Copilot

The Java Stream API, introduced in Java 8, provides a powerful way to process sequences of elements. Here's a brief overview of how to use some of the most common operations:

1. Filter

Filters elements based on a condition.

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
List<String> filteredNames = names.stream()
                                  .filter(name -> name.startsWith("A"))
                                  .collect(Collectors.toList());
```

2. Map

Transforms each element.

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
List<Integer> nameLengths = names.stream()
                                 .map(String::length)
                                 .collect(Collectors.toList());
```

3. FlatMap

Flattens nested structures.

```
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());
```

4. Sorted

Sorts elements.

```
List<String> names = Arrays.asList("Charlie", "Alice", "Bob");
List<String> sortedNames = names.stream()
                                .sorted()
                                .collect(Collectors.toList());
```

5. Distinct

Removes duplicates.

```
List<Integer> numbers = Arrays.asList(1, 2, 2, 3, 4, 4, 5);
List<Integer> distinctNumbers = numbers.stream()
                                       .distinct()
                                       .collect(Collectors.toList());
```

6. Limit

Limits the number of elements.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> limitedNumbers = numbers.stream()
                                      .limit(3)
                                      .collect(Collectors.toList());
```

7. Skip

Skips the first N elements.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> skippedNumbers = numbers.stream()
                                      .skip(2)
                                      .collect(Collectors.toList());
```

8. ForEach

Performs an action for each element.

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.stream().forEach(System.out::println);
```

9. Collect

Collects the elements into a collection.

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
List<String> collectedNames = names.stream()
                                   .collect(Collectors.toList());
```

10. Find

Finds an element based on a condition.

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
Optional<String> name = names.stream()
                             .filter(n -> n.startsWith("B"))
                             .findFirst();
```

These operations can be combined to create powerful data processing pipelines. For more detailed examples and explanations, you can check out resources like Baeldung's Java 8 Stream API Tutorial and HowToDoInJava's Stream Examples.

Do you have a specific use case in mind for using the Stream API?