STREAM

MÉTODOS TERMINALES

ÍNDICE

Realizado por: Cristina Rus Morales

1. Introducción

1.1. Definición

Un Stream es una librería en Java, la cual se introdujo en Java 8, sobre el año 2014. Se añadió al paquete java.util ya que es el que contiene los frameworks de colecciones, nombrándolo como **java.util.stream**; en la cual, se encuentra la Interfaz Stream, con los métodos intermedios (se puede ver su explicación en el proyecto de Hugo) y los terminales, en estos últimos son en los que nos vamos a centrar en el proyecto; además, son las que producen la ejecución de los métodos intermedios.

Esta librería nos facilita el manejo de las colecciones. Su funcionamiento es relativamente sencillo: funcionan como una **tubería por la que pasan datos** y, cuando pasan por esta, se le pueden realizar a estos datos distintas operaciones.

Al no almacenar los datos en la memoria (a diferencia de las listas y arrays), su procesamiento es mucho **más eficiente** y **veloz**. No es necesario indicar el método de recorrer la colección, por lo que el código queda mucho más claro y limpio.

Existen dos tipos de Stream: el **secuencial** y el **paralelo**. Se podría decir que el secuencial es el Stream "normal", este sólo utiliza un núcleo del procesador, mientras que el paralelo utiliza todos los disponibles que tenga. El uso de los Streams paralelos está recomendado para cuando se tienen que procesar una gran cantidad de datos.

En este proyecto nos centraremos en los secuenciales, aunque haremos alguna mención a los paralelos.

1.2. Métodos terminales y su diferencia con intermedios

A diferencia de los métodos intermedios, los finales no pueden ser encadenados y tampoco devuelve un Stream, sino una colección, un valor primitivo o un objeto. Estos **finalizan el procesamiento de los datos** y siempre son necesarios para terminar un Stream.

1.3. Interfaces relacionadas

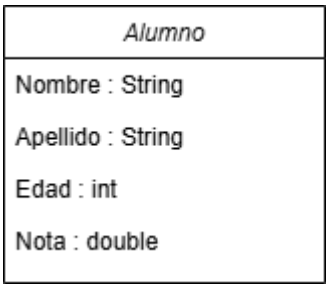
- **AutoCloseable**: Superinterfaz de Stream, permite que se cierren automáticamente.
- **BaseStream**: Superinterfaz de Stream y de los que hay a continuación. Les proporciona todos los métodos que tienen en común.
- **IntStream**: Es un tipo de Stream que se especializa en el valor primitivo int.
- **LongStream**: Es un tipo de Stream que se especializa en el valor primitivo long.
- **DoubleStream** :Es un tipo de Stream que se especializa en el valor primitivo double.

A continuación nombraremos las interfaces que se nombran en el proyecto:

- **Interfaces funcionales**:
  - Consumer, BinaryOperator, BiFunction, IntFunction, Predicate, Supplier y BiConsumer.
- **Otras**:
  - Comparator, Optional y Collector.

2. Métodos terminales

En la mayoría de métodos se utilizan diversas interfaces funcionales dentro de los parámetros, lambdas en su ejecución y referencias a métodos, estas cosas se explican con detalle en los proyectos de Pablo y Gonzalo.Explicaremos los métodos basándonos en los que hay en la versión 11 de java.
Para los ejemplos de los métodos, se va a utilizar la siguiente clase de ayuda:



La estructura de los métodos es la siguiente:

colección.stream().método();

2.1. Métodos forEach()

El parámetro que tienen estos dos métodos es una interfaz funcional(explicada por Pablo y Gonzalo) llamada Consumer, que en este caso indica la acción que queremos que se le haga a la colección.

2.1.1. void **forEach**(Consumer<? super T> action)

Este método actúa como un **forEach** normal; sin tener que usarlo de su forma tradicional. Además, no siguen ningún orden en específico; no es recomendable su uso cuando se quieren modificar los elementos del Stream, solo para consumir datos; es decir, para realizar una acción sobre los elementos que no los modifique.
A continuación mostramos un ejemplo:

```
Alumno a=new Alumno("Pepito","Pérez",9.5,20);
Alumno a1=new Alumno("Sara","Martinez",6.8,18);
Alumno a2=new Alumno("Marcos","Díaz",7.3,19);
Alumno a3=new Alumno("Miguel","García",9.1,19);
```

```
        List<Alumno> listado=new ArrayList<>();
        listado.add(a);
        listado.add(a1);
        listado.add(a2);
        listado.add(a3);

//Se llama a la colección, en este caso es listado y al stream con el método forEach:
        listado.stream()
        .forEach((alum)->{
            System.out.println("Nombre: "+alum.getNombre()+"\n"
                    + "Apellido: "+alum.getApellido()+"\n"
                    + "Edad: "+alum.getEdad()+"\n"
                    + "Nota: "+alum.getNota());
            System.out.println();
        });

        //En el interior del forEach, encontramos una expresión lambda que,
        //en este caso, coge un alumno y va imprimiendo lo que hemos indicado
        //hasta que termine de recorrer toda la colección.
        //Para usar un Stream paralelo, se usaría lo siguiente:
        //listado.parallelStream().forEach();
```

Salida resultante:

```
Nombre: Pepito
Apellido: Pérez
Edad: 20
Nota: 9.5

Nombre: Sara
Apellido: Martinez
Edad: 18
Nota: 6.8

Nombre: Marcos
Apellido: Díaz
Edad: 19
Nota: 7.3

Nombre: Miguel
Apellido: García
Edad: 19
Nota: 9.1
```

En este caso, la salida se ha mostrado de la misma forma en la que hemos introducido los objetos en la lista, por lo que podríamos decir que ha salido de forma "*ordenada*"; sin embargo, esto no es verídico, ya que podrían no haber salido así los datos. Esta forma es respetada al ser un **Stream secuencial**, por lo que todo pasa por un solo núcleo y no da posibilidad de desorden. Sin embargo, si fuera un **Stream paralelo**, la salida aparecería sin un orden en concreto al no respetar el flujo original. Si queremos que respete el flujo original, es necesario usar el siguiente método (forEachOrdered()).

2.1.2. void **forEachOrdered**(Consumer<? super T> action)

Este método tiene la misma función que el anterior y no es necesario modificar nada del interior de los parámetros para que realice lo mismo; su estructura es la siguiente:
A continuación mostramos un ejemplo:

```
listado.stream()
.forEachOrdered((alum)->{
        System.out.println("Nombre: "+alum.getNombre()+"\n"
                        + "Apellido: "+alum.getApellido()+"\n"
                        + "Edad: "+alum.getEdad()+"\n"
                        + "Nota: "+alum.getNota());
        System.out.println();
});
//Para usar un Stream paralelo, se usaría lo siguiente:
//listado.parallelStream().forEachOrdered();
```

Salida resultante:

```
Nombre: Pepito
Apellido: Pérez
Edad: 20
Nota: 9.5

Nombre: Sara
Apellido: Martinez
Edad: 18
Nota: 6.8

Nombre: Marcos
Apellido: Díaz
Edad: 19
Nota: 7.3

Nombre: Miguel
Apellido: García
```

Edad: 19
Nota: 9.1

Aunque este método y el forEach() tengan la misma salida, no significa que sean iguales. Como hemos comentado anteriormente, al usar un forEachOrdered() obliga al Stream a **respetar el flujo original**; no es necesario usarlo con Stream secuenciales porque al usar un solo núcleo todos los datos van a salir en el orden original. Se utilizan con Stream paralelos para garantizar que los datos se procesen con el flujo original, haciendo así que se impriman en el mismo orden que el método anterior.

2.2. Métodos reduce()

La función general de este método, tal y como su nombre indica, es reducir una serie de datos en uno solo. Este método no es óptimo cuando se quiere sacar de una lista un valor que puede estar repetido(una nota más alta, la edad más baja…) ya que acaba guardando sólo el primero.

Este método está **sobrecargado** y puede tener los diferentes parámetros:

- **Identity**: Este es el que indica el valor inicial que se le da a la operación, también aparece si no ha habido métodos intermedios previos.
- **Accumulator**: *BinaryOperator* de la interfaz funcional *BiFuntion*, en este caso recoge dos parámetros que dan un resultado parcial de la operación hasta que esta termina.
- **Combiner**: Este une el Accumulator cuando el Stream es paralelo o cuando el tipo de los parámetros del Accumulator no son del mismo tipo.

2.2.1. T **reduce**(T identity, BinaryOperator<T> accumulator)

Este método devuelve T, que es cualquier tipo de objeto (aunque no puede devolver un *Optional* porque el identity siempre garantiza un valor aún si la lista está vacía)
A continuación mostramos un ejemplo:

```
double media;
double sumaNotas = listado.stream()
.map(alum -> alum.getNota())
.reduce(0.0, Double::sum);

media= sumaNotas/listado.size();

 System.out.printf("La media de las notas es: %.2f", media);
```

Salida resultante:

La media de las notas es: 7,75

En este ejemplo estamos utilizando un **método intermedio** (.map) que nos cambia el Stream para que sea de las notas de los alumnos. A continuación, se usa el reduce() y usamos como accumulator una referencia a método que, en este caso, está sumando las notas.

2.2.2. Optional<T> **reduce**(BinaryOperator<T> accumulator)

Este método realiza la misma operación que el anterior; sin embargo, este devuelve un *Optional* (explicado en los trabajos de Mauro y Germán) y **no tiene un identity** porque no tiene un valor predeterminado para empezar.
A continuación mostramos un ejemplo:

```
Optional<Alumno> alumnoNombreCorto = listado.stream() .reduce((alum1, alum2) -> alum1.getNombre().length() <= alum2.getNombre().length() ? alum1 : alum2);

    alumnoNombreCorto.ifPresentOrElse(
    alum -> System.out.println("El alumno con el nombre más corto es: " + alum),
    () -> System.out.println("No hay alumnos en el listado.")
  );
```

Salida resultante:

El alumno con el nombre más corto es: Alumno [nombre=Sara, apellido=Martinez, nota=6.8, edad=18]

En este ejemplo hemos creado la variable "*alumnoNombreCorto*" la cual es un *Optional*. El Stream compara los carácteres de dos alumnos y guarda el más corto; realiza esta operación hasta que termina de recorrer la lista. Luego, hemos utilizado el método de *Optional* llamado .ifPresentOrElse() para que nos imprima una cosa u otra dependiendo de si el *Optional* está vacío o no.

2.2.3. <U> U **reduce**(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)

Este método reduce() utiliza un combiner, a diferencia de los dos métodos anteriores. Puede usarse en dos situaciones distintas, cuando los parámetros del accumulator son distintos y cuando se utiliza un Stream en paralelo.
A continuación mostramos un ejemplo:

```
int sumaEdad= listado.stream().reduce(0, (edad,alum)-> edad+ alum.getEdad(),Integer::sum);
int edadMedia=sumaEdad/listado.size();
```

```
System.out.println("La edad media es de "+edadMedia+" años.");
//Para usar un Stream paralelo, se usaría lo siguiente:
//listado.parallelStream().reduce();
```

Salida resultante:

La edad media es de 19 años.

En este ejemplo al accumulator le hemos dado dos parámetros: "edad" y "alum" y, si lo quisiéramos usar como el primer tipo de reduce() que no tiene el combiner **daría error,** ya que edad es una variable tipo int y alum es una tipo Alumno (se ve al indicar "alum.getEdad()"); por ello, es **necesario usar el combiner** ya que nos ayuda a indicar explícitamente que queremos que se sumen los dos datos("*Integer::sum*").
Sería conveniente usar un Stream en paralelo si tuviéramos una gran cantidad de alumnos.

2.3. Métodos toArray()

Estos métodos, como su propio nombre indica, devuelve un Array con los elementos del Stream.

2.3.1. Object[] **toArray**()

Este método devuelve un Array tipo *Object*, lo cual no es lo que normalmente se busca.

2.3.2. <A> A[] **toArray**(IntFunction<A[]> generator)

Este método devuelve un Array del tipo indicado en los parámetros.
A continuación mostramos un ejemplo:

```
String[] nombres=listado.stream()
.map(alum -> alum.getNombre())
.toArray(String[]::new);

System.out.println(Arrays.toString(nombres));
```

Salida resultante:

[Pepito, Sara, Marcos, Miguel]

En este ejemplo, hemos especificado mediante una referencia a método, que queremos un Array tipo *String*. Nos devuelve los nombres de los alumnos.

2.4. Métodos de consulta

2.4.1. Optional<T> **min**(Comparator<? super T> comparator)

Este método busca entre los elementos del Stream el que sea **menor**, en referencia al comparador introducido como parámetro. Devuelve un *Optional* porque la colección podría estar vacía.
A continuación mostramos un ejemplo:

```
Optional<Alumno> alumnoMasJoven = listado.stream()
        .min(Comparator.comparingInt(Alumno::getEdad));


    alumnoMasJoven.ifPresentOrElse(
                alum -> System.out.println("El alumno con el nombre más joven es: " +
alum),
                () -> System.out.println("No hay alumnos en el listado.")
            );
```

Salida resultante:

```
El alumno con el nombre más joven es: Alumno [nombre=Sara, apellido=Martinez, nota=6.8,
edad=18]
```

En este método es **necesario** llamar a la interfaz *Comparator* ya que necesita como parámetro un *Comparator*; a través de este podemos acceder a su método "comparingInt" porque en este caso necesitamos comparar las edades de los alumnos. Hemos utilizado a continuación como parámetro una referencia al método "getEdad" de la clase Alumno para comparar las edades.
A continuación, hemos usado el mismo método que anteriormente para ver si el *Optional* está vacío o no.

2.4.2. Optional<T> **max**(Comparator<? super T> comparator)

Este método es exactamente igual que el anterior, en vez de devolver el mínimo devuelve el **máximo**.

2.4.3. boolean **allMatch**(Predicate<? super T> predicate)

Este método recibe una condición como predicate y devuelve un true o false **si se cumple la condición para todos los elementos del stream**. Si el Stream se encuentra vacío, este método devolverá un true.
A continuación mostramos un ejemplo:

```
        boolean presente=listado.stream().allMatch(alum -> alum.getEdad()>=18);

        System.out.println(presente);
```

Salida resultante:

true

En este ejemplo estamos comprobando si todos los Alumnos del listado son mayores de 18 años, hemos utilizado una expresión lambda para crear la condición.

2.4.4. boolean **anyMatch**(Predicate<? super T> predicate)

Este método es similar al anterior; funciona de la misma manera y la única diferencia es que este devuelve un true **si la condición se cumple para algún elemento del listado**, solo hace falta que se cumpla para uno para que este método devuelva true.

2.4.5. boolean **noneMatch**(Predicate<? super T> predicate)

Este es el método contrario al allMatch(), ya que este devolverá true **si la condición introducida como parámetro no se cumple para ningún elemento**.

2.4.6. Optional<T> **findFirst**()

Este método devuelve el **primer elemento** del Stream; cuando no hay un orden concreto devuelve cualquier elemento de este. Los Streams pueden no tener un orden en concreto ya que dependen del tipo de colección usada y de los métodos intermedios que tenga el Stream. Además, no es necesario proporcionar ninguna condición en los parámetros.
A continuación mostramos un ejemplo:

```
Optional<Alumno> primerAlumno = listado.stream().findFirst();

System.out.println("El primer alumno del listado es: "+primerAlumno);
```

Salida resultante:

El primer alumno del listado es: Optional[Alumno [nombre=Pepito, apellido=Pérez, nota=9.5, edad=20]]

Devuelve un Optional ya que el Stream podría estar vacío.

2.4.7. Optional<T> **findAny**()

Este método es similar al anterior ya que devuelve un objeto de la lista; si no hay ningún método intermedio antes en el Stream y este no es un Stream paralelo, seguramente devuelva también el primer elemento del Stream, aunque podría no ser así. Suele ser útil cuando anteriormente en el Stream hemos hecho algún tipo de filtro y queremos saber si han quedado elementos en el Stream. Además, si el Stream fuera paralelo y no hubiera ningún método intermedio en este, este método devolvería cualquier elemento del Stream.

La **diferencia** entre este método y el anyMatch() es lo que devuelven; el método anyMatch() devuelve una variable tipo boolean y el findAny() devuelve un objeto tipo *Optional* con el elemento que haya encontrado o vacío si no hay ningún elemento en el Stream.
A continuación mostramos un ejemplo:

```
Optional<Alumno> alumnoEncontrado = listado.stream().findAny();

System.out.println("El alumno encontrado del listado es: "+alumnoEncontrado);
```

Salida resultante:

```
El alumno encontrado del listado es: Optional[Alumno [nombre=Pepito, apellido=Pérez, nota=9.5, edad=20]]
```

Como podemos observar en el resultado, devuelve lo mismo que el método findFirst(), ya que al ser un Stream secuencial selecciona el primero que encuentra; sin embargo, si este fuera paralelo, no nos devolvería siempre el primero.

2.4.8. long **count**()

Este método, como su propio nombre indica, cuenta el número de elementos que tiene el Stream y devuelve el número en una variable tipo long. Es comúnmente **utilizado con métodos intermedios.**
A continuación mostramos un ejemplo:

```
long total= listado.stream().count();

System.out.println("En el listado hay un total de "+total+" alumnos.");
```

Salida resultante:

```
En el listado hay un total de 4 alumnos.
```

2.5. Métodos collect()

La función de este método es pasar de un Stream a un **contenedor mutable**, como una colección o un *StringBuilder*(que se explican con detalle en los trabajos de Iván y Gabriel).
La diferencia entre este método y el reduce() es en lo que transforman un Stream, el método reduce() los transforma en un **contenedor no mutable**; es decir, cuando se necesita un dato primitivo para sacar por ejemplo: un número máximo, la media de los valores…

2.5.1. <R> R **collect**(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)

Esta versión del método es la más compleja; sin embargo, nos proporciona una personalización más amplia para indicar el cómo queremos que funcione el método.

A continuación explicaremos brevemente los parámetros de este método:

- **Supplier**: Este parámetro indica el tipo de colección o variable que se quiere crear
- **Accumulator**: Indica cómo se va a ir uniendo cada elemento del Stream.
- **Combiner**: Este se utiliza solo en el caso de que el Stream fuera paralelo e indica cómo se van a unir los diversos acumuladores.

A continuación mostramos un ejemplo:

```
List<String> nombres = listado.stream()
            .map(Alumno::getNombre)
        .collect(
           ArrayList::new,
           (union,alum)-> union.add(alum),
           (union,total)-> total.addAll(union)
        );

            System.out.println(nombres);
```

Salida resultante:

[Pepito, Sara, Marcos, Miguel]

En este ejemplo, hemos creado una lista con los nombres de los alumnos que tenemos en el listado, y en los parámetros hemos utilizado una referencia a método para indicar que queremos un ArrayList; a continuación, hemos usado otras dos expresiones lambdas para indicar que queremos que se vayan uniendo los nombres de los alumno a la lista de solo los nombres.

2.5.2. <R,A> R **collect**(Collector<? super T,A,R> collector)

La ejecución de este método en relación al anterior, es relativamente sencilla. Es necesario llamar a *Collectors* como parámetro y esta clase tiene unos métodos que nos permite pasar el Stream a una colección directamente; como por ejemplo los siguientes métodos: toList(), toMap(), toSet()...

A continuación mostramos un ejemplo:

```
List<String> nombres = listado.stream()
            .map(Alumno::getNombre)
        .collect(Collectors.toList());

            System.out.println(nombres);
```

Salida resultante:

[Pepito, Sara, Marcos, Miguel]

Este ejemplo se podría hacer directamente usando el método *toList()*, introducido en la versión 19 de java; sin embargo, lo hemos utilizado en este ejemplo para que se vea que también se pueden usar de esa forma.

3. Novedades en versiones

- **Versión 10**: Se han añadido los métodos toUnmodifiableList(), toUnmodifiableSet() y toUnmodifiableMap() a la clase Collectors para poder transformar (con el método collect()) el Stream en estos tipos de colecciones.

- **Versión 16**: Se ha añadido el método toList() para que no se tenga que importar el método toList() de la clase Collectors.

- **Versión 22**:Se ha añadido un "First preview" de soporte a la API para personalizar las operaciones intermedias (métodos intermedios).

- **Versión 23**:Se ha añadido un "Second preview" de soporte a la API para personalizar las operaciones intermedias (métodos intermedios).

- **Versión 24**: Se ha mejorado el soporte anterior. Mirar: JEP 485: Stream Gatherers

4. Bibliografía

Oracle. (2018). *Java SE 10 Update Release Notes*. Oracle Corporation. https://www.oracle.com/java/technologies/javase/10u-relnotes.html

Oracle. (2021). *Java SE 16 Update Release Notes*. Oracle Corporation. https://www.oracle.com/java/technologies/javase/16u-relnotes.html

Oracle. (2024). *Java SE 22 Update Release Notes*. Oracle Corporation. https://www.oracle.com/java/technologies/javase/22u-relnotes.html

Oracle. (2024). *Java SE 23 Update Release Notes*. Oracle Corporation. https://www.oracle.com/java/technologies/javase/23u-relnotes.html

Oracle. (2024). *Java SE 24 Update Release Notes*. Oracle Corporation. https://www.oracle.com/java/technologies/javase/24u-relnotes.html

Baeldung. (s.f.). *Guide to Java Collectors*. Baeldung. https://www.baeldung.com/java-collectors

Baeldung. (s.f.). *Java Stream to List – Collecting*. Baeldung. https://www.baeldung.com/java-stream-to-list-collecting

Stack Overflow. (2019). *Java streams: Should I use reduce or collect?*. Stack Overflow. https://stackoverflow.com/questions/60573521/java-streams-should-i-use-reduce-or-collect

Realnamehidden1_61. (2021). *How does reduce differ from collect in Java streams?*. Dev.to. https://dev.to/realnamehidden1_61/how-does-reduce-differ-from-collect-in-java-streams-k83

Lolcio, K. (2021). *Java Stream: reduce vs collect*. Medium. https://medium.com/@k.lolcio/java-stream-reduce-vs-collect-d14577b20645

Stack Overflow. (2023). *Streams collect(Supplier<R> supplier, BiConsumer<R, ? super T> accumulator, BiConsumer<R, R> combiner)*. Stack Overflow. https://stackoverflow.com/questions/77582717/streams-collectsupplierr-supplier-biconsumerr-super-t-accumulator-bicon

Baeldung. (s.f.). *Java Streams: findAny() and anyMatch()*. Baeldung. https://www.baeldung.com/java-streams-findany-anymatch

Baeldung. (s.f.). *Java Stream findFirst vs. findAny*. Baeldung. https://www.baeldung.com/java-stream-findfirst-vs-findany

Stack Overflow. (2015). *Why does Stream.allMatch() return true for an empty stream?*. Stack Overflow. https://stackoverflow.com/questions/30223079/why-does-stream-allmatch-return-true-for-an-empty-stream

Stackify. (s.f.). *A Guide to Streams in Java 8: In-Depth Tutorial with Examples*. Stackify. https://stackify.com/streams-guide-java-8/

Baeldung. (s.f.). *Finding the Minimum and Maximum of a Collection in Java*. Baeldung. https://www.baeldung.com/java-collection-min-max

Stack Overflow. (2016). *Java Stream toArray – Convert to a specific type of array*. Stack Overflow.

https://stackoverflow.com/questions/40902315/java-stream-toarray-convert-to-a-specific-type-of-array

Baeldung. (s.f.). *Java Stream to Array*. Baeldung. https://www.baeldung.com/java-stream-to-array

Baeldung. (s.f.). *Guide to Stream.reduce()*. Baeldung. https://www.baeldung.com/java-stream-reduce

Stack Overflow. (2017). *How does Stream.reduce(BinaryOperator<T> accumulator, T identity) work?*. Stack Overflow. https://stackoverflow.com/questions/44121799/how-does-stream-reducebinaryoperatort-accumulator-initialized

Arquitectura Java. (s.f.). *Java Stream forEach y Colecciones*. Arquitectura Java. https://www.arquitecturajava.com/java-stream-foreach-y-colecciones/

Stack Overflow. (2016). *forEach vs forEachOrdered in Java 8 Stream*. Stack Overflow. https://stackoverflow.com/questions/32797579/foreach-vs-foreachordered-in-java-8-stream

Stack Overflow. (2018). *The difference between parallel and sequential stream in terms of Java 1.8*. Stack Overflow. https://stackoverflow.com/questions/47961159/the-difference-between-parallel-and-sequential-stream-in-terms-of-java-1-8

Stack Overflow. (2014). *Should I always use a parallel stream when possible?*. Stack Overflow. https://stackoverflow.com/questions/20375176/should-i-always-use-a-parallel-stream-when-possible/23370799#23370799

KeepCoding. (s.f.). *¿Qué es Stream en Java y cómo funciona?*. KeepCoding. https://keepcoding.io/blog/que-es-stream-en-java-y-como-funciona/

Gino, R. L. (2021). *Streams en Java*. Dev.to. https://dev.to/rlgino/streams-en-java-ikl

GeeksforGeeks. (2021). *Stream min() method in Java with examples*. GeeksforGeeks. https://www.geeksforgeeks.org/stream-min-method-in-java-with-examples/

SomosPNT. (s.f.). *Métodos de la interfaz Stream de Java*. SomosPNT. https://somospnt.com/blog/308-metodos-de-la-interface-stream-de-java

GeeksforGeeks. (2021). *Java Stream findAny() with examples*. GeeksforGeeks. https://www.geeksforgeeks.org/java-stream-findany-with-examples/

Barco, C. (2021). *Streams en Java*. Medium.
https://barcochrist.medium.com/streams-en-java-5712989fc62b

Be Tech with Santander. (2021). *Programación en Java: Java Stream para Dummies*. Medium.
https://medium.com/be-tech-with-santander/programación-en-java-java-stream-para-dummies-428258c03688

Arquitectura Java. (s.f.). *Java Stream*. Arquitectura Java.
https://www.arquitecturajava.com/java-stream/

KeepCoding. (s.f.). *¿Qué es un stream en Java y cómo funciona?* KeepCoding.
https://keepcoding.io/blog/que-es-stream-en-java-y-como-funciona/

Oracle. (2018). *Stream (Java SE 11)*. Oracle Corporation.
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html

Oracle. (2018). *Comparator (Java SE 11)*. Oracle Corporation.
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Comparator.html

Oracle. (2018). *Collectors (Java SE 11)*. Oracle Corporation.
https://docs.oracle.com/en/java/javase/11/docs//api/java.base/java/util/stream/Collectors.html