**PARADIGMA FUNCIONAL EN JAVA**

El paradigma funcional en Java ha ganado popularidad gracias a la incorporación de características funcionales en las versiones recientes del lenguaje, especialmente a partir de Java 8. Aquí hay una introducción a los conceptos clave y las características funcionales en Java:

**Conceptos Clave del Paradigma Funcional**

1. **Funciones Puras**: Son funciones que siempre producen el mismo resultado para los mismos argumentos y no tienen efectos secundarios.
2. **Inmutabilidad**: Los datos no pueden ser modificados una vez creados. En lugar de modificar objetos existentes, se crean nuevos.
3. **Funciones de Orden Superior**: Son funciones que pueden aceptar otras funciones como parámetros y/o devolver funciones.
4. **Expresiones Lambda**: Son una forma compacta de representar funciones anónimas.
5. **Funciones Como Ciudadanos de Primera Clase**: Las funciones pueden ser asignadas a variables, pasadas como argumentos y devueltas desde otras funciones.

**Características Funcionales en Java**

1. **Expresiones Lambda**: Permiten escribir funciones anónimas de forma concisa.

(int x, int y) -> x + y

1. **Interfaces Funcionales**: Interfaces que tienen un único método abstracto. Java 8 introdujo varias interfaces funcionales en el paquete java.util.function, como Function, Predicate, Consumer, y Supplier.
2. **Streams**: Proveen una forma funcional de procesar secuencias de elementos. Permiten operaciones como map, filter, reduce, etc.

List<String> myList = Arrays.asList("a", "b", "c", "d");

myList.stream()

.map(String::toUpperCase)

.filter(s -> s.startsWith("A"))

.forEach(System.out::println);

1. **Optional**: Clase que encapsula un valor que puede estar presente o no, proporcionando una forma funcional de manejar valores nulos.

Optional<String> optional = Optional.of("hello");

optional.ifPresent(System.out::println);

**Ejemplos**

**Uso de Expresiones Lambda**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

List<Integer> doubled = numbers.stream()

.map(n -> n \* 2)

.collect(Collectors.toList());

System.out.println(doubled); // [2, 4, 6, 8, 10]

**Uso de Interfaces Funcionales**

Function<Integer, Integer> square = x -> x \* x;

System.out.println(square.apply(5)); // 25

**Uso de Streams**

List<String> names = Arrays.asList("John", "Jane", "Jack", "Doe");

List<String> filteredNames = names.stream()

.filter(name -> name.startsWith("J"))

.collect(Collectors.toList());

System.out.println(filteredNames); // [John, Jane, Jack]

**Uso de Optional**

Optional<String> name = Optional.of("John");

name.ifPresent(n -> System.out.println("Hello, " + n)); // Hello, John

**Beneficios del Paradigma Funcional**

1. **Concisión**: Las expresiones lambda y las operaciones de stream permiten escribir código más compacto y legible.
2. **Paralelización**: Los streams facilitan la paralelización de operaciones.
3. **Mantenibilidad**: El código funcional tiende a ser más fácil de razonar y mantener, ya que evita efectos secundarios y estados mutables.

**Conclusión**

El paradigma funcional en Java ofrece herramientas poderosas para escribir código más conciso, mantenible y eficiente. Aunque Java sigue siendo un lenguaje predominantemente orientado a objetos, las características funcionales introducidas en las versiones recientes permiten aprovechar los beneficios de la programación funcional sin abandonar el ecosistema de Java.

**¿Qué es un Stream?**

Un Stream en Java es una secuencia de elementos que soporta diferentes operaciones que se pueden canalizar para producir el resultado deseado. Estas operaciones pueden ser intermedias o terminales, y permiten manipular y procesar colecciones de datos de manera declarativa y funcional.

**Características de los Streams**

1. **No almacenan datos**: Los Streams no almacenan los elementos, sino que los procesan sobre la fuente de datos.
2. **Son funcionales**: Permiten utilizar expresiones lambda y referencias a métodos para operaciones.
3. **Operaciones perezosas**: Las operaciones intermedias son perezosas y se ejecutan solo cuando se invoca una operación terminal.
4. **Unidireccionales**: Los Streams se pueden consumir una sola vez.
5. **Soporte para paralelismo**: Facilitan el procesamiento paralelo.

**Creación de Streams**

1. **A partir de colecciones**

List<String> list = Arrays.asList("a", "b", "c");

Stream<String> stream = list.stream();

1. **A partir de Arrays**

String[] array = {"a", "b", "c"};

Stream<String> stream = Arrays.stream(array);

1. **A partir de valores individuales**

Stream<String> stream = Stream.of("a", "b", "c");

1. **A partir de archivos**

Stream<String> lines = Files.lines(Paths.get("path/to/file.txt"));

1. **A partir de generadores**

Stream<Integer> infiniteStream = Stream.iterate(0, n -> n + 1);

**Operaciones con Streams**

**Operaciones Intermedias**

1. **filter(Predicate<T>)**: Filtra los elementos según un predicado.

List<String> filtered = list.stream()

.filter(s -> s.startsWith("a"))

.collect(Collectors.toList());

1. **map(Function<T, R>)**: Transforma cada elemento aplicando una función.

List<Integer> lengths = list.stream()

.map(String::length)

.collect(Collectors.toList());

1. **flatMap(Function<T, Stream<R>>)**: Transforma cada elemento en un Stream y los aplana.

List<String> words = Arrays.asList("hello", "world");

List<String> characters = words.stream()

.flatMap(word -> Arrays.stream(word.split("")))

.collect(Collectors.toList());

1. **distinct()**: Elimina elementos duplicados.

List<Integer> distinctNumbers = list.stream()

.distinct()

.collect(Collectors.toList());

1. **sorted()**: Ordena los elementos de manera natural o según un comparador.

List<String> sorted = list.stream()

.sorted()

.collect(Collectors.toList());

1. **peek(Consumer<T>)**: Permite realizar una acción en cada elemento sin consumir el stream.

List<String> peeked = list.stream()

.peek(System.out::println)

.collect(Collectors.toList());

**Operaciones Terminales**

1. **collect(Collector<T, A, R>)**: Recoge los elementos en una colección.

List<String> collected = list.stream()

.collect(Collectors.toList());

1. **forEach(Consumer<T>)**: Realiza una acción en cada elemento.

list.stream()

.forEach(System.out::println);

1. **reduce(BinaryOperator<T>)**: Reduce los elementos a un solo valor.

int sum = list.stream()

.reduce(0, Integer::sum);

1. **count()**: Devuelve el número de elementos en el stream.

long count = list.stream().count();

1. **findFirst()**: Encuentra el primer elemento del stream.

Optional<String> first = list.stream().findFirst();

1. **allMatch(Predicate<T>)**, **anyMatch(Predicate<T>)**, **noneMatch(Predicate<T>)**: Verifican si todos, algunos o ninguno de los elementos coinciden con un predicado.

boolean

**Conceptos Clave de Streams**

1. **Fuente**: Un Stream obtiene sus elementos de una colección, array, o cualquier otra fuente.
2. **Operaciones Intermedias**: Transforman un Stream en otro Stream. Estas operaciones son perezosas, es decir, no se ejecutan hasta que se realiza una operación terminal. Ejemplos incluyen filter, map, distinct, sorted, limit.
3. **Operaciones Terminales**: Producen un resultado o un efecto secundario. Estas operaciones hacen que se ejecuten las operaciones

Ejemplo de prueba de STREAMS

import java.util.Arrays;  
import java.util.List;  
import java.util.Optional;  
import java.util.stream.Collectors;  
  
public class Main {  
  
 public static void main(String[] args) {  
 List<String> list = Arrays.*asList*("apple", "banana", "cherry", "date", "elderberry", "fig", "grape");  
  
 *// Operaciones Intermedias* System.*out*.println("Filter: ");  
 List<String> filtered = list.stream()  
 .filter(s -> s.startsWith("a"))  
 .collect(Collectors.*toList*());  
 System.*out*.println(filtered);  
  
 System.*out*.println("Map: ");  
 List<Integer> lengths = list.stream()  
 .map(String::length)  
 .collect(Collectors.*toList*());  
 System.*out*.println(lengths);  
  
 System.*out*.println("FlatMap: ");  
 List<String> words = Arrays.*asList*("hello", "world");  
 List<String> characters = words.stream()  
 .flatMap(word -> Arrays.*stream*(word.split("")))  
 .collect(Collectors.*toList*());  
 System.*out*.println(characters);  
  
 System.*out*.println("Distinct: ");  
 List<Integer> distinctNumbers = list.stream()  
 .map(String::length)  
 .distinct()  
 .collect(Collectors.*toList*());  
 System.*out*.println(distinctNumbers);  
  
 System.*out*.println("Sorted: ");  
 List<String> sorted = list.stream()  
 .sorted()  
 .collect(Collectors.*toList*());  
 System.*out*.println(sorted);  
  
 System.*out*.println("Peek: ");  
 List<String> peeked = list.stream()  
 .peek(System.*out*::println)  
 .collect(Collectors.*toList*());  
 System.*out*.println(peeked);  
  
 *// Operaciones Terminales* System.*out*.println("Collect: ");  
 List<String> collected = list.stream()  
 .collect(Collectors.*toList*());  
 System.*out*.println(collected);  
  
 System.*out*.println("ForEach: ");  
 list.stream()  
 .forEach(System.*out*::println);  
  
 System.*out*.println("Reduce: ");  
 int sum = list.stream()  
 .map(String::length)  
 .reduce(0, Integer::*sum*);  
 System.*out*.println(sum);  
  
 System.*out*.println("Count: ");  
 long count = list.stream().count();  
 System.*out*.println(count);  
  
 System.*out*.println("FindFirst: ");  
 Optional<String> first = list.stream().findFirst();  
 System.*out*.println(first.orElse("No hay elementos"));  
  
 System.*out*.println("AllMatch: ");  
 boolean allMatch = list.stream()  
 .allMatch(s -> s.length() > 3);  
 System.*out*.println(allMatch);  
  
 System.*out*.println("AnyMatch: ");  
 boolean anyMatch = list.stream()  
 .anyMatch(s -> s.startsWith("a"));  
 System.*out*.println(anyMatch);  
  
 System.*out*.println("NoneMatch: ");  
 boolean noneMatch = list.stream()  
 .noneMatch(s -> s.isEmpty());  
 System.*out*.println(noneMatch);  
 }

**Vamos a analizar la clase de ejemplo paso a paso:**

**1. Importaciones**

import java.util.Arrays;

import java.util.List;

import java.util.Optional;

import java.util.stream.Collectors;

La clase importa las clases necesarias para trabajar con listas, arrays, Optional y Streams.

**2. Declaración de la lista**

List<String> list = Arrays.asList("apple", "banana", "cherry", "date", "elderberry", "fig", "grape");

Se declara una lista de strings llamada list que contiene 7 elementos.

**3. Filter**

System.out.println("Filter: ");

List<String> filtered = list.stream()

.filter(s -> s.startsWith("a"))

.collect(Collectors.toList());

System.out.println(filtered);

Se crea un Stream a partir de la lista list utilizando el método stream().

Se aplica el método filter() para filtrar los elementos que comienzan con la letra "a". El predicado s -> s.startsWith("a") se utiliza para determinar si un elemento cumple con la condición.

Se utiliza el método collect() para recopilar los elementos filtrados en una nueva lista.

Se imprime la lista filtrada.

**4. Map**

System.out.println("Map: ");

List<Integer> lengths = list.stream()

.map(String::length)

.collect(Collectors.toList());

System.out.println(lengths);

Se crea un Stream a partir de la lista list utilizando el método stream().

Se aplica el método map() para transformar cada elemento en su longitud utilizando el método String::length.

Se utiliza el método collect() para recopilar los resultados en una nueva lista de enteros.

Se imprime la lista de longitudes.

**5. FlatMap**

System.out.println("FlatMap: ");

List<String> words = Arrays.asList("hello", "world");

List<String> characters = words.stream()

.flatMap(word -> Arrays.stream(word.split("")))

.collect(Collectors.toList());

System.out.println(characters);

Se declara una lista de strings words que contiene 2 elementos.

Se crea un Stream a partir de la lista words utilizando el método stream().

Se aplica el método flatMap() para transformar cada elemento en un Stream de caracteres utilizando el método Arrays.stream(word.split("")).

Se utiliza el método collect() para recopilar los caracteres en una nueva lista.

Se imprime la lista de caracteres.

**6. Distinct**

System.out.println("Distinct: ");

List<Integer> distinctNumbers = list.stream()

.map(String::length)

.distinct()

.collect(Collectors.toList());

System.out.println(distinctNumbers);

Se crea un Stream a partir de la lista list utilizando el método stream().

Se aplica el método map() para transformar cada elemento en su longitud utilizando el método String::length.

Se aplica el método distinct() para eliminar los elementos duplicados.

Se utiliza el método collect() para recopilar los resultados en una nueva lista de enteros.

Se imprime la lista de números únicos.

**7. Sorted**

System.out.println("Sorted: ");

List<String> sorted = list.stream()

.sorted()

.collect(Collectors.toList());

System.out.println(sorted);

Se crea un Stream a partir de la lista list utilizando el método stream().

Se aplica el método sorted() para ordenar los elementos en orden alfabético.

Se utiliza el método collect() para recopilar los resultados en una nueva lista.

Se imprime la lista ordenada.

8. Peek

System.out.println("Peek: ");

List<String> peeked = list.stream()

.peek(System.out::println)

.collect(Collectors.toList());

System.out.println(peeked);

Se crea un Stream a partir de la lista list utilizando el método stream().

Se aplica el método peek() para imprimir cada elemento en la consola utilizando el método System.out::println.

Se utiliza el método collect() para recopilar los resultados en una nueva lista.

Se imprime la lista.

**9. Collect**

System.out.println("Collect: ");

List<String> collected = list.stream()

.collect(Collectors.toList());

System.out.println(collected);

Se crea un Stream a partir de la lista list utilizando el método stream().

Se utiliza el método collect() para recopilar los resultados en una nueva lista.

Se imprime la lista.

**10. ForEach**

System.out.println("ForEach: ");

list.stream()

.forEach(System.out::println);

Se crea un Stream a partir de la lista list utilizando el método stream().

Se aplica el método forEach() para imprimir cada elemento en la consola utilizando el método System.out::println.

**11. Reduce**

System.out.println("Reduce: ");

int sum = list.stream()

.map(String::length)

.reduce(0, Integer::sum);

System.out.println(sum);

Se crea un Stream a partir de la lista list utilizando el método stream().

Se aplica el método map() para transformar cada elemento en su longitud utilizando el método String::length.

Se aplica el método reduce() para reducir los resultados en una suma utilizando el método Integer::sum.

Se imprime la suma.

12. Count

System.out.println("Count: ");

long count = list.stream().count();

System.out.println(count);

Se crea un Stream a partir de la lista list utilizando el método stream().

Se aplica el método count() para contar el número de elementos en el Stream.

Se imprime el conteo.

**13. FindFirst**

System.out.println("FindFirst: ");

Optional<String> first = list.stream().findFirst();

System.out.println(first.orElse("No hay elementos"));

Se crea un Stream a partir de la lista list utilizando el método stream().

Se aplica el método findFirst() para encontrar el primer elemento en el Stream.

Se utiliza el método orElse() para imprimir el primer elemento o un mensaje de "No hay elementos" si el Stream está vacío.

**14. AllMatch, AnyMatch, NoneMatch**

System.out.println("AllMatch: ");

boolean allMatch = list.stream()

.allMatch(s -> s.length() > 3);

System.out.println(allMatch);

System.out.println("AnyMatch: ");

boolean anyMatch = list.stream()

.anyMatch(s -> s.startsWith("a"));

System.out.println(anyMatch);

System.out.println("NoneMatch: ");

boolean noneMatch = list.stream()

.noneMatch(s -> s.isEmpty());

System.out.println(noneMatch);

Se crea un Stream a partir de la lista list utilizando el método stream().

Se aplica el método allMatch() para verificar si todos los elementos cumplen con la condición s.length() > 3.

Se aplica el método anyMatch() para verificar si algún elemento cumple con la condición s.startsWith("a").

Se aplica el método noneMatch() para verificar si ningún elemento cumple con la condición s.isEmpty().

Se imprimen los resultados.