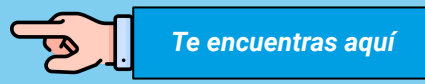
Kotlin para el desarrollo de aplicaciones

Kotlin y Android (Parte III)

{desafío}
latam_

Reconocer las principales características del lenguaje Kotlin para el desarrollo de aplicaciones móviles Android Nativo.

- Unidad 1: Kotlin para el desarrollo de aplicaciones.



- Unidad 2: Ciclo de vida de componentes.
- Unidad 3: Arquitectura en Android.
- Unidad 4: Programación asíncrona en Android.

{desafío} latam_

¿Qué aprenderás en esta sesión?

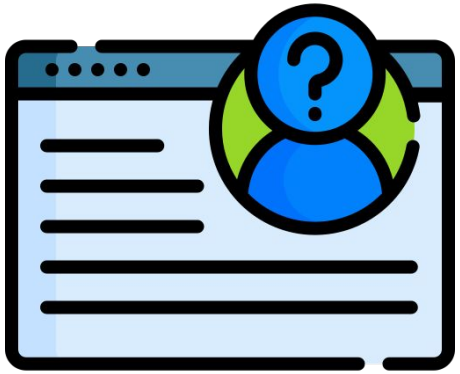
- *Codificar una aplicación básica utilizando Event Listeners y View Bindings acorde al lenguaje Kotlin*

{desafío} latam_

¿Recuerdas Kotlin y sus ventajas? ¿Puedes nombrar alguna?

/* Declaraciones anónimas de interfaces */

Interfaz Anónima



Kotlin se ha desarrollado sobre JVM, por lo que es compatible con la mayoría de las funciones de JVM. Java proporciona una característica llamada clases internas anónimas para manejar los casos en los que tenemos que crear un objeto de una clase con una ligera modificación, sin declarar una nueva subclase. Una clase interna anónima no tiene nombre, lo definimos directamente en la línea de instanciación.

Sin embargo, Kotlin usa expresiones de objetos para proporcionar la misma funcionalidad de subclase. En Kotlin, podemos crear una expresión de objeto de una interfaz implementando sus métodos abstractos.

Esta técnica de implementación se conoce como interfaz anónima.

Ejemplo

Interfaz Anónima

```
fun interface AnonymousInterface<T> {
    fun call(context: T)
}

fun main() {

    val a = AnonymousInterface<String> { someText ->
        println("Hola $someText!!!!!")
    }

    a.call("Android")
}

Resultado:
Hola Android!!!!!
```

{desafío} latam_

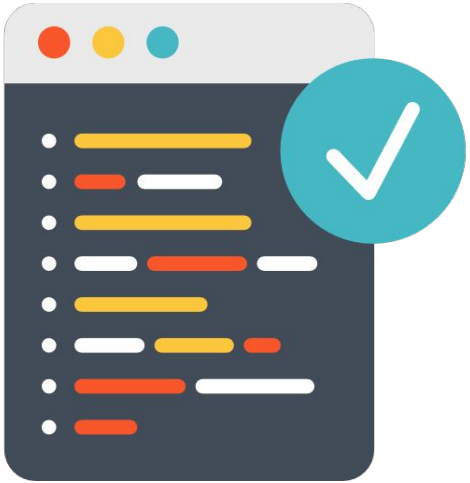
Ejemplo
Interfaz Anónima

Otro ejemplo haciendo uso de la misma definición:

```
val number = AnonymousInterface<ArrayList<Int>> { array ->
        array.forEach {
            println(it * 2)
        }
    }
    number.call(arrayListOf(1, 2, 3, 4, 5))

Resultado:
2
4
6
8
10
```

/* Clases abstractas para los event listeners */

Event Listeners en clases abstractas



Definir clases abstractas para los event listeners nos permite crear código escalable y más ordenado, tomemos el siguiente ejemplo:

Supongamos que cada vez que se cree un Activity, tengamos que agregar las siguientes interfaces de event listeners `OnClickListener`, `OnLongClickListener`, ¿no sería más fácil poder llamar todas esas interfaces con un solo llamado?

Event Listeners en clases abstractas

Supongamos que tenemos un Activity el cual extiende de BaseActivity (clase abstracta):

```
class MainActivity :
BaseActivity<ActivityMainBinding>() {

    override fun getViewBinding():
ActivityMainBinding =

ActivityMainBinding.inflate(layoutInflater)
}
```

Podemos crear otra clase la cual maneje las interfaces que necesitamos, por ejemplo:

```
abstract class BaseListener :
    View.OnClickListener,
    View.OnLongClickListener
```

Event Listeners en clases abstractas

El problema que se nos presenta en este caso, es que no podemos extender nuestro Activity de dos clases abstractas. Por tanto, lo que podemos hacer es que nuestra clase BaseActivity extienda de BaseListener, en vez de `AppCompatActivity()`

```
abstract class BaseActivity<T : ViewBinding> : BaseListener() {

    lateinit var binding: T

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = getViewBinding()
        setContentView(binding.root)
    }

    abstract fun getViewBinding(): T
}
```

Event Listeners en clases abstractas

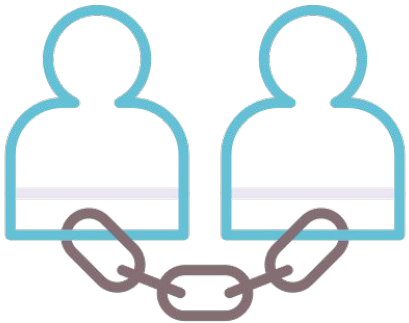
Finalmente, nuestro MainActivity queda de la siguiente forma:

```
class MainActivity : BaseActivity<ActivityMainBinding>() {

    override fun getViewBinding(): ActivityMainBinding =
        ActivityMainBinding.inflate(layoutInflater)

    override fun onClick(view: View?) {
    }

    override fun onLongClick(view: View?): Boolean {
    }
}
```

Después de esto, cada vez que necesitemos crear un Activity podemos extender de nuestra clase BaseActivity, la cual está hecha de todos los elementos que necesitamos.

/* Companion object */

Companion object



Los Companion object proporcionan un mecanismo para definir variables o funciones que están vinculadas conceptualmente a un tipo, pero que no están vinculadas a un objeto en particular.

Los objetos complementarios son similares al uso de la palabra clave `static` de Java para variables y métodos.

Ejemplo Companion object

Imaginemos que queremos pasar un dato desde un activity a un fragment, podemos hacerlo haciendo uso del companion object, por ejemplo:

En nuestro fragment (`HomeFragment`) definimos en companion object, en este caso decimos que queremos setear un entero:

```
companion object {
        @JvmStatic
        fun newInstance(number: Int) = HomeFragment().apply {
            arguments = Bundle().apply
{putInt(COMPANION_OBJECT_NUMBER, number)}
        }
      }
```

Ejemplo Companion object

Luego en el activity llamamos la función `newInstance` del `HomeFragment`

```
HomeFragment.newInstance(123456)
```

Finalmente en el `HomeFragment` dentro de `onAttach` llamamos el Bundle que se definió con el companion object:

```
arguments?.getInt(COMPANION_OBJECT_NUMBER)?
.let {
    number -> println(number)
}
```

¡Ahora practiquemos!

Ejercicio propuesto

Companion object

1. Descarga el archivo de ayuda (09 - Material de apoyo - Kotlin y Android (Parte III)
2. Abre el proyecto con Android Studio
3. Busca el archivo **`MainFragment`** y crea el companion object

```
companion object {
        @JvmStatic
        fun newInstance(number: Int) = MainFragment().apply {
            arguments = Bundle().apply {
                putInt(COMPANION_OBJECT_NUMBER, number)
            }
        }
    }
```

{desafío} latam_

Ejercicio propuesto

Companion object

1. Busca el `MainActivity` y utiliza `supportFragmentManager` para cargar `MainFragment` cada vez que inicie la app.

```
supportFragmentManager.beginTransaction()
            .replace(R.id.fragment_content_main,
MainFragment.newInstance(12345))
            .commitAllowingStateLoss()
```

2. En el `MainFragment`, dentro del método `onViewCreated` copia el siguiente código:

```
arguments?.getInt(COMPANION_OBJECT_NUMBER)?.let { number ->
    binding.tvTitle.text = number.toString()
}
```

- Ejecuta la app, podrás ver **12345** al centro de la pantalla

Reflexionemos…

¿Recuerdas las funciones lambdas? Compáralas con las interfaces anónimas

Reflexionemos…

¿Qué es la interfaz anónima y para qué sirve?

Reflexionemos...

¿Cuál es la función de los Event Listeners en las clases abstractas? ¿Puedes dar un ejemplo?

Próxima sesión...

- *Desarrollaremos la **guía de ejercicios** con la cual podrás practicar todo lo aprendido en estas sesiones.*

{desafío}
latam_

{desafío}
latam_

Academia de
talentos digitales