Metodologías de Diseño y Programación Avanzadas

Ignacio Slater Muñoz
reachme@ravenhill.cl

Contents

1. Unidad 1: Introducción 3
1.1. Kotlin 3
1.1.1. A Taste of Kotlin 3
1.2. Lo básico 4
1.2.1. Expresiones vs. Declaraciones 4
1.2.2. Declaración de funciones 4
1.2.3. Declaración de variables 5
1.2.4. Expresión `if` 7
1.2.5. Expresión `when` 8
1.2.6. Declaración `for` 9
1.2.7. Declaración `while` 10
1.2.8. Rangos 11
1.3. Null-Safety 12
1.3.1. El Problema de los Punteros Nulos 12
1.3.2. Motivación 12
1.3.3. Enfoques en Diferentes Lenguajes 12
1.3.4. Solución en Kotlin: Seguridad de Tipos frente a Nulos 13
1.4. Manejo de Entrada de Usuario en Kotlin 15
1.4.1. Funciones de Lectura en Kotlin 15
1.4.2. Ejemplo de Uso de `readlnOrNull()` 15
1.5. Build systems 15
1.5.1. Usos Comunes 16
1.5.2. Ejemplos de Build Systems 16
1.5.3. Gradle 16
1.6. Análisis Estático 21
1.6.1. Objetivos del Análisis Estático 21
1.6.2. Detekt 21
1.7. Respuestas 22
2. Unidad 2: Programación orientada a objetos 23
2.1. Objetos 23
2.2. Clases 23
2.3. Null-safety: Inicialización tardía 24
2.3.1. Uso de `lateinit` 24
2.3.2. Uso de `Delegates.notNull()` 24
2.4. Encapsulamiento 24
2.5. Herencia 25
2.5.1. Propósito y Beneficios de la Herencia 25
2.6. Constructores 25
2.6.1. Constructor Primario 25
Bibliography 26

Iniciar un proyecto requiere considerar varios *aspectos críticos*, incluyendo la selección de tecnologías y la planificación del tiempo y estructura del proyecto. Sin embargo, un factor frecuentemente subestimado es el cambio. El cambio es inevitable y debe considerarse desde el inicio para evitar que el proyecto se vuelva obsoleto antes de su finalización.

Esta consideración es igualmente crucial al desarrollar un curso. Si no anticipamos los cambios, el contenido del curso podría quedar desactualizado antes de que esté completo.

Por esta razón, hemos diseñado este curso para ser modular y fácil de actualizar. Cada unidad se compone de secciones independientes que pueden ser modificadas sin afectar el resto del contenido. De esta forma, podemos mantener el curso actualizado y relevante para lxs estudiantes.

Al hablar de la preparación de un curso, es esencial elegir las tecnologías adecuadas. Para este proyecto, hemos seleccionado C# y Blazor para el desarrollo del sitio web, Cloudflare Pages para el alojamiento, Kotlin como lenguaje de programación y Typst para este apunte.

El sitio web puede ser encontrado en https://ravenhill.pages.dev.

1. Unidad 1: Introducción

1.1. Kotlin

Kotlin es un lenguaje de programación multiplataforma, desarrollado por JetBrains, que integra características de la programación orientada a objetos y funcional. Es conocido por su sintaxis concisa y capacidad para compilar no solo en JavaScript (JS) y WebAssembly (WASM) para ejecución en navegadores, sino también en Java Virtual Machine (JVM) para servidores y aplicaciones Android, así como en LLVM para aplicaciones de escritorio y sistemas embebidos.

En este curso, nos centraremos en la programación en Kotlin para la JVM, que es la plataforma más utilizada para este lenguaje. Sin embargo, los conceptos y técnicas que aprenderás son ampliamente aplicables a otras plataformas que soporta Kotlin y pueden ser útiles incluso en el aprendizaje de otros lenguajes de programación modernos.

1.1.1. A Taste of Kotlin

A continuación, te presentamos un ejemplo simple de Kotlin para darte una idea de cómo se ve y se siente el lenguaje.

```kotlin
data class Person(        // (1)
    val name: String,
    val age: Int? = null  // (2)
)

fun main() {
    val persons = listOf( // (3)
        Person("Harrier Du Bois"),
        Person("Kim Kitsuragi", age = 43) // (4)
    )
    val youngest = persons.minByOrNull { it.age ?: Int.MAX_VALUE }  // (5)
    println("The youngest is: $youngest") // (6)
}
// Output: The youngest is: Person(name=Kim Kitsuragi, age=43)
```

1. Declara una clase de datos `Person` con dos propiedades: `name` de tipo `String` y `age` de tipo `Int` opcional.
2. La propiedad `age` tiene un valor predeterminado de `null`.
3. Declara una lista inmutable de personas con dos elementos.
4. El segundo elemento de la lista tiene un valor de edad nombrado de 43.
5. Encuentra la persona más joven en la lista utilizando `minByOrNull` y el operador de elvis `?:`.
6. Interpola la variable `youngest` en una cadena y la imprime en la consola.

1.2. Lo básico

1.2.1. Expresiones vs. Declaraciones

En programación, es crucial distinguir entre **expresiones** y **declaraciones**, ya que cada una juega un papel diferente en la estructura y ejecución de los programas. A continuación, se explican estos conceptos en el contexto de Kotlin, aunque es importante notar que en otros lenguajes de programación, como Scala o Rust, los ciclos pueden ser expresiones o los condicionales pueden ser declaraciones.

Definición 1.2.1.1 (Expresiones): Las expresiones son fragmentos de código que producen un valor y pueden ser compuestas con otras expresiones. En Kotlin, las expresiones pueden ser tan simples como una constante o tan complejas como una función anónima. Ejemplos comunes de expresiones incluyen operaciones aritméticas, operadores lógicos y llamadas a funciones.

Definición 1.2.1.2 (Declaraciones): Las declaraciones son fragmentos de código que realizan una acción pero no retornan un valor. En Kotlin, las declaraciones no pueden ser compuestas con otras declaraciones, lo que significa que no pueden ser anidadas. Ejemplos comunes de declaraciones incluyen la asignación de variables, la ejecución de ciclos y la definición de funciones.

1.2.2. Declaración de funciones

Una función es un bloque de código designado para realizar una tarea específica. Está estructurada para ejecutar una secuencia de declaraciones y expresiones, y puede retornar un valor. La sintaxis básica para declarar una función en Kotlin se muestra a continuación:

```
fun functionName(parameter1: Type1, parameter2: Type2, ...): ReturnType {
    // Cuerpo de la función
    return result
}
```

Donde:

- `fun` – Palabra clave utilizada para declarar funciones.
- `functionName` – Nombre de la función que la identifica y permite su invocación.
- `parameter1: Type1, parameter2: Type2, ...` – Parámetros de la función con sus tipos correspondientes.
- `ReturnType` – Tipo de dato que la función retorna al finalizar su ejecución.

A continuación, se presenta un ejemplo de una función simple en Kotlin que suma dos números enteros y retorna el resultado:

```
fun sum(a: Int, b: Int): Int {
    return a + b
}
```

Como lo que hace la función es retornar un valor, diremos que es una función de una sola expresión. En Kotlin, las funciones de una sola expresión pueden simplificarse aún más, eliminando las llaves y la palabra clave `return`. El ejemplo anterior se puede reescribir de la siguiente manera:

```
fun sum(a: Int, b: Int): Int = a + b
```

Esta forma más concisa de definir funciones es útil para funciones simples que consisten en una sola expresión. Si la función es de una expresión, se puede omitir el tipo de retorno y dejar que el compilador lo infiera. Por ejemplo, la función anterior se puede reescribir de la siguiente manera:

```
fun sum(a: Int, b: Int) = a + b
```

En este caso, el compilador infiere que la función `sum` retorna un valor de tipo `Int` al sumar los dos parámetros de tipo `Int`. La inferencia de tipo es una característica que simplifica la sintaxis en funciones sencillas. Sin embargo, esta característica no se aplica en funciones que contienen más de una expresión, para evitar ambigüedades y confusiones.

1.2.2.1. Función `main`

El punto de entrada de un programa Kotlin es la función `main`, que es el punto de inicio de la ejecución. A continuación, se muestra un ejemplo de cómo la función `main` imprime un mensaje en la consola:

```
fun main(args: Array<String>): Unit {
    println("Hello, ${args[0]}!")
}
```

En este ejemplo, la función `main` recibe un argumento de tipo `Array<String>` y utiliza la función `println` para imprimir un mensaje en la consola. La interpolación de cadenas en Kotlin se realiza utilizando el signo de dólar `$` seguido por la variable o expresión entre llaves `{}`, permitiendo la inserción directa de valores dentro de las cadenas de texto. La función `println`, parte de la biblioteca estándar de Kotlin, imprime mensajes en la consola y automáticamente añade un salto de línea al final de cada mensaje. Las funciones estándar como `println` están disponibles globalmente sin necesidad de importaciones explícitas. Cuando una función como `main` no retorna un valor significativo, su tipo de retorno es `Unit`, indicando que no hay retorno relevante. Este tipo es similar al `void` en otros lenguajes de programación. En Kotlin, el tipo de retorno `Unit` se puede omitir en la declaración de la función, lo que simplifica la sintaxis, especialmente en funciones que no están destinadas a devolver ningún resultado. Por ejemplo, la declaración de la función `main` puede omitir `Unit` y quedar de la siguiente manera:

```
fun main(args: Array<String>) {
    println("Hello, ${args[0]}!")
}
```

Si quisieramos simplificar aún más la función `main`, podemos notar que la función `main` es de una sola expresión, por lo que podemos eliminar las llaves y la palabra clave `return`</code>:

```
fun main(args: Array<String>) = println("Hello, ${args[0]}!")
```

1.2.3. Declaración de variables

En Kotlin, las variables se declaran con las palabras clave `var` y `val`. La diferencia entre ambas es que `var` define una variable mutable, es decir, su valor puede cambiar en cualquier momento, mientras que `val` define una variable inmutable, cuyo valor no puede ser modificado una vez asignado.

La sintaxis básica para declarar una variable en Kotlin es la siguiente:

```
val/var variableName: Type = value
```

Donde:

- `val`/`var` – Palabra clave utilizada para declarar variables inmutables y mutables, respectivamente.
- `variableName` – Nombre de la variable que la identifica y permite su manipulación.
- `Type` – Tipo de dato de la variable.
- `value` – Valor inicial de la variable.

Por ejemplo:

```
val a: Int = 1 // asignación inmediata
var b = 2      // el tipo `Int` es inferido
b = 3          // Reasignar a `var` es OK
val c: Int     // Tipo requerido cuando no se provee un inicializador
```

```
c = 3           // asignación diferida
a = 4           // Error: Val no puede ser reasignado
```

En el ejemplo anterior, la variable `a` es inmutable, por lo que no se puede reasignar después de su inicialización. Por otro lado, la variable `b` es mutable, lo que permite cambiar su valor en cualquier momento. Finalmente, la variable `c` es inmutable y se inicializa posteriormente.

Noten que si bien `val` denota una variable inmutable, no significa que el objeto al que hace referencia sea inmutable. Por ejemplo, si la variable hace referencia a una lista mutable (representada por `MutableList`), los elementos de la lista pueden ser modificados, aunque la variable en sí no puede ser reasignada. Por otro lado, si la variable hace referencia a una lista inmutable (representada por `List`), los elementos de la lista no pueden ser modificados.

1.2.3.1. Declaración de constantes

Además de las variables, Kotlin también facilita la declaración de constantes utilizando la palabra clave `const`. Las constantes son variables inmutables cuyo valor se define en tiempo de compilación y permanece constante, sin cambios durante la ejecución del programa. Por ejemplo:

```
const val NAME = "Kotlin"
const val VERSION = "1.5.21"
```

La declaración de constantes solo es permitida en el ámbito global de un archivo o dentro de un objeto de nivel superior. No es posible declarar constantes locales dentro de funciones o bloques de código. Además, las constantes solo pueden ser de tipos primitivos como `Int`, `Double`, `Boolean`, o `String`, y no pueden ser inicializadas con funciones o expresiones que requieran cálculo en tiempo de ejecución.

En Kotlin, los tipos primitivos son un conjunto de tipos básicos que el sistema maneja de manera más eficiente debido a su representación directa en la máquina subyacente. A diferencia de muchos otros lenguajes de programación, Kotlin no tiene tipos primitivos tradicionales como en Java; en cambio, tiene clases envoltorio que corresponden a los tipos primitivos de Java, pero con una mejor integración en el sistema de tipos de Kotlin. Estos incluyen:

- `Int`: Representa un entero de 32 bits.
- `Double`: Representa un número de punto flotante de doble precisión.
- `Boolean`: Representa un valor verdadero o falso.
- `Long`: Representa un entero de 64 bits.
- `Short`: Representa un entero de 16 bits.
- `Byte`: Representa un byte de 8 bits.
- `Float`: Representa un número de punto flotante de precisión simple.
- `Char`: Representa un carácter Unicode.

Aunque internamente Kotlin maneja estos tipos como objetos para garantizar la compatibilidad con Java y permitir una programación más segura y versátil, en tiempo de ejecución, Kotlin optimiza el código usando representaciones primitivas donde es posible, similar a los tipos primitivos en Java. Esta optimización asegura que las operaciones que involucran tipos numéricos sean rápidas y eficientes.

Ejercicio: Cálculo del Área de un Círculo

1. **Definir la Constante PI**: Declara una constante `PI` y asígnale el valor `3.14159`.
2. **Programar la Función `circleArea`**: Implementa una función llamada `circleArea` que reciba un parámetro de tipo `Double` representando el radio del círculo y devuelva otro `Double` que sea el área del círculo calculada según la fórmula proporcionada.

   La fórmula para calcular el área de un círculo es la siguiente:

$$A(r) = \pi \times r^2$$

Donde:

- $A(r)$ es el área del círculo.
- π es la constante PI.
- r es el radio del círculo.

3. **Uso de la Función**: Una vez definida la función, puedes utilizarla para calcular el área de círculos con diferentes radios. No necesitas manejar radios negativos en esta implementación.

```
fun main() {
    println("El área de un círculo de radio 5.0 es ${circleArea(5.0)}")
}
```

1.2.4. Expresión `if`

En Kotlin, `if` puede ser utilizado no solo como una declaración de control, sino también como una expresión que devuelve un valor. Esto permite que `if` se incorpore directamente en el retorno de una función. A continuación, se muestran tres formas de utilizar `if` para definir una función que devuelve el mayor de dos números.

1.2.4.1. Forma Tradicional

En esta forma, `if` se utiliza en un estilo similar al de otros lenguajes de programación, donde se maneja como una declaración condicional dentro de una función:

```
fun maxOf(a: Int, b: Int): Int {
    if (a > b) {
        return a
    } else {
        return b
    }
}
```

1.2.4.2. Forma de Expresión con Bloques

Aquí, `if` es usado como una expresión directamente en la declaración de retorno. Esto reduce la redundancia y simplifica la función:

```
fun maxOf(a: Int, b: Int): Int {
    return if (a > b) {
        a
    } else {
        b
    }
}
```

1.2.4.3. Forma de Expresión Simplificada

Esta versión es la más concisa. `If` se utiliza aquí como una expresión inline dentro de la declaración de la función. Esta forma es particularmente útil para funciones simples que se pueden expresar en una sola línea, mejorando la claridad y concisión:

```
fun maxOf(a: Int, b: Int) = if (a > b) a else b
```

Cada una de estas formas tiene sus ventajas en diferentes contextos. La elección entre ellas depende de la complejidad de la función y de la preferencia por la claridad o la concisión en el código.

1.2.5. Expresión `when`

La expresión `when` en Kotlin es una forma más flexible y clara de manejar múltiples condiciones condicionales, comparada a las cadenas de `if-else`. Funciona de manera similar a `switch` en otros lenguajes de programación pero con capacidades superiores.

1.2.5.1. Ejemplo sin Usar `when`

Aquí utilizamos múltiples `if-else` para evaluar y retornar un valor basado en el tipo o valor de `obj`:

```
fun describe(obj: Any) =
    if (obj == 1) "One"
    else if (obj == "Hello") "Greeting"
    else if (obj is Long) "Long"
    else if (obj !is String) "Not a string"
    else "Unknown"
```

1.2.5.2. Utilizando `when` con Condiciones sin Argumento

Este enfoque es similar al anterior pero usando `when` sin un argumento específico, permitiendo que las condiciones sean expresiones booleanas arbitrarias:

```
fun describe(obj: Any): String = when {
    obj == 1 -> "One"
    obj == "Hello" -> "Greeting"
    obj is Long -> "Long"
    obj !is String -> "Not a string"
    else -> "Unknown"
}
```

1.2.5.3. Utilizando `when` con Argumento

Aquí, `when` se utiliza de manera más idiomática, con el objeto de la evaluación (`obj`) como argumento de `when`, simplificando aún más las condiciones:

```
fun describe(obj: Any): String = when (obj) {
    1 -> "One"
    "Hello" -> "Greeting"
    is Long -> "Long"
    !is String -> "Not a string"
    else -> "Unknown"
}
```

1.2.5.4. Ventajas de Usar `when`

- **Claridad**: `when` reduce la complejidad visual y mejora la legibilidad, especialmente con múltiples condiciones.
- **Flexibilidad**: `when` permite la evaluación de tipos, valores y condiciones complejas en una única estructura controlada.
- **Mantenibilidad**: Código escrito con `when` es generalmente más fácil de modificar que las largas cadenas de `if-else`.

Ejercicio: Reescribir usando una expresión `when`

Reescribe el siguiente código como una función de una expresión que utilice una expresión `when`:

```
fun login(username: String, passowrd: String): Boolean {
    if (loginAttempts >= MAX_LOGIN_ATTEMPTS) {
        return false
    }
    if (isValidPassword(password)) {
```

```
        loginAttempts = 0
        return true
    }
    loginAttempts++
    return false
}
```

1.2.6. Declaración `for`

La declaración `for` en Kotlin es una herramienta poderosa para iterar sobre colecciones y rangos. A continuación, se presenta cómo se puede utilizar para recorrer diferentes estructuras de datos y realizar operaciones sobre sus elementos.

1.2.6.1. Ejemplo Básico: Iteración sobre una Lista

El siguiente ejemplo muestra cómo usar el ciclo `for` para iterar sobre una lista de strings e imprimir cada elemento:

```
fun forExample() {
    val items = listOf("apple", "banana", "kiwi")
    for (item in items) {
        println(item)
    }
}
```

Explicación:

- `items` es una lista de strings.
- `for (item in items)` inicia un bucle que recorre cada elemento de la lista `items`.
- `println(item)` imprime cada elemento de la lista.

1.2.6.2. Uso Avanzado: Iteración sobre un Rango de Números

Kotlin permite iterar no solo sobre colecciones, sino también sobre rangos de números. Esto es especialmente útil para realizar operaciones repetitivas un número específico de veces o para generar secuencias numéricas.

```
fun rangeExample() {
    for (i in 1..5) {
        println(i)
    }
}
```

Explicación:

- `for (i in 1..5)` inicia un bucle que recorre los números del 1 al 5, inclusive.
- `println(i)` imprime cada número del 1 al 5.

1.2.6.3. Iteración con Índices

En ocasiones, puede ser útil tener acceso al índice de cada elemento durante la iteración. Kotlin facilita esto con la función `withIndex()`.

```
fun indexExample() {
    val items = listOf("apple", "banana", "kiwi")
    for ((index, value) in items.withIndex()) {
        println("Item at $index is $value")
    }
}
```

Explicación:

- `items.withIndex()` devuelve una colección de pares, cada uno compuesto por un índice y el valor correspondiente.
- `for ((index, value) in items.withIndex())` itera sobre estos pares.
- `println("Item at $index is $value")` imprime el índice y el valor de cada elemento en la lista.

1.2.7. Declaración `while`

La declaración `while` es fundamental para realizar bucles basados en una condición que necesita ser evaluada antes de cada iteración del ciclo. Es especialmente útil cuando el número de iteraciones no se conoce de antemano.

1.2.7.1. Ejemplo Básico: Conteo Regresivo

Aquí, `while` se utiliza para realizar un conteo regresivo desde 5 hasta 1:

```
fun whileExample() {
    var x = 5
    while (x > 0) {
        println(x)
        x--  // Decrementa x en 1 en cada iteración
    }
}
```

Explicación:

- `var x = 5` inicializa una variable x con el valor 5.
- `while (x > 0)` continúa el bucle mientras x sea mayor que 0.
- `println(x)` imprime el valor actual de x.
- `x--` reduce el valor de x en 1 después de cada iteración, asegurando que el bucle eventualmente terminará cuando x sea 0.

1.2.7.2. Ejemplo Avanzado: Búsqueda en Lista

`while` también puede ser útil para buscar un elemento en una lista hasta que se encuentre un elemento específico o se agote la lista:

```
fun searchExample() {
    val items = listOf("apple", "banana", "kiwi")
    var index = 0
    while (index < items.size && items[index] != "banana") {
        index++
    }
    if (index < items.size) {
        println("Found banana at index $index")
    } else {
        println("Banana not found")
    }
}
```

Explicación:

- `while (index < items.size && items[index] != "banana")` sigue iterando mientras el índice sea menor que el tamaño de la lista y el elemento actual no sea “banana”.
- `index++` incrementa el índice en cada iteración para revisar el siguiente elemento en la lista.
- La condición de salida del ciclo asegura que no se exceda el límite de la lista y se detenga la búsqueda una vez que se encuentre “banana”.

1.2.7.3. Comparación con `do-while`

Es útil comparar `while` con `do-while` para resaltar que `while` evalúa su condición antes de la primera iteración del bucle, mientras que `do-while` garantiza que el cuerpo del ciclo se ejecutará al menos una vez porque la condición se evalúa después de la ejecución del cuerpo.

```
fun doWhileExample() {
    var y = 5
    do {
        println(y)
        y--
    } while (y > 0)
}
```

Este ejemplo garantiza que el contenido dentro de `do` se ejecuta al menos una vez, independientemente de la condición inicial, lo cual es una distinción crucial en ciertos escenarios de programación.

Estas expansiones y discusiones proporcionarán a los estudiantes una comprensión más completa de cuándo y cómo usar `while` de manera efectiva en Kotlin.

1.2.8. Rangos

En Kotlin, los rangos permiten iterar de manera eficiente y elegante sobre secuencias numéricas. Recientemente, se ha introducido el estándar `..<` para crear rangos exclusivos, reemplazando el uso más antiguo de `until` en nuevos desarrollos.

1.2.8.1. Ejemplos de Rangos

1. **Rango Inclusivo (`..`)**: Este tipo de rango incluye ambos extremos, ideal para situaciones donde se necesita incluir el valor final en las operaciones.

   ```
   for (i in 1..5) print(i) // Imprime: 12345
   ```

   `1..5` crea un rango que incluye del 1 al 5.

2. **Rango Exclusivo (`..<`)**: `..<` se usa para generar rangos que excluyen el valor final, proporcionando una forma directa y legible de definir límites en iteraciones.

   ```
   for (i in 1..<5) print(i) // Imprime: 1234
   ```

   `1..<5` produce un rango desde 1 hasta 4, excluyendo el 5. Esta sintaxis es más intuitiva para quienes están familiarizados con lenguajes como Swift.

3. **Rango Decreciente con Paso (`downTo` con `step`)**: Kotlin también permite definir rangos decrecientes con un intervalo específico entre valores, lo que es útil para decrementos no estándar.

   ```
   for (i in 5 downTo 1 step 2) print(i) // Imprime: 531
   ```

   `5 downTo 1 step 2` crea un rango que empieza en 5 y termina en 1, incluyendo solo cada segundo número (decrementando de dos en dos).

Ejercicio: Suma de un Rango de Números

Desarrolla una función en llamada `sumRange(a: Int, b: Int): Int` que calcule y retorne la suma de todos los números entre dos enteros `a` y `b`, incluyendo ambos extremos.

Instrucciones

1. **Implementación de la Función**: La función debe usar una declaración `for` para iterar a través de un rango de números desde `a` hasta `b`.
2. **Manejo de Rangos**:

- Si `a` es menor o igual que `b`, el rango debe ir de `a` a `b`.
- Si `a` es mayor que `b`, el rango debe ir de `a` a `b` en orden inverso (es decir, decreciendo).

1.3. Null-Safety

1.3.1. El Problema de los Punteros Nulos

Los punteros o referencias nulas han sido denominados "el error de mil millones de dólares" por Tony Hoare, quien introdujo el concepto de valores nulos. Este problema ocurre cuando una variable que se espera que contenga una referencia a un objeto en realidad contiene un valor nulo, lo que lleva a errores en tiempo de ejecución cuando se intenta acceder a métodos o propiedades de dicho objeto.

1.3.2. Motivación

El manejo inadecuado de valores nulos puede causar una serie de errores en tiempo de ejecución que son difíciles de detectar y corregir, especialmente en grandes bases de código. Estos errores pueden comprometer la robustez y la seguridad de las aplicaciones. Por lo tanto, la gestión eficaz de los valores nulos es un aspecto crítico del diseño del software.

1.3.3. Enfoques en Diferentes Lenguajes

- **Java**: Tradicionalmente, Java ha permitido referencias nulas y lxs desarrolladores deben chequear explícitamente si una variable es nula antes de usarla. Esto puede llevar a mucho código de validación redundante y a veces se pasa por alto, resultando en `NullPointerException`.
- **C#**: Introduce el concepto de tipos anulables y no anulables. Los tipos de referencia son no anulables por defecto, pero pueden ser declarados como anulables con un signo de interrogación (`?`) después del tipo.
- **Scala**: Utiliza el concepto de `Option` para manejar valores nulos de manera segura. Un `Option` puede contener un valor (`Some`) o no contener nada (`None`), lo que obliga a los desarrolladores a manejar explícitamente ambos casos.

El problema con los approach de otros lenguajes como Scala es que todos los valores pueden ser nulos, lo que no garantiza la seguridad en tiempo de ejecución. Por ejemplo, una aplicación en Scala podría contener la siguiente línea de código:

```
var option: Option[String] = None
```

Esto impondrá la restricción de que para trabajar con `option`, se debe manejar explícitamente el caso en que no contenga un valor. Pero hay un malentendido común en torno a este enfoque: aunque `Option` es una forma segura de manejar valores nulos, no garantiza que los valores no sean nulos. En otras palabras, `Option` no elimina la posibilidad de `NullPointerException`, sino que la traslada a un nivel superior de abstracción.

```
var option: Option[String] = null
```

En este caso, `option` es nulo, lo que significa que el problema de los valores nulos no se ha resuelto, sino que se ha trasladado a la abstracción de `Option`. Por lo tanto, aunque `Option` es una mejora con respecto a los punteros nulos tradicionales, no es una solución completa al problema de los valores nulos. Para mejorar y clarificar la sección sobre cómo Kotlin maneja la nulabilidad, es crucial enfocar la redacción para que sea precisa, concisa y directamente informativa. Aquí te ofrezco una versión revisada de tu texto que corrige algunos errores tipográficos y mejora la claridad del mensaje:

1.3.4. Solución en Kotlin: Seguridad de Tipos frente a Nulos

Kotlin introduce un sistema de tipos que distingue claramente entre referencias que pueden ser nulas y aquellas que no pueden serlo. Esto evita los errores comunes de `NullPointerException` que son frecuentes en muchos otros lenguajes de programación.

1.3.4.1. Tipos no nullables y nullables

En Kotlin, las variables son no nulables por defecto. Esto significa que no puedes asignar `null` a una variable a menos que se declare explícitamente como nullable. Para declarar una variable nullable, añade `?` al tipo de la variable.

```
var a: String = "Definitivamente no nulo"
var b: String? = "Posiblemente nulo"
a = null // Error de compilación: tipo no nulable
b = null // Permitido: tipo nulable
```

En el ejemplo anterior:

- `a` es una variable de tipo `String`, que no puede ser nula. Intentar asignar `null` resulta en un error de compilación.
- `b` es una variable de tipo `String?`, que puede ser nula. Kotlin permite asignar `null` a este tipo sin problema.

1.3.4.2. Inferencia de tipos y nulabilidad

Kotlin también soporta la inferencia de tipos, lo que permite omitir la declaración explícita del tipo cuando este puede ser inferido del contexto. Sin embargo, la inferencia de tipos no cambia las reglas de nulabilidad:

```
var c = "Hola" // Tipo inferido como String, no nullable
c = null // Error de compilación: `c` es inferido como no nullable
```

En este caso:

- `c` es automáticamente inferido como `String` debido a la asignación inicial y, por defecto, no admite `null`.

1.3.4.3. Llamadas Seguras en Kotlin

En Kotlin, el manejo de variables que pueden ser nulas es fundamental debido al sistema de tipos diseñado para prevenir `NullPointerExceptions`. El compilador de Kotlin obliga a lxs desarrolladores a tratar las nulidades de manera explícita, asegurando que los accesos a variables nullables se manejen adecuadamente antes de su uso.

Comprobación de Nulos Obligatoria:

Para acceder a una propiedad o método de un objeto que podría ser nulo, Kotlin **no permite** simplemente hacerlo sin verificar si el objeto no es nulo. Si se intenta, el código no compilará, lo que obliga a manejar estas situaciones para garantizar la seguridad del programa:

```
val a: String? = TODO()
if (a != null) {
    println(a.length) // Acceso seguro porque se ha comprobado que 'a' no es nulo.
} else {
    println("a es nulo")
}
```

En este código:

- `a` es una variable que puede ser nula (`String?`).
- `TODO()` es una instrucción que significa "voy a implementar esto más tarde", es útil para ejemplos, pero no debe usarse en producción ya que siempre arrojará un error.

- Se utiliza una instrucción `if` para comprobar explícitamente que `a` no es nulo antes de acceder a su propiedad `length`.
- Este enfoque asegura que no se lanzará un `NullPointerException`.

Sintaxis de Llamadas Seguras

Para simplificar el manejo de nulos, Kotlin ofrece el operador de llamada segura (`?.`). Este operador permite realizar una operación sobre un objeto solo si no es nulo, de lo contrario devuelve `null`, evitando así el error en tiempo de ejecución:

```
println(a?.length) // Evalúa a 'null' y no hace nada si 'a' es nulo.
```

Este fragmento de código:
- Evalúa `a?.length`: si `a` no es nulo, devuelve la longitud; si `a` es nulo, devuelve `null`.
- `println` imprimirá el resultado, que será `null` si `a` es nulo.

Operador Elvis en Kotlin

El operador Elvis (`?:`) en Kotlin es una herramienta eficiente para manejar valores nulos en expresiones. Funciona evaluando la primera parte de la expresión (a la izquierda del operador) y, si el resultado no es nulo, lo retorna directamente. Si es nulo, evalúa y retorna la segunda parte de la expresión (a la derecha del operador).

El operador Elvis permite proporcionar un valor por defecto para expresiones que puedan resultar en nulo, reduciendo así la necesidad de bloques condicionales explícitos en el código. La expresión a la derecha del operador solo se evalúa si la expresión de la izquierda es nula, lo que lo hace eficiente en términos de rendimiento.

```
val a: String? = TODO()
val l = a?.length ?: -1
```

En este ejemplo:
- `a?.length` intenta obtener la longitud de `a`. Si que `a` es nulo, esta parte de la expresión también evaluaría a nulo.
- `?: -1` se activaría debido a que el resultado de la primera parte es nulo, y por lo tanto, el operador Elvis retorna `-1`.

Este patrón es especialmente útil cuando se necesita un valor por defecto para evitar valores nulos en el flujo de un programa.

Detalles Importantes:

- **Evaluación de Cortocircuito**: El operador Elvis realiza una evaluación de cortocircuito similar a los operadores lógicos en muchos lenguajes de programación. Esto significa que si la parte izquierda de la expresión es no nula, la parte derecha no se evalúa en absoluto.
- **Uso en Cadenas de Llamadas Seguras**: El operador Elvis es particularmente útil en combinación con llamadas seguras (`?.`), permitiendo manejar cadenas de métodos o propiedades que podrían ser nulas de una manera muy concisa y legible.

Ejercicio: Procesamiento de Temperaturas de Ciudades

Desarrolla un programa que maneje y procese información sobre temperaturas de diferentes ciudades, que pueden o no estar disponibles (null).

1. **Datos de Entrada**:

- Define un diccionario (`Map<String, Int?>`) donde cada clave es el nombre de una ciudad y cada valor asociado es la temperatura registrada en esa ciudad, que puede ser nula si no se ha registrado correctamente.

2. **Procesamiento de Datos**:
   - Itera sobre el diccionario usando un bucle. Durante la iteración, utiliza llamadas seguras para verificar si las temperaturas son nulas.

3. **Salida del Programa**:
   - Imprime el nombre de cada ciudad junto con su temperatura. Si la temperatura es nula, imprime el mensaje: `Temperatura no disponible.`

1.4. Manejo de Entrada de Usuario en Kotlin

La entrada de usuario es fundamental para los programas interactivos. Kotlin ofrece métodos convenientes y seguros para leer la entrada desde la consola, adaptándose a diversas necesidades de manejo de entrada.

1.4.1. Funciones de Lectura en Kotlin

- **`readlnOrNull()`**: Esta función es la recomendada para leer una línea de entrada. Devuelve un `String?`, que será `null` si no hay más datos disponibles (por ejemplo, si el usuario no introduce nada y presiona Enter). Es segura porque permite manejar los casos de nulos de forma explícita y evita excepciones innecesarias.

- **`readln()`**: Esta función ha sido deprecada en versiones recientes de Kotlin debido a que lanza una excepción si no hay más entrada disponible, lo cual puede conducir a interrupciones no manejadas en el flujo del programa. Se recomienda usar `readlnOrNull()` para evitar estos problemas y escribir un código más robusto y predecible.

1.4.2. Ejemplo de Uso de `readlnOrNull()`

A continuación, se muestra cómo utilizar `readlnOrNull()` para leer nombres de usuario de manera segura, terminando cuando el usuario no ingresa ningún dato:

```
fun main() {
    println("Introduce nombres. Presiona solo Enter para terminar.")
    while (true) {
        println("Ingresa un nombre:")
        val input = readlnOrNull()
        if (input.isNullOrEmpty()) {
            println("No se ingresó ningún nombre. Terminando el programa.")
            break  // Interrumpimos el ciclo
        } else {
            println("Nombre ingresado: $input")
        }
    }
}
```

Explicación del Código:
- **Loop Infinito**: Se usa un ciclo `while(true)` para pedir repetidamente al usuario que ingrese datos.
- **Lectura Segura**: `readlnOrNull()` se usa para leer la entrada. Si el usuario no introduce nada y presiona Enter, `input` será `null` o vacío, y el programa imprimirá un mensaje de salida y se terminará.
- **Manejo de Entrada Válida**: Si el usuario proporciona una entrada, el programa la muestra y continúa pidiendo más nombres.

1.5. Build systems

Definición 1.5.1 (Build system): Un sistema de compilación es una herramienta de software que automatiza el proceso de convertir código fuente en un ejecutable o una librería, realizando tareas como la compilación de código, gestión de dependencias y empaquetado de software.

Estos sistemas son esenciales para la gestión eficiente de proyectos de software, especialmente en entornos donde la escalabilidad y la reproducibilidad son importantes.

1.5.1. Usos Comunes

1. **Configuración Reutilizable**: Permite definir el proceso de compilación una sola vez y reutilizar esa configuración en múltiples proyectos, lo que ahorra tiempo y asegura consistencia a través de diferentes entornos de desarrollo y producción.
2. **Gestión de Dependencias**: Automatiza la resolución y actualización de librerías y otros paquetes necesarios para el desarrollo del proyecto. Esto incluye la descarga de las versiones correctas de cada dependencia y asegurar que todas sean compatibles entre sí.
3. **Automatización de Pruebas**: Integra y ejecuta automáticamente suites de pruebas para verificar que el software funcione correctamente antes de ser desplegado, mejorando la calidad y seguridad del software.
4. **Integración y Despliegue Continuos**: Facilita la implementación de prácticas de integración continua (CI) y despliegue continuo (CD), permitiendo que los cambios en el código sean automáticamente compilados, probados y desplegados.

1.5.2. Ejemplos de Build Systems

- **Make**: Uno de los primeros sistemas de compilación, ampliamente usado en proyectos de C y C++.
- **Maven y Gradle**: Muy utilizados en proyectos Java y Android por su poderosa gestión de dependencias y facilidades para la construcción de proyectos.
- **Apache Ant**: Utilizado en Java, se centra en ser flexible y extensible.
- **Webpack y Babel**: Populares en el desarrollo de aplicaciones web modernas, manejan la transformación y empaquetado de módulos JavaScript.

1.5.3. Gradle

Gradle es una herramienta poderosa y flexible utilizada para automatizar el proceso de construcción de software, incluyendo compilación, prueba, despliegue y empaquetado. Su diseño modular y su capacidad de personalización lo hacen adecuado para una amplia variedad de proyectos, desde aplicaciones móviles hasta grandes sistemas empresariales.

1.5.3.1. Características Principales

1. **DSL Basado en Groovy y Kotlin**: Gradle utiliza lenguajes de configuración específicos del dominio (DSL) basado en Groovy y Kotlin. Esto permite a los desarrolladores escribir scripts de construcción expresivos y mantenibles.
2. **Personalización y Extensibilidad**: Uno de los puntos fuertes de Gradle es su capacidad para ser personalizado y extendido. Los desarrolladores pueden crear tareas personalizadas y añadir funcionalidades específicas al proceso de construcción utilizando su API extensa.
3. **Compatibilidad con Entornos de Desarrollo**: Gradle es compatible con los principales entornos de desarrollo integrado (IDE) como Eclipse, IntelliJ IDEA y Android Studio, lo que permite una integración sin fisuras y un flujo de trabajo eficiente para los desarrolladores.

4. **Integración con Herramientas de CI/CD**: Se integra perfectamente con sistemas de integración continua y despliegue continuo como Jenkins, facilitando la automatización del ciclo de vida del desarrollo de software.
5. **Gestión de Dependencias Avanzada**: Ofrece un robusto sistema de gestión de dependencias que simplifica el manejo de librerías y módulos necesarios para el desarrollo de proyectos.

1.5.3.2. A Taste of Gradle

```
// build.gradle.kts
plugins {
  kotlin("jvm") version "1.9.23"
}

group = "cl.ravenhill"
version = "1.0-SNAPSHOT"

repositories {
  mavenCentral()
}

dependencies {
  testImplementation(kotlin("test"))
}

tasks.test {
  useJUnitPlatform()
}

kotlin {
  jvmToolchain(21)
}
```

1.5.3.3. Configuración de Repositorios en Gradle

En Gradle, los repositorios especifican las ubicaciones de las librerías necesarias para un proyecto. La configuración de los repositorios determina el orden en que Gradle buscará las dependencias requeridas, procediendo de arriba hacia abajo en el script.

Aquí se muestra cómo configurar varios tipos de repositorios en un archivo `build.gradle.kts`:

```
repositories {
    mavenCentral()  // (1)
    google()        // (2)
    maven {         // (3)
        url = uri("https://your.company.com/maven")
        credentials { // (4)
            username = System.getenv("MAVEN_USERNAME") ?: "defaultUser"
            password = System.getenv("MAVEN_PASSWORD") ?: "defaultPassword"
        }
    }
    flatDir {       // (5)
        dirs("lib")
    }
}
```

1. **Maven Central**: Este es el repositorio central de Sonatype, utilizado por defecto por proyectos Maven y Gradle en todo el mundo. Contiene una vasta cantidad de librerías para Java, Kotlin y Scala.

2. **Google**: Especialmente importante para proyectos Android, el repositorio de Google alberga librerías y herramientas específicas necesarias para el desarrollo en Android.
3. **Repositorio Maven Personalizado**: Utilizado para alojar artefactos privados o de terceros no disponibles en repositorios públicos. Es ideal para empresas que necesitan un control más estricto sobre las librerías utilizadas en sus proyectos.
4. **Seguridad de Credenciales**: Es vital no "hardcodear" credenciales directamente en los archivos de configuración. Utiliza variables de entorno para gestionar las credenciales de forma segura, como se muestra en el ejemplo. Esto ayuda a prevenir la exposición de información sensible.
5. **Flat Directory Repository**: Un repositorio de directorio plano se usa para incluir librerías que se encuentran directamente en el sistema de archivos local del proyecto, sin un repositorio Maven o Ivy. Es útil para librerías que no están disponibles en ningún repositorio remoto o para desarrollo y pruebas rápidas.

Consejos y Mejores Prácticas

- **Orden de Repositorios**: El orden en la que declaras los repositorios es importante, ya que Gradle buscará las dependencias en el orden en que aparecen. Si una dependencia está disponible en más de un repositorio, Gradle descargará la versión del primero que encuentre.
- **Uso de Repositorios Seguros**: Asegúrate de usar URLs seguras (`https`), especialmente cuando configures repositorios personalizados para garantizar que las transferencias de datos sean cifradas.

1.5.3.4. Dependencias

Definición 1.5.3.4.1 (Dependencia): Las dependencias son componentes externos o bibliotecas que un proyecto requiere para compilar y ejecutarse correctamente.

Gradle facilita la automatización de la descarga e integración de estas dependencias desde repositorios configurados, ya sean locales o remotos.

Gradle define varias configuraciones que determinan cómo y cuándo las dependencias están disponibles para tu proyecto durante su ciclo de vida:

- **`compileOnly`**: Las dependencias están disponibles solo durante la fase de compilación. No se incluyen en el tiempo de ejecución, útil para anotaciones, preprocesadores, etc.
- **`runtimeOnly`**: Las dependencias solo están disponibles en tiempo de ejecución. No están disponibles durante la compilación, adecuadas para implementaciones de interfaces que son proporcionadas en tiempo de ejecución.
- **`implementation`**: Las dependencias están disponibles tanto en tiempo de compilación como en tiempo de ejecución. No se exponen a los consumidores del proyecto, lo que ayuda a mantener el encapsulamiento.
- **`api`**: Similar a `implementation`, pero estas dependencias también se exponen a los consumidores, lo que significa que cualquier módulo que dependa de tu biblioteca tendrá acceso a ellas.
- **`testCompileOnly`**: Dependencias que solo son necesarias para compilar el código de prueba, no para ejecutarlo.
- **`testRuntimeOnly`**: Dependencias que son necesarias solo en el tiempo de ejecución de las pruebas.
- **`testImplementation`**: Dependencias que son necesarias tanto para compilar como para ejecutar las pruebas.
- **`testApi`**: Dependencias de la API utilizadas en el código de prueba, disponibles tanto para la compilación como para la ejecución de pruebas.

En el siguiente ejemplo de configuración de Gradle, definimos algunas dependencias que se utilizarán en el curso, incluyendo la librería de reflexión de Kotlin y varias librerías de Kotest para pruebas:

```
val kotestVersion = "5.8.0"

dependencies {
    implementation(kotlin("reflect"))
    testImplementation("io.kotest:kotest-property:$kotestVersion")
    testImplementation("io.kotest:kotest-runner-junit5:$kotestVersion")
    testImplementation("io.kotest:kotest-framework-datatest:$kotestVersion")
}
```

1.5.3.5. Tasks

En Gradle, las tareas son la unidad fundamental de trabajo. Son conjuntos de instrucciones ejecutables que realizan acciones específicas como compilar código, correr tests, construir un archivo JAR, publicar a un repositorio MAVEN, entre otras.

Gradle proporciona varias tareas predeterminadas que están configuradas para realizar acciones comunes de manera eficiente. Además, los usuarios pueden definir tareas personalizadas para adaptarse a necesidades específicas del proyecto.

Ejemplos de Tareas en Gradle

1. **Ejecutar Tests con JUnit**: Gradle facilita la configuración para utilizar frameworks de testing como JUnit. Por ejemplo, para configurar Gradle para usar el motor de JUnit Platform en la ejecución de tests:

   ```
   tasks.test {
       useJUnitPlatform()
   }
   ```

   Nota: Otros frameworks de testing como Kotest utilizan el motor de JUnit, por lo que esta configuración también se aplica a ellos.

2. **Copiar Archivos**: Puedes crear tareas para copiar archivos de un directorio a otro. Este ejemplo muestra cómo definir una tarea para copiar recursos:

   ```
   tasks.create<Copy>("copy") {
       description = "Copies resources to the output directory"
       group = "Custom"
       from("src")
       into("dst")
   }
   ```

3. **Calcular un Número de Fibonacci**: Este ejemplo ilustra cómo definir una tarea personalizada para calcular el 12º número de Fibonacci:

   ```
   tasks.register("Fib") {
       var first = 0
       var second = 1
       doFirst {
           println("Calculating the 12th Fibonacci number...")
           for (i in 1..11) {
               second += first
               first = second - first
           }
       }
       doLast {
   ```

```
            println("The 12th Fibonacci number is $second")
        }
    }
```

- `doFirst`: Se ejecuta antes de las demás acciones en la tarea.
- `doLast`: Se ejecuta después de todas las demás acciones en la tarea.

Para ejecutar tareas en Gradle, utiliza el Gradle Wrapper, que garantiza que todos los desarrolladores del proyecto usen la misma versión de Gradle, proporcionando consistencia a través del entorno de desarrollo:

- **En sistemas Unix**:

```
./gradlew test
./gradlew copy
./gradlew Fib
```

- **En sistemas Windows**:

```
.\gradlew.bat test
.\gradlew.bat copy
.\gradlew.bat Fib
```

Ejercicio: Crear una Tarea de Gradle para Calcular el Tamaño del Proyecto Compilado

Desarrolla una tarea de Gradle que determine y reporte el tamaño total de los archivos compilados de tu proyecto.

1. **Definición de la Tarea**: Crea una nueva tarea en tu archivo `build.gradle.kts` que calcule el tamaño total de los archivos en el directorio de clases compiladas de Kotlin.
2. **Acceso a los Archivos Compilados**: Utiliza el método `project.fileTree` para acceder a los archivos en el directorio de salida de compilación (`build/classes/kotlin/main`).
3. **Cálculo del Tamaño**: Itera sobre los archivos obtenidos y suma sus tamaños utilizando el método `length()` para obtener el tamaño total en bytes.
4. **Reportar el Tamaño**: Imprime el tamaño total calculado en la consola.

1.5.3.6. Plugins

Los plugins son componentes esenciales en Gradle que extienden sus capacidades al introducir nuevas tareas, configuraciones y funcionalidades a los scripts de build. Permiten modularizar y reutilizar configuraciones de construcción, evitando la duplicación de código en múltiples proyectos y facilitando la gestión de procesos de construcción complejos.

Beneficios de los Plugins

- **Extensión de Funcionalidades**: Los plugins pueden añadir tareas específicas para compilar código, ejecutar pruebas, generar documentación, entre otras.
- **Reutilización de Configuraciones**: Facilitan la estandarización de las configuraciones de construcción a través de diferentes proyectos, mejorando la coherencia y la eficiencia.
- **Automatización Mejorada**: Con los plugins, se puede automatizar desde la gestión de dependencias hasta la integración y despliegue continuos (CI/CD).

Ejemplos de Plugins en Gradle

En el curso, utilizaremos dos plugins, en particular habilitar el desarrollo en Kotlin y realizar análisis estático de código. Aquí te mostramos cómo puedes aplicar estos plugins en tu archivo `build.gradle.kts`:

```
plugins {
    kotlin("jvm") version "1.9.23" // Plugin de Kotlin para soporte de JVM
    id("io.gitlab.arturbosch.detekt") version "1.23.6" // Plugin para análisis estático
de código Kotlin
}
```

1.6. Análisis Estático

El análisis estático de código es un método crítico utilizado en el desarrollo de software para examinar el código fuente sin ejecutarlo. Esta técnica contrasta con el análisis dinámico, que analiza el programa en ejecución. El análisis estático se realiza generalmente con herramientas especializadas diseñadas para inspeccionar automáticamente el código en busca de errores y problemas de calidad.

1.6.1. Objetivos del Análisis Estático

1. **Detección Temprana de Errores**: Identificar y corregir errores de programación tempranamente en el ciclo de desarrollo, como uso incorrecto de tipos, referencias nulas y violaciones de sintaxis. Esto ayuda a prevenir fallos en etapas posteriores del desarrollo o después de la implementación del software.
2. **Cumplimiento de Estándares**: Asegurar que el código fuente cumpla con estándares de programación y mejores prácticas establecidas. Esto incluye convenciones de codificación, estructuras de datos adecuadas y uso eficiente de patrones de diseño, lo cual mejora la legibilidad y mantenibilidad del código.
3. **Seguridad**: Descubrir y mitigar vulnerabilidades de seguridad potenciales. Al analizar el código en busca de patrones conocidos de riesgos de seguridad, las herramientas de análisis estático ayudan a proteger la aplicación contra ataques y fallos de seguridad.

1.6.2. Detekt

Detekt es una herramienta de análisis estático robusta y configurable diseñada específicamente para el lenguaje de programación Kotlin. Es ampliamente utilizada para mejorar la calidad del código identificando problemas relacionados con la complejidad del código, estilo de codificación, posibles bugs y patrones de código no recomendados.

Características Principales

1. **Configuración Flexible**: Detekt ofrece opciones de configuración extensas que permiten personalizar el análisis según las necesidades específicas de cada proyecto. Esto incluye habilitar o deshabilitar ciertas reglas y modificar los umbrales de complejidad del código.
2. **Reglas Predefinidas y Personalizables**:
   - Viene con un amplio conjunto de reglas predefinidas que cubren varios aspectos del desarrollo de software, desde el estilo de codificación hasta la complejidad del código y los riesgos de seguridad.
   - Lxs desarrolladorxs pueden crear y añadir sus propias reglas personalizadas, lo que permite adaptar Detekt a las políticas de codificación específicas de un equipo o empresa.
3. **Identificación de Problemas de Calidad de Código**: Analiza el código para detectar antipatrones, uso ineficiente de la sintaxis de Kotlin, redundancias, complejidad innecesaria y otros problemas que pueden degradar la calidad del código.

Detekt se puede ejecutar como una tarea independiente desde la línea de comandos o integrarse en el ciclo de vida del build de Gradle. Esto facilita su incorporación en procesos de integración continua y revisión de código automatizada.

- En sistemas Unix:

  ```
  ./gradlew detekt
  ```

- En sistemas Windows:

  ```
  .\gradlew.bat detekt
  ```

1.7. Respuestas

Ejercicio: Cálculo del Área de un Círculo

```
const val PI = 3.14159

fun circleArea(radius: Double): Double = PI * radius * radius
```

Ejercicio: Reescribir usando una expresión `when`

```
fun login(username: String, password: String): Boolean = when {
    loginAttempts >= MAX_LOGIN_ATTEMPTS -> false
    isValidPassword(password) -> {
        loginAttempts = 0
        true
    }
    else -> {
        loginAttempts++
        false
    }
}
```

Ejercicio: Suma de un Rango de Números

```
fun sumRange(a: Int, b: Int): Int {
    var sum = 0
    if (a <= b) {
        for (i in a..b) {
            sum += i
        }
    } else {
        for (i in a downTo b) {
            sum += i
        }
    }
    return sum
}
```

Ejercicio: Procesamiento de Temperaturas de Ciudades

```
fun main() {
    val temperaturasPorCiudad = mapOf(
        "Madrid" to 22,
        "París" to null,
        "Berlín" to 18,
        "Roma" to null,
        "Londres" to 15
    )
    for ((ciudad, temperatura) in temperaturasPorCiudad) {
```

```
      val temperaturaFormateada = temperatura?.toString() ?: "Temperatura no disponible"
        println("La temperatura en $ciudad es: $temperaturaFormateada")
    }
}
```

Ejercicio: Crear una Tarea de Gradle para Calcular el Tamaño del Proyecto Compilado

```
tasks.register("countCompiledSize") {
    doLast {
        val files = fileTree("build/classes/kotlin/main")
        var size = 0L
        for (file in files) {
            size += file.length()
        }
        println("El tamaño del proyecto compilado es: $size bytes")
    }
}
```

2. Unidad 2: Programación orientada a objetos

La programación orientada a objetos es un paradigma de programación que utiliza "objetos" para diseñar aplicaciones y programas de computadora. A diferencia de la programación procedimental, que se centra en funciones y la lógica para manipular datos, OOP se organiza en torno a los datos, es decir, los objetos, y los métodos que operan sobre estos datos.

1. **¿Qué es un Objeto?**
   - Un objeto es una entidad dentro de un programa que tiene un estado y comportamiento asociados.
2. **Características de un Objeto**:
   - **Estado**: Representado por atributos o campos. Define las características del objeto.
   - **Comportamiento**: Representado por métodos. Define lo que el objeto puede hacer.
   - **Encapsulamiento**: Los detalles internos del objeto, como su estado, están ocultos del exterior. Esto se logra mediante el uso de modificadores de acceso que restringen el acceso directo a los componentes del objeto.

2.1. Objetos

En Kotlin, la palabra clave `object` se usa para declarar un objeto singleton. Este patrón es útil para casos donde se necesita una única instancia global de una clase para coordinar acciones a través del sistema.

```
object Dice {
    val sides = 6

    fun roll() = (1..sides).random()
}
```

2.2. Clases

Una clase en programación es un modelo o plantilla que define las características y comportamientos de los objetos. En Kotlin, una clase sirve para crear objetos específicos, definiendo sus propiedades (atributos) y sus capacidades (métodos).

Consideremos una clase `Player` para un juego. Esta clase define dos propiedades básicas para un jugador: su nombre y sus puntos de vida.

```
class Player {
    var name: String = ""
```

```
    var lifePoints: Int = 8000
}
```

Para utilizar esta clase en Kotlin, crearemos una instancia de `Player` y asignaremos valores a sus propiedades. Luego, accederemos a estas propiedades para realizar operaciones, como imprimir el estado del jugador.

```
fun main() {
    val player = Player()  // Creación de un objeto de tipo Player
    player.name = "Seto Jaiva"  // Asignación de nombre al jugador
    player.lifePoints = 7000  // Asignación de puntos de vida

    // Impresión de los datos del jugador
    println("El jugador ${player.name} tiene ${player.lifePoints} puntos de vida")
}
```

2.3. Null-safety: Inicialización tardía

A veces no es posible inicializar una variable en el momento de su declaración sin comprometer la limpieza del código o la eficiencia. Para estos casos, Kotlin ofrece dos herramientas útiles: `lateinit` y `Delegates.notNull()`.

2.3.1. Uso de `lateinit`

- **Propósito**: La palabra clave `lateinit` permite declarar variables no nulas que serán inicializadas posteriormente. Es especialmente útil cuando una variable depende de inyección de dependencias o configuración inicialización que no está disponible en el momento de la creación del objeto.
- **Aplicabilidad**: Solo se puede usar con variables de tipos que son inherentemente no nullables y que no son primitivos (como `Int`, `Float`, etc.).
- **Verificación**: Puedes verificar si una variable `lateinit` ha sido inicializada mediante `::variable.isInitialized` antes de acceder a ella para evitar errores en tiempo de ejecución.

```
class Player {
    lateinit var name: String  // Declaración de una variable que se inicializará más
tarde

    fun initializePlayer(name: String) {
        this.name = name
    }
}
```

2.3.2. Uso de `Delegates.notNull()`

- **Propósito**: `Delegates.notNull()` es una forma de crear una propiedad que debe ser inicializada eventualmente, y se utiliza cuando las variables son de tipo primitivo o cuando `lateinit` no es aplicable.
- **Comportamiento**: Acceder a una propiedad delegada por `notNull()` antes de su inicialización lanzará una `IllegalStateException`, indicando que la propiedad no ha sido inicializada.

```
class Player {
    var lifePoints: Int by Delegates.notNull()  // Delegación para asegurar la
inicialización antes de uso
}
```

2.4. Encapsulamiento

Técnica de ocultamiento de los datos de un objeto de manera que solo se pueda cambiar mediante las funciones definidas para ese objeto Restringe el acceso directo a algunos de los componentes del objeto

Modificadores de visibilidad

Modificador	Visibilidad
`public`	Accesible desde cualquier parte del código
`private`	Accesible solo desde la clase que lo contiene
`protected`	Accesible desde la clase que lo contiene y sus subclases
`internal`	Accesible solo desde el módulo que lo contiene

2.5. Herencia

La herencia es un principio fundamental de la programación orientada a objetos que permite a una clase derivar o heredar propiedades y comportamientos (métodos) de otra clase. Esto establece una relación jerárquica entre la clase superior (superclase) y la clase derivada (subclase).

2.5.1. Propósito y Beneficios de la Herencia

- **Especialización**: La herencia permite crear nuevas clases a partir de clases existentes, proporcionando un método eficaz para reutilizar y extender el código existente.
- **Reutilización de Código**: Aunque es un efecto secundario, la herencia facilita la reutilización de código, permitiendo que las subclases utilicen métodos y propiedades de la superclase sin necesidad de reescribirlos.
- **Jerarquía de Objetos**: Organiza y estructura el código en una jerarquía natural que refleja relaciones reales entre entidades, facilitando el mantenimiento y la comprensión del código.

La herencia debe usarse con un propósito claro y con coherencia lógica:

- **Relación "Es un(a)"**: La herencia debe reflejar una relación lógica y natural de "es un(a)" entre la superclase y la subclase. Por ejemplo, un `Perro` es un tipo de `Mamífero`, por lo tanto, es lógico que `Perro` herede de `Mamífero`.
- **Evitar la Herencia Improcedente**: No se debe utilizar la herencia simplemente para reutilizar código si no existe una relación lógica clara. Por ejemplo, decir que un `Perro` es un tipo de `Lobo` solo para reutilizar métodos como `aullar()` es incorrecto, pues aunque ambos pueden compartir comportamientos, son especies distintas con características propias.

2.6. Constructores

Un constructor en Kotlin es un bloque especial de código que se ejecuta al momento de crear una instancia de una clase. Su principal función es inicializar el objeto recién creado con los datos necesarios o realizar cualquier configuración inicial necesaria.

Kotlin ofrece dos tipos de constructores: primarios y secundarios.

2.6.1. Constructor Primario

El constructor primario está integrado en la declaración de la clase. Es el medio más directo y conciso para inicializar objetos.

```
class Player(val name: String, var lifePoints: Int)
```

En este ejemplo, `Player` es una clase con un constructor primario que recibe dos parámetros: `name` y `lifePoints`. Nota que no es necesario especificar la visibilidad `public` ya que es la visibilidad por defecto en Kotlin y su declaración es redundante.

Uso de Bloques `init`:

Los bloques `init` se utilizan para realizar validaciones o inicializaciones adicionales que no pueden ser expresadas directamente en los parámetros del constructor.

```
class Player(val name: String, var lifePoints: Int) {
    init {
        require(lifePoints >= 0) { "Los puntos de vida no pueden ser negativos" }
    }
}
```

Aquí, el bloque `init` asegura que los puntos de vida no sean negativos al momento de crear un objeto de tipo `Player`.

Bibliography

[1] "What is hashing and how does it work?." Accessed: Sep. 15, 2022. [Online]. Available: https://www.techtarget.com/searchdatamanagement/definition/hashing

[2] Dmitry Jemerov and Svetlana Isakova, *Kotlin in action*. Shelter Island, NY: Manning Publications Co, 2017.

[3] Michael Pilquist, R. Bjarnason, P. Chiusano, and P. Chiusano, *Functional programming in Scala*, Second edition. Shelter Island: Manning Publications, 2023.

[4] Josh Skeen and David Greenhalgh, *Kotlin programming: the Big Nerd Ranch guide*, First edition. Atlanta, GA: Big Nerd Ranch, 2018.

[5] "Gradle Build Tool." Accessed: Apr. 20, 2024. [Online]. Available: https://gradle.org/

[6] "Hello from detekt | detekt." Accessed: Apr. 20, 2024. [Online]. Available: https://detekt.dev/

[7] Joshua Bloch, *Effective Java*, Third edition. Boston: Addison-Wesley, 2018.

[8] K. Beck, *Test-driven development: by example*. in The Addison-Wesley signature series. Boston: Addison-Wesley, 2003.

[9] "Kotest | Kotest." Accessed: Apr. 25, 2024. [Online]. Available: https://kotest.io/

[10] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black, "Traits: Composable Units of Behaviour," in *ECOOP 2003 – Object-Oriented Programming*, L. Cardelli, Ed., Berlin, Heidelberg: Springer, 2003, pp. 248–274. doi: 10.1007/978-3-540-45070-2_12.