Here's a **detailed explanation** of the main methods used in the Java **Stream API**, categorized into **intermediate operations** and **terminal operations**, along with comprehensive descriptions and examples.

---

1. Intermediate Operations

a. filter(Predicate)

- **Purpose**: Select elements based on a condition.
- **How it works**: Applies a **boolean condition** (via Predicate) to each element and includes only those that pass the condition.
- **Use case**: Filter out unwanted data.

Example:

java

Copy code

```java
List<String> names = List.of("Alice", "Bob", "Charlie");
List<String> filteredNames = names.stream()
   .filter(name -> name.startsWith("A"))
   .toList(); // Collects the stream elements into a List
System.out.println(filteredNames); // Output: [Alice]
```

---

b. map(Function)

- **Purpose**: Transform each element of the stream into another form.
- **How it works**: Applies a **Function** to each element, transforming it.

- **Use case**: Extract specific fields or convert data types.

Example:

java

Copy code

```
List<String> names = List.of("Alice", "Bob", "Charlie");
List<Integer> nameLengths = names.stream()
   .map(String::length) // Transform each name into its length
   .toList();
System.out.println(nameLengths); // Output: [5, 3, 7]
```

---

c. sorted()

- **Purpose**: Sort the elements of the stream.
- **Variants**:
  - sorted() for natural order.
  - sorted(Comparator) for custom order.
- **Use case**: Organize data.

Example 1: Natural Order:

java

Copy code

```
List<Integer> numbers = List.of(5, 2, 8, 1);
List<Integer> sortedNumbers = numbers.stream()
   .sorted()
   .toList();
System.out.println(sortedNumbers); // Output: [1, 2, 5, 8]
```

Example 2: Custom Order:

java

Copy code

```
List<String> names = List.of("Charlie", "Alice", "Bob");
List<String> sortedNames = names.stream()
   .sorted((a, b) -> b.compareTo(a)) // Descending order
   .toList();
System.out.println(sortedNames); // Output: [Charlie, Bob, Alice]
```

---

d. distinct()

- **Purpose**: Removes duplicate elements from the stream.
- **Use case**: Deduplicate data.

Example:

java

Copy code

```
List<Integer> numbers = List.of(1, 2, 2, 3, 4, 4, 5);
List<Integer> uniqueNumbers = numbers.stream()
   .distinct()
   .toList();
System.out.println(uniqueNumbers); // Output: [1, 2, 3, 4, 5]
```

---

e. limit(long)

- **Purpose**: Limits the number of elements in the stream to the specified count.

- **Use case**: Get only a subset of data.

Example:

java

Copy code

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5);
List<Integer> limitedNumbers = numbers.stream()
   .limit(3)
   .toList();
System.out.println(limitedNumbers); // Output: [1, 2, 3]
```

---

f. skip(long)

- **Purpose**: Skips the first n elements of the stream.
- **Use case**: Ignore unwanted initial elements.

Example:

java

Copy code

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5);
List<Integer> skippedNumbers = numbers.stream()
   .skip(2)
   .toList();
System.out.println(skippedNumbers); // Output: [3, 4, 5]
```

---

g. flatMap(Function)

- **Purpose**: Flatten nested data structures into a single stream.
- **Use case**: Process lists of lists.

Example:

java

Copy code

```
List<List<Integer>> nestedList = List.of(
   List.of(1, 2),
   List.of(3, 4),
   List.of(5)
);
List<Integer> flattenedList = nestedList.stream()
   .flatMap(List::stream) // Flatten inner lists
   .toList();
System.out.println(flattenedList); // Output: [1, 2, 3, 4, 5]
```

---

2. Terminal Operations

a. forEach(Consumer)

- **Purpose**: Applies an action to each element in the stream.
- **Use case**: Printing or performing side-effects.

Example:

java

Copy code

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

```
names.stream()
    .forEach(System.out::println);
// Output:
// Alice
// Bob
// Charlie
```

---

b. collect(Collector)

- **Purpose**: Gathers the stream elements into a collection or data structure.
- **Common Collectors**:
    - Collectors.toList() for lists.
    - Collectors.toSet() for sets.
    - Collectors.groupingBy() for maps.
    - Collectors.joining() for strings.

Example:

java

Copy code

```
List<String> names = List.of("Alice", "Bob", "Charlie");
Set<String> nameSet = names.stream()
    .collect(Collectors.toSet());
System.out.println(nameSet); // Output: [Alice, Bob, Charlie]
```

---

c. reduce(BinaryOperator)

- **Purpose**: Combines elements into a single result.
- **Use case**: Aggregate data, such as sum or product.

Example:

java

Copy code

```
List<Integer> numbers = List.of(1, 2, 3, 4);
int sum = numbers.stream()
   .reduce(0, Integer::sum); // Start with 0, add each number
System.out.println(sum); // Output: 10
```

---

d. count()

- **Purpose**: Counts the number of elements in the stream.
- **Use case**: Get the size of filtered or processed data.

Example:

java

Copy code

```
List<String> names = List.of("Alice", "Bob", "Charlie");
long count = names.stream()
   .filter(name -> name.startsWith("A"))
   .count();
System.out.println(count); // Output: 1
```

---

e. findFirst() and findAny()

- **Purpose**: Retrieve a single element from the stream.

- **Use case**: Quickly get a value (often for validation).

Example:

java

Copy code

```
List<String> names = List.of("Alice", "Bob", "Charlie");
Optional<String> first = names.stream()
   .findFirst();
first.ifPresent(System.out::println); // Output: Alice
```

---

f. anyMatch(), allMatch(), noneMatch()

- **Purpose**: Check if elements match a condition.
- **Use case**: Validation or condition checks.

Examples:

- **Any element matches**:

java

Copy code

```
boolean hasAlice = names.stream()
   .anyMatch(name -> name.equals("Alice"));
System.out.println(hasAlice); // Output: true
```

- **All elements match**:

java

Copy code

```
boolean allShort = names.stream()
   .allMatch(name -> name.length() <= 7);
```

```
System.out.println(allShort); // Output: true
```