

Métodos de la API Stream de Java 📁

1. Métodos de Filtrado

Estos métodos se usan para seleccionar elementos específicos del Stream según ciertas condiciones.

Método	Descripción	Parámetro esperado
filter	Filtra elementos según una condición.	Predicate<T> 🧠
distinct	Elimina elementos duplicados según equals().	-
limit	Limita el Stream a un número específico de elementos.	long (número de elementos) 🏠
skip	Salta los primeros N elementos y continúa con el resto.	long (número de elementos) 🏠

2. Métodos de Transformación o Mapeo

Estos métodos se utilizan para transformar los elementos de un Stream, generalmente creando un nuevo Stream con los resultados de la transformación.

Método	Descripción	Parámetro esperado
map	Aplica una función a cada elemento, transformándolo en otro valor.	Function<T, R> 🧴
flatMap	Descompone cada elemento en un Stream , luego los aplanan en uno solo.	Function<T, Stream<R>> 🧴
mapToInt	Transforma cada elemento en un int.	ToIntFunction<T> 🧴
mapToLong	Transforma cada elemento en un long.	ToLongFunction<T> 🧴
mapToDouble	Transforma cada elemento en un double.	ToDoubleFunction<T> 🧴

3. Métodos de Reducción y Acumulación

Estos métodos permiten reducir o acumular elementos de un Stream en un solo valor, como una suma, concatenación, etc.

Método	Descripción	Parámetro esperado
reduce	Combina los elementos en un solo resultado usando una función de acumulación.	BinaryOperator<T> o T identity, BinaryOperator<T> 🔧
collect	Recopila los elementos en una colección o estructura de datos.	Collector<T, A, R> 💰
count	Cuenta los elementos en el Stream.	-
max	Encuentra el elemento máximo según un comparador.	Comparator<T> ⚖️
min	Encuentra el elemento mínimo según un comparador.	Comparator<T> ⚖️

4. Métodos de Ordenación

Permiten ordenar los elementos en el Stream.

Método	Descripción	Parámetro esperado
sorted	Ordena los elementos de forma natural o usando un comparador.	Comparator<T> (opcional) 🏴‍☠️

5. Métodos de Acceso o Visita (Recorrer)

Estos métodos ejecutan una operación en cada elemento del Stream, generalmente para efectos secundarios.

Método	Descripción	Parámetro esperado
forEach	Aplica una operación a cada elemento del Stream.	Consumer<T> 🍷
forEachOrdered	Aplica una operación en orden al elemento, si es Stream paralelo	Consumer<T> 🍷
peek	Permite ejecutar una operación en cada elemento, sin modificar el Stream.	Consumer<T> 🍷

6. Métodos de Búsqueda y Coincidencia

Permiten verificar si ciertos elementos cumplen una condición o buscar elementos específicos.

Método	Descripción	Parámetro esperado
findFirst	Devuelve el primer elemento en el Stream, si existe.	-
findAny	Devuelve algún elemento en el Stream (útil en Stream paralelo)	-
anyMatch	Verifica si algún elemento cumple una condición dada.	Predicate<T> 🧠
allMatch	Verifica si todos los elementos cumplen una condición dada.	Predicate<T> 🧠
noneMatch	Verifica si ningún elemento cumple una condición dada.	Predicate<T> 🧠

7. Métodos de Conversión

Permiten transformar el Stream a otros tipos de Stream o colecciones.

Método	Descripción	Parámetro esperado
toArray	Convierte el Stream a un array.	IntFunction<A[]> (opcional) 🍼
collect	Recopila los elementos en una colección, lista, mapa, etc.	Collector<T, A, R> 💰

Métodos de la clase Optional de Java 🎁



1. Métodos de Creación

Método	Descripción	Parámetro esperado
empty	Retorna un Optional vacío.	-
of	Crea un Optional con un valor no nulo.	T (valor no nulo)
ofNullable	Crea un Optional que acepta valores nulos (o no).	T (valor que puede ser nulo)





2. Métodos de Consulta y Verificación

Método	Descripción	Parámetro esperado
isPresent	Verifica si el <code>Optional</code> contiene un valor.	-
isEmpty	Verifica si el <code>Optional</code> está vacío.	-




3. Métodos de Acceso y Recuperación

Método	Descripción	Parámetro esperado
get	Retorna el valor si está presente; lanza <code>NoSuchElementException</code> si está vacío.	-
orElse	Retorna el valor si está presente; si no, retorna el valor especificado.	<code>T</code> (valor alternativo)
orElseGet	Retorna el valor si está presente; si no, llama a un <code>Supplier</code> para obtenerlo.	<code>Supplier<? extends T></code> 
orElseThrow	Retorna el valor si está presente; si no, lanza una excepción usando un <code>Supplier</code> .	<code>Supplier<? extends X></code> 

4. Métodos de Transformación

Método	Descripción	Parámetro esperado
map	Aplica una función al valor, si está presente, y retorna un <code>Optional</code> con el resultado.	<code>Function<? super T, U></code> 
flatMap	Aplica una función que retorna un <code>Optional</code> y aplanar el resultado en un solo <code>Optional</code> .	<code>Function<? super T, Optional<U>></code>  
filter	Retorna un <code>Optional</code> vacío si el valor no cumple el <code>Predicate</code> dado.	<code>Predicate<? super T></code> 

5. Métodos de Ejecución Condicional

Método	Descripción	Parámetro esperado
ifPresent	Ejecuta una acción <code>Consumer</code> si el valor está presente.	<code>Consumer<? super T></code> 
ifPresentOrElse	Ejecuta una acción si el valor está presente; si no, ejecuta otra acción.	<code>Consumer<? super T></code>  , <code>Runnable</code> 

6. Otros Métodos de Utilidad

Método	Descripción	Parámetro esperado
stream	Convierte el <code>Optional</code> en un <code>Stream</code> (vacío o con un solo elemento).	-

Funciones en Streams

1. Consumer<T> 🍷

Descripción	Uso en Streams	Ejemplo
Representa una operación que acepta un argumento de entrada y no devuelve ningún resultado.	Se utiliza en métodos como <code>forEach</code> , <code>peek</code> , e <code>ifPresent</code> para realizar acciones sobre los elementos de un stream.	<pre>Stream<String> stream = Stream.of("a", "b", "c"); stream.forEach(s -> System.out.println(s)); // Imprime cada elemento</pre>

2. Supplier<T> 💊

Descripción	Uso en Streams	Ejemplo
Representa una función que no toma argumentos y devuelve un valor.	Se utiliza en métodos como <code>orElseGet</code> para proporcionar un valor alternativo que se genera bajo demanda.	<pre>Optional<String> optional = Optional.empty(); String result = optional.orElseGet(() -> "Valor por defecto"); // "Valor por defecto"</pre>

3. Predicate<T> 🍷

Descripción	Uso en Streams	Ejemplo
Representa una función que acepta un argumento y devuelve un valor booleano.	Se utiliza en métodos de filtrado como <code>filter</code> , <code>anyMatch</code> , <code>allMatch</code> , y <code>noneMatch</code> para evaluar condiciones sobre los elementos.	<pre>Stream<Integer> numbers = Stream.of(1, 2, 3, 4); long count = numbers.filter(n -> n % 2 == 0).count(); // 2 (solo el 2 y el 4 son pares)</pre>

4. Function<T, R> 🍷

Descripción	Uso en Streams	Ejemplo
Representa una función que acepta un argumento y produce un resultado.	Se utiliza en métodos como <code>map</code> y <code>flatMap</code> para transformar los elementos de un stream.	<pre>java Stream<String> stream = Stream.of("1", "2", "3"); Stream<Integer> intStream =stream.map(Integer::parseInt); // Convierte a enteros</pre>

5. UnaryOperator<T> 🍷

Descripción	Uso en Streams	Ejemplo
Es una especialización de <code>Function</code> que toma un argumento y devuelve un resultado del mismo tipo.	Se utiliza para transformar un elemento en otro del mismo tipo.	<pre>Stream<Integer> numbers = Stream.of(1, 2, 3); Stream<Integer> incremented = numbers.map(n -> n + 1); // Incrementa cada número</pre>

6. BinaryOperator<T> 🍷

Descripción	Uso en Streams	Ejemplo
Es una especialización de <code>BiFunction</code> que toma dos argumentos del mismo tipo y devuelve un resultado del mismo tipo.	Se utiliza en operaciones de reducción como <code>reduce</code> .	<pre>java Stream<Integer> numbers = Stream.of(1, 2, 3); Optional<Integer> sum = numbers.reduce((a, b) -> a + b); // Suma todos los números</pre>

7. ToIntFunction<T>, ToLongFunction<T>, ToDoubleFunction<T>

Descripción	Uso en Streams	Ejemplo
Estas interfaces son especializaciones de <code>Function</code> que convierten un valor a un tipo primitivo: <code>int</code> , <code>long</code> o <code>double</code> .	Se utilizan en métodos como <code>mapToInt</code> , <code>mapToLong</code> , y <code>mapToDouble</code> para transformar elementos a tipos primitivos.	<pre>Stream<String> stream = Stream.of("1.5", "2.5", "3.5"); DoubleStream doubleStream = stream.mapToDouble(Double::parseDouble); // Convierte a DoubleStream</pre>

8. BiConsumer<T, U>

Descripción	Uso en Streams	Ejemplo
Representa una operación que acepta dos argumentos de entrada y no devuelve ningún resultado.	Se puede usar en operaciones que requieren dos entradas, como en el método <code>forEach</code> de una colección que tiene pares de clave-valor.	<pre>Map<String, Integer> map = new HashMap<>(); map.put("a", 1); map.put("b", 2); map.forEach((key, value) -> System.out.println(key + ": " + value)); // Imprime cada par clave-valor</pre>

Collector y Comparator

Collector 💰

Los objetos `Collector` en Java son utilizados para recopilar o acumular los elementos de un stream en una estructura de datos específica, como una lista, conjunto o mapa. La clase `Collectors` ofrece métodos estáticos que permiten realizar esta operación de forma sencilla. Aquí tienes los principales métodos de `Collectors` en Java:

Método	Descripción	Ejemplo
<code>toList</code>	Recopila los elementos en una <code>List</code> .	<code>Stream.of("a", "b", "c").collect(Collectors.toList());</code> // [a, b, c]
<code>toSet</code>	Recopila los elementos en un <code>Set</code> .	<code>Stream.of("a", "b", "a").collect(Collectors.toSet());</code> // [a, b]
<code>toMap</code>	Recopila los elementos en un <code>Map</code> usando funciones para claves y valores.	<code>Stream.of("a", "b").collect(Collectors.toMap(String::toUpperCase, String::length));</code> // {A=1, B=1}
<code>joining</code>	Concatena los elementos de tipo <code>String</code> .	<code>Stream.of("a", "b", "c").collect(Collectors.joining(", "));</code> // "a, b, c"
<code>counting</code>	Cuenta el número de elementos en el stream.	<code>Stream.of("a", "b", "c").collect(Collectors.counting());</code> // 3
<code>summarizingInt / Double / Long</code>	Devuelve estadísticas como conteo, suma, media, min y max para tipos numéricos.	<code>Stream.of(1, 2, 3).collect(Collectors.summarizingInt(Integer::intValue));</code> // <code>IntSummaryStatistics</code> con <code>count=3</code> , <code>sum=6</code>
<code>groupingBy</code>	Agrupar los elementos de acuerdo con una función de clasificación.	<code>Stream.of("apple", "banana", "cherry")</code> <code>.collect(Collectors.groupingBy(String::length));</code> // {5=[apple], 6=[banana, cherry]}
<code>partitioningBy</code>	Particiona los elementos en dos grupos según una condición (<code>Predicate</code>).	<code>Stream.of(1, 2, 3, 4).collect(Collectors.partitioningBy(n -> n % 2 == 0));</code> // {false=[1, 3], true=[2, 4]}
<code>reducing</code>	Realiza una operación de reducción sobre los elementos con un acumulador inicial y una función de combinación.	<code>Stream.of(1, 2, 3).collect(Collectors.reducing(0, (a, b) -> a + b));</code> // 6

Comparator ⚖️

Los objetos **Comparator** en Java permiten comparar y ordenar elementos en un stream. La clase **Comparator** proporciona métodos estáticos y métodos de instancia para personalizar el ordenamiento de los elementos. Aquí están los métodos más comunes:

Método	Descripción	Ejemplo
naturalOrder	Ordena los elementos en orden natural.	<code>Stream.of(3, 1, 2).sorted(Comparator.naturalOrder()).collect(Collectors.toList());</code> // [1, 2, 3]
reverseOrder	Ordena los elementos en orden inverso al natural.	<code>Stream.of(1, 2, 3).sorted(Comparator.reverseOrder()).collect(Collectors.toList());</code> // [3, 2, 1]
comparing	Compara los elementos usando una función de clave para el ordenamiento.	<code>Stream.of("apple", "banana").sorted(Comparator.comparing(String::length)).collect(Collectors.toList());</code> // [apple, banana]
comparingInt / Double/Long	Compara elementos usando una función de clave que devuelve un primitivo específico.	<code>Stream.of("a", "bb", "ccc").sorted(Comparator.comparingInt(String::length)).collect(Collectors.toList());</code> // [a, bb, ccc]
thenComparing	Encadena comparadores para aplicar varios criterios de ordenamiento.	<code>Stream.of("ab", "b", "aa").sorted(Comparator.comparing(String::length).thenComparing(String::compareTo)).collect(Collectors.toList());</code> // [b, aa, ab]
nullsFirst	Ordena con null primero, luego aplica otro comparador especificado.	<code>Stream.of("b", null, "a").sorted(Comparator.nullsFirst(Comparator.naturalOrder())).collect(Collectors.toList());</code> // [null, a, b]
nullsLast	Ordena con null al final, luego aplica otro comparador especificado.	<code>Stream.of("a", null, "b").sorted(Comparator.nullsLast(Comparator.naturalOrder())).collect(Collectors.toList());</code> // [a, b, null]