



# Kotlin

# Índice

Variables y tipos de datos .....	4
Funciones .....	5
Lista ( <i>Array</i> ) .....	7
Diccionario ( <i>Map</i> ) .....	8
Bucles: <i>for</i> , <i>while</i> .....	9
NullSafety .....	10
Clases .....	12
Valores por defecto y SafetyNull .....	13
Constructor .....	14
Enumerados ( <i>enum</i> ) .....	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.

```
// Variables  
var unaVariable = "hola" // un string  
var otraVariable = 123 // un número
```

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.

```
// Constante  
val unaConstante = false // un boolean
```

Se puede forzar el tipo de dato de una variable:

```
// Variable forzada a ser de tipo entero  
var variableForzada: Int
```

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

```
private fun myFuncion(myNumber: Int) {  
  
    if (myNumber > 10)  
        println("$myNumber es menor que 10")// imprime en consola  
    else if(myNumber == 10)  
        println("El numero es 10")  
    else  
        println("El numero es menor que 10")  
  
}
```

## Sentencia de control de flujo

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

```
private fun sentenciaWhen(pais: String) {  
  
    when (pais) {  
        "Francia" -> {  
            println("El idioma es Francés")  
        }  
        "Argentina", "España", "Mexico" -> {  
            println("El idioma es Español")  
        }  
        else -> {  
            println("Idioma desconocido")  
        }  
    }  
  
}
```

```
when (edad) {  
    0 -> {  
        println("Eres recién nacido")  
    }  
    1, 2, 3 -> {  
        println("Eres niño")  
    }  
    4 ≤ .. ≤ 99 -> {  
        println("Sos grandecito")  
    }  
    else -> {  
        println("Idioma desconocido")  
    }  
}
```



## Funciones con valores de retorno

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

```
private fun sumar(primerNumero: Int, segundoNumero: Int) : Int {  
    val suma = primerNumero + segundoNumero  
    return suma  
}
```

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.

```
private fun arrays() {  
  
    // Siempre debe usarse un mismo tipo de dato  
    val nombre = "Guillermo"  
    val apellido = "Hernandez"  
    val compania = "Enel"  
    val age = "12"  
  
    val myArray = arrayListOf<String>()  
    myArray.add(nombre)  
    myArray.add(apellido)           o      myArray.addAll(  
    myArray.add(compania)           |      listOf(nombre, apellido, compania, age)  
    myArray.add(age)                )  
}
```

Para eliminar un elemento en específico se utiliza:

```
myArray.removeAt(2) // Se elimina compania
```

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

```
myArray.forEach { // Recorre el array uno a uno  
    println(it) // it es cada elemento  
}
```

```
myArray.count() // Devuelve la cantidad de elementos  
myArray.clear() // Vacía el array  
myArray.first() // Devuelve el primer elemento del array  
myArray.last()  // Devuelve el ultimo elemento del array  
myArray.sort()  // Ordena el array
```



## Map / Diccionario

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

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

```
clave , valor  
var myMap: Map<String, Int> = mapOf()
```

### Map inmodificable

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

```
// map inmodificable  
myMap = mapOf(  
    "Guille" to 2,  
    "Lucas" to 5  
)
```

valor  
clave

### Map modificable

Diccionario que puede ser modificado luego de crearse.

```
// map modificable  
myMap = mutableMapOf()  
  
myMap["Ana"] = 7  
myMap.put("Maria", 7)
```

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:

```
println(myMap["Ana"]) // 7
```

Para remover una clave se utiliza:

```
myMap.remove("Ana")
```

clave





## Bucles

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

### Bucles de tipo FOR

```
for (i in 0 ≤ .. ≤ 10) {                                // for(int i = 0; i <= 10; i++)
|   println(i) // 0 1 2 3 4 5 6 7 8 9 10
}

for (i in 0 until 10) {                                  // for(int i = 0; i < 10; i+=2)
|   println(i) // 0 1 2 3 4 5 6 7 8 9
}

for (i in 0 ≤ .. ≤ 10 step 2) {                          // for(int i = 0; i <= 10; i+=2)
|   println(i) // 0 2 4 6 8 10
}

for(i in 10 downTo 0 /*step 2*/) {
|   println(i) // 10 9 8 7 6 5 4 3 2 1 0
}

val myNumericArray: IntRange = (0 ≤ .. ≤ 20)
for(i in myNumericArray) {
|   println(i) // 0 1 2 3 ... 19 20
}
```

```
// Recorrer un array
val myArray: List<String> = listOf("Guillermo", "Hernandez", "Enel", "12")
for (myString in myArray) { // Se ejecuta la cantidad de veces de elementos del array
|   // myString, en cada loop, será el elemento del array
}
```

```
// Recorrer un diccionario
val myMap: MutableMap<String, Int> = mutableMapOf("Guille" to 1, "Lucas" to 5)
for (myElement in myMap) { // Se ejecuta la cantidad de veces de elementos del map
|   println("${myElement.key} - ${myElement.value}") // Puede no estar ordenado
}
```

### Bucles de tipo WHILE

Ejecuta un bucle mientras la condición sea verdadera.

```
var i = 0
while (i < 10) {
|   println(i)
|   i++
}
```



## 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.

```
// Variable null safety
var mySafetyString: String? = "Un string nullable"
mySafetyString = null
println(mySafetyString) // No dará error, sino que
                        // imprime la palabra "null"
```

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.

```
// Verificar que una variable no sea nula
if(mySafetyString != null)
    println(mySafetyString!!) // Obliga que no sea nullable
```

## Safe call

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

```
var mySafetyString: String? = null // el signo de pregunta permite que sea nullable

println(mySafetyString.length) // dará error porque la variable es nula

println(mySafetyString?.length) // si la variable es nula, devolverá "null"
```



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**

```
var mySafetyString: String? = null // el signo de pregunta permite que sea nullable

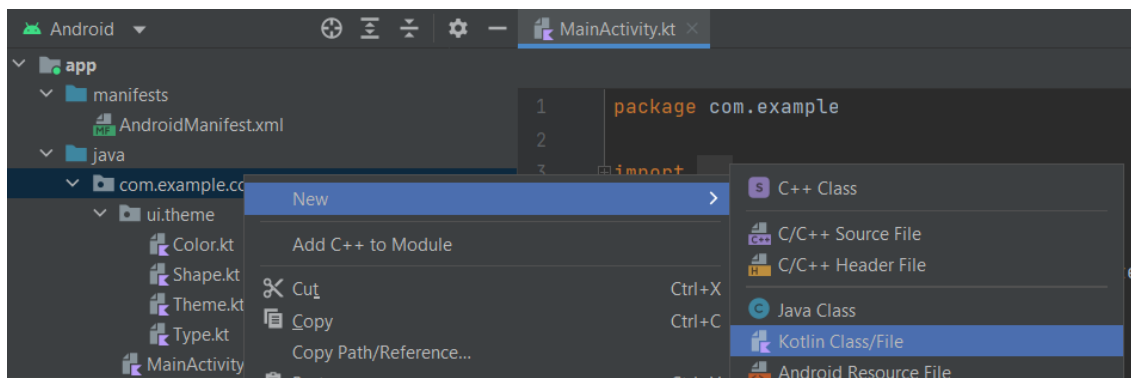
mySafetyString?.let { // se ejecuta cuando la variable no es nula
    println(it) // it es el valor de mySafetyString!! (forzando que no sea nulo)
} ?: run { // se ejecuta cuando la variable es nula
    ...
}
```



## 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)

```
class Alumno(val nombre: String, var edad: Int, materias: Array<String>) {
```

público  
constante

público  
variable

privado  
(sólo accesible desde la misma class)

Listado de strings

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

```
19 val guillermo = Alumno( nombre: "Guillermo", edad: 27, arrayOf("Algebra", "Algo2"))
20 guillermo.nombre = "Nicolas" // Se puede acceder pero no modificar
21 guillermo.edad = 28 // Se puede acceder y modificar
22 guillermo.materias.add("Algo3") // No se puede acceder
```



## Métodos

Se pueden agregar métodos (funciones de la class):

```
class Alumno(val nombre: String, val edad: Int, val materias: Array<String>) {  
  
    fun estudiar() {  
        for(materia in materias) {  
            println("Estoy estudiando $materia")  
        }  
    }  
}
```

guillermo.estudiar()

## Valores por defecto

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

```
class Alumno(val nombre: String, val materias: Array<String>? = null) {
```

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:

```
// Instancias de clase  
val daniel = Alumno( nombre: "Daniel", arrayOf("Algebra", "Algo2"))  
val martin = Alumno( nombre: "Martin")
```

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

```
println( "${daniel.materias?.first()} es la primer materia de ${daniel.nombre}" )
```

O bien, utilizando los métodos Let-Run:

```
daniel.materias?.let { // La variable no es nula (it será el valor de la variable)  
    println( "${it.first()} es la primer materia de ${daniel.nombre}" )  
} ?: run { // La variable es nula  
    println( "${daniel.nombre} no está cursando materias" )  
}
```



## 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**.

```
class Alumno {  
  
    // Atributos (privados)  
    private val nombre: String  
    private var edad: Int  
    private val materias: Array<String>?  
  
    // Constructor  
    constructor(nombre: String, edad: Int, materias: Array<String>?) {  
        this.nombre = nombre  
        this.edad = edad  
        this.materias = materias  
    }  
  
    // Métodos  
    public fun estudiar() {  
        materias?.let { // La variable no es nula  
            for (materia in it) {  
                println("Estoy estudiando $materia")  
            }  
        } ?: run { // La variable es nula  
            println("No estoy estudiando")  
        }  
    }  
}
```



## 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.

```
class Estudiante(val nombre: String, var materias: Array<Materia>) {  
  
    enum class Materia {  
        ANALISIS,  
        ALGEBRA,  
        MAT_DISCRETA,  
        ALG02  
    }  
  
}
```

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:

```
MainActivity.kt x Alumno.kt x  
19 // Instancia de clase  
20 val guillermo = Estudiante(  
21     nombre: "Guillermo",  
22     arrayOf(Estudiante.Materia.ALG02, Estudiante.Materia.ALGEBRA)  
23 )
```



- Propiedades: *name* y *ordinal*

```
enum class Direccion {  
    NORTE, SUR, ESTE, OESTE  
} // 0 1 2 3
```

```
private fun unaFuncion() {  
    var direccion: Direccion? = null  
    direccion = Direccion.OESTE  
  
    println("name: ${direccion.name}") // OESTE  
    println("ordinal: ${direccion.ordinal}") // 3  
}
```

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

```
enum class Direccion {  
    NORTE, SUR, ESTE, OESTE;  
  
    fun descripcion(): String {  
        return when (this) { // switch  
            NORTE -> "Santa Claus"  
            SUR -> "Fin del mundo"  
            ESTE -> "Sol naciente"  
            OESTE -> "Sol poniente"  
        }  
    }  
}
```

```
println( "descripcion: ${direccion.descripcion()}" )
```

- Parámetros (propiedades extras)

```
enum class Direccion (val ecuatorial: Boolean) {  
    NORTE( ecuatorial: false), SUR( ecuatorial: false), ESTE( ecuatorial: true), OESTE( ecuatorial: true);  
}  
  
println( "descripcion: ${direccion.ecuatorial}" )
```





## 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**.

```
class MyNestedAndInnerClass {  
  
    // Atributo privado de la class  
    private val one = 1  
  
    // Clase anidada  
    class MyNestedClass {  
        fun suma (valor1: Int, valor2: Int) : Int {  
            return valor1 + valor2 + one  
        }  
    }  
  
    // Clase interna  
    inner class MyInnerClass {  
        fun agregar (numero: Int) : Int {  
            return numero + one  
        }  
    }  
}
```

No puede acceder a la variable privada externa

```
// Instancia de una clase anidada (nested)  
val myNestedClass = MyNestedAndInnerClass.MyNestedClass()  
val resultadoSuma = myNestedClass.suma( valor1: 10, valor2: 5) // -> 15  
  
// Instancia de una clase interna (inner)  
val myInnerClass = MyNestedAndInnerClass().MyInnerClass()  
val resultadoAgregado = myInnerClass.agregar( numero: 10) // -> 11
```



## 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.

```
// Clase padre
open class Persona(val nombre: String, var edad: Int) {

    fun trabajar() {
        println("$nombre está trabajando")
    }
}
```

```
// Clase hija
class Enfermero(nombre: String, edad: Int, var licenciado: Boolean): Persona(nombre, edad) {

    fun inyectar() {
        if(licenciado)
            println("Lic. $nombre está inyectando")
        else
            println("$nombre está inyectando")
    }
}
```

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

```
// Instancia de una clase hija
val guillermo = Enfermero( nombre: "Guillermo", edad: 27, licenciado: true)
guillermo.trabajar() // -> "Guillermo está trabajando"
guillermo.inyectar() // -> "Lic. Guillermo está inyectando"
```



- 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.

```
// Clase padre
open class Persona(val nombre: String, var edad: Int) {

    open fun trabajar() {
        println("$nombre está trabajando")
    }
}
```

```
// Clase hija
class Enfermero(nombre: String, edad: Int, var licenciado: Boolean): Persona(nombre, edad) {

    override fun trabajar() {
        if(licenciado)
            println("Lic. $nombre está inyectando")
        else
            println("$nombre está inyectando")
    }
}
```

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.

```
// Instancia de una clase hija
val guillermo = Enfermero( nombre: "Guillermo", edad: 27, licenciado: true)
guillermo.trabajar() // -> "Lic. Guillermo está inyectando"
```

- 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 hija
class Enfermero(nombre: String, edad: Int) {

    override fun trabajar() {
        super.trabajar()
    }
}
```

```
// Instancia de una clase hija
val guillermo = Enfermero( nombre: "Guillermo", edad: 27, licenciado: true)
guillermo.trabajar() // -> "Guillermo está trabajando"
```



## Clase abstracta

No define una implementación, sino un comportamiento.

```
// Clase abstracta incluye open
abstract class Trabajar {

    // Método abstracto no definido
    abstract fun goToWork()

    // Método definido
    fun renunciar() {
        println("Renuncio!")
    }
}
```

```
// Clase hija (hereda de una clase abstracta)
open class Persona(val nombre: String, var edad: Int): Trabajar() {

    open fun trabajar() {
        println("$nombre está trabajando")
    }

    /* Método que obligatoriamente debe ser
       implementado de la clase padre abstracta */
    override fun goToWork() {
        println("$nombre está yendo al trabajo")
    }
}
```



## 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**.

```
interface Juego {  
  
    val nombreJuego: String  
  
    // Función sin implementación (abstract)  
    fun jugar ()  
  
    fun streamear () {  
        println("Estoy haciendo stream del juego $nombreJuego")  
    }  
  
}
```

```
// Clase abstracta incluye open  
abstract class Trabajar {  
  
    // Método abstracto no definido  
    abstract fun goToWork()  
  
}
```

```
// Clase hija (hereda de una clase abstracta y una interface)  
open class Persona(val nombre: String, var edad: Int): Trabajar(), Juego {  
  
    // Método propio  
    open fun trabajar() {  
        println("$nombre está trabajando")  
    }  
  
    // Método de la clase abstracta Trabajar  
    override fun goToWork() {  
        println("$nombre está yendo al trabajo")  
    }  
  
    /***** INTERFACE *****/  
  
    // Atributo de la interface Juego  
    override val nombreJuego: String  
        get() = "Among Us"  
  
    // Método de la interface Juego  
    override fun jugar() {  
        println("$nombre está jugando")  
    }  
  
    /*****/
```



## 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.

```
data class Trabajador(val nombre: String, val edad: Int, val trabajo: String) {  
  
    var trabajoAnterior: String = "" // Requiere un valor por defecto  
    ...  
  
    // Se pueden crear funciones ligadas al tratamiento de los datos  
}
```

```
// Instancia de una DataClass  
val guillermo = Trabajador( nombre: "Guillermo", edad: 27, trabajo: "Enfermero")  
val carla = Trabajador( nombre: "Carla", edad: 34, trabajo: "Enfermera")  
  
carla.trabajoAnterior = "Supervisora"
```

Algunos de los métodos por defecto de una DataClass son:

- equals      Permite comparar si una DataClass es igual a otra.

```
if (guillermo.equals(carla) || guillermo == carla) {  
    println("Son iguales")  
}
```

- toString      Transforma en texto los datos almacenados en el constructor principal.

```
guillermo.toString() // Trabajador(nombre=Guillermo, edad=27, trabajo=Enfermero)
```

- copy      Permite copiar una DataClass en otra, como parámetro se puede alterar alguno de los atributos que contenga.

```
// Copiando alterando la edad  
val guillermo2 = guillermo.copy(edad = 28)
```

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

```
// componentN  
val (nombre, edad) = guillermo  
println(nombre) // "Guillermo"  
println(edad) // 27
```

*clave , valor*

```
var myMap: Map<String, Int> = mapOf()
```



## 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

```
// Sin type alias
val myMap: MutableMap<Int, ArrayList<String>> = mutableMapOf()
myMap[1] = arrayListOf("Guillermo", "Hernandez")
myMap[2] = arrayListOf("Enel", "3030")
```

Requiere escribir todo el tipo de dato del Map

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

```
import ...

typealias MyMapList = MutableMap<Int, ArrayList<String>>

class MainActivity : ComponentActivity() {...}
```

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

```
// Con type alias
val myMap: MyMapList = mutableMapOf()
myMap[1] = arrayListOf("Guillermo", "Hernandez")
myMap[2] = arrayListOf("Enel", "3030")
```

Clave

Valor





## Funciones

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

```
typealias MyFuncion = (Int, String, ArrayList<String>) -> Boolean
```

## Clases anidadas

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

```
typealias MyNestedClass = MyNestedAndInnerClass.MyNestedClass
```

```
// Instancia de una clase anidada (nested)  
val myNestedClass = MyNestedAndInnerClass.MyNestedClass()
```

```
// Instancia de una clase anidada (nested)  
val myNestedClass = MyNestedClass()
```



## 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.

```
data class Trabajador(val nombre: String, val edad: Int, val trabajo: String) {  
  
    var trabajoAnterior: String = "" // Requiere un valor por defecto  
}
```

```
val guillermo = Trabajador( nombre: "Guillermo", edad: 27, trabajo: "Enfermero")  
val (nombre, edad, trabajo) = guillermo  
println("$nombre, $edad, $trabajo") // Guillermo, 27, Enfermero
```

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

```
val guillermo = Trabajador( nombre: "Guillermo", edad: 27, trabajo: "Enfermero")  
println(guillermo.compo)
```

component1()	-> nombre	String
component2()	-> edad	Int
component3()	-> trabajo	String

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

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

```
private fun miTrabajador(): Trabajador {  
    return Trabajador( nombre: "Guillermo", edad: 27, trabajo: "Enfermero")  
}
```

→

```
val (nombre, edad, trabajo) = miTrabajador()  
println("$nombre, $edad, $trabajo") // Guillermo, 27, Enfermero
```



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

```
val (nombre, edad) = guillermo
println("$nombre, $edad")
// Guillermo, 27
```

```
val (nombre, _, trabajo) = guillermo
println("$nombre, $trabajo")
// Guillermo, Enfermero
```

Se pueden desestructurar **Maps**

```
val myMap = mapOf(1 to "Guille", 2 to "Nico", 3 to "Hern")
for (elemento in myMap) {
    println("${elemento.key}, ${elemento.value}")
    println("${elemento.component1()}, ${elemento.component2()}")
}
```


son equivalentes



1, Guille
2, Nico
3, Hern

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

```
val myMap = mapOf(1 to "Guille", 2 to "Nico", 3 to "Hern")
for ((clave, valor) in myMap) {
    println("$clave, $valor")
}
```



1, Guille
2, Nico
3, Hern



## 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:

```
activity.kt x Extensiones.kt x

package com.example.contador

import java.text.SimpleDateFormat
import java.util.*

// Función aplicada a la class que se quiere extender y cuyo fichero base es inmodificable
fun Date.customFormat() : String {
    val formatter = SimpleDateFormat( pattern: "yyyy-MM-dd'T'HH:mm:ssZZZ", Locale.getDefault())
    return formatter.format( date: this) // this hace referencia a Date
}

/*
Se añadió una nueva función a la class Date (por defecto de java.util)
llamada "customFormat" la cual establece un formato de día y hora específico
que nosotros queremos utilizar con frecuencia
*/
*/
```

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

```
val myDate = java.util.Date()
println(myDate.customFormat())
```

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

```
fun Date?.customFormat() : String {
```

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

```
val Date?.formatSize : Int
    get() = this.customFormat().length
```

```
val myDate = java.util.Date()
println(myDate.customFormat())
println(myDate.formatSize)
```

Si customFormat pudiera ser nulo, se puede definir así:

```
val Date?.formatSize : Int
    get() = this.customFormat()?.length ?: 0 // valor por defecto
```



## 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.

```
val lista = arrayListOf<Int>(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

val listaFiltrada = lista.filter { it: Int
    it > 5
}

// listaFiltrada: 6, 7, 8, 9, 10
```

```
val listaFiltrada = lista.filter { miElemento ->
    miElemento > 5
}
```

equivalentes

Se pueden agregar excepciones mediante *return@filter*

```
val lista = arrayListOf<Int>(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

val listaFiltrada = lista.filter { it: Int

    if(it == 1) {
        return@filter true
    }

    it > 5 ^filter
}

// listaFiltrada: 1, 6, 7, 8, 9, 10
```



Una forma de crear funciones lambda personalizadas es la siguiente:

```
private fun cualquierFuncion() {  
  
    val funcionSuma = fun(a: Int, b: Int): Int {  
        return a + b  
    }  
  
    val funcionProducto = fun(a: Int, b: Int): Int = a * b  
  
    miFuncionOperaciones(a: 5, b: 10, funcionSuma) // ----> 15  
    miFuncionOperaciones(a: 5, b: 10, funcionProducto) // -> 50  
  
}  
  
private fun miFuncionOperaciones(a: Int, b: Int, unaFuncion: (Int, Int) -> Int): Int {  
    return unaFuncion(a, b)  
}
```

distintas sintaxis

funcion dentro de otra

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

```
miFuncionOperaciones(a: 5, b: 10) { a, b ->  
    a - b // --> -5  
}
```



## 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:

```
private fun unaFuncion() {  
  
    myAsyncFunction( nombre: "Guille") { // como sólo tiene 1 parámetro, se puede usar it  
        println(it)  
    }  
  
}  
  
private fun myAsyncFunction(nombre: String, enviarMensaje: (String) -> Unit) {  
    val myNewString = "Hello $nombre"  
    enviarMensaje(myNewString)  
}
```

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

```
private fun unaFuncion() {  
  
    myAsyncFunction( nombre: "Guille") { // como sólo tiene 1 parámetro, se puede usar it  
        println(it) // el mensaje se enviará 5 segundos después  
    }  
  
}  
  
private fun myAsyncFunction(nombre: String, enviarMensaje: (String) -> Unit) {  
    val myNewString = "Hello $nombre"  
  
    thread {  
        Thread.sleep( millis: 5000)  
        enviarMensaje(myNewString)  
    }  
}
```



## **Jetpack Compose**

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



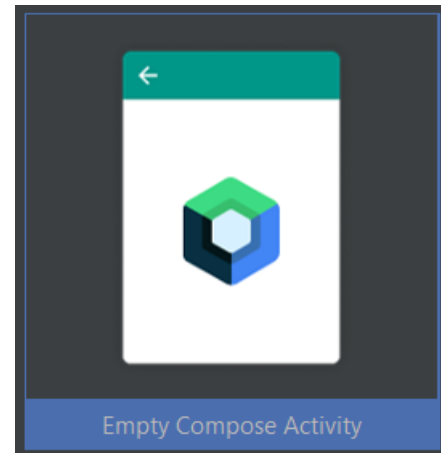




Para comenzar a utilizar Jetpack Compose, es necesario crear una nueva activity y seleccionar la que incluya la palabra “Compose”.

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

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



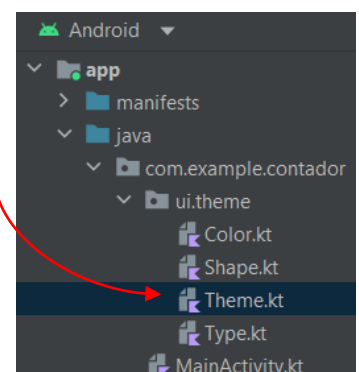
Una vez creada la activity, se debe empezar a crear una “vista” en **onCreate**, dentro de un bloque de código denominado **setContent**.

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            ContadorTheme {
                // A surface container using the 'background' color from the theme
                Surface(modifier = Modifier.fillMaxSize(), color = MaterialTheme.colors.background) {
                    Greeting(name: "Android")
                }
            }
        }
    }
}

@Composable
fun Greeting(name: String) {
    Text(text = "Hello $name!")
}
```

función composable  
(permite modificar la vista)

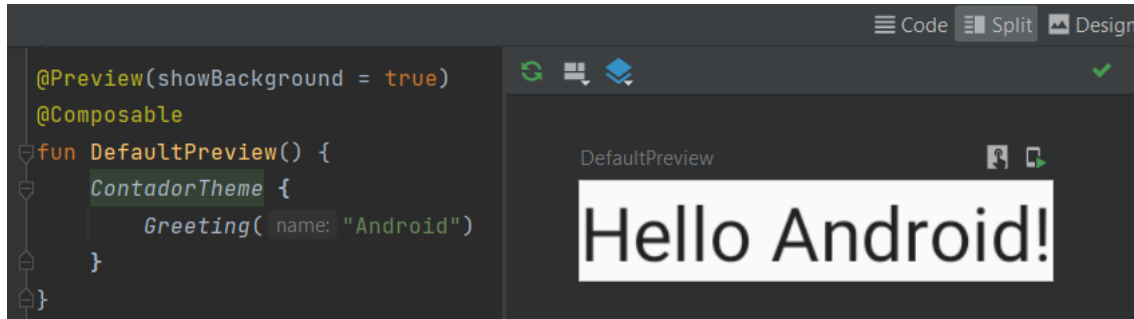
Donde el Theme está predefinido dentro de Theme.kt





## 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:

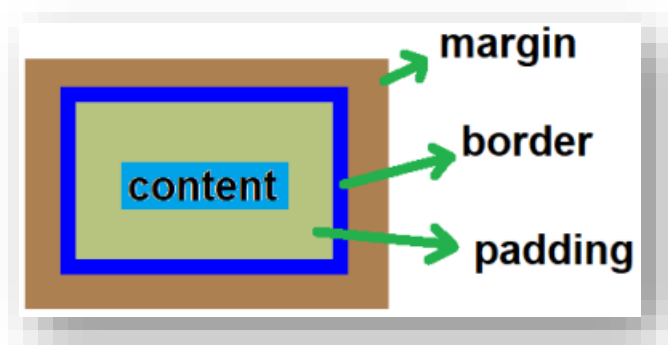
Si se utiliza **showSystemUi = true**, se visualizará en modo pantalla real.

```
@Preview()
@Preview(uiMode = Configuration.UI_MODE_NIGHT_YES)
@Composable
fun DefaultPreview() {
    ContadorTheme {
        Componentes()
    }
}
```

## 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

```
Text( text: "Soy un texto", modifier = Modifier.padding(top = 8.dp))
```

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

```
Text( text: "Soy un texto",
      modifier = Modifier
        .padding(top = 8.dp)
        .border(1.dp, Color.White)
    )
```



## Material

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

- Text: permite escribir un texto (*TextView*)

```
Text( text: "Soy un texto",  
      color = Color.Red // Color del texto  
)
```

- Spacer: permite colocar un separador

```
Spacer(modifier = Modifier.height(16.dp)) // altura de 16dp
```

- Image: permite agregar una imagen

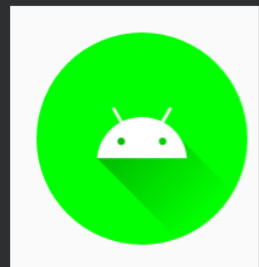
```
Image(  
    painterResource(R.drawable.ic_launcher_foreground),  
    contentDescription: "My imagen de prueba"  
)
```



La cual puede ser modificada con ciertos Modificadores:

```
@Composable  
fun MyImage() {  
    Image(  
        painterResource(R.drawable.ic_launcher_foreground),  
        contentDescription: "My imagen de prueba",  
        modifier = Modifier  
            .clip(CircleShape) // Recorte circular (debe ir primero)  
            .background(Color.Green) // Fondo verde  
            .size(64.dp) // Define el tamaño fijo proporcional  
    )  
}
```

DefaultPreview

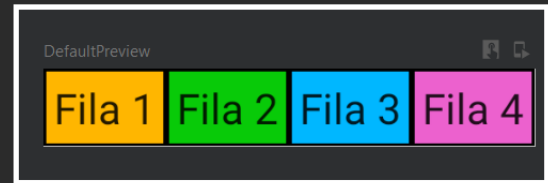




- Row y Column

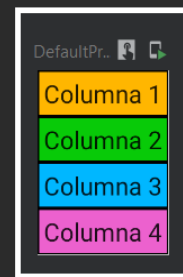
- Row

```
@Composable
fun Componentes() {
    Row() { this: RowScope
        Text( text: "Fila 1", modifier = colorAmarillo)
        Text( text: "Fila 2", modifier = colorVerde)
        Text( text: "Fila 3", modifier = colorCeleste)
        Text( text: "Fila 4", modifier = colorRosa)
    }
}
```



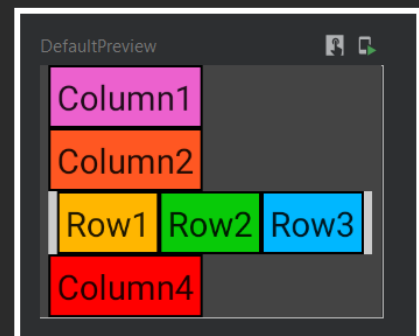
- Column

```
@Composable
fun Componentes() {
    Column() { this: ColumnScope
        Text( text: "Columna 1", modifier = colorAmarillo)
        Text( text: "Columna 2", modifier = colorVerde)
        Text( text: "Columna 3", modifier = colorCeleste)
        Text( text: "Columna 4", modifier = colorRosa)
    }
}
```



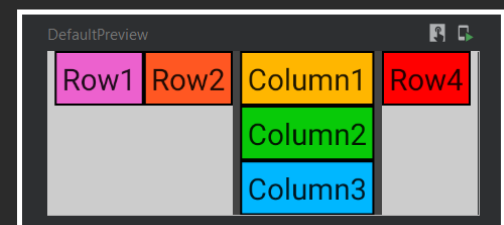
- Row en Column

```
@Composable
fun Componentes() {
    Column(modifier = fondoOscuro) { this: ColumnScope
        Text( text: "Column1", modifier = colorRosa)
        Text( text: "Column2", modifier = colorNaranja)
        Row(modifier = fondoClaro) { this: RowScope
            Text( text: "Row1", modifier = colorAmarillo)
            Text( text: "Row2", modifier = colorVerde)
            Text( text: "Row3", modifier = colorCeleste)
        }
        Text( text: "Column4", colorRojo)
    }
}
```



- Column en Row

```
@Composable
fun Componentes() {
    Row(modifier = fondoClaro) { this: RowScope
        Text( text: "Row1", modifier = colorRosa)
        Text( text: "Row2", modifier = colorNaranja)
        Column(modifier = fondoOscuro) { this: ColumnScope
            Text( text: "Column1", modifier = colorAmarillo)
            Text( text: "Column2", modifier = colorVerde)
            Text( text: "Column3", modifier = colorCeleste)
        }
        Text( text: "Row4", colorRojo)
    }
}
```



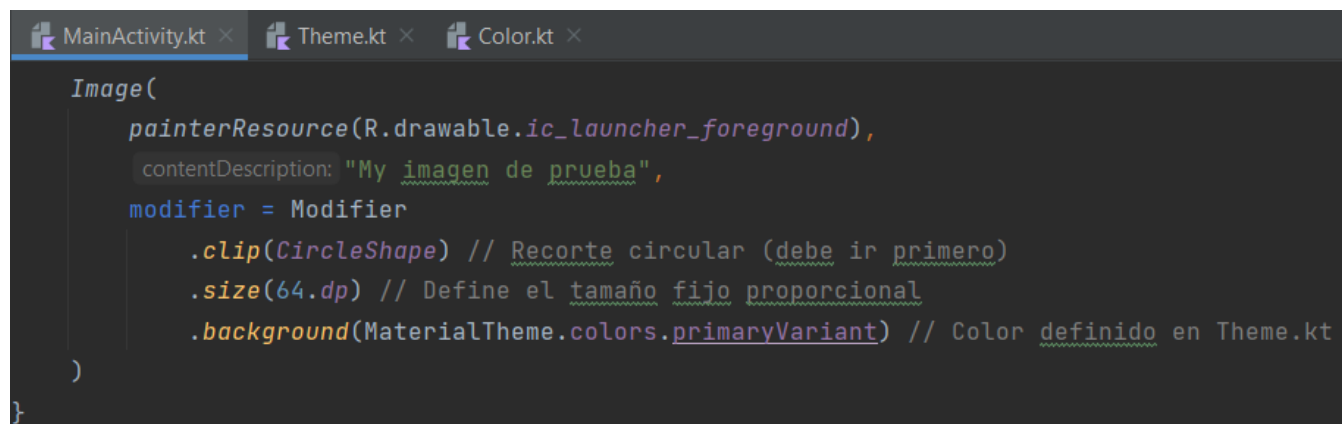


## Tema (theme)

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

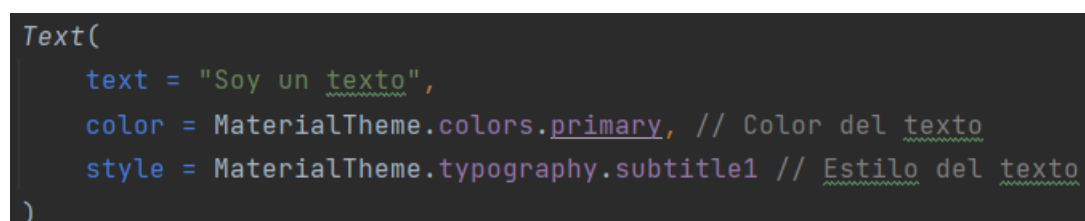


Hecho esto, se podrán utilizar los colores definidos dentro de nuestro Theme:



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



