Lambdas and Streams

Key Terms

Lambdas

> A lambda expression is a block of code that gets passed around, like an anonymous method. It is a way to pass behavior as an argument to a method invocation and to define a method without a name.

Streams

> A stream is a sequence of data. It is a way to write code that is more declarative and less imperative to process collections of objects.

Functional Interfaces

> A functional interface is an interface that contains one and only one abstract method. It is a way to define a contract for behavior as an argument to a method invocation.

Lambdas

Motivation

To understand, the motivation behind lambdas, remember how we create a thread in Java. We create a class that implements the `Runnable` interface and override the `run()` method. Then we create a new instance of the class and pass it to the `Thread` constructor.

```
class Printer implements Runnable {
    public void run() {
        System.out.println("Hello World: " +
Thread.currentThread().getName());
    }
}

public class Application {
    public static void main(String[] args) {
        Thread thread = new Thread(new Printer());
        thread.start();
    }
}
```

This is a lot of code to write just to print a simple message. Here, the `Runnable` interface is a functional interface. It contains only one abstract method, `run()`. An interface with a single abstract method (SAM) is called a functional interface. Such interfaces can be implemented using lambdas.

Runnables with Lambdas

We can rewrite the above code using lambdas as follows:

```
public class Application {
    public static void main(String[] args) {
        Thread thread = new Thread(() -> System.out.println("Hello World:
" + Thread.currentThread().getName()));
        thread.start();
    }
}
```

Here, we have passed a lambda expression to the `Thread` constructor. The lambda expression is a block of code that gets passed around, like an anonymous method. It is a way to pass behavior as an argument to a method invocation and to define a method without a name. The syntax of a lambda expression is as follows:

1. **Parameter List:** This represents the parameters passed to the lambda expression. It can be empty or contain one or more parameters enclosed in parentheses. If there's only one parameter and its type is inferred, you can omit the parentheses.

2. **Arrow Operator (`->`):** This separates the parameter list from the body of the lambda expression.

3. **Lambda Body:** This contains the code that makes up the implementation of the abstract method of the functional interface. The body can be a single expression or a block of code enclosed in curly braces.

Here's the general syntax of a lambda expression:

```
(parameter list) -> { lambda body }
```

For example, let's break down the lambda expression in the above code:

```
() -> System.out.println("Hello World: " +
Thread.currentThread().getName())
```

- `()` indicates an empty parameter list.
- `->` is the arrow operator.
- The lambda body is `{ System.out.println("Hello World: " + Thread.currentThread().getName()); }`.

In this example, the lambda expression implements the `run()` method of the `Runnable` interface. The expression prints a message along with the name of the current thread.

Functional Interfaces

As mentioned earlier, a functional interface is an interface that contains one and only one abstract method. It is a way to define a contract for behavior as an argument to a method invocation. You can create your own functional interfaces using the `@FunctionalInterface` annotation. This annotation is used to indicate that an interface is a functional interface. It is optional, but it is a good practice to use it. If you try to add another abstract method to a functional interface, the compiler will throw an error.

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

Here, we have created a functional interface called `MathOperation`. It contains one abstract method, `operate()`. We can use this interface to implement different mathematical operations. For example, we can implement the addition and subtraction operations as follows:

```
MathOperation addition = (a, b) -> a + b;
MathOperation subtraction = (a, b) -> a - b;

System.out.println(addition.operate(10, 5)); // 15
System.out.println(subtraction.operate(10, 5)); // 5
```

Let's break down the above code:

- `(a, b)` is the parameter list expected by the `operate()` method.

- `->` is the arrow operator.
- `a + b` is the lambda body. It adds the two numbers and returns the result.

Passing Lambdas as Arguments

We can pass lambdas as arguments to methods. For example, we can create a method that takes a lambda expression as an argument and invokes it. Let's create a calculate method that takes two numbers and a `MathOperation` as arguments and invokes the `operate()` method of the `MathOperation` interface.

```
public class Application {
    public static void main(String[] args) {
        MathOperation addition = (a, b) -> a + b;
        MathOperation subtraction = (a, b) -> a - b;

        System.out.println(calculate(10, 5, addition)); // 15
        System.out.println(calculate(10, 5, subtraction)); // 5
    }

    public static int calculate(int a, int b, MathOperation mathOperation)
{
        return mathOperation.operate(a, b);
    }
}
```

You can also use the `Function` interface to pass lambdas as arguments. The `Function` interface is a functional interface that takes one argument and returns a result. It is defined in the `java.util.function` package. It contains the `apply()` method that takes an argument of type `T` and returns a result of type `R`. For example, we can rewrite the above code using the `Function` interface as follows:

```
public class Application {
    public static void main(String[] args) {
        Function<Integer, Integer> addition = a -> a + 5;
        Function<Integer, Integer> subtraction = a -> a - 5;

        System.out.println(calculate(10, addition)); // 15
        System.out.println(calculate(10, subtraction)); // 5
    }

    public static int calculate(int a, Function<Integer, Integer>
mathOperation) {
        return mathOperation.apply(a);
    }
}
```

Some related interfaces in the `java.util.function` package are:

- `Consumer<T>`: It takes an argument of type `T` and returns no result.

- `Supplier<T>`: It takes no argument and returns a result of type `T`.
- `Predicate<T>`: It takes an argument of type `T` and returns a boolean result.
- `UnaryOperator<T>`: It takes an argument of type `T` and returns a result of type `T`.

Streams

Motivation

Let's say we have a list of numbers, and we want to print the square of each number. We can do this using a for loop as follows:

```
public class Application {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        for (int number : numbers) {
            System.out.println(number * number);
        }
    }
}
```

The motivation for introducing streams in Java was to provide a more concise, readable, and expressive way to process sequences of data elements, such as collections or arrays. Streams were designed to address several challenges and limitations that traditional imperative programming with loops and conditionals presented:

1. **Readability and Expressiveness:** Traditional loops often involve low-level details like index manipulation and explicit iteration, which can make the code harder to read and understand. Streams provide a higher-level, declarative approach that focuses on expressing the operations you want to perform on the data rather than the mechanics of how to perform them.

2. **Code Reduction:** Streams allow you to perform complex operations on data elements in a more concise and compact manner compared to traditional loops. This leads to fewer lines of code and improved code maintainability.

3. **Parallelism:** Streams can be easily converted to parallel streams, allowing you to take advantage of multi-core processors and perform operations concurrently. This can lead to improved performance for certain types of data processing tasks.

4. **Separation of Concerns:** With traditional loops, you often mix the concerns of iterating over elements, filtering, mapping, and aggregation within a single loop. Streams encourage a separation of concerns by providing distinct operations that can be chained together in a more modular way.

5. **Lazy Evaluation:** Streams introduce lazy evaluation, which means that operations are only performed when the results are actually needed. This can lead to improved performance by avoiding unnecessary computations.

6. **Functional Programming:** Streams embrace functional programming concepts by providing operations that transform data in a functional and immutable manner. This makes it easier to reason about the behavior of your code and reduces the potential for side effects.

7. **Data Abstraction:** Streams abstract away the underlying data source, allowing you to work with different data sources (collections, arrays, I/O channels) in a consistent way. This makes your code more flexible and reusable.

In summary, the motivation behind introducing streams in Java was to provide a modern, expressive, and functional programming paradigm for processing data elements, enabling developers to write more readable, maintainable, and efficient code. Streams simplify complex data manipulations, encourage separation of concerns, and support parallel processing, contributing to improved code quality and developer productivity.

Creating Streams

There are several ways to create a stream in Java. You can create a stream from a collection, an array, or a stream builder. You can also create a stream from a file, a directory, or a string. Let's look at some examples.

Creating a Stream from a Collection

You can create a stream from a collection using the `stream()` method of the `Collection` interface. For example, let's create a stream from a list of numbers:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
Stream<Integer> stream = numbers.stream();
```

Creating a Stream from an Array

You can create a stream from an array using the `stream()` method of the `Arrays` class. For example, let's create a stream from an array of numbers:

```
int[] numbers = {1, 2, 3, 4, 5};
IntStream stream = Arrays.stream(numbers);
```

Creating a Stream using Stream.of()

You can create a stream from a sequence of elements using the `of()` method of the `Stream` interface. For example, let's create a stream of numbers:

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
```

Processing Streams

Once you have created a stream, you can perform various operations on it. There are two types of operations that you can perform on a stream:

1. **Intermediate Operations:** These operations are used to transform the stream into another stream. They are lazy, which means that they are not executed until a terminal operation is invoked on the stream. Some examples of intermediate operations are `filter()`, `map()`, `sorted()`, `distinct()`, `limit()`, and `skip()`.
2. **Terminal Operations:** These operations are used to produce a result from a stream. They are eager, which means that they are executed immediately. Some examples of terminal operations are `forEach()`, `count()`, `collect()`, `reduce()`, `min()`, `max()`, `anyMatch()`, `allMatch()`, and `noneMatch()`.

Intermediate Operations

Filtering

The `filter()` method is used to filter elements from a stream based on a predicate. It takes a predicate as an argument and returns a stream that contains only those elements that match the predicate. For example, let's filter out the even numbers from a stream of numbers:

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
Stream<Integer> evenNumbers = stream.filter(number -> number % 2 == 0);
```

Here, we have created a stream of numbers and filtered out the even numbers from the stream. The `filter()` method takes a predicate as an argument. A predicate is a functional interface that takes an argument and returns a boolean result. It is defined in the `java.util.function` package. It contains the `test()` method that takes an argument of type `T` and returns a boolean result. For example, let's create a predicate that checks if a number is even:

```
Predicate<Integer> isEven = number -> number % 2 == 0;
```

Here, we have created a predicate called `isEven` that checks if a number is even. We can use this predicate to filter out the even numbers from a stream of numbers as follows:

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
Stream<Integer> evenNumbers = stream.filter(isEven);
```

Mapping

The `map()` method is used to transform elements in a stream. It takes a function as an argument and returns a stream that contains the results of applying the function to each element in the stream. For example, let's convert a stream of numbers to a stream of their squares:

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
Stream<Integer> squares = stream.map(number -> number * number);
```

Here, we have created a stream of numbers and converted it to a stream of their squares. The `map()` method takes a function as an argument. A function is a functional interface that takes an argument and returns a result. It is defined in the `java.util.function` package. It contains the `apply()` method that takes an argument of type `T` and returns a result of type `R`. For example, let's create a function that converts a number to its square:

```
Function<Integer, Integer> square = number -> number * number;
```

Here, we have created a function called `square` that converts a number to its square. We can use this function to convert a stream of numbers to a stream of their squares as follows:

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
Stream<Integer> squares = stream.map(square);
```

Sorting

The `sorted()` method is used to sort elements in a stream. It takes a comparator as an argument and returns a stream that contains the elements sorted according to the comparator. For example, let's sort a stream of numbers in ascending order:

```
Stream<Integer> stream = Stream.of(5, 3, 1, 4, 2);
Stream<Integer> sortedNumbers = stream.sorted();
```

Here, we have created a stream of numbers and sorted it in ascending order. The `sorted()` method takes a comparator as an argument. A comparator is a functional interface that compares two objects of the same type. It is defined in the `java.util.function` package. It contains the `compare()` method that takes two arguments of type `T` and returns an integer result. For example, let's create a comparator that compares two numbers:

```
Comparator<Integer> comparator = (number1, number2) -> number1 - number2;
```

Here, we have created a comparator called `comparator` that compares two numbers. We can use this comparator to sort a stream of numbers in ascending order as follows:

```
Stream<Integer> stream = Stream.of(5, 3, 1, 4, 2);
Stream<Integer> sortedNumbers = stream.sorted(comparator);
```

Terminal Operations

Iterating

The `forEach()` method is used to iterate over the elements in a stream. It takes a consumer as an argument and invokes the consumer for each element in the stream. For example, let's iterate over a stream of numbers and print each number:

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
stream.forEach(number -> System.out.println(number));
```

Reducing

The `reduce()` method is used to reduce the elements in a stream to a single value. It takes an identity value and a binary operator as arguments and returns the result of applying the binary operator to the identity value and the elements in the stream. For example, let's find the sum of all the numbers in a stream:

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
int sum = stream.reduce(0, (number1, number2) -> number1 + number2);
```

Here, we have created a stream of numbers and found the sum of all the numbers in the stream. The `reduce()` method takes an identity value and a binary operator as arguments. A binary operator is a functional interface that takes two arguments of the same type and returns a result of the same type. It is defined in the `java.util.function` package. It contains the `apply()` method that takes two arguments of type `T` and returns a result of type `T`. For example, let's create a binary operator that adds two numbers:

```
BinaryOperator<Integer> add = (number1, number2) -> number1 + number2;
```

Here, we have created a binary operator called `add` that adds two numbers. We can use this binary operator to find the sum of all the numbers in a stream as follows:

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
int sum = stream.reduce(0, add);
```

Collecting

The `collect()` method is used to collect the elements in a stream into a collection. It takes a collector as an argument and returns the result of applying the collector to the elements in the stream. For example, let's collect the elements in a stream into a list:

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
List<Integer> numbers = stream.collect(Collectors.toList());
```

You can now use the `toList()` method on streams to collect the elements in a stream into a list.

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
List<Integer> numbers = stream.toList();
```

Similarly, you can use the `toSet()` method on streams to collect the elements in a stream into a set.

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
Set<Integer> numbers = stream.toSet();
```

Finding the first element

The `findFirst()` method is used to find the first element in a stream. It returns an `Optional` that contains the first element in the stream. For example, let's find the first even number in a stream of numbers:

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
Optional<Integer> firstEvenNumber = stream.filter(number -> number % 2 ==
0).findFirst();
```
