Java Stream API

Alan Biju
@itsmeambro

Java Stream API

1. Introduction

- Stream API was introduced in Java 8.
- It provides a functional programming approach to processing collections of data.
- Streams do not modify the original data; they return a new stream.

2. Features of Stream API

✅ Functional Programming – Uses lambda expressions.
✅ Lazy Execution – Operations are executed only when a terminal operation is called.
✅ Parallel Processing – Use parallelStream() for multi-threaded execution.
✅ Immutable Processing – Streams don't modify the original collection.
✅ Method Chaining – Operations are chained for better readability.

3. How Stream API Works

A stream pipeline consists of:

- Source – Collection, Arrays, I/O Channels, etc.
- Intermediate Operations – Transform the stream (Lazy).
- Terminal Operations – Produce a result (Trigger execution).

Example: Basic Stream Usage

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

List<Integer> evenSquares = numbers.stream()  // Convert List to Stream
    .filter(n -> n % 2 == 0)                 // Keep only even numbers
    .map(n -> n * n)                         // Square each number
    .toList();                               // Collect results into a List

System.out.println(evenSquares); // Output: [4, 16, 36]
```

4. Creating Streams

From a Collection (List, Set, etc.)

```
List<String> names = List.of("Alice", "Bob", "Charlie");
Stream<String> stream = names.stream();
```

From an Array

```
  Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
```

From a Range (IntStream, LongStream)

```
  IntStream.range(1, 5).forEach(System.out::println);  // Output: 1 2 3 4
```

From Files (Large Data Processing)

```
  try (Stream<String> lines = Files.lines(Paths.get("data.txt"))) {
    lines.filter(line ->
  line.contains("Java")).forEach(System.out::println);
  } catch (IOException e) {
    e.printStackTrace();
  }
```

5. Intermediate Operations (Transforming Data)

These operations return a new Stream and are Lazy.

1. filter(Predicate<T>) — Filters elements

```
  List<Integer> evens = numbers.stream()
    .filter(n -> n % 2 == 0)
    .toList();
```

2. map(Function<T, R>) — Transforms elements

```
  List<String> upperCaseNames = names.stream()
    .map(String::toUpperCase)
    .toList();
```

3. sorted() — Sorts elements

```
  List<Integer> sortedNumbers = numbers.stream()
    .sorted()
    .toList();
```

Sorting by Custom Comparator

```
  List<Employee> sortedEmployees = employees.stream()
    .sorted(Comparator.comparingInt(Employee::getSalary).reversed())
    .toList();
```

4. distinct() – Removes duplicates

```
List<Integer> uniqueNumbers = numbers.stream()
  .distinct()
  .toList();
```

5. limit(n) – Limits the number of elements

```
List<Integer> firstThree = numbers.stream()
  .limit(3)
  .toList();
```

6. skip(n) – Skips first n elements

```
List<Integer> remaining = numbers.stream()
  .skip(3)
  .toList();
```

7. flatMap() – Flattens nested lists

```
List<List<String>> nestedLists = List.of(List.of("A", "B"),
   List.of("C", "D"));
List<String> flatList = nestedLists.stream()
  .flatMap(List::stream)
  .toList();
```

6. Terminal Operations (Ending Stream Processing)

These operations trigger execution and return a result (non-stream).

1. forEach(Consumer<T>) – Iterates through elements

```
numbers.stream().forEach(System.out::println);
```

2. collect(Collectors.toList()) – Collects data into a List, Set, or Map

```
Set<String> uniqueNames = names.stream()
 .collect(Collectors.toSet());
Map<String, Integer> employeeMap = employees.stream()
 .collect(Collectors.toMap(Employee::getName,
      Employee::getSalary));
```

3. count() – Counts elements

```
long count = numbers.stream().filter(n -> n > 5).count();
```

4. reduce() – Reduces elements to a single value

Find Sum

```
      int sum = numbers.stream().reduce(0, Integer::sum);
```

Find Maximum

```
      Optional<Integer> max = numbers
               .stream()
               .reduce(Integer::max);
```

5. findFirst() – Returns the first element

```
      Optional<String> first = names.stream().findFirst();
```

6. anyMatch(), allMatch(), noneMatch() – Check conditions

```
      boolean allEven = numbers.stream()
               .allMatch(n -> n % 2 == 0);
      boolean anyEven = numbers.stream()
               .anyMatch(n -> n % 2 == 0);
      boolean noneNegative = numbers.stream()
               .noneMatch(n -> n < 0);
```

7. Grouping & Partitioning

Grouping by Department

```
Map<String, List<Employee>> employeesByDept = employees.stream()
     .collect(Collectors.groupingBy(Employee::getDepartment));
```

Partitioning into True/False Groups

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

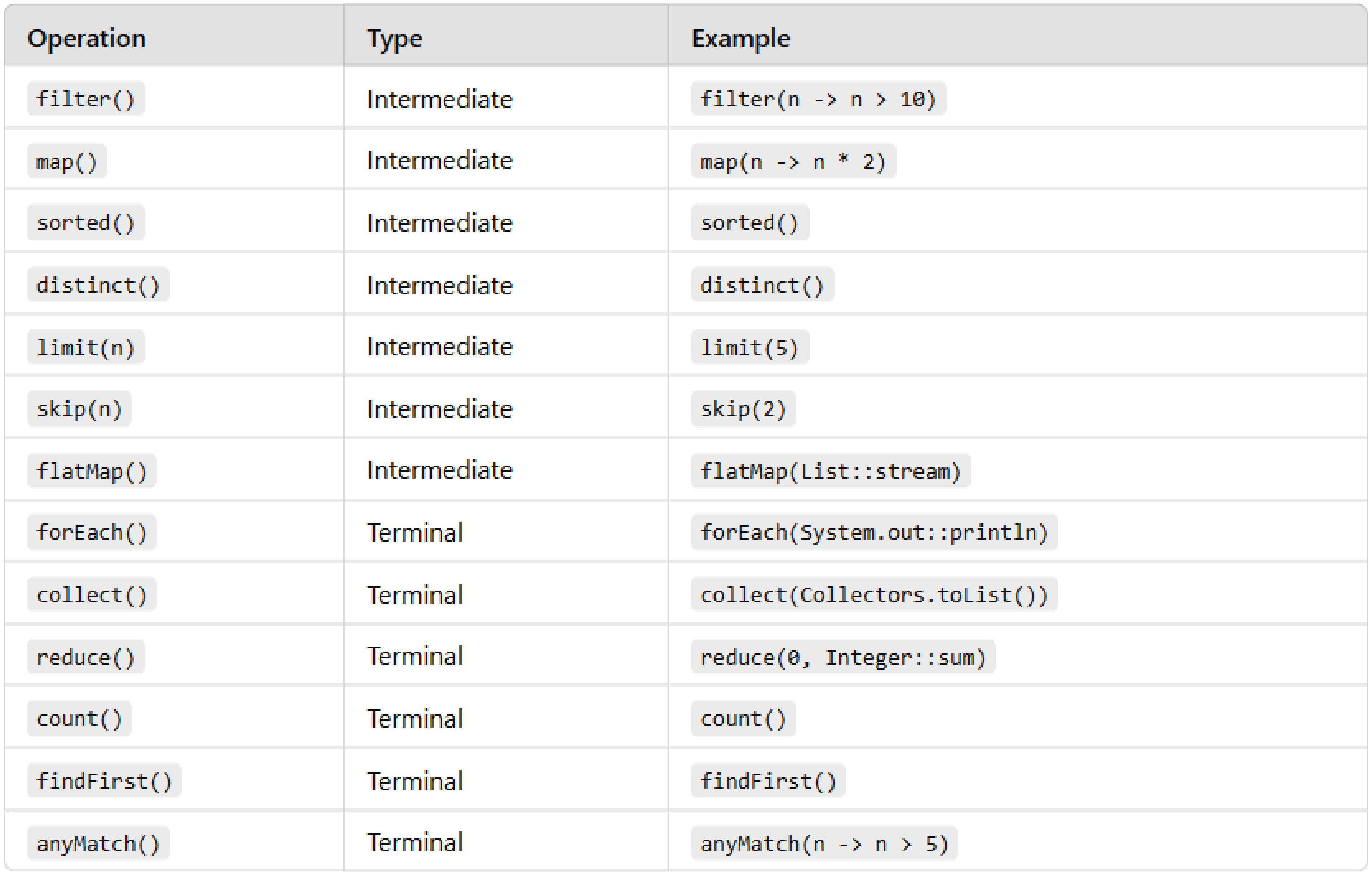
8. Parallel Streams (Performance Optimization)

For large datasets, use parallelStream() to speed up execution.

```
   long count = numbers.parallelStream()
       .filter(n -> n > 10)
       .count();
```

9. Performance Considerations

- ✅ Use sequential streams for small datasets.
- ✅ Use parallel streams for large datasets where CPU-bound tasks are involved.
- ✅ Avoid modifying shared mutable states inside streams (side-effects).



Operation	Type	Example
`filter()`	Intermediate	`filter(n -> n > 10)`
`map()`	Intermediate	`map(n -> n * 2)`
`sorted()`	Intermediate	`sorted()`
`distinct()`	Intermediate	`distinct()`
`limit(n)`	Intermediate	`limit(5)`
`skip(n)`	Intermediate	`skip(2)`
`flatMap()`	Intermediate	`flatMap(List::stream)`
`forEach()`	Terminal	`forEach(System.out::println)`
`collect()`	Terminal	`collect(Collectors.toList())`
`reduce()`	Terminal	`reduce(0, Integer::sum)`
`count()`	Terminal	`count()`
`findFirst()`	Terminal	`findFirst()`
`anyMatch()`	Terminal	`anyMatch(n -> n > 5)`

If you find this helpful, like and share it with your friends

Alan Biju
@itsmeambro