Java Streams - Comprehensive Guide

Designed by Naveen Automation Labs

Table of Contents

1. Introduction to Streams
2. Creating Streams
3. Stream Operations
   - Intermediate Operations
   - Terminal Operations
4. Common Stream Operations with Examples
5. Specialized Streams
6. Parallel Streams
7. Best Practices and Common Pitfalls
8. Practice Exercises

Introduction to Streams

Java Streams were introduced in Java 8 as part of the Java Collections Framework and represent a revolutionary approach to processing collections of data. A Stream is a sequence of elements supporting sequential and parallel aggregate operations.

Key Characteristics of Streams

1. **Not a Data Structure**: Streams don't store data; they convey elements from a source through a pipeline of operations.
2. **Functional in Nature**: Stream operations use lambda expressions and method references for behavior parameterization.
3. **Laziness-seeking**: Many stream operations are implemented lazily, executing only when necessary.
4. **Possibly Unbounded**: Collections have a finite size, but streams need not. Operations like `limit(n)` can make infinite streams finite.
5. **Consumable**: Elements of a stream are visited only once during the life of a stream.

Stream Pipeline Structure

A typical stream pipeline consists of:

1. **Source**: Where the stream comes from (e.g., a Collection, an array, a generator function)

2. **Intermediate Operations**: Transform the stream into another stream (e.g., `filter`, `map`)
3. **Terminal Operation**: Produces a result or side-effect (e.g., `collect`, `forEach`)

Creating Streams

There are several ways to create streams in Java:

1. From Collections

java

```
List<String> names = Arrays.asList("Naveen", "Bob", "Charlie");
Stream<String> nameStream = names.stream();
```

2. From Arrays

java

```
String[] namesArray = {"Alice", "Bob", "Charlie"};
Stream<String> arrayStream = Arrays.stream(namesArray);
```

3. From Static Factory Methods

java

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

// Stream.iterate (infinite)
Stream<Integer> infiniteStream = Stream.iterate(0, n -> n + 2); // even numbers

// Stream.generate (infinite)
Stream<Double> randomNumbers = Stream.generate(Math::random);
```

4. From File Lines (Java NIO)

java

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

5. From String Characters

```java
String str = "hello";
IntStream charStream = str.chars(); // Returns IntStream of char values
```

Stream Operations

Stream operations are divided into two categories: intermediate and terminal operations.

Intermediate Operations

Intermediate operations return a new stream. They are lazy; they don't perform any processing until a terminal operation is invoked.

Key intermediate operations include:

Operation	Description	Example
`filter(Predicate<T>)`	Filters elements based on a predicate	`stream.filter(n -> n > 5)`
`map(Function<T,R>)`	Transforms elements using a function	`stream.map(String::toUpperCase)`
`flatMap(Function<T,Stream<R>>)`	Transforms and flattens	`stream.flatMap(s -> Arrays.stream(s.split("")))`
`distinct()`	Removes duplicates	`stream.distinct()`
`sorted()`	Sorts elements (natural ordering)	`stream.sorted()`
`sorted(Comparator<T>)`	Sorts using a comparator	`stream.sorted(Comparator.reverseOrder()`
`peek(Consumer<T>)`	Performs action on elements	`stream.peek(System.out::println)`
`limit(long n)`	Truncates stream to n elements	`stream.limit(5)`
`skip(long n)`	Skips first n elements	`stream.skip(2)`

Terminal Operations

Terminal operations produce a result or a side-effect. After a terminal operation is performed, the stream pipeline is considered consumed.

Key terminal operations include:

Operation	Description	Example
`forEach(Consumer<T>)`	Performs action for each element	`stream.forEach(System.out::println)`
`collect(Collector<T,A,R>)`	Accumulates elements into a collection	`stream.collect(Collectors.toList())`
`reduce(BinaryOperator<T>)`	Reduces elements to a single value	`stream.reduce(0, Integer::sum)`
`count()`	Counts elements	`stream.count()`
`anyMatch(Predicate<T>)`	Checks if any elements match predicate	`stream.anyMatch(s -> s.startsWith("A"))`
`allMatch(Predicate<T>)`	Checks if all elements match predicate	`stream.allMatch(n -> n > 0)`
`noneMatch(Predicate<T>)`	Checks if no elements match predicate	`stream.noneMatch(n -> n < 0)`
`findFirst()`	Returns first element (Optional)	`stream.findFirst()`
`findAny()`	Returns any element (Optional)	`stream.findAny()`
`min(Comparator<T>)`	Returns minimum element	`stream.min(Comparator.naturalOrder())`
`max(Comparator<T>)`	Returns maximum element	`stream.max(Comparator.naturalOrder())`
`toArray()`	Converts stream to array	`stream.toArray()`

Common Stream Operations with Examples

Filtering Elements

```java
List<String> names = Arrays.asList("Naveen", "Bob", "Charlie", "David", "Eva");
List<String> filteredNames = names.stream()
    .filter(name -> name.length() > 4)
    .collect(Collectors.toList());
// Result: [Naveen, Charlie, David]
```

Transforming Elements

```java
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
List<Integer> nameLengths = names.stream()
    .map(String::length)
    .collect(Collectors.toList());
// Result: [5, 3, 7]
```

Flattening Nested Collections

```java
List<List<Integer>> nestedLists = Arrays.asList(
    Arrays.asList(1, 2, 3),
    Arrays.asList(4, 5, 6),
    Arrays.asList(7, 8, 9)
);

List<Integer> flatList = nestedLists.stream()
    .flatMap(Collection::stream)
    .collect(Collectors.toList());
// Result: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Sorting

java

```java
List<String> names = Arrays.asList("Charlie", "Naveen", "Bob", "Eva", "David");

// Natural ordering
List<String> sortedNames = names.stream()
    .sorted()
    .collect(Collectors.toList());
// Result: [Bob, Charlie, David, Eva, Naveen]

// Custom ordering
List<String> sortedByLength = names.stream()
    .sorted(Comparator.comparing(String::length))
    .collect(Collectors.toList());
// Result: [Bob, Eva, David, Naveen, Charlie]
```

Aggregating with reduce()

java

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

// Sum of all numbers
int sum = numbers.stream()
    .reduce(0, Integer::sum);
// Result: 15

// Finding maximum
int max = numbers.stream()
    .reduce(Integer.MIN_VALUE, Integer::max);
// Result: 5

// Concatenating strings
String concatenated = Stream.of("A", "B", "C")
    .reduce("", String::concat);
// Result: "ABC"
```

Collecting Results

java

```java
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David", "Eva");

// To List
List<String> namesList = names.stream()
    .filter(n -> n.length() > 3)
    .collect(Collectors.toList());

// To Set
Set<String> namesSet = names.stream()
    .filter(n -> n.length() > 3)
    .collect(Collectors.toSet());

// To Map
Map<String, Integer> nameLengthMap = names.stream()
    .collect(Collectors.toMap(
        Function.identity(),  // Key mapper
        String::length        // Value mapper
    ));
// Result: {Naveen=6, Bob=3, Charlie=7, David=5, Eva=3}

// Joining strings
String joined = names.stream()
    .collect(Collectors.joining(", "));
// Result: "Naveen, Bob, Charlie, David, Eva"

// Grouping
Map<Integer, List<String>> groupedByLength = names.stream()
    .collect(Collectors.groupingBy(String::length));
// Result: {3=[Bob, Eva], 5=[David], 6=[Naveen], 7=[Charlie]}

// Partitioning
Map<Boolean, List<String>> partitioned = names.stream()
    .collect(Collectors.partitioningBy(n -> n.length() > 4));
// Result: {false=[Bob, Eva], true=[Naveen, Charlie, David]}
```

Statistical Operations

java

```java
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

// Count
long count = numbers.stream().count();
// Result: 10

// Sum
int sum = numbers.stream().mapToInt(Integer::intValue).sum();
// Result: 55

// Average
OptionalDouble average = numbers.stream().mapToInt(Integer::intValue).average();
// Result: OptionalDouble[5.5]

// Min and Max
OptionalInt min = numbers.stream().mapToInt(Integer::intValue).min();
// Result: OptionalInt[1]
OptionalInt max = numbers.stream().mapToInt(Integer::intValue).max();
// Result: OptionalInt[10]

// Statistics
IntSummaryStatistics stats = numbers.stream().mapToInt(Integer::intValue).summaryStati
// Result: IntSummaryStatistics{count=10, sum=55, min=1, average=5.500000, max=10}
```

Short-Circuiting Operations

java

```java
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David", "Eva");

// anyMatch
boolean anyStartsWithN = names.stream()
    .anyMatch(name -> name.startsWith("N"));
// Result: true

// allMatch
boolean allLongerThan2 = names.stream()
    .allMatch(name -> name.length() > 2);
// Result: true

// noneMatch
boolean noneStartWithZ = names.stream()
    .noneMatch(name -> name.startsWith("Z"));
// Result: true

// findFirst
Optional<String> first = names.stream()
    .filter(name -> name.startsWith("D"))
    .findFirst();
// Result: Optional[David]

// findAny (may return any matching element, useful in parallel streams)
Optional<String> any = names.stream()
    .filter(name -> name.length() > 3)
    .findAny();
// Result: Optional containing any matching name
```

Specialized Streams

Java provides specialized stream classes for primitives:

IntStream, LongStream, DoubleStream

```java
// Creating IntStream
IntStream intStream = IntStream.range(1, 6); // 1, 2, 3, 4, 5
IntStream closedRange = IntStream.rangeClosed(1, 5); // 1, 2, 3, 4, 5

// Mapping to specialized stream
List<String> names = Arrays.asList("Naveen", "Bob", "Charlie");
IntStream lengths = names.stream()
    .mapToInt(String::length); // Returns IntStream instead of Stream<Integer>

// Boxing back to Stream<Integer>
Stream<Integer> boxedStream = IntStream.range(1, 6).boxed();

// Statistical operations on primitive streams
double avg = IntStream.rangeClosed(1, 100).average().orElse(0);
// Result: 50.5
```

Parallel Streams

Parallel streams allow you to perform operations concurrently, potentially improving performance on large data sets.

```java
// Creating parallel stream from a collection
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
int sum = numbers.parallelStream()
    .reduce(0, Integer::sum);

// Converting sequential stream to parallel
int sumParallel = numbers.stream()
    .parallel()
    .reduce(0, Integer::sum);
```

When to Use Parallel Streams

- **Dataset Size**: Use parallel streams for large datasets where operations are CPU-intensive
- **Independence**: Operations should be independent and not relying on state or order
- **Data Structure**: Some collections like ArrayList, arrays are better for parallelization than LinkedList
- **Hardware**: More CPU cores means better parallel performance

Caveats with Parallel Streams

- **Overhead**: There's overhead in parallelizing operations, so small datasets may run slower
- **Order**: Parallel streams may not preserve encounter order unless explicitly requested
- **Non-associative operations**: Operations must be associative to work properly in parallel
- **Side-effects**: Side-effects in stream operations can cause unexpected results

```java
// Example showing potential issues with parallelism and state
// DO NOT DO THIS:
StringBuilder sb = new StringBuilder();
IntStream.range(0, 1000)
    .parallel()
    .forEach(i -> sb.append(i)); // Incorrect: shared mutable state
// Result will be unpredictable!

// CORRECT APPROACH:
String result = IntStream.range(0, 1000)
    .parallel()
    .mapToObj(String::valueOf)
    .collect(Collectors.joining());
```

Best Practices and Common Pitfalls

Best Practices

1. **Favor Method References**: When possible, use method references instead of lambda expressions

    ```java
    // Instead of:
    stream.map(s -> s.toUpperCase())
    // Use:
    stream.map(String::toUpperCase)
    ```

2. **Avoid Side Effects**: Stream operations should not modify shared state

    ```java
    // BAD:
    List<String> collected = new ArrayList<>();
    stream.forEach(s -> collected.add(s));

    // GOOD:
    List<String> collected = stream.collect(Collectors.toList());
    ```

3. **Use Specialized Streams**: For primitive types, use IntStream, LongStream, DoubleStream

4. **Chain Operations Thoughtfully**: Order operations for maximum efficiency

   java

   ```
   // INEFFICIENT (filters all elements, then limits):
   stream.filter(predicate).limit(n)

   // EFFICIENT (stops filtering after finding n matching elements):
   stream.limit(n).filter(predicate)
   ```

5. **Use Parallel Streams Judiciously**: Not all operations benefit from parallelism

Common Pitfalls

1. **Reusing Streams**: Streams can only be operated on once

   java

   ```
   Stream<String> stream = list.stream();
   long count = stream.count();
   List<String> collected = stream.collect(Collectors.toList()); // ERROR: stream alre
   ```

2. **Ignoring Return Values**: Intermediate operations return new streams

   java

   ```
   // INCORRECT:
   stream.filter(predicate); // Does nothing without terminal operation

   // CORRECT:
   stream.filter(predicate).collect(Collectors.toList());
   ```

3. **Infinite Streams Without Limits**: Always limit infinite streams

   java

   ```
   // Will never terminate:
   Stream.iterate(0, n -> n + 1).forEach(System.out::println);

   // Correct:
   Stream.iterate(0, n -> n + 1).limit(100).forEach(System.out::println);
   ```

4. **Non-Deterministic Parallel Operations**: Some operations may produce different results when parallelized

   java

   ```
   // May return different values in parallel:
   list.parallelStream().findAny()
   ```

5. **Overlooking Collector Methods**: Many common operations have dedicated collector methods

    ```java
    // Instead of:
    stream.filter(s -> s.length() > 3).collect(Collectors.toList());

    // Consider:
    stream.collect(Collectors.filtering(s -> s.length() > 3, Collectors.toList()));
    ```

Practice Exercises

Exercise 1: Basic Stream Operations

Given a list of integers, perform the following operations:

1. Filter out odd numbers
2. Double each remaining number
3. Sum the resulting values

```java
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
int sum = numbers.stream()
    .filter(n -> n % 2 == 0)  // Keep only even numbers
    .mapToInt(n -> n * 2)     // Double each number
    .sum();                   // Sum them up
System.out.println(sum);      // Result: 60 (2*2 + 4*2 + 6*2 + 8*2 + 10*2)
```

Exercise 2: Stream with Objects

Given a list of Person objects with name and age, find the average age of people whose name starts with 'A':

```java
class Person {
    private String name;
    private int age;

    // Constructor, getters, setters...

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() { return name; }
    public int getAge() { return age; }
}

List<Person> people = Arrays.asList(
    new Person("Naveen", 25),
    new Person("Bob", 30),
    new Person("Nina", 20),
    new Person("Nick", 35),
    new Person("Charlie", 40)
);

double averageAge = people.stream()
    .filter(p -> p.getName().startsWith("N"))
    .mapToInt(Person::getAge)
    .average()
    .orElse(0);

System.out.println(averageAge);  // Result: 26.67
```

Exercise 3: Complex Collection Transformation

Given a list of sentences, count the frequency of each word:

```java
List<String> sentences = Arrays.asList(
    "Hello world",
    "Hello Java",
    "Java streams are powerful",
    "Streams in Java"
);

Map<String, Long> wordFrequency = sentences.stream()
    .flatMap(sentence -> Arrays.stream(sentence.toLowerCase().split("\\s+")))
    .collect(Collectors.groupingBy(
        Function.identity(),
        Collectors.counting()
    ));

System.out.println(wordFrequency);
// Result: {hello=2, world=1, java=3, streams=2, are=1, powerful=1, in=1}
```

Exercise 4: Custom Collector

Create a custom collector to concatenate strings with a prefix, delimiter, and suffix:

```java
List<String> words = Arrays.asList("apple", "banana", "cherry", "date");

String result = words.stream()
    .collect(Collectors.joining(
        ", ",           // delimiter
        "Fruits: [",    // prefix
        "]"             // suffix
    ));

System.out.println(result);
// Result: "Fruits: [apple, banana, cherry, date]"
```
