The Java 8 Stream API

The Stream API, introduced in Java 8, is a powerful abstraction for processing sequences of elements. It allows for functional-style operations on collections of data, such as filtering, mapping, and reducing. The Stream API enables parallel and sequential operations, making it easier to write concise and expressive code.

The Stream API is a significant enhancement to the Java language, enabling developers to write more expressive and efficient code for data processing tasks.

Here's an overview of its key concepts and components:

Key Concepts

- **Stream**:
  - A sequence of elements supporting sequential and parallel aggregate operations.
  - Streams are not data structures; they do not store elements but rather convey elements from a source (e.g., collections, arrays, I/O channels).
- **Source**:
  - The source of a stream can be a collection, an array, a generator function, or an I/O channel.
  - Example: Collection.stream(), Arrays.stream(array).
- **Intermediate Operations**:
  - Operations that transform a stream into another stream.
  - They are lazy, meaning they are not executed until a terminal operation is invoked.
  - Examples: filter(), map(), flatMap(), sorted(), distinct().
- **Terminal Operations**:
  - Operations that produce a result or a side-effect.
  - They trigger the processing of the stream.
  - Examples: collect(), forEach(), reduce(), count(), findFirst(), allMatch().
- **Pipelines**:
  - A stream pipeline consists of a source, zero or more intermediate operations, and a terminal operation.
  - Example: source.stream().filter(...).map(...).collect(...).

Examples of Stream API Usage

- **Creating a Stream**:

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

- **Filtering a Stream**:
    - The filter method is used to select elements from the stream that match a given predicate.
    - This method takes a Predicate as an argument, which is a boolean-valued function.
    - Only elements that return true for the predicate are included in the resulting stream.
    - filter selects elements based on a condition.

```
Example:
     List<String> filteredNames = nameStream
          .filter(name -> name.startsWith("A"))
          .collect(Collectors.toList());

Output:
     filteredNames will be [Alice]
```

- **Mapping Elements**:
    - The map method is used to apply a function to each element of the stream, producing a new stream of the transformed elements.
    - This method takes a Function as an argument, which is applied to each element of the stream.
    - The function must return a single value for each element.
    - map transforms each element into another object.

```
Example:
     List<Integer> nameLengths = nameStream
          .map(String::length)
          .collect(Collectors.toList());
```

```
Output:
     nameLengths will be [5, 3, 7]
```

- **Flat Mapping**:
    - The flatMap method is used to flatten a stream of streams into a single stream.
    - This method takes a Function that returns a stream for each element, and then concatenates the resulting streams into one.
    - It is particularly useful when dealing with nested collections.
    - flatMap flattens a stream of streams into a single stream.

```
Example:
     List<List<String>> nestedNames = Arrays.asList(
          Arrays.asList("Alice", "Bob"),
          Arrays.asList("Charlie", "Dave"));
     List<String> flatNames = nestedNames.stream()
          .flatMap(Collection::stream)
          .collect(Collectors.toList());

Output:
     flatNames will be [Alice, Bob, Charlie, Dave]
```

- **Sorting:**
    - The sorted() method is used to sort the elements of the stream according to natural order or a provided comparator.

```
Example: 1.)
     List<String> names = Arrays.asList("Charlie", "Alice",
"Bob");
     List<String> sortedNames = names.stream()
          .sorted().collect(Collectors.toList());
     System.out.println(sortedNames);

Output:
     [Alice, Bob, Charlie]
```

```
Example: 2.) With a custom comparator
     List<String> names = Arrays
          .asList("Charlie", "Alice", "Bob");
     List<String> sortedNames = names.stream()
          .sorted(Comparator.reverseOrder())
          .collect(Collectors.toList());
     System.out.println(sortedNames);

Output:
     [Charlie, Bob, Alice]
```

- **Removing Duplicates:**
  - The distinct() method is used to eliminate duplicate elements from the stream.

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

Output:
     [1, 2, 3, 4, 5]
```

Examples of Terminal Operations:

- **collect():**
  - The collect() method is used to gather the elements of the stream into a collection or another data structure.

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

```
Output:
     [Alice]
```

- **forEach()**
  - The forEach() method is used to perform an action for each element of the stream.

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

Output:
     Alice Bob Charlie
```

- **reduce()**
  - The reduce() method is used to combine the elements of the stream into a single result.

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

Output:
     15
```

- **count()**
  - The count() method is used to count the number of elements in the stream.

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

Output:
     1
```

- **findFirst()**
  - The findFirst() method is used to find the first element of the stream that matches the given criteria.

```
Example:
     List<String> names = Arrays
          .asList("Alice", "Bob", "Charlie");
     Optional<String> first = names.stream()
          .filter(name -> name.startsWith("C"))
          .findFirst();
     first.ifPresent(System.out::println);

Output:
     Charlie
```

- **allMatch()**
  - The allMatch() method is used to check if all elements of the stream match the given predicate.

```
Example:
     List<Integer> numbers = Arrays
          .asList(2, 4, 6, 8);
     boolean allEven = numbers.stream()
          .allMatch(num -> num % 2 == 0);
     System.out.println(allEven);

Output:
     true
```

Benefits of the Stream API

- **Conciseness and Readability**: Stream operations are typically more concise and readable compared to traditional for-loops.
- **Parallel Processing**: Streams can be processed in parallel to leverage multi-core architectures with parallelStream().
- **Functional Programming**: The API embraces functional programming principles, allowing for more declarative code.
- **Lazy Evaluation**: Intermediate operations are lazy, improving performance by avoiding unnecessary computations.