{desafío} latam_

Android y Kotlin (Parte I)

Iniciando con Kotlin

Competencias:

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

Motivación

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.

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.

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.

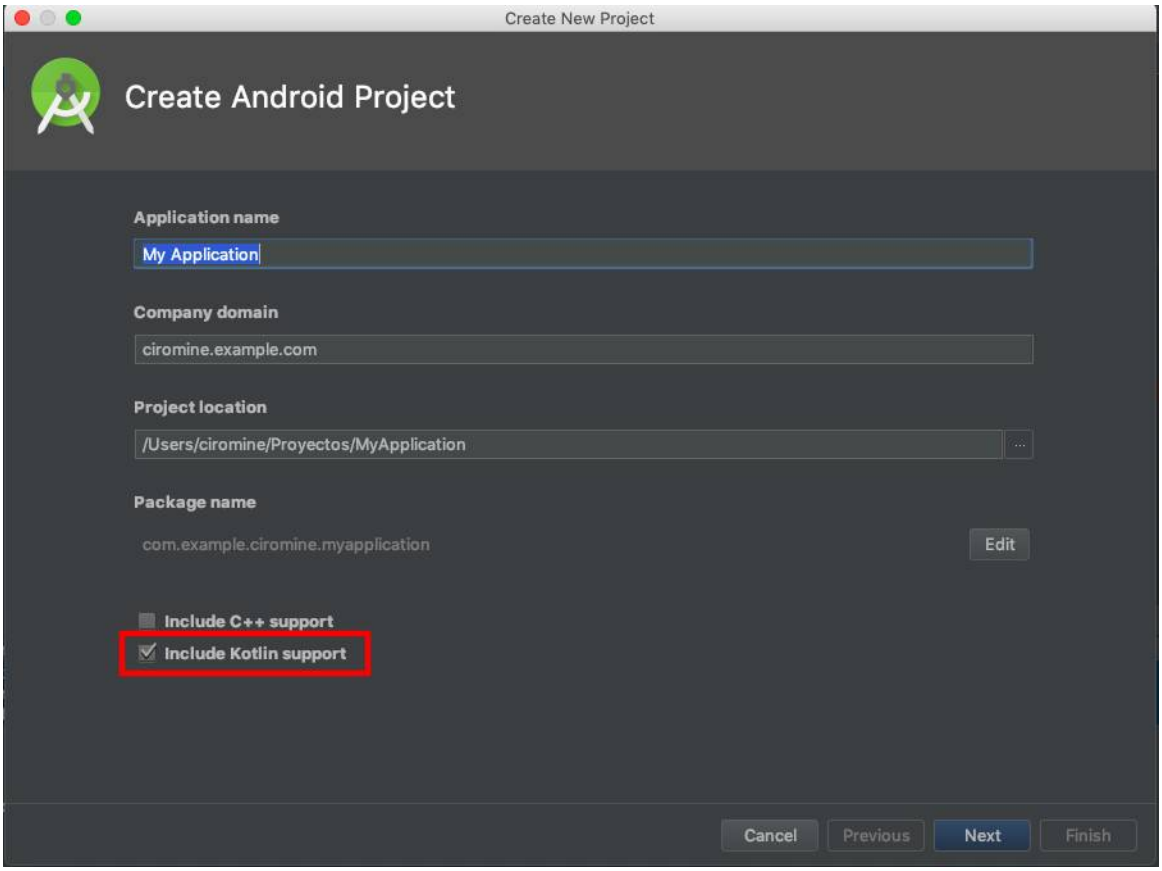

Imagen 1. Crear proyecto Android.

Ya con esta opción seleccionada, nuestro nuevo proyecto partirá con nuestra 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.

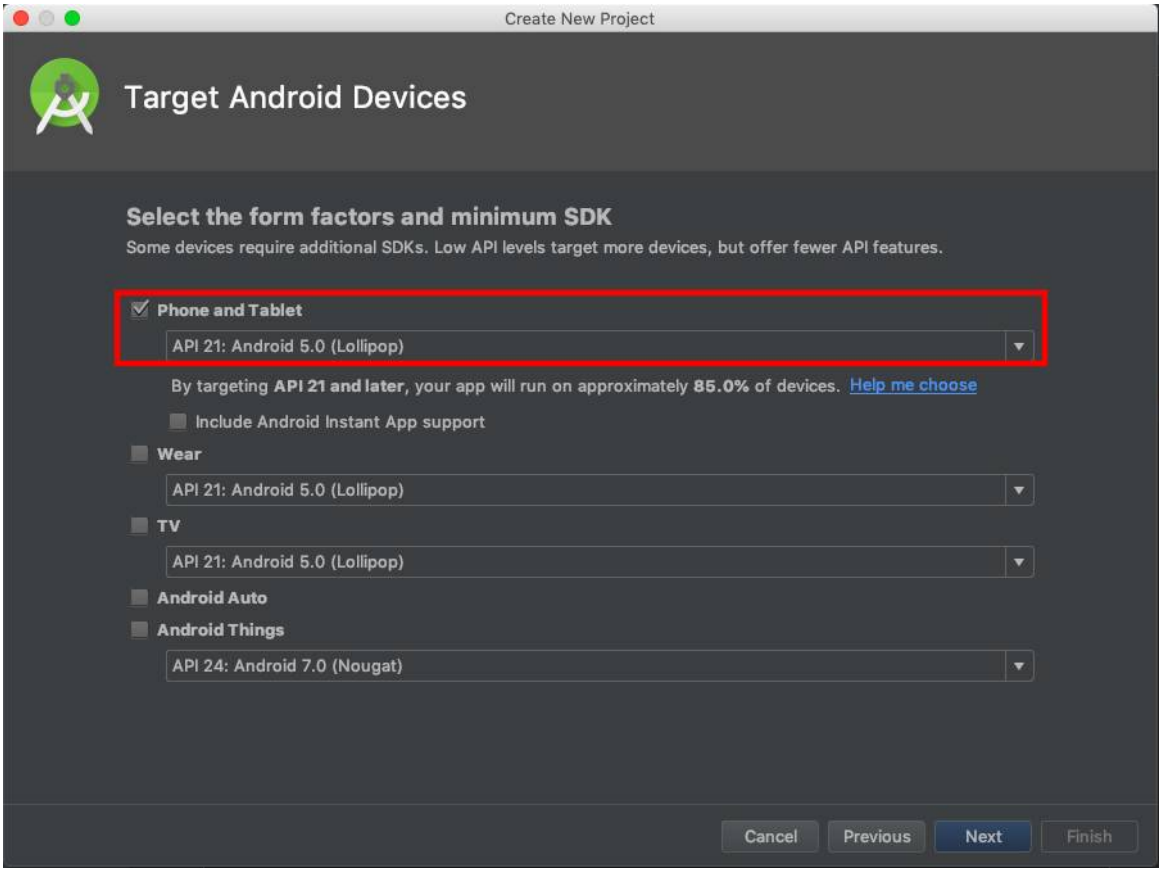

Imagen 2. Soporte del proyecto.

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**. 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.

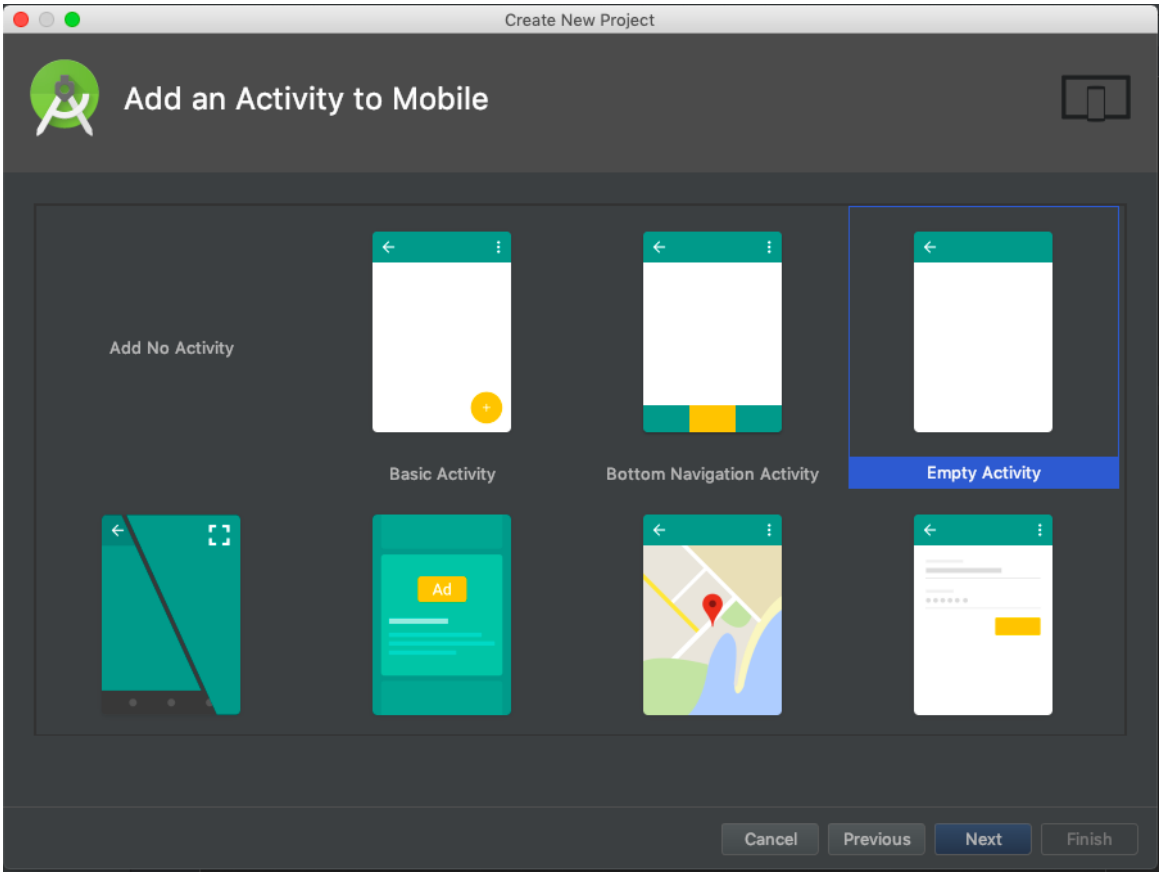

Imagen 3. Agregar actividad al proyecto.

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.

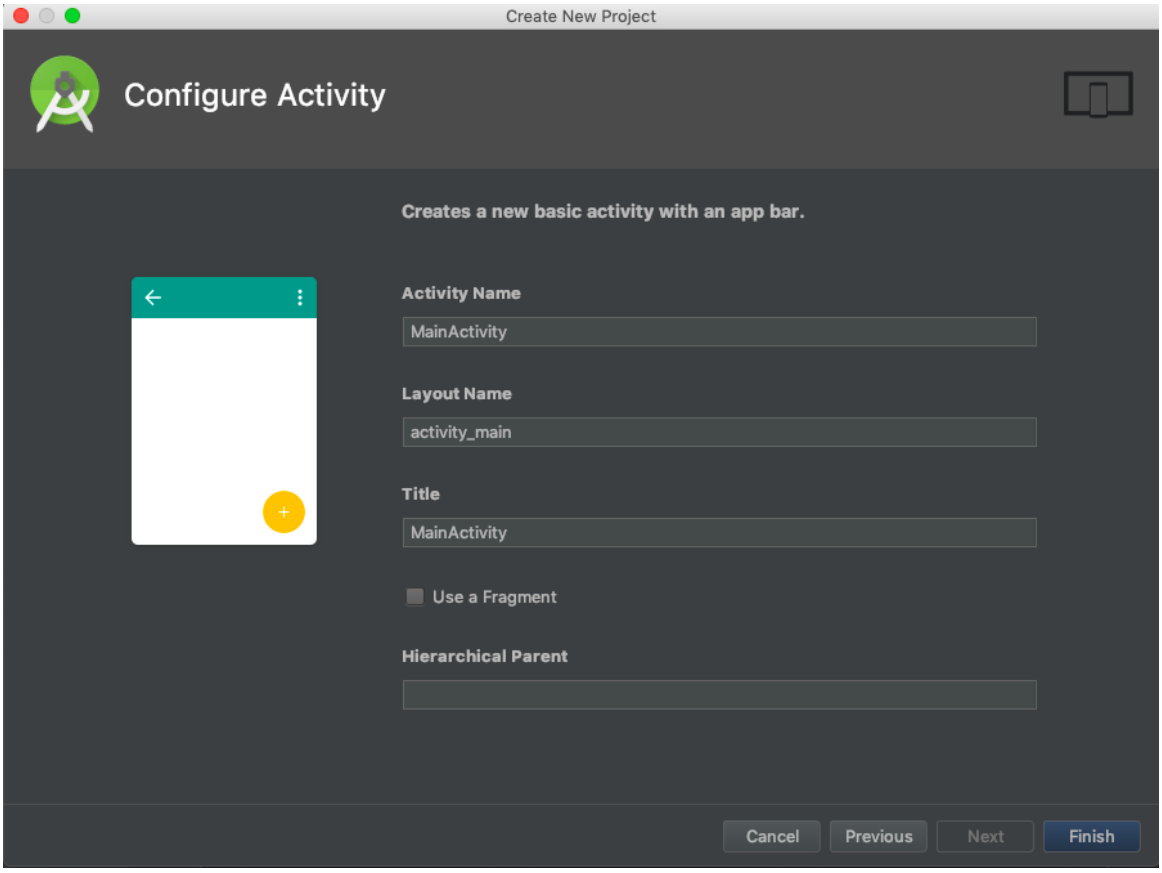

Imagen 4. Proyecto nuevo.

Acá en el último paso le podemos cambiar el nombre a esa activity, al layout y el título, en este caso nosotro, 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.

```
class MainActivity : AppCompatActivity() {

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

    }
}
```

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

Ahora veremos cómo crear una clase kotlin desde Android Studio.

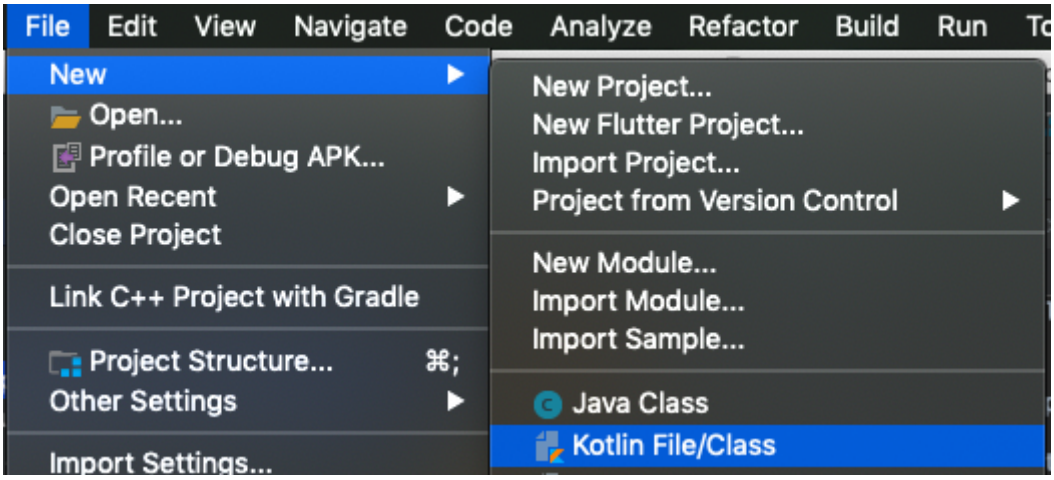

Imagen 5. Crear una clase Kotlin desde Android Studio.

Debemos hacer click en File/New/Kotlin file Class.

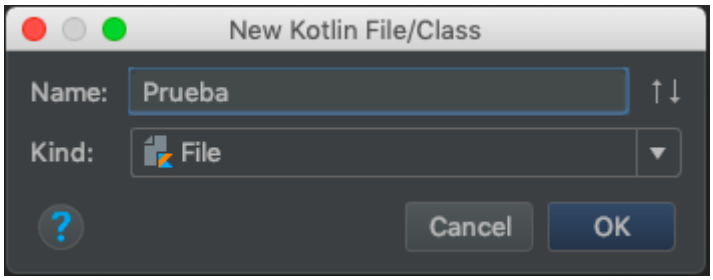

Imagen 6. Crear archivo llamado Prueba.

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
```

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 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. Para esto supongamos que tenemos la misma clase anterior Calculadora

```
public class Calculadora {

  public Calculadora() {}

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

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

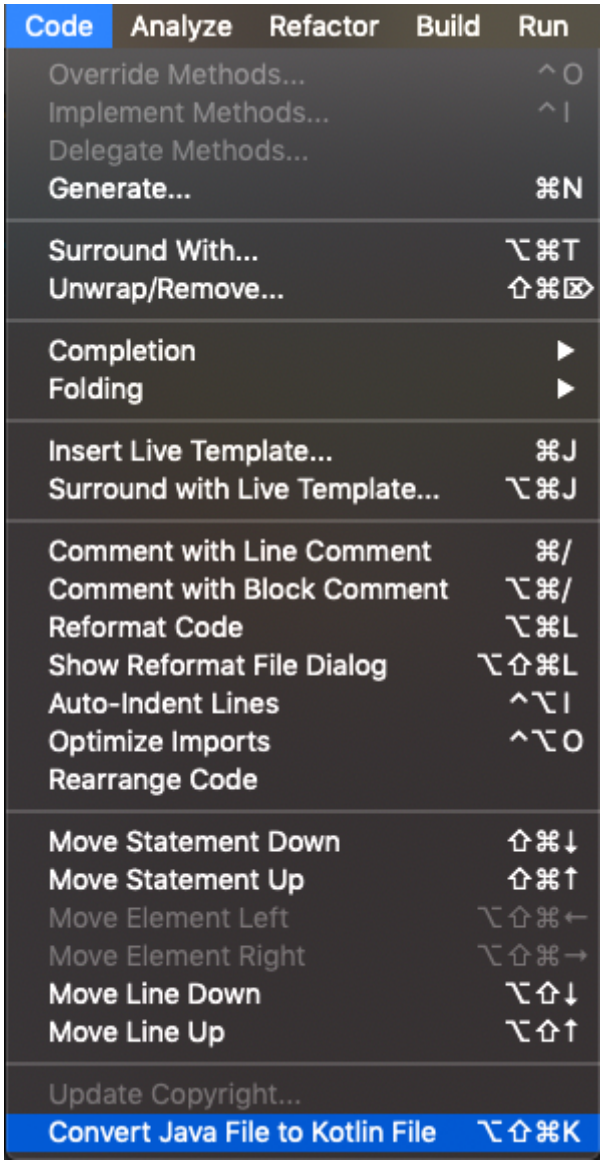

Imagen 7. Opciones del menú - Convertir archivo Java a Kotlin

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
    }
}
```

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

Competencias:

- Usando la Interoperabilidad entre Android y Kotlin

Contexto

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.

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.

Ahora veremos bien, como es el tema de la interoperabilidad con kotlin y como nos puede simplificar la vida en nuestro desarrollo en Android.

Interoperabilidad Java y Kotlin

¿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.

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.

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.

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.

```
<?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>
```

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)
  }
}
```

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;
  }
}
```

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)
  }
}
```

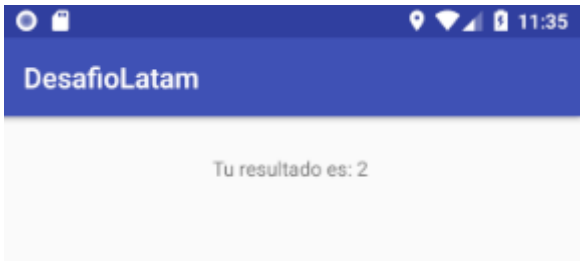

Imagen 8. Resultado al correr el proyecto.

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.

¿Por qué se usa textview.text y no textview.setText() que es el clásico método de Textview para setear un valor?

La razón de esto, es debido a que kotlin, para facilitar el código, adaptó el método setText(“”), por este nuevo textview.text = “”. De esta manera tenemos un código más simple.

Ventajas de usar Kotlin

Competencias:

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

Contexto

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.

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.

```
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;
  }

  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;
  }
```

```
    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)
```

Como podemos observar, Kotlin tiene muchas mejoras, que nos simplifican el desarrollo y esta son parte de las ventajas que profundizaremos en el siguiente capítulo.

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 generase un código engorroso. Para estos casos, podemos usar when.

La sentencia **when**, tiene la siguiente estructura:

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

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")
}
```

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")
}
```

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.

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\"")
```

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")
}
```

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"
}
```

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")
}
```

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

¿Que es un 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.

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.

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
```

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 ?
```

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
```

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
  .
  .
  .
}
```

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
```

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
```

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.

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.

Veamos el siguiente código de ejemplo.

```
var lista: ArrayList<String>? = null
.
.
.
adapter.setData(lista)
```

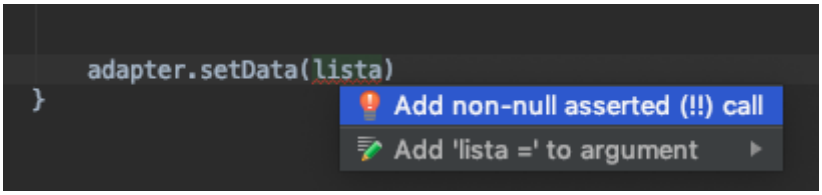

Imagen 9. Error de sintaxis.

Como vemos, el mismo IDE nos recomienda agregar el operador **!!**, ya que eso le indica que esta lista no será null, ya que nosotros nos estamos asegurando de ello, esta es una de las formas en que kotlin nos hace responsables sobre el código.

Imagen 10. Agregar operador `!!`.

Este es un detalle importante que tenemos que tener en cuenta cuando desarrollemos en Android usando Kotlin.

¿Qué pasaría si necesitamos poner algún valor null dentro de una lista?

La verdad en Kotlin se ha pensando en todo, veamos un ejemplo.

```
val listaDeRazas: List<String?> = listOf("bulldog", "labrador", null, "pitbull")
```

Usando el mismo operador **?**, podemos definir que nuestra lista contendrá elementos del tipo String, pero soporta null, maravillosamente simple.

Con esto ya vimos funciona el null safety en kotlin y cómo debemos darle uso a los operadores para hacer un correcto desarrollo de código y evitar errores.