Streams

JAVA STREAMS: A PARADIGM SHIFT IN DATA PROCESSING

introduction

- In the realm of programming, writing concise, readable, and efficient code is a craft honed over time. Java's Stream API, introduced in Java 8, has been a transformative addition, offering a more declarative approach to handling collections compared to the traditional imperative style.
- The Java Stream API represents a paradigm shift in Java's approach to handling data. It encourages developers to embrace a functional programming style, bringing the benefits of expressiveness, conciseness, and maintainability.

- **Historical Context:**

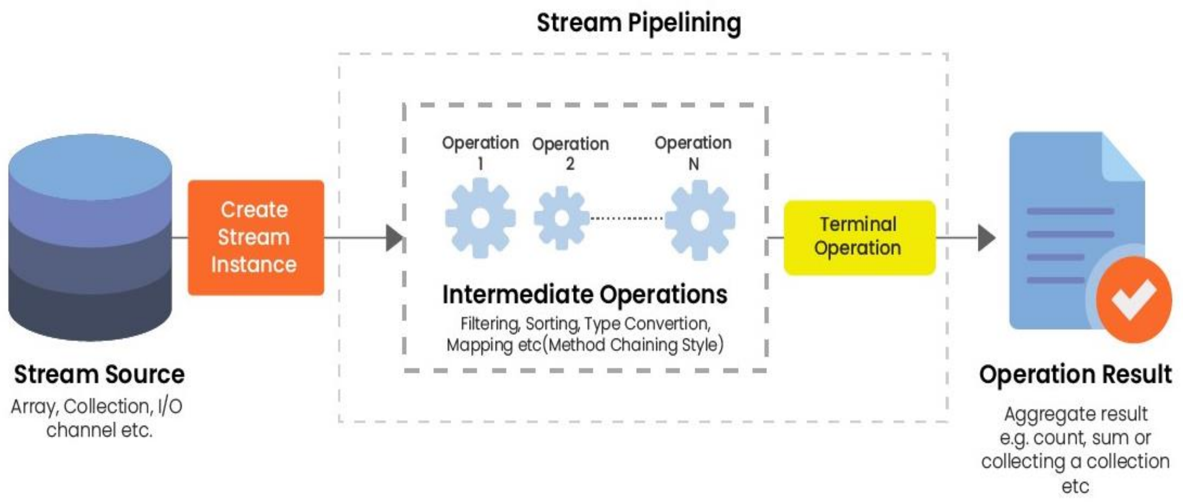
Before Java 8, Java was predominantly an imperative programming language, leading to verbose code, especially when dealing with collections. The rise of functional programming languages emphasized the need for Java to evolve.

- **The Rise of Functional Programming:**

Functional programming, emphasizing computations as evaluations of mathematical functions, can lead to more concise, predictable, and maintainable code. The Stream API was introduced to blend the functional programming paradigm into Java.

What is a Stream?

A STREAM IN JAVA REPRESENTS A SEQUENCE OF ELEMENTS, TYPICALLY FROM A COLLECTION, THAT CAN BE PROCESSED IN PARALLEL OR SEQUENTIALLY. UNLIKE COLLECTIONS, STREAMS DO NOT STORE DATA BUT CONVEY IT, ALLOWING MULTIPLE OPERATIONS TO BE DEFINED ON THE DATA SOURCE.


Stream Pipelining
Stream Source
Array, Collection, I/O channel etc.
Create Stream Instance
Operation 1
Operation 2
Operation N
Intermediate Operations
Filtering, Sorting, Type Convertion, Mapping etc(Method Chaining Style)
Terminal Operation
Operation Result
Aggregate result e.g. count, sum or collecting a collection etc

How Streams Transform Operations

- **The Stream API's power lies in transforming operations on data from external iterations to internal iterations. This allows for more optimized iterations and a seamless flow of data through a pipeline of operations.**

- **Common Operations with Streams**

The Stream API provides a multitude of operations that can be chained together to form complex data manipulations. These operations can be broadly categorized into intermediate and terminal operations.

Filtering

- Filtering: Selectively picks elements based on a given condition.

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");
List<String>namesStartingWithA= names.stream()
                                        .filter(n -> n.startsWith("A"))
                                        .collect(Collectors.toList());
```

Mapping

- **Mapping: Transforms each element in   the stream.**

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");
List<Integer>nameLengths= names.stream()
                                        .map(String::length)
                                        .collect(Collectors.toList());
```

Aggregating

- Aggregation operations help condense the stream into a single summary result. The Stream API provides methods like sum, average, count, and reduce.

- Example, using reduce to concatenate strings:

```
List<String>names = Arrays.asList("Alice", "Bob", "Charlie", "David");
String concatenatedNames = names.stream()
                    .reduce("",(name1, name2)-> name1 + " " + name2);
```

Sorting:

- **The Stream API facilitates sorting through the sorted method. You can use natural ordering or provide a custom comparator.**

- **Example, sorting names by length:**

```
List<String> sortedByLength = names.stream()
                              .sorted(Comparator.comparingInt(String::length))
                              .collect(Collectors.toList());
```

Distinct:

- Sometimes, you may need to eliminate duplicates or limit the number of results from a stream.
- distinct removes duplicate values:

```
List<Integer> numbers = Arrays.asList(1, 2, 2, 3, 3, 3, 4, 4);
List<Integer> uniqueNumbers = numbers.stream()
                                     .distinct()
                                     .collect(Collectors.toList());
```

limiting:

- **limit restricts the size of the result:**

```
List<String> firstTwoNames = names.stream()
                                  .limit(2)
                                  .collect(Collectors.toList());
```

FlatMapping:

- FlatMap is a special operation that can transform each element of the stream into zero or more elements by "flattening" the structure. It's particularly useful when dealing with streams of collections.
- Example, finding unique characters in a list of strings:

```
List<String> listOfWords = Arrays.asList("Hello", "World");
List<String> uniqueChars = listOfWords.stream()
                                        .map(w -> w.split(""))
                                       .flatMap(Arrays::stream)
                                       .distinct()
                                       .collect(Collectors.toList());
```

Intermediate vs. Terminal Operations:

Intermediate operations:

- **Intermediate Operations: These operations return another stream and set up a new operation on the stream pipeline. Examples include filter, map, and sorted.**

Terminal Operations:

- **Terminal Operations: These are operations that produce a result or a side-effect, causing the stream pipeline to be processed. Examples include collect, forEach, and reduce.**

Lazy Evaluation in Streams:

- Lazy evaluation refers to the deferment of the actual computation until it's absolutely necessary. In the context of streams, this means that intermediate operations do not process the data when they're called. Instead, they set up a new operation on the stream pipeline and wait. Actual computation occurs only when a terminal operation is invoked.
- This behavior has several benefits:

Performance Optimizations: Since the data is not processed until required, you can avoid unnecessary computations, especially when chained operations are involved.

Short-Circuiting: Some operations, like findFirst or anyMatch, don't need to process the whole dataset to produce a result. With lazy evaluation, as soon as the result is found, the processing stops.

- **Example to discuss lazy Evaluation:**

```
List<String> list = Arrays.asList(“abc1”, “abc2”, “abc3”);
Optional<String> stream = list.stream()
                                    .filter(element -> {
                                         log.info("filter() was called");
                                         return element.contains("2");})
                                     .map(element -> {
                                         log.info("map() was called");
                                         return element.toUpperCase();})
                                     .findFirst();
```

In the above example:

The resulting log shows that we called the filter() method twice and the map() method once. This is because the pipeline executes vertically. In our example, the first element of the stream didn't satisfy the filter's predicate. Then we invoked the filter() method for the second element, which passed the filter. Without calling the filter() for the third element, we went down through the pipeline to the map() method.

- The findFirst() operation satisfies by just one element. So in this particular example, the lazy invocation allowed us to avoid two method calls, one for the filter() and one for the map().

Infinite Streams:

- Lazy evaluation also makes it possible to work with infinite streams. Since computations are deferred, you can define a stream with an infinite source and yet not run into issues, as long as you limit the operations you perform on it.

- For instance, using the Stream.iterate method, one can create an infinite stream of even numbers:

```
Stream<Integer> infiniteEvens = Stream.iterate(0, n -> n + 2);
```

- But, if you wish to collect the first 10 even numbers from this stream, you can do so without processing the entire infinite source:

```
List<Integer> firstTenEvens = infiniteEvens.limit(10).collect(Collectors.toList());
```

Parallel Processing with Streams:

- The ability to easily parallelize operations on data is one of the standout features of the Java Stream API. With the increasing availability of multi-core processors, parallel processing has become crucial in exploiting the full power of modern hardware. Thankfully, the Stream API provides an intuitive mechanism to harness this potential.

- Introducing Parallel Streams:

Parallel streams split the data into multiple chunks, with each chunk being processed by a separate thread. This concurrent processing can lead to significant performance improvements for CPU-bound tasks, especially when dealing with large datasets.

Creating a parallel stream

- Creating a parallel stream is remarkably simple. You can convert a regular stream into a parallel stream using the parallel() method or directly create one from a collection using parallelStream():

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);


// Using parallel()
Stream<Integer> parallelStream1 = numbers.stream().parallel();


// Using parallelStream()
Stream<Integer> parallelStream2 = numbers.parallelStream();
```

Under the Hood:

- **Under the Hood: The Fork/Join Framework:**

Java's parallel streams leverage the Fork/Join Framework introduced in Java 7. This framework is designed for parallelizing recursive tasks, efficiently using a pool of worker threads. The Stream API divides the data into smaller chunks and distributes them among available threads in the Fork/Join pool for concurrent processing.

- **Benefits and Caveats:**

While parallel processing can provide significant speedups, it's not a silver bullet. Some considerations to keep in mind:

Overhead   Stateful Operations   Ordering   Shared Data Structures

- Overhead: Parallelism introduces overhead due to tasks' decomposition, threads management, and results' combination. For small datasets or tasks, this overhead might outweigh the benefits, making the parallel version slower than the sequential one.

- Stateful Operations: Stateful lambda expressions (those that maintain state across invocations) can lead to unpredictable results when used in parallel streams. It's best to ensure that operations are stateless and free of side-effects.

- Ordering: Parallel processing might not maintain the order of the original data, especially during operations like map or filter. If order is essential, it can reduce the effectiveness of parallelism since additional steps are required to maintain it.

- Shared Data Structures: Using shared mutable data structures can lead to data corruption or concurrency issues. It's recommended to use concurrent data structures or avoid shared mutable data altogether.

A Practical Example

- Consider a scenario where you want to compute the square of each number in a large list:

```
List<Integer> numbers = /* ... a large list ... */;
List<Integer> squares = numbers.parallelStream()
                                .map(n -> n * n)
                                .collect(Collectors.toList());
```

- By merely using parallelStream(), the task is automatically split and processed concurrently, potentially providing a significant speedup, especially for larger lists.

Conclusion & References :

- **Conclusion:**

The Java Stream API represents a significant stride in the evolution of Java as a programming language. It promotes a functional programming style that leads to more concise, readable, and often more efficient code. By leveraging its features, like lazy evaluation and parallel processing, developers can craft optimized and elegant solutions to data processing challenges.

References:

- Java how to program tenth edition: paul deitel, Harvey deitel -> Java how to program tenth edition | GitHub
- The Power of Java Stream API article, by Alexander Obregon -> Mastering Java Stream API: A Guide | Medium
- Baeldung website by Eugen -> The Java 8 Stream API Tutorial | Baeldung