TEMA: Kotlin, Parte 2.

Objetivo.

Codificación de aplicaciones de Android funcionales en Kotlin: seguridad null, lambdas, funciones de extensión y singletons.

Introducción.

Utilizar Kotlin para recortar código repetitivo en los proyectos de Android, ampliar las clases existentes con nuevas funcionalidades y evitar las `NullPointerExceptions`.

Es muy común encontrarse con resultados del tipo `NullPointerException` mientras se prueba el código. Intentar utilizar una referencia de objeto que tiene un valor nulo es una gran fuente de errores en diversos lenguajes de programación. Java 8 agregó la anotación `@NonNull` específicamente para tratar de corregir esta falla en su sistema de tipos.

Java permite establecer cualquier variable de un tipo de referencia en `null`, pero si en algún momento se intenta utilizar una referencia que apunte a `null`, entonces la variable no sabrá dónde buscar porque el objeto en realidad no existe. En este punto, la JVM no puede continuar la ruta de ejecución normal: la aplicación fallará y encontrará una `NullPointerException`. Intentar llamar a un método en una referencia `null` o intentar acceder a un campo de referencia `null` activará una `NullPointerException`.

La seguridad de `null` está integrada en el lenguaje de Kotlin. El compilador de Kotlin no permite asignar un valor `null` a una referencia de objeto, ya que comprueba específicamente su código para la presencia de posibles objetos nulos durante la compilación. Si el compilador de Kotlin detecta que se puede lanzar una `NullPointerException` en el tiempo de ejecución, el código fallará en tiempo de compilación.

La única excepción es que desencadena una `NullPointerException` utilizando incorrectamente uno de los operadores especiales de Kotlin, o si usa uno de estos operadores para activar deliberadamente una `NullPointerException`.

Uso de la seguridad `null`

En Kotlin, todas las variables se consideran no anulables de manera predeterminada, por lo que al intentar asignar un valor `null` a una variable dará como resultado un error de compilación. Por ejemplo, lo siguiente no se compilará:

```
var ejemplo: String = null
```

Si se desea que una variable acepte un valor `null`, se deberá marcar explícitamente esa variable como `null`. Esto es exactamente lo contrario de Java, donde cada objeto se considera anulable de manera predeterminada.

Si desea declarar explícitamente que una variable puede aceptar un valor `null`, entonces deberá agregar un **?** al tipo de variable. Por ejemplo, se compilará lo siguiente:

```
var ejemplo: String? = null
```

Cuando el compilador ve este tipo de declaración, reconoce que se trata de una variable que admite `null` y lo trata de forma un tanto diferente, evitando que invoque un método o acceda a una propiedad en esta referencia anulable, lo que ayuda a evitar las `NullPointerExceptions` . Por ejemplo, lo siguiente no se compilará:

```
var ejemplo: String? = null
ejemplo.size
```

Aunque el diseño de Kotlin significa que es difícil encontrar `NullPointerExceptions` que se originan en el código de Kotlin, si el proyecto presenta una mezcla de Kotlin y Java, las partes de Java del proyecto aún pueden ser una fuente de `NullPointerExceptions`.

Si se trabaja con una combinación de archivos Kotlin y Java, Kotlin incluye una gama de operaciones especiales diseñadas para manejar cualquier valor nulo que encuentre.

1. **El operador ? de llamada segura**

El operador **?** de llamada segura ofrece una forma de tratar con las referencias que potencialmente podrían contener un valor `null`, al tiempo que garantiza que cualquier llamada a esta referencia no dará como resultado una `NullPointerException`.

Cuando se agrega un operador **?** de llamada segura a una referencia, esa referencia se probará para un valor nulo. Si el valor es nulo, se devuelve `null`; de lo contrario, la referencia del objeto se usará como se pretendía originalmente. Si bien puede lograr el mismo efecto utilizando una instrucción `if`, el operador **?** de llamada segura permite realizar el mismo trabajo en mucho menos código.

Por ejemplo, lo siguiente devolverá `a.size` solo si `a` no es nulo; de lo contrario, devolverá `null`:

```
a?.size
```

También puede encadenar operadores de llamada segura juntos:

```
val number = person?.address?.street?.number
```

Si alguna de las expresiones es nula, el resultado será `null`; de lo contrario, el resultado será el valor solicitado.

2. **El operador Elvis `?:`**

Algunas veces se podría tener una expresión que potencialmente podría contener un valor nulo, pero sin lanzar una `NullPointerException`, incluso si este valor resulta ser `null`.

En estas situaciones, se puede usar el operador Elvis `?:` de Kotlin para proporcionar un valor alternativo que se usará cada vez que el valor resulte nulo, lo cual es una buena forma de evitar la propagación de valores nulos en el código. Por ejemplo:

```
fun setName (name: String?) {
username = name?: "N/A"
```

Aquí, si el valor a la izquierda del operador de Elvis no es nulo, entonces se devuelve el valor de `name` del lado izquierdo. Pero si el valor a la izquierda del operador de Elvis es `null`, se devolverá la expresión de la derecha, que en este caso es `N/A`.

3. **El operador !!**

Si alguna vez se quiere obligar al código de Kotlin a lanzar una `NullPointerException` de estilo Java, se puede usar el operador **!!**. Por ejemplo:

```
val number = firstName!!.length
```

Aquí, se utiliza el operador `!!` para afirmar que la variable `firstName` no es nula. Siempre que `firstName` contenga una referencia válida, el valor de la variable se asigna con la longitud de la cadena. Si `firstName` no contiene una referencia válida, entonces Kotlin lanzará una `NullPointerException`.

Expresiones Lambda

Una expresión lambda representa una función anónima. Las lambdas son una buena forma de reducir la cantidad de código necesario para realizar tareas en el desarrollo de Android; por ejemplo, escribir escuchas y devoluciones de llamada.

Java 8 ha introducido expresiones lambda nativas, y ahora son compatibles con Android Nougat, aunque esta característica no es exclusiva de Kotlin pero se pueden usar en todo el proyecto.

Una expresión lambda consta de un conjunto de parámetros, un operador lambda (**->**) y un cuerpo de función, dispuestos en el siguiente formato:

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

Al construir expresiones lambda en Kotlin, se deben tener en cuenta las siguientes reglas:

- La expresión lambda debe estar rodeada de llaves.
- Si la expresión contiene algún parámetro, se les debe declarar antes del símbolo **->** flecha.
- Si se trabaja con múltiples parámetros, se les debe separar con comas.
- El cuerpo se escribe después del signo **->**.

El principal beneficio de las expresiones lambda es que permiten definir funciones anónimas y luego pasar estas funciones de inmediato como una expresión. Esto permite realizar muchas tareas de desarrollo comunes de manera más precisa pues ya no se necesita escribir la especificación de la función en una clase o interfaz abstracta, lo que había sido tradicionalmente fallo en Android.

Por ejemplo, agregar un escucha de un clic a un botón. En Java 7 y versiones anteriores, esto normalmente requeriría el siguiente código:

```
button.setOnClickListener (new View.OnClickListener () {
   @Override
   public void onClick (Ver v) {
       Toast.makeText (this, "Botón digitado", Toast.LENGTH_LONG) .show ();
   }
});
```

Sin embargo, las funciones lambda de Kotlin permiten configurar un escucha de clic con una sola línea de código:

```
button.setOnClickListener( {view -> toast ("Botón digitado")} )
```

Inclusive, si una función toma a otra función como último parámetro, se le puede pasar fuera de la lista de paréntesis:

```
button.setOnClickListener(){ toast("Botón pulsado") }
```

Y si la función sólo tiene un parámetro que es una función, se pueden eliminar los paréntesis por completo:

```
button.setOnClickListener{ toast("Botón digitado") }
```

Funciones de extensión

Al igual que en C#, Kotlin permite agregar nuevas funcionalidades a las clases existentes que de otro modo no se podrían modificar. Entonces, si a una clase le falta un método útil, se le puede agregar a través de una función de extensión.

Para crear una función de extensión se prefija el nombre de la clase que se desea extender con el nombre de la función que se está creando. Por ejemplo:

```
fun AppCompatActivity.toast(msg: String){
   Toast.makeText(this, msg, Toast.LENGTH_LONG).show()
}
```

Notar que la palabra clave `this` dentro de la función de extensión corresponde a la instancia de `AppCompatActivity` con la que se llama a `.today`.

En este ejemplo, sólo se necesitan importar las clases `AppComptActivity` y `Toast` en el archivo **.kt** y ya se está listo para llamar a la notación **.** (punto) en instancias de la clase extendida.

En lo que respecta al desarrollo de Android, es posible que las funciones de extensión sean particularmente útiles para dar a los `ViewGroups` la capacidad de inflarse. Por ejemplo:

```
fun ViewGroup.inflate (
    @LayoutRes layoutRes: Int,
    attachToRoot: Boolean = false): View{

    return LayoutInflater
        .from(context)
       .inflate(layoutRes, this, attachToRoot)
}
```

Entonces, en lugar de tener que escribir lo siguiente:

```
val view = LayoutInflater
   .from(parent)
   .inflate(R.layout.activity_main, parent, false)
```

Simplemente se puede usar su función de extensión:

```
val view = parent.inflate(R.layout.activity_main)
```

Objetos Singleton

En Java, la creación de singletons ha sido típicamente muy detallada, lo que requiere que se cree una clase con un constructor privado y luego crear esa instancia como un atributo privado.

El resultado final suele ser algo como esto:

```
class public Singleton {
  private static Singleton singleton = new Singleton ();

  private Singleton() {}

  public static Singleton getInstance () {
     return singleton;
  }
```

En lugar de declarar una clase, Kotlin permite definir un sólo objeto, que es semánticamente el mismo que un singleton, en una línea de código:

```
object KotlinSingleton {}
```

A continuación, se puede utilizar este singleton de inmediato, por ejemplo:

```
object KotlinSingleton {
   fun myFunction (){ [...] }
}
KotlinSingleton.myFunction ()
```

Ahora, se tiene una combinación de códigos Java y Kotlin mediante el uso de una gama de operadores especiales. También se pueden usar expresiones lambda para simplificar algunas tareas de desarrollo comunes en Android y agregar nuevas funcionalidades a las clases existentes.

Ejercicios.

Desarrollar un ejemplo de cada uno de los siguientes operadores.

1. Operador ? de llamada segura.
2. Operador Elvis ?:
3. Operador !!
4. Expresión Lambda.
5. Función de extensión.
6. Un objeto Singleton.

NOTA: Entregar los ejercicios en una carpeta con la carpeta de los resultados. Sintaxis AlumnoEjercicio.zip.