Seguridad contra nulos en Kotlin: Despídete del error del billón de dólares

1. Introducción

Si has llegado hasta aquí, ya sabes cómo crear tus propias clases, agrupar objetos y organizar tu código. ¡Enhorabuena! Pero hay un enemigo invisible que acecha a casi todos los programadores cuando empiezan a desarrollar aplicaciones reales: la ausencia de datos o, como se le conoce técnicamente, el valor `null`.

En lenguajes más antiguos como Java, si intentas acceder a una propiedad de un objeto (por ejemplo, el correo electrónico de un usuario) y resulta que ese usuario no existe en la base de datos (es `null`), tu programa se cuelga abruptamente lanzando un fatídico `NullPointerException`. Es un fallo tan común, molesto y

destructivo que a su propio inventor le gusta llamarlo «el error del billón de dólares».

Aquí es donde Kotlin brilla con luz propia. Para evitar que tu aplicación explote en las manos de tus usuarios, Kotlin implementa en su núcleo la **seguridad contra nulos (Null Safety)**. Este sistema es como un escudo protector que detecta los posibles problemas mientras escribes tu código (en tiempo de compilación), obligándote a manejarlos de forma elegante mucho antes de que el código llegue a ejecutarse.

2. ¿Qué son los tipos anulables (Nullable types)?

Por defecto, en Kotlin está estrictamente prohibido que una variable normal guarde un valor nulo. El compilador, simplemente, no te dejará hacerlo. Y esto es fantástico para dormir tranquilos.

Sin embargo, a veces es necesario representar que algo «falta», está vacío o aún no se ha configurado (como un segundo apellido opcional en un formulario). Para decirle a Kotlin que una variable *sí* tiene permiso para ser nula, debes declarar un **tipo anulable** añadiendo explícitamente el símbolo de interrogación `?` justo después de su tipo de dato.

```
fun main() {
    // neverNull es un String normal, jamás aceptará nulos
    var neverNull: String = "Esto no puede ser nulo"

    // Si intentas hacer esto, el compilador te lanzará un error
    // neverNull = null

    // nullable es un String? (String anulable)
    var nullable: String? = "Aquí puedes guardar texto"

    // Esto es perfectamente válido
    nullable = null
}
```

Si intentas acceder directamente a una propiedad de una variable que podría ser nula, por ejemplo `nullable.length`, Kotlin te detendrá y te dará un error. ¿Por qué? Porque podría ser peligroso. Necesitas utilizar herramientas seguras.

3. Comprobación tradicional de nulos

La forma más básica de manejar un tipo anulable es usar una clásica estructura condicional `if` para asegurarte de que hay algo ahí dentro antes de usarlo.

```
fun describirTexto(texto: String?): String {
    // Kotlin comprueba primero que la variable no esté vacía
    if (texto != null && texto.length > 0) {
        return "El texto tiene ${texto.length} caracteres"
    } else {
        return "El texto está vacío o es nulo"
    }
}
```

> *A tener en cuenta: Cuando realizas una comprobación manual como `if (texto != null)`, el compilador de Kotlin es lo bastante inteligente como para aplicar un «smart cast» (conversión inteligente). Dentro de ese bloque `if`, Kotlin tratará automáticamente a tu variable anulable como si fuera totalmente segura, permitiéndote acceder a `.length` sin que te salte ningún error.*

4. Llamadas seguras (?.)

Si tuviéramos que escribir un `if` cada vez que usamos algo que puede ser nulo, nuestro código sería larguísimo y muy difícil de leer. Afortunadamente, a Kotlin le encanta el código conciso.

Para acceder de forma rápida a las propiedades de un objeto que podría contener un valor nulo, utilizamos el operador de **llamada segura** `?.`.

Este operador lo hace todo por detrás: intenta leer la propiedad que le pides, y si resulta que el objeto original es nulo, en lugar de crashear tu aplicación, simplemente **devuelve** `null` de forma pacífica y sigue ejecutando el resto de tu código.

```
fun longitudDelTexto(texto: String?): Int? = texto?.length

fun main() {
```

```
    val miTextoNulo: String? = null
    println(longitudDelTexto(miTextoNulo))
    // Salida por consola: null
}
```

Una de las características más potentes de este operador es que **las llamadas seguras se pueden encadenar**. Si quieres acceder a un dato que se encuentra escondido muy profundamente en tu estructura de clases, puedes hacerlo así:

```
val paisDelJefe = empleado.empresa?.departamento?.jefe?.pais
```

Si el empleado no tiene empresa, o la empresa no tiene departamento, el viaje se detiene ahí mismo y la variable `paisDelJefe` guardará simplemente un `null`, sin pánicos ni errores de ejecución.

5. El operador Elvis (?:)

Muchas veces no queremos que el resultado final de nuestra llamada sea `null`. Es muy común querer proporcionar un **valor por defecto** si las cosas fallan. Para esto tenemos al espectacular **operador Elvis** `?:` (se llama así porque si giras la cabeza a la izquierda, el símbolo recuerda al mítico tupé de Elvis Presley).

La estructura es muy simple: escribes a la izquierda del Elvis lo que quieres intentar leer, y a la derecha escribes el valor de respaldo que quieres devolver en caso de que lo de la izquierda haya resultado ser nulo.

```
fun main() {
    val texto: String? = null

    // Si texto?.length es nulo, usamos el 0 como comodín
    val longitud = texto?.length ?: 0

    println("La longitud es: $longitud")
    // Salida por consola: La longitud es: 0
}
```

6. La ejecución condicionada con let (?.let)

¿Qué ocurre si quieres ejecutar un bloque entero de código *únicamente* si tu variable no es nula? Combinando la llamada segura `?.` con la función `let`, construimos un entorno hermético y totalmente blindado.

```
1.  fun main() {
2.      val correo: String? = "hola@empresa.com"
3.
4.      // Todo lo que hay dentro de las llaves solo se ejecuta si hay
    correo
5.      correo?.let { correoSeguro ->
6.          println("Enviando mensaje de bienvenida a $correoSeguro")
7.      }
8.  }
```

> *La regla de oro: Acostúmbrate a usar siempre llamadas seguras `?.` o el operador Elvis `?:` para lidiar con tipos anulables. Únicamente hay un escenario en el que podrías forzar la ejecución del código, usando el operador de aserción `!!` (la doble exclamación). Este operador fuerza la lectura de la variable ignorando la seguridad; si el valor resulta ser nulo, tu aplicación se cerrará inmediatamente. Úsalo solo si pondrías la mano en el fuego de que ese dato existe, o mejor aún, ¡evítalo a toda costa en tu día a día!*

7. Ejercicios

Aquí tienes varios ejercicios de menor a mayor dificultad (el primero es una traducción adaptada de la documentación oficial de Kotlin y el resto son extras para asentar los conceptos que hemos ampliado). Abre tu Kotlin Playground y realiza los siguientes ejercicios.

7.1. El empleado sin sueldo

Tienes la función `employeeById` que te da acceso a la base de datos de los empleados de una compañía según su ID. Por desgracia, la función devuelve un tipo `Employee?`, por lo que el resultado puede ser `null` si el empleado no existe. Tu

objetivo es escribir el código de la función `salaryById(id: Int)` para que devuelva el salario del empleado si lo encuentra, o el valor `0` si el empleado ha desaparecido de los registros.

```
data class Empleado(val nombre: String, var salario: Int)

fun empleadoPorId(id: Int) = when(id) {
    1 -> Empleado("Pedro", 20)
    2 -> null
    3 -> Empleado("Juan", 21)
    4 -> Empleado("Ana", 23)
    else -> null
}

fun salarioPorId(id: Int): Int {
    // Escribe tu código aquí (Pista: usa llamada segura + operador Elvis)
}

fun main() {
    // Esto debería imprimir la suma total de los salarios existentes (64)
    println((1..5).sumOf { id -> salarioPorId(id) })
}
```

7.2. El formulario de contacto

Imagina que estás programando un formulario de registro. Crea una función llamada `obtenerLongitudTelefono` que reciba un número de teléfono (`String?`) que puede ser nulo. Utiliza una llamada segura y el operador Elvis en **una sola línea** para devolver el número de caracteres del teléfono, o `0` si el usuario decidió no teclear ninguno.

```
// Escribe aquí tu función obtenerLongitudTelefono

fun main() {
    val telefono1: String? = "654321987"
    val telefono2: String? = null

    println(obtenerLongitudTelefono(telefono1)) // Debería imprimir 9
    println(obtenerLongitudTelefono(telefono2)) // Debería imprimir 0
}
```

7.3. La cadena de mando

A veces los objetos contienen a otros objetos que, a su vez, también pueden ser nulos. Declara tres *data classes*:

1. `Jefe` (con una propiedad `nombre` de tipo String).
2. `Departamento` (con una propiedad `jefe` de tipo `Jefe?`).
3. `Empresa` (con una propiedad `departamento` de tipo `Departamento?`).

Luego, en tu función `main()`, crea una instancia de `Empresa` donde el departamento sea `null`. Usando **llamadas seguras encadenadas**, intenta recuperar el nombre del jefe. Si en algún momento la cadena se rompe por un nulo, haz que se asigne automáticamente el texto *«Sin jefe asignado»* usando el operador Elvis. Imprime el resultado final.

```
// Escribe aquí tus clases

fun main() {
    // 1. Crea una Empresa con departamento = null
    // 2. Encadena las llamadas e imprime el resultado
}
```

7.4. El sistema de alertas

Tienes un mensaje de alerta opcional. Usando la llamada segura `?.` junto a la función `let`, haz que el sistema imprima en mayúsculas (usando el método `.uppercase()`) el texto de la alerta **solo y exclusivamente** si este no es nulo.

```
fun main() {
    val alertaActiva: String? = "Peligro de sobrecalentamiento en el núcleo"
    val alertaApagada: String? = null

    // Escribe aquí tu código usando let para procesar alertaActiva

    // Y luego repite exactamente lo mismo para alertaApagada
    // (este segundo bloque no debería imprimir nada por consola)
}
```

8. Soluciones a los ejercicios

No hagas trampas. Échale un vistazo a las soluciones únicamente cuando te hayas peleado un buen rato con tu editor de código y el compilador te haya gritado un par de veces.

8.1. El empleado sin sueldo

```
data class Empleado(val nombre: String, var salario: Int)

fun empleadoPorId(id: Int) = when(id) {
    1 -> Empleado("Pedro", 20)
    2 -> null
    3 -> Empleado("Juan", 21)
    4 -> Empleado("Ana", 23)
    else -> null
}

// Usamos la llamada segura '?.salario' junto al valor por defecto '?: 0'
fun salarioPorId(id: Int): Int = empleadoPorId(id)?.salario ?: 0

fun main() {
    println((1..5).sumOf { id -> salarioPorId(id) })
}
```

8.2. El formulario de contacto

```
fun obtenerLongitudTelefono(telefono: String?): Int {
    return telefono?.length ?: 0
}

fun main() {
    val telefono1: String? = "654321987"
    val telefono2: String? = null

    println(obtenerLongitudTelefono(telefono1))
    println(obtenerLongitudTelefono(telefono2))
}
```

8.3. La cadena de mando

```
data class Jefe(val nombre: String)
data class Departamento(val jefe: Jefe?)
data class Empresa(val departamento: Departamento?)

fun main() {
    // Creamos nuestra empresa pasándole un departamento nulo
    val miEmpresa = Empresa(departamento = null)

    // Encadenamos con '?.'. Como el departamento es null, salta directamente al Elvis
    val nombreDelJefe = miEmpresa.departamento?.jefe?.nombre ?: "Sin jefe asignado"

    println(nombreDelJefe)
}
```

8.4. El sistema de alertas

```
fun main() {
    val alertaActiva: String? = "Peligro de sobrecalentamiento en el núcleo"
    val alertaApagada: String? = null

    alertaActiva?.let { mensaje ->
        println(mensaje.uppercase())
    }

    // Como esta variable es null, la ejecución ignorará todo el bloque let
    alertaApagada?.let { mensaje ->
        println(mensaje.uppercase())
    }
}
```