Programación Funcional en Java

Círculo Siete Capacitación

Clase 6 de 12

8 Marzo 2025


VAVR.IO

vavr

- **Colecciones inmutables**: Vavr ofrece una variedad de colecciones inmutables como ***List, Set, Map***, etc., que son más seguras en entornos concurrentes y siguen los principios de la programación funcional.
- **Tuplas**: Vavr proporciona tuplas de diferentes tamaños (***Tuple2, Tuple3***, etc.) que permiten agrupar múltiples valores de manera segura y eficiente.
- **Funciones**: Vavr permite definir funciones de manera más flexible, incluyendo la composición de funciones, ***currying***, y la aplicación parcial de funciones.
- **Option y Try**: Vavr incluye tipos como ***Option*** (similar a ***Optional*** de Java) y ***Try*** para manejar valores opcionales y operaciones que pueden fallar, respectivamente.
- **Pattern Matching**: Vavr introduce un sistema de ***pattern matching*** que permite manejar diferentes casos de manera más clara y concisa.
- **Lazy Evaluation**: Vavr soporta evaluación perezosa, lo que permite retrasar la evaluación de expresiones hasta que sea necesario.

```java
private static void lab01VavrWay() {
    System.out.println("Vavr");
    // Crear una lista inmutable
    final var numbers = io.vavr.collection.List.of(1, 2, 3, 4, 5);

    // Definir una función que suma 10 a un número
    io.vavr.Function1<Integer, Integer> addTen = x -> x + 10;

    // Aplicar la función a cada elemento de la lista
    final var incrementedNumbers = numbers.map(addTen);

    // Imprimir la lista resultante
    System.out.println(incrementedNumbers); // [11, 12, 13, 14, 15]
}
```

```java
private static void lab01JavaWay() {
    System.out.println("Java");
    // Crear una lista inmutable
    final var numbers = java.util.List.of(1, 2, 3, 4, 5);

    // Definir una función que suma 10 a un número
    java.util.function.Function<Integer, Integer> addTen = x -> x + 10;

    // Aplicar la función a cada elemento de la lista
    final var incrementedNumbers = numbers
            .stream()
            .map(addTen)
            .toList();

    // Imprimir la lista resultante
    System.out.println(incrementedNumbers); // [11, 12, 13, 14, 15]
}
```

```java
package com.circulosiete.curso.funcional.clase06;

import io.vavr.collection.List;

public class Lab02 {
    public static void main(String[] args) {
        // Crear una lista inmutable
        final var numbers = List.of(1, 2, 3, 4, 5);

        // Multiplicar cada elemento por 2
        final var doubled = numbers.map(x -> x * 2);
        // Filtrar elementos mayores a 3
        final var filtered = numbers.filter(x -> x > 3);
        // Sumar todos los elementos
        final var sum = numbers.reduce(Integer::sum);

        System.out.println("Original: " + numbers); // [1, 2, 3, 4, 5]
        System.out.println("Doubled: " + doubled); // [2, 4, 6, 8, 10]
        System.out.println("Filtered: " + filtered); // [4, 5]
        System.out.println("Sum: " + sum); // 15
    }
}
```

```java
package com.circulosiete.curso.funcional.clase06;

import io.vavr.collection.HashSet;
import io.vavr.collection.Set;

public class Lab03 {
    public static void main(String[] args) {
        // Crear un conjunto inmutable
        // Los duplicados se eliminan
        Set<Integer> numbers = HashSet.of(1, 2, 3, 3, 4);

        // Elevar al cuadrado
        Set<Integer> squared = numbers.map(x -> x * x);
        // Verificar si contiene el número 3
        boolean containsThree = numbers.contains(3);

        System.out.println("Original: " + numbers); // [1, 2, 3, 4]
        System.out.println("Squared: " + squared); // [1, 4, 9, 16]
        System.out.println("Contains 3: " + containsThree); // true
    }
    // HashSet: En la mayoría de los casos, es la opción por defecto.
    // TreeSet: Mantiene los elementos ordenados
    //          (requiere que los elementos implementen Comparable o
    //           que se proporcione un Comparator).
    // LinkedHashSet: Mantiene el orden de inserción de los elementos.
}
```

```java
package com.circulosiete.curso.funcional.clase06;

import io.vavr.collection.HashMap;
import io.vavr.collection.Map;
import io.vavr.control.Option;

public class Lab04 {
    public static void main(String[] args) {
        // Crear un mapa inmutable
        Map<String, Integer> map = HashMap.of(
                "Alicia", 30,
                "Roberto", 25,
                "Carlos", 35
        );

        // Agregar un nuevo par clave-valor
        Map<String, Integer> updatedMap = map.put("David", 40);
        // Obtener la edad de Alicia
        Option<Integer> aliceAge = map.get("Alicia");

        System.out.println("Original: " + map); // {Alicia=30, Roberto=25, Carlos=35}
        System.out.println("Updated: " + updatedMap); // {Alicia=30, Roberto=25, Carlos=35, David=40}
        System.out.println("Edad de Alicia: " + aliceAge.getOrElse(0)); // 30
    }
    // LinkedHashMap, respeta el orden de inserción.
    // TreeMap, mantiene un orden natural (o el que definas con un Comparator).
}
```

```java
package com.circulosiete.curso.funcional.clase06;

import io.vavr.control.Option;

public class Lab05 {
    public static void main(String[] args) {
        // Crear un Option con valor
        Option<String> name = Option.of("Alicia");
        // Crear un Option vacío
        Option<String> emptyName = Option.none();

        System.out.println(name.getOrElse("Unknown")); // Alicia
        System.out.println(emptyName.getOrElse("Unknown")); // Unknown

        // Transformar el valor
        Option<String> upperCaseName = name.map(String::toUpperCase);
        System.out.println(upperCaseName.getOrElse("Unknown")); // ALICIA
    }
}
```

```java
package com.circulosiete.curso.funcional.clase06;

import io.vavr.control.Try;

/**
 * Try se utiliza para manejar operaciones que pueden lanzar excepciones.
 */
public class Lab06 {
    public static void main(String[] args) {
        // Ejecutar una operación que puede fallar
        Try<Integer> result = Try.of(() -> 10 / 0);

        // Manejar el resultado
        if (result.isSuccess()) {
            System.out.println("Resultado: " + result.get());
        } else {
            System.out.println("Error: " + result.getCause().getMessage()); // Error: / by zero
        }
    }
}
```

```java
package com.circulosiete.curso.funcional.clase06;

import io.vavr.Function1;

public class Lab07 {
    public static void main(String[] args) {
        Function1<Integer, Integer> addTen = x -> x + 10;
        Function1<Integer, Integer> multiplyByTwo = x -> x * 2;

        // Componer funciones
        Function1<Integer, Integer> composed = addTen.andThen(multiplyByTwo);

        System.out.println(composed.apply(5)); // (5 + 10) * 2 = 30
    }
}
```

```java
package com.circulosiete.curso.funcional.clase06;

import io.vavr.Function1;
import io.vavr.Function2;

/**
 * El currying permite dividir una función
 * en múltiples funciones más pequeñas.
 */
public class Lab08 {
    public static void main(String[] args) {
        Function2<Integer, Integer, Integer> sum = Integer::sum;

        // Aplicar currying
        Function1<Integer, Integer> addFive = sum
                .curried()
                .apply(5);

        System.out.println(addFive.apply(10)); // 15
    }
}
```

```java
package com.circulosiete.curso.funcional.clase06;

import java.util.Scanner;
import java.util.function.Function;
import java.util.function.Supplier;

import static io.vavr.API.*;

/**
 * Vavr introduce un sistema de pattern matching
 * que permite manejar diferentes casos de manera más clara.
 */
public class Lab09 {
    public static void main(String[] args) {
        int number = read();

        String result = Match(number).of(
                Case($(1), "Uno"),
                Case($(2), supplierForDos(number)),
                Case($(3), functionForTres()),
                Case($(), "No soportado")
        );

        System.out.println(result);
    }
}
```

```java
package com.circulosiete.curso.funcional.clase06;

import io.vavr.Lazy;

/**
 * Vavr soporta evaluación perezosa,
 * lo que permite retrasar la evaluación
 * de expresiones hasta que sea necesario.
 */
public class Lab10 {
    public static void main(String[] args) {
        Lazy<Integer> lazyValue = Lazy.of(() -> {
            System.out.println("Calculando...");
            return 42;
        });

        System.out.println("Valor tardío aún no calculado");
        System.out.println(lazyValue.get()); // Calculando... 42
        System.out.println(lazyValue.get()); // 42 (no se vuelve a calcular)
    }
}
```

Tipos de Lambdas

- Vavr proporciona una serie de interfaces funcionales (similares a las de Java) que permiten definir funciones de manera más flexible y potente.
- Estas interfaces funcionales están diseñadas para trabajar con programación funcional y admiten características como ***currying***, composición y aplicación parcial.

1. Funciones Básicas

- Vavr define interfaces funcionales para funciones de 0 a 8 parámetros.
- Estas interfaces se llaman ***Function0***, ***Function1***, ***Function2***, ..., ***Function8***.

```java
package com.circulosiete.curso.funcional.clase06;

import io.vavr.Function0;

/**
 * Function0 representa una función que
 * no toma argumentos y devuelve un valor.
 */
public class Lab11 {
    public static void main(String[] args) {
        Function0<String> saluda = () -> "Hola mundo!";
        System.out.println(saluda.apply()); // Hola mundo!
    }
}
```

```java
package com.circulosiete.curso.funcional.clase06;

import io.vavr.Function2;

/**
 * Function2 representa una función que
 * toma dos argumentos y devuelve un valor.
 */
public class Lab12 {
    public static void main(String[] args) {
        Function2<Integer, Integer, Integer> sum = Integer::sum;
        System.out.println(sum.apply(3, 5)); // 8
    }
}
```

2. Currying

- El currying es una técnica que permite transformar una función que toma múltiples argumentos en una secuencia de funciones que toman un solo argumento.
- Vavr soporta currying directamente en sus interfaces funcionales.
- Ver Lab08

3. Aplicación Parcial

- La aplicación parcial permite fijar algunos argumentos de una función y crear una nueva función con los argumentos restantes.

4. Funciones de Orden Superior

- Vavr permite pasar funciones como argumentos y devolver funciones como resultados, lo que es típico en la programación funcional.

```java
package com.circulosiete.curso.funcional.clase06;

import io.vavr.Function1;

/**
 * Ejemplo de Función de Orden Superior
 */
public class Lab14 {
    public static void main(String[] args) {
        // Función que toma una función como argumento
        Function1<
                Function1<Integer, Integer>,
                Integer
                > applyFunction = f -> f.apply(5);

        Function1<Integer, Integer> square = x -> x * x;

        System.out.println(applyFunction.apply(square)); // 25
    }
}
```

5. Funciones con Efectos Secundarios

- Vavr proporciona ***CheckedFunction*** para manejar funciones que pueden lanzar excepciones.

```java
package com.circulosiete.curso.funcional.clase06;

import io.vavr.CheckedFunction1;
import io.vavr.control.Try;

/**
 * CheckedFunction
 */
public class Lab15 {
    public static void main(String[] args) {
        CheckedFunction1<Integer, Integer> divideByZero = x -> 10 / x;

        // Usar Try para manejar excepciones
        Try<Integer> result = Try.of(() -> divideByZero.apply(0));

        if (result.isSuccess()) {
            System.out.println("Resultado: " + result.get());
        } else {
            System.out.println("Error: " + result.getCause().getMessage());
        }
    }
}
```

Interoperabilidad con SDK

- La interoperabilidad entre Vavr y la biblioteca estándar de Java es un aspecto importante a considerar, ya que Vavr está diseñado para complementar y mejorar las capacidades de Java, no para **reemplazarlas**.

1. Conversión entre Colecciones de Vavr y Java

- Vavr proporciona métodos convenientes para convertir entre sus colecciones inmutables y las colecciones estándar de Java (como ***List, Set, Map***, etc.).

```java
package com.circulosiete.curso.funcional.clase06;

import io.vavr.collection.List;

/**
 * De Vavr a Java
 */
public class Lab16 {
    public static void main(String[] args) {
        // Crear una lista de Vavr
        List<Integer> vavrList = List.of(1, 2, 3, 4, 5);

        // Convertir a una lista de Java
        java.util.List<Integer> javaList = vavrList.toJavaList();

        System.out.println("Vavr List: " + vavrList); // List(1, 2, 3, 4, 5)
        System.out.println("Java List: " + javaList); // [1, 2, 3, 4, 5]
    }
}
```

```java
package com.circulosiete.curso.funcional.clase06;

import io.vavr.collection.List;
import java.util.Arrays;

/**
 * De Java a Vavr
 */
public class Lab17 {
    public static void main(String[] args) {
        // Crear una lista de Java
        java.util.List<Integer> javaList = Arrays.asList(1, 2, 3, 4, 5);

        // Convertir a una lista de Vavr
        List<Integer> vavrList = List.ofAll(javaList);

        System.out.println("Java List: " + javaList); // [1, 2, 3, 4, 5]
        System.out.println("Vavr List: " + vavrList); // List(1, 2, 3, 4, 5)
    }
}
```

2. Interoperabilidad con Optional de Java

- Vavr proporciona ***Option***, que es similar a ***Optional*** de Java, pero con más funcionalidades. Puedes convertir fácilmente entre ***Option*** y ***Optional***.
- Ver ***Lab18.java***

3. Interoperabilidad con Streams de Java

- Vavr puede trabajar con ***Streams*** de Java, aunque Vavr tiene su propia implementación de ***Stream*** que es más funcional y poderosa.
- Ver ***Lab19.java***

4. Interoperabilidad con Funciones de Java

- Vavr puede trabajar con las interfaces funcionales de Java (***Function***, ***Consumer***, ***Supplier***, etc.), pero también proporciona sus propias interfaces funcionales (***Function1***, ***Function2***, etc.).
- Ver ***Lab20.java***

5. Interoperabilidad con Try y Optional

- Vavr proporciona Try para manejar operaciones que pueden fallar, similar a ***Optional*** pero con más capacidades.
- Puedes combinar ***Try*** con ***Optional*** de Java.

```java
package com.circulosiete.curso.funcional.clase06;

import io.vavr.control.Try;
import java.util.Optional;

public class Lab21 {
    public static void main(String[] args) {
        // Usar Try para manejar una operación que puede fallar
        Try<Integer> result = Try.of(() -> 10 / 0);

        // Convertir a Optional
        Optional<Integer> optionalResult = result.toJavaOptional();

        System.out.println("Optional: " + optionalResult); // Optional.empty
    }
}
```