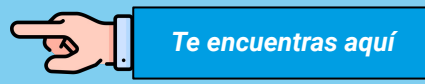
Kotlin para el desarrollo de aplicaciones

Kotlin y Android (Parte I)

{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?

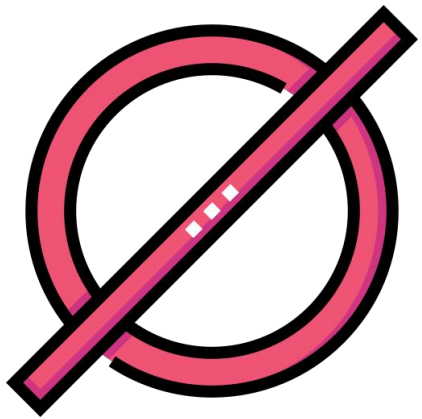
- *Codificar una aplicación básica utilizando Event Listeners y View Bindings acorde al lenguaje Kotlin*

{desafío} latam_

¿Has escuchado alguna vez sobre el Billion Dollar Mistake?

/* Kotlin null safety */

Tipos Nullable y Tipos non-null



- El sistema de tipos de Kotlin tiene como objetivo eliminar el peligro de las referencias nulas, también conocido como The Billion Dollar Mistake.
- Uno de los errores más comunes en muchos lenguajes de programación, incluido Java, es que acceder a un miembro de una referencia nula dará como resultado una excepción de referencia nula. En Java esto sería el equivalente a una **NullPointerException**, o NPE para abreviar.
- En Kotlin, el sistema de tipos distingue entre referencias que pueden contener valores nulos (referencias nullable) y aquellas que no pueden (non-null)

Ejemplo Tipos Nullable y Tipos non-null

Ejemplo, una variable regular de tipo String que **NO** puede contener nulo:

```
var a: String = "abc" // Inicialización regular que por defecto significa no nulo
a = null // error en tiempo de compilación
```

Para permitir nulos, puede declarar una variable como una cadena nullable escribiendo "**?**", en el caso de String sería, String**?**:

```
var b: String? = "abc" // el tipo String acepta valores nulos
b = null //NO hay error en tiempo de compilación
print(b)
```

Ejemplo Tipos Nullable y Tipos non-null

- Basándose en el ejemplo anterior, es posible hacer algo como:

```
val lenA = a.length
```

- Ya que "**a**" está definido como `var a: String = "abc",` lo que nos asegura que **a**`.length` no nos dará un NPE, sin embargo, aún queremos acceder a las mismas propiedades en "**b**"

Comprobación de null en condiciones

Recordemos de qué forma se definió `b`

```
var b: String? = "abc"
```

Es decir, "`b`" podría ser nulo

Para evitar un NPE, podemos validar si `b` es nulo o no, por ejemplo:

```
val len = if (b != null) b.length else -1
```

A continuación veremos diferentes formas de manejar valores nulos

Safe calls

Una opción para acceder a una propiedad en una variable nullable es usar el operador safe call **?**
Por ejemplo:

```
var b: String? = "abc"

val len = b?.length
```

En este caso, obtendremos el valor de **`b.length`** siempre que **`b`** no sea nulo

Recordando lo aprendido en la sesión anterior sobre scope functions, podemos hacer uso de **`let`**, para obtener el valor de **`b.length`** cuando este no es nulo.
Por ejemplo:

```
b?.length?.let { len ->
    println(len)
}
```

Elvis operator

Cuando tiene una referencia a un nullable, `b`, se puede decir que:

"si **b no es nulo**, úselo; de lo contrario, use algún valor **no nulo**"

La forma Java o más tradicional de escribir lo anterior sería:

```
if (b != null) {
    result = b.length
} else {
    result = -1
}
```

Kotlin por otra parte, te permite retornar el valor resultante de la condición, por ejemplo:

```
result = if (b != null) {
        b.length
    } else {
        -1
    }
```

En Kotlin, en vez de escribir la expresión `if` completa, se puede expresar haciendo uso de **Elvis operator**.

Ejemplo

Del ejemplo anterior tomamos el siguiente bloque if

```
val len: Int
if (b != null) {
    len = b.length
} else {
    len = -1
}
println("b.length = $len")
```

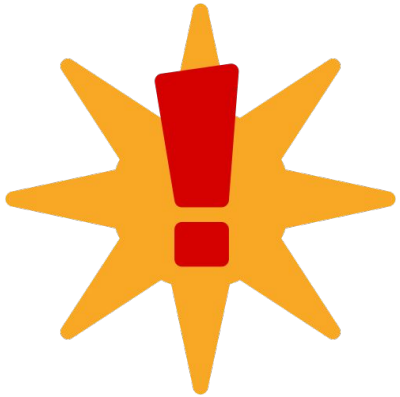
Al transformarlo a un estilo más propio de Kotlin nos queda:

```
val len: Int = if (b != null) {
    b.length
} else {
    -1
}
println("b.length = $len")
```

Finalmente, haciendo uso de Elvis operator:

```
val len: Int = b?.length ?: -1
println("b.length = $len")
```

!! operator u Operador de aserción



El operador de aserción no nulo(`!!`) convierte cualquier valor en un tipo no nulo y lanza una excepción si el valor es nulo.

Puedes escribir `b!!`, y esto devolverá un valor no nulo de `b` (por ejemplo, una cadena en nuestro ejemplo) o arrojará un NPE si b es nulo, ejemplo:

```
val len = b!!.length
```

Se debe tener mucho cuidado al utilizar esta opción, ya que se asume que, si bien, el valor nulo está controlado, la excepción también debe ser controlada.

Advertencia: Android Studio no mostrará ningún error en tiempo de compilación, sin embargo, esto no significa que en tiempo de ejecución no haya errores en el futuro.

Safe cast

Si el objeto no es del tipo de destino, las conversiones(cast) regulares pueden generar una ClassCastException. En ese caso, se puede usar conversiones seguras (safe cast) que devuelvan un valor nulo si el intento no tuvo éxito.
Por ejemplo:

```
open class Fruit
class Apple: Fruit()
class Banana: Fruit()

val apple = Apple()
apple as? Fruit // apple es tipo Fruta, OK
apple as? Banana // apple no es tipo Banana, por lo que retornará nulo
```

{desafío} latam_

Colecciones de un tipo nullable

En la sesión *Características del Lenguaje Kotlin (Parte II)*, vimos cómo se podían definir un list, set y map. Supongamos entonces que queremos crear una lista de items, pero que sea capaz de almacenar valores nulos:

Para hacer esto vemos el siguiente ejemplo:

```
val numbers = listOf("uno", "dos", "tres", "cuatro")
```

El ejemplo anterior es lo mismo que:

```
val numbers = listOf<String>("uno", "dos", "tres", "cuatro")
```

Si queremos que almacene valores nulos, entonces debemos especificar que el tipo String es nullable, ejemplo:

```
val numbers = listOf<String?>("uno", "dos", "tres", "cuatro", null)
```

Ejemplo

En el caso de que queramos aplicar lo anterior a una colección map seria:

```
val numbers = mapOf<Int, String>(1 to "uno", 2 to "dos", 3 to "tres", 4 to null) → error en tiempo de compilación
```

Para corregir el error el tipo “**`String`**” a “**`String?`**”

```
val numbers = mapOf<Int, String?>(1 to "uno", 2 to "dos", 3 to "tres", 4 to null) → ahora podemos almacenar valores nulos
```

¡Hagamos un ejercicio paso a paso!

Ejercicio guiado

Crea una variable tipo map y nula, con clave tipo Int, y que permita almacenar los siguientes tipos de dato, Int, String, Boolean, (incluido nulo)

- Creamos la variable tipo map:
```
var valueMap: Map<> = null
```
- Hacemos que permita ser nula
```
var valueMap: Map<>? = null
```
- Definimos la el tipo de la clave
```
var valueMap: Map<Int,>? = null
```
- Usamos tipo Any para permitir almacenar cualquier tipo de dato
```
var valueMap: Map<Int,Any>? = null
```
- Incluimos null en los tipos de datos
```
var valueMap: Map<Int, Any?>? = null
```

Ejercicio guiado

- Comprobemos que **`valueMap`** quedó bien definido

```
var valueMap: Map<Int, Any?>? = null
```

- Creamos algunos datos de prueba en nuestra variable:

```
valueMap = mapOf(1 to 1, 2 to "dos", 3 to true, 4 to null)
println("map: $valueMap")
```

El resultado debiera ser el siguiente:

```
Resultado: map: {1=1, 2=dos, 3=true, 4=null}
```

{desafío} latam_

Reflexionemos

- Cuando se dice que Kotlin es null safe, no significa que no existan nulos en la programación.

- Haz uso de lo aprendido en la clase *Características del Lenguaje Kotlin (Parte II)*, **`let`** y **`run`** son especialmente útiles al momento de trabajar con valores nulos.

- ¿Sabes en qué se relaciona Elvis Operator con Elvis Presley? Toma un tiempo para leer sobre su historia
- https://en.wikipedia.org/wiki/Elvis_operator

De todo lo aprendido el día de hoy ¿Qué me resultó más difícil y por qué?

{desafío} latam_

Próxima sesión...

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

{desafío}
latam_

Academia de
talentos digitales