Adv Java 03 - Lambdas & Streams

Agenda

- Key Terms
    - Lambdas
    - Streams
    - Functional Interfaces
- Lambdas Expressions
    - Motivation
    - Examples
        - Runnable
        - Addition
        - Passing Lambdas as Arguments
        - Lambdas in Collections
        - Sorting Example
- Streams
    - Basics
        - Creation
        - Intermediate Operations
            - Filtering, Mapping, Sorting etc.
        - Terminal Operations
            - Iterating, Reducing, Collecting etc
    - Examples
    - Advantages of Streams
    - Sequential Streams & Parallel Streams
- Additional Reading
    - Collect Method()
    - Collectors Interface

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`

Lambda Expressions

Lambda expressions, also known as anonymous functions, provide a way to create concise and expressive code by allowing the definition of a function in a more compact form.

The basic syntax of a lambda expression consists of the parameter list, the arrow (->), and the body. The body can be either an expression or a block of statements.

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

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.

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

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.

Lambda expressions are most commonly used with functional interfaces, which are interfaces containing only one abstract method. Java 8 introduced the @FunctionalInterface annotation to mark such interfaces.

```
@FunctionalInterface
interface MyFunctionalInterface {
    void myMethod();
}
```

Examples

Let's start with some simple examples to illustrate the basic syntax:

1. Hello World Runnable

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.

```
// Traditional approach
Runnable traditionalRunnable = new Runnable() {
    @Override
    public void run() {
        System.out.println("Hello, World!");
    }
};
```

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.

Using Lambda Expression.

```java
// Lambda expression
Runnable lambdaRunnable = () -> System.out.println("Hello, World!");
```

2. Add Numbers

```java
// Traditional approach
MathOperation traditionalAddition = new MathOperation() {
    @Override
    public int operate(int a, int b) {
        return a + b;
    }
};
```

Using Lambda expression

```java
MathOperation lambdaAddition = (a, b) -> a + b;
```

3. Lambda Expressions with Parameters

Lambda expressions can take parameters, making them versatile for various use cases.

```java
NumberChecker traditionalChecker = new NumberChecker() {
    @Override
    public boolean check(int number) {
        return number % 2 == 0;
    }
};
```

Using Lambda expression

```java
NumberChecker lambdaChecker = number -> number % 2 == 0;
```

4. Lambda Expressions in Collections

Lambda expressions are commonly used with collections for concise iteration and processing.

Filtering a List Example (Traditonal Approach)

```java
List<String> fruits = Arrays.asList("Apple", "Banana", "Orange", "Mango");

// Traditional approach
List<String> filteredTraditional = new ArrayList<>();
for (String fruit : fruits) {
    if (fruit.startsWith("A")) {
        filteredTraditional.add(fruit);
    }
}
```

Using Lambda expression & Java Stream API

```java
List<String> filteredLambda = fruits.stream()
                                    .filter(fruit -> fruit.startsWith("A"))
                                    .collect(Collectors.toList());
```

4. Sorting Example

Method references provide a shorthand notation for lambda expressions, making the code even more concise.

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

// Lambda expression for sorting
Collections.sort(names, (a, b) -> a.compareTo(b));

// Method reference for sorting
Collections.sort(names, String::compareTo);
```

Java 8 Streams

Streams

A stream in Java is simply a wrapper around a data source, allowing us to perform bulk operations on the data in a convenient way. The Java Stream API, introduced in Java 8, is a powerful abstraction for processing sequences of elements, such as collections or arrays, in a functional and declarative way.

Streams are designed to be used in a chain of operations, allowing you to create complex data processing pipelines.

In this tutorial we will learn about Sequential Streams, Parallel Streams and Collect() Method of stream.

1. Creating Streams

Example 1: Creating a Stream from a Collection

```
List<String> fruits = Arrays.asList("Apple", "Banana", "Orange", "Mango");

// Creating a stream from a collection
Stream<String> fruitStream = fruits.stream();
```

Example 2: Creating a Stream from an Array

```
String[] cities = {"New York", "London", "Tokyo", "Paris"};

// Creating a stream from an array
Stream<String> cityStream = Arrays.stream(cities);
```

Example 3: Creating a Stream of Integers

```
IntStream intStream = IntStream.rangeClosed(1, 5);
// Creating a stream of integers
intStream.forEach(System.out::println); // Output: 1 2 3 4 5
```

2. Intermediate Operations

Intermediate operations are operations that transform a stream into another stream. They are lazy, meaning they don't execute until a terminal operation is invoked. There are two types of operations that you can perform on a stream: Some examples of intermediate operations are filter(), map(), sorted(), distinct(), limit(), and skip().

Filtering and Mapping Example:

```
List<String> fruits = Arrays.asList("Apple", "Banana", "Orange", "Mango");

// Filtering fruits starting with 'A' and converting to uppercase
List<String> result = fruits.stream()
                            .filter(fruit -> fruit.startsWith("A"))
                            .map(String::toUpperCase)
                            .collect(Collectors.toList());

System.out.println(result); // Output: [APPLE]
```

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);
```

3. Terminal operations

Terminal operations trigger the processing of elements and produce a result or a side effect. They are the final step in a stream pipeline. 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().

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:

```java
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:

```java
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:

```java
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:

```java
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.

```java
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:

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

More Examples

Example1: Collecting into a List

```java
List<String> fruits = Arrays.asList("Apple", "Banana", "Orange", "Mango");
```

```
// Collecting filtered fruits into a new list
List<String> result = fruits.stream()
                            .filter(fruit -> fruit.length() > 5)
                            .collect(Collectors.toList());

System.out.println(result); // Output: [Banana, Orange]
```

Example2: Counting Elements

```
List<String> fruits = Arrays.asList("Apple", "Banana", "Orange", "Mango");

// Counting the number of fruits
long count = fruits.stream()
                   .filter(fruit -> fruit.length() > 5)
                   .count();

System.out.println("Number of fruits: " + count); // Output: Number of fruits: 2
```

Example3: Joining Strings

```
List<String> words = Arrays.asList("Hello", " ", "Stream", " ", "API");

// Concatenating strings
String result = words.stream()
                    .collect(Collectors.joining());

System.out.println("Concatenated String: " + result); // Output: Concatenated String:
Hello
```

Advantages of Streams

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: **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.

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.

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.

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.

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.

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.

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.

Multithreading using Java Streams

Sequential Streams

By default, any stream operation in Java is processed sequentially, unless explicitly specified as parallel.

Sequential streams use a single thread to process the pipeline:

```
List<Integer> listOfNumbers = Arrays.asList(1, 2, 3, 4);
listOfNumbers.stream().forEach(number ->
    System.out.println(number + " " + Thread.currentThread().getName())
);
```

The output of this sequential stream is predictable. The list elements will always be printed in an ordered sequence:

```
1 main
2 main
3 main
4 main
```

Parallel Streams

Stream API also simplifies multithreading by providing the `parallelStream()` method that runs operations over stream’s elements in parallel mode. Any stream in Java can easily be transformed from sequential to parallel.

We can achieve this by adding the parallel method to a sequential stream or by creating a stream using the parallelStream method of a collection:

The code below allows to run method doWork() in parallel for every element of the stream:

```
list.parallelStream().forEach(element -> doWork(element));
```

For the above sequential example, the code will looks like this -

```
List<Integer> listOfNumbers = Arrays.asList(1, 2, 3, 4);
listOfNumbers.parallelStream().forEach(number ->
    System.out.println(number + " " + Thread.currentThread().getName())
);
```

Parallel streams enable us to execute code in parallel on separate cores. The final result is the combination of each individual outcome.

However, the order of execution is out of our control. It may change every time we run the program:

```
4 ForkJoinPool.commonPool-worker-3
2 ForkJoinPool.commonPool-worker-5
1 ForkJoinPool.commonPool-worker-7
3 main
```

Parallel streams make use of the fork-join framework and its common pool of worker threads. Parallel processing may be beneficial to fully utilize multiple cores. But we also need to consider the overhead of managing multiple threads, memory locality, splitting the source and merging the results. Refer this Article to learn more about when to use parallel streams.

Additonal Topics

Collect() Method

A stream represents a sequence of elements and supports different kinds of operations that lead to the desired result. The source of a stream is usually a Collection or an Array, from which data is streamed from.

Streams differ from collections in several ways; most notably in that the streams are not a data structure that stores elements. They're functional in nature, and it's worth noting that operations on a stream produce a result and typically return another stream, but do not modify its source.

To "solidify" the changes, you **collect** the elements of a stream back into a Collection.

The `stream.collect()` method is used to perform a mutable reduction operation on the elements of a stream. It returns a new mutable object containing the results of the reduction operation.

This method can be used to perform several different types of reduction operations, such as:

- Computing the sum of numeric values in a stream.
- Finding the minimum or maximum value in a stream.
- Constructing a new String by concatenating the contents of a stream.
- Collecting elements into a new List or Set.

```
public class CollectExample {
   public static void main(String[] args) {
      Integer[] intArray = {1, 2, 3, 4, 5};
```

```
        // Creating a List from an array of elements
        // using Arrays.asList() method
        List<Integer> list = Arrays.asList(intArray);

        // Demo1: Collecting all elements of the list into a new
        // list using collect() method
        List<Integer> evenNumbersList = list.stream()
              .filter(i -> i%2 == 0)
              .collect(toList());
        System.out.println(evenNumbersList);

        // Demo2: finding the sum of all the values
        // in the stream
        Integer sum = list.stream()
           .collect(summingInt(i -> i));
        System.out.println(sum);

        // Demo3: finding the maximum of all the values
        // in the stream
        Integer max = list.stream()
           .collect(maxBy(Integer::compare)).get();
        System.out.println(max);

        // Demo4: finding the minimum of all the values
        // in the stream
        Integer min = list.stream()
           .collect(minBy(Integer::compare)).get();
        System.out.println(min);

        // Demo5: counting the values in the stream
        Long count = list.stream()
           .collect(counting());
        System.out.println(count);
    }
}
```

In Demo1: We use the stream() method to get a stream from the list. We filter the even elements and collect them into a new list using the collect() method.

In Demo2: We use the collect() method summingInt(ToIntFunction) as an argument. The summingInt() method returns a collector that sums the integer values extracted from the stream elements by applying an int producing mapping function to each element.

In Demo 3: We use the collect() method with maxBy(Comparator) as an argument. The maxBy() accepts a Comparator and returns a collector that extracts the maximum element from the stream according to the given Comparator.

Lets learn more about Collectors.

Collectors Class

Collectors represent implementations of the Collector interface, which implements various useful reduction operations, such as accumulating elements into collections, summarizing elements based on a specific parameter, etc.

All predefined implementations can be found within the Collectors class.

Within the Collectors class itself, we find an abundance of unique methods that deliver on the different needs of a user. One such group is made of summing methods - `summingInt()`, `summingDouble()` and `summingLong()`.

Let's start off with a basic example with a List of Integers:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
Integer sum = numbers.stream().collect(Collectors.summingInt(Integer::intValue));
System.out.println("Sum: " + sum);
```

We apply the .stream() method to create a stream of Integer instances, after which we use the previously discussed `.collect()` method to collect the elements using `summingInt()`. The method itself, again, accepts the `ToIntFunction`, which can be used to reduce instances to an integer that can be summed.

Since we're using Integers already, we can simply pass in a method reference denoting their `intValue`, as no further reduction is needed.

More often than not - you'll be working with lists of custom objects and would like to sum some of their fields. For instance, we can sum the quantities of each product in the productList, denoting the total inventory we have.

Let us try to understand one of these methods using a custom class example.

```
public class Product {
    private String name;
    private Integer quantity;
    private Double price;
    private Long productNumber;

    // Constructor, getters and setters
    ...
}
...
List<Product> products = Arrays.asList(
        new Product("Milk", 37, 3.60, 12345600L),
        new Product("Carton of Eggs", 50, 1.20, 12378300L),
        new Product("Olive oil", 28, 37.0, 13412300L),
        new Product("Peanut butter", 33, 4.19, 15121200L),
        new Product("Bag of rice", 26, 1.70, 21401265L)
);
```

In such a case, the we can use a method reference, such as `Product::getQuantity` as our `ToIntFunction`, to reduce the objects into a single integer each, and then sum these integers:

```
Integer sumOfQuantities =
products.stream().collect(Collectors.summingInt(Product::getQuantity));
System.out.println("Total number of products: " + sumOfQuantities);
```

This results in:

```
Total number of products: 174
```

You can also very easily implement your own collector and use it instead of the predefined ones, though - you can get pretty far with the built-in collectors, as they cover the vast majority of cases in which you might want to use them.

The following are examples of using the predefined collectors to perform common mutable reduction tasks:

```

     // Accumulate names into a List
     List<String> list =
people.stream().map(Person::getName).collect(Collectors.toList());

     // Accumulate names into a TreeSet
     Set<String> set =
people.stream().map(Person::getName).collect(Collectors.toCollection(TreeSet::new));

     // Convert elements to strings and concatenate them, separated by commas
     String joined = things.stream()
                           .map(Object::toString)
                           .collect(Collectors.joining(", "));

     // Compute sum of salaries of employee
     int total = employees.stream()
                          .collect(Collectors.summingInt(Employee::getSalary)));

     // Group employees by department
     Map<Department, List<Employee>> byDept
         = employees.stream()
                    .collect(Collectors.groupingBy(Employee::getDepartment));

     // Compute sum of salaries by department
     Map<Department, Integer> totalByDept
         = employees.stream()
                    .collect(Collectors.groupingBy(Employee::getDepartment,

Collectors.summingInt(Employee::getSalary)));

     // Partition students into passing and failing
     Map<Boolean, List<Student>> passingFailing =
         students.stream()
                 .collect(Collectors.partitioningBy(s -> s.getGrade() >=
PASS_THRESHOLD));
```

You can look at the offical documentation for more details on these methods.
https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html

--- End ---