Android y Kotlin (Parte I)

Utilidades de Kotlin

Competencias:

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

Introducción

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.

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 nuestro elementos visuales definidos en nuestro layout.

Concatenar Strings

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

```
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)
  }
}
```

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.

```
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)}"
  }
}
```

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 ayb).

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 engorroso.

Listeners

Cuando en Android queremos darle acción al tocar un elemento usamos el Listener Onclick, el cual es un 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.

```
<?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" />

  <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>
```

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.

```
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"
      }
    })
  }
}
```

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.

```
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" }
}
```

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.

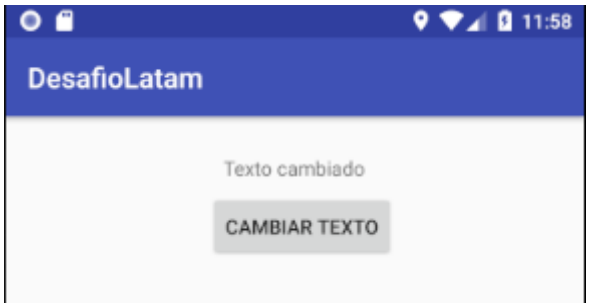

Imagen 1. Resultado.

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

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')
}
```

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.

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

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 **->**.

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"
        }
    })
```

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" }
```

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.

En Kotlin podemos crear nuestras propias funciones lambdas, pero es algo en lo que no profundizaremos ahora.

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.

```
<?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" />

  <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" />

  <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>
```

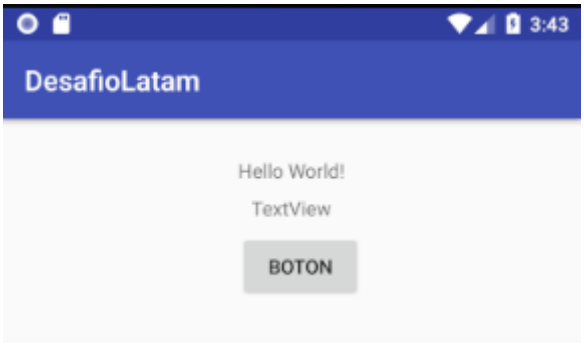

Imagen 2. Resultado

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"
  }
}
```

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"
  }
}
```

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"
```

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.*
```

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.

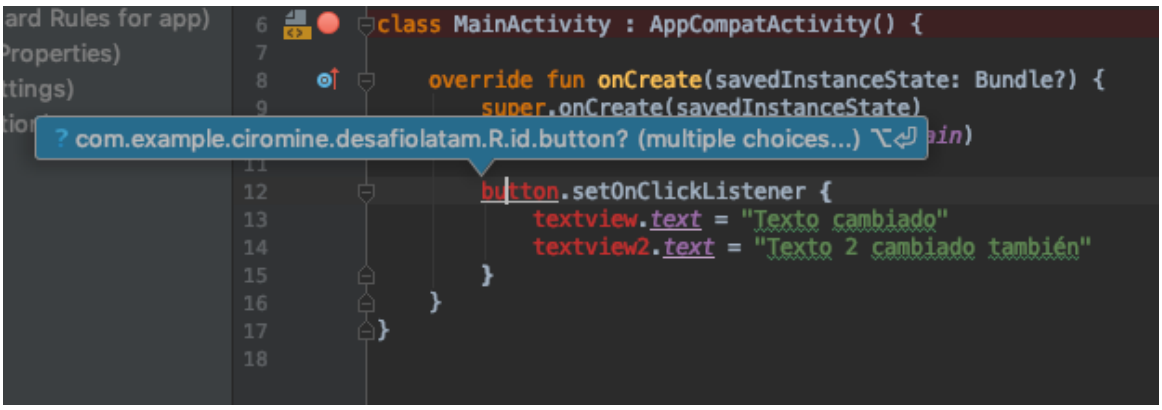

Imagen 3. Nombre de la variable.

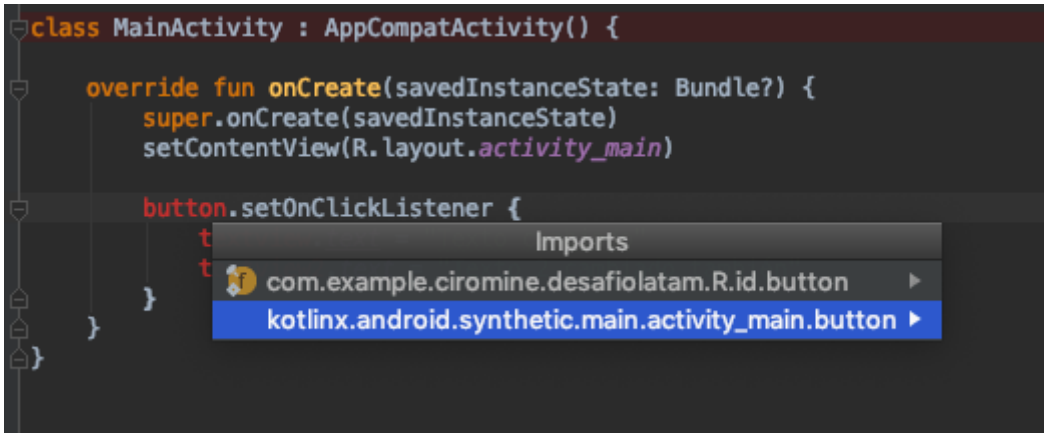

Imagen 4. Importar.

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

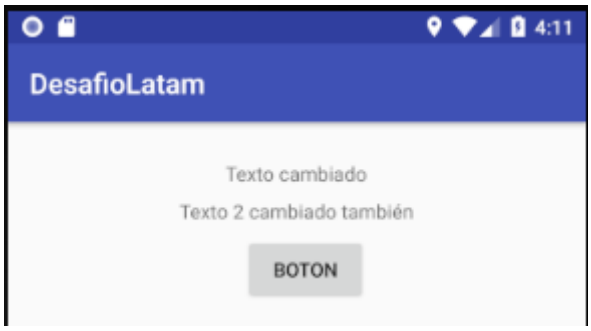

Imagen 5. Resultado.

Profundizando collections en Kotlin

Competencias:

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

Contexto

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.

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")
```

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)
```

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 requerimos, 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 }
```

Primero expliquemos un poco el código anterior, **listOf** nos generará la lista que ingreamos. En este caso **largoMayorA3** contiene una lista con lo 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

```
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()
  }
}
```

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.

Imagen 6. Resultado.

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}
```

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.

```
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()
  }
}
```

Imagen 7. Resultado.

Ahora veamos otro ejemplo, filterIndexed.

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

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

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 **queda eliminado automáticamente el primer elemento** y luego una condición que el largo del contenido no sea mayor a 5, por lo que queda **eliminado el elemento que contiene la palabra "three"**.

```
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()
  }
}
```

Imagen 8. Resultado.

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 }
```

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.

```
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()
    }
}
```

Imagen 9. Resultado.

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()
}
```

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.

```
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()
  }
}
```

Imagen 10. Resultado.

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”**.

```
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()
  }
}
```

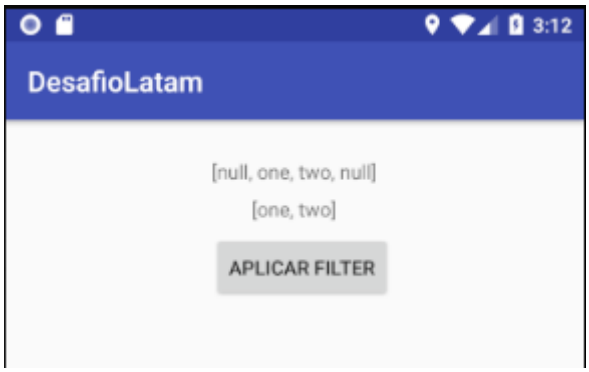

Imagen 11. Resultado.

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.

```
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()}"
    }
}
```

Imagen 12. Resultado.

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

Prueba de predicados

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 lo 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() }
}
```

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**.

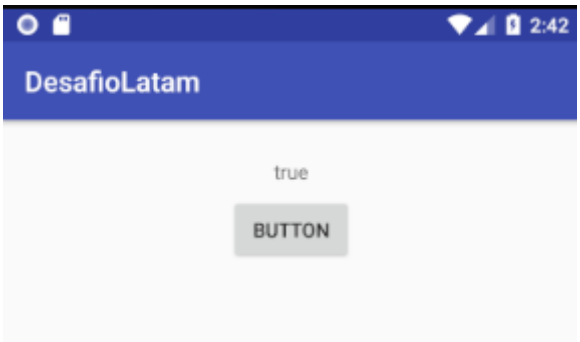

Imagen 13. Resultado.

Como podemos ver se cumple la condición. Ahora revisemos un ejemplo de none.

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() }
}
```

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**.

Imagen 14. Resultado.

Como vemos nuevamente se cumplió la condición señalada, ahora veamos un caso de all.

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() }
}
```

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

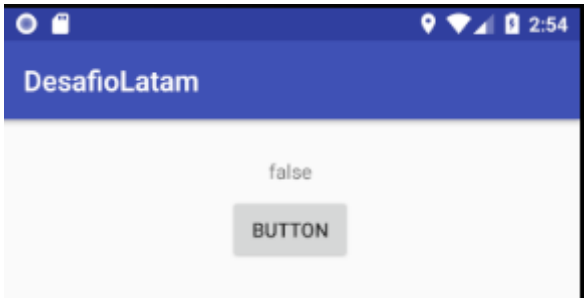

Imagen 15. Resultado.

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.

```
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() }
}
```

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**.

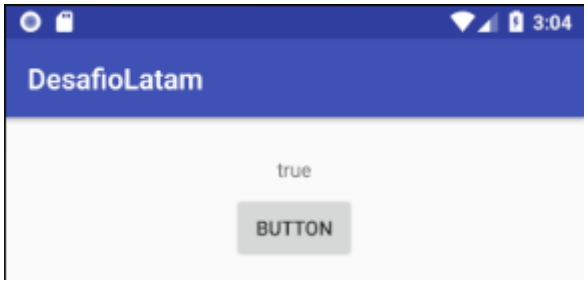

Imagen 16. Resultado.

Ya con esto hicimos una gran revisión de los filters y como estos nos pueden ayudar a nuestro trabajo.

Sorting

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

```
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()
    }
}
```

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.

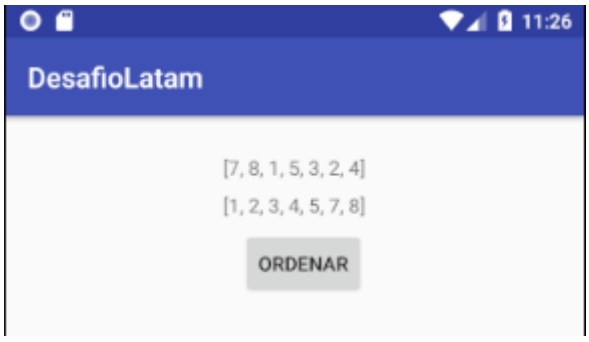

Imagen 17. Resultado.

Y funciona perfectamente y nos ayudará a hacer ordenamiento rápido, ahora veamos otro caso.

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

```
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()
    }
}
```

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

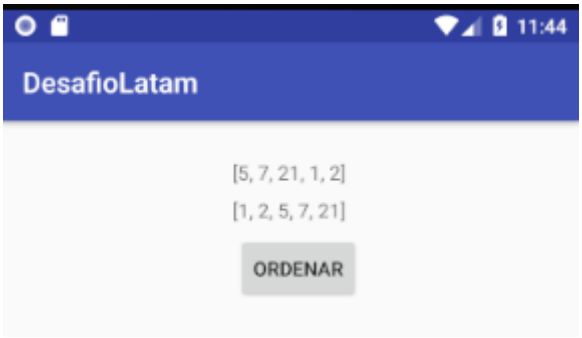

Imagen 18. Resultado.

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)]
```

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.

```
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()
    }
}
```

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

Imagen 19. Resultado.

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.

```
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()
  }
}
```

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

Imagen 20. Resultado.

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.

```
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()
  }
}
```

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

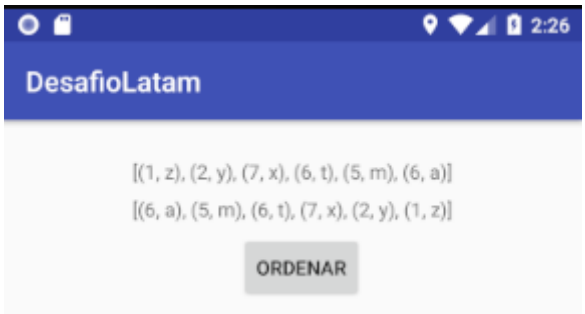

Imagen 21. Resultado.

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.

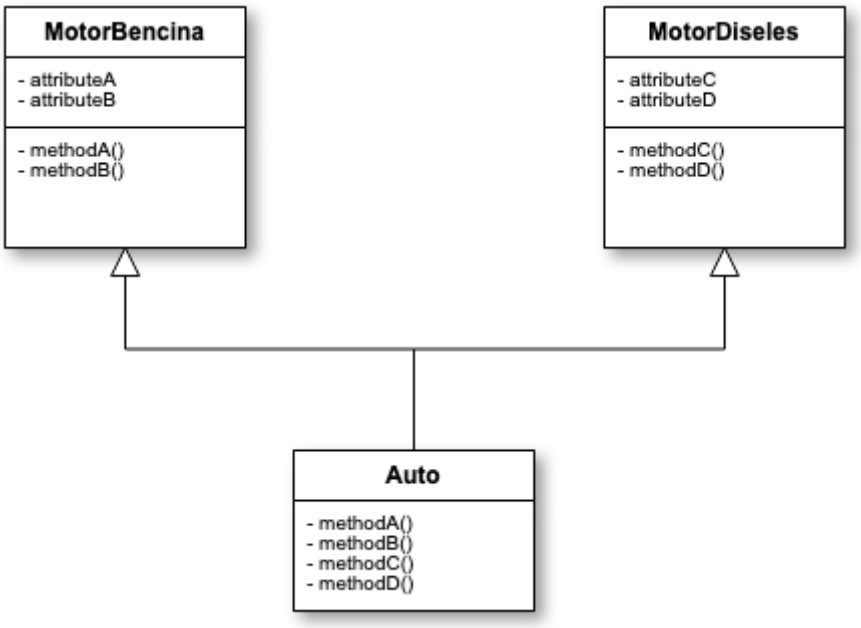

Imagen 22. Herencia múltiple.

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 trabajan usando este patrón de manera más simple. A continuación veremos un ejemplo explicando este patrón usando Kotlin.

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"
}
```

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()}"
}
```

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.

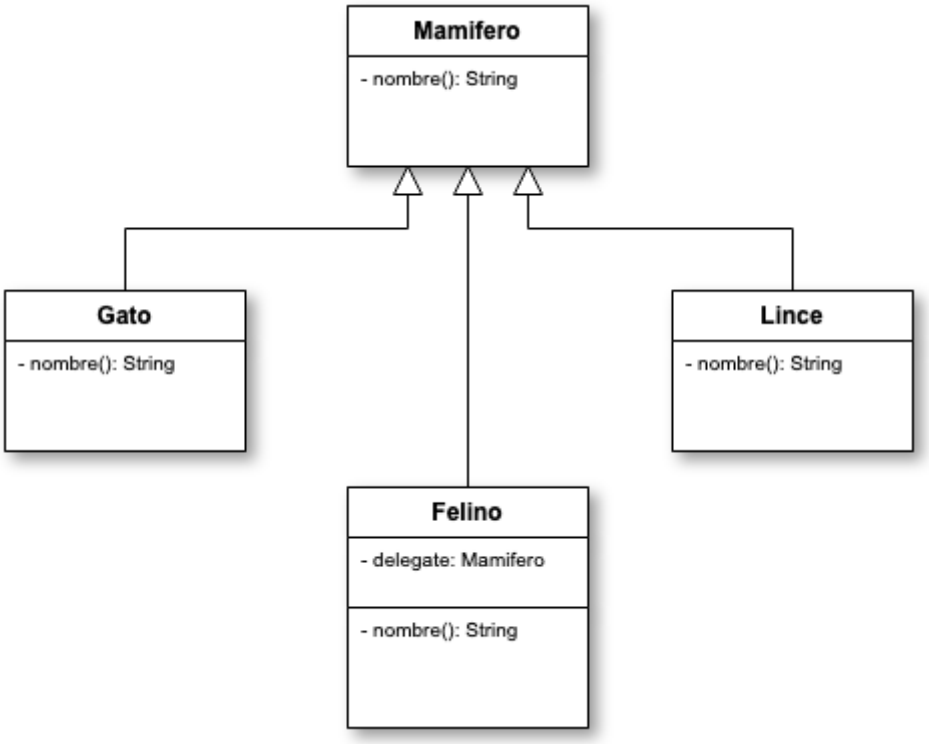

Imagen 23. Ejemplo.

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()
}
```

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.

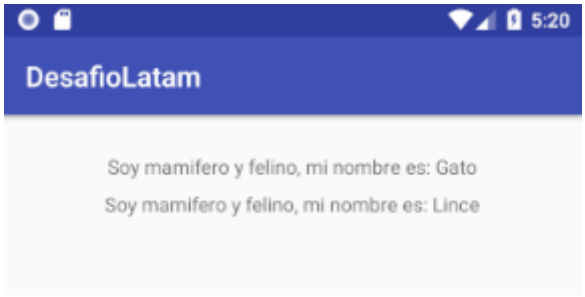

Imagen 24. Resultado.

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.