



Programación Funcional en Java

Círculo Siete Capacitación


Clase 7 de 12
10 Marzo 2025

¿Qué es una Mónada en Programación Funcional?


- En programación funcional, una mónada (monad) es una estructura que envuelve un valor y proporciona un mecanismo para encadenar operaciones de manera segura y componible.
- Definición:
- Una mónada es un tipo de dato que implementa tres reglas fundamentales:
 - Debe tener un constructor (unit/pure) para encapsular un valor.
 - Debe permitir encadenar operaciones (*flatMap*).
 - Debe seguir las leyes monádicas (identidad izquierda, derecha y asociatividad).
- Imagina una mónada como una caja que envuelve un valor y te permite operar sobre él sin abrir la caja directamente.

¿Por qué usar Mónadas?

- Evitan null y errores inesperados (***Optional***, ***Either***). Pero no solo eso.
- Encadenan operaciones sin necesidad de estructuras de control (***flatMap***).
- Hacen que el código sea más expresivo y funcional.



```
record Cliente(String nombre) {  
}  
  
public class Main {  
  
    public String obtenerNombreCliente(int id) {  
        Cliente cliente = buscarClienteEnDB(id);  
        if (cliente != null) {  
            return cliente.nombre();  
        } else {  
            return "Cliente no encontrado";  
        }  
    }  
  
    public Cliente buscarClienteEnDB(int id) {  
        // bsucar de alguna forma...  
        return null;  
    }  
}
```



```
import java.util.Optional;

record Cliente(String nombre) {
}

public class Main {

    public String obtenerNombreCliente(int id) {
        return buscarClienteEnDB(id)
            .map(Cliente::nombre)
            .orElse("Cliente no encontrado");
    }

    public Optional<Cliente> buscarClienteEnDB(int id) {
        // bsucar de alguna forma...
        return Optional.empty();
    }
}
```

Diferencia entre `map()` y `flatMap()` en Mónadas

- ***map()*** aplica una función y devuelve una mónada dentro de otra mónada.
- ***flatMap()*** aplana el resultado para evitar anidaciones.
- Revisar Lab01.java



- <http://www.flatmapthatshit.com/>

¿Qué es *flatMap* en Java y Vavr?

- ***flatMap*** es un método en programación funcional que aplanar estructuras de datos anidadas para evitar que se generen capas innecesarias de envolturas como ***Optional<Optional<T>>***, ***Try<Try<T>>***, ***Either<Either<L, R>>***, etc.

Diferencia clave entre map y flatMap

- ***map()*** → Transforma valores sin aplanar la estructura.
- ***flatMap()*** → Aplana la estructura al devolver directamente la transformación.



```
package com.circulosiete.curso.funcional.clase07;

import io.vavr.control.Try;

public class Lab02 {
    public static void main(String[] args) {
        Lab02 lab02 = new Lab02();
        int resultado = lab02.calcula(10, 0, 2);

        System.out.println("El resultado es: " + resultado);
    }

    private int calcula(int one, int two, int multiplicador) {
        return aplicaFunciones(one, two, multiplicador)
            .getOrElseGet(throwable -> 0);
    }

    private int multiplicar(int multiplicador, int x) {
        return x * multiplicador;
    }

    private int division(int one, int two) {
        return one / two;
    }

    private Try<Integer> aplicaFunciones(int one, int two, int multiplicador) {
        return Try.of(() -> division(one, two))
            .onFailure(throwable -> System.out.println(throwable.getMessage()))
            .flatMap(x -> Try.of(() -> multiplicar(multiplicador, x)));
    }
}
```



```
package com.circulosiete.curso.funcional.clase07;

import io.vavr.control.Either;

public class Lab03 {
    public static void main(String[] args) {
        final var resultado = Either.right(10)
            .map(x -> Either.right(x * 2));
        System.out.println(resultado); // Right(Right(20))
        // notan algo raro en el tipo de la variable resultado?
    }
}
```

¿Cuándo usar flatMap() en Java y Vavr?

- Usar **flatMap()** cuando:
 - Se trabaja con estructuras anidadas (**Optional<Optional<T>>**, **Either<Either<L, R>>**).
 - Se encadenan operaciones que devuelven estructuras funcionales (**Try<T>**, **Either<L, R>**, **List<T>**).
 - Se necesita mantener el código limpio y evitar verificaciones manuales.
- Usar **map()** cuando:
 - Se transforma un valor dentro de una estructura sin cambiar el tipo (**Optional<T>** → **Optional<R>**).
 - No se necesita aplanar estructuras anidadas.

No olvidar

- ***flatMap()*** en Java y Vavr permite evitar estructuras anidadas innecesarias y hace que el código sea más expresivo y funcional.
- ***flatMap()*** es útil en ***Optional***, ***Try***, ***Either*** y ***List*** para evitar anidación.
- Se usa en operaciones encadenadas para transformar estructuras funcionales.
- Hace que el código sea más limpio, legible y declarativo.

Manejo Avanzado de Errores con Try en Vavr

- El manejo tradicional de errores en Java con ***try-catch*** puede hacer que el código sea verboso y difícil de mantener.
- Vavr proporciona la clase ***Try<T>***, que permite manejar excepciones de manera más funcional y declarativa.

¿Qué es $\text{Try}<T>$ en Vavr?

- $\text{Try}<T>$ es una monada que encapsula operaciones que pueden fallar, eliminando la necesidad de try-catch explícitos.
- Estados de $\text{Try}<T>$:
 - $\text{Success}<T>$ → Contiene el resultado exitoso.
 - $\text{Failure}<T>$ → Contiene la excepción capturada.

```
package com.circulosiete.curso.funcional.clase07;

import io.vavr.control.Try;

import java.nio.file.Files;
import java.nio.file.Paths;

public class Lab05 {
    public static void main(String[] args) {
        final var contenido = Try
            .of(() -> new String(
                Files.readAllBytes(Paths.get("archivo.txt"))
            ))
        );

        System.out.println(contenido.getOrElse("Archivo no encontrado"));
    }
}
```


Manejo de Errores con `recover()` y `recoverWith()`

- ***recover()*** → Retorna un valor alternativo si hay un fallo.
- ***recoverWith()*** → Retorna otro ***Try<T>*** en caso de fallo.
- Ventaja: Permite proporcionar estrategias de recuperación.
- Ver Lab06.java

onFailure() y onSuccess() para Registrar Eventos

- Ejecutar una acción en caso de éxito o error
- Ventaja: Se pueden registrar logs o métricas sin afectar el flujo.
- Ver Lab07.java

toEither() para Integración con Either<L, R>

- Convertir Try<T> en Either<L, R> para un manejo más estructurado
- Ver Lab08.java

filter() para Validaciones en Try<T>

- Ver Lab09.java

Comparación: Try<T> vs try-catch Tradicional

- Cuándo usar ***Try<T>*** en lugar de ***try-catch***:
 - Cuando se necesita encadenar operaciones con errores.
 - Cuando se quiere evitar interrupciones en el flujo del código.
 - Cuando se requiere un manejo más funcional y declarativo

Comparación

Característica	Try<T> de Vavr	try-catch Tradicional
Verboso	No	Sí
Encadenable	Sí (map(), flatMap())	No
Seguridad	Sin excepciones no controladas	Puede lanzar excepciones
Expresividad	Más declarativo	Más imperativo

Notas finales

- ***Try<T>*** permite manejar errores en Java de manera funcional, eliminando try-catch anidados y mejorando la composición de funciones.
- Ventajas de ***Try<T>*** en Vavr:
 - Elimina try-catch verbosos.
 - Encadena operaciones con ***map()*** y ***flatMap()***.
 - Manejo de errores más expresivo (***recover()***, ***toEither()***).
 - Evita excepciones no controladas y mantiene el flujo.