Programación Funcional en Java

Círculo Siete Capacitación

Clase 8 de 12

12 Marzo 2025

Operaciones con Option y Either en Vavr para Evitar null y Manejar Estados

- En Java, ***null*** es una de las principales fuentes de errores (***NullPointerException***). Para evitarlo, Vavr proporciona dos estructuras clave:
  - ***Option<T>*** → Representa la posible ausencia de un valor.
  - ***Either<L, R>*** → Representa dos posibles estados (Left para errores, Right para éxito).
- Ambas estructuras permiten evitar ***null*** y manejar estados de forma funcional sin ***if-else*** ni ***try-catch*** innecesarios.

1. Option<T>: Alternativa Segura a null

- ***Option<T>*** representa un valor opcional que puede ser:
  - ***Some<T>*** → Contiene un valor válido.
  - ***None*** → Representa la ausencia de valor (***null***).
- Ver Lab01.java

2. Operaciones con Option<T>

- ***map()*** → Transformar el valor si existe
- ***flatMap()*** → Evita ***Option<Option<T>>*** (anidación)
- ***filter()*** → Filtra valores dentro de ***Option<T>***
- ***orElse()*** → Valor por defecto si es ***None***
- Ventaja: Evita validaciones manuales y código innecesario.
- Ver Lab02.java

3. Either<L, R>: Manejo de Estados Sin Excepciones

- ***Either<L, R>*** representa dos posibles estados:
  - ***Left<L>*** → Contiene un error o estado fallido.
  - ***Right<R>*** → Contiene el resultado exitoso.
- Ver Lab03.java

4. Operaciones con Either<L, R>

- ***map()*** → Transformar el valor si es ***Right<R>***
- ***mapLeft()*** → Transformar el valor si es ***Left<R>***
- ***flatMap()*** → Encadenar operaciones sin anidar ***Either<Either<T>>***
- ***getOrElse()*** → Valor por defecto si es ***Left***
- ***fold()*** → Manejo centralizado de ambos casos (***Left*** y ***Right***)
- Ver Lab04.java

5. Comparación: Option<T> vs Either<L, R>

- ¿Cuándo usar Option<T>?
  - Cuando un valor puede estar presente o ausente (Some o None).
  - Para evitar null en métodos que pueden devolver un valor o nada.
- ¿Cuándo usar Either<L, R>?
  - Para representar errores sin lanzar excepciones (Left para errores, Right para éxito).
  - Cuando se necesita manejar estados de éxito o error en un solo flujo.

6. Uso Combinado de Option<T> y Either<L, R>

- Ver Lab05.java

Notas finales

- ***Option<T>*** y ***Either<L, R>*** en ***Vavr*** mejoran la seguridad y claridad del código en Java, evitando ***null*** y excepciones innecesarias.
- Resumen:
  - ***Option<T>*** → Evita ***null***, pero no maneja errores.
  - ***Either<L, R>*** → Maneja errores y evita excepciones.
  - Encadenar operaciones con ***map()***, ***flatMap()*** y ***fold()*** hace el código más limpio.
- Adoptar estas estructuras permite escribir código más seguro, funcional y expresivo en Java

Y si....

- Ver Lab06.java

Expresiones Lambda para facilitar implementar el patron Decorator

- El patrón Decorator (o Decorador) consiste en envolver un objeto o función con “capas” adicionales de comportamiento sin modificar su implementación original.
- En lenguajes que soportan funciones como ciudadanos de primera clase (por ejemplo, Python) y/o expresiones lambda (por ejemplo, Java 8+, C#), podemos implementar el patrón de manera muy concisa y funcional.

Importante

- Las expresiones lambda permiten implementar el patrón Decorator de forma funcional y concisa, enfocada a la composición de funciones.
- Cada “capa” de decoración es simplemente una función que toma el resultado de la anterior y lo transforma.
- En lenguajes como Python, la idea de “decorar” funciones está muy integrada a nivel de lenguaje. Las lambdas hacen que el código sea más compacto.
- En Java (y otros lenguajes con soporte funcional), la combinación de lambdas + interfaces funcionales provee un estilo de programación muy amigable a la hora de ir añadiendo comportamientos de manera incremental.
- Si se busca un enfoque 100% orientado a objetos, también se pueden crear distintas clases decoradoras que implementen la misma interfaz que el objeto original. Sin embargo, para funciones puras o transformaciones, el uso de lambdas es un atajo sumamente práctico y limpio.

Patrones de diseño que se pueden implementar con FP

- En Java (a partir de Java 8 en adelante) existe un ecosistema de interfaces funcionales (Function, Consumer, Supplier, Predicate, etc.) y la posibilidad de usar expresiones lambda y métodos por referencia.
- Esto hace que varios patrones de diseño se puedan implementar o simplificar con un enfoque más “funcional”.

Strategy

- El patrón Strategy consiste en encapsular comportamientos (estrategias) que se puedan intercambiar en tiempo de ejecución.
- Con lambdas, en lugar de crear clases concretas que implementen una interfaz, podemos usar directamente funciones anónimas.
- Ver Lab07.java

Command

- El patrón Command encapsula una acción o “comando” para ser ejecutado posteriormente o bajo ciertas condiciones.
- En un enfoque imperativo, se crean clases que implementan una interfaz ***Command*** con un método ***execute()***.
- En lugar de eso, podemos usar un ***Runnable*** o un ***Consumer<T>*** o cualquier otra interfaz funcional.
- Ver Lab08.java

Chain of Responsibility

- El patrón ***Chain of Responsibility*** conecta varios manejadores (***handlers***) en “cadena”.
- Cada manejador decide si procesa la petición o la delega al siguiente.
- Con funciones, podemos componerlas para que cada una “decida” si continúa o no.
- Ver Lab09.java

Template Method

- En el patrón Template Method, definimos el esqueleto de un algoritmo en un método “template”, delegando algunos pasos a subclases.
- Con programación funcional, podemos pasar lambdas como “ganchos” (hooks) en lugar de extender clases.
- Ver Lab10.java