Lambdas and Stream API in Java

Object-Oriented Programming Class

January 6, 2025

Introduction to Lambdas

Definition: A Lambda Expression is a concise way to represent an anonymous function.

- Introduced in Java 8.
- Enables functional programming in Java.
- Simplifies the use of single-method interfaces (functional interfaces).

Syntax:

- `(parameters) -> expression`
- `(parameters) -> { statements }`

Simple Lambda Expression Example

Example in Java:

```java
@FunctionalInterface
interface Greeting {
    void sayHello(String name);
}

public class Main {
    public static void main(String[] args) {
        Greeting greeting = (name) -> System.out.
            println("Hello, " + name);
        greeting.sayHello("John");
    }
}
// Output: Hello, John
```

Functional Interfaces

Definition: An interface with exactly one abstract method.

- Used as the target for lambda expressions.
- Annotated with `@FunctionalInterface` (optional).

Common Functional Interfaces:

- `Predicate<T>` - Evaluates a condition on an input.
- `Consumer<T>` - Performs an action on an input.
- `Function<T, R>` - Maps an input to an output.
- `Supplier<T>` - Provides an instance of a type.
- `BinaryOperator<T>` - Takes two inputs and produces a result of the same type.
- `UnaryOperator<T>` - A specialization of `Function<T, R>` for the case where input and output are the same type.

Custom Functional Interface Example

Example in Java:

```java
@FunctionalInterface
interface Calculator {
    int operate(int a, int b);
}

public class CustomFunctionalInterface {
    public static void main(String[] args) {
        // Lambda for addition
        Calculator addition = (a, b) -> a + b;
        System.out.println("Sum: " + addition.operate
            (5, 3));

        // Lambda for multiplication
        Calculator multiplication = (a, b) -> a * b;
        System.out.println("Product: " +
            multiplication.operate(5, 3));
    }
}
// Output:
// Sum: 8
// Product: 15
```

Stream API Introduction

Definition: A Stream is a sequence of elements supporting sequential and parallel operations.

- Introduced in Java 8.
- Focuses on processing collections of data.
- Supports functional programming through lambda expressions.

Characteristics:

- Not a data structure.
- Does not store elements.
- Provides a pipeline for computation.

Creating Streams

Examples of Creating Streams:

```java
import java.util.stream.Stream;

public class StreamCreation {
    public static void main(String[] args) {
        // From a collection
        List<String> list = Arrays.asList("A", "B", "C");
        Stream<String> streamFromList = list.stream();

        // From an array
        String[] array = {"X", "Y", "Z"};
        Stream<String> streamFromArray = Arrays.stream(array);

        // Using Stream.of
        Stream<String> streamOf = Stream.of("1", "2", "3");
    }
}
```

Stream Operations

Types of Operations:

- **Intermediate Operations:** Transform a stream into another stream.
  - `map`, `filter`, `sorted`
- **Terminal Operations:** Produce a result or a side-effect.
  - `forEach`, `collect`, `reduce`

Intermediate Operations Example

Example in Java:

```java
import java.util.Arrays;
import java.util.List;

public class IntermediateExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3,
             4, 5);
        numbers.stream()
               .filter(n -> n % 2 == 0) // Keep even
                   numbers
               .map(n -> n * n)          // Square each
                    number
               .forEach(System.out::println);
        // Output: 4, 16
    }
}
```

Terminal Operations Example

Example in Java:

```java
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class TerminalExample {
    public static void main(String[] args) {
        List<String> items = Arrays.asList("apple", "banana", "cherry");
        List<String> result = items.stream()
                                   .filter(item -> item.startsWith("b"))
                                   .collect(Collectors.toList());
        System.out.println(result); // Output: [banana]
    }
}
```

Parallel Streams

Definition: Parallel streams enable parallel processing of data.
Example in Java:

```java
import java.util.stream.IntStream;

public class ParallelExample {
    public static void main(String[] args) {
        IntStream.range(1, 10)
                     .parallel()
                     .forEach(n -> System.out.println(n +
                         " " + Thread.currentThread().
                         getName()));
    }
}
// Output may vary due to parallel processing.
```