Stream API Java

[java.util.stream (Java Platform SE 8) (oracle.com)]

Java Streams:

- **Description**: Java Streams provide functional-style operations for processing sequences of elements. They support map-reduce transformations on collections.
- **Key Abstraction**: The main abstraction introduced is the `Stream` interface. It represents a sequence of elements supporting various operations.
- **Stream Characteristics**:
  - No Storage: Streams do not store elements; they convey elements from a source to operations through a pipeline.
  - Functional in Nature: Stream operations produce a result without modifying the source.
  - Laziness-Seeking: Operations are often implemented lazily, providing optimization opportunities.
  - Possibly Unbounded: Streams can be infinite, allowing for short-circuiting operations.
  - Consumable: Elements are visited once during the life of a stream, like an iterator.
- **Stream Sources**:
  - From Collection: `stream()` and `parallelStream()` methods.
  - From Array: `Arrays.stream(Object[])`.
  - From Factory Methods: `Stream.of(Object[])`, `IntStream.range(int, int)`, etc.
  - From File I/O: `BufferedReader.lines()`, `Files.lines()`.
  - From Other Sources: Random numbers, file paths, etc.

- **Stream Operations**:
    - Intermediate: Produce a new stream (lazy), e.g., `filter()`, `map()`.
    - Terminal: Produce a result or side-effect, e.g., `forEach()`, `collect()`.
- **Stream Pipelines**:
    - Comprise a source, zero or more intermediate operations, and a terminal operation.
    - Operations are lazily executed; traversal begins upon terminal operation invocation.
- **Parallelism**:
    - Streams can execute operations either in serial or parallel.
    - Parallel execution is facilitated by aggregate operations and explicit parallelism request.
- **Stateless and Stateful Operations**:
    - Stateless: Operations like `filter` and `map` retain no state from previous elements.
    - Stateful: Operations like `distinct` and `sorted` may incorporate state from previous elements.
- **Short-circuiting Operations**:
    - Produce a finite stream result even with infinite input.
- **Non-interference and Stateless Behaviors**:
    - Streams enable aggregate operations over various data sources, even non-thread-safe collections.
    - Behavioral parameters should be non-interfering and stateless to prevent exceptions or incorrect results.
    - For well-behaved stream sources, the source can be modified before the terminal operation commences and those modifications will be reflected in the covered elements.
- **Side-effects**:

- Side-effects in stream operations are discouraged due to potential thread-safety hazards.
- Operations like `forEach` and `peek` operate via side-effects but should be used with care.

Examples:

1. **Sum of Weights of Red Widgets**:

```
int sum = widgets.stream()
                .filter(b -> b.getColor() == RED)
                .mapToInt(b -> b.getWeight())
                .sum();
```

2. **Searching for Matches using Regular Expression**:

```
List<String> results = stream.filter(s -> pattern.matcher(s).matches())
                             .collect(Collectors.toList());
```

3. **Example of a Stateful Lambda**:

```
Set<Integer> seen = Collections.synchronizedSet(new HashSet<>());
stream.parallel().map(e -> { if (seen.add(e)) return 0; else return e; })...
```

4. **Example of source can be modified before the terminal operation:**

```
List<String> l = new ArrayList(Arrays.asList("one", "two"));
Stream<String> sl = l.stream();
l.add("three");
String s = sl.collect(joining(" "))
```

Ordering

- **Encounter Order**:
  - Streams may or may not have a defined encounter order.
  - The encounter order depends on the source and intermediate operations.
- **Intrinsic Ordering**:
  - Certain stream sources, like Lists or arrays, are intrinsically ordered.
  - Others, like HashSet, are not ordered.
- **Impact of Intermediate Operations**:
  - Some intermediate operations, like `sorted()`, impose an encounter order.
  - Others, like `unordered()`, render an ordered stream unordered.
- **Terminal Operations**:
  - Certain terminal operations, like `forEach()`, may ignore encounter order.
- **Performance Considerations**:
  - For sequential streams, encounter order affects determinism but not performance.
  - For parallel streams, relaxing ordering constraints can sometimes improve efficiency.
  - Certain operations, like `distinct()` or `groupingBy()`, can be more efficient without ordering constraints.
  - However, operations like `limit()` may require buffering for proper ordering, undermining parallelism.
- **De-ordering Streams**:
  - If encounter order is not important, explicitly de-ordering the stream with `unordered()` may improve parallel performance for some operations.

- Most stream pipelines parallelize efficiently even under ordering constraints.

Reduction Operations

A **reduction** operation (also known as a **fold**) combines a sequence of input elements into a single summary result by repeatedly applying a combining operation, such as finding the sum or maximum of a set of numbers, or accumulating elements into a list.

- **General Reduction Operations**:
  - Java Streams provide multiple forms of general reduction operations, such as `reduce()` and `collect()`.
  - Specialized forms include `sum()`, `max()`, or `count()`.
- **Example**:

  ```
  int sum = numbers.stream().reduce(0, (x, y) -> x + y);
  ```

- **Parallelization**:
  - Properly constructed reduce operations are inherently parallelizable if the combining functions are associative and stateless.
  - Example:

    ```
    int sum = numbers.parallelStream().reduce(0, Integer::s
    um);
    ```

- **General Form of Reduction**:
  - A general reduction operation requires an identity element, an accumulator function, and a combiner function.
  - Formal representation:

    ```
    <U> U reduce(U identity, BiFunction<U, ? super T, U> ac
    ```

```
cumulator, BinaryOperator<U> combiner);
```

Mutable Reduction

A **mutable reduction operation** accumulates input elements into a mutable result container, such as a Collection or StringBuilder, as it processes the elements in the stream.

- **Example**:

  ```
  String concatenated = strings.reduce("", String::concat);
  ```

- **Performance Considerations**:
  - Performance can be improved by using mutable containers like StringBuilder.
  - Mutable reduction is achieved using the `collect()` operation.
- **Form of Mutable Reduction**:
  - Requires a supplier function, an accumulator function, and a combiner function.
  - Formal representation:

    ```
    <R> R collect(Supplier<R> supplier, BiConsumer<R, ? super T> accumulator, BiConsumer<R, R> combiner);
    ```

- **Example with StringBuilder**:

  ```
  List<String> strings = stream.collect(ArrayList::new, ArrayList::add, ArrayList::addAll);
  ```

- **Collector Abstraction**:
  - A Collector captures the supplier, accumulator, and combiner functions for mutable reduction.
  - Example:

```
List<String> strings = stream.collect(Collectors.toList
());
```

- **Advantage of Collectors**:
    - Collectors provide composability and offer predefined factories for collectors, including combinators that transform one collector into another.
- **Example**:

```
Collector<Employee, ?, Integer> summingSalaries = Collecto
rs.summingInt(Employee::getSalary);
```

- **Parallelization Considerations**:
    - Collect operations can only be parallelized if certain conditions are met, ensuring equivalent results regardless of splitting computation.