Programación Funcional en Java

Círculo Siete Capacitación

Clase 5 de 12

5 Marzo 2025

Tipos comunes en Programación Funcional

- Maybe / Option
- Either
- Try / Result
- Validation
- List (o estructuras de datos inmutables)
- NonEmptyList / NonEmpty Sequences
- Monadas de efectos (IO, Task, etc.)
- Reader / Writer / State
- Free / Free Monad
- Futuro / Future (u otras abstracciones de concurrencia asíncrona)

Maybe / Option

- Descripción: Representa la posibilidad de que un valor exista o no.
- Función principal: Evita el uso de valores nulos y los “null pointer exceptions” asociados.
- Ejemplo de uso: En Haskell es Maybe, en Scala y otros lenguajes se llama Option u Optional.
  - Generalmente, se define como un tipo con dos constructores:
    - Some(x) / Just x (valor presente)
    - None / Nothing (valor ausente)
- Ventaja: Obliga a tratar explícitamente el caso en el que un valor puede faltar, lo que mejora la seguridad y legibilidad.

Either

- Descripción: Se suele usar para representar un resultado que puede ser de dos tipos: éxito o error.
- Función principal: Manejo de fallos (o “ramas”) en el flujo normal del programa sin recurrir a excepciones.
- Ejemplo de uso:
  - Left(errorValue) indica la rama de error.
  - Right(correctValue) indica la rama de éxito.
- Ventaja: Evita la mezcla de control de flujo con excepciones y promueve el tratamiento explícito de errores.

Try / Result

- Descripción: Muy parecido a Either, pero pensado especialmente para capturar y propagar excepciones.
- Función principal: Capturar fallos en operaciones “peligrosas” (cálculos que podrían lanzar excepciones) de forma funcional.
- Ejemplo de uso:
  - En Scala se llama Try (con constructores Success y Failure).
  - En Rust se utiliza Result, con Ok y Err.
- Ventaja: Permite componer operaciones que pueden fallar sin detener el programa ni usar excepciones imperativas.

Validation

- Descripción: Similar a Either, pero diseñado para acumular múltiples errores en lugar de interrumpir en el primero que aparezca.
- Función principal: Recopilación de todos los fallos en procesos como la validación de formularios, análisis de datos, etc.
- Ventaja: A diferencia de Either, donde un Left detiene el flujo, Validation suele acumular todos los errores, ofreciendo más información al usuario.

List (o estructuras de datos inmutables)

- Descripción: Estructura de lista enlazada inmutable muy típica en lenguajes funcionales (Haskell, por ejemplo).
- Función principal: Representa colecciones inmutables que se pueden recorrer de forma recursiva o con funciones de orden superior (map, filter, etc.).
- Ejemplo de uso:
  - En Haskell, las listas se denotan con [].
  - Muchas librerías FP (en Scala, F#, etc.) proveen listas inmutables como List<T> o similares.
- Ventaja: Su inmutabilidad facilita la concurrencia y la programación declarativa.

NonEmptyList / NonEmpty Sequences

- Descripción: Variantes de las listas inmutables que siempre tienen al menos un elemento.
- Función principal: Garantiza que no tendrás casos vacíos, eliminando la necesidad de manejar “lista vacía” como un caso especial.
- Ventaja: Simplifica la lógica cuando se asume que la colección no puede estar vacía (por ejemplo, en validaciones o casos de cálculo estadístico mínimo).

Monadas de efectos (IO, Task, etc.)

- Descripción: Encapsulan efectos secundarios (entrada/salida, acceso a red, lectura de archivos, etc.) de modo que la función que los usa sigue siendo pura desde la perspectiva del lenguaje.
- Función principal: Permite mantener la transparencia referencial, retrasando la ejecución de efectos hasta un momento controlado.
- Ejemplo de uso:
  - En Haskell existe IO.
  - En bibliotecas de Scala (ZIO, Cats Effect) encontramos IO, Task, etc.
- Ventaja: Mantiene el corazón del programa libre de efectos no predecibles y facilita el testing.

Reader / Writer / State

- Descripción: Monadas o “efectos” para manejar contexto, logging y estado de manera pura.
- Reader: Inyecta de forma funcional un contexto (configuración, dependencias…) que las funciones pueden “leer”.
- Writer: Acumula un log o un historial de manera inmutable mientras se hacen cálculos.
- State: Modela y pasa de forma implícita un estado inmutable de una función a otra, evitando variables globales mutables.
- Función principal: Abstraer estos efectos y permitir su composición y testeo sencillo.

Free / Free Monad

- Descripción: Construcción abstracta que permite definir DSLs funcionales (lenguajes internos) y luego “interpretarlos” en diferentes contextos.
- Función principal: Separar la descripción de las operaciones de su ejecución concreta, lo que habilita distintas interpretaciones (log, test, ejecución real, etc.).
- Ventaja: Alta flexibilidad y capacidad de testear o reusar la misma descripción con múltiples implementaciones.

Futuro / Future

(u otras abstracciones de concurrencia asíncrona)

- Descripción: Representan un cálculo que se ejecuta en paralelo o de forma asíncrona y que eventualmente producirá un valor o un error.
- Función principal: Abordar la programación asíncrona y paralela sin perder la capacidad de composición funcional (métodos como map, flatMap, etc.).
- Ventaja: Permite encadenar transformaciones sobre valores que todavía no existen sin romper la inmutabilidad ni la composición pura del resto del programa.

Conclusión

- Estos tipos y patrones son herramientas esenciales para estructurar el código de forma declarativa, componible y segura.
- Cada uno sirve para un problema específico dentro del paradigma funcional, ya sea para evitar nulos, manejar errores, expresar efectos secundarios o tratar la concurrencia.
- Usados de manera adecuada, mejoran la robustez y la legibilidad de las aplicaciones, especialmente a gran escala.

Mónada

- Es un “patrón” o estructura que nos permite encadenar operaciones (funciones) dentro de un contexto controlado.
- Dicho contexto puede ser muy variado: computaciones que pueden fallar, cálculos asíncronos, manejo de estado, lectura de configuraciones, etc.
- Lo importante es que la mónada organiza cómo se aplican y combinan dichas operaciones, de tal manera que se logra mantener el estilo funcional (funciones puras y composición) aunque internamente se trabaje con aspectos como errores, estados mutables, entrada/salida, etc.

Definición formal (simplificada)

- En la mayoría de lenguajes con inspiración de Haskell, podemos decir que una mónada se define a través de:
  - Un constructor de tipo ***M<A>*** (donde A es el tipo “interno”).
  - Una función return (o ***pure***, dependiendo del lenguaje/biblioteca), que inyecta un valor “normal” de tipo A en el contexto ***M<A>***.
  - Una función ***bind*** (a menudo llamada ***>>=*** en Haskell), que toma un ***M<A>*** y una función ***(A -> M<B>)***, y produce un ***M<B>***.

En pseudo-Haskell se ve así:

```haskell
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

- La idea es que return (o ***pure***) convierte un valor simple en el contexto monádico, mientras que ***bind*** describe cómo conectar (o encadenar) una operación monádica con la siguiente.

- Contexto: Las mónadas representan “cálculos en un contexto” (por ejemplo, cálculos que pueden devolver ***Nothing*** si fallan, como en la mónada ***Maybe/Option***).
- Encadenamiento de operaciones: A través de ***bind***, podemos tomar el resultado de una operación monádica y pasarlo a la siguiente operación. La mónada se encarga de gestionar los detalles de ese “transporte” de un cálculo a otro (por ejemplo, si hay un error, si la operación es asíncrona, etc.).
- Leyes monádicas: Para que algo se considere una mónada, además de las funciones ***return*** y ***bind***, debe cumplir tres leyes importantes:
  - Ley de identidad izquierda: ***return x >>= f*** equivale a ***f x***.
  - Ley de identidad derecha: ***m >>= return*** equivale a ***m***.
  - Ley de asociatividad:
    - ***(m >>= f) >>= g*** equivale a ***m >>= (\|x -> f x >>= g)***.
  - Estas leyes aseguran coherencia y consistencia al componer múltiples operaciones.

Ejemplo sencillo con Maybe (o Option)

- Imaginemos que tenemos una operación que puede fallar y devolver ***null***. Para evitar nulos, usamos el tipo ***Maybe*** a:
  - ***Just x*** indica que hay un valor.
  - ***Nothing*** indica que no lo hay (equivale a un "fallo" o resultado vacío).

Ejemplo Maybe en pseudo-Haskell

```haskell
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing  >>= f = Nothing
Just x   >>= f = f x
```

- Si nuestra operación actual ya es ***Nothing***, no hay nada que “pasar” a la siguiente, así que todo sigue siendo ***Nothing***.
- Si es ***Just x***, extraemos ***x*** y lo usamos en la siguiente función.
- En otras palabras, la mónada ***Maybe*** maneja la posibilidad de ausencia de valor, “saltándose” automáticamente el resto de cálculos si algo falla.

¿Por qué son útiles?

- Permiten componer funciones y procesos complejos de forma modular.
- Manejan automáticamente la “lógica de pegado” de contextos (errores, listas, estado, logs, asíncrono…), haciendo que tu código sea más declarativo.
- Separan la lógica principal de los detalles de implementación, como el manejo de errores o efectos secundarios, mejorando la mantenibilidad y testabilidad.

Conclusión

- Una mónada es una abstracción que facilita la composición de cálculos en un contexto determinado, cumpliendo ciertas propiedades que garantizan que todo siga siendo predecible y modular, aún cuando el contexto implique complejidades como errores, asíncrono, efectos secundarios, etc.

Conclusión

- Una mónada es una abstracción que facilita la composición de cálculos en un contexto determinado, cumpliendo ciertas propiedades que garantizan que todo siga siendo predecible y modular, aún cuando el contexto implique complejidades como errores, asíncrono, efectos secundarios, etc.

Tuplas

- Las tuplas son simplemente un tipo de datos compuesto que permite agrupar varios valores (cada uno de un tipo potencialmente distinto) sin necesidad de crear una estructura más compleja o de definir un registro/struct/objeto con nombre.

1. Tuplas como producto de tipos

- En la teoría de tipos, se suele decir que las tuplas representan un producto (“product type”):
  - Una tupla de dos elementos (A, B) contiene un valor de tipo A y otro de tipo B.
  - De manera análoga, una tupla de 3 elementos (A, B, C) es un producto de tres tipos, y así sucesivamente.
- La denominación de “producto” viene de que, si hay m posibilidades para elegir el primer valor y n para el segundo, el total de combinaciones posibles es $m \times n$.

2. Usos en lenguajes funcionales (1/2)

- Retorno múltiple de funciones:
  - En muchos lenguajes funcionales (p. ej. Haskell, ML, F#), si una función necesita regresar más de un valor, simplemente se devuelven como (valor1, valor2, valor3, ...).
  - Esto evita la necesidad de crear un objeto/struct específico para agrupar esos valores, sobre todo en casos simples.
- Patrón de correspondencia (pattern matching):
  - En Haskell o F#, podemos escribir algo como:

```
sumaPares :: (Int, Int) -> Int
sumaPares (x, y) = x + y
```

    - Se desestructura la tupla ***(x, y)*** directamente en los parámetros de la función, algo muy conveniente para evitar accesos como ***fst(first)*** y ***snd(second)*** manualmente.

3. Relación con las mónadas y otras estructuras

- ¿Es la tupla una mónada?
  - No en general. Sin embargo, si fijas uno de los tipos de la tupla, ***(r, _)***, esta estructura se comporta como un ***functor*** e incluso puede definirse como una mónada cuando ***r*** forma un ***monoid***, transformándose en la llamada "Writer monad" (o algo muy parecido).
  - Por ejemplo, ***((,) w)*** es un ***functor*** si ***w*** es un ***monoid***. Esto se aprovecha en Haskell para la ***Writer w monad***, que acumula un log o información extra en ***w***.
- ¿Dónde encaja la tupla en los tipos algebraicos?
  - Las tuplas son un producto (como decíamos antes).
  - Otros tipos como ***Either*** se consideran una suma (o disyunción), porque un valor de ***Either<a,b>*** es o bien un ***Left*** con algo de tipo a o bien un ***Right*** con algo de tipo ***b***, pero no ambas a la vez.

Ventajas e inconvenientes

- Ventajas
  - Simples de usar y de razonar.
  - Permiten agrupar datos de manera rápida sin generar tipos nuevos.
  - Fáciles de desestructurar con pattern matching.
- Inconvenientes
  - Para casos más grandes (por ejemplo, agrupar 7 u 8 campos), las tuplas se vuelven poco legibles. A veces conviene un tipo con nombre de campos.
  - La semántica de cada elemento puede no ser clara: ***(String, Int)*** no dice tanto como un registro con nombre, por ejemplo:
    - ***{ nombre :: String, edad :: Int }***.

Resumen

- En FP, las tuplas son fundamentales para agrupar datos sin crear tipos específicos.
- Son la forma más sencilla de retornar múltiples valores y la base de los tipos de producto.
- Se integran perfectamente con el pattern matching, facilitando la desestructuración y la claridad de código.
- Aunque no son una mónada general, la tupla de forma parcial (cuando el primer elemento es un ***monoid***) se utiliza para construir estructuras monádicas de logging o acumulación (Writer).

Functor y Monoid

- **Functor**: Permite mapear una función sobre los valores dentro de un contexto/estructura sin romper la estructura.
- **Monoid**: Define una operación binaria asociativa con un elemento neutro, útil para combinar o “reducir” múltiples valores en uno solo.


VAVR.IO

vavr

- Vavr es una librería funcional para Java que amplía el lenguaje con estructuras inmutables, control de errores funcional, y abstracciones típicas de lenguajes funcionales como Scala o Haskell.
- Características principales de Vavr:
  - Colecciones inmutables (List, Set, Map).
  - Manejo funcional de errores (Validation, Try, Either).
  - Evita null con Option.
  - Soporte para Tuple, Lazy, Future y patrones funcionales.

vavr

```java
import io.vavr.control.Option;

public class Main {
    public static void main(String[] args) {
        Option<String> nombre = Option.of(null);
        System.out.println(nombre.getOrElse("Valor por defecto")); // "Valor por defecto"
    }
}
```

Instalación de Vavr

```xml
<dependency>
    <groupId>io.vavr</groupId>
    <artifactId>vavr</artifactId>
    <version>0.10.6</version>
</dependency>
```

```
dependencies {
    implementation 'io.vavr:vavr:0.10.6'
}
```

```java
package com.circulosiete.curso.funcional.clase05;

import io.vavr.collection.List;

public class Lab01 {
    public static void main(String[] args) {
        List<String> lista = List.of("A", "B", "C");
        List<String> nuevaLista = lista.append("D");

        System.out.println("vavr List: " + nuevaLista); // [A, B, C, D]

        // interoperabilidad con Java Standard Library
        // Convertir del tipo de vavr a SDK
        final var javaList = nuevaLista.toJavaList();
        System.out.println("Java List: " + nuevaLista); // [A, B, C, D]

        // Crear una List de vavr a partir de una List de SDK
        final var vavrList = List.ofAll(javaList);
        System.out.println("vavr List: " + vavrList); // [A, B, C, D]
    }
}
```

```java
package com.circulosiete.curso.funcional.clase05;

import io.vavr.collection.HashMap;

/**
 * HashMaps en vavr, evitan errores en concurrencia sin necesidad de synchronized.
 */
public class Lab02 {
    public static void main(String[] args) {
        HashMap<String, Integer> edades = HashMap.of("Juan", 30, "Ana", 25);
        HashMap<String, Integer> nuevasEdades = edades.put("Luis", 40);

        System.out.println(nuevasEdades); // {Juan=30, Ana=25, Luis=40}
    }
}
```

```java
package com.circulosiete.curso.funcional.clase05;

import io.vavr.control.Option;

public class Lab03 {
    public static void main(String[] args) {
        Option<String> nombre = Option.of(null);
        System.out.println(nombre.getOrElse("Valor por defecto")); // "Valor por defecto"

        //Encadenar Option<T> con map()
        Option<String> nombre2 = Option.of("Juan");
        Option<Integer> longitud = nombre2.map(String::length);
        System.out.println(longitud.getOrElse(0)); // 4
    }
}
```

```java
package com.circulosiete.curso.funcional.clase05;

import io.vavr.control.Try;

public class Lab04 {
    public static void main(String[] args) {
        Try<Integer> resultado = Try.of(() -> 10 / 0)
                .onFailure(e -> System.out.println("Error manejado"));

        System.out.println(resultado.getOrElse(-1)); // -1

        Try<Integer> resultado2 = Try.of(() -> Integer.parseInt("123"))
                .map(n -> n * 2)
                .onFailure(e -> {
                    e.printStackTrace();
                    System.out.println("Error manejado.");
                });
        System.out.println(resultado2.getOrElse(-1)); // 246
    }
}
```

```java
package com.circulosiete.curso.funcional.clase05;

import io.vavr.control.Either;

public class Lab05 {
    public static Either<String, Integer> dividir(int a, int b) {
        return b == 0 ? Either.left("No se puede dividir por cero") : Either.right(a / b);
    }

    public static void main(String[] args) {
        final var resultado = dividir(10, 0);
        System.out.println(resultado.isLeft() ? resultado.getLeft() : resultado.get());
    }
}
```

```java
package com.circulosiete.curso.funcional.clase05;

import io.vavr.control.Either;
import io.vavr.control.Try;

import java.util.function.Function;

public class Lab06 {
    public static Either<String, Integer> dividir(int a, int b) {
        return Try.of(() -> a / b)
                .toEither()
                .mapLeft(throwable -> "No se puede dividir por cero");
    }

    public static void main(String[] args) {
        dividir(10, 0)
                .peek(System.out::println)
                .peekLeft(System.out::println);

        final var resultado = dividir(10, 0)
                .fold(Function.identity(), Object::toString);

        System.out.printf("El resultado es: '%s'%n", resultado);
    }
}
```