**Copilot**

Java's Stream API offers a wide range of operations that you can perform on streams. These operations are generally categorized into **intermediate** and **terminal** operations. Here are some common ones:

## Intermediate Operations

1. **`map`**: Transforms each element of the stream.

```
List<String> words = Arrays.asList("hello", "world");
List<String> upperCaseWords = words.stream()
    .map(String::toUpperCase)
    .collect(Collectors.toList());
```

2. **`filter`**: Filters elements based on a predicate.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> evenNumbers = numbers.stream()
    .filter(n -> n % 2 == 0)
    .collect(Collectors.toList());
```

3. **`sorted`**: Sorts the elements of the stream.

```
List<String> words = Arrays.asList("banana", "apple", "cherry");
List<String> sortedWords = words.stream()
    .sorted()
    .collect(Collectors.toList());
```

4. **`flatMap`**: Flattens a stream of collections into a single stream.

```
List<List<String>> listOfLists = Arrays.asList(
    Arrays.asList("a", "b"),
    Arrays.asList("c", "d")
);
List<String> flatList = listOfLists.stream()
    .flatMap(List::stream)
    .collect(Collectors.toList());
```

5. **`distinct`**: Removes duplicate elements.

```
List<Integer> numbers = Arrays.asList(1, 2, 2, 3, 3, 3);
List<Integer> distinctNumbers = numbers.stream()
    .distinct()
    .collect(Collectors.toList());
```

6. **`peek`**: Performs an action on each element without modifying the stream.

```
List<String> words = Arrays.asList("one", "two", "three");
List<String> result = words.stream()
    .peek(System.out::println)
    .collect(Collectors.toList());
```

## Terminal Operations

1. **`collect`**: Collects the elements of the stream into a collection.

```
List<String> words = Arrays.asList("hello", "world");
List<String> result = words.stream()
    .collect(Collectors.toList());
```

2. **`forEach`**: Performs an action for each element of the stream.

```
List<String> words = Arrays.asList("hello", "world");
words.stream()
    .forEach(System.out::println);
```

3. **`reduce`**: Reduces the elements of the stream to a single value.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
int sum = numbers.stream()
    .reduce(0, Integer::sum);
```

4. **`count`**: Returns the count of elements in the stream.

```
List<String> words = Arrays.asList("hello", "world");
long count = words.stream()
    .count();
```

5. **`anyMatch`**: Checks if any element matches a predicate.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
boolean hasEven = numbers.stream()
    .anyMatch(n -> n % 2 == 0);
```

6. **`findFirst`**: Returns the first element of the stream.

```
List<String> words = Arrays.asList("hello", "world");
Optional<String> firstWord = words.stream()
    .findFirst();
```

These operations allow you to perform complex data processing tasks in a concise and readable manner. If you have any specific use cases or need more examples, feel free to ask!