Diseño e implementación de bibliotecas de software

Ignacio Slater Muñoz
reachme@ravenhill.cl

Contents

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. Introducción al desarrollo de bibliotecas de software

1.1.1. APIs

Una API (Application Programming Interface) define bloques de construcción reutilizables que permiten incorporar funcionalidades a una aplicación de manera eficiente y estandarizada. Las APIs son fundamentales para el desarrollo de software moderno y se proporcionan generalmente mediante bibliotecas.

1.1.1.1. Características de una buena API

1.1.1.1.1. Modela el Problema

Una API se escribe para solucionar un problema específico y debe proporcionar una buena abstracción de dicho problema. A continuación, se detallan los principios clave para modelar el problema de manera efectiva:

- **Propósito Claro**: Cada función, clase y variable debe tener un propósito central y claramente definido. Este propósito debe reflejarse en su nombre para que sea intuitivo y fácil de entender por otrxs desarrolladorxs.
- **Consistencia**: Mantén una nomenclatura y estructura consistentes en toda la API. Esto facilita su comprensión y uso, y reduce la posibilidad de errores.

1.1.1.1.2. Esconde Detalles

La razón principal para escribir una API es ocultar los detalles de la implementación, permitiendo que estos puedan ser modificados sin afectar a lxs clientes que utilizan la API. A continuación, se presentan los principios clave para ocultar los detalles de implementación de manera efectiva:

- **Encapsulación**: Utiliza la encapsulación para restringir el acceso a los detalles internos de la implementación. Mantén las variables y métodos internos privados y expón solo lo necesario a través de métodos públicos.
- **Interfaz Clara**: Proporciona una interfaz clara y bien definida que permita a lxs usuarixs interactuar con la API sin conocer los detalles internos. Esto facilita la comprensión y uso de la API.
- **Separación de Preocupaciones**: Divide la API en módulos o componentes, cada uno con responsabilidades claramente definidas. Esto facilita el mantenimiento y evolución de la API, ya que los cambios en un componente no afectan a otros.

1.1.1.1.3. Mínimamente completa

Una API debe ser tan pequeña como sea posible para mantener su simplicidad y facilidad de mantenimiento. A continuación, se presentan los principios clave para lograr una API mínimamente completa:

- **Simplicidad**: Diseña la API con la menor cantidad de elementos públicos necesarios. Cada elemento público aumenta la superficie de mantenimiento y promesas que debes cumplir a largo plazo.

- **No te repitas (DRY)**: Evita la duplicación de funcionalidades. Cada funcionalidad debe implementarse una sola vez y ser reutilizada en lugar de replicarse en diferentes partes de la API.
- **Enfoque en la Esencia**: Identifica y proporciona solo las funcionalidades esenciales que resuelven el problema central de la API.
- **Abstracciones Claras**: Mantén las abstracciones claras y concisas. No añadas métodos o clases innecesarios que no aporten valor significativo a la API.
- **Principio de Responsabilidad Única**: Asegúrate de que cada componente de la API tenga una única responsabilidad claramente definida. Esto ayuda a mantener la API enfocada y reduce la complejidad.

> Every public element in your API is a promise—a promise that you'll support that functionality for the lifetime of the API.
>
> — Reddy, 2011

1.1.1.1.4. Fácil de Usar

Una API debe ser intuitiva y difícil de usar incorrectamente. Aquí hay algunos principios clave para lograr una API fácil de usar:

> It should be possible for a client to look at the method signatures of your API and be able to glean how to use it without any additional documentation.
>
> — Reddy, 2011

- **Intuitiva**: Los usuarios deben poder entender cómo usar la API simplemente observando las firmas de los métodos, sin necesidad de documentación adicional.
- **Difícil de Usar Mal**: La API debe diseñarse de manera que sea difícil cometer errores al utilizarla. Esto puede lograrse proporcionando valores predeterminados razonables, validando los parámetros y utilizando tipos de datos adecuados.
- **Consistente**: Mantén la consistencia en los nombres, el orden de los parámetros y los patrones utilizados en la API. Esto facilita el aprendizaje y la utilización de la API por parte de los usuarios.
- **Evita Abreviaciones**: Usa nombres completos y claros en lugar de abreviaciones. Las abreviaciones pueden ser confusas y menos descriptivas.
- **Ortogonalidad**: Asegúrate de que los cambios en una parte de la API no afecten otras partes. Las variables públicas y los métodos deben ser independientes entre sí en la medida de lo posible. Los cambios en una variable pública no deberían afectar a otras variables públicas.

1.1.1.1.5. Alta Cohesión y Bajo Acoplamiento

Para diseñar una API efectiva, es crucial lograr una alta cohesión y un bajo acoplamiento entre sus componentes:

- **Acoplamiento**: Mide el grado de dependencia entre componentes.
  - **Bajo Acoplamiento**: Es deseable porque implica que los componentes pueden modificarse independientemente sin afectar a otros componentes. Esto facilita la mantenibilidad y la flexibilidad del código.
- **Cohesión**: Mide el grado en que las funciones de un componente están relacionadas entre sí.
  - **Alta Cohesión**: Es deseable porque significa que el componente realiza una única tarea o un conjunto de tareas relacionadas de manera eficiente y clara. Los componentes cohesivos son más fáciles de entender, probar y mantener.

Implementar alta cohesión y bajo acoplamiento en el diseño de una API ayuda a crear un sistema más modular, donde los cambios en una parte del sistema tienen un impacto mínimo en otras partes. Esto resulta en un código más robusto, flexible y fácil de mantener.

1.1.1.2. Estable, Documentada y Testeada

Para garantizar la calidad y usabilidad de una API, es fundamental que cumpla con los siguientes criterios:

- **Estabilidad**:
  - La interfaz de la API debe ser versionada.
  - Los cambios entre versiones no deben ser incompatibles, asegurando así que los clientes de la API puedan actualizar sin problemas.
- **Documentación Completa**:
  - La API debe estar bien documentada.
  - La documentación debe proporcionar información clara y detallada sobre las capacidades de la API, su comportamiento esperado, mejores prácticas y posibles condiciones de error.
  - La documentación ayuda a los desarrolladores a entender y usar la API de manera efectiva y correcta.
- **Pruebas Automatizadas**:
  - La implementación de la API debe estar respaldada por un conjunto exhaustivo de pruebas automatizadas.
  - Estas pruebas aseguran que los cambios nuevos no rompan casos de uso existentes, proporcionando confianza en la estabilidad y confiabilidad de la API.

1.1.2. Bibliotecas de Software

Claro, aquí tienes una versión mejor

- **Definición**:
  - También conocidas como librerías.
  - Son colecciones de código precompilado que proporcionan funciones, clases y procedimientos reutilizables.
- **Funcionalidad**:
  - Facilitan la realización de tareas específicas o comunes en el desarrollo de software.
  - Permiten a lxs desarrolladorxs reutilizar código existente, ahorrando tiempo y esfuerzo.
- **API**:
  - Las bibliotecas exponen una API que define cómo los clientes pueden interactuar con ellas.
  - La API proporciona los bloques de construcción necesarios para integrar la funcionalidad de la biblioteca en aplicaciones más grandes.

Las bibliotecas de software son esenciales para la eficiencia y efectividad en el desarrollo de aplicaciones, proporcionando soluciones probadas y optimizadas para problemas comunes.

1.1.2.1. Bibliotecas vs Aplicaciones

Bibliotecas:

- **Funcionalidad**:
  - Proveen funcionalidades específicas y reutilizables.
  - No son ejecutables por sí mismas; requieren ser incluidas en una aplicación.
  - Ofrecen una API para que lxs desarrolladorxs interactúen con sus funcionalidades.

Aplicaciones:

- **Propósito**:
  - Resuelven un problema o realizan tareas específicas de principio a fin.
  - Son ejecutables autónomos; pueden funcionar independientemente.
  - Generalmente proporcionan una interfaz de usuario (UI) o una interfaz de línea de comandos (CLI) para que lxs usuarios finales interactúen.

Las bibliotecas y las aplicaciones juegan roles complementarios en el desarrollo de software, con las bibliotecas proporcionando herramientas y funcionalidades reutilizables que las aplicaciones integran y utilizan para cumplir con tareas completas y específicas.

1.1.2.2. Principios de Diseño de Bibliotecas

- **Interfaces Simples y Fáciles de Entender**:
  - Mantén las interfaces de la biblioteca lo más simples posible.
  - Facilita a lxs usuarios la comprensión y el uso de las funcionalidades proporcionadas.
- **Coherencia en Patrones de Diseño**:
  - Sigue patrones de diseño coherentes en toda la biblioteca.
  - Asegura que lxs usuarios puedan predecir el comportamiento y uso de diferentes componentes.
- **Componentes Independientes y Reutilizables**:
  - Diseña componentes que sean independientes entre sí.
  - Facilita la reutilización de componentes en diferentes contextos sin dependencias innecesarias.
- **Extensibilidad**:
  - Permite a los usuarios ampliar la funcionalidad de la biblioteca sin necesidad de modificar el código fuente.
  - Utiliza patrones de diseño como herencia, composición y polimorfismo para ofrecer extensibilidad.
- **Buena Documentación y Ejemplos de Uso**:
  - Proporciona documentación clara y completa.
  - Incluye ejemplos de uso prácticos que muestren cómo utilizar las diferentes funcionalidades de la biblioteca.
  - Asegura que la documentación esté actualizada con cada nueva versión de la biblioteca.

1.2. 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.

1.2.1. 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.

```
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.

1.3. Lo básico

1.3.1. 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 1.3.1.1 (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 1.3.1.2 (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.

1.3.2. 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.

1.3.2.1. 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]}!")
```

1.3.3. 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.

1.3.3.1. 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:

```kotlin
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.

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

1.3.4. 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.

1.3.4.1. 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
    }
}
```

1.3.4.2. 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
    }
}
```

1.3.4.3. 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.

1.3.5. 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.

1.3.5.1. 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"
```

1.3.5.2. 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"
}
```

1.3.5.3. 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"
}
```

1.3.5.4. 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
}
```

1.3.6. 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.

1.3.6.1. 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.

1.3.6.2. 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.

1.3.6.3. 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.

1.3.7. 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.

1.3.7.1. 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.

1.3.7.2. 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".

1.3.7.3. 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.

1.3.8. 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.

1.3.8.1. 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).

1.4. Null-Safety

1.4.1. 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.

1.4.2. 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.

1.4.3. 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:

1.4.4. 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.

1.4.4.1. 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.

1.4.4.2. 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`.

1.4.4.3. 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.`

1.5. 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.

1.5.1. 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.

1.5.2. 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.

1.6. Código limpio y mantenible

- Código fácil de leer y entender:
  - Escribe código que sea claro y comprensible para otros desarrolladores.
  - Usa nombres de variables y funciones descriptivos y significativos.
- Facilita el trabajo en equipo:
  - Un código limpio y bien estructurado mejora la colaboración entre los miembros del equipo.
  - Facilita la revisión de código y la incorporación de nuevos desarrolladores al proyecto.
- Evolución del software a lo largo del tiempo:
  - Un código mantenible permite realizar cambios y agregar nuevas funcionalidades de manera eficiente.
  - Reduce la deuda técnica y el riesgo de introducir errores en el sistema.

1.6.1. Impacto en la mantenibilidad

- Otrxs desarrolladorxs pueden entender rápidamente el propósito y funcionamiento del código:
  - Facilita la colaboración y el trabajo en equipo.
  - Permite a nuevxs desarrolladorxs integrarse más rápidamente al proyecto.
- Código más legible reduce la probabilidad de introducir errores:
  - Ayuda a identificar problemas potenciales antes de que se conviertan en errores.
  - Mejora la calidad general del software.
- Código bien estructurado y documentado es más fácil de modificar y mejorar:
  - Facilita la implementación de nuevas funcionalidades.
  - Permite realizar refactorizaciones y mejoras con menor riesgo.
  - Asegura que el conocimiento se mantenga accesible para todo el equipo.

1.6.2. Impacto en la escalabilidad

- Código limpio puede adaptarse más fácilmente a nuevos requisitos y cambios:
  - Permite una respuesta ágil a las necesidades del negocio.

  - Facilita la implementación de nuevas características sin comprometer la calidad del código existente.
- Componentes bien definidos y modulares pueden ser reutilizados en diferentes partes del proyecto o en otros proyectos:
  - Fomenta la reutilización del código.
  - Reduce el tiempo de desarrollo al evitar la duplicación de esfuerzos.
- Equipos pueden trabajar de manera más eficiente y productiva, acelerando el desarrollo de nuevas funcionalidades y mejoras:
  - Mejora la colaboración y la distribución de tareas.
  - Minimiza los conflictos y la necesidad de refactorizaciones extensivas.

1.6.3. Principios de código limpio

1.6.3.1. Nombres descriptivos

Utilizar nombres que revelen la intención.

Si un nombre requiere un comentario, entonces el nombre no revela la intención adecuadamente.

Ejemplo con nombres poco descriptivos:

```
fun getThem(theList: List<IntArray>): List<IntArray> {
    val list1 = mutableListOf<IntArray>()
    for (x in theList) {
        if (x[0] == 4) {
            list1.add(x)
        }
    }
    return list1
}
```

Ejemplo mejorado con nombres descriptivos:

```
fun getFlaggedCells(gameBoard: List<IntArray>): List<IntArray> {
    val flaggedCells = mutableListOf<IntArray>()
    for (cell in gameBoard) {
        if (cell[STATUS_VALUE] == FLAGGED) {
            flaggedCells.add(cell)
        }
    }
    return flaggedCells
}
```

En este último ejemplo:
- `gameBoard` es un nombre que sugiere claramente el contexto.
- `flaggedCells` indica explícitamente lo que se está almacenando en la lista.
- `cell` describe cada elemento de la lista original.

Aquí tienes una versión mejorada y más clara, con la corrección indicada:

Evitar desinformación.

Evitar usar nombres que varíen ligeramente:

Incorrecto: `XYZControllerForEfficientHandlingOfStrings` vs `XYZControllerForEfficientStorageOfStrings`

Usar nombres de series como `a1`, `a2`, ..., `aN` no es informativo.

Ejemplo con nombres poco informativos:

```
fun copyChars(a1: CharArray, a2: CharArray) {
    for (i in a1.indices) {
        a2[i] = a1[i]
    }
}
```

Ejemplo mejorado con nombres descriptivos:

```
fun copyChars(source: CharArray, destination: CharArray) {
    for (i in source.indices) {
        destination[i] = source[i]
    }
}
```

Las palabras "ruido" son redundantes y deben evitarse:

- **Incorrecto**: `ProductInfo` vs `ProductData`
- **Incorrecto**: `name` vs `theName` vs `aName`
- **Incorrecto**: `nameString`

Ejemplo de nombres claros y concisos:

```
val currentAccount
val activeUserAccounts
val accountDetails
```

Utilizar nombres pronunciables:

> If you can't pronounce it, you can't discuss it without sounding like an idiot. (...) This matters because programming is a social activity.
>
> — Martin, 2009

Ejemplo con nombre poco pronunciable:

```
// "generation date, year, month, day, hour, minute, and second"
val genymdhms: String = TODO()
```

1.6.3.2. Funciones

- Las funciones deben ser pequeñas.

> The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that.
>
> — Martin, 2009

- Idealmente, una función no debería tener más de 20 líneas.
- Cada función debe hacer una sola cosa.
  - Una función que hace una sola cosa "no puede" dividirse en secciones.
- El número ideal de argumentos de una función es 0, seguido por 1 y luego 2.
  - Tener 3 o más argumentos debe estar justificado fuertemente.
  - Evitar el uso de argumentos "flag".
- Las funciones deben hacer algo o responder una pregunta, pero no ambas.

1.6.3.3. Comentarios

> The proper use of comments is to compensate for our failure to express ourselves in code.
>
> — Martin, 2009

- Los comentarios son propensos a errores.
- Es importante mantener los comentarios actualizados.
- Los comentarios deben usarse cuando el código por sí solo no sea suficiente para expresar la intención.

1.6.3.4. Formato de código

- **Archivos pequeños**: Son más fáciles de entender y manejar que archivos gigantes.
- **Orden lógico**: El código generalmente se lee de arriba hacia abajo, por lo que las funciones deben estar organizadas de manera lógica para facilitar la lectura.
- **Estándares de formato**: Es crucial definir y seguir estándares de formato para cada proyecto. Estos estándares aseguran consistencia y legibilidad.
- **Herramientas de análisis estático**: Utilizar herramientas de análisis estático puede ayudar a mantener el formato del código consistente y detectar desviaciones de los estándares establecidos.

Mantener un formato de código coherente y organizado no solo mejora la legibilidad, sino que también facilita la colaboración y el mantenimiento del proyecto a largo plazo.

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")
    }
}
```

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()
}
```

Ejercicio: Implementación de un Gestor de Eventos

Desarrolla un objeto llamado `EventManager`, que se encargue de administrar una lista de eventos. Cada evento será representado como una cadena de texto (`String`).

Requisitos:

1. **Estructura de Datos:** Utiliza un `MutableList<String>` para almacenar los eventos. La lista puede ser inicializada utilizando `mutableListOf()`.
2. **Métodos Requeridos:**
   - **Agregar Evento:** Define un método `addEvent(String)` que permita añadir un nuevo evento a la lista.
   - **Obtener Eventos:** Define un método `getEvents(): List<String>` que retorne una copia inmutable de la lista de eventos. Puedes utilizar el método `toList()` para crear una copia de la lista mutable.

Ejemplo de uso:

```
fun main() {
    EventManager.addEvent("Concierto de Rock")
    EventManager.addEvent("Festival de Cine")

    println(EventManager.getEvents())
    // Debería imprimir: ["Concierto de Rock", "Festival de Cine"]
}
```

Nota: Por ahora no te preocupes de la visibilidad de los métodos y variables.

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
}
```

Ejercicio: Implementación de un Perfil de Usuario

Desarrolla una clase `UserProfile` que administre la información de un usuario, utilizando inicialización tardía para ciertos campos.

Descripción de la Clase:

- **Campos con Inicialización Tardía:** La clase `UserProfile` debe incluir dos campos que se inicializarán tardíamente:
  - `username`: Una cadena de texto (`String`) que representa el nombre de usuario.
  - `age`: Un entero (`Int`) que representa la edad del usuario.
- **Requisitos Funcionales:**
  - **Función de Visualización de Información:** Define una función `displayUserInfo()` dentro de la clase que:
    - Imprima los datos del usuario (nombre de usuario y edad) si ambos campos han sido inicializados.
    - Muestre un mensaje de error si el nombre de usuario no se ha inicializado

---

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 en Kotlin

Un constructor en Kotlin es un bloque especial de código que se ejecuta durante la creación de una instancia de una clase. Su función principal es inicializar el objeto recién creado con los datos necesarios o realizar configuraciones iniciales.

Kotlin ofrece dos tipos de constructores: **primarios** y **secundarios**.

2.6.1. Constructor Primario

El constructor primario se integra directamente en la declaración de la clase, ofreciendo una forma directa y concisa de inicializar objetos.

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

En este ejemplo, `Player` es una clase con un constructor primario que acepta dos parámetros: `name` y `lifePoints`. Es importante notar que la visibilidad `public` es la predeterminada en Kotlin y declararla explícitamente es redundante.

2.6.1.1. Bloques `init`

Los bloques `init` son útiles para realizar validaciones o inicializaciones adicionales que no se pueden expresar 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` garantiza que los puntos de vida no sean negativos en el momento de crear un objeto de tipo `Player`.

2.6.2. Constructores Secundarios

Una clase puede tener múltiples constructores, lo cual se realiza utilizando la palabra clave `constructor`.

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

    constructor() : this("Unknown", 8000) {
        println("New player $name has $lifePoints life points.")
    }
}
```

Un constructor secundario siempre comienza llamando a otro constructor de la clase (usualmente el primario), como se indica con `: this(...)`.

2.6.2.1. Uso de Parámetros por Defecto

Frecuentemente, los constructores secundarios pueden ser reemplazados por parámetros por defecto en el constructor primario para simplificar el código.

```
class Player(
    val name: String = "Unknown",
    var lifePoints: Int = 8000
) {
    init {
        require(lifePoints >= 0) { "Los puntos de vida no pueden ser negativos." }
    }
}
```

Este enfoque hace que el código sea más limpio y fácil de mantener, al reducir la cantidad de constructores secundarios necesarios.

2.7. Interfaces

Una interfaz define un contrato que debe ser cumplido por las clases que la implementan. Las interfaces especifican qué métodos y propiedades debe implementar una clase, sin proporcionar una implementación completa.

2.7.1. Características de las Interfaces

- **No instanciables:** No se pueden crear instancias directamente de una interfaz.
- **Sin estado:** Las interfaces no almacenan estado.

2.7.2. Definición de una Interfaz

Aquí tienes un ejemplo de una interfaz simple en Kotlin, que define propiedades de solo lectura:

```
interface ReadPlayer {
    val name: String
    val lifePoints: Int
}
```

2.7.3. Extensión de Interfaces

Las interfaces pueden heredar de otras interfaces. En este caso, una interfaz extendida puede sobreescribir las propiedades de la interfaz base para cambiar sus características, como convertir una propiedad de solo lectura en una propiedad mutable.

```
interface ReadWritePlayer : ReadPlayer {
    override var lifePoints: Int
}
```

2.7.4. Aspectos Importantes de la Herencia y Sobrescritura

- **Herencia:** Se declara usando el operador `:` después del nombre de la interfaz.
- **Sobrescritura explícita:** En Kotlin, cualquier miembro de una interfaz que se sobrescribe en una subinterfaz o en una clase implementadora debe ser marcado explícitamente con la palabra clave `override`.

2.7.5. Ejemplo Práctico

Aquí un ejemplo de cómo una clase puede implementar `ReadWritePlayer`:

```
class Player(override val name: String, override var lifePoints: Int) :
ReadWritePlayer {
    fun updateLifePoints(points: Int) {
        lifePoints += points
    }
}
```

Este código define una clase `Player` que implementa la interfaz `ReadWritePlayer`. La clase proporciona una implementación concreta para las propiedades definidas en la interfaz y agrega un método adicional para actualizar los puntos de vida del jugador.

2.8. Clases abstractas

Una clase abstracta es un tipo de clase que no está completa por sí misma; es decir, no puede ser instanciada directamente. Estas clases son útiles como base para otras clases, permitiendo compartir métodos y propiedades comunes mientras se obliga a las clases derivadas a implementar ciertos métodos.

2.8.1. Características de las Clases Abstractas

- **No Instanciables:** A diferencia de las clases regulares, no puedes crear instancias de una clase abstracta directamente.
- **Pueden Tener Estado:** A diferencia de las interfaces, las clases abstractas pueden tener propiedades con estado.
- **Uso como Tipo:** Generalmente, se recomienda no usar clases abstractas como tipo en parámetros, retornos de funciones o variables, para promover el uso de interfaces y favorecer la composición sobre la herencia.
- **Nomenclatura:** Como buena práctica, se sugiere que el nombre de una clase abstracta comience con la palabra `Abstract` para clarificar su propósito y naturaleza.

2.8.2. Definición de una Clase Abstracta

Aquí tienes un ejemplo de una clase abstracta que implementa la interfaz `ReadWritePlayer` y declara un método abstracto:

```
abstract class AbstractPlayer(
  override val name: String,
  override var lifePoints: Int
) : ReadWritePlayer {
    abstract fun attack(player: ReadWritePlayer)
}
```

2.8.3. Métodos Abstractos

- **Declaración Explícita:** Todos los métodos abstractos deben ser declarados explícitamente como tal. No tienen una implementación en la clase abstracta y deben ser implementados por cualquier clase no abstracta que herede de esta.

2.8.4. Ejemplo de Implementación

Una clase que extiende `AbstractPlayer` podría verse así:

```
class Warrior(name: String, lifePoints: Int) : AbstractPlayer(name, lifePoints) {
    override fun attack(player: ReadWritePlayer) {
        player.lifePoints -= 10  // Simula un ataque restando puntos de vida
    }
}
```

Este ejemplo muestra cómo una clase concreta `Warrior` implementa el método `attack` definido en `AbstractPlayer`. Al hacerlo, proporciona la funcionalidad específica para el método abstracto.

2.9. Clases Abiertas y Cerradas

2.9.1. Problema de la Base Frágil (Fragile Base Class Problem)

Este problema surge cuando cambios en una clase base causan que las clases derivadas dejen de funcionar correctamente. Una causa común es la sobreescritura no intencionada de miembros de la clase base, lo que puede alterar el comportamiento esperado de las clases derivadas.

Joshua Bloch, en su obra "Effective Java", resume este desafío con la frase: "Design and document for inheritance or else prohibit it". Esto sugiere que las clases deberían diseñarse con la herencia en mente o, de lo contrario, limitar quién puede extender de ellas para prevenir problemas.

2.9.2. Clases Abiertas y Cerradas

- **Clase Abierta:** En Kotlin, una clase es abierta a la extensión (puede ser heredada) solo si se marca explícitamente con la palabra clave `open`.
- **Clase Cerrada o Final:** Una clase es cerrada o final si no permite extensión. Esta es la configuración por defecto en Kotlin para evitar el problema de la base frágil inadvertidamente.

2.9.2.1. Métodos en Clases Abiertas

Los métodos en una clase abierta que se pueden sobreescribir deben ser marcados también como `open`. Por el contrario, los métodos en una clase cerrada no necesitan esta marca, ya que no pueden ser heredados.

2.9.2.2. Clases y Métodos Abstractos

Las clases y métodos abstractos son intrínsecamente abiertos porque esperan ser completados en las clases derivadas.

2.9.3. Ejemplos de Clases Abiertas y Cerradas

Clase Abierta:

```
// Esta clase es abierta, así que puede ser heredada.
open class UniversityStudent(
    name: String,
    val university: String
) : AbstractStudent(name) {
    override fun study() {
        println("Studying at $university")
    }

    open fun party() {
        println("Partying at $university")
    }
}
```

Clase Cerrada:

```
// Esta clase es cerrada, así que no puede ser heredada.
class PhDStudent(
    name: String,
    university: String,
    val thesisTopic: String
) : UniversityStudent(name, university) {
    override fun study() {
        println("I'm studying a lot!")
    }

    override fun party() {
        println("I'm too busy to party!")
    }
}
```

En el ejemplo de `PhDStudent`, se demuestra cómo una clase derivada puede sobrescribir métodos de su clase base abierta, pero al mismo tiempo, se declara como final para prevenir más herencia.

2.10. Sobrecarga de Operadores

La sobrecarga de operadores es una funcionalidad de algunos lenguajes de programación que permite a lxs desarrolladorxs definir implementaciones personalizadas para los operadores estándar como `+`, `-`, `*`, `/`, entre otros. En Kotlin, esto se logra mediante el uso de la palabra clave `operator` seguido de `fun`, indicando que se está definiendo una función que actúa como un operador.

2.10.1. Cómo Funciona

Para sobrecargar un operador, se declara una función miembro o una función de extensión con el prefijo `operator`. El nombre de la función debe corresponder a una de las funciones de operador predefinidas en Kotlin, como `plus`, `minus`, `times`, etc.

2.10.2. Ejemplo Práctico

Consideremos un ejemplo con un tipo de dato definido por el usuario, `Complex`, que representa un número complejo. Queremos permitir que estos números complejos se puedan sumar utilizando el operador `+`. Aquí está cómo podríamos implementar esto:

```
class Complex(val real: Double, val imaginary: Double) {
    operator fun plus(other: Complex) =
        Complex(real + other.real, imaginary + other.imaginary)
}
```

En este ejemplo:

- `Complex` es una clase que almacena dos propiedades: `real` e `imaginary`.
- `plus` es una función miembro que sobrecarga el operador `+`. Toma otro objeto `Complex` como parámetro y devuelve un nuevo objeto `Complex` cuyo valor real e imaginario es la suma de los valores correspondientes de los dos números complejos.

2.10.3. Uso

Gracias a la sobrecarga de operadores, podemos usar el operador `+` de forma natural con instancias de `Complex`:

```
val number1 = Complex(1.0, 2.0)
val number2 = Complex(3.0, 4.0)
val sum = number1 + number2
println("Sum: (${sum.real}, ${sum.imaginary})")
```

2.10.4. Consideraciones

Al diseñar clases que sobrecarguen operadores, es importante mantener el comportamiento intuitivo y esperado. Por ejemplo, asegurarse de que la operación sea conmutativa si eso tiene sentido para el tipo de datos, como es el caso de la suma de números complejos.

2.11. Propiedades en Kotlin

- Las propiedades en Kotlin son esencialmente variables que son miembros de una clase.
- Son equivalentes a los campos combinados con sus métodos getter y setter en otros lenguajes de programación como Java.
- Kotlin genera automáticamente getters para las propiedades inmutables y getters y setters para las propiedades mutables.

2.11.1. Ejemplo de Propiedad con Getter y Setter Personalizados

```
class GettersAndSetters {
    var name: String = "John"
        get() { // Getter personalizado
            return field
        }
        set(value) { // Setter personalizado
            field = value
        }
}
```

- `field` se refiere al valor almacenado de la propiedad.

2.11.2. Propiedad con Getter como Expresión

```
class GettersAndSetters {
    var name: String = "John"
        get() = field // Getter como expresión
        set(value) {
            field = value
        }
}
```

2.11.3. Propiedad con Getters y Setters por Defecto

```
class GettersAndSetters {
    var name: String = "John"
}
```

- Este código es equivalente a tener getters y setters por defecto.

2.11.4. Propiedad que Calcula su Valor

```
class GettersAndSetters {
    val now: String
```

```
        get() = Clock.System
                     .now()
                     .toString()
}
```

- Esta propiedad no almacena un valor, sino que calcula un valor cada vez que se accede a ella.

2.11.5. Propiedad con Setter Privado

```
class GettersAndSetters(age: Int) {
    var age: Int = age
        private set // Setter privado
}
```

- Podemos restringir la visibilidad del método set.

2.11.6. Propiedad con Backing Field

```
class GettersAndSetters(private var _age: Int) {
    val age: Int
        get() = _age
}
```

- Utilizando un backing field, podemos lograr lo mismo que con un setter privado.

Aquí tienes una versión mejorada y más explicativa del código sobre interfaces con getters y setters en Kotlin:

2.11.7. Interfaces con Getters y Setters Personalizados

En Kotlin, las interfaces pueden incluir propiedades con getters y setters personalizados, lo que permite añadir lógica adicional al acceder o modificar dichas propiedades. A continuación se presenta un ejemplo de una interfaz con getters y setters personalizados.

```
interface InterfaceWithGettersAndSetters {
    val name: String
        get() {
            println("Getting name")
            return "InterfaceWithGettersAndSetters"
        }
    var age: Int
        get() {
            println("Getting age")
            return 0
        }
        set(value) {
            println("Setting age")
        }
}
```

En este ejemplo, la interfaz `InterfaceWithGettersAndSetters` define dos propiedades: `name` y `age`. El getter de `name` imprime un mensaje y devuelve una cadena, mientras que el getter y el setter de `age` imprimen mensajes al acceder o modificar el valor de la propiedad. Este enfoque es útil para agregar lógica adicional, como validaciones o registros, al trabajar con propiedades.

La clase `ClassWithGettersAndSetters` implementa esta interfaz y proporciona su propia lógica para los getters y setters. Además, utiliza la palabra clave `super` para invocar los getters y setters de la interfaz.

```
class ClassWithGettersAndSetters : InterfaceWithGettersAndSetters {
    override val name: String
        get() {
            super.name
            return "ClassWithGettersAndSetters"
        }
    override var age: Int = 0
        get() {
            super.age
            return field
        }
        set(value) {
            super.age = value
            field = value
        }
}
```

En la clase `ClassWithGettersAndSetters`, el getter de `name` llama al getter de la interfaz y luego devuelve un valor diferente. El getter y el setter de `age` llaman a los métodos correspondientes de la interfaz antes de acceder o modificar el valor del campo respaldado (`field`). Este enfoque permite realizar validaciones o cualquier otra lógica adicional antes de completar la operación.

2.12. Funciones de Extensión

Las funciones de extensión en Kotlin permiten añadir nuevas funcionalidades a una clase sin necesidad de heredar de ella o modificar su código original. Estas funciones no modifican la clase extendida de ninguna manera y se resuelven estáticamente, lo que significa que no forman parte de la clase real.

Es importante notar que, si la clase original ya tiene un método con la misma firma que la función de extensión, el método original tendrá preferencia sobre la función de extensión.

2.12.1. Ejemplo de una Función de Extensión

```
fun String.replaceSpaces(replacement: String): String {
    return this.replace(" ", replacement)
}
```

En este ejemplo, estamos extendiendo la clase `String` con una nueva función `replaceSpaces`. La palabra clave `this` se refiere a la instancia de `String` sobre la que se llama la función.

2.12.2. Uso de la Función de Extensión

Una vez definida, la función de extensión puede ser llamada como si fuera un método miembro de la clase.

```
fun main() {
    val text = "Hello World"
    println(text.replaceSpaces("_"))
}
```

En este caso, la función `replaceSpaces` se llama sobre una instancia de `String`, reemplazando todos los espacios en blanco por guiones bajos.

2.12.3. Limitaciones de las Funciones de Extensión

Es importante tener en cuenta que las funciones de extensión no pueden acceder a los miembros protegidos o privados de la clase que están extendiendo. Esto asegura que no se violen los principios de encapsulación y acceso seguro a los datos.

2.12.4. Operadores de Extensión

Al igual que con las funciones de extensión, Kotlin permite definir operadores de extensión, que son operadores personalizados añadidos a las clases existentes.

2.12.4.1. Definición de un Operador de Extensión

```
operator fun String.times(n: Int): String = this.repeat(n)
```

En este ejemplo, estamos extendiendo la clase `String` con el operador de multiplicación (*). Este operador utiliza la función `repeat` para repetir la cadena `n` veces.

2.12.4.2. Uso del Operador de Extensión

Una vez definido, el operador de extensión puede ser utilizado como cualquier otro operador nativo de la clase.

```
fun main() {
    val text = "Hello"
    println(text * 3)  // Output: HelloHelloHello
}
```

En este caso, el operador * se utiliza para repetir la cadena `"Hello"` tres veces.

2.12.5. Propiedades de Extensión

En Kotlin, también es posible definir propiedades de extensión, lo que permite añadir nuevas propiedades a las clases existentes sin modificar su código fuente. Estas propiedades se implementan mediante getters y setters.

2.12.5.1. Definición de Propiedades de Extensión

Las propiedades de extensión permiten añadir nuevas propiedades a las clases existentes de manera similar a como se añaden nuevas funciones.

```
val String.wordCount: Int
    get() = this.split("""\s+""".toRegex()).size
```

En este ejemplo, se define una propiedad de extensión `wordCount` para la clase `String`, que cuenta el número de palabras en la cadena dividiéndola por espacios en blanco.

2.12.5.2. Propiedades de Extensión con Getter y Setter

También es posible definir propiedades de extensión con un setter, permitiendo modificar el valor de la propiedad.

```
var MutableList<String>.firstElement: String
    get() = this[0]
```

```
    set(value) {
        this[0] = value
    }
```

En este caso, se añade una propiedad de extensión `firstElement` a la clase `MutableList<String>`, que permite acceder y modificar el primer elemento de la lista.

2.12.5.3. Uso de Propiedades de Extensión

Una vez definidas, las propiedades de extensión pueden ser utilizadas como cualquier otra propiedad miembro de la clase.

```
fun main() {
    val text = "Hello World"
    println(text.wordCount)  // Output: 2

    val list = mutableListOf("Apple", "Banana", "Cherry")
    println(list.firstElement)  // Output: Apple
    list.firstElement = "Apricot"
    println(list.firstElement)  // Output: Apricot
}
```

En este ejemplo, la propiedad `wordCount` se utiliza para contar las palabras en una cadena, y la propiedad `firstElement` se utiliza para acceder y modificar el primer elemento de una lista mutable.

2.13. Funciones Infijas

Las funciones infijas en Kotlin permiten llamar a una función sin usar un punto y paréntesis, haciendo el código más legible. Estas funciones solo requieren un argumento y tienen ciertos requisitos para ser definidas.

2.13.1. Requisitos para Funciones Infijas

- Debe ser miembro de una clase o una extensión de una clase.
- Debe tener un solo parámetro.
- Debe estar marcada con el modificador `infix`.

2.13.2. Ejemplo de Función Infija en una Clase

```
class Point(val x: Int, val y: Int) {
    infix fun shift(dx: Int) = Point(x + dx, y)
}
```

En este ejemplo, `shift` es una función infija que desplaza el punto en el eje x por una cantidad `dx`.

```
fun main() {
    val point = Point(1, 2)
    val newPoint = point shift 3
    println("Point shifted to x=${newPoint.x}, y=${newPoint.y}")
}
```

En el código anterior, la función infija `shift` se llama sin paréntesis, haciendo el código más conciso y legible.

2.13.3. Ejemplo de Función Infija como Extensión

También es posible definir funciones infijas como extensiones de una clase.

```
class Point(val x: Int, val y: Int)

infix fun Point.shift(dx: Int) = Point(x + dx, y)
```

Aquí, `shift` se define como una función de extensión infija para la clase `Point`.

```
fun main() {
    val point = Point(1, 2)
    val newPoint = point shift 3
    println("Point shifted to x=${newPoint.x}, y=${newPoint.y}")
}
```

Este ejemplo demuestra cómo se puede usar una función infija de extensión de manera similar a una función infija miembro de una clase.

2.14. Data Classes

Las data classes en Kotlin están diseñadas principalmente para contener datos. Estas clases proporcionan automáticamente funcionalidades estándar como `toString()`, `equals()`, `hashCode()`, y `copy()` sin necesidad de implementarlas manualmente.

2.14.1. Características de las Data Classes

- **Funcionalidades Automáticas:**Las data classes generan automáticamente métodos como `toString()`, `equals()`, `hashCode()`, y `copy()`.
- **No Heredables:** No pueden ser heredadas, pero pueden ser extendidas mediante funciones de extensión.

2.14.2. Ejemplo de una Data Class

```
data class Person(val name: String, val age: Int)
```

En este ejemplo, `Person` es una data class que tiene dos propiedades: `name` y `age`.

2.14.3. Constructor Primario

Las data classes necesitan tener un constructor primario. Aunque se puede usar `var`, es recomendable utilizar `val` para propiedades inmutables y el método `copy` para crear nuevas instancias con modificaciones.

```
val person1 = Person("Alice", 29)
val person2 = person1.copy(age = 30)
```

En el ejemplo anterior, `person2` es una copia de `person1` pero con una edad modificada.

2.14.4. Descomposición de Data Classes

Las data classes pueden ser descompuestas en sus componentes individuales utilizando la sintaxis de desestructuración.

```

```

```
fun main() {
    val (name, age) = Person("Alice", 29)
    println("name = $name, age = $age")
}
```

En este ejemplo, la data class `Person` se descompone en las variables `name` y `age`.

2.15. Companion Object

Un `companion object` es un objeto que es común a todas las instancias de una clase. Está vinculado a la clase en sí misma en lugar de a sus instancias. Los `companion objects` proporcionan un lugar para almacenar propiedades y funciones que son lógicamente parte de una clase, pero que no requieren que se cree una instancia de la clase para ser accedidos.

2.15.1. Características del Companion Object

- **Acceso Estático:** Permite acceder a propiedades y métodos sin necesidad de instanciar la clase.
- **Extensión de Interfaces:** Pueden extender interfaces o clases, aunque esto es poco común.
- **No Heredados:** No son heredados por las subclases.

2.15.2. Ejemplo de Uso

```
class Number(val value: Int) {
    operator fun plus(other: Number) = Number(value + other.value)

    companion object {
        fun add(a: Int, b: Int) = Number(a + b)
    }
}
```

En este ejemplo, la clase `Number` tiene un `companion object` que contiene una función estática `add`. Esta función puede ser llamada sin crear una instancia de `Number`.

```
fun main() {
    val a = Number(10)
    println("10 + 5 = ${a + Number(5)}")
    println("10 + 5 = ${Number.add(10, 5)}")
}
```

En el código anterior, se muestra cómo utilizar tanto la función de operador `plus` como la función estática `add` definida en el `companion object`.

2.16. Enumeraciones

Vamos a desarrollar un sistema para una tienda en línea que necesita manejar distintos estados de pedidos:

- Pendiente
- Pagado
- Enviado
- Entregado
- Cancelado

2.16.1. Primer Enfoque: Strings

```
fun handleOrderState(state: String) = when (state) {
    "Pending" -> println("Order is pending")
    "Paid" -> println("Order is paid")
    "Shipped" -> println("Order is shipped")
    "Delivered" -> println("Order is delivered")
    "Cancelled" -> println("Order is cancelled")
    else -> println("Unknown state")
}
```

2.16.2. Problemas con el uso de Strings

Usar strings o enteros para representar estados puede llevar a varios problemas:

- **Errores en tiempo de ejecución**: Debido a valores inválidos o mal escritos, los estados pueden no ser manejados correctamente.
- **Manejo complicado**: El manejo de múltiples estados con estructuras de control puede volverse complicado y propenso a errores si los estados o las transiciones cambian.
- **Falta de verificación en tiempo de compilación**: No hay una verificación en tiempo de compilación para las transiciones de estado, lo que puede resultar en errores difíciles de detectar.

```
fun main() {
    handleOrderState("Delibered") // Estado incorrecto
}
```

2.16.3. Segundo Enfoque: Enumeraciones

Una enumeración (`enum`) es un tipo de dato especial que permite a los desarrolladores definir variables que pueden tomar uno de un conjunto fijo de constantes predefinidas. Las enumeraciones mejoran la legibilidad del código y reducen errores, al garantizar que las variables solo puedan contener uno de los valores definidos en el enum. Cada elemento de una enumeración puede actuar como una instancia de la enumeración.

2.16.3.1. Ventajas de las Enumeraciones

- **Seguridad de Tipos**: Aseguran que solo se puedan asignar valores válidos a las variables del tipo enumerado, evitando errores en tiempo de ejecución.
- **Legibilidad del Código**: Proporcionan nombres significativos para un conjunto de constantes, mejorando la claridad del código.
- **Mantenibilidad**: Facilitan la actualización y mantenimiento del código, ya que los valores válidos están centralizados y son fácilmente auditables.

2.16.3.2. Definición de Enumeraciones

Las enumeraciones se definen usando la palabra clave `enum class`. Aquí un ejemplo simple:

```
enum class Bool { TRUE, FALSE }
```

Este `enum` define un tipo `Bool` que puede tener uno de dos valores: `TRUE` o `FALSE`.

2.16.3.3. `when` exhaustivo

Un `when` es exhaustivo cuando cubre todas las posibilidades lógicas para la expresión que se está evaluando. Para enumeraciones, Kotlin fuerza que el `when` sea exhaustivo.

```
enum class DeliveryState {
    PENDING, PAID, SHIPPED, DELIVERED, CANCELLED
}
```

En el siguiente ejemplo, el `when` es exhaustivo porque cubre todos los posibles valores de la enumeración `DeliveryState`:

```
fun handleOrderState(state: DeliveryState) = when (state) {
    DeliveryState.PENDING -> println("Order is pending")
    DeliveryState.PAID -> println("Order is paid")
    DeliveryState.SHIPPED -> println("Order is shipped")
    DeliveryState.DELIVERED -> println("Order is delivered")
    DeliveryState.CANCELLED -> println("Order is cancelled")
    else -> println("Unknown state")  // Este else es redundante si el when es
exhaustivo
}
```

Cuando el `when` es exhaustivo, el bloque `else` puede ser omitido:

```
fun handleOrderState(state: DeliveryState) = when (state) {
    DeliveryState.PENDING -> println("Order is pending")
    DeliveryState.PAID -> println("Order is paid")
    DeliveryState.SHIPPED -> println("Order is shipped")
    DeliveryState.DELIVERED -> println("Order is delivered")
    DeliveryState.CANCELLED -> println("Order is cancelled")
}
```

Aquí tienes un ejemplo de uso en el método `main`:

```
fun main() {
    handleOrderState(DeliveryState.PENDING)
    handleOrderState(DeliveryState.PAID)
    handleOrderState(DeliveryState.SHIPPED)
    handleOrderState(DeliveryState.DELIVERED)
    handleOrderState(DeliveryState.CANCELLED)
}
```

En este código, la función `handleOrderState` maneja todos los posibles estados de un pedido utilizando un `when` exhaustivo, asegurando que no haya estados no manejados.

Aquí tienes la versión mejorada del texto:

2.16.3.4. Métodos en enumeraciones

Las enumeraciones pueden tener métodos abstractos que deben ser sobrescritos por cada uno de los casos de la enumeración. También pueden tener métodos concretos que serán heredados por cada uno de los elementos.

```
enum class DeliveryState {
    PENDING {
        override fun signal() = "Order is pending"
    },
```

```
    PAID {
        override fun signal() = "Order is paid"
    },
    SHIPPED {
        override fun signal() = "Order is shipped"
    },
    DELIVERED {
        override fun signal() = "Order is delivered"
    },
    CANCELLED {
        override fun signal() = "Order is cancelled"
    }; // El ; es necesario si hay métodos o propiedades en la enumeración

    abstract fun signal(): String

    // Método concreto heredado por todos los elementos de la enumeración
    fun isFinalState() = this == DELIVERED || this == CANCELLED
}
```

En este ejemplo, cada estado de la enumeración `DeliveryState` sobrescribe el método abstracto `signal`, proporcionando un mensaje específico para cada estado. Además, la enumeración tiene un método concreto `isFinalState` que determina si el estado es final (es decir, `DELIVERED` o `CANCELLED`). Este método concreto es heredado por todos los elementos de la enumeración.

Aquí tienes la versión mejorada del texto:

2.16.3.5. Herencia en enumeraciones

Las enumeraciones pueden implementar interfaces, pero no pueden ser heredadas.

```
// Definición de la interfaz State
interface State {
    fun signal(): String
}
```

```
// Implementación de la interfaz State por la enumeración DeliveryState
enum class DeliveryState : State {
    PENDING {
        override fun signal() = "Order is pending"
    },
    PAID {
        override fun signal() = "Order is paid"
    },
    SHIPPED {
        override fun signal() = "Order is shipped"
    },
    DELIVERED {
        override fun signal() = "Order is delivered"
    },
    CANCELLED {
        override fun signal() = "Order is cancelled"
    };

    // Método concreto heredado por todos los elementos de la enumeración
```

```
    fun isFinalState() = this == DELIVERED || this == CANCELLED
}
```

En este ejemplo, la enumeración `DeliveryState` implementa la interfaz `State`, lo que obliga a cada uno de los estados de la enumeración a sobrescribir el método `signal`. Además, la enumeración incluye un método concreto `isFinalState` que determina si el estado es final (es decir, `DELIVERED` o `CANCELLED`). Este método concreto es heredado por todos los elementos de la enumeración.

```
fun handleOrderState(state: DeliveryState) = if (state.isFinalState()) {
    println("Final state: ${state.signal()}")
} else {
    println("Non-final state: ${state.signal()}")
}
```

Se puede acceder a todas las entradas con entries

```
fun listOrderStates() = DeliveryState.entries.forEach { println(it) }
```

Puedo "buscar" un enum con valueOf, si el valor no existe se arroja una excepción

```
fun getOrderState(name: String) = DeliveryState.valueOf(name)
```

Ejercicio: Interfaz y Enumeración de Acciones del Juego

Implementa una interfaz `GameAction`, una clase `Player` y una enumeración `GameEvent` que representen las acciones de un juego.

Instrucciones:

1. **Interfaz `GameAction`**:
   - Define una interfaz `GameAction` que incluya un método `execute(player: Player)`.
   - El método `execute` debe afectar al estado de un jugador de acuerdo con el tipo de evento.
2. **Clase `Player`**:
   - Crea una clase `Player` que contenga dos propiedades: `healthPoints` y `manaPoints`.
   - La clase debe incluir métodos para aumentar y disminuir los puntos de salud (`healthPoints`) y los puntos de maná (`manaPoints`) en una cantidad dada.
   - No es necesario considerar casos de borde ni validaciones de datos.
3. **Enumeración `GameEvent`**:
   - Crea una enumeración `GameEvent` que implemente la interfaz `GameAction`.
   - Cada constante de la enumeración debe sobrescribir el método `execute` y definir cómo afecta al jugador.

Aquí tienes una versión mejorada de la sección sobre las limitaciones de las enumeraciones:

2.16.3.6. Limitaciones de las Enumeraciones

- **Datos Asociados**: Los `enum` no pueden tener datos asociados específicos de instancia sin definirlos de manera estática para todos los estados. Esto significa que no puedes agregar información dinámica a cada instancia sin complicar el diseño.

- **Información Dinámica**: Tienen una capacidad limitada para manejar información dinámica, como un identificador de seguimiento para el estado `SHIPPED`. No es posible asignar datos específicos de instancia sin definirlos estáticamente para todos los estados.
- **Métodos Abstractos y Propiedades**: Aunque los `enum` pueden tener métodos abstractos que los estados individuales implementan, agregar nuevos métodos o propiedades que solo se aplican a algunos estados puede volverse complicado y menos intuitivo. Esto puede llevar a un diseño inconsistente y difícil de mantener.
- **Complejidad del Manejo del Estado**: Si el manejo del estado se vuelve más complejo y requiere más lógica y datos, mantener todo dentro de una definición de `enum` puede hacer que la clase sea demasiado pesada y difícil de mantener. En tales casos, es mejor considerar el uso de otras estructuras de datos más flexibles, como clases selladas.

Estas limitaciones pueden hacer que el uso de enumeraciones no sea la mejor opción para sistemas con estados muy dinámicos o complejos.

2.17. Respuestas:

Ejercicio: Implementación de un Gestor de Eventos

```
object EventManager {
    val events = mutableListOf<String>()

    fun addEvent(event: String) {
        events.add(event)
    }

    fun getEvents(): List<String> {
        return events.toList()
    }
}
```

Ejercicio: Interfaz y Enumeración de Acciones del Juego

```
class Player(health: Int, mana: Int) {
    var health = health
        private set
    var mana = mana
        private set

    fun increaseHealth(amount: Int) {
        health += amount
    }

    fun decreaseHealth(amount: Int) {
        health -= amount
    }

    fun increaseMana(amount: Int) {
        mana += amount
    }

    fun decreaseMana(amount: Int) {
        mana -= amount
```

```
    }
}
```

```
enum class GameEvent : GameAction {
    HEALTH_BOOST {
        override fun execute(player: Player) {
            player.increaseHealth(20)
        }
    },
    MANA_DRAIN {
        override fun execute(player: Player) {
            player.decreaseMana(15)
        }
    },
    POISON {
        override fun execute(player: Player) {
            player.decreaseHealth(10)
        }
    },
    ENERGY_SURGE {
        override fun execute(player: Player) {
            player.increaseMana(20)
        }
    };
}
```

3. Unidad 3: Build systems

Definición 3.1 (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.

3.1. 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.

3.2. 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.

3.3. 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.

3.3.1. 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.

3.3.2. 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)
}
```

3.3.3. `gradle.properties`

El archivo `gradle.properties` es un archivo de configuración para Gradle que utiliza el formato `clave=valor` para definir las propiedades. Este archivo permite definir propiedades que actúan como variables de entorno durante la compilación. Es una buena práctica almacenar valores como las versiones de las librerías utilizadas en el proyecto en este archivo.

```
kotlin.code.style=official
kotlin.version=2.0.0
build-systems-kt.version=1.0.0

org.gradle.parallel=true
org.gradle.caching=true

detekt.version=1.23.1
dokka.version=1.9.10
kotest.version=5.9.0
kotlinx.datetime.version=0.4.1
```

- `kotlin.code.style=official`:
  - Define el estilo de código de Kotlin a utilizar, en este caso, el estilo oficial recomendado por Kotlin.
- `kotlin.version=2.0.0`:
  - Especifica la versión de Kotlin a utilizar en el proyecto.
- `build-systems-kt.version=1.0.0`:
  - Define la versión del proyecto.
- `org.gradle.parallel=true`:
  - Habilita la construcción en paralelo para mejorar el rendimiento.
- `org.gradle.caching=true`:
  - Habilita la caché de Gradle para acelerar la construcción reutilizando los resultados de compilaciones anteriores.
- `detekt.version=1.23.1`:
  - Especifica la versión de Detekt, una herramienta de análisis estático de código para Kotlin.
- `dokka.version=1.9.10`:
  - Define la versión de Dokka, una herramienta de documentación para Kotlin.
- `kotest.version=5.9.0`:
  - Especifica la versión de Kotest, un framework de pruebas para Kotlin.
- `kotlinx.datetime.version=0.4.1`:
  - Define la versión de `kotlinx-datetime`, una librería de fecha y hora para Kotlin.

3.3.4. 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.

1. **Google**: Especialmente importante para proyectos Android, el repositorio de Google alberga librerías y herramientas específicas necesarias para el desarrollo en Android.

1. **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.

1. **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.

1. **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.

3.3.5. Dependencias

Definición 3.3.5.1 (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.

```
dependencies {
    implementation(kotlin("reflect"))
    testImplementation("group:name:version")
```

```
    implementation(group = "group", name = "name", version = "version")
}
```

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 = extra["kotest.version"] as String

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

3.3.6. `settings.gradle.kts`

El archivo `settings.gradle.kts` se utiliza para configurar y gestionar los ajustes de configuración de un proyecto Gradle. Este archivo define la estructura de módulos del proyecto y puede incluir configuraciones adicionales para la gestión de plugins.

```
// El nombre del proyecto raíz
rootProject.name = "build-systems-kt"

// Incluye los subproyectos del proyecto raíz
include(
    ":subproject1",
    ":subproject2",
    ":subproject3"
)
```

A continuación, un ejemplo más completo que incluye la configuración de la gestión de plugins:

```
// Define el nombre del proyecto raíz
rootProject.name = "build-systems-kt"

// Incluye los subproyectos
include(":subproject1", ":subproject2", ":subproject3")

pluginManagement {
    // Define los repositorios para la resolución de plugins
    repositories {
        gradlePluginPortal()
    }

    // Configura los plugins con sus respectivas versiones
    plugins {
        kotlin("jvm") version extra["kotlin.version"] as String
       id("io.gitlab.arturbosch.detekt") version extra["detekt.version"] as String
    }
}
```

- `rootProject.name = "build-systems-kt"`:

- Define el nombre del proyecto raíz, que es el identificador principal del proyecto en Gradle.

- `include(":subproject1", ":subproject2", ":subproject3")`:
  - Incluye los subproyectos en el proyecto raíz. Cada subproyecto puede tener su propia configuración y código independiente.

- `pluginManagement`:
  - Configura la gestión de plugins para el proyecto. Esto incluye la definición de los repositorios y las versiones de los plugins necesarios.

- `repositories { gradlePluginPortal() }`:
  - Define el repositorio `gradlePluginPortal` para resolver y descargar los plugins utilizados en el proyecto.

- `plugins`:
  - Especifica los plugins y sus versiones que serán utilizados en el proyecto. En este ejemplo, se utilizan los plugins de Kotlin JVM y Detekt con versiones definidas en el archivo `gradle.properties`.

Aquí tienes la versión mejorada usando la sintaxis de Typst:

3.3.7. `build.gradle.kts`

El archivo `build.gradle.kts` es el script principal de Gradle que define las tareas y configuraciones de construcción del proyecto. En este archivo se especifican las dependencias, plugins, tareas personalizadas y otros ajustes necesarios para compilar y ejecutar el proyecto.

```
// Define los plugins que se aplicarán al proyecto raíz
plugins {
    kotlin("jvm")
}

// Configura el grupo y la versión para todos los proyectos
allprojects {
    group = "build-systems-kt"
    version = extra["build-systems-kt.version"] as String
}

// Define el repositorio de Maven Central como fuente para las dependencias de los
subproyectos
subprojects {
    repositories {
        mavenCentral()
    }
}
```

- **plugins**:
  - Define los plugins que se aplicarán al proyecto raíz.
  - En este caso, se aplica el plugin de Kotlin para JVM.
- **allprojects**:
  - Configura el grupo y la versión para todos los proyectos.
  - El grupo se establece como “build-systems-kt” y la versión se obtiene de las propiedades extra.
- **subprojects**:
  - Define el repositorio de Maven Central como fuente para las dependencias de los subproyectos.
  - Esto asegura que todas las dependencias necesarias para los subproyectos se puedan resolver desde Maven Central.

3.3.7.1. `subproject1/build.gradle.kts`

```
val kotestVersion = extra["kotest.version"] as String
val kotlinxDatetimeVersion = extra["kotlinx.datetime.version"] as String

plugins {
    id("io.gitlab.arturbosch.detekt")
    kotlin("jvm")
}

dependencies {
    implementation(kotlin("reflect"))
                    implementation("org.jetbrains.kotlinx:kotlinx-datetime:
$kotlinxDatetimeVersion")
    testImplementation("io.kotest:kotest-property:$kotestVersion")
    testImplementation("io.kotest:kotest-runner-junit5:$kotestVersion")
    testImplementation("io.kotest:kotest-framework-datatest:$kotestVersion")
}

tasks.test {
    useJUnitPlatform()
}

kotlin {
    jvmToolchain(17)
}
```

- **kotestVersion** y **kotlinxDatetimeVersion**:
  - Obtiene las versiones de Kotest y kotlinx-datetime de las propiedades extra.
- **plugins**:
  - Aplica el plugin de Detekt para análisis estático y el plugin de Kotlin para JVM.
- **dependencies**:
  - Define las dependencias necesarias para el subproyecto:
    - `implementation(kotlin("reflect"))`: Añade la dependencia de reflexión de Kotlin.
    - `implementation("org.jetbrains.kotlinx:kotlinx-datetime: $kotlinxDatetimeVersion")`: Añade la dependencia de kotlinx-datetime con la versión especificada.
    - `testImplementation("io.kotest:kotest-property:$kotestVersion")`: Añade la dependencia de Kotest para pruebas de propiedades.
    - `testImplementation("io.kotest:kotest-runner-junit5:$kotestVersion")`: Añade la dependencia de Kotest para JUnit5.
    - `testImplementation("io.kotest:kotest-framework-datatest:$kotestVersion")`: Añade la dependencia de Kotest para pruebas basadas en datos.
- **tasks.test**:
  - Configura las pruebas para usar JUnit Platform.
- **kotlin**:
  - Configura el toolchain de JVM para usar la versión 17.

3.3.8. Construyendo el proyecto

```
# Windows
.\gradlew.bat :subproject1:build
.\gradlew.bat build
```

```
# Unix
./gradlew :subproject1:build
./gradlew build
```

3.4. 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.

3.4.1. 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.

3.4.2. 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
```

3.5. Tareas

Una tarea es una unidad de trabajo que Gradle puede ejecutar. Puede representar cualquier cosa, desde compilar código, ejecutar pruebas, generar documentación, hasta desplegar una aplicación. Las tareas se definen en algún archivo `build.gradle.kts` y pueden ser muy simples o bastante complejas dependiendo de lo que necesiten hacer.

3.5.1. Tareas Predefinidas

Gradle proporciona un conjunto de tareas predefinidas que son comunes en la mayoría de los proyectos. Algunos ejemplos de tareas predefinidas son:

3.5.1.1. Ejemplo de Tarea de Prueba

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

Esta tarea configura Gradle para utilizar el motor de JUnit para correr los tests.

3.5.1.2. Ejemplo de Tarea de Copiado

```
tasks.create<Copy>("copy") {
    // Es buena práctica añadir esto a nuestras tareas
    description = "Copies sources to the destination directory"
    group = "Custom"

    from("src")
    into("dst")
}
```

Esta tarea de copiado mueve archivos desde el directorio `src` al directorio `dst`.

- **description**:
  - Proporciona una descripción de la tarea.

  Es una buena práctica para hacer nuestras tareas más comprensibles.
- **group**:
  - Asigna la tarea a un grupo.

  En este caso, se asigna al grupo “Custom”.

3.5.1.3. Ejecución de Tareas Predefinidas

```
# Windows
.\gradlew.bat test
.\gradlew.bat copy
```

```
# Unix
./gradlew test
./gradlew copy
```

Para ejecutar las tareas predefinidas, utiliza el comando `gradlew` seguido del nombre de la tarea. En Windows, usa `.\gradlew.bat`, y en Unix, usa `./gradlew`.

3.5.1.4. Tareas Predefinidas Comunes

- **assemble**:
  - Ensambla el proyecto completo.
- **build**:
  - Construye el proyecto.
- **clean**:
  - Elimina los archivos generados por la construcción previa.
- **check**:
  - Ejecuta todas las verificaciones, incluyendo pruebas.

Estas tareas predefinidas facilitan la configuración y ejecución de las operaciones comunes en los proyectos Gradle.

3.5.2. Acciones

- `doFirst`:
  - Agrega una acción que se ejecutará al inicio de la tarea, antes de cualquier otra acción configurada en la tarea.
  - Se usa cuando necesitas realizar alguna preparación o configuración previa antes de la ejecución principal de la tarea.
- `doLast`:
  - Agrega una acción que se ejecutará al final de la tarea, después de todas las demás acciones configuradas en la tarea.
  - Se usa cuando necesitas realizar alguna operación de limpieza, validación final o cualquier acción que debe ejecutarse después de la ejecución principal de la tarea.

3.5.2.1. Ejemplo de Uso de `doFirst` y `doLast`

```
tasks.register("Fib") {
    var first = 0
    var second = 1
    doFirst {
        println("What's going on?")
        for (i in 1..11) {
            second += first
            first = second - first
        }
    }
    doLast {
        println("The 12th Fibonacci number is $second")
```

```
    }
}
```

En este ejemplo:

- **doFirst**:
  - Imprime un mensaje antes de iniciar el cálculo de la serie de Fibonacci y calcula los primeros 12 números de la serie.
- **doLast**:
  - Imprime el 12.º número de Fibonacci después de que se ha completado el cálculo.

Para ejecutar esta tarea, utiliza el comando:

```
# Windows
.\gradlew.bat Fib
```

```
# Unix
./gradlew Fib
```

Aquí tienes la versión mejorada y desarrollada usando la sintaxis de Typst:

3.5.2.2. Orden de ejecución de acciones

Las acciones agregadas con `doFirst` se ejecutan en el orden inverso en que se definen (las últimas se ejecutan primero). Las acciones agregadas con `doLast` se ejecutan en el orden en que se definen (las primeras se ejecutan primero).

Ejercicio: Orden de Ejecución de Acciones

¿Qué se imprimirá al ejecutar la siguiente tarea?

```
tasks.register("advancedTask") {
    doFirst {
          println("First action - Preparation Step 1")
    }
    doFirst {
          println("First action - Preparation Step 2")
    }
    doLast {
          println("Last action - Cleanup Step 1")
    }
    doLast {
          println("Last action - Cleanup Step 2")
    }
}
```

3.5.3. Dependencia de Tareas

Las tareas pueden depender de otras tareas. Esto asegura que las tareas se ejecuten en un orden específico.

```
tasks.register("cleanAll") {
    // Define una tarea que limpia todos los subproyectos
    dependsOn(":subproject1:clean", ":subproject2:clean", ":subproject3:clean")
}
```

En este ejemplo, la tarea `cleanAll` depende de las tareas `clean` de los subproyectos `subproject1`, `subproject2`, y `subproject3`. Esto significa que cuando se ejecuta `cleanAll`, Gradle primero ejecutará las tareas `clean` de cada uno de los subproyectos especificados.

Ejercicio: Tamaño del Proyecto

Crea una tarea de Gradle que calcule el tamaño de su proyecto luego de compilarse. Para esto, considera que puedes acceder a los archivos compilados haciendo `project.fileTree("build/classes/kotlin/main").files`. Luego, puedes acceder al tamaño del archivo con el método `length()`. Incluye un grupo y descripción para tu tarea. Tu tarea debe depender de `compileKotlin`.

Aquí tienes la versión mejorada y desarrollada usando la sintaxis de Typst:

3.5.4. Tareas como clases

Supongamos que tenemos que calcular el número de Fibonacci de forma repetida. Podemos definir una clase (abstracta)[1] para representar familias de tareas.

3.5.4.1. Caso de estudio: Tarea de Fibonacci

```
abstract class FibonacciTask : DefaultTask() {  // 1
    @get:Input
    abstract val number: Property<Int>  // 2

    @TaskAction // 3
    fun calculateFibonacci() {
        val n = number.get()
        if (n < 0) {
            throw StopExecutionException("The number must be positive")
        }
        var first = 0
        var second = 1
        repeat(n) {
            second += first
            first = second - first
        }
        println("The $n-th Fibonacci number is $first")
    }
}
```

1. **Todas las tareas deben extender de DefaultTask**:
   - Esto asegura que nuestra clase personalizada se comporte como una tarea de Gradle.
2. **Podemos definir inputs para nuestra tarea**:
   - Utilizamos propiedades anotadas con `@Input` para definir los parámetros que nuestra tarea necesitará.

[1] Podemos usar una clase abierta, pero tiene más sentido que sea abstracta ya que es abierta por definición y sólo será ocupada para ser extendida.

3. **`@TaskAction` marca el código principal que ejecutará la tarea**:
   - Este método contiene la lógica principal que se ejecutará cuando la tarea se ejecute.

3.5.4.2. Registro de Tareas

```
tasks.register<FibonacciTask>("Fib_10") {
    number.set(10)
}

tasks.register<FibonacciTask>("Fib_20") {
    number.set(20)
}
```

En este ejemplo, registramos dos instancias de la tarea `FibonacciTask`, una para calcular el 10.º número de Fibonacci y otra para el 20.º.

3.5.4.3. Ejecución de Tareas

```
# Windows
.\gradlew.bat Fib_10
.\gradlew.bat Fib_20
```

```
# Unix
./gradlew Fib_10
./gradlew Fib_20
```

Para ejecutar las tareas `Fib_10` y `Fib_20`, utiliza el comando `gradlew` seguido del nombre de la tarea correspondiente. En Windows, usa `.\gradlew.bat`, y en Unix, usa `./gradlew`.

Ejercicio: Tarea de Fibonacci

¿Qué imprime el siguiente código?

```
tasks.register<FibonacciTask>("Fib_10") {
    number.set(10)
    doFirst {
        println("Preparing to calculate the 10th Fibonacci number")
    }
    doLast {
        println("Finished calculating the 10th Fibonacci number")
    }
}
```

3.5.4.4. Caso de estudio: Procesar archivos de texto

Implementaremos una tarea para procesar archivos de texto. Tendremos archivos de entrada y salida. El texto debe ser tomado de un archivo de texto (input), transformado en mayúsculas y ser guardado en otro archivo de texto (output).

```
abstract class TextProcessingTask : DefaultTask() {
```

```
    @get:InputFile
    abstract val inputFile: RegularFileProperty

    @get:OutputFile
    abstract val outputFile: RegularFileProperty

    @TaskAction
    fun processText() {
        val inputText = inputFile.get().asFile.readText()
        val processedText = inputText.uppercase(Locale.getDefault())
        outputFile.get().asFile.writeText(processedText)
        println(
            "Text has been processed and written to " +
                outputFile.get().asFile.absolutePath
        )
    }
}
```

3.5.4.4.1. Registro de la Tarea

```
tasks.register<TextProcessingTask>("processText") {
    inputFile.set(file("src/main/resources/input.txt"))
    outputFile.set(file("build/output.txt"))

    // Acción que se ejecuta antes de la acción principal
    doFirst {
        println(
           "Preparing to process text from ${inputFile.get().asFile.absolutePath}"
        )
    }

    // Acción que se ejecuta después de la acción principal
    doLast {
        println(
          "Finished processing text. Output written to " +
              outputFile.get().asFile.absolutePath
        )
    }
}
```

3.5.4.4.2. Ejecución de la Tarea

```
# Windows
.\gradlew.bat processText
```

```
# Unix
./gradlew processText
```

Para ejecutar la tarea `processText`, utiliza el comando `gradlew` seguido del nombre de la tarea correspondiente. En Windows, usa `.\gradlew.bat`, y en Unix, usa `./gradlew`.

3.5.4.4.3. Explicación del Código

- **TextProcessingTask**:

- **`@InputFile` y `@OutputFile`**:
  - Define los archivos de entrada y salida usando las propiedades `RegularFileProperty`.
- **`@TaskAction`**:
  - Marca el método `processText` como la acción principal de la tarea.
  - Lee el texto del archivo de entrada, lo transforma a mayúsculas y lo escribe en el archivo de salida.

- **Registro de la tarea**:
  - Configura los archivos de entrada y salida.
  - Usa `doFirst` para realizar acciones antes de la tarea principal y `doLast` para acciones después de la tarea principal.

3.6. Plugins

Los plugins extienden las capacidades de Gradle al agregar nuevas tareas, configuraciones y funcionalidades a los scripts de build. Permiten la reutilización de configuraciones de build sin necesidad de duplicar código en múltiples proyectos.

```
plugins {
    kotlin("jvm") version "2.0.0"
    id("io.gitlab.arturbosch.detekt") version "1.23.6"
}
```

3.6.1. Plugins vs Tareas

3.6.1.1. Propósito

Tareas:
- Se utilizan para definir una acción específica dentro del build script.
- Las tareas son componentes individuales que realizan trabajos concretos.

Plugins:
- Son componentes reutilizables que pueden agregar y configurar múltiples tareas y extender la funcionalidad del build script.
- Los plugins encapsulan lógica y configuración que puede aplicarse a uno o más proyectos.

3.6.1.2. Alcance

Tareas:
- Tienen un alcance limitado al build script o proyecto en el que se definen.
- Las tareas pueden ser específicas a una parte del proceso de construcción.

Plugins:
- Tienen un alcance más amplio, ya que pueden aplicarse a múltiples proyectos y pueden incluir lógica para configurar y extender esos proyectos de manera más compleja.

Aquí tienes la versión mejorada y desarrollada usando la sintaxis de Typst:

3.6.2. Plugins personalizados

Los plugins personalizados se crean para extender Gradle con funcionalidades específicas a las necesidades de un proyecto o equipo. Estos plugins permiten:

- Automatizar tareas repetitivas.
- Centralizar configuraciones comunes.
- Modularizar scripts de construcción para mejorar la mantenibilidad.

Se pueden crear como clases dentro del proyecto o como proyectos independientes (no haremos esto último).

3.7. Ejemplo de Plugin Personalizado

```
// build.gradle.kts
class SamplePlugin : Plugin<Project> {
    override fun apply(target: Project) {
        target.tasks.register("pluginTask") {
            doLast {
                println("A plugin task was called")
            }
        }
    }
}

apply<SamplePlugin>()
```

En este ejemplo, se define un plugin personalizado `SamplePlugin` que registra una tarea llamada `pluginTask` que imprime un mensaje cuando se ejecuta.

3.8. Ejecución del Plugin

```
# Windows
.\gradlew.bat pluginTask
```

```
# Unix
./gradlew pluginTask
```

Para ejecutar la tarea `pluginTask` definida en el plugin personalizado, utiliza el comando `gradlew` seguido del nombre de la tarea correspondiente. En Windows, usa `.\gradlew.bat`, y en Unix, usa `./gradlew`.

3.9. Beneficios de los Plugins Personalizados

- **Automatización**:
  - Los plugins automatizan tareas repetitivas, reduciendo errores manuales y aumentando la eficiencia.
- **Centralización**:
  - Centralizan configuraciones comunes, facilitando la gestión y actualización de las mismas.
- **Modularización**:
  - Modularizan scripts de construcción, mejorando la mantenibilidad y organización del código de construcción.

Ejercicio: Reporte de archivos

Implementa un plugin que se encargue de generar un reporte con la lista de todos los archivos `.kt` y `.kts` mediante una tarea. Basta con que imprima en pantalla los nombres de todos los archivos. Puedes obtener la lista de archivos haciendo:

```

```

```
val kotlinFiles = project.fileTree(".").matching {
    include("**/*.kt", "**/*.kts")
}.files
```

3.10. Compilación de bibliotecas

Creemos un proyecto simple que se compone de dos servicios:

- **EchoLib**: Una librería con una función `id` que recibe un string de `texto` y retorna el mismo `texto` sin transformaciones.
- **EchoApp**: Una aplicación que pide palabras a un usuario y las imprime sin realizar modificación.

Recordemos que queremos que nuestras APIs:

- Estén versionadas, por lo que necesitamos almacenar el número de versión.
- Sigan un estándar de formato, por lo que es conveniente configurar una herramienta de análisis estático como Detekt.

3.10.1. Configuración de Propiedades

```
# gradle.properties
kotlin.code.style=official
kotlin.version=2.0.0
echo.version=1.0.0
detekt.version=1.23.1
```

3.10.2. Configuración de Proyectos

```
// settings.gradle.kts
rootProject.name = "echo"
include("echoLib", "echoApp")

pluginManagement {
    repositories {
        gradlePluginPortal()
    }

    plugins {
        kotlin("jvm") version extra["kotlin.version"] as String
       id("io.gitlab.arturbosch.detekt") version extra["detekt.version"] as String
    }
}
```

3.10.3. Configuración del Proyecto Raíz

```
// build.gradle.kts
plugins {
    kotlin("jvm")
    id("io.gitlab.arturbosch.detekt")
}

allprojects {
    group = "cl.ravenhill.echo"
    version = extra["echo.version"] as String
```

```
}

subprojects {
    repositories {
        mavenCentral()
    }
}
```

3.10.4. Configuración del Proyecto EchoApp

```
// echoApp/build.gradle.kts
plugins {
    id("io.gitlab.arturbosch.detekt")
    kotlin("jvm")
}

kotlin {
    jvmToolchain(17)
}
```

3.10.5. Configuración del Proyecto EchoLib

```
// echoLib/build.gradle.kts
plugins {
    id("io.gitlab.arturbosch.detekt")
    kotlin("jvm")
}

kotlin {
    jvmToolchain(17)
}
```

3.10.6. Implementación de la Librería

Supongamos que nuestra librería se compone de una función `id(String): String`. Esta función recibe un parámetro y lo retorna sin modificaciones.

```
// echoLib/src/main/kotlin/cl/ravenhill/echo/util/Id.kt
package cl.ravenhill.echo.util

fun id(text: String): String = text
```

3.10.7. Creando una biblioteca .jar

Un archivo JAR (Java ARchive) es un formato de archivo de paquete utilizado para agrupar múltiples archivos de clase Java, metadatos y recursos (como textos, imágenes, etc.) en un solo archivo para distribución y despliegue. Los archivos JAR se basan en el formato de archivo ZIP y tienen una estructura similar.

Dentro del archivo JAR, hay un directorio especial llamado META-INF que contiene un archivo llamado MANIFEST.MF. El archivo de manifiesto contiene metadatos sobre el archivo JAR, como la versión, la clase principal (para JARs ejecutables), y otra información del paquete. Para los JARs ejecutables, el archivo de manifiesto especifica el punto de entrada de la aplicación utilizando el atributo Main-Class.

3.10.7.1. Configuración del Archivo JAR en echoLib

```
// echoLib/build.gradle.kts
/* ... */
tasks.jar {
    manifest {
        attributes["Implementation-Title"] = "echoLib"
        attributes["Implementation-Version"] = project.version
    }
    from(sourceSets.main.get().output)
}
```

3.10.7.2. Copia del Archivo JAR en echoApp

```
// build.gradle.kts
/* ... */
tasks.register<Copy>("copyLib") {
    group = "build"
    description = "Copia el archivo JAR de la biblioteca a la aplicación"
    dependsOn(":echoLib:jar")
    from("echoLib/build/libs")
    into("echoApp/libs")
}
```

3.10.7.3. Configuración de Dependencias en echoApp

```
// echoApp/build.gradle.kts
/* ... */
repositories {
    flatDir {
        dirs("libs")
    }
}

dependencies {
    implementation(fileTree("libs") { include("*.jar") })
}
```

3.10.7.4. Explicación del Código

- **Configuración del Archivo JAR en echoLib**:
  - Configura la tarea `jar` para incluir un manifiesto con los atributos `Implementation-Title` y `Implementation-Version`.
  - Incluye los archivos de salida del conjunto de fuentes principal en el archivo JAR.
- **Copia del Archivo JAR en echoApp**:
  - Registra una tarea `copyLib` que copia el archivo JAR generado en `echoLib` al directorio `libs` de `echoApp`.
  - La tarea `copyLib` depende de la tarea `jar` en `echoLib`, asegurando que el archivo JAR se genere antes de copiarlo.
- **Configuración de Dependencias en echoApp**:
  - Configura el repositorio `flatDir` para incluir el directorio `libs`.

- Define una dependencia de implementación que incluye todos los archivos JAR en el directorio `libs`.

3.10.8. Creando un Ejecutable

Supongamos que tenemos un archivo `Echo.kt` en el paquete `cl.ravenhill.echo` con el siguiente código:

```
// echoApp/src/main/kotlin/cl/ravenhill/echo/Echo.kt
package cl.ravenhill.echo

import cl.ravenhill.echo.util.id

fun main(args: Array<String>) {
    for (arg in args) {
        println(id(arg))
    }
}
```

3.10.9. Configuración del Proyecto EchoApp

```
// echoApp/build.gradle.kts
plugins {
    id("io.gitlab.arturbosch.detekt")
    kotlin("jvm")
    // Declaramos que nuestro programa es una aplicación
    application
}

/* ... */
application {
    mainClass.set("cl.ravenhill.echo.EchoKt")
}
```

En esta configuración, declaramos el punto de entrada de la aplicación. Si la función `main` está definida en el archivo `cl.ravenhill.echo.Echo.kt`, entonces el punto de entrada es `cl.ravenhill.echo.EchoKt`.

3.10.10. Ejecución de la Aplicación

```
.\gradlew.bat run
```

```
./gradlew run
```

Esta configuración es útil para probar la aplicación, pero no es ideal para "publicarla". Nos gustaría que nuestro "cliente" pueda ejecutar la aplicación sin necesidad de Gradle.

Aquí tienes la versión mejorada y desarrollada usando la sintaxis de Typst:

3.10.11. Creación de un JAR Ejecutable

Podemos crear JAR ejecutables con Gradle. Un fat JAR (también conocido como uber JAR) es un archivo JAR que contiene no solo las clases y recursos de tu propia aplicación, sino también todas las dependencias necesarias para ejecutarla.

```
tasks.register<Jar>("fatJar") { // 1
    archiveClassifier.set("fat")  // 2
    duplicatesStrategy = DuplicatesStrategy.EXCLUDE // 3
    manifest {  // 4
        attributes["Main-Class"] = application.mainClass.get()
        attributes["Implementation-Title"] = "Echo"
        attributes["Implementation-Version"] = "1.0"
    }
    from(sourceSets.main.get().output)  // 5
    dependsOn(configurations.runtimeClasspath)  // 6
    from({  // 7
        configurations.runtimeClasspath
            .get()
            .filter { it.name.endsWith("jar") }
            .map { project.zipTree(it) }
    })
}
```

1. Registramos una tarea `fatJar` para empaquetar la aplicación.
2. Configura el nombre del archivo JAR resultante con el sufijo "fat".
3. Establece la estrategia de manejo de duplicados a `EXCLUDE` (excluir archivos duplicados).
4. Configura el manifiesto del JAR.
5. Añade el código fuente de la aplicación al JAR.
6. Establece que esta tarea depende de las clases en el classpath de tiempo de ejecución.
7. Añade las dependencias del classpath de tiempo de ejecución al JAR.

Con la tarea creada podemos compilar el JAR:

```
# Windows
.\gradlew.bat fatJar
```

```
# Unix
./gradlew fatJar
```

Para ejecutar un JAR necesitamos Java:

```
java -jar echoApp/build/libs/echoApp-1.0.0-fat.jar Hello world
```

3.11. Publicación de bibliotecas

La publicación de bibliotecas es el proceso de distribuir un conjunto de funcionalidades empaquetadas en un archivo para que otrxs desarrolladorxs puedan utilizarlas en sus proyectos. Este proceso facilita la reutilización de código, promueve la colaboración y fomenta el desarrollo de software modular.

3.11.1. Beneficios

- **No reinventar la rueda**:
  - Permite a los desarrolladores reutilizar funcionalidades ya implementadas, reduciendo el tiempo y esfuerzo necesarios para desarrollar nuevas aplicaciones.
- **Mejora de la calidad del software**:
  - Utilizar bibliotecas probadas y mantenidas ayuda a mejorar la calidad del software, ya que se basan en soluciones robustas y bien documentadas.

- **Facilitación de actualizaciones y mejoras centralizadas**:
  - Facilita la implementación de actualizaciones y mejoras de manera centralizada, asegurando que todos los proyectos que utilizan la biblioteca se beneficien de las mejoras y correcciones.

3.11.2. Proceso de Publicación

1. **Compilación de la biblioteca**:
   - Asegurarse de que el código fuente de la biblioteca se compila correctamente sin errores.
2. **Generación de artefactos**:
   - **JAR** en el caso de Java/Kotlin/Scala.
   - **Wheels** para Python.
   - **Static Library (lib)**, **Shared Library (dll/so/dylib)** en C++.
   - **RubyGems** con Ruby.
3. **Subida a un repositorio o plataforma de distribución**:
   - Por ejemplo, GitHub Releases, GitHub Packages, Maven Central.

3.11.3. Documentación

Una biblioteca sin una documentación clara es "desechable". Una buena documentación facilita el entendimiento y uso de la biblioteca por otrxs desarrolladorxs, ahorra tiempo y recursos al reducir la necesidad de soporte y consultas frecuentes, y mejora la adopción de la biblioteca, ya que los desarrolladores prefieren usar herramientas bien documentadas.

3.11.3.1. Documentación inadecuada

- Lxs usuarixs pueden encontrar difícil entender cómo utilizar la biblioteca correctamente.
- Incrementa los errores y malentendidos, lo que lleva a una percepción negativa de la biblioteca.
- Reducción en la tasa de adopción, ya que lxs desarrolladorxs buscarán alternativas mejor documentadas.

3.11.3.2. Buena documentación

- Proporciona una visión general de cómo instalar y comenzar a usar la biblioteca.
- Incluye ejemplos de código que muestran cómo utilizar las funciones y características principales.
- Documenta todas las funciones, clases y métodos disponibles en la biblioteca con descripciones detalladas.
- Explica cómo la biblioteca puede resolver problemas específicos con ejemplos concretos.
- Aborda preguntas comunes y problemas que los usuarios pueden encontrar.

3.11.3.3. Buenas prácticas

- Mantén un estilo y formato consistente en toda la documentación.
- Asegúrate de que la documentación se mantenga actualizada con las nuevas versiones de la biblioteca.
- Evita el uso de jerga técnica innecesaria y sé claro y directo en las explicaciones.
- Recoge y actúa sobre el feedback de los usuarios para mejorar la documentación.

3.11.3.4. Herramientas

Herramientas como Javadoc (Java), KDoc y Dokka (Kotlin), Sphinx (Python), entre otros, pueden ayudar a generar documentación a partir del código fuente. Publica la documentación en plataformas accesibles como GitHub Pages, Read the Docs, o sitios web dedicados.

3.11.3.5. Documentación eficaz

- Cómo instalar la biblioteca y un ejemplo simple de uso.
- Ejemplos prácticos de diferentes funcionalidades de la biblioteca.
- Detalles de todas las funciones, métodos y clases disponibles, con ejemplos de uso.

- Respuestas a preguntas comunes y soluciones a problemas típicos.

3.11.4. Configuración del proyecto

Utilizaremos la librería Echo de la sección anterior. Publicaremos la biblioteca como un *release* en GitHub y como una dependencia en GitHub Packages.

Lo primero que necesitamos es convertir nuestra librería en un proyecto de git. Luego, debemos crear un repositorio en GitHub para nuestro proyecto. NO haremos *push* todavía, ya que necesitamos configurar algunas cosas antes de hacerlo.

1. **Inicializar un Repositorio Git**:
   - Abre tu terminal y navega hasta el directorio de tu proyecto.
   - Ejecuta `git init` para inicializar un nuevo repositorio git.
2. **Crear un Repositorio en GitHub**:
   - Ve a GitHub y crea un nuevo repositorio.
   - Sigue las instrucciones para crear un nuevo repositorio.
   - Asegúrate de no inicializarlo con un README, ya que ya tienes archivos en tu proyecto local.

Aquí tienes la versión mejorada y desarrollada usando la sintaxis de Typst:

3.11.4.1. Dokka

Dokka es una herramienta de generación de documentación para proyectos Kotlin. Similar a Javadoc para Java, Dokka convierte comentarios de documentación en código Kotlin en HTML, Markdown y otros formatos de salida.

```
# gradle.properties
# ...
dokka.version=1.9.20
```

```
// settings.gradle.kts
/* ... */
pluginManagement {
    /* ... */
    plugins {
        /* ... */
        id("org.jetbrains.dokka") version extra["dokka.version"] as String
    }
}
```

```
// echoLib/build.gradle.kts
import org.jetbrains.dokka.gradle.DokkaTask

plugins {
    /* ... */
    id("org.jetbrains.dokka")
}

val dokkaVersion = extra["dokka.version"] as String

dependencies {
    dokkaHtmlPlugin("org.jetbrains.dokka:kotlin-as-java-plugin:$dokkaVersion")
}
```

```
val dokkaHtml by tasks.getting(DokkaTask::class)
/* ... */
```

```
/**
 * Returns the input string as it is.
 *
 * This function takes a string as input and returns the same string without any
 * modifications.
 *
 * ## Usage:
 * ```
 * val text = "Hello, world!"
 * val result = id(text)
 * // result == "Hello, world!"
 * ```
 *
 * @param text The input string to be returned.
 * @return The same input string.
 */
fun id(text: String) = text
```

Para generar la documentación, ejecuta el siguiente comando:

```
# Windows
.\gradlew.bat :echoLib:dokkaHtml
```

```
# Unix
./gradlew :echoLib:dokkaHtml
```

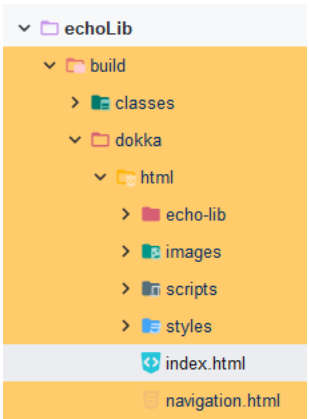

Figure 1: Documentación compilada por Dokka, con `index.html` como punto de entrada.

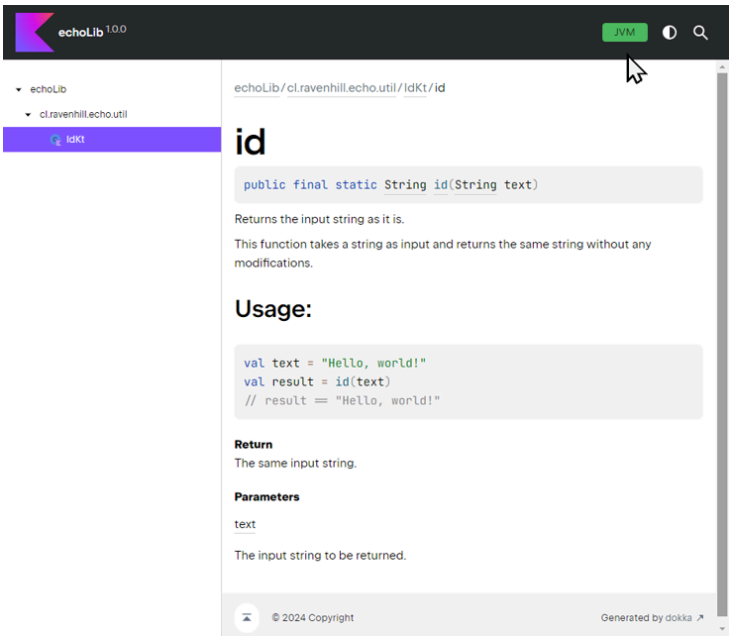

Figure 2: Documentación de la función `id` generada por Dokka.

3.11.5. Publicando un *Release* en GitHub

La primera forma que veremos de publicar una biblioteca es hacer un release en GitHub Para esto "simularemos" el proceso de publicación localmente y publicaremos el release "a mano"

```
// echoLib/build.gradle.kts
/* ... */
```

val dokkaJar by tasks.registering(Jar::class archiveClassifier.set("javado ") from(dokkaHtml.outputDirecto y

sourcesJar by tasks.registering(Jar::class { dependsOn("ja " asks.register("releaseLocal" {

```
# Windows
.\gradlew.bat :echoLib:releaseLocal
```

```
# Unix
./gradlew :echoLib:releaseLocal
```

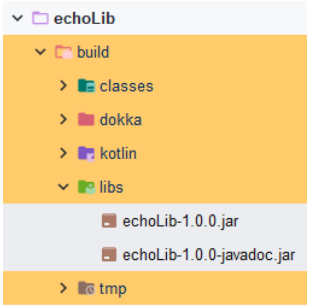

Figure 3: Archivos generados por la tarea `releaseLocal`.

Antes de publicar nuestro proyecto debiéramos crear un archivo .gitignore Una página útil para esto es https://www.toptal.com/developers/gitignore/ Luego podemos hacer commit y push al proyecto

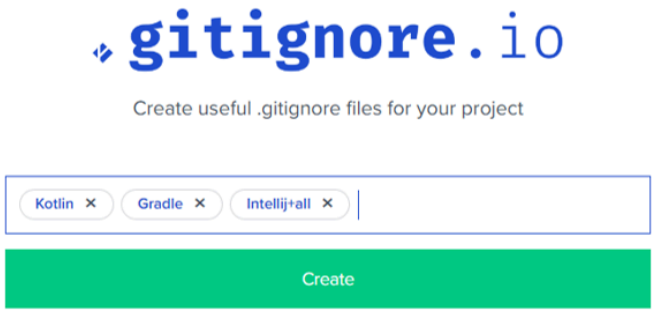

Figure 4: Sitio web para generar archivos `.gitignore` personalizados.

Creamos un nuevo release en GitHub:

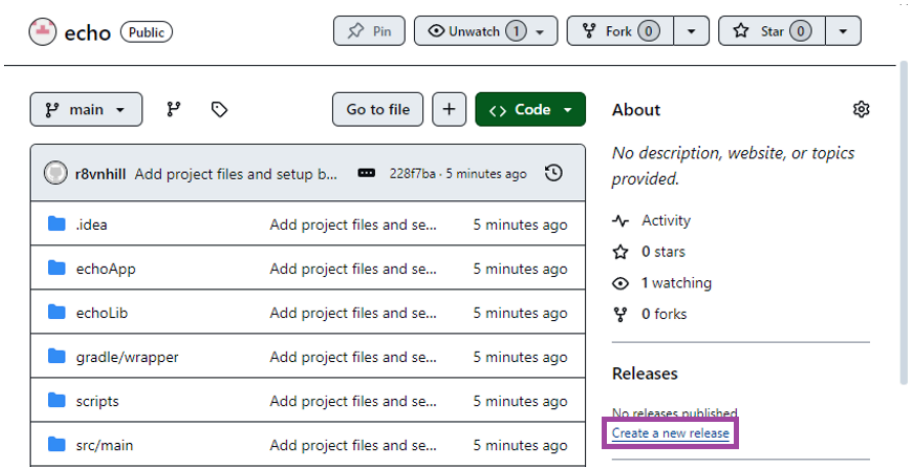

Figure 5: Creación de un nuevo release en GitHub.

./gradlew :echoLib:releaseLocal dependsOn(“dokkaJar”, “sourcesJa “) }

3.12. Respuestas

Ejercicio: Orden de Ejecución de Acciones

```
First Action - Preparation Step 2
First Action - Preparation Step 1
Last Action - Cleanup Step 1
Last Action - Cleanup Step 2
```

Ejercicio: Tamaño del Proyecto

```
tasks.register("countCompiledSize") {
    group = "build"
    description = "Count the size of the compiled classes"
    dependsOn("compileKotlin")
    doLast {
        val files = fileTree("build/classes/kotlin/main")
        var size = 0L
        for (file in files) {
            size += file.length()
        }
        println(
            "The size of the compiled classes in subproject1 is $size bytes"
```

```
        )
    }
}
```

Ejercicio: Tarea de Fibonacci

```
Preparing to calculate the 10th Fibonacci number
The 10th Fibonacci number is 55
Finished calculating the 10th Fibonacci number
```

Ejercicio: Reporte de Archivos

```
class KotlinFilesReportPlugin : Plugin<Project> {
    override fun apply(project: Project) {
        project.tasks.register("generateKotlinFilesReport") {
            group = "reporting"
            description = "Generates a report of all Kotlin files in the project."

            doLast {
                val kotlinFiles = project.fileTree(".").matching {
                    include("**/*.kt", "**/*.kts")
                }.files

                println("Kotlin Files Report:\n\n")
                for (file in kotlinFiles) {
                        println(file.name)
                }
            }
        }
    }
}

apply<KotlinFilesReportPlugin>()
```

Bibliografías

[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.