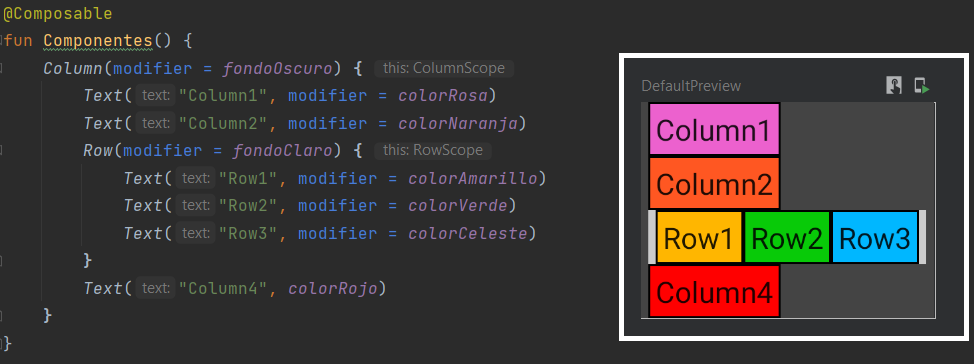
* Row y Column
  + Row



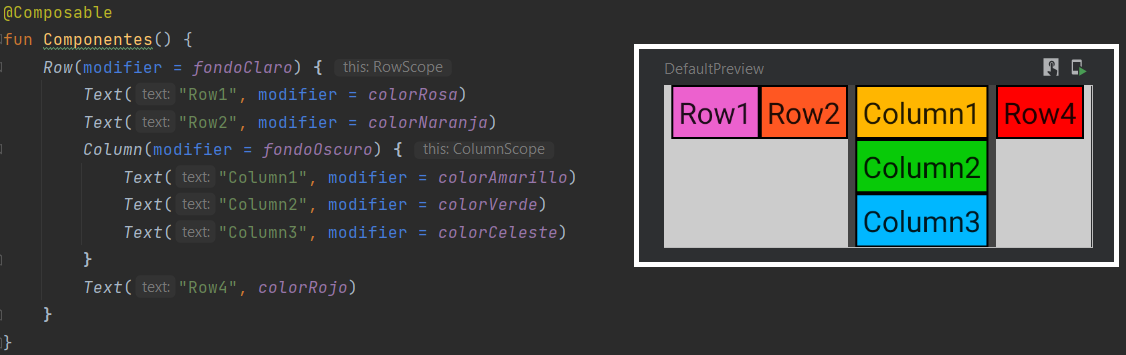
* + Column



* + Row en Column



* + Column en Row

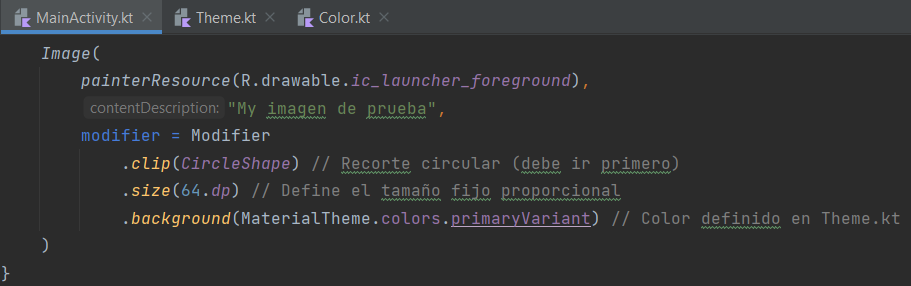


Tema (theme)

Los temas también funcionarán como elementos composables, se pueden configurar dentro de la carpeta Theme.kt y se aplican dentro de setContent.

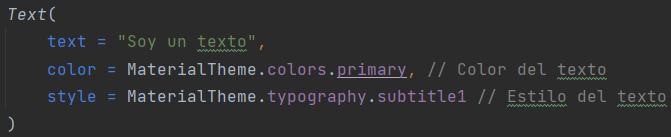


Hecho esto, se podrán utilizar los colores definidos dentro de nuestro Theme:



Dentro de Theme.kt, las propiedades que inician con la palabra ***on*** hacen referencia a los textos que actúan sobre los colores de estos mismos valores. Es decir, por ejemplo, **onBackground** es el color del texto dentro de los elementos con la propiedad ***background***.

Los textos son altamente modificables con nuestros Temas:



Listas y Scroll vertical (*LazyColumn*)

Animaciones

Navegación entre pantallas