Side Effects

A method or an expression has a side effect if, in addition to returning or producing a value, it also modifies the state of the computer. Examples include mutable reductions (operations that use the `collect` operation; see the section Reduction for more information) as well as invoking the `System.out.println` method for debugging. The JDK handles certain side effects in pipelines well. In particular, the `collect` method is designed to perform the most common stream operations that have side effects in a parallel-safe manner. Operations like `forEach` and`peek` are designed for side effects; a lambda expression that returns void, such as one that invokes `System.out.println`, can do nothing but have side effects. Even so, you should use the `forEach` and `peek` operations with care; if you use one of these operations with a parallel stream, then the Java runtime may invoke the lambda expression that you specified as its parameter concurrently from multiple threads. In addition, never pass as parameters lambda expressions that have side effects in operations such as `filter` and `map`. The following sections discuss interference and stateful lambda expressions, both of which can be sources of side effects and can return inconsistent or unpredictable results, especially in parallel streams. However, the concept of laziness is discussed first, because it has a direct effect on interference.

Laziness

All intermediate operations are *lazy*. An expression, method, or algorithm is lazy if its value is evaluated only when it is required. (An algorithm is *eager* if it is evaluated or processed immediately.) Intermediate operations are lazy because they do not start processing the contents of the stream until the terminal operation commences. Processing streams lazily enables the Java compiler and runtime to optimize how they process streams. For example, in a pipeline such as the `filter`-`mapToInt`-`average` example described in the section Aggregate Operations, the `average` operation could obtain the first several integers from the stream created by the `mapToInt` operation, which obtains elements from the `filter` operation. The`average` operation would repeat this process until it had obtained all required elements from the stream, and then it would calculate the average.

Interference

Lambda expressions in stream operations should not *interfere*. Interference occurs when the source of a stream is modified while a pipeline processes the stream. For example, the following code attempts to concatenate the strings contained in the `List listOfStrings`. However, it throws a `ConcurrentModifiedException`:

```
try {
    List<String> listOfStrings =
        new ArrayList<>(Arrays.asList("one", "two"));

    // This will fail as the peek operation will attempt to add the
    // string "three" to the source after the terminal operation has
    // commenced.

    String concatenatedString = listOfStrings
        .stream()

        // Don't do this! Interference occurs here.
        .peek(s -> listOfStrings.add("three"))

        .reduce((a, b) -> a + " " + b)
        .get();

    System.out.println("Concatenated string: " + concatenatedString);

} catch (Exception e) {
    System.out.println("Exception caught: " + e.toString());
}
```

This example concatenates the strings contained in `listOfStrings` into an `Optional<String>` value with the `reduce` operation, which is a terminal operation. However, the pipeline here invokes the intermediate operation `peek`, which attempts to add a new element to `listOfStrings`. Remember, all intermediate operations are lazy. This means that the pipeline in this example begins execution when the operation `get` is invoked, and ends execution when the `get` operation completes. The argument of the `peek` operation attempts to modify the stream source during the execution of the pipeline, which causes the Java runtime to throw a `ConcurrentModifiedException`.

Stateful Lambda Expressions

Avoid using *stateful lambda expressions* as parameters in stream operations. A stateful lambda expression is one whose result depends on any state that might change during the execution of a pipeline. The following example adds elements from the `List` `listOfIntegers` to a new `List` instance with the `map` intermediate operation. It does this twice, first with a serial stream and then with a parallel stream:

```
List<Integer> serialStorage = new ArrayList<>();

System.out.println("Serial stream:");
listOfIntegers
    .stream()

    // Don't do this! It uses a stateful lambda expression.
    .map(e -> { serialStorage.add(e); return e; })

    .forEachOrdered(e -> System.out.print(e + " "));
System.out.println("");

serialStorage
    .stream()
    .forEachOrdered(e -> System.out.print(e + " "));
System.out.println("");

System.out.println("Parallel stream:");
List<Integer> parallelStorage = Collections.synchronizedList(
    new ArrayList<>());
listOfIntegers
    .parallelStream()

    // Don't do this! It uses a stateful lambda expression.
    .map(e -> { parallelStorage.add(e); return e; })

    .forEachOrdered(e -> System.out.print(e + " "));
System.out.println("");

parallelStorage
    .stream()
    .forEachOrdered(e -> System.out.print(e + " "));
System.out.println("");
```

The lambda expression `e -> { parallelStorage.add(e); return e; }` is a stateful lambda expression. Its result can vary every time the code is run. This example prints the following:

```
Serial stream:
8 7 6 5 4 3 2 1
8 7 6 5 4 3 2 1
Parallel stream:
8 7 6 5 4 3 2 1
1 3 6 2 4 5 8 7
```

The operation `forEachOrdered` processes elements in the order specified by the stream, regardless of whether the stream is executed in serial or parallel. However, when a stream is executed in parallel, the `map` operation processes elements of the stream specified by the Java runtime and compiler. Consequently, the order in which the lambda expression `e -> { parallelStorage.add(e); return e; }` adds elements to the `List` `parallelStorage` can vary every time the code is run. For deterministic and predictable results, ensure that lambda expression parameters in stream operations are not stateful.

Note: This example invokes the method `synchronizedList` so that the `List` `parallelStorage` is thread-safe. Remember that collections are not thread-safe. This means that multiple threads should not access a particular collection at the same time. Suppose that you do not invoke the method `synchronizedList` when creating `parallelStorage`:

```
List<Integer> parallelStorage = new ArrayList<>();
```

The example behaves erratically because multiple threads access and modify `parallelStorage` without a mechanism like synchronization to schedule when a particular thread may access the `List` instance. Consequently, the example could print output similar to the following:

```
Parallel stream:
8 7 6 5 4 3 2 1
null 3 5 4 7 8 1 2
```