Java Stream API

What is Stream API?

The **Stream API** in Java was introduced in **Java 8** as part of the `java.util.stream` package. It allows functional-style operations on collections of elements such as **map-reduce transformations** and **pipelines**.

A **Stream** is **not a data structure**, it is an abstraction that represents a **sequence of elements** and supports different kinds of **operations to perform computations** on those elements.

Characteristics of Streams

Property	**Description**
Not a Data Structure	Stream does not store data; it simply conveys elements from a source.
Lazy Evaluation	Operations are not executed until a terminal operation is invoked.
Can be Consumed Once	Once a stream is consumed, it cannot be reused.
Functional Style	Encourages using lambda expressions and method references.
Possibility of Parallelism	Can execute operations in **parallel** using `parallelStream()`.

Types of Streams

1. **Sequential Stream**
   - Processes elements in a single thread.
   - Invoked using `stream()` method.
2. **Parallel Stream**

- Splits tasks and processes them **in parallel** using multiple threads.
- Invoked using `parallelStream()` method.

Commonly Used Stream Methods

Method	Type	Description
`filter()`	Intermediate	Filters elements based on a condition (Predicate).
`map()`	Intermediate	Transforms each element.
`sorted()`	Intermediate	Sorts elements naturally or using a comparator.
`distinct()`	Intermediate	Removes duplicates from the stream.
`limit(n)`	Intermediate	Limits the result to the first 'n' elements.
`skip(n)`	Intermediate	Skips the first 'n' elements.
`forEach()`	Terminal	Performs an action for each element.
`collect()`	Terminal	Converts the stream back to a collection or other structure.
`count()`	Terminal	Counts the number of elements.
`reduce()`	Terminal	Reduces elements to a single value using an associative function.

Stream Operation Types

1. **Intermediate Operations**

   - Return another stream
   - Are **lazy** and executed only when a terminal operation is invoked
   - Examples: `filter()`, `map()`, `sorted()`, `distinct()`, etc.

2. **Terminal Operations**

   - Produce a result or a side-effect
   - Trigger the actual processing

    - Examples: `forEach()`, `collect()`, `count()`, `reduce()`.

Stream Pipeline Example

```
List<String> names = Arrays.asList("Vijay", "Shivam", "Ajay");
List<String> result = names.stream()
                           .filter(name -> name.startsWith("V"))
                           .map(String::toUpperCase)
                           .collect(Collectors.toList());
System.out.println(result); // [VIJAY]
```

Explanation:

- `stream()` – Source
- `filter()` – Intermediate
- `map()` – Intermediate
- `collect()` – Terminal

💎 Advantages of Stream API

- Reduces **boilerplate code** (no need for loops).
- **Improves readability** using declarative style.
- Supports **parallel processing**.
- Encourages **functional programming** practices.
- **Chainable operations** using method chaining.

⚠️ Limitations / Disadvantages

- Slightly harder to debug due to chained calls.
- Cannot reuse streams once consumed.
- Parallel streams can lead to performance issues if not handled properly.

📌 Points to Remember for Exams

- Stream API is part of **java.util.stream**.
- It works on **Collections only**, not on Arrays directly (but you can use `Arrays.stream()`).
- A stream **does not modify** the source collection.
- You can use stream operations on **Lists, Sets, and Maps** (indirectly via `.entrySet()` etc.).
- Best suited for **bulk data processing**, **data filtering**, **transformation**, and **aggregation**.

Stream API – Method-wise Explanation with Examples

Let’s use this base list for reference:

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

---

1 `filter()`

🔹 **Purpose**: Filters elements based on a condition.

🔹 **Use case**: Get even numbers from a list.

🔹 **How it works**: Returns a stream that includes only elements that match the given predicate.

```
numbers.stream()
       .filter(n -> n > 20)
       .forEach(System.out::println); // Output: 2 4 6
```

---

2 `map()`

🔹 **Purpose**: Transforms each element of the stream.

🔹 **Use case**: Square each number in the list.

🔹 **How it works**: Applies the provided function to each element and returns a new stream of the results.

```
numbers.stream()
       .map(n -> n * n)
       .forEach(System.out::println); // Output:
```

---

3 `forEach()`

- **Purpose**: Performs an action (like printing) for each element.
- **Use case**: Print each element of the list.
- **How it works**: A terminal operation that consumes the stream and applies the given action.

```java
numbers.forEach(n -> n>20); // Output: 1 2 3 4 5 6
```

---

4 `collect()`

- **Purpose**: Collects stream elements into a collection (List, Set, etc.)
- **Use case**: Get a list of even numbers.
- **How it works**: Terminal operation that gathers elements into another structure.

```java
List<Integer> evens = numbers.stream()
                             .filter(n -> n % 2 == 0)
                             .collect(Collectors.toList());
System.out.println(evens); // Output: [2, 4, 6]
```

---

5 `sorted()`

- **Purpose**: Sorts the stream elements in natural (ascending) order.
- **Use case**: Sort a list of numbers.
- **How it works**: Intermediate operation that returns a new sorted stream.

```java
List<Integer> unsorted = Arrays.asList(5, 1, 3, 4, 2);
unsorted.stream()
        .sorted()
        .forEach(System.out::print); // Output: 12345
```

---

6 `distinct()`

- **Purpose**: Removes duplicate elements from the stream.
- **Use case**: Get unique values from a list.
- **How it works**: Intermediate operation that filters out duplicates based on `equals()`.

```
List<Integer> withDuplicates = Arrays.asList(1, 2, 2, 3, 3, 4);
withDuplicates.stream()
              .distinct()
              .forEach(System.out::print); // Output: 1234
```

---

7 `limit()`

- **Purpose**: Limits the stream to a specific number of elements.
- **Use case**: Get the first 3 elements.
- **How it works**: Truncates the stream to contain only the first `n` elements.

```
numbers.stream()
       .limit(3)
       .forEach(System.out::println); // Output: 1 2 3
```

---

8 `skip()`

- **Purpose**: Skips the first `n` elements of the stream.
- **Use case**: Ignore first 3 elements.
- **How it works**: Returns a stream after discarding the first `n` elements.

```
numbers.stream()
       .skip(3)
       .forEach(System.out::println); // Output: 4 5 6
```

9 `count()`

- **Purpose**: Counts the number of elements in the stream.
- **Use case**: Count how many even numbers exist.
- **How it works**: Terminal operation that returns a `long`.

```
long evenCount = numbers.stream()
                        .filter(n -> n % 2 == 0)
                        .count();
System.out.println(evenCount); // Output: 3
```

---

10 `reduce(T identity, BinaryOperator<T> accumulator)`

- **Purpose**: Reduces all elements into a single value (like sum, product, min, max).
- **Use case**: Find the sum of all numbers.
- **How it works**: Takes an identity value and a function that combines two values.

```
int sum = numbers.stream()
                 .reduce(0, (a, b) -> a + b);
System.out.println(sum); // Output: 21
```