Java Stream API Guide for Absolute Beginners

What is a Stream?

A stream is a sequence of elements that supports various methods to perform computations. It allows for functional-style operations on collections of objects, such as filtering, mapping, and reducing.

Creating Streams

You can create streams from collections, arrays, or custom data sources.

From Collections

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

From Arrays

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

From Values

```java
Stream<String> nameStream = Stream.of("Alice", "Bob", "Charlie");
```

Intermediate Operations

Intermediate operations transform a stream into another stream. They are lazy, meaning they are not executed until a terminal operation is invoked.

filter

Filters elements based on a condition.

```
Stream<String> filteredStream = nameStream.filter(name -> name.
    startsWith("A"));
```

map

Transforms each element in the stream.

```
Stream<Integer> lengthStream = nameStream.map(String::length);
```

sorted

Sorts the elements in the stream.

```
Stream<String> sortedStream = nameStream.sorted();
```

distinct

Removes duplicate elements.

```
Stream<String> uniqueStream = nameStream.distinct();
```

Terminal Operations

Terminal operations produce a result or a side effect and mark the end of the stream.

collect

Collects the elements of the stream into a collection.

```
List<String> nameList = nameStream.collect(Collectors.toList());
```

forEach

Performs an action for each element.

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

reduce

Combines elements of the stream into a single result.

```java
Optional<String> concatenatedNames = nameStream.reduce((a, b) -> a + ", " + b);
```

count

Counts the number of elements in the stream.

```java
long count = nameStream.count();
```

Putting It All Together

Let's create a complete example where we perform several operations on a stream.

Example: Filtering, Mapping, Sorting, and Collecting

```java
import java.util.*;
import java.util.stream.*;

public class StreamExample {
    public static void main(String[] args) {
        // Create a list of names
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David", "Edward");

        // Create a stream from the list
        List<String> result = names.stream()
            // Filter names that start with "A" or "D"
            .filter(name -> name.startsWith("A") || name.startsWith("D"))
            // Convert names to uppercase
            .map(String::toUpperCase)
            // Sort the names
            .sorted()
            // Collect the result into a list
            .collect(Collectors.toList());

        // Print the result
        System.out.println(result); // Output: [ALICE, DAVID]
    }
}
```

Summary

The Java Stream API provides a powerful way to perform operations on collections in a functional programming style. By using streams, you can write more readable and concise code. Start by creating a stream, apply intermediate operations to transform the stream, and finish with a terminal operation to produce a result or a side effect.

Happy coding!