STREAMS INTERMEDIOS

Introducción

Los Streams procesan colecciones de datos de forma eficiente y declarativa. Las operaciones que se pueden realizar se dividen en dos tipos: intermedias y terminales.

Las operaciones intermedias son aquellas que transforman un Stream en otro Stream, lo que permite modificar, filtrar o mapear los elementos sin consumirlo, lo que significa que se pueden seguir realizando operaciones sobre este. Estas operaciones son perezosas (lazy), lo que significa que no se ejecutan inmediatamente, sino hasta que se ejecuta una operación terminal. Esto optimiza el rendimiento y permite encadenar transformaciones de forma eficiente.

Características

- **Lazy evaluation:** No se ejecutan inmediatamente cuando se invocan y solo se aplican cuando se encadena una operación terminal. Permiten optimizar el rendimiento al evitar cálculos innecesarios.

- **Transforman el Stream sin consumirlo:** Devuelven un Stream con los datos transformados.

- **Son inmutables:** No modifican la fuente de datos original, lo que significa que si tenemos un array al que le queremos hacer operaciones con un stream, este array no se ve modificado.

- **Pueden ser de tipo stateful o stateless:** Los stateful son *filter()*, *map()*, *flatMap()* y *peek()*, ya que no dependen de los elementos que han procesado anteriormente. Los stateless son *distinct()*, sorted(), *limit()*, *skip()*, *takeWhile()* y *dropWhile()*, estos si necesitan conocer otros elementos para operar.

- **Pueden mejorar el rendimiento:** Operaciones como *limit()*, *skip()*, *takeWhile()* y *dropWhile()*.

Métodos

Estos son los métodos de Stream intermediate que hay en la API de Java 11.

- filter(): Se utiliza para filtrar los elementos de un Stream según una condición específica.

```java
// Filtra y devuelve los productos que cuestan más de 50€
public Stream<Producto> productosCaros() {
    return list.stream().filter(p -> p.getPrecio() > 50);
}
```

- map(): Transforma cada elemento de un stream aplicando una función a cada uno, y devuelve un nuevo stream con los resultados de esa transformación.

```java
// Extrae y devuelve solo los nombres de los productos
public Stream<String> mostrarNombresProductos() {
    return list.stream().map(Producto::getNombre);
}
```

- mapToInt/Double/Long(): Transforma los elementos del stream en un IntStream/DoubleStream/LongStream de valores primitivos int/double/long.

```java
// Convierte los precios de los productos a enteros y los devuelve en un Stream
public IntStream preciosRedondeados() {
    return list.stream().mapToInt(p -> (int) p.getPrecio());
}
```

- flatMap(): Aplica una función a cada elemento de un stream, y la función debe devolver un Stream de elementos. Luego, flatMap aplana los resultados y los combina en un único Stream.

```java
// Extrae y devuelve todas las letras de los nombres de los productos, una por
// una
public Stream<Character> mostrarLetrasDeProductos() {
    return list.stream().flatMap(p -> p.getNombre().chars().mapToObj(c -> (char) c));
}
```

- flatMapToInt/Double/Long(): Funciona igual que el flatMap, pero los convierte en valores primitivos int/double/long.

```
// Convierte los caracteres de los nombres de los productos a sus valores ASCII
public Stream<Integer> mostrarValoresASCII() {
    return list.stream().flatMapToInt(p -> p.getNombre().chars()).boxed();
}
```

- distinct(): Elimina los elementos duplicados de un stream basándose en la igualdad de los elementos según el método equals() y devuelve un nuevo Stream.

```
// Devuelve un Stream sin productos repetidos
public Stream<Producto> mostrarProductosUnicos() {
    return list.stream().distinct();
}
```

- sorted(): Está reescrito de dos formas:
    - sorted(): Ordena los elementos del stream según el orden natural de los elementos. Funciona si implementan la interfaz Comparable.

```
// Ordena los productos por precio de menor a mayor
public Stream<Producto> ordenarPorPrecioAscendente() {
    return list.stream().sorted(Comparator.comparingDouble(Producto::getPrecio));
}
```

    - sorted(Comparator<? super T> comparator): Ordena los elementos del stream utilizando una interfaz Comparator que define cómo comparar los elementos.

```
// Ordena los productos por precio de mayor a menor
public Stream<Producto> ordenarPorPrecioDescendente() {
    return list.stream().sorted((p1, p2) -> Double.compare(p2.getPrecio(), p1.getPrecio()));
}
```

- peek(): Permite realizar una acción en cada elemento del stream sin modificar el flujo. Se usa generalmente para depuración o verificación, ya que no altera los elementos del stream.

```java
// Muestra detalles de cada producto sin modificar la lista
public Stream<Producto> mostrarProductosConDetalles() {
    return list.stream().peek(System.out::println);
}
```

- limit(): Restringe el número de elementos de un stream, devolviendo un nuevo stream con solo el número de elementos especificado como argumento.

```java
// Devuelve los 3 productos más baratos de la lista
public Stream<Producto> tresProductosMasEconomicos() {
    return list.stream().sorted(Comparator.comparingDouble(Producto::getPrecio)).limit(3);
}
```

- skip(): Omite el número de elementos especificado como argumento de un stream y devuelve un nuevo stream con los elementos restantes.

```java
// Omite los 2 productos más baratos y devuelve el resto
public Stream<Producto> omitirProductosEconomicos() {
    return list.stream().sorted(Comparator.comparingDouble(Producto::getPrecio)).skip(2);
}
```

- takeWhile(): Devuelve los elementos de un Stream hasta que se cumple una condición, deteniéndose en el primer elemento que no cumpla esta condición.

```java
// Devuelve los productos hasta encontrar el primero que cuesta más de 100€
public Stream<Producto> mostrarProductosHasta100() {
    return list.stream().takeWhile(p -> p.getPrecio() <= 100);
}
```

- dropWhile(): Omite los elementos de un Stream mientras cumplan una condición, y luego devuelve el resto de los elementos, comenzando desde el primer elemento que no cumpla la condición.

```java
// Omite los productos baratos y devuelve la lista a partir del primero que
// cuesta más de 100€
public Stream<Producto> omitirHastaEncontrarProductoCaro() {
    return list.stream().dropWhile(p -> p.getPrecio() <= 100);
}
```

Ventajas y desventajas

- **Ventajas:**
  - Permiten escribir código más corto y fácil de entender, sobre todo cuando se usan operaciones complejas de filtrado, mapeo y reducción.
  - Pueden ser procesados en paralelo, lo que mejora el rendimiento en sistemas con múltiples núcleos de CPU.
  - Permiten expresar operaciones de manera declarativa, lo que significa que describimos lo que queremos hacer pero no el cómo hacerlo, esto mejora la claridad del código y reduce los errores.
  - Podemos operar sobre colecciones grandes de manera más eficiente, mediante la ejecución perezosa, lo que significa que solo se procesan los elementos necesarios.
  - Se pueden encadenar múltiples operaciones de transformación y filtrado.

- **Desventajas:**
  - Al principio puede ser complicado entender cómo funcionan los streams, sobre todo si no se tiene experiencia en este tipo de programación.
  - Para tareas sencillas, como recorrer una lista y hacer algo básico con sus elementos, los streams pueden ser más lentos debido a la sobrecarga de su infraestructura.

- El código que usa streams puede ser más difícil de depurar, ya que no siempre es claro dónde ocurre un error debido a la forma en que se procesan los datos de manera declarativa.
- Pueden ser más rápidos en algunos casos, pero en otros no ofrecen mejoras significativas respecto a los bucles tradicionales, especialmente si las tareas no son muy complejas.
- No modifican directamente las colecciones con las que trabajan, lo que puede ser un problema si necesitamos hacer cambios directamente en los datos.