Operaciones intermedias

En clases anteriores hablamos de dos tipos de operaciones: **intermedias** y **finales**. No se explicaron a profundidad, pero en esta lectura iremos más a fondo en las **operaciones intermedias** y trataremos de entender qué sucede por dentro de cada una.

¿Qué son?

Se le dice **operación intermedia** a toda operación dentro de un `Stream` que como resultado devuelva un nuevo `Stream`. Es decir, tras invocar una operación intermedia con un cierto tipo de dato, obtendremos como resultado un nuevo `Stream` conteniendo los datos ya modificados.

El `Stream` que recibe la operación intermedia pasa a ser "*consumido*" posterior a la invocación de la operación, quedando inutilizable para posteriores operaciones. Si decidimos usar el `Stream` para algún otro tipo de operaciones tendremos un `IllegalStateException`.

Viéndolo en código con un ejemplo debe ser mas claro:

```
Stream initialCourses = Stream.of("Java", "Spring", "Node.js");

Stream lettersOnCourses = initialCourses.map(course -> course.length());
//De este punto en adelante, initialCourses ya no puede agregar mas
operaciones.

Stream evenLengthCourses = lettersOnCourses.filter(courseLength ->
courseLength % 2 == 0);
//lettersOnCourses se consume en este punto y ya no puede agregar mas
operaciones. No es posible usar el Stream mas que como referencia.
```

El ejemplo anterior puede ser reescrito usando *chaining*. Sin embargo, la utilidad de este ejemplo es demostrar que las operaciones intermedias generan un

nuevo `Stream`.

Operaciones disponibles

La interfaz `Stream` cuenta con un grupo de operaciones intermedias. A lo largo de esta lectura explicaremos cada una de ellas y trataremos de aproximar su funcionalidad. Cada operación tiene implementaciones distintas según la implementación de `Stream`, en nuestro caso, haremos solo aproximaciones de la lógica que sigue la operación.

Las operaciones que ya están definidas son:

- `filter(…)`
- `map(…)`
- `flatMap(…)`
- `distinct(…)`
- `limit(…)`
- `peek(…)`
- `skip(…)`
- `sorted(…)`

Analicemos qué hace cada una de ellas y hagamos código que **se aproxime** a lo que hacen internamente.

filter

La operación de filtrado de `Stream` tiene la siguiente forma:

```
Stream filter(Predicatesuper T> predicate)
```

Algunas cosas que podemos deducir únicamente viendo los elementos de la operación son:

- La operación trabaja sobre un `Stream` y nos devuelve un nuevo `Stream` del mismo tipo (`T`)
- Sin embargo, el `Predicate` que recibe como parámetro trabaja con elementos de tipo `T` y cualquier elemento siempre que sea un subtipo de `T`. Esto quiere decir que si tenemos la clase `PlatziStudent extends Student` y tenemos un `Stream` donde también tenemos elementos de tipo `PlatziStudent`, podemos filtrarlos sin tener que revisar o aclarar el tipo
- `Predicate` es una `@FunctionalInterface` (como lo viste en la clase 11), lo cual nos permite pasar como parámetro objetos que implementen esta interfaz o lambdas

El uso de esta operación es sencillo:

```
public Stream getJavaCourses(List courses){
    return courses.stream()
        .filter(course -> course.contains("Java"));
}
```

Lo interesante radica en la condición que usamos en nuestra lambda, con ella determinamos si un elemento debe permanecer o no en el `Stream` resultante. En la lectura anterior hicimos una aproximación de la operación `filter`:

```
public Stream filter(Predicate predicate) {
    List filteredData = new LinkedList<>();
    for(T t : this.data){
        if(predicate.test(t)){
            filteredData.add(t);
        }
    }

    return filteredData.stream();
}
```

`filter` se encarga de iterar cada elemento del `Stream` y evaluar con el `Predicate`si el elemento debe estar o no en el `Stream` resultante. Si nuestro `Predicate` es sencillo y no incluye ningún ciclo o llamadas a otras funciones que puedan tener ciclos, la complejidad del tiempo es de `O(n)`, lo cual hace que el filtrado sea bastante rápido.

Usos comunes de `filter` es limpiar un `Stream` de datos que no cumplan un cierto criterio. Como ejemplo podrías pensar en un `Stream` de transacciones bancarias, mantener el `Stream` solo aquellas que superen un cierto monto para mandarlas a auditoria, de un grupo de calificaciones de alumnos filtrar únicamente por aquellos que aprobaron con una calificación superior a 6, de un grupo de objetos `JSON` conservar aquellos que tengan una propiedad en especifico, etc.

Entre mas sencilla sea la condición de filtrado, más legible sera el código. Te recomiendo que, si tienes más de una condición de filtrado, no le temas a usar varias veces `filter`:

```
courses.filter(course -> course.getName().contains("Java"))
    .filter(course -> course.getDuration() > 2.5)
    .filter(course -> course.getInstructor().getName() ==
Instructors.SINUHE_JAIME)
```

Tu código sera más legible y las razones de por qué estás aplicando cada filtro tendrán más sentido. Como algo adicional podrías mover esta lógica a funciones individuales en caso de que quieras hacer más legible el código, tener más facilidad de escribir pruebas o utilices en más de un lugar la misma lógica para algunas lambdas:

```
courses.filter(Predicates::isAJavaCourse)
    .filter(Predicates::hasEnoughDuration)
    .filter(Predicates::hasSinuheAsInstructor);
```

```
// La lógica es la misma:
public final class Predicates {
    public static final boolean isAJavaCourse(Course course){
        return course.getName().contains("Java");
    }
}
```

map

La operación `map` puede parecer complicada en un principio e incluso confusa si estas acostumbrado a usar `Map`, pero cabe resaltar que no hay relación entre la estructura y la operación. La operación es meramente una transformación de un tipo a otro.

```
Stream map(Functionsuper T, ? extends R> mapper)
```

Los detalles a resaltar son muy similares a los de `filter`, pero la diferencia clave está en `T` y `R`. Estos *generics* nos dicen que `map` va a tomar un tipo de dato `T`, cualquiera que sea, le aplicara la función `mapper` y generara un `R`.

Es algo similar a lo que hacías en la secundaria al convertir en una tabla datos, para cada `x` aplicabas una operación y obtenías una `y` (algunos llaman a esto *tabular*). `map` operará sobre cada elemento en el `Stream` inicial aplicando la `Function` que le pases como lambda para generar un nuevo elemento y hacerlo parte del `Stream`resultante:

```
Stream ids = DatabaseUtils.getIds().stream();

Stream users = ids.map(id -> db.getUserWithId(id));
```

O, puesto de otra forma, por cada `DatabaseID` en el `Stream` inicial, al aplicar `map`genera un `User`:

- DatabaseID(1234) -> `map(…)` -> User(Sinuhe Jaime, @Sierisimo)

- DatabaseID(4321) -> `map(…)` -> User(Diego de Granda, @degranda10)
- DatabaseID(007) -> `map(…)` ->User(Oscar Barajas, @gndx)

Esto resulta bastante practico cuando queremos hacer alguna conversión de datos y realmente no nos interesa el dato completo (solo partes de él) o si queremos convertir a un dato complejo partiendo de un dato base.

Si quisiéramos replicar qué hace internamente `map` sería relativamente sencillo:

```
public  Stream filter(Function mapper) {
    List mappedData = new LinkedList<>();
    for(T t : this.data){
        R r = mapper.apply(t);
        mappedData.add(r);
    }

    return mappedData.stream();
}
```

La operación `map` parece simple ya vista de esta manera. Sin embargo, por dentro de las diferentes implementaciones de `Stream` hace varias validaciones y optimizaciones para que la operación pueda ser invocada en paralelo, para prevenir algunos errores de conversión de tipos y hacer que sea mas rápida que nuestra versión con un `for`.

flatMap

En ocasiones no podremos evitar encontrarnos con *streams* del tipo `Stream>`, donde tenemos datos con muchos datos…

Este tipo de *streams* es bastante común y puede pasarte por multiples motivos. Se puede tornar difícil operar el `Stream` inicial si queremos aplicar alguna operación a cada uno de los elementos en cada una de las listas.

Si mantener la estructura de las listas (o colecciones) no es importante para el procesamiento de los datos que contengan, entonces podemos usar `flatMap` para simplificar la estructura del `Stream`, pasándolo de `Stream>` a `Stream`.

Visto en un ejemplo más “visual”:

```
Stream> coursesLists; // Stream{List["Java", "Java 8 Functional", "Spring"],
List["React", "Angular", "Vue.js"], List["Big Data", "Pandas"]}
Stream allCourses; // Stream{ ["Java", "Java 8 Functional", "Spring",
"React", "Angular", "Vue.js", "Big Data", "Pandas"]}
```

`flatMap` tiene la siguiente forma:

```
Stream flatMap(Functionsuper T, ? extends Stream> mapper)
```

Lo interesante es que el resultado de la función `mapper` debe ser un `Stream`. `Stream` usará el resultado de `mapper` para “acumular” elementos en un `Stream`desde otro `Stream`. Puede sonar confuso, por lo que ejemplificarlo nos ayudará a entenderlo mejor:

```
//Tenemos esta clase:
public class PlatziStudent {
    private boolean emailSubscribed;
    private List emails;

    public boolean isEmailSubscribed() {
        return emailSubscribed;
    }

    public List getEmails(){
        return new LinkedList<>(emails); //Creamos una copia de la lista para
mantener la clase inmutable por seguridad
    }
}

//Primero obtenemos objetos de tipo usuario registrados en Platzi:
Stream platziStudents = getPlatziUsers().stream();
```

```
// Despues, queremos enviarle un correo a todos los usuarios pero… solo nos
interesa obtener su correo para notificarlos:
Stream allEmailsToNotify =

platziStudents.filter(PlatziStudent::isEmailSubscribed) //Primero evitamos
enviar correos a quienes no estén subscritos
                                    .flatMap(student ->
student.getEmails().stream()); // La lambda genera un nuevo Stream de la
lista de emails de cada studiante.

sendMonthlyEmails(allEmailsToNotify);
//El Stream final solo es un Stream de emails, sin mas detalles ni
información adicional.
```

`flatMap` es una manera en la que podemos depurar datos de información adicional que no sea necesaria.

distinct

Esta operación es simple:

```
Stream distinct()
```

Lo que hace es comparar cada elemento del `Stream` contra el resto usando el método `equals`. De esta manera, evita que el `Stream` contenga elementos duplicados. La operación, al ser intermedia, retorna un nuevo `Stream` donde los elementos son únicos. Recuerda que para garantizar esto es recomendable que sobrescribas el método `equals` en tus clases que representen datos.

limit

La operación `limit` recibe un `long` que determina cuántos elementos del `Stream`original seran preservados. Si el número es mayor a la cantidad inicial de elementos en el `Stream`, básicamente, todos los elementos seguirán en el `Stream`. Un detalle interesante es que algunas implementaciones

de `Stream` pueden estar ordenadas (`sorted()`) o explícitamente no ordenadas (`unordered()`), lo que puede cambiar drásticamente el resultado de la operación y el performance.

```
Stream limit(long maxSize)
```

La operación asegura que los elementos en el `Stream` resultante serán los primeros en aparecer en el `Stream`. Es por ello que la operación es ligera cuando el `Stream` es *secuencial* o se usó la operación `unordered()` (no disponible en todos los `Streams`, ya que la operación pertenece a otra clase).

Como reto adicional, crea el código para representar lo que hace la operación `limit`.

peek

`peek` funciona como una lupa, como un momento de observación de lo que está pasando en el `Stream`. Lo que hace esta operación es tomar un `Consumer`, pasar los datos conforme van estando presentes en el `Stream` y generar un nuevo `Stream`idéntico para poder seguir operando.

La estructura de `peek` es simple:

```
Stream peek(Consumersuper T> consumer)
```

Usarlo puede ayudarnos a generar logs o registros de los datos del `Stream`, por ejemplo:

```
Stream serverConnections =
    server.getConnectionsStream()
        .peek(connection -> logConnection(connection, new Date()))
        .filter(…)
```

```
        .map(…)
    //Otras operaciones…
```

skip

Esta operación es contraria a `limit()`. Mientras `limit()` reduce los elementos presentes en el `Stream` a un numero especifico, `skip` descarta los primeros *n*elementos y genera un `Stream` con los elementos restantes en el `Stream`.

Esto es:

```
Stream first10Numbers = Stream.of(0,1,2,3,4,5,6,7,8,9);
Stream last7Numbers = first10Numbers.skip(3); // 3,4,5,6,7,8,9
```

Esto puede ser de utilidad si sabemos qué parte de la información puede ser descartable. Por ejemplo, descartar la primera línea de un XML ( ), descartar metadatos de una foto, usuarios de prueba al inicio de una base de datos, el intro de un video, etc.

sorted

La operación `sorted()` requiere que los elementos presentes en el `Stream`implementen la interfaz `Comparable` para poder hacer un ordenamiento de manera natural dentro del `Stream`. El `Stream` resultante contiene todos los elementos pero ya ordenados, hacer un ordenamiento tiene muchas ventajas, te recomiendo los cursos de algoritmos de Platzi para poder conocer a mas detalle estas ventajas.

Conclusiones

Las operaciones intermedias nos permiten tener control sobre los streams y manipular sus contenidos de manera sencilla sin preocuparnos realmente por

cómo se realizan los cambios.

Recuerda que las operaciones intermedias tienen la funcionalidad de generar nuevos streams que podremos dar como resultado para que otras partes del código los puedan utilizar.

Aunque existen otras operaciones intermedias en diferentes implementaciones de `Stream`, las que aquí listamos están presentes en la interfaz base, por lo que entender estas operaciones te facilitara la vida en la mayoría de los usos de `Stream`.