

---

# Interfaz Stream

---

## Tabla de contenido

Formas de crear un Stream.....	2
A partir de una colección, llamando al método stream() .....	2
A partir de un array .....	3
Creándolo directamente.....	3
Operaciones intermedias.....	4
Encadenar streams formando tuberías o pipelines.....	4
Filtrado y mapeo: .....	4
Operaciones terminales.....	5
Resultados numéricos .....	6
Referencias.....	7

La interfaz Stream es una herramienta de programación que permite trabajar con flujos de datos secuenciales en tiempo real. Con esta herramienta, los desarrolladores pueden trabajar con datos de forma más efectiva y realizar tareas como filtrado, mapeo, agrupación, agregación y transformación en tiempo real.

Cuando una clase implementa la interfaz Stream, los objetos de esta clase son sucesiones de elementos sobre los que se puede realizar una serie de operaciones sucesivamente; es decir, se van encadenando hasta dar un resultado final.

Al stream inicial se le denomina **fuentes**.

A partir de ahí, las **operaciones** realizadas con un stream pueden ser:

- **intermedias:** dan como resultado un nuevo stream, al que se le pueden seguir aplicando nuevas operaciones; es lo que se conoce como *tubería o pipeline*,
- o **terminales** (dan un resultado final, numérico o de otro tipo, pero no un stream). Tras realizar una de estas operaciones ya no se puede aplicar ninguna más.

Suponiendo que tenemos un ArrayList de objetos de tipo Transacción, cuyos atributos son idTransacción, comprador, ciudad, fecha y precio, un ejemplo de stream para calcular el importe total de las transacciones realizadas en Londres sería:

```
double totalLondres = transacciones.stream() Stream<Transacción>
    .filter(t -> t.getCiudad().equals("Londres"))
    .mapToDouble(t -> t.getPrecio()) DoubleStream
    .sum();
```

Donde:

- el objeto que resulta de aplicar el método `stream()` es la fuente;
- `filter()` y `mapToDouble()` son operaciones intermedias;
- `sum()` es una operación final.

Los stream se suelen crear **a partir de arrays o colecciones**. La ventaja que tienen es que disponen de muchos más métodos para procesar datos que las colecciones o los arrays.

Muchas de las operaciones de stream utilizan **interfaces funcionales** de la API (Predicate, Consumer... etc.). De hecho, stream se ha diseñado para trabajar con expresiones lambda.

Una cosa muy importante de los stream es que **no son reusables**; cada operación intermedia que hacemos devuelve un stream diferente, con lo que el stream original se pierde. Si intentamos reutilizar un stream que ya ha llegado a una operación terminal, nos aparecerá un error como este:

Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed

## Formas de crear un Stream

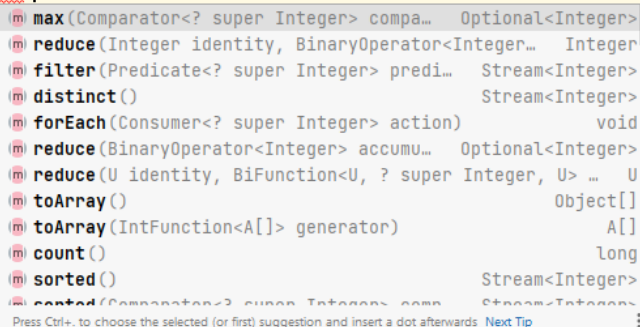
### A partir de una colección, llamando al método `stream()`

```
// ejemplo: creamos una lista
List<Integer> lista = new ArrayList<>();

// lo rellenamos con 20 números aleatorios entre 10 y -10
for (int i = 0; i < 20; i++) {
    lista.add((Integer) (int) (Math.random() * 21 - 10));
}

// creamos un stream a partir de una lista
Stream<Integer> listaStream = lista.stream();
listaStream.
System.out.println(listaStream);
}

// ahora un ejemplo de uso de stream
(1) x
```



The screenshot shows an IDE with a list of methods for `Stream<Integer>`. The methods listed are: `max(Comparator<? super Integer> compa... Optional<Integer>`, `reduce(Integer identity, BinaryOperator<Integer... Integer`, `filter(Predicate<? super Integer> predi... Stream<Integer>`, `distinct() Stream<Integer>`, `forEach(Consumer<? super Integer> action) void`, `reduce(BinaryOperator<Integer> accumu... Optional<Integer>`, `reduce(U identity, BiFunction<U, ? super Integer, U> ... U`, `toArray() Object[]`, `toArray(IntFunction<A[]> generator) A[]`, `count() long`, and `sorted() Stream<Integer>`. The `sorted()` method is highlighted.

Nada más crear el stream vemos que tenemos a nuestra disposición unos cuantos métodos bastante prometedores de cara a ahorrarnos teclear código. También vemos que muchos de ellos reciben por parámetro interfaces funcionales: Comparador, Predicate, Function...

En nuestro ejemplo, podemos empezar imprimiendo todos los números, así:

```
listaStream.forEach(n -> System.out.print(n + " "));
```

Como `forEach` es una operación terminal, al utilizarlo cerramos `listaStream` y no nos va a dejar aplicarle ninguna operación más. Es entonces cuando recurriremos al truco de reasignar, así:

```
listaStream = lista.stream();
```

## A partir de un array

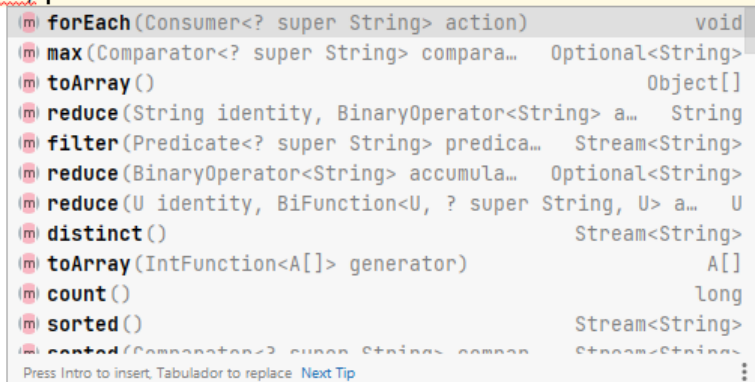
---

- a) **Usando el método `stream()` de la clase `Arrays`, pasándole el array como parámetro.** Ejemplo:

```
String[] frutas = {"mora", "naranja", "nispero", "pera", "plátano",  
                  "pomelo", "sandía", "melón", "uva", "piña", "melocotón"};
```

```
Stream<String> streamArray = Arrays.stream(frutas);
```

```
streamArray.|
```



- b) **Llamando al método `of()` de la interfaz `Stream`, pasándole el array como parámetro.**

Ejemplo:

```
String[] frutas = {"mora", "naranja", "nispero", "pera", "plátano",  
                  "pomelo", "sandía", "melón", "uva", "piña", "melocotón"};
```

```
Stream<String> streamArray = Stream.of(frutas);
```

## Creándolo directamente

---

Podemos inicializar un stream directamente, pasando como parámetros los valores a `Stream.of()`.

Ejemplo:

```
Stream<String> stream3 = Stream.of("mora", "naranja", "níspero", "pera",  
"plátano", "pomelo", "sandía", "melón", "uva", "piña", "melocotón");
```

Cuando creamos un stream, lo que **contiene es una copia** de los datos de la colección u array inicial, no una referencia al original. En el ejemplo anterior, por tanto, todo lo que hagamos con el stream no va a afectar, en este caso, al array `frutas`.

## Operaciones intermedias

---

### Encadenar streams formando tuberías o pipelines

---

Podemos encadenar operaciones intermedias una tras otra (el resultado de cada operación se pasará como entrada a la operación siguiente) para formar lo que se llama una *tubería* o *pipeline*.

Una tubería es un stream fuente al que se aplica una serie de operaciones intermedias encadenadas y se acaba con una operación terminal.

Como las tuberías son largas, para poder leerlas mejor se suele poner cada operación en una línea:

Por ejemplo:

```
Arrays.stream(frutas)  
    .filter(x -> x.startsWith("p"))  
    .forEach(x -> System.out.println(x));
```

### Filtrado y mapeo:

---

Se hace con el método `filter`, que recibe un predicado y devuelve un stream con los elementos que cumplen el predicado.

- **Stream<T> `filter(Predicate<? super T> predicate)`**

Ejemplo: tenemos la lista original con los números

```
-5 6 0 5 0 -7 -7 3 -9 8 0 6 -5 -8 10 -7 -9 5 -1 5
```

Y queremos filtrar todos los números positivos:

```
lista.stream().filter(n -> n >= 0)
```

Otro ejemplo: del array de frutas queremos mostrar todas las frutas que comiencen por "p".

```
streamArray.filter(s -> s.startsWith("p"))
```

- **Stream<T> distinct()**

Devuelve un stream con los elementos diferentes (sin repeticiones) del stream al que se aplica.

Ejemplo con la lista anterior:

```
lista.stream().distinct()
```

Si trabajamos con números podemos ordenarlos directamente con el método sort()

```
lista.stream().distinct().sort()
```

- **<R> Stream<R> map(Function<? super T,? extends R> mapper)**

Devuelve un stream que consiste en los resultados a aplicar la función a los elementos del stream.

Ejemplo: a partir de una lista de números mapeamos cada elemento a su cuadrado:

```
lista.stream().map(n -> n * n)
```

Otro ejemplo: a partir del array de frutas las ponemos todas en mayúsculas:

```
Stream.of(frutas).map(s -> s.toUpperCase());
```

Hay métodos parecidos que funcionan como map pero producen streams de datos primitivos en vez de objetos.: **mapToInt**, **mapToDouble**, **mapToLong**.

Por ejemplo, imaginemos que tenemos unos códigos en formato String que pueden ser convertidos a números. Lo haríamos así:

```
String[] códigos = {"001", "002", "003", "004"};

Arrays.stream(códigos)
    .mapToInt(cod -> Integer.parseInt(cod))
    .forEach(num -> System.out.print(num + " "));
```

## Operaciones terminales

---

- **Optional<T> findFirst()**

Devuelve un objeto de tipo Optional que describe el primer elemento del stream, o un Optional vacío si el stream está vacío.

Un objeto de tipo Optional sirve para evitar el problema de que salte una excepción en caso de que el objeto sea null. Optional tiene una serie de métodos, el más importante de los cuales para nosotros es .get(), que recupera el elemento que hay dentro del objeto de tipo Optional devuelto. Ejemplo:

```
List<Integer> lista1 = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> lista2 = Arrays.asList(5, 4, 3, 2, 1);
List<Integer> lista3 = Arrays.asList(6, 1, 5, 2, 1);
List<List<Integer>> listas = Arrays.asList(lista1, lista2, lista3);
```

```
// recorremos la lista de listas y en cada una,
for (int i = 0; i < listas.size(); i++) {
    System.out.print("Lista" + String.valueOf(i + 1) + ": ");
    // filtramos las listas que empiecen por números mayores que 3
    Optional<Integer> optionalInteger = listas.get(i).stream()
        .findFirst()
        .filter(n -> n > 3);

    // si el optional no contiene null, nos imprime el número que
    contiene
    if (optionalInteger.isPresent()) {
        System.out.println(optionalInteger.get());
    } else {
        System.out.println("Primer número <= 3");
    }
}
}
```

Resultado:

```
Lista1: Primer número <= 3
Lista2: 5
Lista3: 6
```

Para más información sobre Optional:

- **Object[] toArray()**

Devuelve un array que contiene los elementos del stream.

Ejemplo: queremos guardar una lista de números en un array:

```
int[] arrayNúmeros = lista.stream()
    .mapToInt(n -> n).toArray();
```

(Observa que para que esto funcione hemos tenido que mapear a enteros primitivos los elementos del stream).

## Resultados numéricos

---

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

Devuelve el elemento mayor del stream según el comparador proporcionado.

Recuerda que para obtener el valor que hay dentro del objeto Optional, hay que utilizar su método get().

Ejemplo:

```
lista.stream()
    .max((n1, n2) -> n1 - n2)
    .get()
```

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

Devuelve el elemento menor del stream según el comparador proporcionado.

Ejemplo:

```
lista.stream()
.min((n1, n2) -> n1 - n2)
.get()
```

- **long count()**

Devuelve cuántos elementos contiene el stream.

Ejemplo: queremos saber cuántas frutas tenemos

```
Stream.of(frutas).count()
```

- **T reduce(T identity, BinaryOperator<T> accumulator)**

Nos permite combinar (reducir) los elementos de un stream en un único valor. Para ello tenemos que expresar cómo se van acumulando los valores.

La función reduce utiliza la interfaz funcional BinaryOperator<T>, que recibe dos objetos del mismo tipo y opera con ellos para devolver un resultado del mismo tipo que los operandos.



El valor de identity es el de inicialización del acumulador.

Ejemplo:

```
List<Double> notas = Arrays.asList(6.0, 7.5, 5.5, 9.0, 8.1);

double total = notas.stream()
    .reduce(0.0, (a, b) -> a + b);

double promedio = total / notas.stream().count();

System.out.printf("Nota media: %.2f" , promedio);
```

## Referencias

---

Jiménez Martín, Alfonso y Pérez Montes, Francisco Manuel (2021). *Programación*. Editorial Paraninfo.

<https://www.arquitecturajava.com/que-es-un-java-optional/>

[Y un poco de ChatGPT](#)