STREAM

¿Qué es?

Un stream es una interfaz de java que nos permite concatenar distintos métodos que están dentro de esta interfaz, permitiendo realizar operaciones ahorrando una gran cantidad de código. Facilita el trabajo con colecciones y nos permite transformar y manipular los datos de estas.

Es una interfaz la cual no es necesario implementar en nuestra clase, podemos llamar a sus métodos en cualquier clase y cualquier punto de nuestro código, con una sola condición, un stream debe aplicarse a un objeto. Encontramos dos tipos de stream, los paralelos y los secuenciales.

Los stream están compuestos por :

- **Fuente**

- **Operaciones intermedia**

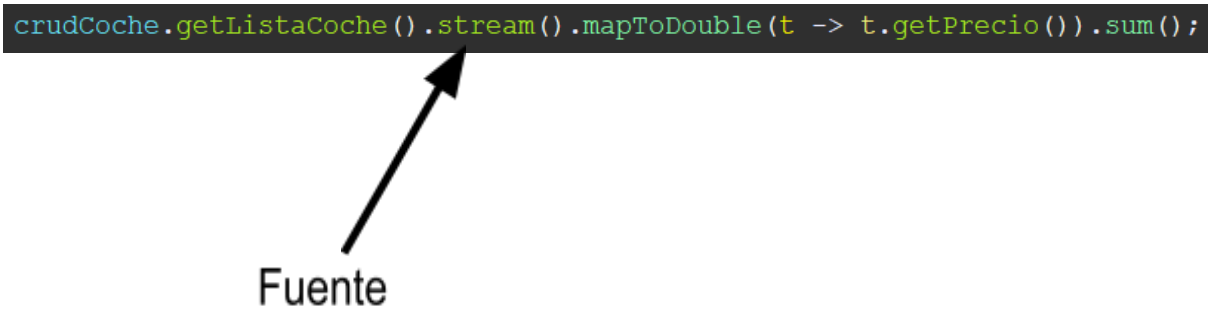
- **Operaciones terminales**

FUENTE

¿Qué es?

La fuente es la “llamada” al stream, al escribirlo detrás de nuestro objeto podremos comenzar a operar con ellos. Una fuente puede ser **.stream()** para trabajar con stream secuenciales o .**parallelStream()**, para trabajar con stream paralelos.

En resumen, las fuentes nos permiten elegir de qué manera vamos a operar con los datos de nuestro stream, pudiendo escoger unas u otras en función de nuestras necesidades.

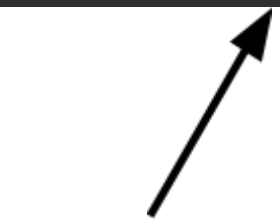


OPERACIONES INTERMEDIAS:

¿Qué son?

Cuando hablamos de las operaciones intermedias de un stream nos referimos a aquellas que se aplican sobre la fuente de datos del stream. Estas operaciones no serán ejecutadas hasta que el stream incluya una operación terminal. Se encargan de crear un nuevo stream con el resultado de estas operaciones unidas con el de la terminal. Estas operaciones no son obligatorias, es decir, pueden o no aparecer en el stream, al igual que un stream puede contener una o más operaciones intermedias.

```
crudCoche.getListaCoche().stream().mapToDouble(t -> t.getPrecio()).sum();
```



Operación intermedia

Como en cada elemento del API de java, existen muchas operaciones intermedias, algunos ejemplos son los siguientes:

- Filter :

Filter nos permite filtrar objetos de una colección a partir de un criterio que debemos definir. Este método nos devuelve un Stream <Object> con los elementos que cumplan el requisito que hayamos impuesto. Realiza una función parecida al método buscar que estamos acostumbrados.

```
listaCoche.stream().filter(coche -> coche.getModelo().equalsIgnoreCase(modelo)).toList()
```

En este caso, el requisito dado es que filtre una lista de coches, y que sólo devuelva aquellos que son iguales que un modelo dado por parámetros.

- Limit:

Limit es un método que se usa de la mano con filter. Permite limitar la cantidad de resultados dados por el filter a N números. Los resultados variarán en función de la colección usada, es decir, para un List devolverá los N elementos en orden de inserción, para un Set devolverá los N primeros elementos que encuentre.

El resultado de limit será el mismo que el de Filter, un Stream <Object>.

```
listaCoche.stream().filter(coche -> coche.getModelo().equalsIgnoreCase(modelo)).limit(limit).toList();
```

En este caso, el filtrado es similar al anterior, pero el resultado será limitado a un número dado como parámetro.

- Skip:

Skip realiza una función similar a limit, con la diferencia de que este se encarga de mostrar todos los resultados del filtrado, excepto una cantidad N, que omitirá. Al igual que limit, la forma en que salta esos elementos depende de la colección usada.

Al igual que los anteriores, devuelve un Stream<Objetc> con los resultados de la búsqueda.

```
listaCoche.stream().filter(coche -> coche.getModelo().equalsIgnoreCase(modelo)).skip(skip).toList()
```

En este caso, la búsqueda salta N cantidad de elementos de la lista filtrada, pasando esa cantidad como un parámetro.

- Sorted:

Sorted es un método que se encarga de ordenar nuestro stream siguiendo unos criterios deseados, usando las interfaces Comparator y Comparable. Puede usarse también con una lista filtrada, siendo muy útil.

Al igual que los métodos anteriores, devuelve un Stream <Object>

```
Comparator<Coche> compararPrecios = Comparator.comparing(Coche::getPrecio);

return crudCoche.getListaCoche().stream().sorted(compararPrecios).toList();
```

Este ejemplo, ordena una lista de coches por su precio, habiendo definido la forma de ordenarlo anteriormente.

- Map:

Map nos permite realizar una transformación de los datos del stream a un tipo en concreto, permitiéndonos operar con ellos. Map es un método que solo permite una entrada y devuelve una única salida. También permite implementar unas operaciones terminales concretas, que explicaremos más adelante.

Map devuelve un Stream <Object>

Existen diferentes tipos de map. MapToInt y MapToDouble.

MapToInt nos permite operar con datos de tipo int, devolviendo como resultado de la operación un IntStream.

MapToDouble permite operar con datos de tipo double, devolviendo como resultado un DoubleStream. En caso de que los datos fueran una mezcla de int y double y el resultado pudiera ser ambiguo entre ambas, la operación devolverá un dato tipo optionalDouble.

```
crudCoche.getListaCoche().stream().mapToDouble(t -> t.getPrecio()).sum();
```

Este ejemplo calcula el total del precio de los coches de una lista de coches usando mapToDouble.

- FlatMap:

Tiene una función parecida a Map, con la diferencia de que flatMap acepta una sola entrada, pero devuelve más de una salida, es decir, permite concatenar más de un stream para operar con ambos a conjunto.

FlatMap devuelve un Stream <Object>

Al igual que map, cuenta con flatMapToInt y flatMapToDouble.

flatMapToInt nos permite operar con datos de tipo int, devolviendo como resultado de la operación un IntStream.

flatMapToDouble permite operar con datos de tipo double, devolviendo como resultado un DoubleStream. En caso de que los datos fueran una mezcla de int y double y el resultado pudiera ser ambiguo entre ambas, la operación devolverá un dato tipo optionalDouble.

```
crudCoche.getListaCoche().stream().map(coche -> coche.getListaExtra()).flatMap(vi -> vi.stream())
        .forEach((Consumer<? super Extras>) new Consumer<Extras>() {
            public void accept(Extras e) {
                System.out.println(e.getTipoExtras());
            }
        });
```

En este caso, primero usamos un map, que se encarga de darnos la lista de Extras que tiene un coche como atributo. Esta lista la usaremos para conocer cuales son los extras que tiene cada coche en particular.

Operaciones terminales

¿Qué son?

Son las encargadas de devolver el resultado y son las que inician el stream. Tiene que haber una operación terminal obligatoriamente para que stream devuelva un resultado.

```
return crudCoche.buscarVendidos().stream().count();
```

- FindFirst

Este método es un <Optional> nos devuelve el primer resultado que encuentre en un stream, si no tiene orden puede devolver cualquier elemento. Lo que nos devuelve es un <Optional> que es un elemento del stream y si está vacío un empty <Optional>.

```
return listaCoche.stream().filter(coche -> coche.getNumBastidor().equalsIgnoreCase(numBastidor)).findFirst()
        .get();
```

- FindAny

Este método es un <Optional>similar al FindFirst, busca cualquier elemento del stream y si se utiliza varias veces puede devolver diferentes resultados.Lo que nos devuelve es un <Optional>que es un elemento del stream y si está vacío un empty <Optional>.

```
return listaCoche.stream().filter(coche -> coche.getNumBastidor().equalsIgnoreCase(numBastidor)).findAny()
        .get();
```

- toArray(toList)

Este método es un array de objetos y devuelve un array con los elementos del stream, el toList es similar al toArray pero en vez de devolver un array devuelve un <List>.

```
return listaCoche.stream().filter(coche -> coche.getModelo().equalsIgnoreCase(modelo)).skip(skip).toList();
```

- count

Este método es de tipo long y sirve para contar los elementos del stream, no hay que darle ningún parámetro y devuelve long.

```
return crudCoche.buscarVendidos().stream().count();
```

- average

Este método es un <OptionalDouble> que hace la media aritmética de los elementos del stream devuelve un <OptionalDouble> con la media si hay elementos en el stream y si está vacío un empty Optional.

```
return crudCoche.buscarPorMarca(marca).stream().mapToDouble(t -> t.getPrecio()).average().getAsDouble();
```

- forEach

Este método hace la misma función que un for each sobre el stream que se ejecuta.

```
crudCoche.getListaCoche().stream().map(coche -> coche.getListaExtra()).flatMap(vi -> vi.stream())
        .forEach((Consumer<? super Extras>) new Consumer<Extras>() {
            public void accept(Extras e) {
                System.out.println(e.getTipoExtras());
            }
        });
```

- sum

Este método devuelve la suma de los elementos del stream.

```
return crudCoche.getListaCoche().stream().mapToDouble(t -> t.getPrecio()).sum();
```

- collect

Este método puede volver el stream en list o set y podemos hacer funciones más complejas que con el toArray o toList usando una interfaz que implementa este método que es Collectors

```
return listaCoche.stream().filter(coche -> coche.getModelo().equalsIgnoreCase(modelo)).skip(skip)
        .collect(Collectors.toList());
```