

CURSO DESARROLLO DE APLICACIONES MÓVILES ANDROID TRAINEE

Módulo 4.

Desarrollo de aplicaciones móviles Android Kotlin



Partner
laboratoria >

TALENTO
DIGITAL
INTELIGENCIA
HUMANA

Temario

Unidad 1

- a) Kotlin**
- b) Integrar Kotlin en un proyecto Android**

Unidad 1

Kotlin – Un lenguaje llamado Kotlin

Muchos desarrolladores de software piensan que Kotlin es un lenguaje muy nuevo, esto debido a que se popularizó el año 2017 cuando fue anunciado por Google como un lenguaje más para desarrollar aplicaciones Android, junto con Java y C++.

Pero Kotlin no nace el 2017, este lenguaje fue anunciado el año 2011 por los desarrolladores de JetBrains como un proyecto que corre sobre la máquina virtual de Java y que puede interoperar con código Java.

Kotlin

El proyecto empezó el año 2010 debido a una necesidad del equipo de IntelliJ IDEA, este IDE está desarrollado en Java, por lo que necesitaban algo mejor que Java y que fuera igual de rápido, existía un posible candidato SCALA, pero se descarto debido a que era mucho para lo que ellos necesitaban, por lo que al no existir nada que solucione sus problemas con Java, crearon su propio lenguaje de programación.

Hoy en día Kotlin es un lenguaje que permite trabajar sobre la JVM, Java Script y Kotlin Native. Hoy ya Kotlin es uno de los lenguajes más queridos por los desarrolladores según StackOverFlow y Google ya anunció este año a Kotlin-first como el lenguaje más utilizado para desarrollar aplicaciones en Android.

Kotlin

Hoy en día Kotlin es un lenguaje que permite trabajar sobre la JVM, Java Script y Kotlin Native. Hoy ya Kotlin es uno de los lenguajes más queridos por los desarrolladores según StackOverFlow y Google ya anunció este año a Kotlin-first como el lenguaje más utilizado para desarrollar aplicaciones en Android.

Variables en Kotlin

- Conocer las variables en Kotlin
- Declarar variables en Kotlin
- Declarar variables por inferencia en Kotlin
- Aprender formato de caracteres en Kotlin

Introducción

Todo el código necesita de variables, en este capítulo aprenderemos cómo funcionan las variables en Kotlin y como se deben declarar para ser utilizadas, aprenderemos sobre el uso de variables por inferencia en Kotlin y cómo usar una variable mutable o inmutable.

Declarar variables en Kotlin

Para poder declarar una variable en Kotlin, necesitamos crear antes una función, para los ejemplos utilizaremos la función “**main**”, que profundizaremos en el capítulo de funciones. Además de crear una función, necesitamos saber 3 cosas:

1. Tipo de variable (“var” o “val”)
2. Nombre de la variable
3. Tipo de valor o clase(Numérico, Cadena de Caracteres, Lógico, etc.)

Ejemplo

```
fun main() {  
    val numero : Int  
    var variable : TipoVariable  
}
```

Variables por inferencia en Kotlin

En Kotlin podemos omitir la declaración del tipo de valor, y esto lo hacemos inicializando la variable con un valor determinado.

```
fun main() {  
    val nombre = "Goro Daimon"  
    val edad = 22  
}
```

Acá podemos ver como omitimos el tipo de valor en la declaración de la variable “nombre” y “edad”.

Diferencias entre val y var

En Kotlin existen dos tipos de variables, las variables mutables y las variables inmutables. **Las variables inmutables** se declaran como “*val*” y contienen un valor que “**No**” puede ser modificado. **Las variables mutables** se declaran como “*var*” y estas variables “**Si**” pueden ser modificadas.

```
fun main() {  
    val numeroFijo = 10  
    var numeroVariable = 10  
    numeroFijo = 12  
}
```

String Template en Kotlin

Una de las mejoras importantes que trae Kotlin como lenguaje de programación, es la facilidad de trabajar con variables String, a esto se les conoce como “**String Template**”. Cuando estemos utilizando una variable String podemos utilizar el signo “\$” para concatenar valores.

```
fun main() {  
    val x = "valor string"  
    val template = "Concatenando **** $x **** valores"  
    println(template)  
}
```

String Template en Kotlin

También podemos llamar a una función del lenguaje o de una variable utilizando el signo “\$” seguido del uso de llaves “{ }”. Dentro de estas llaves podemos hacer la llamada a la función que necesitemos.

```
fun main() {  
    val x = "valor string"  
    val template = "Tiene ${x.count()} caracteres"  
    println(template)  
}
```

Variables básicas en Kotlin

- Variables enteras.
- Variables de coma flotante.
- Variables booleanas.
- Variables String

Variables enteras

En Kotlin existen 4 tipos de variables básicas para trabajar con números, y estos son: Byte, Short, Int y Long y el uso de estas variables dependerá del número que nosotros queramos almacenar en nuestra variable.

Tipo	Tamaño	Rango de valor
Byte	8 bits	-128 a 127
Short	16 bits	-32768 a 32767
Int	32 bits	-2147483648 a 2147483647
Long	64 bits	-9223372036854775808 a 9223372036854775807

Variables de coma flotante

En Kotlin existen 2 tipos de variables básicas para trabajar con valores de coma flotante y estos son: Float y Double, uso de estas variables dependerá del número que nosotros queramos almacenar en nuestra variable.

Para el caso de las variables Float, al momento de asignar valor a la variable, esta debe venir acompañada con una letra “f” o “F”:

```
fun main() {  
    val x = 19.5f  
    var y : Double = 20.5  
}
```

Variables booleanas

Las variables booleanas almacenan sólo dos valores, “verdadero” o “falso”

```
fun main() {  
    val x = true  
    val y : Boolean  
    y = false  
}
```

Cadenas de caracteres

En Kotlin existen dos tipos básicos para trabajar con cadenas de caracteres. En el caso de que se necesite almacenar un carácter único utilizamos el tipo Char. En el caso de que se necesite almacenar una cadena de caracteres, utilizamos String.

```
fun main() {
    val x = 'a'
    val y = "Cadena de caracteres"
    var vocal : Char
    val nombre : String
    vocal = 'u'
    nombre = "Chizuru Kagura"
}
```

Tipos de funciones

- Tipos de funciones
- Métodos Java en relación a las Funciones Kotlin
- Funciones en Kotlin
- Conocer el uso de Void, void y Unit en Kotlin
- Funciones con parámetros

Tipos de funciones - Introducción

Hasta ahora hemos trabajado con la función “main”, en este capítulo aprenderemos a separar nuestro código en funciones Kotlin, veremos los tipos de funciones que tiene Kotlin y cómo podemos utilizar nuestras funciones en distintas partes de nuestro proyecto.

Tipos de funciones - Introducción

Tipos de Funciones

Para poder comprender mejor los tipos de funciones en Kotlin, comparemos las funciones de Kotlin con las funciones de Java.

En Java, a las funciones se llaman métodos, y estos se clasifican en dos grupos, métodos sin tipo de retorno y métodos con tipo de retorno.

Método sin tipo de retorno

Son todos los métodos declarados como “void”, y no necesitan un return dentro de sus llaves { }.

```
public void imprimirNombreCompleto(){  
    System.out.println("Sanji Vinsmoke");  
}
```

Método con tipo de retorno

Son todos los métodos que declaran una clase en antes de especificar su nombre y devuelve un objeto de la clase declarada. Estos métodos tienen la obligación de llevar un return entre sus llaves { }.

```
public String getNombreCompleto(){  
    return "Monkey D. Luffy";  
}
```

En el ejemplo anterior clase declarada es “**String**” y el valor del objeto retornado es “**Monkey D. Luffy**”.

Funciones en Kotlin

En Kotlin, solo existen los métodos con tipo de retorno y a estos métodos se les llama funciones. Y estos se declaran de la siguiente forma:

```
fun getNombreCompleto(): String{  
    return "Monkey D. Luffy"  
}
```

Funciones en Kotlin

- “fun”: Indica que es una función Kotlin.
- “getNombreCompleto”: es el nombre de la función.
- “ () ”: Indica los parámetros que tiene la función, en este caso es vacío o sin parámetro.“
- “ : String” : indica el tipo de retorno que tendrá la función, en este caso un “String”.
- “ return “Monkey D. Luffy” “ : Es el objeto retornado por la función.

Repasando métodos void de Java

Ya se mencionó que Kotlin solo trabaja con funciones que retornan valores, entonces: ¿Qué pasa con los métodos void que tiene Java y existen en Kotlin ?

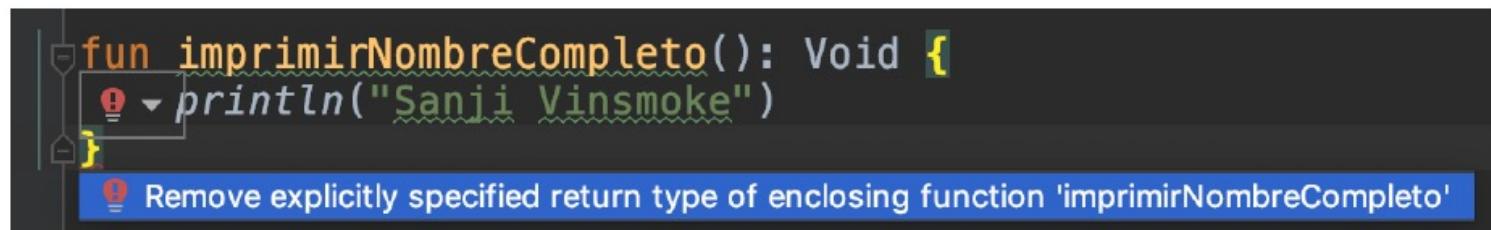
Para responder a esta pregunta, entendamos bien lo que es un método void.“void” es una palabra reservada del lenguaje Java, y lo utilizamos para indicar que un método no retorna valor.“Void” con mayúscula, es el nombre de una clase Java que se encuentra en el package “java.lang”, es un wrapper que transforma un valor primitivo “void” a un objeto Java.

Repasando métodos void de Java

El uso de la clase “Void” no es muy popular, esta clase existe desde que Java soporto el uso de objetos genéricos “Generics”. Todo valor primitivo Java tiene un equivalente en una clase Java, por eso existe la clase Integer, Boolean, Double, Float, etc.

Funciones void en Kotlin

En Kotlin no existen la palabra reservada “void”, sólo existe la clase “Void”, pero si le indicamos a una función que retornará un objeto tipo “Void” nos llevaremos una sorpresa.



A screenshot of an IDE showing a code editor with the following Kotlin code:

```
fun imprimirNombreCompleto(): Void {  
    println("Sanji Vinsmoke")  
}
```

The code editor highlights the word "Void" in orange. A tooltip or status bar at the bottom displays the message: "Remove explicitly specified return type of enclosing function 'imprimirNombreCompleto'".

Me dice que remueva explícitamente el “Void” especificado como tipo de retorno.

¿Cómo declaro una función void en Kotlin?

La clase Unit es el equivalente en Kotlin a la variable primitiva “void” de Java

```
fun imprimirNombreCompleto(): Unit {  
    println("Sanji Vinsmoke")  
}
```

Si declaramos una función que retorna un tipo “Unit”, automáticamente Kotlin detecta que esa función no retorna nada (como los void en Java).

Funciones con parámetros

La forma de declarar los parámetros en una función Kotlin es indicando el nombre de la variable, seguido por “ : ” donde indicamos el tipo de dato de esta variable.

En caso de querer agregar más de un parámetro, este se indica por medio de una “ , ” Ej:

```
fun imprimirNombreCompleto(nombres: String, apellidos: String, edad: Int){  
    println("Nombres: $nombres")  
    println("Apellidos: $apellidos")  
    print("Edad: $edad")  
}
```

Función Main en Kotlin

Una de las funciones más importantes a nivel de programación es el método main. El método main es tan importante, que si desarrollamos una aplicación y no creamos el método main, no podemos ejecutar nuestra aplicación. En Kotlin a los métodos les llamamos funciones, y la función main no es la excepción. A diferencia de Java, la función main de Kotlin es mucho más sencilla de declarar

Ejemplo:

```
fun main(){  
    println("Hola Mundo, soy la función Main y no soy estática.")  
}
```

Función Main en Kotlin

Otro tema importante a destacar, es que no necesitamos que nuestra función main se encuentre dentro de una clase, es decir, solo creando un archivo Kotlin ya es suficiente para poder declarar nuestra función main.

Clases en Kotlin

Tarde o temprano necesitaremos utilizar algo más que las clases básicas de Kotlin (Int, String, Boolean, etc.) , ahí es donde entran nuestras clases. Las clases son plantillas que nos permiten crear nuestros propios objetos, definiendo nosotros las propiedades y funciones.

Nuestra primera clase en Kotlin :

```
class Persona{}
```

Clases en Kotlin

Lo primero que extrañamos en nuestra clase, es el uso de la palabra reservada “public”, en Kotlin todas las clases son “public” por defecto, por lo tanto declarar una clase como public es redundante en Kotlin.



Data Class en Kotlin

Kotlin también puede trabajar con clases tipo P.O.J.O. (Plain Old Java Object) de forma muchos mas eficiente que en otros lenguajes, cuando necesites crear una clase con atributos, get y set. Data class es la solución a este problema.

P.O.J.O en Java

```
public class Persona {  
    private String nombre;  
    private String rut;  
    private int edad;  
  
    public Persona(String nombre, String rut, int edad) {  
        this.nombre = nombre;  
        this.rut = rut;  
        this.edad = edad;  
    }  
}
```

Data Class en Kotlin

P.O.J.O en Java

```
public String getNombre() {  
    return nombre;  
}  
  
public void setNombre(String nombre) {  
    this.nombre = nombre;  
}  
  
public String getRut() {  
    return rut;  
}
```

Data Class en Kotlin

P.O.J.O en Java

```
public void setRut(String rut) {  
    this.rut = rut;  
}  
  
public int getEdad() {  
    return edad;  
}  
  
public void setEdad(int edad) {  
    this.edad = edad;  
}
```

Data Class en Kotlin

Esta misma clase en Kotlin se puede desarrollar con un Data Class

```
data class Persona(val nombre: String, val rut: String, val edad: Int)
```

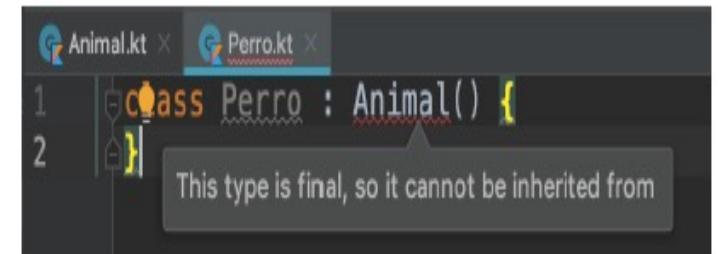
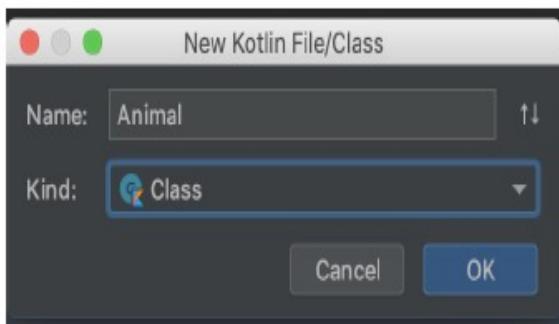
Data Class en Kotlin

Main.kt

```
fun main(){
    val persona = Persona("Juan", "1-9", 39)
    println("Nombre Persona = ${persona.nombre}")
    println("Rut Persona = ${persona.rut}")
    println("Edad Persona = ${persona.edad}")
}
```

Open Class en Kotlin

En Kotlin por defecto las clases son finales, es decir, no se puede heredar de dicha clase. Para poder utilizar la Herencia en Kotlin lo hacemos utilizando los ":" seguido del nombre de la clase padre y la llamada al constructor de esta clase.



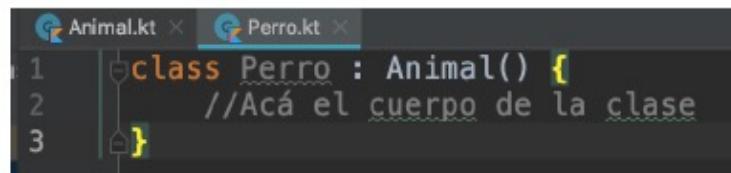
Open Class en Kotlin

Acá vemos como nuestra clase Animal por defecto es una clase “final”



```
Animal.kt × Perro.kt ×
1 open class Animal {
2     //Acá el cuerpo de la clase
3 }
```

Al declarar nuestra clase como open, esta se puede heredar en otra clase:



```
Animal.kt × Perro.kt ×
1 class Perro : Animal() {
2     //Acá el cuerpo de la clase
3 }
```

Ya no tenemos el error de que nuestra clase es “final”.

Sealed Class en Kotlin

Las “Sealed Class” o clases selladas, se utilizan para definir una jerarquía de clases restringidas, esto con el fin de utilizar valores predeterminados. Funciona de forma similar a las enumeraciones, con la ventaja de que las clases selladas son mucho más flexibles que las enumeraciones y lo veremos en el siguiente ejemplo:

Creamos una enumeración llamada TipoDeMensaje.kt

```
enum class TipoDeMensaje(val tipo : String) {  
    ALERTA("Esto es una alerta"),  
    TOAST("Esto es una tostada"),  
    DIALOGO("Esto es un cuadro de diálogo"),  
    VENTANA("Esto es una ventana")  
}  
  
fun main(){  
    val tipoDeMensaje = TipoDeMensaje.ALERTA  
    println(tipoDeMensaje.tipo)  
}
```

Sealed Class en Kotlin

El mismo ejemplo, lo podemos hacer con clases selladas Creamos una clase sealed llamada TipoDeMensaje.kt

```
sealed class TipoDeMensaje
class MensajeAlerta(val mensaje: String) : TipoDeMensaje()
class MensajeToast(val mensaje: String) : TipoDeMensaje()
class MensajeDialog(val mensaje: String) : TipoDeMensaje()
class MensajeVentana(val mensaje: String) : TipoDeMensaje()

fun main() {
    var alerta = MensajeAlerta("Esto es una alerta")
    val toast = MensajeToast("Esto es una tostada")
    val dialog = MensajeDialog("Esto es un cuadro de diálogo")
    val ventana = MensajeVentana("Esto es una ventana")
    println(alerta.mensaje)
    println(toast.mensaje)
    println(dialog.mensaje)
    println(ventana.mensaje)
}
```

Sealed Class en Kotlin

Acá queda en evidencia la ventaja de las clases selladas ya que si quisiéramos agregar un nuevo tipo de mensaje, no necesitamos modificar la clase principal, lo único que debemos hacer es crear una nueva clase que herede de nuestra clase sellada.

Constructores en Kotlin

- Constructores en Kotlin
- Constructor sin parámetros
- Constructor con parámetros
- Sobrecarga de constructores

Constructores en Kotlin

El uso de los constructores es esencial para la programación orientada a objetos, sin constructores nuestras clases no pueden ser instanciadas. En este capítulo veremos qué es un constructor, como se declaran un constructor con y sin parámetros y aprenderemos cómo trabajar con sobrecarga de constructores en una clase Kotlin.

Constructores en Kotlin

Kotlin, al igual que Java necesita crear constructores en sus clases para que estas puedan crear objetos, es tan importante el constructor en una clase, que el mismo lenguaje se encarga de crear un constructor sin parámetros en caso de que nosotros no lo creamos en nuestra clase.

Constructores sin parámetros

El constructor sin parámetros, es el constructor por defecto que tiene una clase, como ya mencionamos, si nosotros creamos una clase Kotlin, podremos llamar al constructor de esa clase por medio de la siguiente instrucción:

```
class Perro {  
}  
  
fun main(){  
    val perroSalchicha = Perro()  
}
```

Constructores sin parámetros

A diferencia de Java, en Kotlin no utilizamos la palabra reservada “new” para llamar al constructor. En Kotlin para llamar al constructor se hace por medio del signo de igualdad “=” seguido del nombre del constructor que nosotros queramos llamar. En este caso se llama al constructor sin parámetros que crea por defecto Kotlin “= Perro()”.

Constructores con parámetros

Cuando nosotros declaramos un constructor con parámetros, esto se debe hacer de la siguiente manera:

```
class Perro(val nombreParam: String, val edadParam: Int) {  
    val nombre = nombreParam  
    val edad = edadParam  
}  
  
fun main(){  
    val perroSalchicha = Perro("Toky", 2)  
    print("El nombre del perro es : ${perroSalchicha.nombre} y la edad es: ${perroSalchicha.edad}.")  
}
```

Constructores con parámetros

En este caso para llamar al constructor con parámetros, llamamos al nombre de la clase,, Seguido de los parámetros de su constructor. “= Perro(“Toky”, 2)“.

Sobre carga de constructores

Cuando necesitemos una clase con más de un constructor a esto se le llama sobrecarga de constructores y esto se hace de la siguiente manera:

```
class Perro {  
    val nombre: String  
    val edad: Int  
    constructor() {  
        nombre = "Sin Nombre"  
        edad = 0  
    }  
    constructor(nombreParam: String, edadParam: Int){  
        nombre = nombreParam  
        edad = edadParam  
    }  
}
```

Sobre carga de constructores

```
fun main() {  
    val perroSalchicha = Perro()  
    val perroBulldog = Perro("Firulais", 15)  
    println("El nombre del perro es : ${perroSalchicha.nombre} y la edad es: ${perroSalchicha.edad}.")  
    println("El nombre del perro es : ${perroBulldog.nombre} y la edad es: ${perroBulldog.edad}.")  
}
```

```
El nombre del perro es : Sin Nombre y la edad es: 0.  
El nombre del perro es : Firulais y la edad es: 15.
```

```
Process finished with exit code 0
```

Funciones del Lenguaje

- Funciones en archivos Kotlin.
- Funciones de extensión en Kotlin

Funciones del Lenguaje

En Kotlin además de crear funciones dentro de una clase, podemos crear funciones fuera de una clase e incluso podemos crear funciones de extensión para clases donde no tengamos acceso al código fuente.

Funciones en archivo Kotlin

En Kotlin al momento de crear una clase también podemos crear archivos, estos archivos Kotlin tienen la misma extensión que una clase “.kt”. En estos archivos podemos crear nuestras funciones kotlin sin la necesidad de crear una clase, hacer esto en Java es imposible, ya que todas las funciones deben estar dentro de una clase.

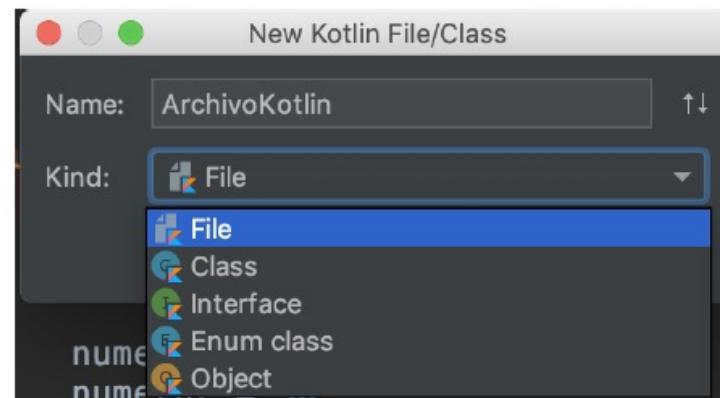
Botón derecho en carpeta “src” -> “New” -> “Kotlin File/Class”.

Funciones en archivo Kotlin

Botón derecho en carpeta “src” -> “New” -> “Kotlin File/Class”.



Crear nueva clase/archivo.



Crear archivo Kotlin.

Funciones en archivo Kotlin

ArchivoKotlin.kt

```
fun saludo(){  
    print("Hola")  
}  
  
fun despedida(){  
    print("Chao")  
}
```

En este ejemplo podemos ver un archivo Kotlin que contiene 2 funciones, estas funciones pueden ser llamadas desde cualquier parte de nuestro proyecto

Main.kt

```
fun main(){  
    saludo() //  
    despedida()  
}
```

Funciones en archivo Kotlin

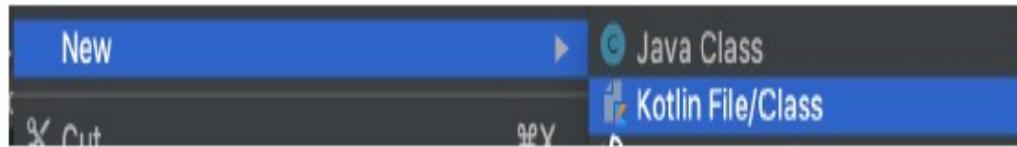
En este ejemplo podemos ver cómo se utilizan 2 funciones sin la necesidad de crear ninguna clase, estos métodos son propios del lenguaje Kotlin.

Los métodos declarados en archivos de tipo Kotlin se consideran como métodos () y gracias a esto podemos utilizarlos en cualquier clase.

Funciones en extensión Kotlin

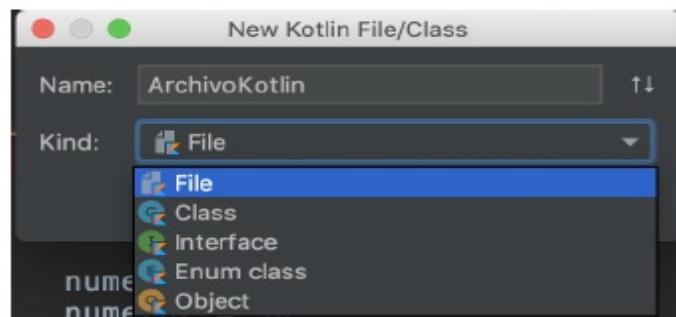
Las funciones de extensión permiten al desarrollador agregar una función a una clase específica sin la necesidad de modificar el código fuente de la clase, estas funciones se crean de la misma manera que las funciones de los archivos Kotlin, pero se le debe indicar la clase donde queramos extender de estas funciones.

Ejemplo: Botón derecho en carpeta “src” -> “New” -> “Kotlin File/Class”.



Funciones en extensión Kotlin

Acá creamos un archivo Kotlin:



ArchivoKotlin.kt

```
fun String.saludo(){
    print("Hola")
}
fun String.despedida(){
    print("Chao")
}
```

Funciones en extensión Kotlin

En este ejemplo podemos ver cómo declaramos una función llamada “String.saludo” cuando al nombre de nuestra función le anteponemos el nombre de una clase (en este caso la clase String) significa que cualquier objeto de la clase String puede utilizar nuestra función “saludo” y “despedida”

Main.kt

```
▶  fun main(){
    val variable = "Hola" //declaramos una variable String
    variable.saludo() //Utilizamos nuestra función de extensión "saludo"
    variable.despedida() //Utilizamos nuestra función de extensión "despedida"
}
```

Funciones en extensión Kotlin

En este ejemplo podemos ver cómo se utilizan 2 funciones en una variable de tipo String sin la necesidad de modificar la clase String, este tipo de función es propios del lenguaje Kotlin

```
fun main(){
    val variable = "Hola"
    variable.saludo()
    variable.despedida()
}
Press ^ to choose the selected (or first) suggestion and insert a dot afterwards >>
```

Las funciones de extensión declarados en Archivos Kotlin se consideran como métodos () para variables String y gracias a esto podemos utilizarlos en cualquier objeto que sea de este tipo.

Unidad 1

Integrar Kotlin a un proyecto en Android

- Integrar Kotlin en Android Studio.
- Aprender a crear clases Kotlin.
- Convertir Clases Java a Kotlin

Integrar Kotlin a un proyecto en Android

Kotlin es un lenguaje relativamente nuevo que contiene muchas diferencias respecto a Java, diferencias que vienen de los lenguajes más modernos como Python o Swift. A pesar de todo esto, Kotlin fue pensado para funcionar en conjunto con Java y esto ha sido parte fundamental para hacer que se convierta en el lenguaje oficial de Android, ya que tiene muy buena compatibilidad con Java.

Integrar Kotlin a un proyecto en Android

Por lo tanto si queremos aprender a desarrollar en Android, será necesario entender cómo se integra con Android Studio, ya que éste es el IDE oficial de Android. Nuestro objetivo ahora será descubrir como Android Studio nos puede facilitar el desarrollo Android usando kotlin, descubrir cómo iniciar un proyecto, como crear archivos kotlin y finalmente entender qué herramientas nos puede entregar este IDE para facilitar nuestra migración de Java a Kotlin.

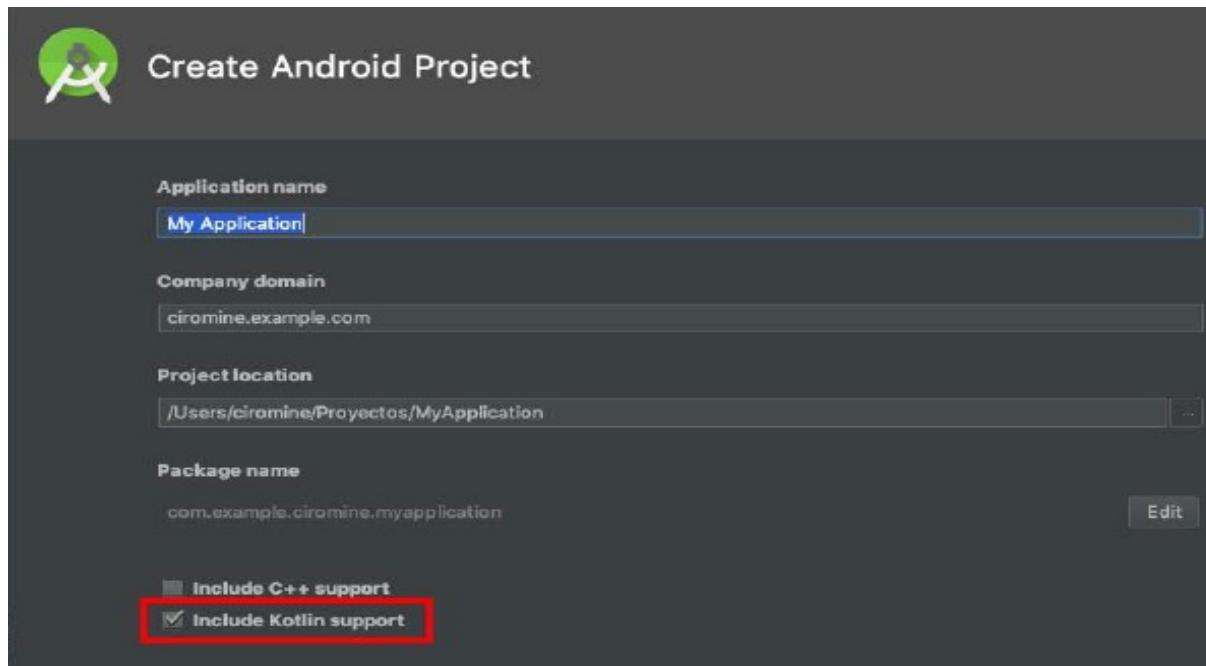
Integrar Kotlin en Android Studio

Para comenzar a trabajar con Kotlin en Android, debemos partir usando Android Studio, que es el IDE oficial de google para desarrollar en Android, este puede ser descargado desde la página oficial de Google para Windows, Linux o Mac, debe ser si la versión 3 o superior, ya que desde ahí en adelante se le hizo soporte oficial a Kotlin. Si tenemos un proyecto antiguo que no tiene soporte para kotlin también se le puede agregar, pero eso es algo que no profundizaremos ahora.

Integrar Kotlin en Android Studio

Para comenzar un proyecto con Kotlin, comenzamos un nuevo proyecto en Android con Android Studio, pero en el primer paso que nos aparece, donde seteamos el nombre de la app, es necesario marcar la opción Include Kotlin Support, como se destaca en la siguiente imagen.

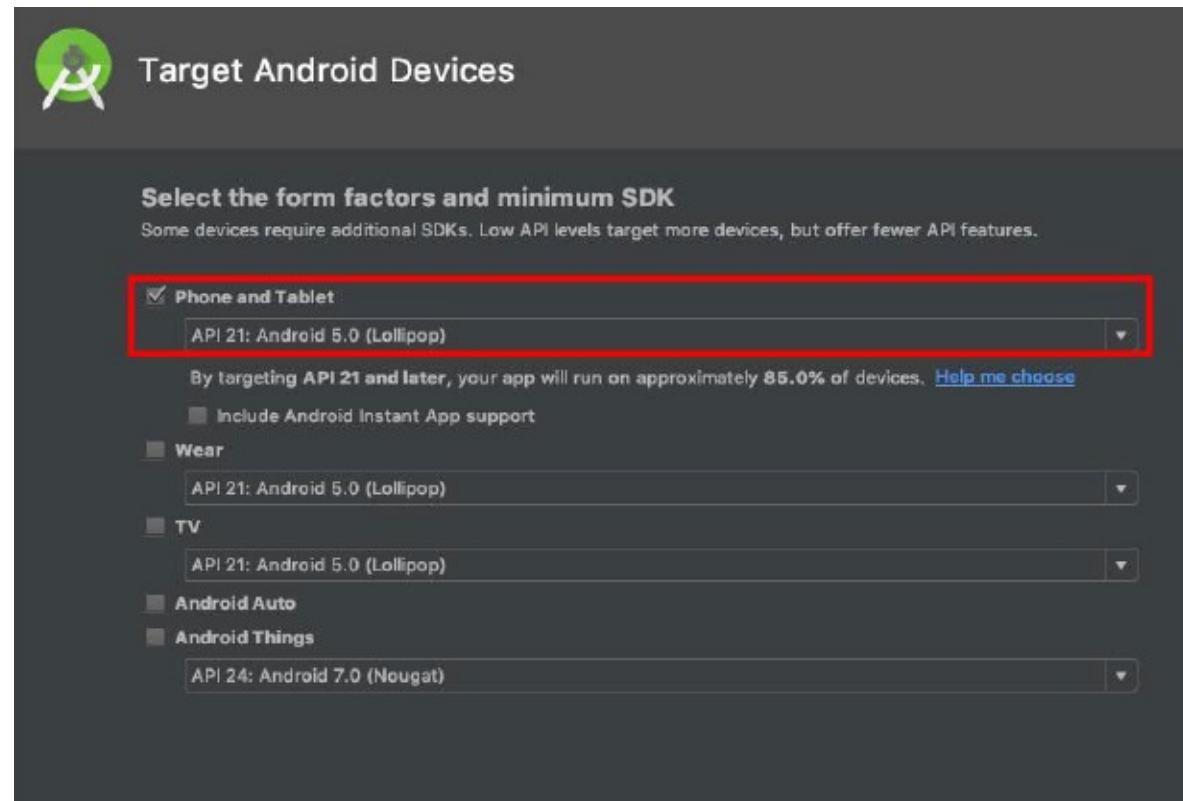
Integrar Kotlin en Android Studio



Integrar Kotlin en Android Studio

Ya con esta opción seleccionada, nuestro nuevo proyecto partirá con nuestras clases creadas en con la extensión .kt, que es el formato usado por Kotlin para sus archivos. Pero sigamos creando nuestro nuevo proyecto, al Application name le pondremos DesafioLatam y ahora haremos click en Next.

Integrar Kotlin en Android Studio



Integrar Kotlin en Android Studio

Acá vemos que nos indica que queremos que soporte nuestro proyecto, si va a ser para Teléfono y Tablet, o también tendrá compatibilidad con Wear (relojes por ejemplo), TV, Android Auto o Android Things. En este caso solo haremos compatible con teléfono y seleccionaremos el API 21 Android 5.0 (Lollipop), ¿qué significa esto? Significa que nuestro proyecto soportará el api de desarrollo como mínimo hasta **Android 5.0**.

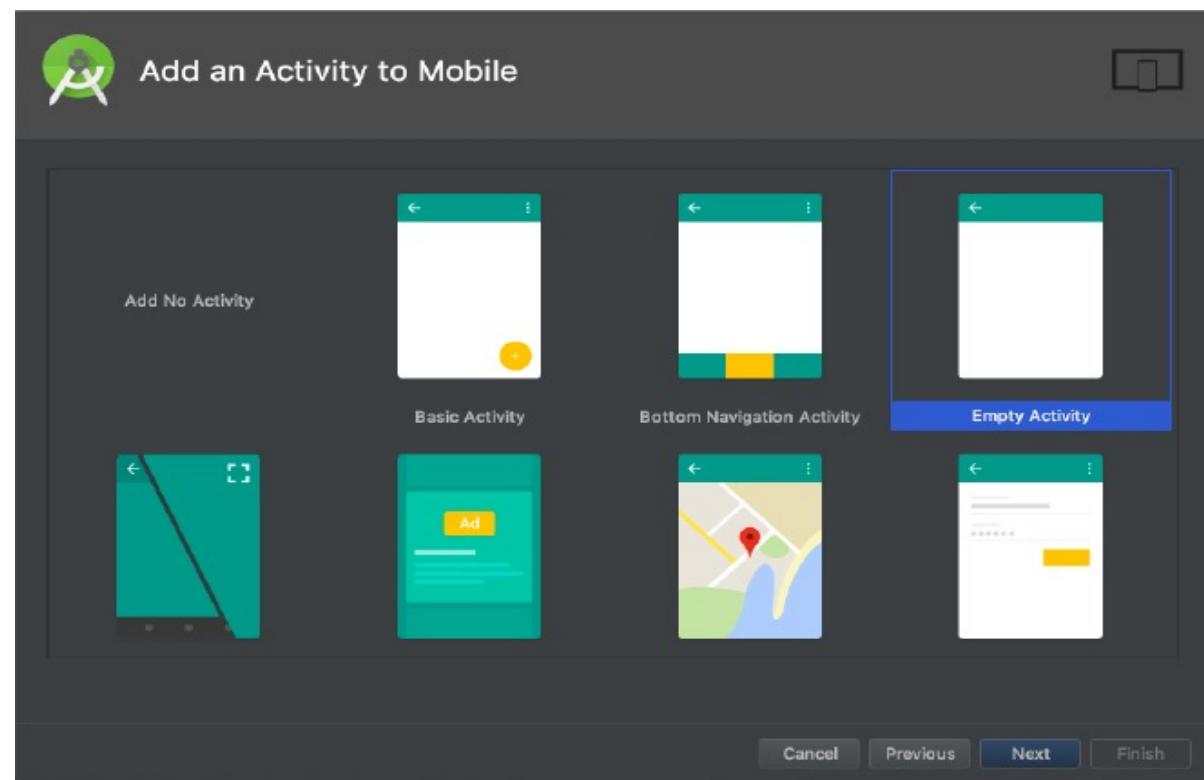
Integrar Kotlin en Android Studio

O sea que la mínima versión que soportaremos será Android 5.0, ya que con esto como el mismo IDE lo indica soportamos el 85% de los dispositivos actuales del mercado. Ahora podemos seguir con el botón Next.

Agregar actividad al proyecto

Escogemos la opción Empty Activity, el cual traerá una Activity vacía con un textview al medio que contiene una frase de “Hello World”. Luego podemos hacer click en Next.

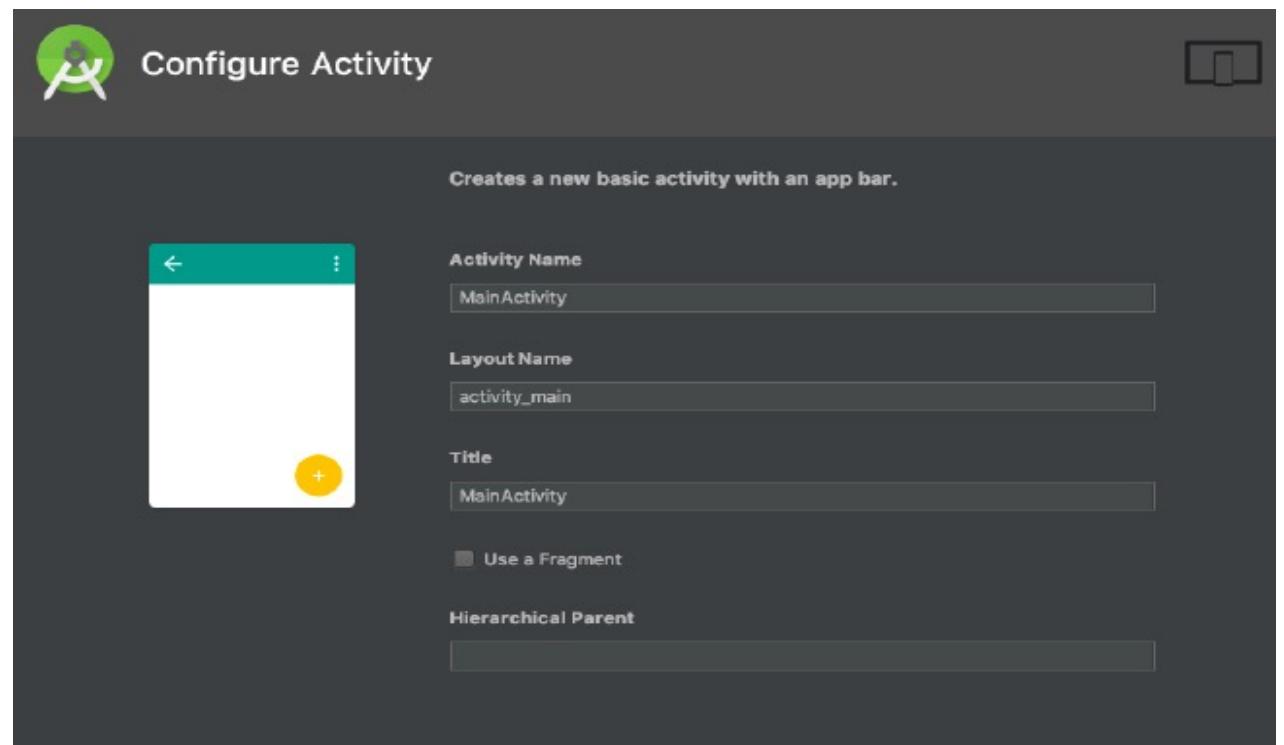
Agregar actividad al proyecto



Agregar actividad al proyecto

Acá en el último paso le podemos cambiar el nombre a esa activity, al layout y el título, en este caso nosotros, no cambiaremos nada de esto, ya podemos apretar el botón Finish y comenzar a codear. Nuestra Activity en Kotlin debería verse de la siguiente forma.

Agregar actividad al proyecto

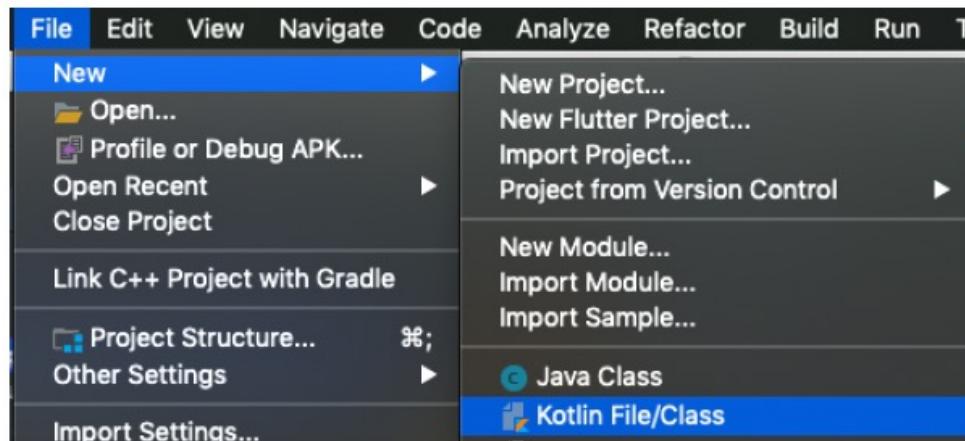


Agregar actividad al proyecto

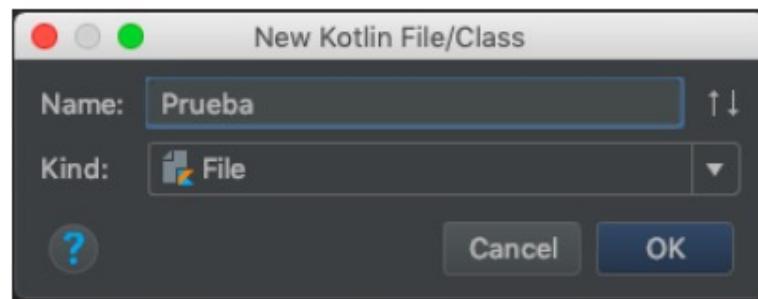
Si nos fijamos es como cualquier clase en Java, que no tiene nada de particular, solamente tiene la sintaxis de kotlin.

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

Crear clase desde Android Studio



Crear clase desde Android Studio



Crear archivo llamado Prueba.

Crear clase desde Android Studio

Luego le seteamos un nombre a nuestro archivo y listo, crearemos nuestro archivo kotlin para crear otra Activity o una clase con código para lo que queramos realmente. A diferencia de las clases de Java, los archivos de kotlin vienen vacíos, por lo que es necesario nosotros escribir todo el código, por ejemplo, el archivo recién creado quedaría así.

```
package com.example.ciromine.desafiolatam
```

Crear clase desde Android Studio

Ahora si quisiéramos que la clase Prueba tuviera por ejemplo un método Suma que retornará un int, deberíamos hacer algo así.

```
package com.example.ciromine.desafiolatam

class Prueba {

    fun suma(): Int {
        return 1+1
    }
}
```

Así podemos generar más clases de Kotlin y podremos seguir generando código para realizar todas las funciones e ideas que nosotros queramos.

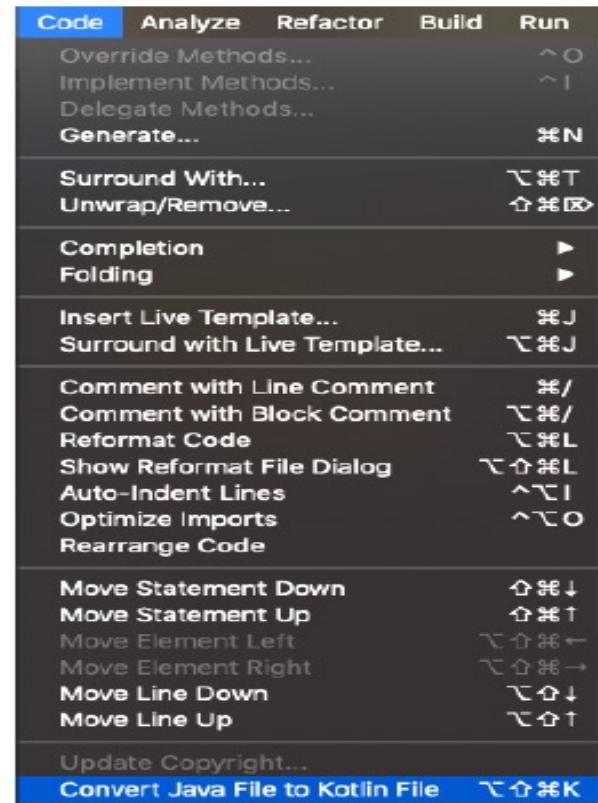
Convertir clases de Java a Kotlin

Aprovechemos de usar otra funcionalidad muy útil de Android Studio que nos servirá para ir transformando clases Java a Kotlin de manera más sencilla.

```
public class Calculadora {  
  
    public Calculadora() {}  
  
    public int suma(int a, int b) {  
        return a + b;  
    }  
}
```

Convertir clases de Java a Kotlin

Ahora si nos vamos a la siguiente opción en el Menú.



Convertir clases de Java a Kotlin

Una vez seleccionada esta opción, el mismo IDE hará la conversión automática de la clase Java a Kotlin, el resultado sería el siguiente.

```
class Calculadora {

    fun suma(a: Int, b: Int): Int {
        return a + b
    }
}
```

Convertir clases de Java a Kotlin

De esta manera, tenemos una herramienta que nos puede ayudar a convertir nuestro código Java en Kotlin, hay que destacar que a veces la conversión no resulta perfecta y quedan algunos pequeños errores de sintaxis, que uno mismo debe corregir, aunque eso solo ocurre con clases más complejas.

Kotlin y Java

Kotlin siempre fue pensado como un lenguaje cien por ciento compatible con Java y esto permite que sean interoperables, de manera que podremos trabajar con ambos lenguajes a la vez, algo que nos permitirá ir migrando lentamente nuestro código. Literalmente podemos ir migrando clase a clase y evitar tener que generar una gran migración de código obligadamente al cambiarnos a Kotlin.

Kotlin y Java

Otra ventaja de esto es: si nos imaginamos un proyecto real, el cual ya tenemos escrito en Java y comenzamos a migrar a kotlin, pero tenemos ciertas libraries que son antiguas pero no podemos prescindir de ellas, no tendríamos problema, ya que aunque nuestro código esté usando kotlin, podría comunicarse sin problemas con cualquier library aunque este hecha en Java.

Interoperabilidad Kotlin y Java

¿Qué es la interoperabilidad?

El Instituto de Ingenieros Eléctricos y Electrónicos (IEEE) define interoperabilidad como la habilidad de dos o más sistemas o componentes para intercambiar información y utilizar la información intercambiada.

En este caso cuando nos referimos interoperabilidad, nos referimos a la forma en que las clases Java y Kotlin pueden comunicarse, como si fueran un solo lenguaje de programación.

Interoperabilidad Kotlin y Java

Kotlin fue concebido como un lenguaje compatible con Java, esto le permite ser interoperable, o sea, que podemos tener clases definidas en kotlin y estás puedes ser llamadas o utilizadas desde clases hechas en java y viceversa.

Interoperabilidad Kotlin y Java

A continuación veremos un ejemplo de interoperabilidad entre ambos lenguajes. Crearemos una app simple con un layout, para que podamos ver nuestro resultado en ejecución, por lo tanto crearemos una app simple donde llamemos a un método de una clase java.

Interoperabilidad Kotlin y Java

Primero crearemos la vista, supongamos que creamos el ejemplo básico de un Activity en blanco con el siguiente layout, que contiene solo un textview, pero que se le agregó un **layout_constraintVertical_bias="0.04000002"** para que no esté centrado en la vista, si no este verticalmente más pegado hacia la parte superior de la pantalla.

Interoperabilidad Kotlin y Java

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    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.example.ciromine.desafiolatam.MainActivity">

    <TextView
        android:id="@+id/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"
        app:layout_constraintVertical_bias="0.04000002" />

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

Interoperabilidad Kotlin y Java

Es un layout simple con un textview, ahora tendremos el siguiente código en la activity.

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

Interoperabilidad Kotlin y Java

Y finalmente tendremos una clase en Java llamada Calculadora, que por temas de simplicidad para el ejemplo solo tendrá un método suma que retornada la suma de 2 enteros.

```
public class Calculadora {  
  
    public Calculadora() {}  
  
    public int suma(int a, int b) {  
        return a + b;  
    }  
}
```

Interoperabilidad Kotlin y Java

Ahora para mostrar la interoperabilidad de Kotlin y Java, llamaremos a esta clase en Java desde nuestra activity en kotlin, y modificaremos el TextView.

```
class MainActivity : AppCompatActivity() {

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

        val textView = findViewById<TextView>(R.id.textview) as TextView

        val calculadora = Calculadora()

        textView.text = "Tu resultado es: " + calculadora.suma(1,1)
    }
}
```

Interoperabilidad Kotlin y Java

Resultado al correr el proyecto



De esta manera, hemos demostrado en un ejemplo como funciona la interoperabilidad que hay entre estos 2 lenguajes. Esta posibilidad nos da una herramienta muy poderosa, nos puede permitir trabajar con bibliotecas antiguas escritas en Java o incluso, poder ir migrando nuestro código a kotlin de a poco.

Ventajas de usar Kotlin

- Utilizar sentencia When
- Usar Null Safety
- Manejar los operadores del null safety.

Ventajas de usar Kotlin

Kotlin al ser un lenguaje moderno, tiene ciertas ventajas que vienen por defecto y que nos ayudarán a acelerar nuestro desarrollo de código y como siempre, a simplificar nuestro código. Así que en el siguiente capítulo profundizaremos cómo funcionan estas herramientas y cómo las podemos usar en nuestro camino del desarrollo de código.

Ventajas de usar Kotlin

Por ejemplo, en kotlin cuando creamos objetos POJO no es necesario definir todos los setters y getters de nuestras variables definidas, basta definir los parámetros en el constructor y kotlin, asume los métodos setters y getters automáticamente, de manera que no es necesario definir nada, veamos un ejemplo en código, para explicar más claro esto.

Ventajas de usar Kotlin

```
public class Cerveza {  
  
    String nombre;  
    String tipo;  
    float grados;  
  
    public Cerveza(String nombre, String tipo, float grados) {  
        this.nombre = nombre;  
        this.tipo = tipo;  
        this.grados = grados;  
    }  
}
```

Ventajas de usar Kotlin

```
public String getNombre() {
    return nombre;
}

public void setNombre(String nombre) {
    this.nombre = nombre;
}

public String getTipo() {
    return tipo;
}

public void setTipo(String tipo) {
    this.tipo = tipo;
}
```

Ventajas de usar Kotlin

```
public float getGrados() {  
    return grados;  
}  
  
public void setGrados(float grados) {  
    this.grados = grados;  
}  
}
```

Este sería un POJO de ejemplo para una clase Cerveza, ahora veamos como Kotlin, reduce el código.

```
class Cerveza(var nombre: String, var tipo: String, var grados: Float)
```

When

When es una sentencia que nos ayuda a controlar el flujo dentro de nuestra aplicación, cuando tenemos un caso en el que tenemos que usar muchos if y else seguidos, puede volverse difícil de leer y generarse un código engorroso. Para estos casos, podemos usar when. La sentencia **when**, tiene la siguiente estructura:

```
when (valor){  
    //serie de condiciones  
}
```

When

Ahora veamos, de donde nace el when, veamos una sentencia un poco compleja con **if** y **else**.

```
if (numero == 0) {  
    println("numero invalido")  
} else if (numero == 1 || numero == 2) {  
    println("numero muy bajo")  
} else if (numero == 3) {  
    println("numero correcto")  
} else if (numero == 4) {  
    println("numero alto, pero aceptable")  
} else {  
    println("numero muy alto")  
}
```

When

Acá vemos un caso engorroso donde tenemos muchas condiciones posibles para una variable llamada numero, entonces ahora veamos como la sentencia when nos puede ayudar a simplificar el código.

```
when(numero) {  
    0 -> println("numero invalido")  
    1, 2 -> println("numero muy bajo")  
    3 -> println("numero correcto")  
    4 -> println("numero alto, pero aceptable")  
    else -> println("numero muy alto")  
}
```

When

Como podemos ver, con la función when se ve un código mucho más legible y simple, ponemos al inicio nuestra variable y luego podemos solo la condición que evaluamos y después de -> ponemos las acciones a hacer en esa condición. Además el resultado fuera más complejo que hacer un println como en este ejemplo, se pone entre **llaves { }** y podemos realizar toda la lógica que nosotros queramos dentro.

When

Ahora como dijimos antes, when nos permite muchas maneras de evaluar y muchas condiciones y variaciones posibles, las cuales veremos a continuación algunos ejemplos.

```
var result = when(number) {
    0 -> "numero invalido"
    1, 2 -> "numero muy bajo"
    3 -> "numero correcto"
    4 -> "numero alto, pero aceptable"
    else -> "numero muy alto"
}
// it prints when returned "Number too low"
println("Retorno \"$result\"")
```

When

Podemos hacer que se retorne el valor y asignarlo en una variable, para después hacer el `println` con la variable, claramente podríamos retornar otro tipo de variable y hacer otra cosa que no sea un `println`. Ahora veamos otro ejemplo sin números

```
val check = true

val result = when(check) {
    true -> println("it's true")
    false -> println("it's false")
}
```

When

Como podemos ver, también podríamos evaluar fácilmente variables booleanas de manera muy simple, otro ejemplo posible.

```
var result = when(number) {
    0 -> "numero invalido"
    1, 2 -> "numero muy bajo"
    3 -> "numero correcto"
    in 4..10 -> "numero alto, pero aceptable"
    else -> "numero muy alto"
}
```

When

Ahora nos basamos en el primer ejemplo que vimos, pero esta vez, agregamos en vez de una evaluación simple, un rango, como vemos donde dice `in 4..10`, quiere decir que cualquier número que esté entre la brecha de 4 y 10, entrará en esa condición.

Ahora veamos qué más podemos hacer con when.

```
when (x) {  
    is Int -> print("es un Entero")  
    is String -> print("Es un String")  
    is IntArray -> print("Es un arreglo de Enteros")  
}
```

When

Podemos evaluar el tipo de la variable y tomar una decisión respecto de la que sea, o sea si miramos la primera condición dice **is Int** o sea, que evalúa si la variable **x**, es del tipo **Int**. La segunda condición ve si es del tipo **String** y la última si es un **IntArray** o **arreglo de int**.

Null Safety

¿Qué es un error de null?

Null o nulo (en español) es literalmente un valor vacío, cuando iniciamos una variable en Java como por ejemplo un String y hacemos **String a**; pero no le asignamos un valor, por defecto esta variable **a** se le asigna un valor nulo, que sería equivalente a un valor vacío. Por eso cuando una variable es nula y tratamos de hacer alguna operación sobre ella, podemos obtener un **nullPointerException**, ya que esa variable no tiene un valor definido y por ende no podríamos calcular el largo del String por ejemplo.

Null Safety

Kotlin es un lenguaje de programación que evita el uso de **null**, esto se debe a que así se mejora la calidad y seguridad del software, porque los errores con variables que son nulas, ocurren en tiempo de ejecución y hacen que la aplicación se caiga completamente. Es por esto que kotlin, tiene una manera segura para trabajar con nulls, comenzaremos a explicar ahora en qué consiste.

Null Safety

Si nosotros intentamos de hacer lo siguiente, tendremos un problema de sintaxis, o sea que el IDE nos mostraría el siguiente código en rojo

```
//esto estaría malo ya que en Kotlin no podemos decir que una variable es igual a null
var palabra: String = "abc"
palabra = null
//Y esto también

var palabra: String = null //esto también estaría malo
```

Null Safety

Para poder hacer algo similar a esto tendríamos que hacer lo siguiente.

```
//esta es la manera correcta, ya que usamos el operador ? que nos permite asignar la variable igual a
null
var palabra: String? = "abc"
palabra = null

//Y esta también

var palabra: String? = null //esta es la manera correcta, ya que usamos el operador ?
```

Null Safety

La clave está en el operador ? que nos está indicando que este campo puede ser null, pero es porque nosotros estamos forzando esa situación, de esta manera somos conscientes de que esa variable es null bajo nuestro propio riesgo. Ahora que tenemos nuestra variable que podría ser null, debemos saber trabajar con ella para evitar errores al correr nuestro código, por ejemplo, si quisiéramos obtener el largo de ese String posiblemente null.

```
var palabra: String? = null  
palabra?.length
```

Null Safety

Tenemos que usar nuestro operador ?, si no, Kotlin no nos lanzará un error de sintaxis, ya que así uno prevé que puede ser null y solo en caso que no lo sea, se podrá llamar a **length**. Veamos un caso un poco más complejo.

```
class Perro {  
  
    var raza: Raza? = null  
  
    -  
    -  
    -  
}  
  
class Raza {  
  
    var nombre: String? = null  
  
    -  
    -  
}
```

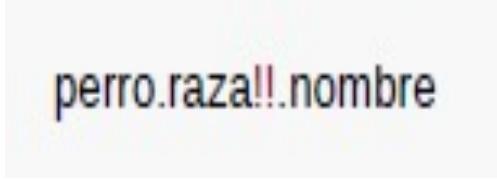
Null Safety

En este caso tenemos clases que están unidas, un Perro tiene raza y a la vez la Raza tiene un nombre y todos esas variables fueron definidas como posibles null. Entonces si quisiéramos obtener el nombre de la raza de un perro, deberíamos hacer lo siguiente.

```
var perro = Perro()  
perro?.raza?.nombre
```

Null Safety

Como podemos ver, en este caso donde hay una serie de relaciones entre clases con otra clases y parámetros, tendríamos que usar el operador `?` en cada nivel a medida que accedemos a la información. Ahora supongamos que en vez de usar `?` usamos otro operador



```
perro.raza!! . nombre
```

Null Safety

El operador !! lo que hace es **asegurar** de que la variable nunca va a ser **null**, o sea le decimos al compilador, que nosotros estamos cien por ciento seguros que esta variable no será null, en caso de que lo sea, obtendremos un **nullpointerException** y esto provocará que la app fallé mientras se ejecuta, por lo que hay que usarlo de manera muy responsable.

Null Safety

Este operador aunque no lo parezca es muy útil e importante, ya que dado que Kotlin es un lenguaje que no acepta variables nulas por defecto, hay veces que si definimos una variable con el operador ? indicando que podría ser null y después necesitamos agregar esa variable como parámetro de algún método, el IDE nos pedirá usar el operador !!, ya que necesita que aseguremos que no será null.

Utilidades de Kotlin

- Concatenar Strings
- Utilizar onClickListener
- Utilizar Funciones lambda
- Utilizar View Binding



Partner
laboratoria >



Utilidades de Kotlin

Kotlin contiene muchas utilidades, que nos ayudarán a hacer un código más simple y legible, ahora nuestro objetivo será aprender herramientas que tiene kotlin y nos ayudarán a mejorar nuestro desarrollo de software en Android.

Utilidades de Kotlin

Lo que veremos en este capítulo, nos ayudarán mucho a simplificar nuestro desarrollo en Android, aprenderemos cómo mejorar la concatenación de strings con una alternativa que nos da kotlin, además de aprender una manera para trabajar con los listener de manera muy simplificada a lo que estamos acostumbrados en Java y finalmente aprenderemos a trabajar de manera más simple con nuestros elementos visuales definidos en nuestro layout.

Concatenar String

Aprovechemos de usar una de las ventajas de Kotlin, para concatenar Strings, usaremos la misma clase MainActivity que usamos anteriormente, pero le haremos un pequeño cambio

Concatenar String

```
class MainActivity : AppCompatActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
  
        val textview = findViewById<TextView>(R.id.textview) as TextView  
  
        val calculadora = Calculadora()  
  
        val a = 1  
        val b = 2  
  
        textview.text = "Tu resultado "+a+" + "+b+" es: "+calculadora.suma(a, b)  
    }  
}
```

Concatenar String

Ahora haremos el siguiente cambio en la concatenación de String que vamos a setear al TextView para simplificar lo que se hacía normalmente en Java.

Concatenar String

```
class MainActivity : AppCompatActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
  
        val textview = findViewById<TextView>(R.id.textview) as TextView  
  
        val calculadora = Calculadora()  
  
        val a = 1  
        val b = 2  
  
        textview.text = "Tu resultado $a + $b es: ${calculadora.suma(a, b)}"  
    }  
}
```

Concatenar String

Si nos fijamos, en vez de agregar un símbolo +, para concatenar el String con una variable, , en Kotlin podemos hacer lo siguiente, que es dejar todo entre comillas y lo que sea variables de código, lo encerramos entre las llaves \${} si es un método (como cuando llamamos a calculadora.suma()), si no, le anteponemos un signo \$ solamente (como el caso de a y b).

Esto nos da una manera más simple para manejar la concatenación de Strings con variables, muy usada en otros lenguajes de programación como Ruby, a diferencia del antiguo uso del símbolo +, que varias veces se volvía bastante engoroso.

Listeners

Cuando en Android queremos darle acción al tocar un elemento usamos el Listener Onclick, el cual es una manera bastante verbosa (con mucho código) de darle acción a un botón, pero en Kotlin tenemos una alternativa que nos permite usar OnClickListener pero de manera mucho más legible.

Veamos un ejemplo a continuación, lo primero que haremos es agregar un botón al layout que teníamos antes.

Listeners

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    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.example.ciromine.desafiolatam.MainActivity">
```

Listeners

```
<TextView  
    android:id="@+id/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"  
    app:layout_constraintVertical_bias="0.04000002" />
```

Listeners

```
<Button  
    android:id="@+id/button"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_marginEnd="8dp"  
    android:layout_marginStart="8dp"  
    android:layout_marginTop="8dp"  
    android:text="Cambiar Texto"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintHorizontal_bias="0.521"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toBottomOf="@+id/textview" />  
  
</android.support.constraint.ConstraintLayout>
```

Listeners

Es un simple botón que no hace nada, ahora agregaremos el botón en la activity y le daremos una acción con el OnClickListener, que será que cambiará el texto de TextView cuando se apriete el botón.

Listeners

```
class MainActivity : AppCompatActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
  
        val textview = findViewById<TextView>(R.id.textview) as TextView  
  
        val button = findViewById<Button>(R.id.button) as Button  
  
        button.setOnClickListener(object : View.OnClickListener {  
            override fun onClick(v: View) {  
                textview.text = "Texto cambiado"  
            }  
        })  
    }  
}
```

Listeners

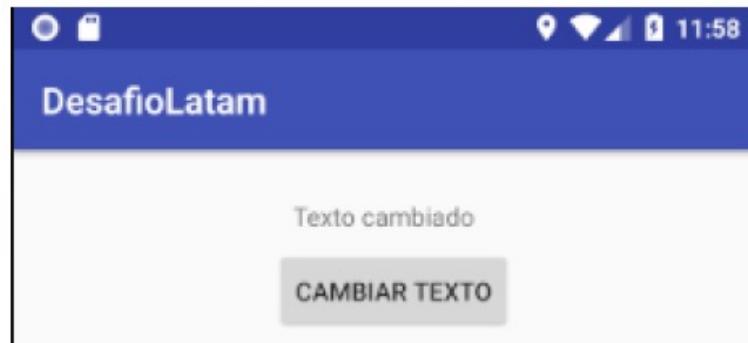
Acá podemos ver cómo asociamos el botón desde el layout con el `findViewById`, pero además usamos `onClickListener` para cambiar el `Textview` una vez que se haga click en el botón, esta es como se dijo antes la manera tradicional de hacer esto, ahora haremos uso de lambda para dejar el código más simple, algo maravilloso que nos permite Kotlin.

Listeners

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
    setContentView(R.layout.activity_main)  
  
    val textview = findViewById<TextView>(R.id.textview) as TextView  
  
    val button = findViewById<Button>(R.id.button) as Button  
  
    button.setOnClickListener { textview.text = "Texto cambiado" }  
}
```

Listeners

Listo, de esta manera kotlin nos permite usar una función lambda lo que simplifica el código, no hace falta llamar a View ni sobrescribir el método onClick, simplemente usamos la expresión setOnClickListener y con llaves y podemos definir todas las funciones que queramos que sucedan.



Función Lambda

Las funciones lambdas las podríamos definir como simplificada de pasar funcionalidad como parámetro sin la necesidad de crear funciones anónimas, esto nos permite hacer funciones complejas, pero con menos código, lo que hace más simple su lectura.

¿Qué es una función anónima?

Una función anónima es una función que se define, pero no se le asigna un nombre, o sea algo como así.

```
func () {  
    println('Esta función no tiene un nombre')  
}
```

Función Lambda

Este tipo de funciones, se usan mucho en lenguajes tipo javascript. Y son una manera excelente de reducir la cantidad de código necesaria para ejecutar ciertas tareas que son repetitivas y con mucho código en el desarrollo Android, como los listeners y los callbacks.

Una función lambda tiene la siguiente estructura.

$$\{ \text{x: Int, y: Int} \rightarrow \text{x + y} \}$$

Función Lambda

Las reglas de las éstas funciones son las siguiente:

- La expresión lambda debe estar delimitada por llaves.
- Si la expresión contiene cualquier parámetro, debes declararlo antes del símbolo `->`.
- Si estás trabajando con múltiples parámetros, debes separarlos con comas.
- El cuerpo de la función va luego del signo `->`.

Función Lambda

Las funciones lambda te permiten definir funciones anónimas y luego pasar estas funciones inmediatamente como una expresión. Entonces que no necesitas escribir la especificación de la función en una clase abstracta o interfaz. Como fue en el ejemplo del onClickListener donde primero teníamos el siguiente código:

```
button.setOnClickListener(object : View.OnClickListener {  
    override fun onClick(v: View) {  
        textView.text = "Texto cambiado"  
    }  
})
```

Función Lambda

Y gracias a la función lambda, logramos simplificarlo a esto:

```
button.setOnClickListener { view -> textView.text = "Texto cambiado" }
```

Sin embargo, dado que la variable view no se está usando kotlin, nos permite simplificar esta función aún más.

```
button.setOnClickListener { textView.text = "Texto cambiado" }
```

Función Lambda

como podemos ver, en este caso este método setOnClickListener {} de kotlin es una función lambda que nos solamente ejecutar toda la funcionalidad, en vez de tener que definir todos los métodos y vistas y recién ahí poder generar las acciones que nosotros queríamos.

View Binding

Kotlin tiene una de las herramientas más útiles que podrían existir para el desarrollo en Android, las Kotlin Android Extensions, que son una library que contiene herramientas muy útiles para Android, a continuación veremos cómo integrarlas al proyecto, en caso de que no esté integrado, aunque en la versión actual de Android Studio ya debería venir por defecto agregadas en los proyectos Android.

Agregando Kotlin Android Extensions

Lo primero que debemos hacer es agregar al archivo app/build.gradle lo siguiente

```
apply plugin: 'com.android.application'  
apply plugin: 'kotlin-android'  
apply plugin: 'kotlin-android-extensions'
```

Usando Kotlin Android Extensions

Esto generalmente se pone al principio del archivo. Ya con este pequeño cambio tenemos habilitadas las extensiones en Android, ahora veamos un ejemplo de cómo usarlas.

Supongamos tenemos el siguiente layout, que tiene 2 textview, uno arriba del otro y un botón debajo de los textview.

Usando Kotlin Android Extensions

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    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.example.ciromine.desafiolatam.MainActivity">
```

Usando Kotlin Android Extensions

```
<TextView  
    android:id="@+id/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"  
    app:layout_constraintVertical_bias="0.04000002" />
```

Usando Kotlin Android Extensions

```
<Button  
    android:id="@+id/button"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_marginEnd="8dp"  
    android:layout_marginStart="8dp"  
    android:layout_marginTop="8dp"  
    android:text="Boton"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintHorizontal_bias="0.521"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toBottomOf="@+id/textview2" />
```

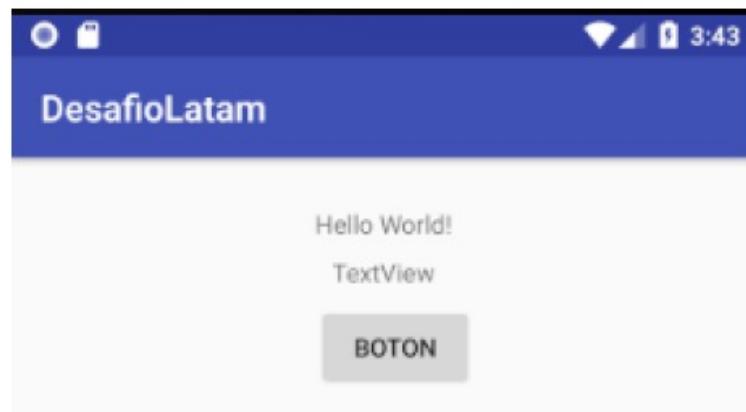
Usando Kotlin Android Extensions

```
<TextView  
    android:id="@+id/textview2"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_marginEnd="8dp"  
    android:layout_marginStart="8dp"  
    android:layout_marginTop="8dp"  
    android:text="TextView"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toBottomOf="@+id/textview" />  
  
</android.support.constraint.ConstraintLayout>
```

Usando Kotlin Android Extensions

```
<TextView  
    android:id="@+id/textview2"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_marginEnd="8dp"  
    android:layout_marginStart="8dp"  
    android:layout_marginTop="8dp"  
    android:text="TextView"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toBottomOf="@+id/textview" />  
  
</android.support.constraint.ConstraintLayout>
```

Usando Kotlin Android Extensions



Usando Kotlin Android Extensions

Ahora nuestro código en java sería como en el ejemplo anterior

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

    val textView = findViewById<TextView>(R.id.textview) as TextView
    val textView2 = findViewById<TextView>(R.id.textview2) as TextView

    val button = findViewById<Button>(R.id.button) as Button

    button.setOnClickListener {
        textView.text = "Texto cambiado"
        textView2.text = "Texto 2 cambiado también"
    }
}
```

Usando Kotlin Android Extensions

Ahora con si usamos las Kotlin Android Extensions, **no tendremos que usar más el `findViewById`**, o sea sería así.

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

    button.setOnClickListener {
        textView.text = "Texto cambiado"
        textView2.text = "Texto 2 cambiado también"
    }
}
```

Usando Kotlin Android Extensions

Como podemos ver, se simplifica mucho el código ¿no? Solo hay que llamar a los componentes con el mismo nombre que les pusimos de ID en el xml. Si nos fijamos en los IDs del xml

```
<TextView  
    android:id="@+id/textview"  
  
<Button  
    android:id="@+id/button"  
  
<TextView  
    android:id="@+id/textview2"
```

Usando Kotlin Android Extensions

Son los mismos nombres de las variables en el código java. Algo importante a destacar es que para que esto funcione, hay que **agregar un import**, que sería.

```
import kotlinx.android.synthetic.main.activity_main.*
```

Usando Kotlin Android Extensions

Si nos fijamos, al final del import está llamando a `activity_main` que es el nombre del archivo xml, donde tenemos el layout.

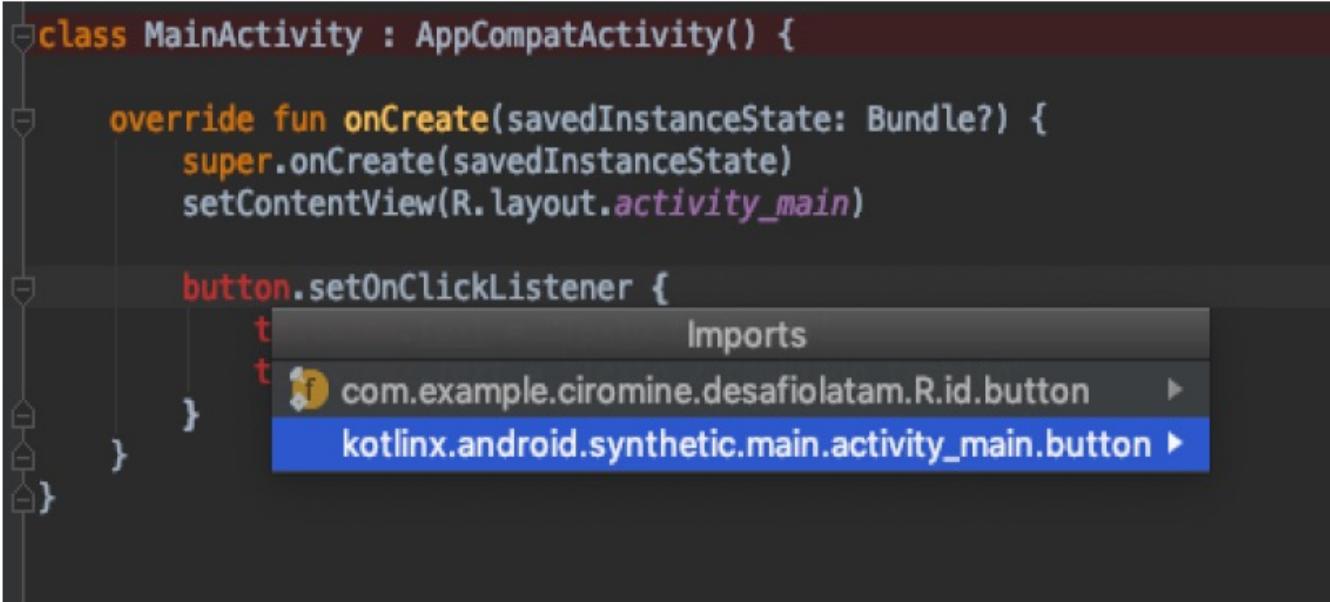
De todas formas una manera simple de tener el import, es sencillamente usando el IDE a nuestro favor, si escribimos el nombre de la variable y sobre ella usamos el shortcut del teclado para importar (que en **Mac** es **Opción + Enter** y en **Windows y Linux** es, **Alt + Intro**), el mismo IDE nos indica el import correcto, como se ve en las siguientes imágenes.

Usando Kotlin Android Extensions

A screenshot of an Android Studio code editor displaying Kotlin code for an Android application. The code is part of the `MainActivity` class, which extends `AppCompatActivity`. The `onCreate` method is overridden to set up UI components. A code completion tooltip is shown at line 8, suggesting the use of the `button` variable to call its `setOnClickListener` method. The tooltip also lists other methods like `textview` and `textview2`.

```
ard Rules for app) 6 class MainActivity : AppCompatActivity() {  
Properties) 7  
ttings) 8     override fun onCreate(savedInstanceState: Bundle?) {  
tion 9         super.onCreate(savedInstanceState)  
? com.example.ciromine.desafiolatam.R.id.button? (multiple choices...) ↴  
10     }  
11     button.setOnClickListener {  
12         textView.text = "Texto cambiado"  
13         textView2.text = "Texto 2 cambiado también"  
14     }  
15 }  
16 }  
17 }  
18 }
```

Usando Kotlin Android Extensions



A screenshot of an IDE showing a code editor with Java code. The code defines a class `MainActivity` that extends `AppCompatActivity`. Inside the `onCreate` method, there is a call to `button.setOnClickListener`. A code completion dropdown is open at this position, showing two suggestions: `com.example.ciromine.desafiolatam.R.id.button` and `kotlinx.android.synthetic.main.activity_main.button`. The second suggestion is highlighted in blue.

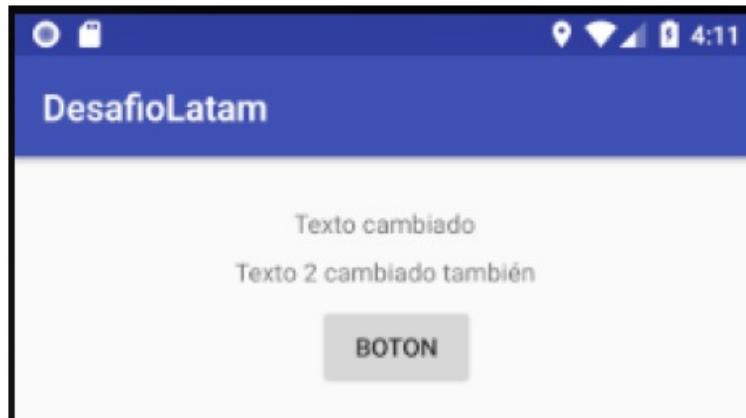
```
class MainActivity : AppCompatActivity() {

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

        button.setOnClickListener {
            t
            Imports
            t
            com.example.ciromine.desafiolatam.R.id.button >
            kotlinx.android.synthetic.main.activity_main.button >
        }
    }
}
```

Usando Kotlin Android Extensions

Con esta herramienta no tendremos problema para agregar el import correcto.



Profundizando Collections en Kotlin

- Usar Filters, Listas y Maps.
- Utilizar funciones de ordenamiento.
- Utilizar el patrón delegate.

Profundizando Collections en Kotlin

Kotlin nos provee varias funcionalidades para trabajar con colecciones, o sea con listas, maps, y otras estructuras de datos. Ya que generalmente siempre que uno trabaja con estos componentes, nos encontraremos con problemas y limitaciones como ordenar o filtrar el contenido de éstas, para poder hacer un uso correcto de la información con la que trabajaremos mientras programamos.

Profundizando Collections en Kotlin

En el siguiente capítulo veremos varias alternativas que tenemos para trabajar con estas estructuras de datos, para así no tener problemas al trabajar con listas o maps. Esto nos ayudará muchísimo para poder mostrar datos de forma más precisa y con menos código que en Java.

Filters

Los Filters son una herramienta muy poderosa que nos da Kotlin para trabajar con ciertas estructuras de datos, como Lists o Maps. A continuación veremos ejemplos para trabajar con Filters y ver cómo nos pueden ayudar para simplificar nuestro trabajo y código.

Cabe destacar que no todos los filtros funcionan para List o Maps, iremos usándolos según podamos.

List

Las listas son una estructura de datos en la cual tenemos un grupo de elemento consecutivos, que pueden ser de varios tipos de variables. Además de esto las listas tienen una serie de funciones como insertar elementos, eliminar, calcular su largo, entre muchos otros que nos ayudan a trabajar con los elementos de manera más óptima.

Para generar listas en kotlin, podemos usar el siguiente método que nos generará una lista con los valores que ingresemos.

```
listOf("one", "two", "three", "four")
```

List

Con este método podemos generar una lista, que en este caso contendría 4 strings, a la vez podemos llenarla de otras variables como int, boolean, float, etc. Como en el siguiente ejemplo.

```
listOf(null, 1, "two", 3.0, "four")
```

Las listas son usadas en Android por ejemplo cuando queremos usar un Spinner o un RecyclerView.

Maps

Los maps, son una estructura de datos, que nos permite tener elemento almacenados con “llave” y “valor”, en otros lenguajes se le conoce como diccionario y la gracia de estos elementos, que al tener una llave para cada elemento, es más rápido y simple acceder al valor que necesitamos.

Para generar maps en kotlin tenemos la siguiente función por ejemplo.

```
mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
```

Maps

Nos generará un map con las respectivas llaves y valores. Podemos agregar al igual que el listOf distintos tipos de variables.

```
mapOf("key1" to null, "key2" to 2.0, "key3" to "algo", "key11" to 11)
```

Con esto ya podemos comenzar a ver los distintos tipos de filtrados.

Filtrando por predicado

El filtrado por predicado es es sencillamente una función que nos devuelve otra lista, ya filtrada según la condición que nosotros queremos, veamos el siguiente ejemplo.

En este caso el predicado es una condición que se evaluará, como cuando hacemos un if y según ese evaluación tendremos un resultado con la lista filtrada.

```
val numeros = listOf("one", "two", "three", "four")
val largoMayorA3 = numbers.filter { it.length > 3 }
```

Filtrando por predicado

Primero expliquemos un poco el código anterior, `listOf` nos generará la lista que ingreamos. En este caso `largoMayorA3` contiene una lista con los elementos que tengan un largo mayor a 3 caracteres.

En este caso serían `three` y `four`, o sea tiene 2 elementos. Veamos esto en el código corriendo para comprobar que lo que estamos diciendo es real, volvamos a nuestra app que el botón y el textview que usamos anteriormente

Filtrando por predicado

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
    setContentView(R.layout.activity_main)  
  
    val inicial = findViewById<TextView>(R.id.textview) as TextView  
    val resultado = findViewById<TextView>(R.id.textview2) as TextView  
  
    val button = findViewById<Button>(R.id.button) as Button  
  
    val numeros = listOf("one", "two", "three", "four")  
    val largoMayorA3 = numeros.filter { it.length > 3 }  
  
    button.setOnClickListener {  
        inicial.text = numeros.toString()  
        resultado.text = largoMayorA3.toString()  
    }  
}
```

Filtrando por predicado

Una vez que levantemos la app y apretemos el botón vamos a ver la lista original en el primer TextView y en el segundo TextView la lista después del filter.



Filtrando por predicado

Ahora veamos el mismo ejemplo pero para Maps.

```
val numeros = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
val filtro = numeros.filter { (key, value) -> key.endsWith("1") && value > 10}
```

Filtrando por predicado

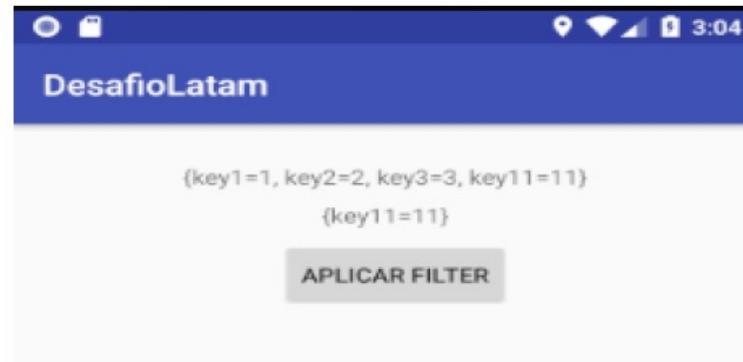
Como vemos acá tenemos un map con key y value, donde agregamos un filtro con dos condiciones, una para la llave que tiene que terminar con el subString “1” y el valor de esa llave sea mayor a 10, en este caso el único que cumple esa condición es el último elemento con **key = “key11”** y **valor = 11**.

Hagamos lo mismo que la vez pasada para comprobar nuestro código.

Filtrando por predicado

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
    setContentView(R.layout.activity_main)  
  
    val inicial = findViewById<TextView>(R.id.textview) as TextView  
    val resultado = findViewById<TextView>(R.id.textview2) as TextView  
  
    val button = findViewById<Button>(R.id.button) as Button  
  
    val numeros = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)  
    val filtro = numeros.filter { (key, value) -> key.endsWith("1") && value > 10}  
  
    button.setOnClickListener {  
        inicial.text = numeros.toString()  
        resultado.text = filtro.toString()  
    }  
}
```

Filtrando por predicado



Filtrando por predicado

Ahora veamos otro ejemplo, filterIndexed.

```
val numeros = listOf("one", "two", "three", "four")
```

```
val filteredIdx = numeros.filterIndexed { index, s -> (index != 0) && (s.length < 5) }
```

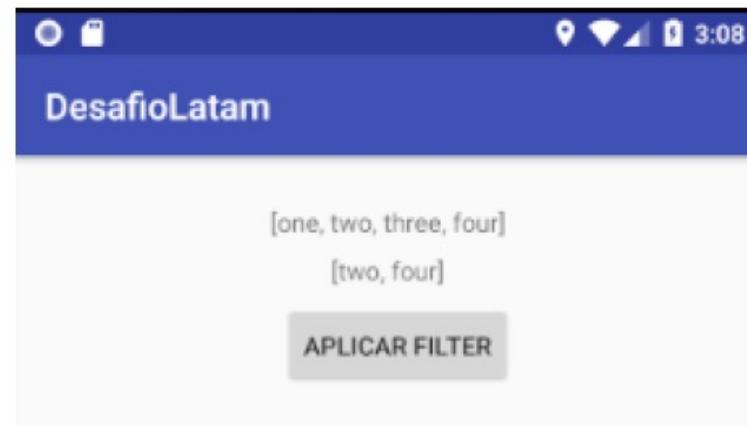
Filtrando por predicado

En este caso filterIndexed recibe 2 parámetros el índice que queramos filtrar y la condición que queramos usar para filtrar, en este caso se va a filtrar por que el índice sea distinto de cero, o sea **quedá eliminado automáticamente el primer elemento** y luego una condición que el largo del contenido no sea mayor a 5, por lo que quedá **eliminado el elemento que contiene la palabra “three”**.

Filtrando por predicado

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
    setContentView(R.layout.activity_main)  
  
    val inicial = findViewById<TextView>(R.id.textview) as TextView  
    val resultado = findViewById<TextView>(R.id.textview2) as TextView  
  
    val button = findViewById<Button>(R.id.button) as Button  
  
    val numeros = listOf("one", "two", "three", "four")  
    val filtro = numeros.filterIndexed { index, s -> (index != 0) && (s.length < 5) }  
  
    button.setOnClickListener {  
        inicial.text = numeros.toString()  
        resultado.text = filtro.toString()  
    }  
}
```

Filtrando por predicado



Filtrando por predicado

Este filtro no está disponible para Maps. Ahora veamos otro ejemplo con filterNot.

```
val numbers = listOf("one", "two", "three", "four")  
  
val filteredNot = numbers.filterNot { it.length <= 3 }
```

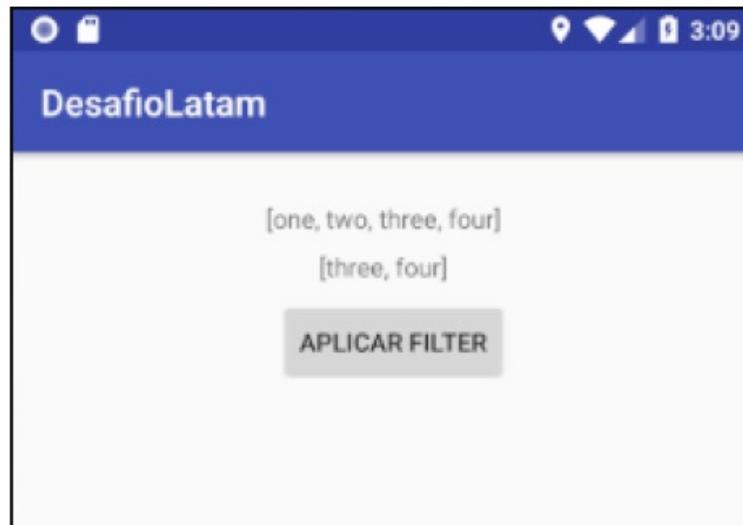
Filtrando por predicado

Este filtro nos permite crear condiciones negadas, en este caso, lo aplicaremos cuando el contenido de la posición de la lista **NO tenga un largo menor o igual a 3**. O sea solo quedarían los elementos “three” y “four” en la lista después de aplicar el filtro.

Filtrando por predicado

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
    setContentView(R.layout.activity_main)  
  
    val inicial = findViewById<TextView>(R.id.textview) as TextView  
    val resultado = findViewById<TextView>(R.id.textview2) as TextView  
  
    val button = findViewById<Button>(R.id.button) as Button  
  
    val numeros = listOf("one", "two", "three", "four")  
    val filtro = numeros.filterNot { it.length <= 3 }  
  
    button.setOnClickListener {  
        inicial.text = numeros.toString()  
        resultado.text = filtro.toString()  
    }  
}
```

Filtrando por predicado



Filtrando por predicado

Este filtro si está disponible para Maps y se usa de la misma manera que filter normal, pero asumimos que es la condición negada. Ahora veamos otro ejemplo con filterIsInstance.

```
val numeros = listOf(null, 1, "two", 3.0, "four")
var result = ""

numeros.filterIsInstance<String>().forEach {
    result += it.toUpperCase()
}
```

Filtrando por predicado

Como podemos ver acá podemos hacer uso de `filterIsInstance`, esto hará que se filtre según el tipo de Objeto que necesitemos hacer el filtro, de esta manera el ejemplo anterior, quedaría después de aplica el filtro solos los campos “`two`” y “`four`”. Veamos el ejemplo.

Filtrando por predicado

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
    setContentView(R.layout.activity_main)  
  
    val inicial = findViewById<TextView>(R.id.textview) as TextView  
    val resultado = findViewById<TextView>(R.id.textview2) as TextView  
  
    val button = findViewById<Button>(R.id.button) as Button  
  
    val numeros = listOf(null, 1, "two", 3.0, "four")  
    val filtro = numeros.filterIsInstance<String>()  
  
    button.setOnClickListener {  
        inicial.text = numeros.toString()  
        resultado.text = filtro.toString()  
    }  
}
```

Filtrando por predicado



Filtrando por predicado

Este filtro no aplica para Maps. Ahora veamos otro filtro filterNotNull.

```
val numeros = listOf(null, "one", "two", null)  
val listaFiltrada = numeros.filterNotNull()
```

FilterNotNull, nos saca los elementos que tengamos null dentro de nuestra lista, algo que puede ser de mucha utilidad cuando uno desarrolla. En este caso los únicos elementos que quedarían después de aplicar el filtro serian “one” y “two”.

Filtrando por predicado

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

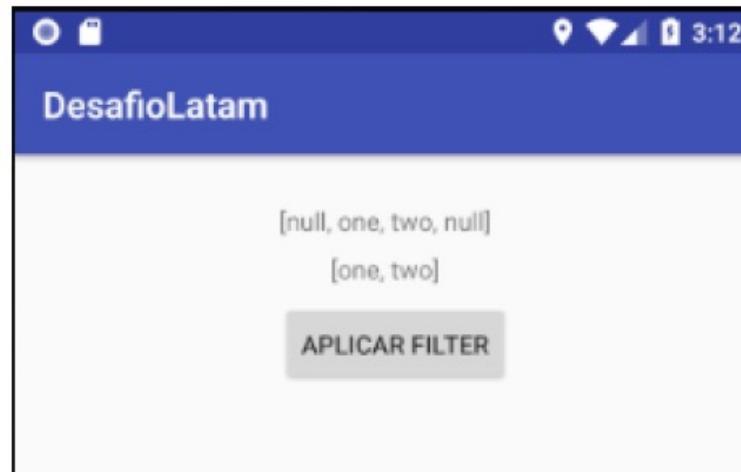
    val inicial = findViewById<TextView>(R.id.textview) as TextView
    val resultado = findViewById<TextView>(R.id.textview2) as TextView

    val button = findViewById<Button>(R.id.button) as Button

    val numeros = listOf(null, "one", "two", null)
    val filtro = numeros.filterNotNull()

    button.setOnClickListener {
        inicial.text = numeros.toString()
        resultado.text = filtro.toString()
    }
}
```

Filtrando por predicado



Partitioning

Otra función que tenemos disponible es Partitioning, que nos permite, dividir en una lista en 2 según un criterio, veamos un ejemplo.

```
val numbers = listOf("one", "two", "three", "four")
val (entranEnElRango, resto) = numbers.partition { it.length > 3 }
```

Aca podemos ver, como usamos el método partition, generamos una condición que si el elemento tiene un largo mayor a 3, pasa a guardarse en la lista **entranEnElRango** y lo que no cumplea la condición, pasa a **resto**. O sea en cada lista quedan con 2 elementos.

Partitioning

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
    setContentView(R.layout.activity_main)  
  
    val inicial = findViewById<TextView>(R.id.textview) as TextView  
    val resultado = findViewById<TextView>(R.id.textview2) as TextView  
  
    val button = findViewById<Button>(R.id.button) as Button  
  
    val numeros = listOf("one", "two", "three", "four")  
    val (entranEnElRango, resto) = numeros.partition { it.length > 3 }  
  
    button.setOnClickListener {  
        inicial.text = numeros.toString()  
        resultado.text = "${entranEnElRango.toString()} - ${resto.toString()}"  
    }  
}
```

Partitioning



Como podemos ver partition es una manera muy simple de dividir listas en base a condiciones.

Pruebas de predicado

Tenemos unos métodos que nos sirven para hacer pruebas sobre las condiciones que usemos.

- **any**: retorna true si al menos uno de los elementos cumple la condición.
- **none**: retorna true si ningún elemento cumple la condición.
- **all**: retorna true si todos los elementos cumplen la condición.

Veamos una prueba de cada uno.

Any

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

    val textView = findViewById<TextView>(R.id.textview) as TextView

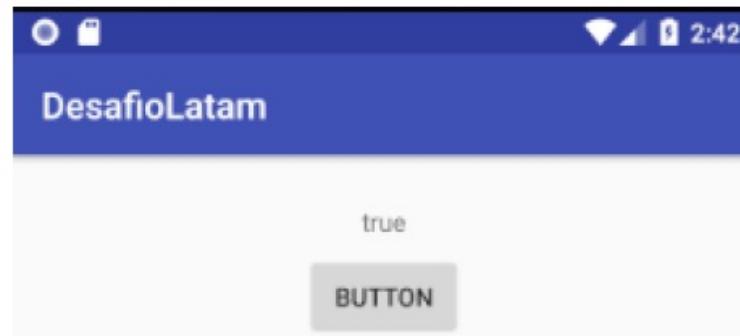
    val button = findViewById<Button>(R.id.button) as Button

    val numbers = listOf("one", "two", "three", "four")

    button.setOnClickListener { textView.text = numbers.any { it.endsWith("e") }.toString() }
}
```

Any

En este caso tenemos la lista que contiene los string con los nombres de los números del 1 al 4 en inglés y corremos la prueba **any**, cuando hacemos click en el botón, en esta prueba, estamos evaluando que al menos uno de los elementos termina con la letra “e”, lo que obviamente es **true**, ya que tenemos el caso del **one** y el **three**.



None

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

    val textview = findViewById<TextView>(R.id.textview) as TextView

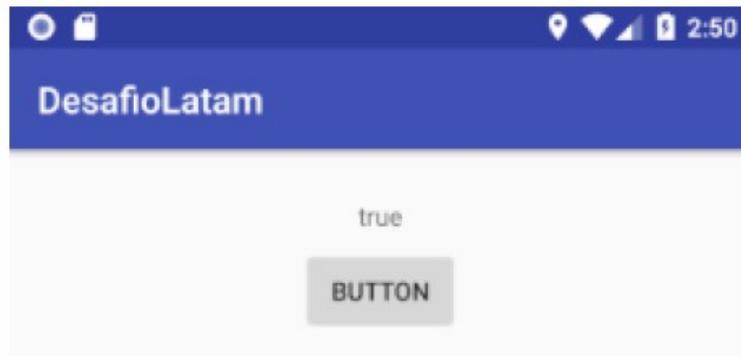
    val button = findViewById<Button>(R.id.button) as Button

    val numbers = listOf("one", "two", "three", "four")

    button.setOnClickListener { textview.text = numbers.none { it.endsWith("a") }.toString() }
}
```

None

En este caso usaremos el mismo ejemplo anterior pero cambiando la prueba, usaremos **none** para ver que **ningún elemento de la lista termina con “a”** y si revisamos los elementos, esto debería cumplirse sin problemas y retornar **true**.



All

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

    val textview = findViewById<TextView>(R.id.textview) as TextView

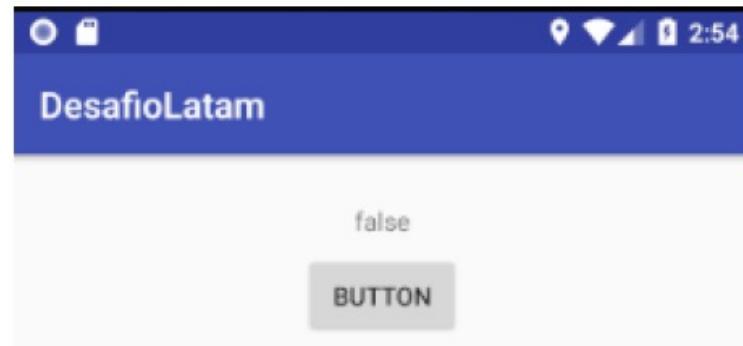
    val button = findViewById<Button>(R.id.button) as Button

    val numbers = listOf("one", "two", "three", "four")

    button.setOnClickListener { textview.text = numbers.all { it.endsWith("e") }.toString() }
}
```

All

Ahora haremos la siguiente evaluación, si es que todos los elementos de la lista terminan con “e”, lo cual obviamente no se cumple y debería retornar **false**.



All

Acá hicimos una revisión bien completa sobre los filters y como estos nos pueden ayudar en el desarrollo. Cabe destacar que estas pruebas podemos aplicarlas también con los maps, revisemos el caso pero solo con un ejemplo.

All

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

    val textView = findViewById<TextView>(R.id.textview) as TextView

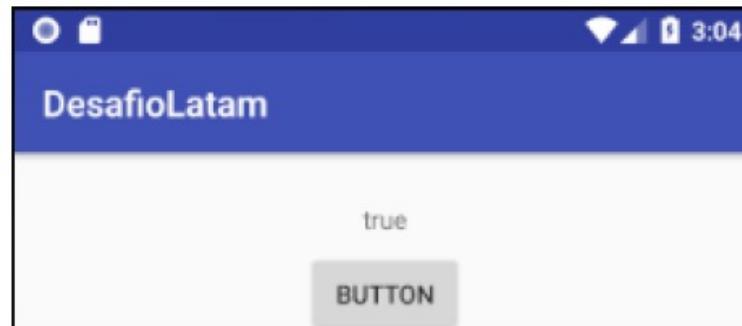
    val button = findViewById<Button>(R.id.button) as Button

    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)

    button.setOnClickListener { textView.text = numbersMap.any { (key, value) -> key.endsWith("1") &&
        value > 10 }.toString() }
}
```

All

en este caso usaremos el mismo map del primer ejemplo de filter y le pondremos la misma condición que teníamos esa vez, pero usando **any**. En este caso, dado que alguna de las key termina con “**1**” y algún value es **mayor a 10**, debería ser **true**.



Sorting

Ahora revisaremos cómo funciona el ordenamiento en Kotlin y las distintas formas que podemos utilizar para simplificar nuestro trabajo.

Sorting

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
    setContentView(R.layout.activity_main)  
  
    val inicial = findViewById<TextView>(R.id.textview) as TextView  
    val resultado = findViewById<TextView>(R.id.textview2) as TextView  
  
    val button = findViewById<Button>(R.id.button) as Button  
  
    val lista = mutableListOf(7, 8, 1, 5, 3, 2, 4)  
  
    button.setOnClickListener {  
        inicial.text = lista.toString()  
        lista.sort()  
        resultado.text = lista.toString()  
    }  
}
```

Sorting

Ahora vemos cómo generamos una lista mutable de números desordenados.

¿Qué quiere decir que sea una lista mutable?

El hecho de que sea mutable, quiere decir que le podemos agregar y quitar elementos.

En este caso, aplicamos el método `sort()` que nos ordenará automáticamente la lista de menor a mayor.



Sorting

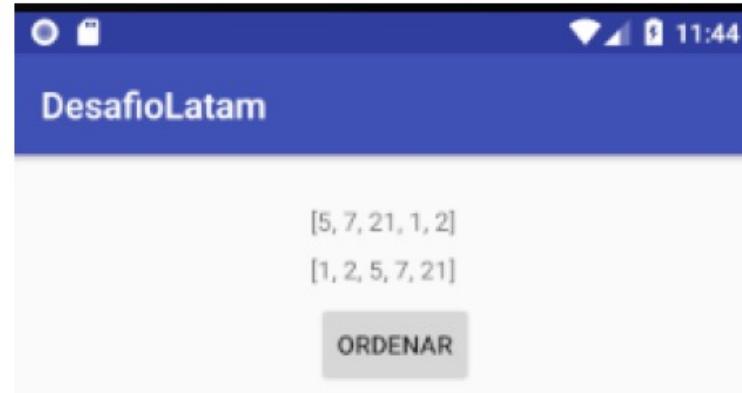
Veamos el mismo ejemplo pero con un arreglo normal, para que veamos un caso más típico sin mutableList.

Sorting

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
    setContentView(R.layout.activity_main)  
  
    val inicial = findViewById<TextView>(R.id.textview) as TextView  
    val resultado = findViewById<TextView>(R.id.textview2) as TextView  
  
    val button = findViewById<Button>(R.id.button) as Button  
  
    val lista = arrayOf(5, 7, 21, 1, 2)  
  
    button.setOnClickListener {  
        inicial.text = lista.contentToString()  
        lista.sort()  
        resultado.text = lista.contentToString()  
    }  
}
```

Sorting

como podemos ver se usa igual el método sort sin problemas y el resultado es el mismo



SortBy

Imaginemos tenemos la siguiente lista.

```
val lista = mutableListOf(1 to "z", 2 to "y", 7 to "x", 6 to "t", 5 to "m", 6 to "a")
```

Esto daría una lista de este estilo:

```
[(1, z), (2, y), (7, x), (6, t), (5, m), (6, a)]
```

SortBy

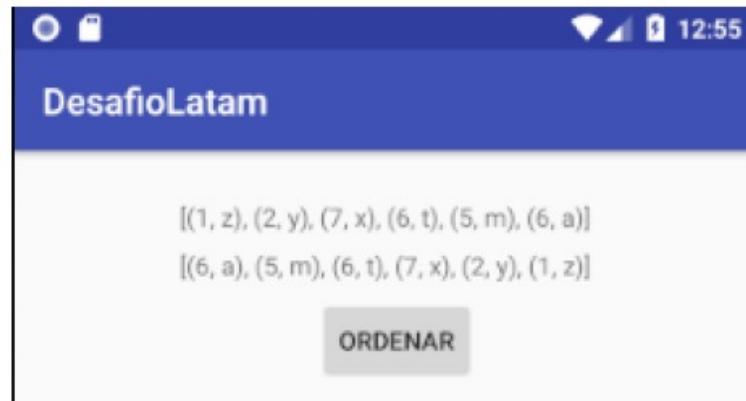
Entonces en casos como este podemos usar SortBy lo que nos permitirá ordenar según qué elemento dentro de cada posición, o sea podemos ordenar todos los elementos según el primero o el segundo de cada paréntesis. Veamos un ejemplo ordenando toda la lista según las letras.

SortBy

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
    setContentView(R.layout.activity_main)  
  
    val inicial = findViewById<TextView>(R.id.textview) as TextView  
    val resultado = findViewById<TextView>(R.id.textview2) as TextView  
  
    val button = findViewById<Button>(R.id.button) as Button  
  
    val lista = mutableListOf(1 to "z", 2 to "y", 7 to "x", 6 to "t", 5 to "m", 6 to "a")  
  
    button.setOnClickListener {  
        inicial.text = lista.toString()  
        lista.sortBy { it.second }  
        resultado.text = lista.toString()  
    }  
}
```

SortBy

Como vemos usamos `sortBy{ it.second }`, o sea que estamos ordenando por el segundo parámetro dentro del elemento (o paréntesis), veamos el resultado.



SortBy

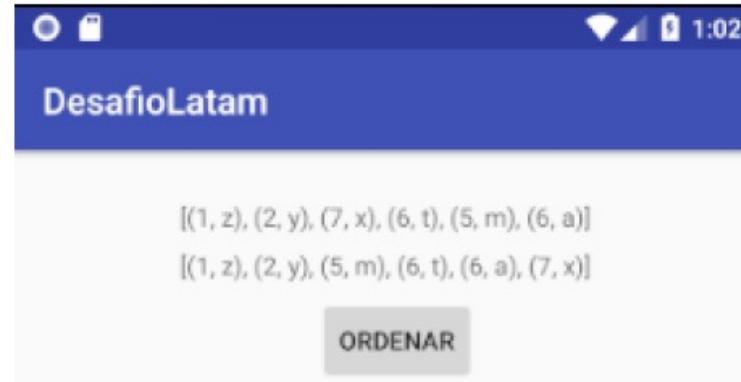
Como podemos ver en el resultado queda primero el **(6, a)** y al ultimo **(1, z)**, esto debido a que se ordena según las letras, ahora veamos el mismo ejemplo pero en base a los números.

SortBy

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
    setContentView(R.layout.activity_main)  
  
    val inicial = findViewById<TextView>(R.id.textview) as TextView  
    val resultado = findViewById<TextView>(R.id.textview2) as TextView  
  
    val button = findViewById<Button>(R.id.button) as Button  
  
    val lista = mutableListOf(1 to "z", 2 to "y", 7 to "x", 6 to "t", 5 to "m", 6 to "a")  
  
    button.setOnClickListener {  
        inicial.text = lista.toString()  
        lista.sortBy { it.first }  
        resultado.text = lista.toString()  
    }  
}
```

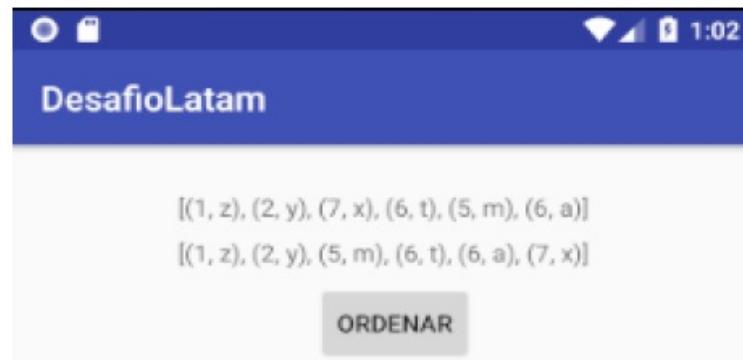
SortBy

Como podemos ver el código es el mismo, pero usamos **sortBy { it.first }**.



SortBy

Como podemos ver el código es el mismo, pero usamos `sortBy { it.first }`.



Se aprecia como ahora la lista parte con **(1, z)** y termina con **(7, x)**, ya que ahora el orden se basa en los números.

Reverse

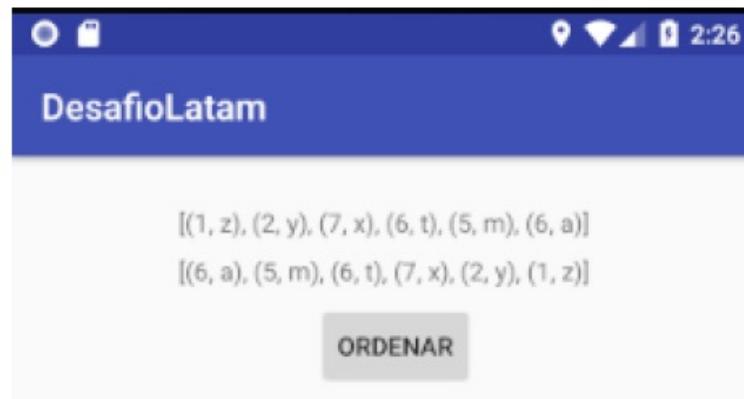
Si necesitamos sencillamente revertir el orden actual, tenemos un método llamado reverse, el cual revisaremos a continuación.

Reverse

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
    setContentView(R.layout.activity_main)  
  
    val inicial = findViewById<TextView>(R.id.textview) as TextView  
    val resultado = findViewById<TextView>(R.id.textview2) as TextView  
  
    val button = findViewById<Button>(R.id.button) as Button  
  
    val lista = mutableListOf(1 to "z", 2 to "y", 7 to "x", 6 to "t", 5 to "m", 6 to "a")  
  
    button.setOnClickListener {  
        inicial.text = lista.toString()  
        lista.reverse()  
        resultado.text = lista.toString()  
    }  
}
```

Reverse

Es muy sencillo al igual que `sort()`, llamamos al método `reverse()` y este nos entregará el ls lista ordenada de manera inversa.



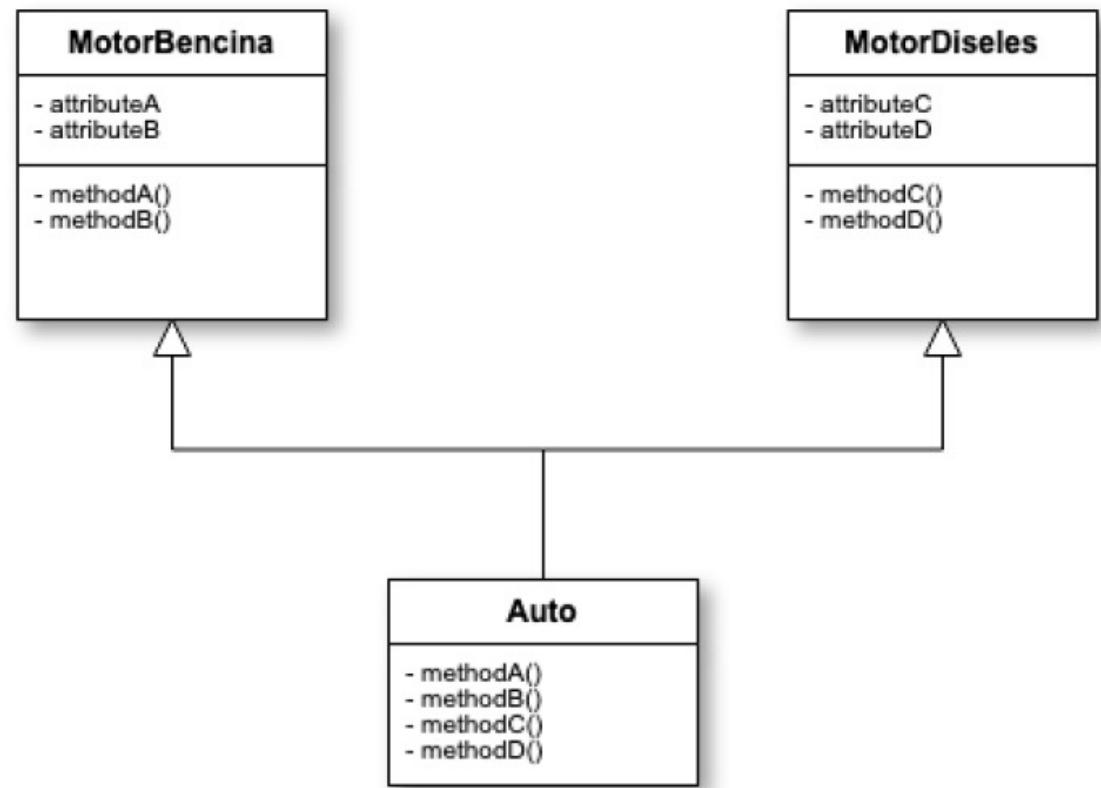
Patrón Delegate

El patrón delegate es una patrón de diseño que se usa en la programación orientada a objetos, que se usa generalmente cuando los lenguajes no soportan la herencia múltiple.

¿Qué es la herencia múltiple?

Herencia múltiple hace referencia a la característica de los lenguajes de programación orientada a objetos en la que una clase puede heredar comportamientos y características de más de una superclase. Esto contrasta con la herencia simple, donde una clase sólo puede heredar de una superclase.

Patrón Delegate



Patrón Delegate

O sea para poner un ejemplo, supongamos tenemos una clase Auto y quisiéramos que esta heredara de las clases MotorBencina y MotorDiesel. En este caso estaríamos tratando se usar herencia múltiple y no podríamos en lenguajes como Java. Una solución para esto es el uso de éste patrón.

Kotlin tiene funciones que nos permiten trabajar usando este patrón de manera más simple. A continuación veremos un ejemplo explicando este patrón usando Kotlin.

Patrón Delegate

Supongamos que tenemos la siguiente interfaz **Mamifero**.

```
interface Mamifero {  
  
    fun nombre(): String  
}
```

Esta tendrá una función nombre que retorna un String. Ahora implementemos esta interfaz en 2 clases que representarán animales.

```
class Gato : Mamifero {  
  
    override fun nombre() = "Gato"  
}
```

```
class Lince : Mamifero {  
  
    override fun nombre() = "Lince"  
}
```

Patrón Delegate

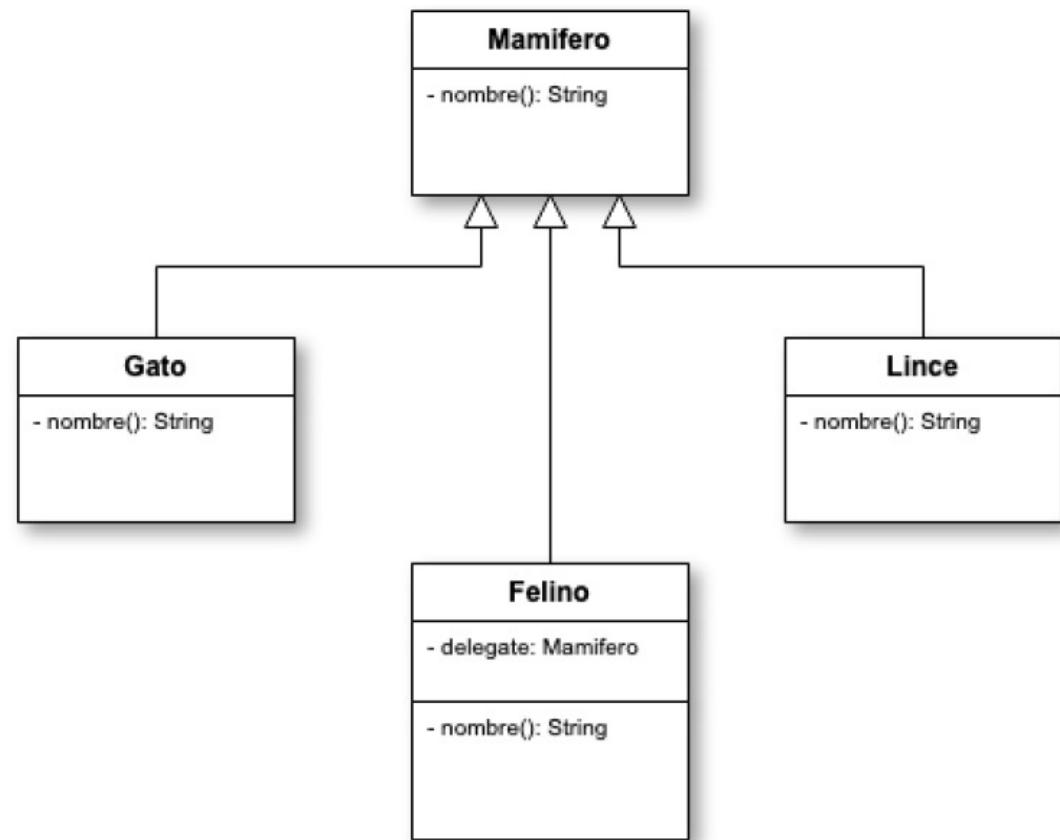
Como podemos ver tenemos 2 clases que implementan la interfaz **Mamifero** y cada una sobrescribe el método nombre y le pone su propio valor. Ahora Crearemos otra clase llamada felinos, que hará uso del patrón delegate.

```
class Felinos(private val delegate: Mamifero) : Mamifero by delegate {  
  
    override fun nombre() = "Soy mamifero y felino, mi nombre es: ${delegate.nombre()}"  
}
```

Patrón Delegate

Creamos la clase **Felinos**, que en este ejemplo, hemos indicado que encapsulará un objeto delegado de tipo **Mamifero** y también puede usar la funcionalidad de la implementación del **Mamifero**, que puede ser **Gato** o **Lince** en este caso.

Patrón Delegate



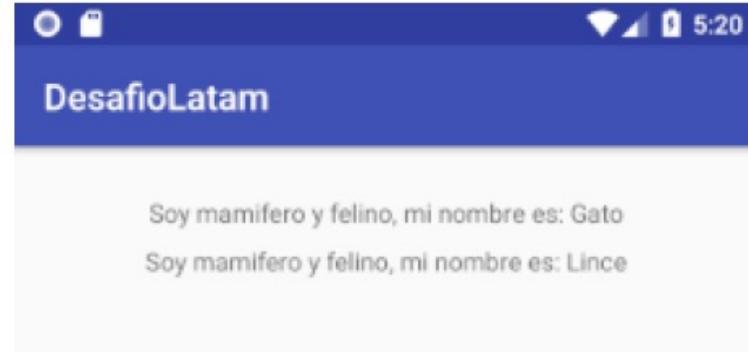
Patrón Delegate

Veamos como usar esto y como se vería si lo ejecutamos.

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
    setContentView(R.layout.activity_main)  
  
    val mamifero1 = Felinos(Gato())  
    textView.text = mamifero1.nombre()  
  
    val mamifero2 = Felinos(Lince())  
    textView2.text = mamifero2.nombre()  
}
```

Patrón Delegate

Como podemos ver, creamos una variable llamada **mamifero1** que usa la clase **Felinos**, pero pasándole el valor de **Gato**. Y lo mismo, pero para la clase **Lince** en otra variable llamada **mamifero2**, veamos la ejecución.



Patrón Delegate

Cómo hemos podido ver acá, un ejemplo simple de como Kotlin, nos permite usar el patrón delegate e implementar la clase **Felinos**, usando 2 clases como **Gato** o **Lince**. Podríamos crear por ejemplo otra clase **Paquidermos** y esta que llame a otros tipos de animales como **Elefante** y **Rinoceronte**, y todas a la vez todas vendrían de **Mamifero** y así podríamos ir agrandando el árbol de clases de nuestro ejemplo y ayudarnos con el patrón delegate, para implementarlo de manera simple, como se pudo ver.