Java 8 - Streams Cont.

Status	Done
Created By	Ⓜ Muhammad Yousuf
Module	Collections
Reviewd	✓

Java Stream API: Detailed Explanation, Syntax, and Examples

Java Stream API, introduced in Java 8, provides a powerful way to process sequences of elements (like collections) in a functional style. It allows for efficient, readable, and declarative manipulation of data. Below is a detailed breakdown of the key operators, their syntax, examples, and some additional tricks and hacks.

---

1. `stream()` – Convert a Collection into a Stream

The `stream()` method is the starting point for working with streams. It converts a collection (like a `List`, `Set`, or `Map`) into a stream, enabling you to perform various operations on the data.

Syntax:

```
Stream<T> stream = collection.stream();
```

Example:

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

Use Case:

- When you need to process a collection of elements in a pipeline (e.g., filtering, mapping, reducing).

---

2. `filter()` – Keep Only What You Need

The `filter()` method is used to select elements that match a given condition (predicate).

Syntax:

```
Stream<T> filteredStream = stream.filter(Predicate<T> condition);
```

Example:

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);
List<Integer> evenNumbers = numbers.stream()
                                   .filter(n → n % 2 == 0)
                                   .collect(Collectors.toList());
System.out.println(evenNumbers); // Output: [2, 4, 6]
```

Use Case:

- When you want to extract elements that satisfy a specific condition (e.g., even numbers, names starting with "A").

---

3. `map()` – Transform Each Element

The `map()` method is used to transform each element in the stream using a given function.

Syntax:

```
Stream<R> mappedStream = stream.map(Function<T, R> mapper);
```

Example:

```
List<String> names = List.of("alice", "bob", "charlie");
List<String> uppercased = names.stream()
                               .map(String::toUpperCase)
                               .collect(Collectors.toList());
System.out.println(uppercased); // Output: [ALICE, BOB, CHARLIE]
```

Use Case:

- When you need to transform elements (e.g., converting strings to uppercase, extracting specific fields from objects).

---

4. `flatMap()` – Flatten Nested Structures

The `flatMap()` method is used to flatten nested structures (e.g., a list of lists) into a single stream.

Syntax:

```
Stream<R> flatMappedStream = stream.flatMap(Function<T, Stream<R>> ma
pper);
```

Example:

```
List<List<String>> listOfLists = List.of(
    List.of("A", "B"),
    List.of("C", "D")
);
List<String> flatList = listOfLists.stream()
                                   .flatMap(List::stream)
                                   .collect(Collectors.toList());
System.out.println(flatList); // Output: [A, B, C, D]
```

Use Case:

- When you have nested collections and want to process all elements as a single stream.

---

5. `forEach()` – Iterate Over Elements

The `forEach()` method is used to perform an action on each element of the stream.

Syntax:

```
stream.forEach(Consumer<T> action);
```

Example:

```
List.of("Java", "Streams", "API").forEach(System.out::println);
```

Use Case:

- When you want to perform an action (e.g., printing) on each element of the stream.

Caution:

- Avoid using `forEach()` to modify data within the stream. Use `map()` for transformations.

---

6. `sorted()` – Sort Elements

The `sorted()` method is used to sort elements in the stream, either by their natural order or using a custom comparator.

Syntax:

```
Stream<T> sortedStream = stream.sorted();
Stream<T> sortedStream = stream.sorted(Comparator<T> comparator);
```

Example:

```
List<Integer> numbers = List.of(5, 3, 8, 1);
List<Integer> sortedNumbers = numbers.stream()
                                     .sorted()
```

```
                                    .collect(Collectors.toList());
System.out.println(sortedNumbers); // Output: [1, 3, 5, 8]
```

Use Case:

- When you need to sort elements in ascending or descending order.

---

7. `reduce()` – Combine Elements into One Value

The `reduce()` method is used to aggregate elements into a single value, such as summing numbers or concatenating strings.

Syntax:

```
T result = stream.reduce(T identity, BinaryOperator<T> accumulator);
Optional<T> result = stream.reduce(BinaryOperator<T> accumulator);
```

Example:

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

Use Case:

- When you need to combine elements into a single result (e.g., sum, max, concatenation).

---

8. `distinct()` – Remove Duplicates

The `distinct()` method is used to eliminate duplicate elements from the stream.

Syntax:

```
Stream<T> distinctStream = stream.distinct();
```

Example:

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

Use Case:

- When you want to remove duplicate elements from a collection.

---

9. `limit()` & `skip()` – Control Elements Processed

The `limit()` method restricts the number of elements processed, while `skip()` skips a specified number of elements.

Syntax:

```
Stream<T> limitedStream = stream.limit(long maxSize);
Stream<T> skippedStream = stream.skip(long n);
```

Example:

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

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

Use Case:

- When you want to process only a subset of elements (e.g., pagination).

---

10. Matching Operators: `anyMatch()` , `allMatch()` , `noneMatch()`

These operators are used to check if elements in the stream match a given condition.

Syntax:

```
boolean anyMatch = stream.anyMatch(Predicate<T> condition);
boolean allMatch = stream.allMatch(Predicate<T> condition);
boolean noneMatch = stream.noneMatch(Predicate<T> condition);
```

Example:

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

boolean anyStartsWithA = names.stream()
                                         .anyMatch(name → name.startsWith("A"));
System.out.println(anyStartsWithA); // Output: true

boolean allStartWithA = names.stream()
                                      .allMatch(name → name.startsWith("A"));
```

```
System.out.println(allStartWithA); // Output: false

boolean noneStartsWithZ = names.stream().noneMatch(name → name.starts
With("Z"));
System.out.println(noneStartsWithZ); // Output: true
```

Use Case:

- When you need to check if elements in the stream satisfy a condition.

Additional Tricks and Guides

1. **Method References**: Use method references for cleaner code.

   ```
   List<String> names = List.of("alice", "bob", "charlie");
   List<String> uppercased = names.stream()
                                  .map(String::toUpperCase)
                                  .collect(Collectors.toList());
   ```

2. **Parallel Streams**: Use `parallelStream()` for large datasets to improve performance, but be cautious with shared state.

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

3. **Collectors**: Use `Collectors` for advanced operations like grouping, partitioning, and joining.

   ```
   Map<Character, List<String>> grouped = names.stream()
                   .collect(Collectors.groupingBy(name → name.charAt(0)));
   ```

4. **Chaining Operations**: Stream operations can be chained for complex transformations.

   ```
   List<String> result = names.stream()
                              .filter(name → name.length() > 3) // filter
                              .map(String::toUpperCase) // transform
                              .sorted()      // sort
                              .collect(Collectors.toList());
   ```

5. **Custom Comparators**: Use custom comparators for sorting.

   ```
   List<String> sortedNames = names.stream()
                                   .sorted((a, b) → b.compareTo(a))
                                   .collect(Collectors.toList());
   ```

Conclusion

The Java Stream API is a powerful tool for processing collections in a functional and declarative manner. By mastering these operators and understanding their use cases, you can write more efficient and readable code. Additionally, leveraging advanced features like parallel streams, method references, and custom collectors can further enhance your stream processing capabilities.