API de Flujo en Java

Para usar `java.util.stream` en Java, puedes aprovechar la API de Streams introducida en Java 8 para procesar colecciones de datos de manera funcional y declarativa. Los streams te permiten realizar operaciones como filtrado, mapeo y reducción en secuencias de elementos (como listas o arreglos) de manera eficiente y concisa. A continuación, explicaré cómo usar `java.util.stream` a través de un ejemplo específico: encontrar el promedio de los números pares mayores que 10 de una lista de enteros. Esto demostrará operaciones y conceptos clave de streams.

Descripción general de `java.util.stream`

El paquete `java.util.stream` proporciona la interfaz `Stream` y clases relacionadas (como `IntStream`, `DoubleStream`, etc.) para procesar datos. Un stream es una secuencia de elementos que soporta operaciones ejecutadas en una tubería. Estas operaciones son: - **Operaciones intermedias**: Transforman o filtran el stream (por ejemplo, `filter`, `map`), devolviendo un nuevo stream. Estas son perezosas y solo se ejecutan cuando se invoca una operación terminal. - **Operaciones terminales**: Producen un resultado o efecto secundario (por ejemplo, `average`, `collect`), desencadenando la tubería para procesar los datos.

Para usar streams, típicamente: 1. Crea un stream a partir de una fuente de datos (por ejemplo, una lista). 2. Aplica operaciones intermedias para transformar o filtrar los datos. 3. Usa una operación terminal para producir un resultado.

Problema de ejemplo

Vamos a resolver este problema: Dada una `List<Integer>`, calcula el promedio de todos los números pares mayores que 10. Si no existen tales números, devuelve 0.0. Aquí está cómo hacerlo usando `java.util.stream`.

Solución paso a paso

1. **Crear un Stream**
   - Comienza con una `List<Integer>` (por ejemplo, `List.of(1, 2, 12, 15, 20, 25, 30)`).
   - Usa el método `stream()` para crear un `Stream<Integer>`:

     ```
     list.stream()
     ```

2. **Filtrar el Stream**
   - Usa el método `filter` para mantener solo los números que son pares y mayores que 10.

- El método `filter` toma un `Predicate` (una función que devuelve un booleano) como una expresión lambda:

  ```
  .filter(number -> number % 2 == 0 && number > 10)
  ```

  - `number % 2 == 0` verifica si un número es par.
  - `number > 10` asegura que el número sea mayor que 10.
  - Para la lista de ejemplo `[1, 2, 12, 15, 20, 25, 30]`, esto mantiene `[12, 20, 30]`.

3. **Convertir a un `IntStream`**

   - Dado que `average()` está disponible en streams primitivos como `IntStream` (no en `Stream<Integer>`), convierte el `Stream<Integer>` a un `IntStream` usando `mapToInt`:

     ```
     .mapToInt(i -> i)
     ```

     - `i -> i` desenvuelve cada `Integer` a un `int`. Alternativamente, podrías usar `Integer::intValue`.
     - Esto da un `IntStream` de `[12, 20, 30]`.

4. **Calcular el Promedio**

   - Usa el método `average()` en `IntStream`, que devuelve un `OptionalDouble` (ya que el stream podría estar vacío):

     ```
     .average()
     ```

     - Para `[12, 20, 30]`, esto calcula `(12 + 20 + 30) / 3 = 20.666...`.
     - Si el stream está vacío, devuelve un `OptionalDouble` vacío.

5. **Manejar el caso vacío**

   - Usa `orElse(0.0)` en el `OptionalDouble` para devolver 0.0 si ningún número cumple con el filtro:

     ```
     .orElse(0.0)
     ```

     - Para `[12, 20, 30]`, esto devuelve `20.666...`.
     - Para una lista como `[1, 3, 5]` (sin números pares > 10), devuelve `0.0`.

Código completo Aquí está la solución completa como un método:

```java
import java.util.List;

public class Main {
    public static double averageOfEvenGreaterThanTen(List<Integer> list) {
        return list.stream()
                   .filter(number -> number % 2 == 0 && number > 10)
                   .mapToInt(i -> i)
                   .average()
                   .orElse(0.0);
    }
```

```
    public static void main(String[] args) {
        // Casos de prueba
        System.out.println(averageOfEvenGreaterThanTen(List.of(1, 2, 12, 15, 20, 25, 30)));  // ~20.666
        System.out.println(averageOfEvenGreaterThanTen(List.of(1, 3, 5)));                   // 0.0
        System.out.println(averageOfEvenGreaterThanTen(List.of(12)));                        // 12.0
        System.out.println(averageOfEvenGreaterThanTen(List.of()));                          // 0.0
        System.out.println(averageOfEvenGreaterThanTen(List.of(10, 8, 6)));                  // 0.0
        System.out.println(averageOfEvenGreaterThanTen(List.of(11, 13, 14, 16)));            // 15.0
    }
}
```

Características clave de `java.util.stream` demostradas

- **Encadenamiento**: Las operaciones se encadenan en un estilo fluido (por ejemplo, `.filter().mapToInt().averag`
- **Perezosidad**: Las operaciones intermedias (como `filter` y `mapToInt`) solo se ejecutan cuando se llama a la operación terminal (`average`).
- **Streams primitivos**: `IntStream` proporciona métodos como `average()` diseñados para operaciones numéricas, evitando el sobrecoste de empaquetamiento.
- **Manejo de `Optional`**: `OptionalDouble` maneja los casos donde no existe un resultado, con `orElse` proporcionando un valor predeterminado.

Enfoque alternativo

También podrías usar la clase `Collectors`:

```
import java.util.stream.Collectors;

double average = list.stream()
                     .filter(number -> number % 2 == 0 && number > 10)
                     .collect(Collectors.averagingInt(i -> i));
```

- Esto devuelve un `Double` directamente y devuelve `0.0` para un stream vacío. Sin embargo, `mapToInt().average()` es más directo para este caso simple y se alinea con el uso de streams primitivos.

Cuándo usar Streams

- Usa `java.util.stream` para código conciso y legible al procesar colecciones.
- Para grandes conjuntos de datos, considera `parallelStream()` en lugar de `stream()` para aprovechar el paralelismo, aunque esto es exagerado para listas pequeñas debido al sobrecoste.

Este ejemplo muestra cómo usar `java.util.stream` para resolver un problema práctico. ¡Puedes adaptar estos pasos —crear un stream, filtrar, mapear y reducir—a otros escenarios según sea necesario!