Ordering

The order in which a pipeline processes the elements of a stream depends on whether the stream is executed in serial or in parallel, the source of the stream, and intermediate operations. For example, consider the following example that prints the elements of an instance of `ArrayList` with the `forEach` operation several times:

```
Integer[] intArray = {1, 2, 3, 4, 5, 6, 7, 8 };
List<Integer> listOfIntegers =
    new ArrayList<>(Arrays.asList(intArray));

System.out.println("listOfIntegers:");
listOfIntegers
    .stream()
    .forEach(e -> System.out.print(e + " "));
System.out.println("");

System.out.println("listOfIntegers sorted in reverse order:");
Comparator<Integer> normal = Integer::compare;
Comparator<Integer> reversed = normal.reversed();
Collections.sort(listOfIntegers, reversed);
listOfIntegers
    .stream()
    .forEach(e -> System.out.print(e + " "));
System.out.println("");

System.out.println("Parallel stream");
listOfIntegers
    .parallelStream()
    .forEach(e -> System.out.print(e + " "));
System.out.println("");

System.out.println("Another parallel stream:");
listOfIntegers
    .parallelStream()
    .forEach(e -> System.out.print(e + " "));
System.out.println("");

System.out.println("With forEachOrdered:");
listOfIntegers
    .parallelStream()
    .forEachOrdered(e -> System.out.print(e + " "));
System.out.println("");
```

This example consists of five pipelines. It prints output similar to the following:

```
listOfIntegers:
1 2 3 4 5 6 7 8
listOfIntegers sorted in reverse order:
8 7 6 5 4 3 2 1
Parallel stream:
3 4 1 6 2 5 7 8
Another parallel stream:
6 3 1 5 7 8 4 2
With forEachOrdered:
8 7 6 5 4 3 2 1
```

This example does the following:

- The first pipeline prints the elements of the list `listOfIntegers` in the order that they were added to the list.
- The second pipeline prints the elements of `listOfIntegers` after it was sorted by the method `Collections.sort`.
- The third and fourth pipelines print the elements of the list in an apparently random order. Remember that stream operations use internal iteration when processing elements of a stream. Consequently, when you execute a stream in parallel, the Java compiler and runtime determine the order in which to process the stream's elements to maximize the benefits of parallel computing unless otherwise specified by the stream operation.
- The fifth pipeline uses the method `forEachOrdered`, which processes the elements of the stream in the order specified by its source, regardless of whether you executed the stream in serial or parallel. Note that you may lose the benefits of parallelism if you use operations like `forEachOrdered` with parallel streams.