Chapter 5: Java Streams and Lambda Expressions

Introduction

As Java has evolved, one of the most revolutionary enhancements came with **Java 8**, introducing **Streams** and **Lambda Expressions**. These features empower developers to write cleaner, more expressive, and functional-style code. With the rise in data processing needs, Java Streams offer a declarative way to process collections of data, while Lambda expressions allow concise implementation of functional interfaces.

This chapter dives deep into understanding how Java Streams and Lambdas work, how they improve code readability and performance, and how you can effectively use them in real-world Java applications.

5.1 What is a Java Stream?

A **Stream** in Java is a sequence of elements supporting **sequential** and **parallel** aggregate operations. Unlike collections, streams do not store data. Instead, they operate on the underlying data source such as **Collections**, **Arrays**, or **I/O channels**.

Key Features of Streams:

- Not a data structure
- Does not modify the source
- Lazy execution
- Can be infinite
- Supports pipelining (method chaining)

5.2 Types of Streams

1. **Sequential Stream**: Processes data one element at a time in a single thread.
2. **Parallel Stream**: Splits data processing across multiple threads for faster execution.

5.3 Creating Streams

From a Collection:

```
javaCopy codeList<String> names = Arrays.asList("Alice", "Bob", "Charlie");
Stream<String> stream = names.stream();
```

From an Array:

```
javaCopy codeint[] numbers = {1, 2, 3, 4};
IntStream stream = Arrays.stream(numbers);
```

Using `Stream.of()`:

```
javaCopy codeStream<String> stream = Stream.of("Java", "Python", "C++");
```

5.4 Stream Operations

Stream operations are of two types:

*1. **Intermediate Operations** (return a stream):*

- `filter()`: Filters elements based on a condition.
- `map()`: Transforms elements.
- `sorted()`: Sorts elements.
- `distinct()`: Removes duplicates.
- `limit(n)`: Limits output to n elements.

*2. **Terminal Operations** (produce a result or side-effect):*

- `forEach()`: Performs an action for each element.
- `collect()`: Converts the stream into a collection or result.
- `reduce()`: Reduces stream to a single value.
- `count()`: Counts elements.
- `anyMatch()`, `allMatch()`, `noneMatch()`: Matching operations.

Example:

```
javaCopy codeList<String> names = Arrays.asList("Alice", "Bob", "Charlie", "D
avid");
names.stream()
     .filter(n -> n.startsWith("C"))
     .map(String::toUpperCase)
     .forEach(System.out::println);
```

5.5 Lambda Expressions in Java

A **Lambda Expression** is a short block of code which takes in parameters and returns a value. Lambda expressions are used primarily to define inline implementation of a **functional interface**.

Syntax:

```
javaCopy code(parameter1, parameter2) -> { expression or block of code }
```

Example:

```
javaCopy code(int a, int b) -> a + b
```

Or with a functional interface:

```
javaCopy codeRunnable r = () -> System.out.println("Hello from Lambda!");
```

5.6 Functional Interfaces

A **Functional Interface** is an interface that contains exactly **one abstract method**.

Common Functional Interfaces in `java.util.function`:

- `Predicate<T>`: returns boolean
- `Function<T, R>`: takes T, returns R
- `Consumer<T>`: performs action on T
- `Supplier<T>`: returns T

Example:

```
javaCopy codePredicate<String> startsWithA = s -> s.startsWith("A");
System.out.println(startsWithA.test("Apple")); // true
```

5.7 Method References

Method references are a shorthand notation of a lambda expression to call a method.

Syntax:

```
javaCopy codeClassName::methodName
```

Example:

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

5.8 Stream Collectors

`Collectors` is a utility class used to accumulate elements into collections or summarization.

Examples:

- `toList()`
- `toSet()`
- `joining()`
- `counting()`
- `groupingBy()`

Example:

```
javaCopy codeList<String> names = Arrays.asList("Java", "Python", "Java");
Set<String> set = names.stream().collect(Collectors.toSet());
```

5.9 Stream Reduction

Reduction is the process of combining elements into a single result.

Example using `reduce()`:

```
javaCopy codeList<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
int sum = numbers.stream().reduce(0, (a, b) -> a + b);
```

5.10 Parallel Streams

Parallel streams split the stream into multiple parts and process them concurrently.

Example:

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

Use Parallel Streams with caution: Thread-safety and overhead need to be considered.

5.11 Best Practices

- Use **Streams** when you need complex data processing.
- Avoid using stateful operations (e.g., modifying external variables).
- Prefer **method references** when possible for better readability.
- Do not mix **Stream API** with loops or other external iterations.
- Use **parallel streams** only when performance improvement is measurable.

✓ Summary

In this chapter, we explored two major features of Java 8: **Streams** and **Lambda Expressions**. Streams allow for powerful and efficient data processing pipelines with a declarative approach, while Lambda expressions enable functional programming in Java, making code more concise and expressive. Mastery of these features is essential for writing modern, performant, and readable Java applications.
