{desafío}
latam_

Kotlin y Android _

Utilidades de Kotlin

¿Qué aprenderemos?

- Concatenar Strings
- OnClickListener
- Funciones lambda

Concatenar Strings

Este es clásico concatenado de String, igual que el que se usa en Java.

```
val calculadora = Calculadora()


 val a = 1

val b = 2



textview.text = "Tu resultado "+a+" + "+b+" es:
"+calculadora.suma(a, b)
```

Concatenar Strings

En kotlin tenemos esta alternativa, que es mucho más legible y usada en otros lenguajes modernos.

```
val calculadora = Calculadora()


val a = 1

val b = 2


textview.text = "Tu resultado $a + $b es:
${calculadora.suma(a, b)}"
```

Listeners

Así es la típica manera en que se maneja el onClickListener en Android.

```
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

Pero kotlin tiene esta nueva manera, que simplifica mucho nuestro código.

Esto puede ser aplicado, a cualquier Listener de los que tenemos en Android como por ejemplo **OnLongClickListener**, **OnFocusChangeListener**, **OnTouchListener**, etc.

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


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


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

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.

Función Lambda

Una función lambda tiene la siguiente estructura.

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 ->.

```
{ x: Int, y: Int -> x + y }
```

Función Lambda

En el ejemplo del onClickListener donde primero teníamos código de la parte de arriba.

Y gracias a la función lambda, kotlin logra simplificarlo al código de abajo.

```
button.setOnClickListener(object :
View.OnClickListener {

          override fun onClick(v: View) {

              textview.text = "Texto cambiado"

          }

      })

-----------------------------------------------

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

View Binding

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'
```

View Binding

Si usamos las Kotlin Android Extensions y tuviéramos por ejemplo los siguientes componentes en nuestro layout.

```
<TextView

       android:id="@+id/textview"

       android:layout_width="wrap_content"

       android:layout_height="wrap_content"

 />

<Button

       android:id="@+id/button"

       android:layout_width="wrap_content"

       android:layout_height="wrap_content"

/>
```

View Binding

Tendríamos que hacer lo siguiente para poder referenciar estos componentes en nuestra Activity.

Usando el método findViewById.

```
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"

}
```

View Binding

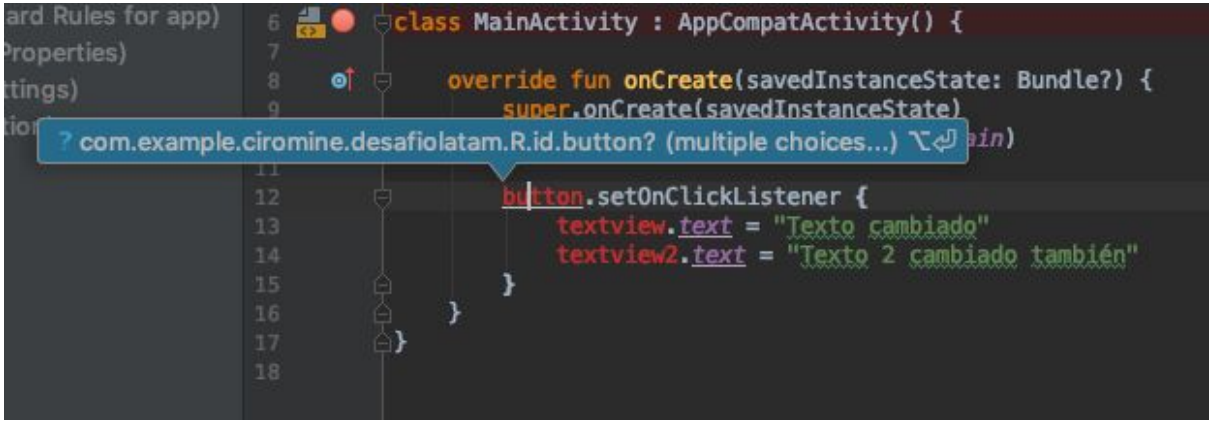
En Kotlin podríamos hacerlo así sin necesidad de usar findViewById.

Solo hay que agregar ese import que se ve en la parte superior.

Que el mismo IDE nos puede ayudar a referenciar.

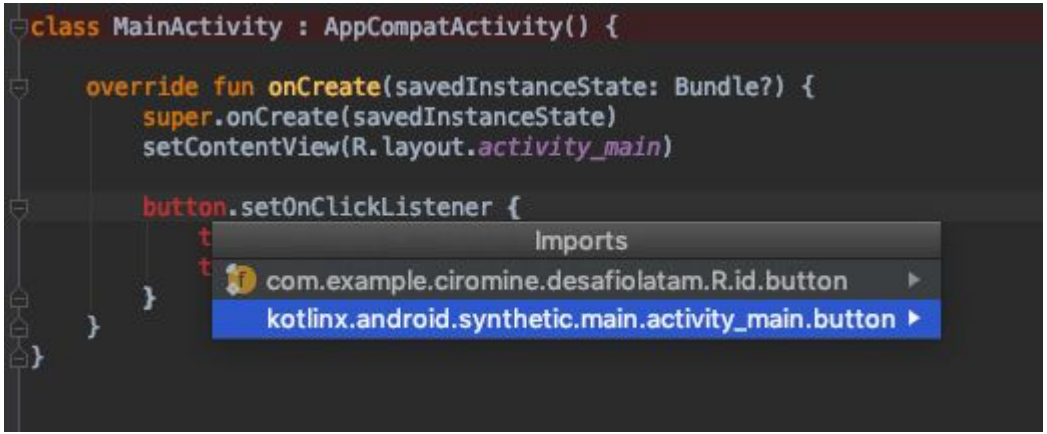
```
import
kotlinx.android.synthetic.main.activity_main.*



button.setOnClickListener {

       textview.text = "Texto cambiado"

       textview2.text = "Texto 2 cambiado también"

   }
```

¿Cómo nos ayuda el IDE con View Binding?



El mismo ide nos ayuda recomendado el import.

¿Cómo nos ayuda el IDE con View Binding?



Profundizando collections en Kotlin

¿Qué aprenderemos?

- Usar Filters
- Usar Listas
- Usar Maps
- Utilizar funciones de ordenamiento

Filters

Los filters nos permiten filtrar el contenidos de las listas o los maps.

A continuación veremos una serie de ejemplos distintos de uso de filters en Kotlin.

Filters


```
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()

   }
```

Filters


```
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()

}
```

Filters


```
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()

}
```

Filters


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

val filtro = numeros.filterNot { it.length <= 3 }



button.setOnClickListener {

       inicial.text = numeros.toString()

       resultado.text = filtro.toString()

}
```

Filters


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

val filtro = numeros.filterIsInstance<String>()


button.setOnClickListener {

       inicial.text = numeros.toString()

       resultado.text = filtro.toString()

}
```

Filters


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

val filtro = numeros.filterNotNull()


button.setOnClickListener {

       inicial.text = numeros.toString()

       resultado.text = filtro.toString()

}
```

Filters


```
 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()}"

   }
```

Probando los predicados de los filters


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


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

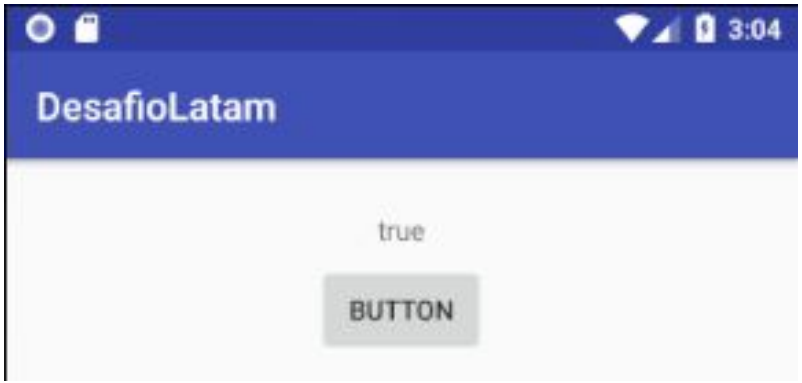
Probando los predicados de los filters


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


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

Probando los predicados de los filters


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


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

Probando los predicados de los filters



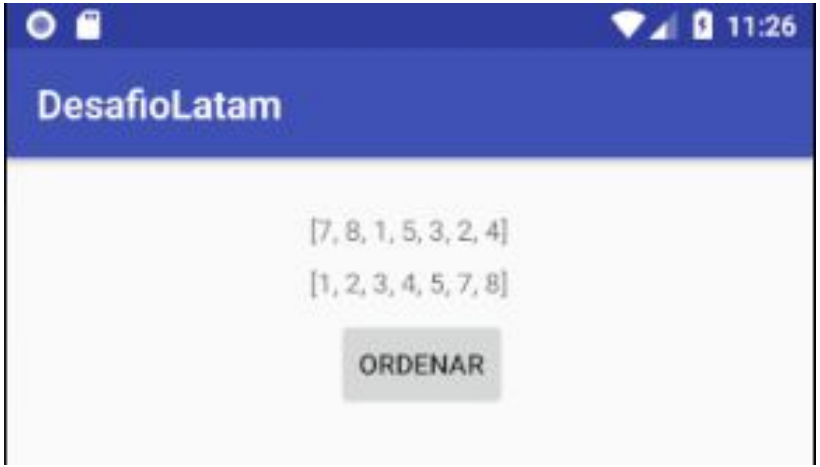

```
val numbersMap = mapOf("key1" to 1, "key2" to 2,
"key3" to 3, "key11" to 11)


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

Sorting

Los métodos de Sorting, nos ayudan a ordenar el contenidos de listas y maps.

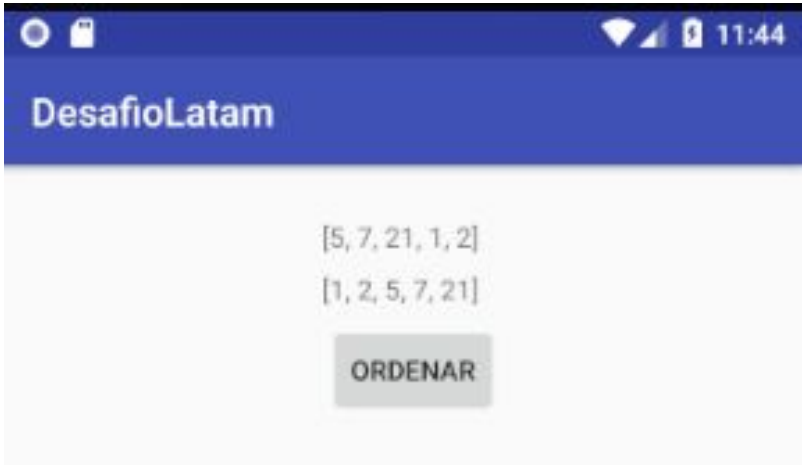
Ahora veamos algunos ejemplos.

Sorting




```
val lista = mutableListOf(7, 8, 1, 5, 3, 2, 4)


   button.setOnClickListener {

       inicial.text = lista.toString()

       lista.sort()

       resultado.text = lista.toString()

   }
```

Sorting




```
val lista = arrayOf(5, 7, 21, 1, 2)


   button.setOnClickListener {

       inicial.text = lista.contentToString()

       lista.sort()

       resultado.text = lista.contentToString()

   }
```

Sorting


```
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()

   }
```

Sorting


```
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()

   }
```

Sorting


```
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()

   }
```

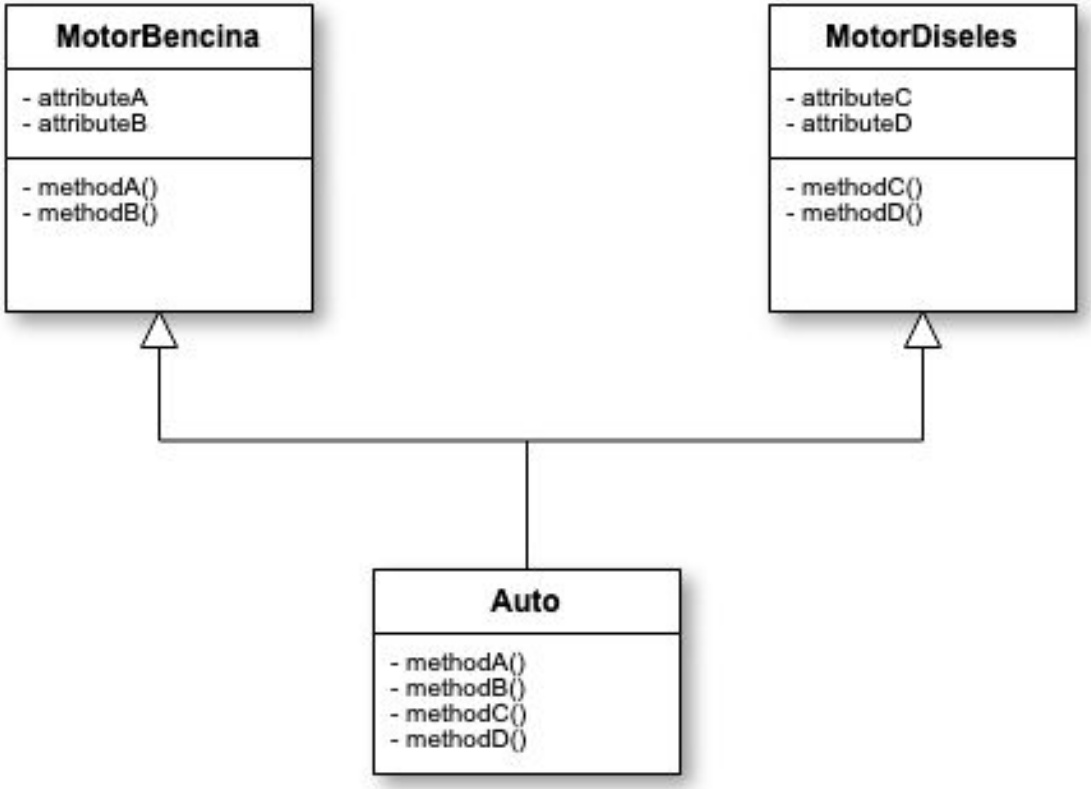
Patrón Delegate

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.

Patrón delegate

Supongamos que tenemos la siguiente interfaz Mamifero.

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

```
interface Mamifero {

   fun nombre(): String

}

-----------------------------------------------

class Gato : Mamifero {

   override fun nombre() = "Gato"

}

-----------------------------------------------

class Lince : Mamifero {

   override fun nombre() = "Lince"

}
```

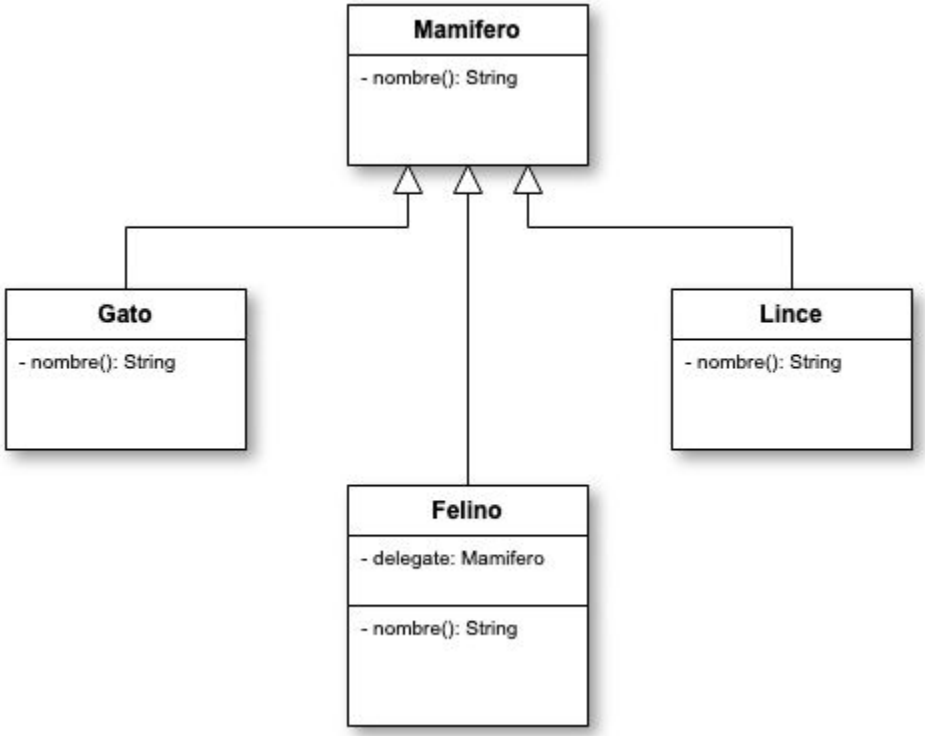
Patrón delegate

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.

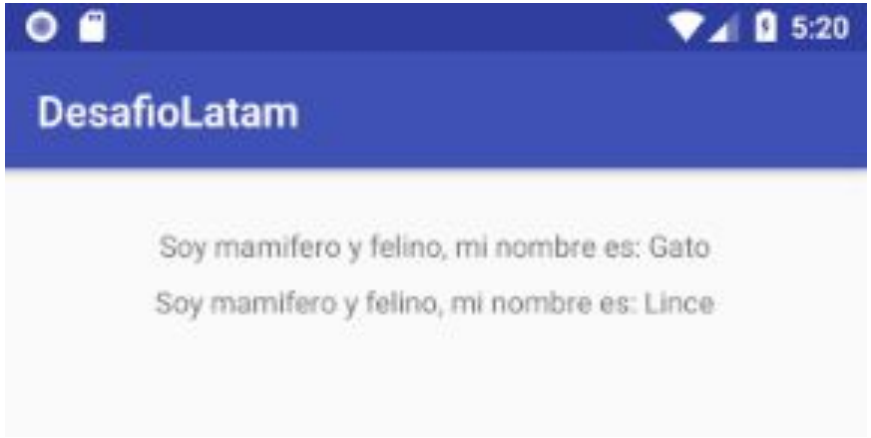
Creamos la clase Felinos, que en este ejemplo, hemos indicado que encapsulará un objeto delegado de tipo Mamifero y también puede usar la funcionalidad de la implementación del Mamifero, que puede ser Gato o Lince en este caso.

```
class Felinos(private val delegate: Mamifero) :
Mamifero by delegate {



   override fun nombre() = "Soy mamifero y felino,
mi nombre es: ${delegate.nombre()}"

}
```

Patrón Delegate



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**.

```
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()

}
```

{desafío} latam_

{desafío} latam_

Academia de talentos digitales

www.desafiolatam.com