Java 8 Funcional: Operaciones y Collectors

Lambdas, operaciones y retornos

Usando `Stream` nos podemos simplificar algunas operaciones, como es el filtrado, el mapeo, conversiones y más. Sin embargo, no es del todo claro cuaádo una operación nos devuelve otro `Stream` para trabajar y cuándo nos da un resultado final…

¡O al menos no era claro hasta ahora!

Cuando hablamos de pasar lambdas a una operación de `Stream`, en realidad, estamos delegando a Java la creación de un objecto basado en una interfaz.

Por ejemplo:

```
Stream<String> coursesStream = Utils.getListOf("Java", "Node.js",
"Kotlin").stream();

Stream<String> javaCoursesStream = coursesStream.filter(course ->
course.contains("Java"));


// En realidad, es lo mismo que:

Stream<String> explicitOperationStream = coursesStream.filter(new
Predicate<String>() {
    public boolean test(String st) {
        return st.contains("Java");
    }
});
```

Estas interfaces las mencionamos en clases anteriores. Solo como repaso, listo algunas a continuación:

- `Consumer<T>`: recibe un dato de tipo `T` y no genera ningún resultado

- `Function<T,R>`: toma un dato de tipo `T` y genera un resultado de tipo `R`
- `Predicate<T>`: toma un dato de tipo `T` y evalúa si el dato cumple una condición
- `Supplier<T>`: no recibe ningún dato, pero genera un dato de tipo `T` cada vez que es invocado
- `UnaryOperator<T>` recibe un dato de tipo `T` y genera un resultado de tipo `T`

Estas interfaces (y otras más) sirven como la base de donde generar los objetos con las lambdas que pasamos a los diferentes métodos de `Stream`. Cada una de ellas cumple esencialmente con recibir el tipo de dato de el `Stream` y generar el tipo de retorno que el método espera.

Si tuvieras tu propia implementación de `Stream`, se vería similar al siguiente ejemplo:

```
public class PlatziStream<T> implements Stream {
    private List<T> data;

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

        return filteredData.stream();
    }
}
```

Probablemente, tendría otros métodos y estructuras de datos, pero la parte que importa es justamente cómo se usa el `Predicate`. Lo que hace `Stream`internamente es pasar cada dato por este objeto que nosotros proveemos como una lambda y, según el resultado de la operación, decidir si debe incluirse o no en el `Stream` resultante.

Como puedes notar, esto no tiene mucha complejidad, puesto que es algo que pudimos fácilmente replicar. Pero `Stream` no solo incluye estas operaciones "triviales", también incluye un montón de utilidades para que la máquina virtual de Java pueda operar los elementos de un `Stream` de manera más rápida y distribuida.

Operaciones

A estas funciones que reciben lambdas y se encargan de trabajar (operar) sobre los datos de un `Stream` generalmente se les conoce como Operaciones.

Existen dos tipos de operaciones: **intermedias** y **finales**.

Cada operación aplicada a un `Stream` hace que el `Stream` original ya no sea usable para más operaciones. Es importante recordar esto, pues tratar de agregar operaciones a un `Stream` que ya esta siendo procesado es un error muy común.

En este punto seguramente te parezcan familiares todas estas operaciones, pues vienen en forma de métodos de la interfaz `Stream`. Y es cierto. Aunque son métodos, se les considera operaciones, puesto que su intención es operar el `Stream` y, posterior a su trabajo, el `Stream` no puede volver a ser operado.

En clases posteriores hablaremos más a detalle sobre cómo identificar una **operación terminal** de una **operación intermedia**.

Collectors

Una vez que has agregado operaciones a tu `Stream` de datos, lo más usual es que llegues a un punto donde ya no puedas trabajar con un `Stream` y necesites enviar tus datos en otro formato, por ejemplo, `JSON` o una `List` a base de datos.

Existe una interfaz única que combina todas las interfaces antes mencionadas y que tiene como única utilidad proveer de una operación para obtener todos los elementos de un `Stream`: `Collector`.

`Collector<T, A, R>` es una interfaz que tomará datos de tipo `T` del `Stream`, un tipo de dato mutable `A`, donde se iran agregando los elementos (mutable implica que podemos cambiar su contenido, como un `LinkedList`), y generara un resultado de tipo `R`.

Suena complicado… y lo es. Por eso mismo, Java 8 incluye una serie de *Collectors*ya definidos para no rompernos las cabeza con cómo convertir nuestros datos.

Veamos un ejemplo:

```
public List<String> getJavaCourses(Stream<String> coursesStream) {
    List<String> javaCourses =
        coursesStream.filter(course -> course.contains("Java"))
            .collect(Collectors.toList());

    return javaCourses;
}
```

Usando `java.util.stream.Collectors` podemos convertir muy sencillamente un `Stream` en un `Set`, `Map`, `List`, `Collection`, etc. La clase `Collectors` ya cuenta con métodos para generar un `Collector` que corresponda con el tipo de dato que tu `Stream` está usando. Incluso vale la pena resaltar que `Collectors` puede generar un `ConcurrentMap` que puede ser de utilidad si requieres de multiples threads.

Usar `Collectors.toXXX` es el proceso inverso de usar `Collection.stream()`. Esto hace que sea fácil generar APIs publicas que trabajen con

estructuras/colecciones comunes e internamente utilizar `Stream` para agilizar las operaciones de nuestro lado.

Tipos de retorno

Hasta este punto, la única manera de obtener un dato que ya no sea un `Stream` es usando `Collectors`, pues la mayoría de operaciones de `Stream` se enfocan en operar los datos del `Stream` y generar un nuevo `Stream` con los resultados de la operación.

Sin embargo, algunas operaciones no cuentan con un retorno. Por ejemplo, `forEach`, que es una operación que no genera ningún dato. Para poder entender qué hace cada operación basta con plantear qué hace la operación para poder entender qué puede o no retornar.

Por ejemplo:

> La operación de `findAny` trata de encontrar cualquier elemento que cumpla con la condición del `Predicate` que le pasamos como parámetro. Sin embargo, la operación dice que es devuelve un `Optional`. ¿Qué pasa cuando no encuentra ningún elemento? ¡Claro, por eso devuelve un `Optional`! Porque podría haber casos en que ningún elemento del `Stream`cumpla la condición.

En las clases posteriores haremos un listado más a detalle y con explicaciones de qué tipos de retorno tiene cada operación. Y entenderemos por qué se categorizan como **operaciones finales e intermedias**.

Conclusiones

Por ahora, hemos entendido que cada operación en un `Stream` consume hasta agotar el `Stream`. Y lo hace en un objeto no reusable. Esto implica que tenemos que decidir en nuestro código cuándo un `Stream` es un elemento temporal para una función o cuándo realmente una función sera la última en tocar los datos del `Stream`.