✨ Java Stream Operations Overview ✨

📜 Introduction

Java Streams, introduced in Java 8, provide a powerful and declarative way to process sequences of elements. They allow for functional-style operations on collections, enabling concise and potentially parallelized data manipulation. Streams support two main types of operations: **Intermediate** and **Terminal**.

Additional Resources

If you found this guide helpful, you might also be interested in my other Spring Framework resources:

- Core Java & Java-8 Interview Questions
- Spring Boot Interview Questions
- Microservices with Spring Cloud Tutorials

Feel free to star and fork these repositories if you find them useful!

⚙️ Intermediate Operations

> Intermediate operations return another `Stream` as a result, allowing them to be chained together to form a processing pipeline. A key characteristic is that they are ***lazy***; they don't execute until a terminal operation is invoked on the stream pipeline.

Common Examples:

- `filter(Predicate<T>)`
- `map(Function<T, R>)`
- `flatMap(Function<T, Stream<R>>)`
- `mapMulti(BiConsumer<T, Consumer<R>>)` (Java 16+)
- `distinct()`
- `sorted()` / `sorted(Comparator<T>)`
- `peek(Consumer<T>)` (Mainly for debugging)
- `limit(long)`
- `skip(long)`
- `takeWhile(Predicate<T>)` (Java 9+)

- `dropWhile(Predicate<T>)` (Java 9+)

🏁 Terminal Operations

> Terminal operations produce a non-stream result, such as a primitive value, an object, a collection, or simply perform a side effect (like `forEach`). They trigger the ***eager*** execution of the entire stream pipeline (including all chained intermediate operations). A stream pipeline can have at most one terminal operation, which must be the final operation.

Common Examples:

- `forEach(Consumer<T>)` / `forEachOrdered(Consumer<T>)`
- `toArray()` / `toArray(IntFunction<A[]>)`
- `reduce(...)`
- `collect(...)`
- `toList()` (Java 16+)
- `min(Comparator<T>)` / `max(Comparator<T>)`
- `count()`
- `anyMatch(Predicate<T>)` / `allMatch(Predicate<T>)` / `noneMatch(Predicate<T>)`
- `findFirst()` / `findAny()`

⚖️ Intermediate vs. Terminal Operations

Feature	Intermediate Operations	Terminal Operations
Return Type	Returns a `Stream`.	Returns a non-stream value (primitive, object, collection, `void`).
Chaining	Can be chained together to form a pipeline.	Cannot be chained after; terminates the pipeline.
Pipeline Composition	Pipeline can contain any number of intermediate ops.	Pipeline can have max one terminal op, always at the end.
Execution (Laziness)	**Lazy** (execution deferred until terminal op).	**Eager** (triggers pipeline execution).
Result	Do not produce the final result themselves.	Produce the final result or side effect of the pipeline.

🚀 Stream Creation Methods

Here are various ways to create a stream:

- ***Differnt ways to create a Stream.***

```
import java.util.stream.Stream;
// ... other necessary imports

Stream<String> emptyStream = Stream.empty();  // Create an
empty stream.
Stream<String> singleElementStream = Stream.of("apple");  //
Create a stream with a single element.
Stream<String> multiElementStream = Stream.of("apple",
"banana", "cherry"); // Create a stream from multiple
elements.

// Create a stream from a value that might be null (becomes
empty stream if null).
Stream<String> nullableStream = Stream.ofNullable(null); //
Empty
Stream<String> valueStream = Stream.ofNullable("value"); //
Stream with "value"

// Concatenate two existing streams.
Stream<String> stream1 = Stream.of("apple", "banana");
Stream<String> stream2 = Stream.of("cherry", "date");
Stream<String> concatenatedStream = Stream.concat(stream1,
stream2);
// Result: Stream containing ["apple", "banana", "cherry",
"date"]
```

- ***Create a stream from an existing array.***

```
String[] array = {"a", "b", "c"};
Stream<String> streamFromArray = Arrays.stream(array);
```

- ***Create a stream from a Collection (like List or Set).***

```
List<String> list = Arrays.asList("x", "y", "z");
Stream<String> streamFromList = list.stream(); // Common way
```

- ***Create a stream using `Stream.builder()`.***

```
Stream<String> builtStream = Stream.<String>builder()
                                 .add("a").add("b").add("c")
                                 .build();
```

- ***Create an infinite sequential ordered stream (often limited).***

```
Stream<Integer> iteratedStream = Stream.iterate(1, n -> n +
1).limit(5);
// Result: Stream containing [1, 2, 3, 4, 5]
```

- ***Create a finite sequential ordered stream using `iterate` with a predicate (Java 9+).***

```
Stream<Integer> boundedIteratedStream = Stream.iterate(1, n ->
n < 10, n -> n + 2);
// Result: Stream containing [1, 3, 5, 7, 9]
         // Creates a stream: 0, 10, 20, 30, 40, 50, 60, 70,
80, 90

Stream<Integer> boundedIteratedStream = Stream.iterate(0, n ->
n < 100, n -> n + 10)
        .skip(5) // Skips first 5 elements: 0, 10, 20, 30, 40,
Remaining : 50, 60, 70, 80, 90
        .limit(3) // Limits to next 3 elements: 50, 60, 70
        .filter(n -> n % 10 == 0); // Keeps only values
divisible by 10 — all pass: 50, 60, 70

    // Print the final elements in the stream
    boundedIteratedStream.forEach(System.out::println);
    /*
     * Output:
     * 50
     * 60
```

```
        * 70
        */
```

- ***Create an infinite sequential unordered stream using a Supplier (often limited).***

```
Stream<Double> generatedStream =
Stream.generate(Math::random).limit(3);
// Result: Stream containing 3 random doubles
```

✨ Intermediate Operations Examples ✨

`filter()`

- ***Keeps only elements matching the predicate (strings starting with "a").***

```
List<String> filtered = Stream.of("apple", "banana", "cherry")
                             .filter(s -> s.startsWith("a"))
                             .collect(Collectors.toList());
// Result: ["apple"]
```

`map()`

- ***Transforms each element (string to its length).***

```
List<Integer> lengths = Stream.of("apple", "banana", "cherry")
                              .map(String::length)
                              .collect(Collectors.toList());
// Result: [5, 6, 6]
```

`mapToInt()`, `mapToLong()`, `mapToDouble()`

- ***Transforms elements to a primitive stream (string to IntStream of lengths).***

```
IntStream intStream = Stream.of("apple", "banana", "cherry")
                           .mapToInt(String::length);
// Result: IntStream containing 5, 6, 6
```

flatMap()

- ***Transforms each element into a stream and flattens the results into one stream (list of lists to a single list).***

```
Stream<List<String>> streamOfLists = Stream.of(
    Arrays.asList("a", "b"),
    Arrays.asList("c", "d")
);
List<String> flatList =
streamOfLists.flatMap(Collection::stream)

.collect(Collectors.toList());
// Result: ["a", "b", "c", "d"]
```

flatMapToInt(), flatMapToLong(), flatMapToDouble()

- ***Flattens elements into a primitive stream (list of integer lists to a single IntStream).***

```
List<List<Integer>> nestedList = Arrays.asList(
    Arrays.asList(1, 2),
    Arrays.asList(3, 4)
);
IntStream flatIntStream = nestedList.stream()
                                   .flatMapToInt(list ->
list.stream().mapToInt(Integer::intValue));
// Result: IntStream containing 1, 2, 3, 4
```

mapMulti() (Java 16+)

- ***Replaces each element with zero or more elements (maps each string to its uppercase and lowercase versions).***

```
List<String> resultMulti = new ArrayList<>();
Stream.of("apple", "banana", "cherry").<String>mapMulti((s,
consumer) -> {
    consumer.accept(s.toUpperCase());
    consumer.accept(s.toLowerCase());
}).forEach(resultMulti::add);
// Result: ["APPLE", "apple", "BANANA", "banana", "CHERRY",
"cherry"]
```

distinct()

- ***Removes duplicate elements based on `equals()`.***

```
List<String> distinct = Stream.of("apple", "banana", "apple")
                              .distinct()
                              .collect(Collectors.toList());
// Result: ["apple", "banana"]
```

sorted()

- ***Sorts elements according to their natural order.***

```
List<String> sorted = Stream.of("banana", "apple", "cherry")
                            .sorted()
                            .collect(Collectors.toList());
// Result: ["apple", "banana", "cherry"]
```

- ***Sorts elements using a custom comparator (reverse alphabetical order).***

```
List<String> namesSort = Arrays.asList("Charlie", "Alice",
"Bob");
List<String> sortedNames = namesSort.stream()
```

```
.sorted(Comparator.reverseOrder())

.collect(Collectors.toList());
// Result: ["Charlie", "Bob", "Alice"]
```

- ***Sorts elements using a custom comparator (by string length).***

```
List<String> sortedByLength = Stream.of("banana", "apple",
"cherry")

.sorted(Comparator.comparingInt(String::length))

.collect(Collectors.toList());
// Result: ["apple", "cherry", "banana"] or ["apple",
"banana", "cherry"] (stability dependent)
```

peek()

- ***Performs an action on each element as it flows through the stream (used mainly for debugging).***

```
List<String> peeked = Stream.of("apple", "banana", "cherry")
                           .peek(s ->
System.out.println("Processing: " + s)) // Debugging action
                           .map(String::toUpperCase)
                           .collect(Collectors.toList());
// Output during processing: Processing: apple, Processing:
banana, Processing: cherry
// Result: ["APPLE", "BANANA", "CHERRY"]
```

limit()

- ***Truncates the stream to be no longer than the specified size.***

```
List<String> limited = Stream.of("a", "b", "c", "d", "e")
                             .limit(3)
                             .collect(Collectors.toList());
// Result: ["a", "b", "c"]
```

`skip()`

- ***Discards the first `n` elements of the stream.***

```
List<String> skipped = Stream.of("a", "b", "c", "d", "e")
                            .skip(2)
                            .collect(Collectors.toList());
// Result: ["c", "d", "e"]
```

`takeWhile()` (Java 9+)

- ***Takes elements from the start while the predicate is true, stops at the first false element.***

```
List<String> takenWhile = Stream.of("a", "b", "c", "d")
                                .takeWhile(s ->
s.compareTo("c") < 0) // Takes "a", "b"
                                .collect(Collectors.toList());
// Result: ["a", "b"]
```

`dropWhile()` (Java 9+)

- ***Drops elements from the start while the predicate is true, keeps the rest starting from the first false element.***

```
List<String> droppedWhile = Stream.of("a", "b", "c", "d")
                                 .dropWhile(s ->
s.compareTo("c") < 0) // Drops "a", "b"
```

```
.collect(Collectors.toList());
// Result: ["c", "d"]
```

✨ Terminal Operations Examples ✨

`forEach()`

- ***Performs an action for each element (prints each element). Order not guaranteed in parallel streams.***

```
System.out.println("forEach example:");
Stream.of("apple", "banana",
"cherry").forEach(System.out::println);
// Output: apple, banana, cherry (order may vary if parallel)
```

`forEachOrdered()`

- ***Performs an action for each element, maintaining encounter order even in parallel streams.***

```
System.out.println("\nforEachOrdered example (parallel):");
Stream.of("apple", "banana",
"cherry").parallel().forEachOrdered(System.out::println);
// Output: apple, banana, cherry (guaranteed order)
```

`toArray()`

- ***Collects stream elements into an `Object` array.***

```
List<String> namesToArray1 = Arrays.asList("Alice", "Bob",
"Charlie");
Object[] nameArrayObj = namesToArray1.stream().toArray();
System.out.println("\ntoArray (Object[]): " +
Arrays.toString(nameArrayObj));
// Output: [Alice, Bob, Charlie]
```

- ***Collects stream elements into a typed array using a generator.***

```
List<String> namesToArray2 = Arrays.asList("Alice", "Bob",
"Charlie");
String[] nameArrayTyped =
namesToArray2.stream().toArray(String[]::new);
System.out.println("toArray (String[]): " +
Arrays.toString(nameArrayTyped));
// Output: [Alice, Bob, Charlie]
```

---

reduce()

- ***Combines stream elements into a single optional result using a binary operator (concatenates strings).***

```
Optional<String> concatenated = Stream.of("a", "b", "c")
                                     .reduce((s1, s2) -> s1 +
s2);
concatenated.ifPresent(s -> System.out.println("\nreduce
(Optional): " + s));
// Output: abc
```

- ***Combines stream elements using an identity value and a binary operator (sums integers).***

```
List<Integer> numbersReduce1 = Arrays.asList(1, 2, 3, 4, 5);
int sumWithIdentity = numbersReduce1.stream()
                                   .reduce(0, Integer::sum);
// Identity = 0
System.out.println("reduce (Identity): " + sumWithIdentity);
// Output: 15

// Step-by-step stream processing to compute sum of specific
elements
// Creates a stream: 0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50,
55, 60, ..., 95
int sumOfFilteredElements = Stream.iterate(0, n -> n < 100, n
-> n + 5)
        .skip(10) // Skips first 10 elements: 0-45, Remaining
stream: 50-95
        .limit(5) // Takes next 5 elements only: 50, 55, 60,
65, 70
        .filter(n -> n % 10 == 0) // Filters only those
divisible by 10: 50, 60, 70
        .reduce(0, Integer::sum); // Reduces (sums up) the
filtered values: 50 + 60 + 70 = 180
System.out.println("Sum of filtered values: " +
sumOfFilteredElements);
/*
 * Output:
 * Sum of filtered values: 180
 */
```

- ***Combines stream elements using identity, accumulator, and combiner (calculates total length of names, parallel-safe).***

```
List<String> namesReduce = Arrays.asList("Alice", "Bob",
"Charlie");
int totalLength = namesReduce.stream()
                             .reduce(0, // Identity
                                     (length, name) -> length
+ name.length(), // Accumulator
                                     Integer::sum); //
Combiner
System.out.println("reduce (parallel capable): " +
totalLength);
// Output: 15
```

collect()

- ***Collects elements into a mutable container using supplier, accumulator, and combiner (creates an ArrayList).***

```
List<String> collectedManual = Stream.of("apple", "banana",
"cherry")
                                    .collect(ArrayList::new,
// Supplier
                                                        ArrayList::add,
// Accumulator

ArrayList::addAll); // Combiner
System.out.println("\ncollect (manual): " + collectedManual);
// Output: [apple, banana, cherry]
```

- ***Collects elements into a List using `Collectors.toList()`.***

```
List<String> namesCollectList = Arrays.asList("Alice", "Bob",
"Charlie");
List<String> nameList = namesCollectList.stream()

.collect(Collectors.toList());
System.out.println("collect (Collectors.toList): " +
nameList);
// Output: [Alice, Bob, Charlie]
```

- ***Collects elements into a Set using `Collectors.toSet()` (removes duplicates).***

```
List<String> namesCollectSet = Arrays.asList("Alice", "Bob",
"Alice");
Set<String> nameSet = namesCollectSet.stream()

.collect(Collectors.toSet());
System.out.println("collect (Collectors.toSet): " + nameSet);
// Output: [Alice, Bob] (order may vary)
```

`toList()` (Java 16+)

- ***Collects elements into an unmodifiable List.***

```
List<String> toListResult = Stream.of("apple", "banana",
"cherry").toList();
System.out.println("\ntoList(): " + toListResult);
// Output: [apple, banana, cherry]
// try { toListResult.add("date"); }
```

```
catch(UnsupportedOperationException e) {
System.out.println("List is unmodifiable"); }
```

---

min()

- ***Finds the minimum element according to a comparator (shortest string).***

```
Optional<String> min = Stream.of("apple", "banana", "cherry")

.min(Comparator.comparingInt(String::length));
min.ifPresent(s -> System.out.println("\nmin (by length): " +
s));
// Output: apple
```

---

max()

- ***Finds the maximum element according to a comparator (longest string).***

```
Optional<String> max = Stream.of("apple", "banana", "cherry")

.max(Comparator.comparingInt(String::length));
max.ifPresent(s -> System.out.println("max (by length): " +
s));
// Output: banana (or cherry, depending on internal stability)
```

---

count()

- ***Counts the number of elements in the stream.***

```
long count = Stream.of("apple", "banana", "cherry").count();
System.out.println("\ncount: " + count);
// Output: 3
```

---

anyMatch()

- ***Checks if at least one element matches the predicate.***

```
boolean hasApple = Stream.of("apple", "banana", "cherry")
                        .anyMatch(s -> s.equals("apple"));
System.out.println("anyMatch ('apple'): " + hasApple);
// Output: true
```

allMatch()

- ***Checks if all elements match the predicate.***

```
boolean allStartWithA = Stream.of("apple", "apricot",
"avocado")
                             .allMatch(s ->
s.startsWith("a"));
System.out.println("allMatch ('a'): " + allStartWithA);
// Output: true

boolean allEndWithE = Stream.of("apple", "banana", "cherry")
                             .allMatch(s -> s.endsWith("e"));
System.out.println("allMatch ('e'): " + allEndWithE);
// Output: false
```

noneMatch()

- ***Checks if no elements match the predicate.***

```
boolean noneStartWithZ = Stream.of("apple", "banana",
"cherry")
                             .noneMatch(s ->
s.startsWith("z"));
System.out.println("noneMatch ('z'): " + noneStartWithZ);
// Output: true
```

`findFirst()`

- ***Finds the first element of the stream (if encounter order exists).***

```
Optional<String> first = Stream.of("apple", "banana",
"cherry").findFirst();
first.ifPresent(s -> System.out.println("\nfindFirst: " + s));
// Output: apple
```

`findAny()`

- ***Finds any element of the stream (useful for parallel streams where performance is key).***

```
Optional<String> any = Stream.of("apple", "banana",
"cherry").parallel().findAny();
any.ifPresent(s -> System.out.println("findAny (parallel): " +
s));
// Output: Could be apple, banana, or cherry
```

🔚 Mapping Objects Example (User to UserDTO)

- ***Transforms a list of `User` objects into a list of `UserDTO` objects using `map`.***

```
// Assume User and UserDTO classes exist as defined
previously...
List<User> users = List.of( /* ... users ... */ );

System.out.println("\n--- DTO Mapping (Stream with map &
collect) ---");
List<UserDTO> usersDTO_stream = users.stream()
    .map(user -> new UserDTO(user.getId(), user.getUserName(),
user.getEmail()))
    .collect(Collectors.toList());

usersDTO_stream.forEach(System.out::println);
```

🔗 Chaining Operations Example

- ***Demonstrates chaining filter, map, and another filter before a terminal operation.***

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

System.out.println("\n--- Chained Operations Example ---");
intList.stream()
       .filter(a -> a % 2 == 0)     // Intermediate: [2, 4, 6]
       .map(a -> a + a)             // Intermediate: [4, 8,
12]
       .filter(a -> a > 5)          // Intermediate: [8, 12]
       .forEach(System.out::println); // Terminal: Prints 8,
then 12
```

⏳ Laziness Example

- ***Illustrates that intermediate operations (`filter`, `map`) only execute when a terminal operation (`forEach`) is called.***

```
// Assume EmployeeLazy class exists as defined previously...
List<EmployeeLazy> empListLazy = Arrays.asList( /* ...
employees ... */ );

System.out.println("\n--- Laziness Example ---");
Stream<String> nameStream = empListLazy.stream()
    .filter(e -> { System.out.println("Filtering employee " +
e.getId()); return e.getId() % 2 == 0; })
    .map(e -> { System.out.println("Mapping employee " +
e.getName()); e.printName(); return e.getName(); });

System.out.println("Pipeline defined. Adding terminal
operation...");
nameStream.forEach(name -> System.out.println("Final result: "
+ name)); // Execution happens now
```