Stream API

The **Stream API** in Java, introduced in **Java 8**, is a powerful feature that allows you to process collections of data (like List, Set,Map etc.) in a **functional programming style**. It provides a high-level abstraction for performing **functional-style operations** on sequences of elements, such as **filtering**, **mapping**, **sorting**, and **reducing**.

- A **stream** is not a data structure; it does not store data.
- It operates on the data from a **collection** or **array**.
- It allows for **declarative** (what to do) rather than **imperative** (how to do) coding.
- **Streams are lazy**; computation happens only when needed.

- Stream is a wrapper on data source
- Stream is not a data structure.it will take data Structure as input to perform operations.
- Strem will not disturb actual data source Object
- Operations of Stream will not impact the data source

Stream Initialization:

- Package “import java.util.stream.Stream ”

Ex:

```
package com.ashi;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;
import java.util.stream.Stream.Builder;
```

```java
public class MainApplication {

    public static void main(String[] args) {

        //1. From Collection Object
        List<String> values = List.of("one","two","three");

        Stream<String> stream = values.stream();

        //2. Array of values
        String[] names = {"a","b","c"};
        Stream<String> stream2 = Arrays.stream(names);

        //3.Stream methods
        Stream<String> stream3 =
Stream.of("one","two","three");


        //4. generate()

        Stream<String> stream4 = Stream.generate(()-
>"ashish");

        //5.bulider design : builder()

        Stream.Builder<Object> builder = Stream.builder();
        Stream<Object> builder2 =
builder.add("one").add("two").build();

        //6.empty()
        Stream<Object> empty = Stream.empty();



    }
}
```

Stream Initialization Method	Description
values.stream()	From a Collection
Arrays.stream(array)	From an array
Stream.of(...)	Directly create from values
Stream.generate(Supplier)	Infinite stream, good for repeating values or random data
Stream.builder()	Builder pattern for dynamically constructing streams
Stream.empty()	Returns a stream with no elements

Note :

- Streams of random numbers can be obtained from *Random.ints()*;
- From static factory methods on the stream classes, such as Stream.of(T[]), IntStream.range(int, int) or Stream.iterate(T, UnaryOperator);

Stream Operations:

Stream operations are of **two types**:

1. Intermediate Operations
2. Terminal Operations

By using this we build => Stream Pipeline

Intermediate Operations:

When we call a methods belongs to intermediate Operations, it will always return another Stream object of the operations results.

- We can call may intermediate operations on same instance as chain of methods.
- This is called as pipelined on same Stream source.
- Intermediate operations are executed only once when we invoke terminal operations.

Note :

- **Intermediate Operations:** Intermediate operations helps the stream pipeline to build the execution strategy. These are lazy in nature, they don't execute until any terminal operations are invoked. They don't modify the original stream, every time they return a new stream. Intermediate operations can again divided into stateless and stateful operations.
    - *Stateless* operations such as filter, map are processed independently of operations on other elements
    - *Stateful* operations such as sorted, distinct require to rememeber the result of operations on

already seen elements to calculate the result for next element. They execute the entire input before producing the final result.

Terminal Operations

We call only one terminal method on any stream instance

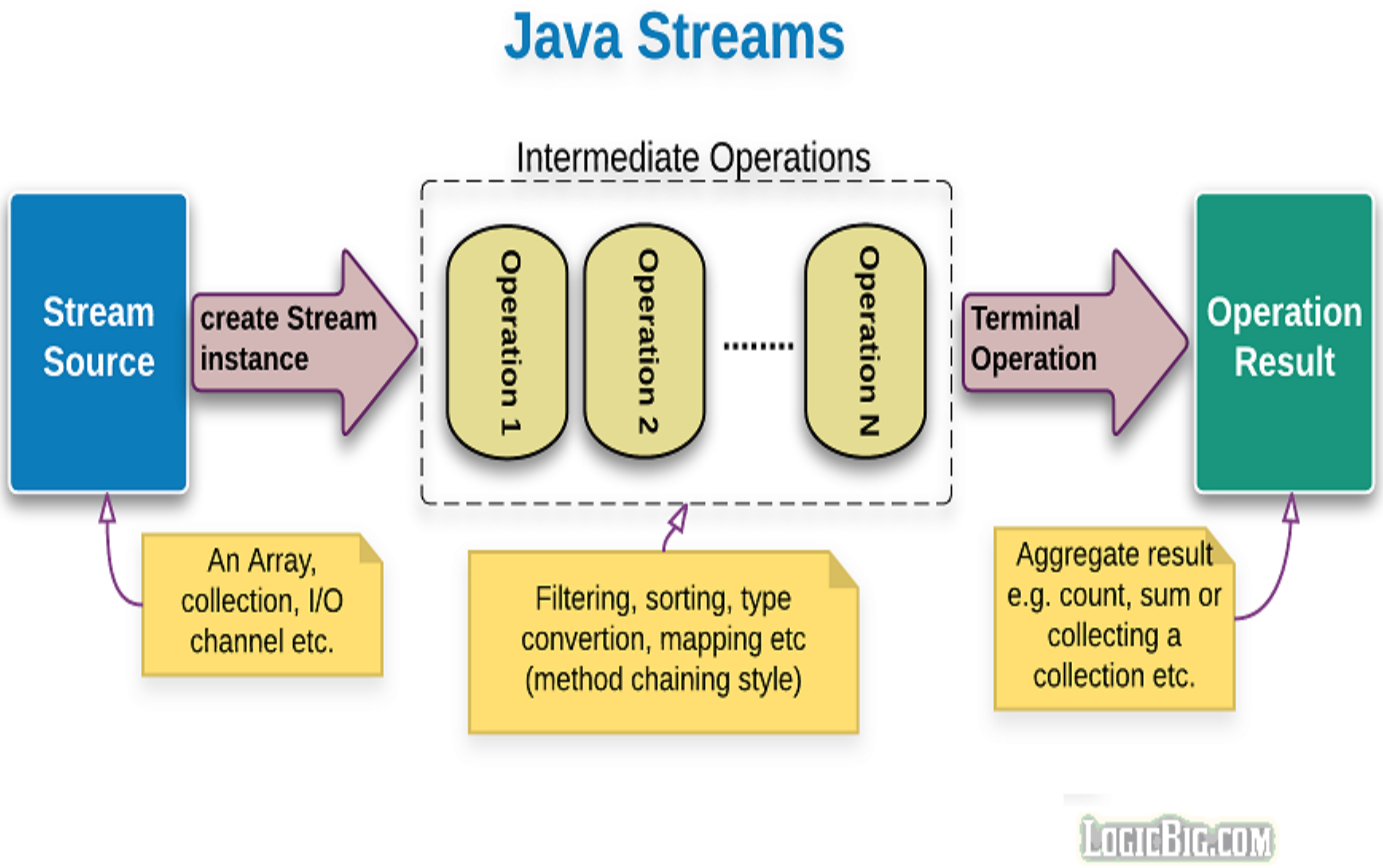
- When we call terminal method, immediately we will get the result of pipelined operations on the steam object.

Note:

- **Terminal Operation:** Terminal operation traverse the stream and execute the pipeline of intermediate operations to produce the result. They are eager in nature. After the terminal operation is performed, the stream pipeline is considered consumed, and can no longer be used. A stream implementation may throw IllegalStateException if it detects that the stream is being reused.

Stream Pipeline

A stream pipeline is nothing but combined intermediate and terminal operations



Note:

Streams are also generated from infinite dataset. Some of the stream operations can be tagged as *short-circuting operations* which acts on these infinite stream or data. An intermediate operation is said to be short-circuting if applying on infinite stream should produce finite stream. As an example `new Random().ints().limit(5)` will return only 5 random numbers. A terminal operation is short-circuting if, when applying on infinite set of input should produce result

in finite time. As an example `new Random().ints().filter(no -> no % 10 == 0).findAny()` will return any one random number divisible by 10.

What is infinite Stream ?

An **infinite stream** in Java is a stream that **does not have a predefined size or end** — it can **generate an unlimited number of elements**. These streams are often created using:

- `Stream.generate(Supplier<T>)`
- `Stream.iterate(seed, UnaryOperator<T>)`

Because they're infinite, you usually use **limit(n)** to control how many elements to process.

◇ 1. Stream.generate()

This method uses a **Supplier** to keep producing values.

Example:

```
Stream<String> infiniteStream = Stream.generate(() -> "hello");


infiniteStream.limit(3).forEach(System.out::println);
```

Output:

```
hello
hello
hello
```

◇ 2. Stream.iterate()

This method starts with a **seed** and applies a function repeatedly.

Example:

```
Stream<Integer> evenNumbers = Stream.iterate(0, n -> n + 2);


evenNumbers.limit(5).forEach(System.out::println);
```

Output:

Copy code

```
0

2

4

6

8
```

⚠ Important:

- Without .limit(), infinite streams will run **forever** and can **crash your program.**
- Use them when you want to generate sequences (like infinite counters, random values, etc.).

☑ Practical Use Cases

- Random number generation
- Fibonacci series generation

Methods in Stream

Intermediate Operations methods

Method	Full Definition
filter(Predicate<T> predicate)	Returns a stream consisting of the elements that match the given predicate (i.e., condition). It's used to filter out unwanted elements. Note: ***Returns a stream consisting of the elements of this stream that match the given predicate.***
map(Function<T, R> mapper)	Transforms each element of the stream by applying the given function. Used to convert or modify the stream data from one form to another. Note : ***Returns a stream consisting of the results of applying the given function to the elements of this stream.***
flatMap(Function<T, Stream<R>> mapper)	Replaces each element of the stream with the contents of a mapped stream and flattens the resulting streams into a single stream. Ideal for working with nested collections.
distinct()	Returns a stream with duplicate elements removed based on equals(). Preserves the order of the first occurrence.

Method	Full Definition
	Note: ***Returns a stream consisting of the distinct elements (according to Object.equals(Object)) of this stream.***
sorted() and **sorted(Comparator<T>)**	Returns a stream with elements sorted either by natural order or using a custom comparator provided. Note : *Returns a stream consisting of the elements of this stream, sorted according to natural order. If the elements of this stream are not Comparable, a java.lang.ClassCastException may be thrown when the terminal operation is executed.*
limit(long maxSize)	Truncates the stream to contain no more than the given number of elements. Commonly used with infinite streams to make them finite.
skip(long n)	Returns a stream that discards the first n elements. Useful for paging or offsetting in data processing. Note: *Returns a stream consisting of the remaining elements of this stream after discarding the first n elements of the stream. If this stream*

Method	Full Definition
	contains fewer than n elements then an empty stream will be returned.
peek(Consumer<T> action)	Allows performing an action on each element (like printing or logging) without modifying the stream. Mostly used for debugging. Note: *Returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream.*
takeWhile(Predicate<T> predicate) (Java 9+)	Returns a stream consisting of elements taken from the original stream **until** the predicate returns false. Once the condition fails, processing stops.
dropWhile(Predicate<T> predicate) (Java 9+)	Returns a stream consisting of the remaining elements of the stream **after** dropping the elements that satisfy the predicate. Stops dropping as soon as the predicate fails.

Terminal Operation methods

Method	Return Type	Description
forEach(Consumer<T>)	void	Performs an action for each element of the stream.
toArray()	Object[]	Collects stream elements into an array.
reduce(BinaryOperator<T>)	Optional<T>	Reduces elements to a single value using an associative accumulation function.
collect(Collector)	<R>	Performs a mutable reduction operation (e.g., convert to List, Set, Map).
min(Comparator<T>)	Optional<T>	Returns the minimum element using the provided comparator.
max(Comparator<T>)	Optional<T>	Returns the maximum element using the provided comparator.
count()	long	Returns the count of elements in the stream.
anyMatch(Predicate<T>)	boolean	Returns true if any element matches the given predicate.

Method	Return Type	Description
		Note: *Returns whether any elements of this stream match the provided predicate. May not evaluate the predicate on all elements if not necessary for determining the result. If the stream is empty then false is returned and the predicate is not evaluated.*
allMatch(Predicate<T>)	boolean	Returns true if all elements match the given predicate.
noneMatch(Predicate<T>)	boolean	Returns true if no elements match the given predicate. *Note:* *any one matching: false* *no one matching : true*
findFirst()	Optional<T>	Returns the first element (if present) in the stream.

Method	Return Type	Description
findAny()	Optional<T>	Returns any element (non-deterministic in parallel streams). Note:*Returns an Optional describing some element of the stream, or an empty Optional if the stream is empty.* *This is a short-circuiting terminal operation*.

Additional methods

Method	Return Type	Description
collect(Collectors.joining())	String	Concatenates stream of String elements into a single string. You can use separators, prefixes, and suffixes too.

Method	Return Type	Description
collect(Collectors.groupingBy())	Map<K, List<T>> or more	Groups elements by a classifier function. Useful for grouping data.
collect(Collectors.partitioningBy())	Map<Boolean, List<T>>	Splits the stream into two groups based on a predicate (true/false).
collect(Collectors.toMap())	Map<K, V>	Converts the stream into a map using key-value mappers.
collect(Collectors.summarizingInt())	IntSummaryStatistics	Collects statistics (count, sum, min, average, max) for int values.
collect(Collectors.averagingInt())	Double	Computes the average of int values in the stream.
collect(Collectors.counting())	Long	Counts the number of

Method	Return Type	Description
		elements in the stream.
collect(Collectors.mapping())	Collector<T, A, R>	Adapts a mapping function to be used downstream with other collectors.
collect(Collectors.collectingAndThen())	Result Type	Allows post-processing of collected result, e.g., making collections unmodifiable.
reduce(identity, accumulator)	T	Performs a reduction using an identity value and a binary operator.

Short-Circuit Operations in Java Stream API

Short-circuiting operations are terminal or intermediate operations that can terminate the processing of a stream early, without examining all elements. These are especially useful for performance optimization, particularly with large or infinite streams.

☑ **Short-Circuiting Terminal Operations**

Method	Description
anyMatch(Predicate<T>)	Returns true as soon as any element matches the predicate – short-circuits remaining elements.
allMatch(Predicate<T>)	Returns false as soon as any element fails the predicate – short-circuits.
noneMatch(Predicate<T>)	Returns false as soon as any element matches the predicate – short-circuits.
findFirst()	Stops as soon as the first element is found.
findAny()	Returns any one element and short-circuits (particularly useful with parallel streams).
limit(n)	Limits the stream to n elements – after n, no more elements are processed. **Note** :*Returns a stream consisting of the elements of this stream, truncated to be no longer than maxSize in length.*
takeWhile(Predicate<T>)_(Java 9+)	Processes elements until the predicate fails – after that, the stream is terminated.

Note :

`peek()` is an intermediate operation in the Stream API that allows you to inspect or debug elements in the stream without modifying them.

- Mainly used for **debugging**, **logging**, or **monitoring** the flow of data in the stream pipeline.

What is a Parallel Stream in Java?

A Parallel Stream in Java is a feature of the Stream API that allows multi-threaded processing of data. It divides the stream into multiple parts, processes them in parallel using multiple CPU cores, and then combines the results.

Creating parallel Stream:

Method	Example
From collection	list.parallelStream()
From Stream itself	stream.parallel()

Benefits

- Utilizes multiple CPU cores to improve performance for large datasets.
- Useful for data-heavy operations like filtering, mapping, reducing large lists.

🔍 Sequential Stream vs Parallel Stream

Feature	Sequential Stream	Parallel Stream
Processing	Processes elements one-by-one, in order	Splits elements and processes in multiple threads
Threading	Single-threaded	Multi-threaded (uses Fork/Join Pool)
Execution Order	Maintains encounter order	Order not guaranteed (unless using forEachOrdered())
Performance (small data)	Better for small datasets	Overhead can make it slower
Performance (large data)	Slower for CPU-heavy large data	Can be faster by using all available CPU cores
Code Example	list.stream()	list.parallelStream()
Use Case	When order is important or data is small	For large, independent, CPU-heavy operations
Debugging	Easier	Harder due to multi-threading
Side Effects / Shared State	Safer, more predictable	Must be cautious (avoid modifying shared state)

Note :

- Use parallelStream() only when:

    - The task is independent
    - The data is large
    - You are performing CPU-intensive work (not I/O)
- Use forEachOrdered() to preserve order in parallel streams, but it reduces performance.

Comparator in Java

The `Comparator<T>` interface in Java is used to define custom sorting logic for objects. It provides a way to compare two objects of type T and determine their order (e.g., ascending or descending).

☑ Basic Syntax

```
@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

- Returns:
  - negative → o1 comes before o2
  - zero → o1 and o2 are equal
  - positive → o1 comes after o2
- **Common Static Methods in `Comparator` (Java 8+)**

Method	Description
`comparing(Function<T, U>)`	Sort by field (e.g., name, age, etc.)
`thenComparing(Comparator<T>)`	For multi-level sorting (e.g., by name, then age)
`reversed()`	Reverses natural or custom order
`naturalOrder()` / `reverseOrder()`	Default comparators for Comparable types
`nullsFirst()` / `nullsLast()`	Handle `null` values while sorting

Using Comparator in Java Streams

The Comparator is commonly used with the **sorted()** intermediate operation of Java Streams to **sort elements** based on custom logic.

☑ **Syntax**

```
stream.sorted();                                    // Uses natural order

stream.sorted(Comparator);                          // Uses custom Comparator
```

☑ **Example 1: Sort Employees by Name (Ascending)**

```
employees.stream()
         .sorted(Comparator.comparing(Employee::getName))
         .forEach(System.out::println);
```

☑ **Example 2: Sort Employees by Salary (Descending)**

```
employees.stream()
```

```
.sorted(Comparator.comparing(Employee::getSalary).reversed()
)
        .forEach(System.out::println);
```

☑ **Example 3: Multi-level Sorting (Department → then Salary)**

```
employees.stream()

.sorted(Comparator.comparing(Employee::getDepartment)

.thenComparing(Employee::getSalary))
        .forEach(System.out::println);
```

☑ **Example 4: Handling null values in Sorting**

```
employees.stream()
        .sorted(Comparator.comparing(Employee::getCity,
                Comparator.nullsLast(String::compareTo)))
        .forEach(System.out::println);
```

☑ **Stream + Limit Example (Top 5 Highest Paid Employees)**

```
employees.stream()
```

```
.sorted(Comparator.comparing(Employee::getSalary).reversed()
)
        .limit(5)
        .forEach(System.out::println);
```

🧠 **When to Use in Streams:**

- Anytime you want sorted results (e.g., top N elements, leaderboard)
- Multi-level comparisons (thenComparing)
- With limit(), collect(), or forEach() for reporting or pagination

---

What is Comparator?

Comparator<T> is a **functional interface** in java.util used to define **custom sorting logic** for objects, **outside** the class.

☑️ **Why use it?**

Use Comparator when:

- The class doesn't implement Comparable.
- You need **multiple sorting options** for the same class.

Collectors:

☑ What is Collectors?

Collectors is a **utility class** that provides **factory methods** to create **Collector objects**, which are used with .collect() to transform a stream into a result like:

- List
- Set
- Map
- String
- Long (count)
- Even **grouped** or **partitioned** data

Think of it like a **toolbox** that gives you ready-to-use ways to finish working with a stream.

🔧 1. Collectors.toList()

Collects stream elements into a List.

```
List<String> list = Stream.of("a", "b", "c")
    .collect(Collectors.toList());
System.out.println(list); // [a, b, c]
```

🔧 2. Collectors.toSet()

Collects into a Set (removes duplicates).

```
Set<String> set = Stream.of("a", "b", "a")
    .collect(Collectors.toSet());
```

```
System.out.println(set); // [a, b]
```

🔧 3. Collectors.joining()

Joins strings in a stream into a single string.

```
String result = Stream.of("java", "stream", "api")
    .collect(Collectors.joining(" - "));
System.out.println(result); // java - stream - api
```

🔧 4. Collectors.counting()

Counts how many elements are in the stream.

```
long count = Stream.of("x", "y", "z")
    .collect(Collectors.counting());
System.out.println(count); // 3
```

🔧 5. Collectors.averagingInt() / averagingDouble()

Calculates average of numeric values.

```
List<Integer> list = Arrays.asList(10, 20, 30);

double avg = list.stream()
    .collect(Collectors.averagingInt(n -> n)); // use lambda
System.out.println(avg); // 20.0
```

🔧 6. Collectors.summingInt()

Sums up values in the stream.

```
int sum = list.stream()
    .collect(Collectors.summingInt(n -> n));
System.out.println(sum); // 60
```

🔧 7. Collectors.groupingBy() – GROUPING

Groups data into a map based on a key.

☑ Example: Group names by their length

```
List<String> names = List.of("Ash", "Ram", "Kiran", "Ajay", "Neha");

Map<Integer, List<String>> grouped = names.stream()
    .collect(Collectors.groupingBy(name -> name.length()));

System.out.println(grouped);
// Example: {3=[Ash, Ram], 5=[Kiran], 4=[Ajay, Neha]}
```

🔧 8. Collectors.partitioningBy() – PARTITIONING

Partitions data into two groups (true or false).

☑ Example: Partition even vs odd

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);
```

```
Map<Boolean, List<Integer>> partitioned = numbers.stream()
    .collect(Collectors.partitioningBy(n -> n % 2 == 0));

System.out.println(partitioned);
// {false=[1, 3, 5], true=[2, 4, 6]}
```

🔧 9. Collectors.mapping() – For Nested Grouping or Transformation

Useful when you're doing grouping + transformation.

☑ **Example: Group by length and map to uppercase**

```
List<String> names = List.of("ram", "ajay", "nina");

Map<Integer, List<String>> upperGrouped = names.stream()
    .collect(Collectors.groupingBy(
        name -> name.length(),
        Collectors.mapping(name -> name.toUpperCase(), Collectors.toList())
    ));

System.out.println(upperGrouped);
// {3=[RAM], 4=[AJAY, NINA]}
```

🔧 10. Collectors.toMap() – Convert Stream to Map

☑ **Example: number → square**

```
List<Integer> list = List.of(2, 3, 4);

Map<Integer, Integer> map = list.stream()
    .collect(Collectors.toMap(
        n -> n,       // key = number
        n -> n * n    // value = square
    ));

System.out.println(map); // {2=4, 3=9, 4=16}
```

❗ **Handling Duplicates in toMap()**

If keys are duplicated, toMap() throws an error unless you provide a **merge function**.

☑ **Example: Count frequency of elements**

```
List<Integer> nums = List.of(2, 3, 2, 4, 3, 3);

Map<Integer, Integer> freq = nums.stream()
    .collect(Collectors.toMap(
        n -> n,
        n -> 1,
        (oldVal, newVal) -> oldVal + newVal // merge duplicates
    ));
```

```
System.out.println(freq); // {2=2, 3=3, 4=1}
```

☑ Summary Table

Collector	Purpose	Result Type
toList()	Collect to list	List<T>
toSet()	Collect to set	Set<T>
joining(delimiter)	Join strings	String
counting()	Count elements	Long
summingInt()	Sum of int values	int
averagingInt()	Average of int values	double
groupingBy()	Group by key	Map<K, List<T>>
partitioningBy()	Partition by boolean	Map<Boolean, List<T>>
mapping()	Transform during grouping	Nested collector
toMap()	Convert to map	Map<K, V>

Difference between Stream and parallelStream():

Aspect	stream()	parallelStream()
Definition	Processes data **sequentially**, one element at a time	Processes data **concurrently** using multiple threads
Threading	Runs on the **main thread**	Utilizes **ForkJoinPool.commonPool()** (multi-threaded)
Performance	Slower for large datasets	Faster for large datasets (if tasks are independent)
Order of Execution	Preserves **encounter order**	**May not preserve order** (unless explicitly handled)
Use Case	Small data sets, simple operations	Large data sets, CPU-intensive or independent tasks
Example	list.stream().forEach(...)	list.parallelStream().forEach(...)
Overhead	No thread overhead	Has thread creation/switching overhead
Deterministic Output	Consistent results every time	May vary due to concurrent execution
Error Handling	Easier to debug and trace	Harder to debug due to multithreading
Best Practice	Default choice unless parallelism is needed	Use with care; only when clear performance benefit exists

Important Questions :

#	Question	Description / Sample Answer
1️⃣	**What is the Java Stream API?**	A Stream is a sequence of elements supporting **sequential and parallel operations**. It's introduced in Java 8 for **functional-style operations on collections.**
2️⃣	**Difference between stream() and parallelStream()?**	stream() processes elements **sequentially**, parallelStream() processes them **concurrently using multiple threads** (ForkJoinPool).
3️⃣	**What is the difference between map() and flatMap()?**	map() transforms each element, flatMap() flattens nested structures (like List<List<T>> to List<T>).👉 Example:list.stream().map(List::size) vs list.stream().flatMap(List::stream)
4️⃣	**How do you remove duplicates using Stream API?**	list.stream().distinct().collect(Collectors.toList())
5️⃣	**How do you sort a list using Stream API?**	list.stream().sorted().collect(Collectors.toList()) or with custom comparator:sorted(Comparator.comparing(Employee::getName))

#	Question	Description / Sample Answer
6	**What are intermediate and terminal operations?**	◇ Intermediate: map(), filter(), sorted() – lazy ◇ Terminal: collect(), forEach(), count() – triggers execution
7	**How do you count elements in a Stream?**	stream.count() or Collectors.counting()
8	**What does collect() do?**	It's a terminal operation that **reduces the stream** to a collection or a single value using a Collector.
9	**What is the use of Collectors.groupingBy()?**	Groups elements by a classifier function:Collectors.groupingBy(Employee::getDepartment)
10	**What is the difference between filter() and map()?**	filter() removes elements that don’t match a condition.map() transforms each element.
11	**Can a stream be reused?**	✕ No, a stream can only be consumed once. You'll get IllegalStateException if reused.
12	**What is peek() used for?**	Mainly for **debugging**, to look at elements **without modifying** them.
13	**How do you convert a List**	java list.stream().collect(Collectors.toMap(obj -> obj.getId(), obj -> obj));

#	Question	Description / Sample Answer
	to Map using streams?	
14	**What is reduce() used for?**	Combines elements of a stream into a **single result** (like sum, min, max). Example:stream.reduce(0, Integer::sum)
15	**How to handle nulls safely in streams?**	Filter them: .filter(Objects::nonNull) before processing.