Stream API en Java

La **Stream API** es una característica introducida en **Java 8** que permite procesar colecciones de datos mediante un enfoque funcional y declarativo.

A diferencia de las colecciones tradicionales, un Stream **no almacena datos**, sino que actúa como un **pipeline** (tubería) que extrae información de una fuente (como listas, arrays o canales de E/S) y aplica operaciones secuenciales sin modificar la fuente original.

Este mecanismo facilita la manipulación eficiente de datos mediante dos tipos principales de operaciones:

- **Operaciones intermedias**: Son **diferidas** (*lazy*) y devuelven un nuevo `Stream`, permitiendo encadenar acciones como `filter`, `map`, `sorted` o `distinct` sin ejecutar el procesamiento inmediato.
- **Operaciones terminales**: Marcan el final del *pipeline*, ejecutan todas las operaciones intermedias acumuladas y producen un resultado final o un efecto secundario, como `collect`, `forEach`, `reduce` o `count`.

Además, la API soporta **procesamiento paralelo** a través del framework *Fork/Join*, lo que permite dividir grandes conjuntos de datos entre varios núcleos de CPU para mejorar el rendimiento, y se integra naturalmente con las **expresiones lambda** y referencias de métodos para escribir código más conciso y legible.

Un Stream API Java

Un ejemplo clásico es filtrar, transformar y ordenar una lista de nombres:

```
List<String> nombres = List.of("patricio", "juan", "maría", "carlos");
List<String> resultado = nombres.stream()
    .filter(n -> n.length() > 4)
    .map(String::toUpperCase)
    .sorted()
    .toList();
```

Características clave del ejemplo:

- **Fuente**: La lista `nombres` se convierte en un stream mediante `.stream()`.
- **Operaciones intermedias**: `.filter()` selecciona elementos, `.map()` los transforma y `.sorted()` los ordena. Estas son **perezosas** (*lazy*) y no ejecutan la lógica hasta que se llama a una operación terminal.
- **Operación terminal**: `.toList()` consume el stream y produce el resultado final.
- **Inmutabilidad**: La lista original no se modifica; se crea una nueva lista con los resultados.

Este enfoque reemplaza bucles tradicionales, haciendo el código más conciso y legible.

Ejemplos Básicos Stream API

Los Streams en Java, introducidos en la versión 8, permiten procesar colecciones de datos de manera declarativa y funcional. A continuación se presentan ejemplos básicos de las operaciones más comunes.

1. Procesar colecciones de datos de manera funcional mediante un *pipeline*:

   - **Una fuente de datos**
   - **Operaciones intermedias** (*lazy*/perezoso) y
   - Una **operación terminal** (*eager*/impaciente)

   Un ejemplo completo típico implica *filtrar elementos*, *transformarlos* y *agruparlos o reducirlos* en un resultado final.

   A continuación se presenta un ejemplo que obtiene:

   - Una lista de nombres de empleados
   - Filtra a aquellos con un salario superior a un umbral
   - Los ordena y los agrupa por departamento
   - Devolviendo un mapa con la suma de salarios por departamento

```
import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class EjemploStream {
    static class Empleado {
        String nombre;
        String departamento;
        double salario;

        public Empleado(String nombre, String departamento, double salario) {
            this.nombre = nombre;
            this.departamento = departamento;
            this.salario = salario;
        }

        public String getNombre() { return nombre; }
        public String getDepartamento() { return departamento; }
        public double getSalario() { return salario; }
    }

    public static void main(String[] args) {
        List<Empleado> empleados = Arrays.asList(
            new Empleado("Ana", "Ventas", 3000),
            new Empleado("Luis", "IT", 4500),
            new Empleado("Carlos", "Ventas", 2800),
            new Empleado("Elena", "IT", 5000),
            new Empleado("Marta", "RRHH", 3200)
        );

        // Pipeline de Stream:
        // 1. Obtener stream
        // 2. Filtrar (salario > 3000)
        // 3. Mapear (extraer departamento y salario para sumar)
        // 4. Reducir/Agrupar (sumar salarios por departamento)
        Map<String, Double> salariosPorDepto = empleados.stream()
            .filter(emp -> emp.getSalario() > 3000)
            .collect(Collectors.groupingBy(
                Empleado::getDepartamento,
                Collectors.summingDouble(Empleado::getSalario)
```

```
            ));

        System.out.println(salariosPorDepto);
        // Salida: {IT=9500.0, RRHH=3200.0}
    }
}
```

Este ejemplo demuestra las características clave:

- **Operaciones Intermedias**: `filter` y `map` (o transformaciones implícitas en `collect`) no ejecutan el procesamiento hasta que se invoca la operación terminal.
- **Operación Terminal**: `collect` consume el `stream` y produce el resultado final (`Map<String, Double>`).
- **Uso de Lambdas**: Las expresiones como `emp ->emp.getSalario() >3000` definen el comportamiento funcional sin necesidad de clases internas anónimas.

2. **Filtrado y Transformación**

   El método `filter()` selecciona elementos según una condición (recibe un `Predicate`), y `map()` transforma cada elemento en otro tipo o valor (recibe un `Function`).

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6);

// Filtrar números pares y calcular su cuadrado
List<Integer> cuadradosPares = numeros.stream()
    .filter(n -> n % 2 == 0)
    .map(n -> n * n)
    .collect(Collectors.toList());

System.out.println(cuadradosPares); // Resultado: [4, 16, 36]
```

3. **Extracción de Campos (Map)**

   Un uso habitual es extraer un campo específico de una lista de objetos utilizando referencias a métodos.

```
List<String> nombres = Arrays.asList("Ana", "Luis", "María");

// Convertir todos los nombres a mayúsculas
List<String> mayusculas = nombres.stream()
    .map(String::toUpperCase)
    .collect(Collectors.toList());

System.out.println(mayusculas); // Resultado: [ANA, LUIS, MARÍA]
```

4. **Ordenación y Recolección**

   El método `sorted()` ordena los elementos (usando el orden natural o un `Comparator`), y `collect()` termina el pipeline almacenando los resultados en una estructura de datos como una `List` o un `Set`.

```
List<String> ciudades = Arrays.asList("Madrid", "Barcelona", "Sevilla");

// Ordenar alfabéticamente y recolectar en una lista
List<String> ordenadas = ciudades.stream()
    .sorted()
    .collect(Collectors.toList());

System.out.println(ordenadas); // Resultado: [Barcelona, Madrid, Sevilla]
```

5. **Operaciones Terminales Básicas**

   Operaciones como `count()`, `findFirst()` o `forEach()` finalizan el procesamiento del Stream.

```
List<String> nombres = Arrays.asList("Ana", "Luis", "Pedro");

// Contar nombres con más de 3 letras
long contador = nombres.stream()
    .filter(n -> n.length() > 3)
    .count();

System.out.println("Nombres largos: " + contador); // Resultado: 3

// Imprimir cada nombre
nombres.stream()
    .forEach(n -> System.out.println(n));
```

   Puntos clave:

   - **Operaciones Intermedias**: `filter`, `map`, `sorted`, `distinct`, `limit`. Devuelven un nuevo Stream y son *lazy* (no se ejecutan hasta que hay una operación terminal).
   - **Operaciones Terminales**: `collect`, `forEach`, `count`, `reduce`, `findFirst`. Devuelven un resultado (valor, colección o void) y disparan la ejecución del pipeline.
   - **Inmutabilidad**: Las operaciones no modifican la fuente original, sino que generan nuevos Streams.