13 Streams

1. Introducción

Un *stream* (flujo) en Java es una secuencia de elementos que se pueden procesar (mapear, filtrar, transformar, reducir y recolectar), de forma secuencial o paralela, mediante una cadena de operaciones especificadas a través de expresiones lambda. Introducidos en Java 8, los *streams* permiten optimizar la forma de procesar grandes colecciones de datos.

La interfaz `Stream<T>` representa un flujo de elementos de tipo `T` aunque también se definen interfaces concretas para los tipos primitivos, como `IntStream`, `LongStream`, etc.

Un *stream* es una abstracción que representa un flujo de datos pero no una estructura de datos, ya que los elementos no son almacenados en el *stream*, sino tan solo procesados por él. De hecho, no se puede acceder individualmente a un determinado elemento del *stream*, sino que se define la fuente de datos origen del *stream* y la

secuencia de operaciones que se deben aplicar sobre sus elementos, especificadas de forma funcional mediante expresiones *lambda*.

Más aún, la fuente de datos origen del *stream* no se ve afectada por las operaciones realizadas dentro del *stream*. Por ejemplo, si se filtran algunos elementos de datos del *stream*, no se eliminan realmente de la fuente de datos origen, simplemente se omiten en el *stream* a partir de ese momento y ya no se tienen en cuenta en la siguiente operación incluida en la secuencia de operaciones del *stream*. Por tanto, los datos con los que trabajamos no se ven afectados por el *stream*.

Los *streams* solo gestionan datos transitorios en memoria, lo que implica que si la aplicación falla dichos datos se perderán.

Un *stream* puede ser finito, es decir, tener un número finito de elementos, o infinito, si genera un número infinito de elementos. Algunas operaciones permiten restringir el número de elementos procesados, como `limit()` o `findFirst()`.

2. Pipeline y modelo MapReduce

Una vez hayamos creado un *stream* a partir de una fuente de datos, podemos ejecutar sobre él cero o más operaciones intermedias y, forzosamente, una operación final. A esta cadena de operaciones se le conoce como *pipeline*. Un *pipeline* tiene los siguientes elementos en el siguiente orden:

1. Una función generadora del *stream*.
2. Cero o más operaciones intermedias.
3. Una operación terminal.

Debemos tener en cuenta que cada operación intermedia del *pipeline* genera un nuevo *stream* resultante de aplicar la operación indicada al *stream* anterior de la cadena.

Veamos cómo se clasifican las operaciones que podemos llevar a cabo en un *pipeline*:

- **Operaciones intermedias** (*aggregate operations*): producen como resultado un nuevo *stream*. Se usan para transformar, filtrar y clasificar los elementos del *stream*. Pueden ser:
    - Operaciones sin estado: al aplicarlas, el procesamiento de un elemento del *stream* es independiente de cualquier otro elemento del mismo. Por ejemplo, la operación de filtrado es sin estado, ya que el filtro de cada elemento sólo depende de una condición, no de ningún otro elemento del *stream*.
    - Operaciones con estado: al aplicarlas, el procesamiento de un elemento del *stream* depende de algún otro elemento del mismo. Por ejemplo, la operación de ordenación es con estado, ya que para posicionar un elemento es necesario compararlo con el resto.

- **Operaciones terminales** (*terminal operations*): procesan todos los elementos del *stream* para generar un resultado o un efecto secundario. De hecho, no retornan un *stream*. Después de su ejecución, el *stream* original no puede ser usado de nuevo, produciendo una excepción si se intenta. De ahí que se denominen operaciones terminales. Por tanto, un determinado *stream* puede ser usado una sola vez; si necesitamos **procesar** la misma fuente de datos, deberemos crear un nuevo *stream* con ella como origen.

Otra característica importantísima del *pipeline* es que es perezoso (*lazy*), lo que quiere decir que las operaciones intermedias sólo son ejecutadas cuando las requiere la operación terminal que se esté ejecutando.

Por defecto, los elementos de un *stream* son procesados secuencialmente de uno en uno en el mismo hilo de ejecución. Es lo se conoce como *stream* secuencial. Sin embargo, podemos convertir un *stream* secuencial en un *stream* paralelo con tan sólo llamar a su método `parallel()`. Los elementos de los *streams* paralelos son agrupados en conjuntos y se usa un grupo de hilos de ejecución, denominado *common fork-join pool*, para procesar estos conjuntos de elementos en hilos de ejecución independientes.

Debemos tener en cuenta que al llamar al método `parallel()` se convierte el *stream* completo en paralelo, no solo desde el punto en el que se llama al método.

Debemos tener en cuenta que las operaciones intermedias con estado no utilizarán todas las posibilidades de paralelismo existentes, dada su naturaleza en lo relativo a la dependencia entre elementos.

En Java, los *streams* utilizan el modelo **MapReduce**, que es un modelo de programación utilizado para procesar conjuntos de datos muy grandes y que ha sido adoptado por la programación funcional. Este modelo se basa en los siguientes tipos de operaciones:

- **Transformación (*map*)**: filtra o crea copias modificadas de los elementos originales. Todas las operaciones intermedias de los *streams* corresponden a operaciones de transformación.
- **Reducción (*reduce*)**: genera un resultado resumen de todos los elementos, por ejemplo, la suma o la media aritmética. Las operaciones terminales de la clase `Stream` corresponden a operaciones de reducción. De hecho, la clase `Stream` implementa dos operaciones de reducción diferentes:
    - **De reducción pura**: implementada en las diferentes versiones del método `reduce`, que procesa un flujo de elementos para obtener un único valor.
    - **De reducción mutable**: implementada en las diferentes versiones del método `collect`, que procesa un flujo de elementos para generar una estructura de datos mutable, como por ejemplo, una colección.

3. Creación de un stream a partir de una fuente de datos

Java permite muchas maneras de crear un *stream*, dependiendo de la fuente de datos origen deseada. Veamos algunas de estas fuentes:

- **Colección**: se ejecuta el método `stream()` sobre una colección para crear un *stream* que tenga como fuente de datos origen dicha colección. También tenemos disponible el método `parallelStream()` para que los elementos sean procesados en modo paralelo.

  Podemos crear un *stream* a partir de cualquier interfaz que extienda de `Collection`, como `List`, `Set` o `Queue` y cualquiera de las clases que implementen dichas interfaces. Por ejemplo, desde una lista:

  ```
  List<Integer> list = Arrays.asList(1, 2, 3, 4);
  Stream<Integer> stream = list.stream();
  ```

  o desde el `Set` correspondiente a las entradas de un `Map`:

  ```
  Map<Integer, String> map = new HashMap<>();
  Stream<Map.Entry<Integer, String>> stream =
  map.entrySet().stream();
  ```

- **Array**: el método estático `Arrays.stream(array)` recibe un array, que actuará como fuente de datos origen del *stream*. Ejemplo:

  ```
  Integer array[] = {1, 2, 3, 4};
  Stream<Integer> stream = Arrays.stream(array);
  ```

- **Conjunto predeterminado de elementos**: el método estático `Stream.of()` recibe un número variable de elementos que actuarán como fuente de datos origen del *stream*. Ejemplo:

  ```
  Stream<Integer> stream = Stream.of(1, 2, 3, 4);
  ```

- **Función suministradora de objetos** (interfaz funcional `Supplier`): el método estático `Stream.generate()` recibe un `Supplier`, es decir, una función suministradora de elementos, que actuará como fuente de datos origen infinita para el *stream*. Ejemplo:

  ```
  Stream<Integer> stream = Stream.generate(new Random()::nextInt);
  ```

- **Un valor inicial y una función que obtiene el siguiente elemento a partir del anterior**: el método estático `Stream.iterate(seed, unaryOperator)` recibe un valor inicial y una función que recibe el elemento anterior y retorna el valor del nuevo elemento, que debe ser del mismo tipo. Se tratará de un *stream* infinito. Ejemplo:

  ```
  Stream<Integer> stream = Stream.iterate(0, x -> x + 5);
  ```

  Java 9 introdujo una nueva versión de este método `Stream.iterate(seed, predicate, unaryOperator)` que recibe un parámetro intermedio adicional correspondiente a un *predicate* que al dejar de cumplir hace que el *stream* no emita más valores, convirtiéndose en un *stream* finito. Ejemplo:

  ```
  Stream<Integer> stream = Stream.iterate(0, x -> x < 100, x -> x + 5);
  ```

- **Detectando patrones en una cadena**: el método `splitAsStream(cadena)` de la clase `Pattern` permite dividir una cadena en base a un patrón y retornar un *stream* de subcadenas. Por ejemplo:

  ```
  Stream<String> stream =
    Pattern.compile(",".splitAsStream("Luis,Paco,Ricardo"));
  ```

  retorna un *stream* cuyos elementos serán Luis, Paco y Ricardo.

- **Generador de números aleatorios:** el método `ints(limiteInf, limiteSup)` de la clase `Random` retorna un *stream* (un `IntStream`) cuya fuente de datos origen es el generador de números aleatorios contenidos entre `limiteInf` y `limiteSup`. Es especialmente útil para hacer pruebas. Ejemplo:

  ```
  IntStream stream = new Random().ints(1, 100);
  ```

- **Métodos estáticos de la clase IntStream**:
  - `IntStream.range(start, end)`: retorna un `IntStream` ordenado cuyos elementos corresponden a los enteros que van desde `start` hasta `end - 1`, es decir, `end` está excluido. Ejemplo:

    ```
    IntStream stream = IntStream.range(1, 8);
    ```

  - `IntStream.rangeClosed(start, end)`: retorna un `IntStream` ordenado cuyos elementos corresponden a los enteros que van desde `start` hasta `end` incluido. Ejemplo:

    ```
    IntStream stream = IntStream.rangeClosed(1, 8);
    ```

- **Método chars() de String**: retorna un `IntStream` cuyos elementos corresponden a los caracteres de la cadena. Ejemplo:

```
IntStream stream = "Programación".chars();
```

- **Un valor inicial**: Java 9 incorpora el método estático `Stream.ofNullable(T value)` que retorna un `Stream<T>` con el valor indicado o vacío si el valor proporcionado es `null`.
- **Un Optional**: Java 9 incorpora el método `stream()` que retorna un `Stream<T>` con un único valor correspondiente al valor contenido en el *optional* o un *stream* vacío si el *optional* no tiene valor presente.
- **Un *stream* vacío**: podemos crear un *stream* vacío mediante el método estático `empty()` de la interfaz `Stream<T>`.
- **Un stream builder**: podemos crear un *stream* a partir de un objeto `Stream.Builder<T>` al que podemos agregar elementos mediante el método `add(item)` y posteriormente usar el método `build()` del mismo para obtener el objeto `Stream<T>`.
- **La concatenación de dos streams**: El método estático `Stream.concat(stream1, stream2)` retorna un *stream* resultante de la concatenación de los dos *streams* recibidos. Ejemplo:

```
Stream<String> stream1 = Stream.of("Luis", "Paco", "Ricardo");
Stream<String> stream2 = Stream.of("Ana", "Lidia", "Esther");
Stream<String> stream = Stream.concat(stream1, stream2);
```

4. Tratamiento individual de elementos

En algunas ocasiones necesitamos realizar algún tratamiento sobre cada uno de los elementos del *stream*. En dicho caso debemos diferenciar entre operaciones terminales y no terminales. Las operaciones terminales no producirán ningún nuevo *stream* mientras que las operaciones intermedias sí que lo producirán.

- `void forEach(Consumer<? super T> action)`: operación terminal para tratar cada uno de los elementos del *stream*. Aplica la acción recibida en forma de *Consumer* a cada uno de los elementos del *stream*.
- `Stream<T> peek(Consumer<? super T> action)`: también aplica la acción recibida en forma de *Consumer* a cada uno de los elementos del *stream* pero retorna un nuevo *stream* con los mismos elementos que el original, por lo que se trata de una operación intermedia. Este método se utiliza para tratar individualmente cada uno de los elementos del *stream* sin tener por ello que terminar la cadena de operaciones.

5. Filtrado

Otra de las operaciones intermedias que se pueden realizar sobre un *stream* es el filtrado de sus elementos, es decir, la generación de un nuevo *stream* que sólo contenga algunos de los elementos del *stream* original. Java nos proporciona distintos métodos:

- `Stream<T> distinct()`: retorna un nuevo *stream* con los elementos del *stream* original, excepto aquellos que estuvieran repetidos. Para determinar que dos elementos son iguales se usará el método `equals()` del elemento. Ejemplo:

```java
package tema13_Streams;

import java.util.stream.Stream;

public class Distinct {

    public void show() {

        Stream.of(1, 3, 2, 3, 1)
        .distinct()
        .forEach(System.out::println);

    }

    public static void main(String[] args) {

        new Distinct().show();

    }

}
```

  Salida por consola:

```
1
3
2
```

- `Stream<T> limit(long maxSize)`: retorna un nuevo *stream* con tan sólo *maxSize* elementos del *stream* original, atendiendo al orden intrínseco del mismo. Tiene un mal rendimiento en *streams* paralelos ordenados. Ejemplo:

```java
package tema13_Streams;

import java.util.stream.Stream;

public class Limit {
```

```
    public void show() {

        Stream.of("Ricardo", "Luis", "Paco")
        .limit(2)
        .forEach(System.out::println);

    }

    public static void main(String[] args) {

        new Limit().show();

    }

}
```

Salida por consola:

```
Ricardo
Luis
```

- `Stream<T> filter(Predicate<? super T> predicate)`: retorna un nuevo *stream* que sólo incorpora los elementos del *stream* original que cumplan el predicado recibido. Ejemplo:

```
package tema13_Streams;

import java.util.stream.Stream;

public class Filter {

    public void show() {

        Stream.of(9, 12, 15, 24, 37, 6)
        .filter(n -> n % 2 == 0)
        .forEach(System.out::println);

    }

    public static void main(String[] args) {

        new Filter().show();

    }

}
```

Salida por consola:

```
12
24
6
```

- `Stream<T> skip(long n)`: retorna un nuevo *stream* en el que no se incluyen los primeros n elementos del *stream* original pero sí se incluye el resto. No proporciona un buen rendimiento en *streams* paralelos ordenados. Ejemplo:

```java
package tema13_Streams;

import java.util.stream.Stream;

public class Skip {

    public void show() {

        Stream.of(9, 12, 15, 24, 37, 6)
        .skip(3)
        .forEach(System.out::println);

    }

    public static void main(String[] args) {

        new Skip().show();

    }

}
```

Salida por consola:

```
24
37
6
```

- `default Stream<T> dropWhile(Predicate<? super T> predicate)`: retorna un nuevo *stream* con el primer elemento que no cumpla el predicado y el resto de elementos, independientemente de si cumplen el predicado o no. Proporciona un mal rendimiento con *streams* paralelos ordenados. Ejemplo:

```java
package tema13_Streams;

import java.util.stream.Stream;

public class DropWhile {

    public void show() {
```

```
        Stream.of(9, 13, 15, 24, 37, 6)
        .dropWhile(n -> n % 2 != 0)
        .forEach(System.out::println);

    }

    public static void main(String[] args) {

        new DropWhile().show();

    }

}
```

Salida por consola:

```
24
37
6
```

- `default Stream<T> takeWhile(Predicate<? super T> predicate)`: mientras los elementos cumplan el predicado se van incluyendo en el *stream*, pero en cuanto se encuentra un elemento que no cumple el predicado se dejan de incluir el resto de elementos, incluso aunque cumplan el predicado. Ejemplo:

```
package tema13_Streams;

import java.util.stream.Stream;

public class TakeWhile {

    public void show() {

        Stream.of(9, 13, 15, 24, 37, 6)
        .takeWhile(n -> n % 2 != 0)
        .forEach(System.out::println);

    }

    public static void main(String[] args) {

        new TakeWhile().show();

    }

}
```

Salida por consola:

```
9
13
15
```

6. Ordenación

Algunos *streams* son ordenados, es decir, que sus elementos poseen un determinado orden intrínseco significativo, conocido como *encounter order*. Por ejemplo, un *stream* cuya fuente de datos corresponda a una lista creará un *stream* ordenado, cuyo *encounter order* será el orden en el que los elementos están situados en la lista. Sin embargo, otros *streams* no son ordenados, en el sentido de que sus elementos no tienen un orden intrínseco significativo. Por ejemplo, un *stream* cuya fuente de datos sea un conjunto (*Set*) será un *stream* sin *encounter order*, ya que en un conjunto los elementos no tienen un orden preestablecido.

El hecho de que un *stream* sea ordenado o no dependerá del tipo de fuente de datos asociada y de las operaciones intermedias anteriores que hayamos realizado mediante las que se ha obtenido el *stream*.

Algunas operaciones trabajan por defecto en base a este *encounter order*, imponiendo una restricción acerca del orden en el que los elementos deben ser procesados, como por ejemplo las operaciones intermedias *limit* o *skip*.

Sin embargo, existen otras operaciones que no tienen en cuenta el *encounter order*, como por ejemplo *forEach*. Si se ejecuta sobre un *stream* paralelo, no hay ninguna garantía sobre en que orden se aplica la acción a los elementos. Si queremos que sí se tenga en cuenta el orden, entonces tendríamos que usar el método `void forEachOrdered(Consumer<? super T> action)`. Normalmente se usa encadenado después de llamar a un método de ordenación que habrá ordenado el *stream*. La ventaja de este método es que se garantiza que la acción se aplica a los elementos en el orden intrínseco del *stream*, incluso aunque éste se trate de un *stream* paralelo, aunque conlleve un peor rendimiento.

Al trabajar con *streams* secuenciales, el *encounter order* no afecta al rendimiento de la aplicación, pero si trabajamos con *streams* paralelos, el empleo del *encounter order* por parte de algunos operadores pueden afectar en gran medida al rendimiento general de la aplicación. Dependiendo de la operación de la que se trate, será necesario procesar a la vez más de un elemento del *stream* o incluso almacenar en un *buffer* intermedio una gran cantidad de datos. En estos casos podemos usar el método `unordered()` para generar un nuevo *stream* a partir del anterior, en el que no se tenga en cuenta el *encounter order*. Al ejecutar el método `unordered()`, tan solo se está creando un nuevo *stream* en el que se ha borrado el indicador de que el *encounter order* debe

tenerse en cuenta. Normalmente, esta operación de desactivación del *encounter order* se realiza con el objetivo de mejorar el rendimiento en *streams* paralelos.

Por otra parte, si queremos obtener un *stream* ordenado a partir de otro desordenado o a partir de otro *stream* ordenado pero por un orden distinto, podemos usar el método `sorted()`, en cuyo caso los elementos del *stream* deben implementar la interfaz `Comparable` para determinar el orden en el que deben ser ordenados. Otra posibilidad es usar una versión sobrecargada de dicho método que recibe un objeto `Comparator` como argumento. Ejemplo:

```java
package tema13_Streams;

import java.util.stream.Stream;

public class Sorted {

    public void show() {

        Stream.of("Ricardo", "Luis", "Paco")
        .sorted()
        .limit(2)
        .forEach(System.out::println);

    }

    public static void main(String[] args) {

        new Sorted().show();

    }

}
```

Salida por consola:

```
Luis
Paco
```

Ejemplo utilizando `Comparator`:

```java
package tema13_Streams;

import java.util.Comparator;
import java.util.stream.Stream;

public class SortedReverseOrder {
```

```
    public void show() {

        Stream.of("Ricardo", "Luis", "Paco")
        .sorted(Comparator.reverseOrder())
        .limit(2)
        .forEach(System.out::println);

    }

    public static void main(String[] args) {

        new SortedReverseOrder().show();

    }

}
```

Salida por consola:

```
Ricardo
Paco
```

7. Transformación

Java nos proporciona distintos métodos:

- `<R> Stream<R> map(Function<? super T,? extends R> mapper)`: retorna un nuevo *stream* obtenido a partir de aplicar la función de transformación indicada a cada uno de los elementos del *stream* original. El tipo del *stream* resultante corresponderá al tipo de retorno de la función de transformación, que puede ser distinto al tipo del *stream* original, pero contendrá tantos elementos como éste. Ejemplo:

```
package tema13_Streams;

import java.util.stream.Stream;

public class Map {

    public void show() {

        Stream.of(20, 27, 31)
        .map(n -> "Número " + n)
        .forEach(System.out::println);

    }

    public static void main(String[] args) {
```

```
        new Map().show();

    }

}
```

Salida por consola:

```
Número 20
Número 27
Número 31
```

- Métodos que permiten obtener un *stream* de un tipo primitivo a partir de uno que no lo sea:
    - `DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)`: retorna un `DoubleStream` correspondiente de aplicar a cada elemento del *stream* original la función de conversión a *double* recibida. Ejemplo:

```java
package tema13_Streams;

import java.util.stream.Stream;

public class MapToDouble {

    public void show() {

        Stream.of(20, 27, 31)
        .mapToDouble(n -> n * 0.5)
        .forEach(System.out::println);

    }

    public static void main(String[] args) {

        new MapToDouble().show();

    }

}
```

Salida por consola:

```
10.0
13.5
15.5
```

- `IntStream mapToInt(ToIntFunction<? super T> mapper)`: retorna un `IntStream` correspondiente de aplicar a cada elemento del *stream* original la función de conversión a *int* recibida. Ejemplo:

```java
package tema13_Streams;

import java.util.stream.Stream;

public class MapToInt {

    public void show() {

        Stream.of("Ricardo", "Luis Miguel", "Paco")
        .mapToInt(n -> n.length())
        .forEach(System.out::println);

    }

    public static void main(String[] args) {

        new MapToInt().show();

    }

}
```

Salida por consola:

```
7
11
4
```

- `LongStream mapToLong(ToLongFunction<? super T> mapper)`: retorna un `LongStream` correspondiente de aplicar a cada elemento del *stream* original la función de conversión a *long* recibida.

```java
package tema13_Streams;

import java.util.stream.Stream;

public class MapToLong {

    public void show() {

        Stream.of(55000, 60000, 72500)
        .mapToLong(n -> (long)n * n)
        .forEach(System.out::println);

    }
```

```
    public static void main(String[] args) {

        new MapToLong().show();

    }

}
```

Salida por consola:

```
3025000000
3600000000
5256250000
```

- `<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)`: cuando una función de transformación retorna un *stream* y se aplica esta función con el método *map*, el *stream* resultante es un `Stream<Stream<Tipo>>`. En estos casos, es más óptimo obtener un único `Stream<Tipo>` que contuviera concatenados todos los elementos de todos los *substreams*. A este proceso se le conoce como aplanado (*flat*) de *substreams*. Ejemplo:

```
package tema13_Streams;

import java.util.stream.IntStream;
import java.util.stream.Stream;

public class FlatMap {

    public void show() {

        Stream.of(1, 2, 3)
        .flatMap(n -> IntStream.rangeClosed(1, n).boxed())
        .forEach(System.out::println);

    }

    public static void main(String[] args) {

        new FlatMap().show();

    }

}
```

Salida por consola:

```
1
1
2
1
2
3
```

En el ejemplo, muestra los valores `1` (proveniente del primer *substream*), `1`, `2` (provenientes del segundo *substream)* y `1`, `2` y `3`, provenientes del tercer *substream*, en este orden.

Si nos interesara que el tipo del *stream* resultante fuera primitivo, podemos usar los métodos `flatMapToDouble`, `flatMapToInt` o `flatMapToLong`. Ejemplo:

```java
package tema13_Streams;

import java.util.stream.IntStream;
import java.util.stream.Stream;

public class FlatMapToInt {

    public void show() {

        Stream.of(1, 2, 3)
        .flatMapToInt(n -> IntStream.rangeClosed(1, n))
        .forEach(System.out::println);

    }

    public static void main(String[] args) {

        new FlatMapToInt().show();

    }

}
```

Salida por consola:

```
1
1
2
1
2
3
```

8. Reducción pura

8.1 Métodos estándar de reducción

- `long count()`: retorna el número de elementos de un *stream*. Ejemplo:

```java
package tema13_Streams;

import java.util.stream.Stream;

public class Count {

    public void show() {

        long howManyAreEven;
        howManyAreEven = Stream.of(30, 23, 24, 57, 8, 15)
                        .filter(n -> n % 2 == 0)
                        .count();
        System.out.println(howManyAreEven);//3

    }

    public static void main(String[] args) {

        new Count().show();

    }

}
```

- Las clases `IntStream`, `LongStream` y `DoubleStream`, correspondientes a *streams* de elementos de tipos primitivos numéricos, disponen del método `sum()` para calcular la suma de los elementos del *stream*:

```java
package tema13_Streams;

import java.util.stream.IntStream;

public class Sum {

    public void show() {

        int sumEvenNumbers;
        sumEvenNumbers = IntStream.of(30, 23, 24, 57, 8, 15)
                        .filter(n -> n % 2 == 0)
                        .sum();
        System.out.println(sumEvenNumbers);//62

    }
```

```
    public static void main(String[] args) {

        new Sum().show();

    }

}
```

Si disponemos de un *stream* genérico de elementos y no de un *stream* de alguna de las clases mencionadas anteriormente, podemos usar alguno de los métodos `mapToInt()`, `mapToLong()`, `mapToDouble()`, `flatMapToInt()`, etc., para obtener un *stream* de un tipo específico. Ejemplo:

```
package tema13_Streams;

import java.util.stream.Stream;

public class SumMapToInt {

    public void show() {

        int sumEvenNumbers;
        sumEvenNumbers = Stream.of(30, 23, 24, 57, 8, 15)
                        .filter(n -> n % 2 == 0)
                        .mapToInt(n -> n)
                        .sum();
        System.out.println(sumEvenNumbers);//62

    }

    public static void main(String[] args) {

        new SumMapToInt().show();

    }

}
```

- Estas clases de *streams* de elementos de tipo primitivo numérico también disponen de métodos para obtener el valor máximo, `max()`, el valor mínimo, `min()` y la media aritmética, `average()`, de los elementos numéricos del *stream*. Dado que el *stream* sobre el que se apliquen puede estar vacío, estos métodos retornan un *Optional*.

```
package tema13_Streams;

import java.util.OptionalDouble;
import java.util.OptionalInt;
```

```java
import java.util.stream.IntStream;

public class MaxMinAverage {

    public void show() {

        OptionalInt minEvenNumbers, maxEvenNumbers;
        OptionalDouble averageEvenNumbers;

        minEvenNumbers = IntStream.of(30, 23, 24, 57, 8, 15)
                        .filter(n -> n % 2 == 0)
                        .min();
        minEvenNumbers.ifPresent(System.out::println);//8

        maxEvenNumbers = IntStream.of(30, 23, 24, 57, 8, 15)
                        .filter(n -> n % 2 == 0)
                        .max();
        maxEvenNumbers.ifPresent(System.out::println);//30

        averageEvenNumbers = IntStream.of(30, 23, 24, 57, 8, 15)
                        .filter(n -> n % 2 == 0)
                        .average();
        averageEvenNumbers.ifPresent(n ->
System.out.printf("%.2f", n));//20,67

    }

    public static void main(String[] args) {

        new MaxMinAverage().show();

    }

}
```

Si se trata de un *stream* genérico, tenemos disponibles métodos para calcular el máximo y el mínimo que reciben un *Comparator* para comparar los elementos del *stream* y así obtener en cada caso el valor mínimo o el máximo:

- `Optional<T> max(Comparator<? super T> comparator)`
- `Optional<T> min(Comparator<? super T> comparator)`

```java
package tema13_Streams;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.Optional;

public class MaxMinComparator {
```

```
    public void show() {

        Optional<Vehicle> minVehicle;
        Optional<Vehicle> maxVehicle;
        List<Vehicle> list = new ArrayList<>();
        list.add(new Vehicle("1705UBG", 4, "blanco"));
        list.add(new Vehicle("1235GTR", 2, "rojo"));

        System.out.println("Vehículo con menor número de
ruedas:");
        minVehicle = list.stream()

.min(Comparator.comparingInt(Vehicle::getWheelCount));
        minVehicle.ifPresent(System.out::println);

        System.out.println("Vehículo con mayor matrícula
alfabéticamente:");
        maxVehicle = list.stream()

.max(Comparator.comparing(Vehicle::getRegistration));
        maxVehicle.ifPresent(System.out::println);

    }

    public static void main(String[] args) {

        new MaxMinComparator().show();

    }

}
```

Salida por consola:

```
Vehículo con menor número de ruedas:
Vehicle [registration=1235GTR, wheelCount=2, speed=0.0,
colour=rojo]
Vehículo con mayor matrícula alfabéticamente
Vehicle [registration=1705UBG, wheelCount=4, speed=0.0,
colour=blanco]
```

8.2 Método reduce

¿Y si queremos realizar una operación de reducción distinta a las anteriores? Para casos más genéricos usaremos el método `Optional<T> reduce(BinaryOperator<T> accumulator)` que recibe una función acumuladora. La función acumuladora debe ser una función asociativa, es decir, da igual en qué orden se opere con los elementos porque siempre se va a obtener el mismo resultado. Se trata de una regla que se

cumple en la suma y en la multiplicación. Por ejemplo, `1+2+3` da el mismo resultado que `2+3+1`.

En este caso, el proceso de reducción comienza cuando se obtiene el segundo elemento, ya que necesitamos al menos dos elementos para hacer la primera reducción. Por este motivo, el método retorna un `Optional`, dado que si el *stream* no tiene elementos suficientes no se puede realizar la reducción ni producir ningún valor. Ejemplo:

```
package tema13_Streams;

import java.util.OptionalInt;
import java.util.stream.IntStream;

public class Reduce {

    public void show() {

        OptionalInt integerSum;

        integerSum = IntStream.of(30, 23, 24, 57, 8, 15)
                    .reduce((subtotal, element) -> subtotal +
element);//Con lambda
        integerSum.ifPresent(System.out::println);//157

        integerSum = IntStream.of(30, 23, 24, 57, 8, 15)
                    .reduce(Integer::sum);//Con referencia a método
        integerSum.ifPresent(System.out::println);//157

        integerSum = IntStream.empty()
                    .reduce(Integer::sum);
        System.out.println(integerSum);//OptionalInt.empty
        integerSum.ifPresent(System.out::println);//No hace nada

    }

    public static void main(String[] args) {

        new Reduce().show();

    }

}
```

Este método está sobrecargado para pasarle como primer parámetro un valor conocido como identidad (*identity*), que es usado como valor inicial de la operación de reducción y el resultado por defecto si el *stream* está vacío: `T reduce(T identity, BinaryOperator<T> accumulator)`.

Debemos elegir cuidadosamente el valor de *identity* atendiendo a la operación que se lleve a cabo en la función *binaryOperator*. El valor de *identity* debe ser una identidad para la función acumuladora, es decir, si se aplica la función acumuladora a cualquier elemento con la identidad, debe devolver el mismo elemento. Por ejemplo, si utilizamos como operación una suma, la identidad debe ser 0 ya que cualquier número al que le sumemos 0, nos devuelve el mismo número. Por el mismo razonamiento, si la operación es una multiplicación, la identidad debe ser 1.

```java
package tema13_Streams;

import java.util.stream.IntStream;

public class ReduceIdentity {

    public void show() {

        Integer sum,mult;

        sum = IntStream.of(30, 23, 24, 57, 8, 15)
              .reduce(0, Integer::sum);
        System.out.println(sum);//157

        sum = IntStream.empty()
              .reduce(0, Integer::sum);
        System.out.println(sum);//0

        mult = IntStream.of(2,3,4)
               .reduce(1, (subtotal,element) -> subtotal *
element);//Con lambda
        System.out.println(mult);//24

        mult = IntStream.of(2,3,4)
               .reduce(1, Math::multiplyExact);//Con referencia a
método
        System.out.println(mult);//24

        mult = IntStream.empty()
              .reduce(1, Math::multiplyExact);
        System.out.println(mult);//1

    }

    public static void main(String[] args) {

        new ReduceIdentity().show();

    }

}
```

Tenemos disponible una tercera versión del método, `<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)`, especialmente útil para *streams* paralelos, que se usa cuando queremos que el método retorne un valor de tipo diferente al del *stream* original. La función *combiner* es necesaria para indicar cómo se deben combinar acumulaciones parciales realizadas en distintos hilos en *streams* paralelos.

```
package tema13_Streams;

import java.util.stream.Stream;

public class ReduceCombiner {

    public void show() {

        int count = Stream.of("Juan", "Pepe", "Luis", "Ricardo",
"Laura")
                    .reduce(0,(subtotal,element) -> {
                        if(element.length() % 2 == 0) {
                            return subtotal + 1;
                        }
                        else {
                            return subtotal;
                        }
                    }, Integer::sum);

        System.out.printf("Cuántos nombres con número de caracteres
pares: %d", count);//3

    }

    public static void main(String[] args) {

        new ReduceCombiner().show();

    }

}
```

Debemos tener en cuenta que la función `reduce()` en cualquiera de sus versiones respeta el orden del *stream* a la hora de combinar los cálculos intermedios.

9. Operaciones terminales de consulta

La clase `Stream` también proporciona una serie de métodos de consulta sobre los elementos de un *stream*, denominadas operaciones de cortocircuito (*short-circuit terminal operations*). Se llaman así porque se deja de procesar el resto de elementos si con los elementos que ya han sido procesados se es capaz de determinar el resultado.

Tenemos un conjunto de métodos que permiten consultar, respectivamente, si todos, ninguno o algunos de los elementos del *stream* cumplen con un determinado predicado, retornando un valor booleano:

- `boolean allMatch(Predicate<? super T> predicate)`
- `boolean noneMatch(Predicate<? super T> predicate)`
- `boolean anyMatch(Predicate<? super T> predicate)`

```java
package tema13_Streams;

import java.util.stream.IntStream;

public class Match {

    public void show() {

        boolean match;
        match = IntStream.of(1, 2, 4)
                .allMatch(n -> n % 2 == 0);
        System.out.println(match);//false

        match = IntStream.of(2, 4, 6)
                .allMatch(n -> n % 2 == 0);
        System.out.println(match);//true

        match = IntStream.of(1, 2, 4)
                .noneMatch(n -> n % 2 == 0);
        System.out.println(match);//false

        match = IntStream.of(1, 3, 5)
                .noneMatch(n -> n % 2 == 0);
        System.out.println(match);//true

        match = IntStream.of(1, 3, 5)
                .anyMatch(n -> n % 2 == 0);
        System.out.println(match);//false

        match = IntStream.of(2, 3, 5)
                .anyMatch(n -> n % 2 == 0);
        System.out.println(match);//true

    }

    public static void main(String[] args) {
```

```
        new Match().show();

    }

}
```

Por otro lado tenemos los métodos `findFirst()` y `findAny()`, que retornan un `Optional<T>` con, respectivamente, el primer elemento del *stream*, o algún elemento del *stream* (no está indicado cuál), si es que existe. Un aspecto curioso es que estos métodos no reciben ningún predicado con el que indicar la condición de búsqueda por lo que normalmente se usan después de haber ejecutado el método `filter` sobre el *stream*.

```
package tema13_Streams;

import java.util.Optional;
import java.util.stream.Stream;

public class Find {

    public void show() {

        Optional<Integer> find;
        find = Stream.of(1, 2, 4)
               .filter(n -> n % 2 == 0)
               .findFirst();
        find.ifPresent(System.out::println);//2

        find = Stream.of(6, 2, 4)
               .filter(n -> n % 2 == 0)
               .findAny();
        find.ifPresent(System.out::println);//6

    }

    public static void main(String[] args) {

        new Find().show();

    }

}
```

Un aspecto curioso es que no se proporciona ningún método para obtener el último elemento de un *stream*. Sin embargo, podemos obtenerlo usando el método `skip` (siempre y cuando se trate de un *stream* finito):

```java
package tema13_Streams;

import java.util.List;
import java.util.Optional;

public class LastElement {

    public void show() {

        List<Integer> list = List.of(30, 23, 24, 57, 8, 15);
        long count = list.stream().count();
        Optional<Integer> last = list.stream()
                                 .skip(count - 1)
                                 .findFirst();
        last.ifPresent(System.out::println);//15

    }

    public static void main(String[] args) {

        new LastElement().show();

    }

}
```

Otra manera de hacerlo es mediante una reducción en la que siempre nos quedemos con el segundo elemento:

```java
package tema13_Streams;

import java.util.OptionalInt;
import java.util.stream.IntStream;

public class LastReduce {

    public void show() {

        OptionalInt last;

        last = IntStream.of(30, 23, 24, 57, 8, 15)
                    .reduce((first, second) -> second);
        last.ifPresent(System.out::println);//15

    }

    public static void main(String[] args) {

        new LastReduce().show();
```

```
    }

}
```

Debemos tener en cuenta que si ejecutamos estos métodos sobre *streams* paralelos el resultado puede ser distinto entre distintas llamadas.

10. Reducción mutable

La operación de **recolección** es una operación terminal que permite crear una estructura de datos con los resultados del procesamiento de datos asociado a un *stream*. La operación de recolección también recibe el nombre de operación de reducción mutable. Para llevarla a cabo usaremos el método `<R, A> R collect(Collector<? super T,A,R> collector)`.

El método recibe un objeto de una clase que implemente la interfaz `Collector`. Aunque podemos crear nuestras propias clases que implementen dicha interfaz, en la mayoría de las ocasiones podremos utilizar alguno de los recolectores estándar proporcionados por Java a través de la clase auxiliar `Collectors`, que contiene métodos estáticos que retornan objetos `Collector` correspondientes a los recolectores más habituales. Todos estos métodos están diseñados para funcionar de manera óptima incluso con *streams* paralelos.

10.1 Recolectores a estructuras de datos clásicas

La clase `Collectors` tiene un conjunto de métodos estáticos que nos permiten recolectar los elementos de un *stream* y almacenarlos en una estructura de datos:

- `public static <T> Collector<T,?,List<T>> toList()`: retorna una lista con los elementos del stream. En Java 16, se ha incorporado también un método `toList()` a la interfaz `Stream<T>`.
- `public static <T> Collector<T,?,List<T>> toUnmodifiableList()`: retorna una lista inmutable de los elementos del *stream* en el orden en que son producidos (*encounter order*).
- `public static <T> Collector<T,?,Set<T>> toSet()`: retorna un *set* (conjunto) con los elementos del *stream*. Si se hay elementos duplicados en el *stream*, son ignorados.
- `public static <T> Collector<T,?,Set<T>> toUnmodifiableSet()`: retorna un *set* (conjunto) inmutable con los elementos del *stream*. Si hay elementos duplicados en el *stream*, son ignorados.
- `public static <T, C extends Collection<T>> Collector<T,?,C> toCollection(Supplier<C> collectionFactory)`: el problema de `toList()` y `toSet()` es que no podemos especificar la implementación concreta que queremos que se use. Por ejemplo, no podemos indicar que se use un `LinkedList`, en el caso de `toList()`, o un `TreeSet`, en el caso de `toSet()`.

Para solucionar este problema, este método recibe un *supplier* que retorna la estructura de datos concreta en la que queremos que se recolecte el *stream*.

```
package tema13_Streams.collect;

import java.util.List;
import java.util.SortedSet;
import java.util.TreeSet;
import java.util.stream.Collectors;

public class CollectorsClassicDataStructures {

    public void show() {

        List<Integer> list = List.of(3, 6, 1, 2, 4, 5);
        List<Integer> listEvenNumbers = list.stream()
                                            .filter(n -> n % 2 == 0)
                                            .collect(Collectors.toList());
        listEvenNumbers.forEach(System.out::println);

        SortedSet<Integer> tree = list.stream()
                                      .filter(n -> n % 2 == 0)

.collect(Collectors.toCollection(TreeSet::new));
        tree.forEach(System.out::println);

    }

    public static void main(String[] args) {

        new CollectorsClassicDataStructures().show();

    }

}
```

Salida por consola:

```
6
2
4
2
4
6
```

- `public static <T, K, U> Collector<T,?,Map<K,U>> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper)`: retorna un mapa obtenido de la siguiente manera:

- Con el `Function` *keyMapper* obtiene la clave, de manera que su valor de retorno será usado como clave del elemento en el mapa resultante. Como el mapa resultante no puede tener claves repetidas, si retorna el mismo valor para dos elementos distintos del *stream*, se lanzará la excepción `IllegalStateException`.
- Con el `Function` *valueMapper* se obtiene el valor del elemento en el mapa resultante.

```java
package tema13_Streams.collect;

import java.util.Arrays;
import java.util.Map;
import java.util.stream.Collectors;

public class CollectorsToMap {

    public void show() {

        Map<String, Vehicle> map;
        Vehicle vehicles[] = new Vehicle[6];
        vehicles[0] = new Vehicle("9685KMX", 4, "azul");
        vehicles[1] = new Vehicle("1235GTR", 2, "rojo");
        vehicles[2] = new Vehicle("7314QWE", 4, "verde");
        vehicles[3] = new Vehicle("5930POI", 2, "negro");
        vehicles[4] = new Vehicle("1705UBG", 4, "blanco");
        vehicles[5] = new Vehicle("3495JZA", 2, "naranja");
        map = Arrays.stream(vehicles)
              .collect(Collectors.toMap(Vehicle::getRegistration,
vehicle -> vehicle));
        map.forEach((k, v) -> System.out.printf("Clave:%s
Valor:%s\n", k, v));

    }

    public static void main(String[] args) {

        new CollectorsToMap().show();

    }

}
```

Salida por consola:

```
Clave:3495JZA Valor:Vehicle [registration=3495JZA,
wheelCount=2,speed=0.0,colour=naranja]
Clave:1705UBG Valor:Vehicle [registration=1705UBG,
wheelCount=4,speed=0.0,colour=blanco]
Clave:1235GTR Valor:Vehicle [registration=1235GTR, wheelCount=2,
speed=0.0, colour=rojo]
Clave:7314QWE Valor:Vehicle [registration=7314QWE, wheelCount=4,
speed=0.0, colour=verde]
Clave:9685KMX Valor:Vehicle [registration=9685KMX, wheelCount=4,
speed=0.0, colour=azul]
Clave:5930POI Valor:Vehicle [registration=5930POI, wheelCount=2,
speed=0.0, colour=negro]
```

Si nos fijamos en el `Function` *valueMapper (vehicle -> vehicle)*, también podemos usar el método estático `static <T> Function<T,T> identity()`, que es una función que siempre devuelve su argumento de entrada:

```
package tema13_Streams.collect;

import java.util.Arrays;
import java.util.Map;
import java.util.function.Function;
import java.util.stream.Collectors;

public class FunctionIdentity {

    public void show() {

        Map<String, Vehicle> map;
        Vehicle vehicles[] = new Vehicle[6];
        vehicles[0] = new Vehicle("9685KMX", 4, "azul");
        vehicles[1] = new Vehicle("1235GTR", 2, "rojo");
        vehicles[2] = new Vehicle("7314QWE", 4, "verde");
        vehicles[3] = new Vehicle("5930POI", 2, "negro");
        vehicles[4] = new Vehicle("1705UBG", 4, "blanco");
        vehicles[5] = new Vehicle("3495JZA", 2, "naranja");
        map = Arrays.stream(vehicles)
              .collect(Collectors.toMap(Vehicle::getRegistration,
Function.identity()));
        map.forEach((k, v) -> System.out.printf("Clave:%s
Valor:%s\n", k, v));

    }

    public static void main(String[] args) {

        new FunctionIdentity().show();

    }
```

```
}
```

En el `Function` *keyMapper*, como el mapa resultante no puede tener claves repetidas, si retorna el mismo valor para dos elementos distintos del *stream*, se lanzará la excepción `IllegalStateException`:

```
package tema13_Streams.collect;

import java.util.Arrays;
import java.util.Map;
import java.util.stream.Collectors;

public class RepeatedKeys {

    public void show() {

        Map<String, String> map;
        Vehicle vehicles[] = new Vehicle[6];
        vehicles[0] = new Vehicle("9685KMX", 4, "azul");
        vehicles[1] = new Vehicle("1235GTR", 2, "rojo");
        vehicles[2] = new Vehicle("7314QWE", 4, "verde");
        vehicles[3] = new Vehicle("5930POI", 2, "rojo");
        vehicles[4] = new Vehicle("1705UBG", 4, "blanco");
        vehicles[5] = new Vehicle("3495JZA", 2, "azul");
        map = Arrays.stream(vehicles)
              .collect(Collectors.toMap(Vehicle::getColour,
Vehicle::getRegistration));
        map.forEach((k, v) -> System.out.printf("Clave:%s
Valor:%s\n", k, v));

    }

    public static void main(String[] args) {

        new RepeatedKeys().show();

    }

}
```

Tenemos disponible una segunda versión del método para que en lugar de lanzar una excepción, proporcionemos una función de combinación de elementos con la misma clave:

- `public static <T, K, U> Collector<T,?,Map<K,U>> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction)`: el tercer parámetro indica cómo se deben combinar dos elementos con la misma clave.

```
```

```
package tema13_Streams.collect;

import java.util.Arrays;
import java.util.Map;
import java.util.stream.Collectors;

public class CombinationRepeatedKeys {

    public void show() {

        Vehicle vehicles[] = new Vehicle[6];
        vehicles[0] = new Vehicle("9685KMX", 4, "azul");
        vehicles[1] = new Vehicle("1235GTR", 2, "rojo");
        vehicles[2] = new Vehicle("7314QWE", 4, "verde");
        vehicles[3] = new Vehicle("5930POI", 2, "rojo");
        vehicles[4] = new Vehicle("1705UBG", 4, "blanco");
        vehicles[5] = new Vehicle("3495JZA", 2, "azul");
        Map<String, String> map = Arrays.stream(vehicles)
              .collect(Collectors.toMap(Vehicle::getColour,
Vehicle::getRegistration,
                      (r1,r2) -> String.format("%s-%s", r1 ,r2)));
        map.forEach((k, v) -> System.out.printf("Clave:%-7s
Valor:%s\n", k, v));

    }

    public static void main(String[] args) {

        new CombinationRepeatedKeys().show();

    }

}
```

Salida por consola:

```
Clave:rojo    Valor:1235GTR-5930POI
Clave:blanco  Valor:1705UBG
Clave:verde   Valor:7314QWE
Clave:azul    Valor:9685KMX-3495JZA
```

Estas dos versiones del método `toMap` retornan por defecto un `HashMap`, por lo que existe una tercera versión para obtener una implementación distinta de la interfaz `Map`, como por ejemplo `LinkedHashMap` o `TreeMap`:

- `public static <T, K, U, M extends Map<K, U>> Collector<T,?,M> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction, Supplier<M> mapFactory)`: recibe un `Supplier` como cuarto parámetro para indicar el tipo

de mapa que se quiere obtener:

```java
package tema13_Streams.collect;

import java.util.Arrays;
import java.util.Map;
import java.util.TreeMap;
import java.util.stream.Collectors;

public class CombinationRepeatedKeysSupplier {

    public void show() {

        Vehicle vehicles[] = new Vehicle[6];
        vehicles[0] = new Vehicle("9685KMX", 4, "azul");
        vehicles[1] = new Vehicle("1235GTR", 2, "rojo");
        vehicles[2] = new Vehicle("7314QWE", 4, "verde");
        vehicles[3] = new Vehicle("5930POI", 2, "rojo");
        vehicles[4] = new Vehicle("1705UBG", 4, "blanco");
        vehicles[5] = new Vehicle("3495JZA", 2, "azul");
        Map<String, String> map = Arrays.stream(vehicles)
              .collect(Collectors.toMap(Vehicle::getColour,
Vehicle::getRegistration,
                      (r1,r2) -> String.format("%s-%s", r1 ,r2),
                      TreeMap::new));
        map.forEach((k, v) -> System.out.printf("Clave:%-7s
Valor:%s\n", k, v));

    }

    public static void main(String[] args) {

        new CombinationRepeatedKeysSupplier().show();

    }

}
```

Salida por consola:

```
Clave:azul    Valor:9685KMX-3495JZA
Clave:blanco  Valor:1705UBG
Clave:rojo    Valor:1235GTR-5930POI
Clave:verde   Valor:7314QWE
```

- Recolección de un *stream* hacia un **array**: no se realizará a través de ningún recolector ni del método `collect`, sino directamente a través del método `toArray()` de la clase `Stream`, que retorna un `Object[]`, es decir un *array* de elementos de la clase `Object`, debido a que los arrays no usan genéricos.

```java
package tema13_Streams.collect;

import java.util.ArrayList;
import java.util.List;

public class ToArray {

    public void show() {

        List<Vehicle> list = new ArrayList<>();
        list.add(new Vehicle("9685KMX", 4, "azul"));
        list.add(new Vehicle("1235GTR", 2, "rojo"));
        list.add(new Vehicle("7314QWE", 4, "verde"));
        list.add(new Vehicle("5930POI", 2, "negro"));
        list.add(new Vehicle("1705UBG", 4, "blanco"));
        list.add(new Vehicle("3495JZA", 2, "naranja"));
        Object vehiclesArray[] = list.stream()
                                     .toArray();
        for(Object v: vehiclesArray) {
            System.out.println(v);
        }

    }

    public static void main(String[] args) {

        new ToArray().show();

    }

}
```

10.2 Recolectores de operaciones de reducción básicas

La clase `Collectors` dispone además de una serie métodos estáticos que retornan recolectores parecidos a las operaciones de reducción:

- `counting()`: Para obtener el **número de elementos**.

```java
package tema13_Streams.collect;

import java.util.stream.Collectors;
import java.util.stream.Stream;

public class CollectorsCounting {

    public void show() {

        long howManyAreEven;
        howManyAreEven = Stream.of(30, 23, 24, 57, 8, 15)
```

```
                        .filter(n -> n % 2 == 0)
                        .collect(Collectors.counting());
        System.out.println(howManyAreEven);//3

    }

    public static void main(String[] args) {

        new CollectorsCounting().show();

    }

}
```

- Para obtener la **suma** de los elementos que son convertidos al tipo indicado mediante la función suministrada:
  - `public static <T> Collector<T,?,Integer> summingInt(ToIntFunction<? super T> mapper)`
  - `public static <T> Collector<T,?,Long> summingLong(ToLongFunction<? super T> mapper)`
  - `public static <T> Collector<T,?,Double> summingDouble(ToDoubleFunction<? super T> mapper)`

```
package tema13_Streams.collect;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class CollectorsSumming {

    public void show() {

        List<Vehicle> list = new ArrayList<>();
        list.add(new Vehicle("9685KMX", 4, "azul"));
        list.add(new Vehicle("1235GTR", 2, "rojo"));
        list.add(new Vehicle("7314QWE", 4, "verde"));
        list.add(new Vehicle("5930POI", 2, "negro"));
        list.add(new Vehicle("1705UBG", 4, "blanco"));
        list.add(new Vehicle("3495JZA", 2, "naranja"));
        int sumWheels = list.stream()

.collect(Collectors.summingInt(Vehicle::getWheelCount));
        System.out.println(sumWheels);//18

    }

    public static void main(String[] args) {
```

```
        new CollectorsSumming().show();

    }

}
```

- Para obtener el valor **mínimo y el máximo** atendiendo a un comparador pasado como argumento:
  - `public static <T> Collector<T,?,Optional<T>> minBy(Comparator<? super T> comparator)`
  - `public static <T> Collector<T,?,Optional<T>> maxBy(Comparator<? super T> comparator)`

```
package tema13_Streams.collect;

import java.util.Comparator;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

public class CollectorsMinMax {

    public void show() {

        Optional<Integer> min,max;
        List<Integer> list = List.of(30, 23, 24, 57, 8, 15);
        min = list.stream()

.collect(Collectors.minBy(Comparator.naturalOrder()));
        min.ifPresent(System.out::println);//8

        max = list.stream()

.collect(Collectors.maxBy(Comparator.naturalOrder()));
        max.ifPresent(System.out::println);//57

    }

    public static void main(String[] args) {

        new CollectorsMinMax().show();

    }

}
```

- Para obtener la **media aritmética** de los valores (que son convertidos al tipo indicado mediante la función suministrada). Si el *stream* no tiene elementos retorna cero:

- `public static <T> Collector<T,?,Double> averagingInt(ToIntFunction<? super T> mapper)`
- `public static <T> Collector<T,?,Double> averagingLong(ToLongFunction<? super T> mapper)`
- `public static <T> Collector<T,?,Double> averagingDouble(ToDoubleFunction<? super T> mapper)`

```java
package tema13_Streams.collect;

import java.util.List;
import java.util.stream.Collectors;

public class CollectorsAveraging {

    public void show() {

        double average;
        List<Integer> list = List.of(30, 23, 24, 57, 8, 15);
        average = list.stream()

.collect(Collectors.averagingInt(Integer::intValue));
        System.out.printf("%.2f",average);//26,17

    }

    public static void main(String[] args) {

        new CollectorsAveraging().show();

    }

}
```

- Para obtener todo lo anterior (número de elementos, suma, mínimo, máximo y la media) de una sola vez:
  - `public static <T> Collector<T,?,IntSummaryStatistics> summarizingInt(ToIntFunction<? super T> mapper)`
  - `public static <T> Collector<T,?,LongSummaryStatistics> summarizingLong(ToLongFunction<? super T> mapper)`
  - `public static <T> Collector<T,?,DoubleSummaryStatistics> summarizingDouble(ToDoubleFunction<? super T> mapper)`

```java
package tema13_Streams.collect;

import java.util.IntSummaryStatistics;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Summarizing {
```

```
    public void show() {

        IntSummaryStatistics oddStatistics;
        oddStatistics = Stream.of(30, 23, 24, 57, 8, 15)
                        .filter(n -> n % 2 != 0)

.collect(Collectors.summarizingInt(Integer::intValue));
        System.out.println(oddStatistics);

    }

    public static void main(String[] args) {

        new Summarizing().show();

    }

}
```

Salida por consola:

```
IntSummaryStatistics{count=3, sum=95, min=15, average=31,666667,
max=57}
```

- Para recolectar con una operación de reducción distinta de las anteriores: `public static <T> Collector<T,?,Optional<T>> reducing(BinaryOperator<T> op)`

```
package tema13_Streams.collect;

import java.util.Optional;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class CollectorsReducing {

    public void show() {

        Optional<Integer> integerSum;

        integerSum = Stream.of(30, 23, 24, 57, 8, 15)
                     .collect(Collectors.reducing((subtotal,
element) -> subtotal + element));//Con lambda
        integerSum.ifPresent(System.out::println);//157

        integerSum = Stream.of(30, 23, 24, 57, 8, 15)

.collect(Collectors.reducing(Integer::sum));//Con referencia a
método
        integerSum.ifPresent(System.out::println);//157
```

```
        integerSum = Stream.<Integer>empty()

.collect(Collectors.reducing(Integer::sum));//Optional.empty
        integerSum.ifPresent(System.out::println);//No hace nada

    }

    public static void main(String[] args) {

        new CollectorsReducing().show();

    }

}
```

También tenemos otra versión del método que recibe como primer parámetro el valor correspondiente a la identidad: `public static <T> Collector<T,?,T> reducing(T identity, BinaryOperator<T> op)`

```
package tema13_Streams.collect;

import java.util.stream.Collectors;
import java.util.stream.Stream;

public class CollectorsReducingIdentity {

    public void show() {

        Integer sum,mult;

        sum = Stream.of(30, 23, 24, 57, 8, 15)
              .collect(Collectors.reducing(0, Integer::sum));
        System.out.println(sum);//157

        sum = Stream.<Integer>empty()
              .collect(Collectors.reducing(0, Integer::sum));
        System.out.println(sum);//0

        mult = Stream.of(2,3,4)
               .collect(Collectors.reducing(1,
Math::multiplyExact));
        System.out.println(mult);//24

        mult = Stream.<Integer>empty()
               .collect(Collectors.reducing(1,
Math::multiplyExact));
        System.out.println(mult);//1

    }
```

```
    public static void main(String[] args) {

        new CollectorsReducingIdentity().show();

    }

}
```

Existe además una tercera versión del método que recibe como segundo parámetro una función de transformación que será ejecutada sobre cada elemento antes de realizar la recolección: `public static <T, U> Collector<T,?,U> reducing(U identity, Function<? super T,? extends U> mapper, BinaryOperator<U> op)`

```
package tema13_Streams.collect;

import java.util.stream.Collectors;
import java.util.stream.Stream;

public class CollectorsReducingCombiner {

    public void show() {

        int count = Stream.of("Juan", "Pepe", "Luis", "Ricardo",
"Laura")
                    .collect(Collectors.reducing(0, element -> {
                        if(element.length() % 2 == 0) {
                            return 1;
                        }
                        else {
                            return 0;
                        }
                    }, Integer::sum));

        System.out.printf("Cuántos nombres con número de
caracteres pares: %d", count);//3

    }

    public static void main(String[] args) {

        new CollectorsReducingCombiner().show();

    }

}
```

Como podemos apreciar, todos los recolectores vistos en este apartado son muy similares (casi iguales) a los métodos estándar de reducción que vimos en un apartado anterior. Entonces, ¿por qué existen estos recolectores? El motivo es que, como veremos más adelante, Java nos va a ofrecer la oportunidad de encadenar varios recolectores, de manera que usaremos los recolectores vistos en este apartado normalmente como acompañante de algún otro recolector. De hecho, **no se recomienda usar estos recolectores si no es en conjunción con otro recolector**. Si se va a emplear de forma aislada, es más óptimo emplear los métodos estándar de recolección que vimos en un apartado anterior.

10.3 Recolectores de transformación

- `public static Collector<CharSequence,?,String> joining()`: permite obtener un `String` correspondiente a la concatenación de los elementos del *stream*. Solo podremos usar este recolector sobre un *stream* de elementos de tipo `CharSequence`, por lo que es posible que antes hayamos tenido que aplicar una operación de transformación mediante el método `map` para obtener un *stream* adecuado. `CharSequence` es una interfaz que representa una secuencia de caracteres. Esta interfaz no impone la mutabilidad, por lo tanto, nos podemos encontrar con clases inmutables y mutables que implementen esta interfaz. Por ejemplo, `String` es inmutable y `StringBuilder` y `StringBuffer` son mutables.

```java
package tema13_Streams.collect;

import java.util.stream.Collectors;
import java.util.stream.Stream;

public class CollectorsJoining {

    public void show() {

        String result = Stream.of("Juan", "Pepe", "Luis",
"Ricardo", "Laura")
                        .collect(Collectors.joining());
        System.out.println(result);

    }

    public static void main(String[] args) {

        new CollectorsJoining().show();

    }

}
```

Salida por consola:

```
JuanPepeLuisRicardoLaura
```

Existe una segunda versión para concatenar los elementos de entrada separados por un delimitador que se pasa por parámetro: `public static Collector<CharSequence,?,String> joining(CharSequence delimiter)`.

```
package tema13_Streams.collect;

import java.util.stream.Collectors;
import java.util.stream.Stream;

public class CollectorsJoiningDelimiter {

    public void show() {

        String result = Stream.of("Juan", "Pepe", "Luis",
"Ricardo", "Laura")
                              .collect(Collectors.joining(" - "));
        System.out.println(result);

    }

    public static void main(String[] args) {

        new CollectorsJoiningDelimiter().show();

    }

}
```

Salida por consola:

```
Juan - Pepe - Luis - Ricardo - Laura
```

Existe una tercera versión del método que permite indicar el prefijo y el sufijo que queremos poner a la cadena resultante de la concatenación: `public static Collector<CharSequence,?,String> joining(CharSequence delimiter, CharSequence prefix, CharSequence suffix)`

```
package tema13_Streams.collect;

import java.util.stream.Collectors;
import java.util.stream.Stream;

public class CollectorsJoiningPrefixSuffix {

    public void show() {
```

```
        String result = Stream.of("Juan", "Pepe", "Luis",
"Ricardo", "Laura")
                              .collect(Collectors.joining(" - ", "Lista
de nombres: ","."));
        System.out.println(result);

    }

    public static void main(String[] args) {

        new CollectorsJoiningPrefixSuffix().show();

    }

}
```

Salida por consola:

```
Lista de nombres: Juan - Pepe - Luis - Ricardo - Laura.
```

- `public static <T, U, A, R> Collector<T,?,R> mapping(Function<? super T,? extends U> mapper,  Collector<? super U,A,R> downstream)`: realiza alguna operación de transformación sobre los elementos justo antes de aplicarles un recolector que se pasa por parámetro.

```
package tema13_Streams.collect;

import java.util.Arrays;
import java.util.stream.Collectors;

public class CollectorsMapping {

    public void show() {

        Vehicle vehicles[] = new Vehicle[6];
        vehicles[0] = new Vehicle("9685KMX", 4, "azul");
        vehicles[1] = new Vehicle("1235GTR", 2, "rojo");
        vehicles[2] = new Vehicle("7314QWE", 4, "verde");
        vehicles[3] = new Vehicle("5930POI", 2, "negro");
        vehicles[4] = new Vehicle("1705UBG", 4, "blanco");
        vehicles[5] = new Vehicle("3495JZA", 2, "naranja");
        String result = Arrays.stream(vehicles)

.collect(Collectors.mapping(Vehicle::getRegistration,
Collectors.joining(", ")));
        System.out.println(result);

    }
```

```
    public static void main(String[] args) {

        new CollectorsMapping().show();

    }

}
```

Salida por consola:

```
9685KMX, 1235GTR, 7314QWE, 5930POI, 1705UBG, 3495JZA
```

10.4 Recolectores de agrupación

`public static <T, K> Collector<T,?,Map<K,List<T>>> groupingBy(Function<? super T,? extends K> classifier)`: permite obtener un mapa `Map<K, List<T>>` donde las claves son los valores resultantes de aplicar la función de clasificación a los elementos de entrada y los valores son listas que contienen los elementos de entrada que al aplicarles la función de clasificación se obtiene la clave correspondiente, es decir, todos aquellos elementos del *stream* original que al aplicarles la función clasificadora retornen el mismo valor, dicho valor será la clave y los elementos serán agrupados en la misma lista con dicha clave. Veamos un ejemplo: vamos a crear un mapa donde la clave será el número de ruedas del vehículo. El mapa tendrá dos entradas, una para los vehículos de 2 ruedas y otra para los vehículos de 4 ruedas. Cada clave tendrá una lista con los vehículos que correspondan con dicho número de ruedas:

```
package tema13_Streams.collect;

import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class CollectorsGroupingBy {

    public void show() {

        Map<Integer, List<Vehicle>> map;
        Vehicle vehicles[] = new Vehicle[6];
        vehicles[0] = new Vehicle("9685KMX", 4, "azul");
        vehicles[1] = new Vehicle("1235GTR", 2, "rojo");
        vehicles[2] = new Vehicle("7314QWE", 4, "verde");
        vehicles[3] = new Vehicle("5930POI", 2, "negro");
        vehicles[4] = new Vehicle("1705UBG", 4, "blanco");
        vehicles[5] = new Vehicle("3495JZA", 2, "naranja");
        map = Arrays.stream(vehicles)
              .collect(Collectors.groupingBy(Vehicle::getWheelCount));
        map.forEach((k, v) -> {
```

```
            System.out.printf("\nVehículos con %d ruedas:", k);
            v.forEach(vehicle -> System.out.printf("\n%s",vehicle));
        });

    }

    public static void main(String[] args) {

        new CollectorsGroupingBy().show();

    }

}
```

Salida por consola:

```
Vehículos con 2 ruedas:
Vehicle [registration=1235GTR, wheelCount=2, speed=0.0, colour=rojo]
Vehicle [registration=5930POI, wheelCount=2, speed=0.0, colour=negro]
Vehicle [registration=3495JZA, wheelCount=2, speed=0.0,
colour=naranja]
Vehículos con 4 ruedas:
Vehicle [registration=9685KMX, wheelCount=4, speed=0.0, colour=azul]
Vehicle [registration=7314QWE, wheelCount=4, speed=0.0, colour=verde]
Vehicle [registration=1705UBG, wheelCount=4, speed=0.0, colour=blanco]
```

Sin embargo, será muy habitual que queramos realizar algún cálculo sobre la lista de elementos de cada grupo. Para hacernos más sencilla dicha tarea, tenemos disponible otra versión del método: `public static <T, K, A, D> Collector<T,?,Map<K,D>> groupingBy(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)`, que recibe un recolector que queremos que se le aplique a la lista de elementos de cada grupo. Para este cometido, podemos usar los recolectores de operaciones de reducción básicas que vimos en el apartado anterior. Si lo aplicamos al ejemplo anterior, podemos obtener cuántos vehículos hay con 2 ruedas y cuántos hay con 4 ruedas:

```
package tema13_Streams.collect;

import java.util.Arrays;
import java.util.Map;
import java.util.stream.Collectors;

public class CollectorsGroupingByDownstream {

    public void show() {

        Map<Integer, Long> map;
        Vehicle vehicles[] = new Vehicle[6];
```

```
        vehicles[0] = new Vehicle("7314QWE", 4, "verde");
        vehicles[1] = new Vehicle("9685KMX", 2, "azul");
        vehicles[2] = new Vehicle("1235GTR", 2, "rojo");
        vehicles[3] = new Vehicle("5930POI", 2, "negro");
        vehicles[4] = new Vehicle("1705UBG", 4, "blanco");
        vehicles[5] = new Vehicle("3495JZA", 2, "naranja");
        map = Arrays.stream(vehicles)
              .collect(Collectors.groupingBy(Vehicle::getWheelCount,
Collectors.counting()));
        map.forEach((k, v) -> System.out.printf("Número de vehículos
con %d ruedas: %d\n", k, v));
    }

    public static void main(String[] args) {

        new CollectorsGroupingByDownstream().show();

    }

}
```

Salida por consola:

```
Número de vehículos con 2 ruedas: 4
Número de vehículos con 4 ruedas: 2
```

Estas dos versiones del método `groupingBy` retornan por defecto un `HashMap`, por lo que existe una tercera versión para obtener una implementación distinta de la interfaz `Map`, como por ejemplo `LinkedHashMap` o `TreeMap`: `public static <T, K, D, A, M extends Map<K, D>> Collector<T,?,M> groupingBy(Function<? super T,? extends K> classifier, Supplier<M> mapFactory, Collector<? super T,A,D> downstream)`.

```
package tema13_Streams.collect;

import java.util.Arrays;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.stream.Collectors;

public class CollectorsGroupingBySupplier {

    public void show() {

        Map<Integer, Long> map;
        Vehicle vehicles[] = new Vehicle[6];
        vehicles[0] = new Vehicle("7314QWE", 4, "verde");
        vehicles[1] = new Vehicle("9685KMX", 2, "azul");
```

```
        vehicles[2] = new Vehicle("1235GTR", 2, "rojo");
        vehicles[3] = new Vehicle("5930POI", 2, "negro");
        vehicles[4] = new Vehicle("1705UBG", 4, "blanco");
        vehicles[5] = new Vehicle("3495JZA", 2, "naranja");
        map = Arrays.stream(vehicles)

.collect(Collectors.groupingBy(Vehicle::getWheelCount,LinkedHashMap::n
ew, Collectors.counting()));
        map.forEach((k, v) -> System.out.printf("Número de vehículos
con %d ruedas: %d\n", k, v));
    }

    public static void main(String[] args) {

        new CollectorsGroupingBySupplier().show();

    }

}
```

Salida por consola:

```
Número de vehículos con 4 ruedas: 2
Número de vehículos con 2 ruedas: 4
```

¿Y si queremos realizar alguna operación de conversión sobre los elementos de la lista de cada grupo antes de aplicarle el recolector *downstream*? Podemos usar el método `Collectors.mapping` que ya conocemos:

```
package tema13_Streams.collect;

import java.util.Arrays;
import java.util.Map;
import java.util.TreeMap;
import java.util.stream.Collectors;

public class CollectorsGroupingByMapping {

    public void show() {

        Map<Integer, String> map;
        Vehicle vehicles[] = new Vehicle[6];
        vehicles[0] = new Vehicle("7314QWE", 4, "verde");
        vehicles[1] = new Vehicle("9685KMX", 2, "azul");
        vehicles[2] = new Vehicle("1235GTR", 2, "rojo");
        vehicles[3] = new Vehicle("5930POI", 2, "negro");
        vehicles[4] = new Vehicle("1705UBG", 4, "blanco");
        vehicles[5] = new Vehicle("3495JZA", 2, "naranja");
        map = Arrays.stream(vehicles)
```

```
                .collect(Collectors.groupingBy(Vehicle::getWheelCount,
TreeMap::new,

Collectors.mapping(Vehicle::getRegistration,

Collectors.joining(" - "))));
        map.forEach((k, v) -> System.out.printf("Matrículas con %d
ruedas: %s\n", k, v));
    }

    public static void main(String[] args) {

        new CollectorsGroupingByMapping().show();

    }

}
```

Salida por consola:

```
Matrículas con 2 ruedas: 9685KMX - 1235GTR - 5930POI - 3495JZA
Matrículas con 4 ruedas: 7314QWE - 1705UBG
```

Si la función de transformación retorna un *stream*, podemos usar `public static <T, U, A, R> Collector<T,?,R> flatMapping(Function<? super T,? extends Stream<? extends U>> mapper,  Collector<? super U,A,R> downstream)` disponible a partir de Java 9.

Por ejemplo, vamos a cambiar la clase *Vehicle* para registrar modelos de coches con un determinado número de ruedas y una lista de todos los colores en los que está disponible dicho modelo:

```
package tema13_Streams.modelVehicle;

import java.util.List;

public class Vehicle {

    private String model;
    private int wheelCount;
    private List<String> colours;

    public Vehicle(String model, int wheelCount, List<String> colours)
{
        this.model = model;
        this.wheelCount = wheelCount;
        this.colours = colours;
    }
```

```
    public String getModel() {
        return model;
    }

    public int getWheelCount() {
        return wheelCount;
    }

    public List<String> getColours() {
        return colours;
    }

}
```

Si quisiéramos obtener de cada número de ruedas cuántos colores hay disponibles, podríamos hacer lo siguiente:

```
package tema13_Streams.modelVehicle;

import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.TreeMap;
import java.util.stream.Collectors;

public class CollectorsGroupingByMapping {

    public void show() {

        Map<Integer, List<Object>> map;
        Vehicle vehicles[] = new Vehicle[4];
        vehicles[0] = new Vehicle("Audi", 4,
List.of("azul","rojo","blanco"));
        vehicles[1] = new Vehicle("Ford", 4,
List.of("naranja","blanco"));
        vehicles[2] = new Vehicle("Audi", 2,
List.of("negro","verde"));
        vehicles[3] = new Vehicle("Ford", 2,
List.of("negro","blanco"));
        map = Arrays.stream(vehicles)
              .collect(Collectors.groupingBy(Vehicle::getWheelCount,
TreeMap::new,
                                             Collectors.mapping(v ->
v.getColours(),

Collectors.toList())));
        map.forEach((k, v) -> System.out.printf("Número de ruedas:%d
Colores: %s\n", k, v));
    }
```

```
    public static void main(String[] args) {

        new CollectorsGroupingByMapping().show();

    }

}
```

Salida por consola:

```
Número de ruedas:2 Colores: [[negro, verde], [negro, blanco]]
Número de ruedas:4 Colores: [[azul, rojo, blanco], [naranja, blanco]]
```

Si nos fijamos en la salida de consola, salen las sublistas. Para quitarlas, podemos pasar estas listas a *streams* y luego utilizar el método `flatMapping` para aplanarlas:

```
package tema13_Streams.modelVehicle;

import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.TreeMap;
import java.util.stream.Collectors;

public class CollectorsGroupingByFlatMapping {

    public void show() {

        Map<Integer, String> map;
        Vehicle vehicles[] = new Vehicle[4];
        vehicles[0] = new Vehicle("Audi", 4,
List.of("azul","rojo","blanco"));
        vehicles[1] = new Vehicle("Ford", 4,
List.of("naranja","blanco"));
        vehicles[2] = new Vehicle("Audi", 2,
List.of("negro","verde"));
        vehicles[3] = new Vehicle("Ford", 2,
List.of("negro","blanco"));
        map = Arrays.stream(vehicles)
              .collect(Collectors.groupingBy(Vehicle::getWheelCount,
TreeMap::new,
                                             Collectors.flatMapping(v
-> v.getColours().stream(),

Collectors.joining("-"))));
        map.forEach((k, v) -> System.out.printf("Número de ruedas:%d
Colores: %s\n", k, v));
    }
```

```
    public static void main(String[] args) {

        new CollectorsGroupingByFlatMapping().show();

    }

}
```

Salida por consola:

```
Número de ruedas:2 Colores: negro-verde-negro-blanco
Número de ruedas:4 Colores: azul-rojo-blanco-naranja-blanco
```

Ahora nos encontramos con colores repetidos. Para quitarlos, podemos hacer uso del método `public static <T, A, R, RR> Collector<T,A,RR> collectingAndThen(Collector<T,A,R> downstream, Function<R,RR> finisher)`. Este método recolecta y después se puede realizar un *Function* sobre el resultado de la recolección:

```
package tema13_Streams.modelVehicle;

import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.TreeMap;
import java.util.stream.Collectors;

public class CollectorsGroupingByCollectingAndThen {

    public void show() {

        Map<Integer, String> map;
        Vehicle vehicles[] = new Vehicle[4];
        vehicles[0] = new Vehicle("Audi", 4,
List.of("azul","rojo","blanco"));
        vehicles[1] = new Vehicle("Ford", 4,
List.of("naranja","blanco"));
        vehicles[2] = new Vehicle("Audi", 2,
List.of("negro","verde"));
        vehicles[3] = new Vehicle("Ford", 2,
List.of("negro","blanco"));
        map = Arrays.stream(vehicles)
              .collect(Collectors.groupingBy(Vehicle::getWheelCount,
TreeMap::new,

Collectors.collectingAndThen(Collectors.flatMapping(v ->
v.getColours().stream(),Collectors.toList()),
```

```
 list -> list.stream().distinct().collect(Collectors.joining("-"))));


        map.forEach((k, v) -> System.out.printf("Número de ruedas:%d
Colores: %s\n", k, v));
    }

    public static void main(String[] args) {

        new CollectorsGroupingByCollectingAndThen().show();

    }

}
```

Salida por consola:

```
Número de ruedas:2 Colores: negro-verde-blanco
Número de ruedas:4 Colores: azul-rojo-blanco-naranja
```

En otras ocasiones, lo que queremos es filtrar los elementos de cada lista en base a algún criterio, antes de aplicarle el recolector *downstream*. Para ello, Java 9 incorporó el método `public static <T, A, R> Collector<T,?,R> filtering(Predicate<? super T> predicate, Collector<? super T,A,R> downstream)`:

```
package tema13_Streams.collectors;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class CollectorsGroupingByFiltering {

    public void show() {

        Map<String, List<Vehicle>> map;
        List<Vehicle> vehicles = new ArrayList<>();
        vehicles.add(new Vehicle("Audi", 4, List.of("azul","rojo")));
        vehicles.add(new Vehicle("Ford", 4,
List.of("naranja","blanco")));
        vehicles.add(new Vehicle("Audi", 2,
List.of("negro","verde")));
        vehicles.add(new Vehicle("Ford", 2,
List.of("amarillo","blanco")));
        map = vehicles.stream()
              .collect(Collectors.groupingBy(Vehicle::getModel,
```

```
                                                Collectors.filtering(v ->
v.getWheelCount() == 4,

 Collectors.toList())));
        map.forEach((k, v) -> System.out.printf("Modelo:%s Vehículos
de 4 ruedas: %s\n", k, v));
    }

    public static void main(String[] args) {

        new CollectorsGroupingByFiltering().show();

    }

}
```

Salida por consola:

```
Modelo:Audi Vehículos de 4 ruedas: [Vehicle [model=Audi, wheelCount=4,
speed=0.0, colours=[azul, rojo]]]
Modelo:Ford Vehículos de 4 ruedas: [Vehicle [model=Ford, wheelCount=4,
speed=0.0, colours=[naranja, blanco]]]
```

Se pueden crear varios niveles de agrupamiento, aplicando `groupingBy` como recolector *downstream* de los elementos de cada grupo:

```
package tema13_Streams.modelVehicle;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class CollectorsVariousLevelsOfGroupingBy {

    public void show() {

        Map<String, Map<Integer, Long>> map;
        List<Vehicle> vehicles = new ArrayList<>();
        vehicles.add(new Vehicle("Audi", 4, List.of("azul","rojo")));
        vehicles.add(new Vehicle("Ford", 4,
List.of("naranja","blanco","verde")));
        vehicles.add(new Vehicle("Seat", 4,
List.of("amarillo","verde")));
        vehicles.add(new Vehicle("Audi", 2, List.of("negro")));
        vehicles.add(new Vehicle("Ford", 2,
List.of("rojo","blanco")));
        vehicles.add(new Vehicle("Seat", 2,
List.of("amarillo","morado")));
```

```
        map = vehicles.stream()
              .collect(Collectors.groupingBy(Vehicle::getModel,

Collectors.groupingBy(Vehicle::getWheelCount,

Collectors.flatMapping(v -> v.getColours().stream(),

                  Collectors.counting())))));
        map.forEach((k, v) -> {
            v.forEach((k2, v2) -> {
                System.out.printf("Modelo %s con %d ruedas está
disponible en %d %s\n",k,k2,v2,v2==1?"color":"colores");
            });
        });
    }

    public static void main(String[] args) {

        new CollectorsVariousLevelsOfGroupingBy().show();

    }

}
```

Salida por consola:

```
Modelo Seat con 2 ruedas está disponible en 2 colores
Modelo Seat con 4 ruedas está disponible en 2 colores
Modelo Audi con 2 ruedas está disponible en 1 color
Modelo Audi con 4 ruedas está disponible en 2 colores
Modelo Ford con 2 ruedas está disponible en 2 colores
Modelo Ford con 4 ruedas está disponible en 3 colores
```

10.5 Recolectores de particionado

`public static <T> Collector<T,?,Map<Boolean,List<T>>> partitioningBy(Predicate<? super T> predicate)`: aplica el predicado proporcionado a cada uno de los elementos del *stream* y crea dos grupos, uno con los que cumplen el predicado y otro con los que no lo cumplen:

```
package tema13_Streams.modelVehicle;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class CollectorsPartitioningBy {
```

```
    public void show() {

        Map<Boolean, List<Vehicle>> map;
        List<Vehicle> vehicles = new ArrayList<>();
        vehicles.add(new Vehicle("Audi", 4, List.of("azul","rojo")));
        vehicles.add(new Vehicle("Ford", 4,
List.of("naranja","blanco")));
        vehicles.add(new Vehicle("Audi", 2,
List.of("negro","verde")));
        vehicles.add(new Vehicle("Ford", 2,
List.of("amarillo","blanco")));
        map = vehicles.stream()
              .collect(Collectors.partitioningBy(v ->
v.getWheelCount() == 4));
        map.forEach((k, v) -> {
            System.out.printf("%s\n", k?"Vehículos de 4
ruedas:":"Vehículos que no son de 4 ruedas:");
            v.forEach(System.out::println);
        });
    }

    public static void main(String[] args) {

        new CollectorsPartitioningBy().show();

    }

}
```

Salida por consola:

```
Vehículos que no son de 4 ruedas:
Vehicle [model=Audi, wheelCount=2, colours=[negro, verde]]
Vehicle [model=Ford, wheelCount=2, colours=[amarillo, blanco]]
Vehículos de 4 ruedas:
Vehicle [model=Audi, wheelCount=4, colours=[azul, rojo]]
Vehicle [model=Ford, wheelCount=4, colours=[naranja, blanco]]
```

Existe otra versión del método que recibe como segundo argumento un recolector para que sea ejecutado sobre la lista de cada grupo:

```
package tema13_Streams.modelVehicle;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;
```

```
public class CollectorsPartitioningByDownstream {

    public void show() {

        Map<Boolean, Long> map;
        List<Vehicle> vehicles = new ArrayList<>();
        vehicles.add(new Vehicle("Audi", 4, List.of("azul","rojo")));
        vehicles.add(new Vehicle("Ford", 4,
List.of("naranja","blanco","verde")));
        vehicles.add(new Vehicle("Seat", 4,
List.of("amarillo","verde")));
        vehicles.add(new Vehicle("Audi", 2, List.of("negro")));
        vehicles.add(new Vehicle("Ford", 2,
List.of("rojo","blanco")));
        map = vehicles.stream()
              .collect(Collectors.partitioningBy(v ->
v.getWheelCount() == 4,

Collectors.counting()));
        map.forEach((k, v) -> System.out.printf("%s hay en %d
modelos\n", k?"Vehículos de 4 ruedas:":"Vehículos que no son de 4
ruedas:",v));
    }

    public static void main(String[] args) {

        new CollectorsPartitioningByDownstream().show();

    }

}
```

Salida por consola:

```
Vehículos que no son de 4 ruedas: hay en 2 modelos
Vehículos de 4 ruedas: hay en 3 modelos
```

10.6 Combinación de dos recolectores

`public static <T, R1, R2, R> Collector<T,?,R> teeing(Collector<? super T,?,R1> downstream1, Collector<? super T,?,R2> downstream2, BiFunction<? super R1,? super R2,R> merger)`: ejecuta ambos recolectores sobre los elementos y después ejecuta sobre los resultados la *BiFunction* proporcionada, que combinará ambos resultados, de manera que la combinación será el producto final de la recolección:

```
package tema13_Streams.modelVehicle;
```

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.stream.Collectors;

public class CollectorsTeeing {

    public void show() {

        Result result;
        List<Vehicle> vehicles = new ArrayList<>();
        vehicles.add(new Vehicle("Audi", 4, List.of("azul","rojo")));
        vehicles.add(new Vehicle("Ford", 4,
List.of("naranja","blanco","verde")));
        vehicles.add(new Vehicle("Seat", 4,
List.of("amarillo","verde")));
        vehicles.add(new Vehicle("Audi", 2, List.of("negro")));
        vehicles.add(new Vehicle("Ford", 2,
List.of("rojo","blanco")));
        vehicles.add(new Vehicle("Seat", 2,
List.of("amarillo","morado")));
        result = vehicles.stream()

.collect(Collectors.teeing(Collectors.minBy(Comparator.comparing((Vehi
cle v) -> v.getColours().size())),

 Collectors.maxBy(Comparator.comparing((Vehicle v) ->
v.getColours().size())),
                                                Result::new));


        System.out.println(result);
    }

    public static void main(String[] args) {

        new CollectorsTeeing().show();

    }

}
```

siendo la clase `Result`:

```
package tema13_Streams.modelVehicle;

import java.util.Optional;

public class Result{
```

```
    private Optional<Vehicle> min;
    private Optional<Vehicle> max;

    public Result(Optional<Vehicle> min,Optional<Vehicle> max) {
        this.min = min;
        this.max = max;
    }

    public Optional<Vehicle> getMin() {
        return min;
    }

    public Optional<Vehicle> getMax() {
        return max;
    }

}
```

Salida por consola:

```
El vehículo que tiene el mínimo número de colores es Vehicle
[model=Audi, wheelCount=2, colours=[negro]]
El vehículo que tiene el máximo número de colores es Vehicle
[model=Ford, wheelCount=4, colours=[naranja, blanco, verde]]
```

Este método es de la versión 12 de Java. En versiones anteriores, sería necesario operar dos veces sobre el *stream*, almacenar los resultados intermedios en variables temporales y después combinar las variables temporales.