# Practices for Lesson 17: Parallel Streams

**Chapter 17**

# Practices for Lesson 17: Overview

## Practice Overview

In these practices, explore the parallel stream options available in Java.

### Old Style Loop

The following example iterates through an `Employee` list. Each member who is from Colorado and is an executive has their information printed out. In addition, the `sum` mutator is used to calculate the total amount of executive pay for the selected group.

**A01OldStyleLoop.java**

```
 9 public class A01OldStyleLoop {
10
11     public static void main(String[] args) {
12
13         List<Employee> eList = Employee.createShortList();
14
15         double sum = 0;
16
17         for(Employee e:eList){
18             if(e.getState().equals("CO") &&
19                 e.getRole().equals(Role.EXECUTIVE)){
20                 e.printSummary();
21                 sum += e.getSalary();
22             }
23         }
24
25         System.out.printf("Total CO Executive Pay: $%,9.2f %n", sum);
26     }
27
28 }
```

There are a couple of key points that can be made about the above code.

- All elements in the collections must be iterated through every time.
- The code is more about "how" information is obtained and less about "what" the code is trying to accomplish.
- A mutator must be added to the loop to calculate the total.
- There is no easy way to parallelize this code.

The output from the program is as follows.

**Output**

```
Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary: $120,000.00
Name: Phil Smith Role: EXECUTIVE Dept: HR St: CO Salary: $110,000.00
Name: Betty Jones Role: EXECUTIVE Dept: Sales St: CO Salary: $140,000.00
Total CO Executive Pay: $370,000.00
```

### Lambda Style Loop

The following example shows the new approach to obtaining the same data using lambda expressions. A stream is created, filtered, and printed. A `map` method is used to extract the salary data, which is then summed and returned.

**A02NewStyleLoop.java**

```
 9 public class A02NewStyleLoop {
10
11     public static void main(String[] args) {
12
13         List<Employee> eList = Employee.createShortList();
14
15         double result = eList.stream()
16             .filter(e -> e.getState().equals("CO"))
17             .filter(e -> e.getRole().equals(Role.EXECUTIVE))
18             .peek(e -> e.printSummary())
19             .mapToDouble(e -> e.getSalary())
20             .sum();
21
22         System.out.printf("Total CO Executive Pay: $%,9.2f %n",
result);
23     }
24
25 }
```

There are also some key points worth pointing out for this piece of code as well.

- The code reads much more like a problem statement.
- No mutator is needed to get the final result.
- Using this approach provides more opportunity for lazy optimizations.
- This code can easily be parallelized.

The output from the example is as follows.

**Output**

```
Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary: $120,000.00
Name: Phil Smith Role: EXECUTIVE Dept: HR St: CO Salary: $110,000.00
Name: Betty Jones Role: EXECUTIVE Dept: Sales St: CO Salary: $140,000.00
Total CO Executive Pay: $370,000.00
```

### Streams with Code

So far all the examples have used lambda expressions and stream pipelines to perform the tasks. In this example, the `Stream` class is used with regular Java statements to perform the same steps as those found in a pipeline.

**A03CodeStream.java**

```
11 public class A03CodeStream {
12
13     public static void main(String[] args) {
14
15         List<Employee> eList = Employee.createShortList();
16
17         Stream<Employee> s1 = eList.stream();
18
19         Stream<Employee> s2 = s1.filter(
20             e -> e.getState().equals("CO"));
21
22         Stream<Employee> s3 = s2.filter(
23             e -> e.getRole().equals(Role.EXECUTIVE));
24         Stream<Employee> s4 = s3.peek(e -> e.printSummary());
25         DoubleStream s5 = s4.mapToDouble(e -> e.getSalary());
26         double result = s5.sum();
27
28         System.out.printf("Total CO Executive Pay: $%,9.2f %n",
result);
29     }
30
31 }
```

Even though the approach is possible, a stream pipeline seems like a much better solution.

The output from the program is as follows.

**Output**

```
Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary: $120,000.00
Name: Phil Smith Role: EXECUTIVE Dept: HR St: CO Salary: $110,000.00
Name: Betty Jones Role: EXECUTIVE Dept: Sales St: CO Salary: $140,000.00
Total CO Executive Pay: $370,000.00
```

### Making a Stream Parallel

Making a stream run in parallel is pretty easy. Just call the `parallelStream` or `parallel` method in the stream. With that call, when the stream executes it uses all the processing cores available to the current JVM to perform the task.

**A04Parallel.java**

```
 9 public class A04Parallel {
10
11     public static void main(String[] args) {
12
13         List<Employee> eList = Employee.createShortList();
14
15         double result = eList.parallelStream()
16             .filter(e -> e.getState().equals("CO"))
17             .filter(e -> e.getRole().equals(Role.EXECUTIVE))
18             .peek(e -> e.printSummary())
19             .mapToDouble(e -> e.getSalary())
20             .sum();
21
22         System.out.printf("Total CO Executive Pay: $%,9.2f %n",
result);
23
24         System.out.println("\n");
25
26         // Call parallel from pipeline
27         result = eList.stream()
28             .filter(e -> e.getState().equals("CO"))
29             .filter(e -> e.getRole().equals(Role.EXECUTIVE))
30             .peek(e -> e.printSummary())
31             .mapToDouble(e -> e.getSalary())
32             .parallel()
33             .sum();
34
35         System.out.printf("Total CO Executive Pay: $%,9.2f %n",
result);
36
37         System.out.println("\n");
38
39         // Call sequential from pipeline
40         result = eList.stream()
41             .filter(e -> e.getState().equals("CO"))
42             .filter(e -> e.getRole().equals(Role.EXECUTIVE))
43             .peek(e -> e.printSummary())
44             .mapToDouble(e -> e.getSalary())
45             .sequential()
46             .sum();
47
48         System.out.printf("Total CO Executive Pay: $%,9.2f %n",
result);
49     }
50 }
```

Remember, the last call wins. So if you call the sequential method after the parallel method in your pipeline, the pipeline will execute serially.

The following output is produced for this sample program.

**Output**

```
Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary: $120,000.00
Name: Betty Jones Role: EXECUTIVE Dept: Sales St: CO Salary: $140,000.00
Name: Phil Smith Role: EXECUTIVE Dept: HR St: CO Salary: $110,000.00
Total CO Executive Pay: $370,000.00


Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary: $120,000.00
Name: Phil Smith Role: EXECUTIVE Dept: HR St: CO Salary: $110,000.00
Name: Betty Jones Role: EXECUTIVE Dept: Sales St: CO Salary: $140,000.00
Total CO Executive Pay: $370,000.00


Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary: $120,000.00
Name: Phil Smith Role: EXECUTIVE Dept: HR St: CO Salary: $110,000.00
Name: Betty Jones Role: EXECUTIVE Dept: Sales St: CO Salary: $140,000.00
Total CO Executive Pay: $370,000.00
```

### Stateful Versus Stateless Operations

You should avoid using stateful operations on collections when using stream pipelines. The `collect` method and `Collectors` class have been designed to work with both serial and parallel pipelines.

**A05AvoidStateful.java**

```
11 public class A05AvoidStateful {
12
13     public static void main(String[] args) {
14
15         List<Employee> eList = Employee.createShortList();
16         List<Employee> newList01 = new ArrayList<>();
17         List<Employee> newList02 = new ArrayList<>();
18
19         eList.parallelStream() // Not Parallel. Bad.
20             .filter(e -> e.getDept().equals("Eng"))
21             .forEach(e -> newList01.add(e));
22
23         newList02 = eList.parallelStream() // Good Parallel
24             .filter(e -> e.getDept().equals("Eng"))
25             .collect(Collectors.toList());
26
27     }
28 }
```

Lines 19 to 21 show you how NOT to extract data from a pipeline. Your operations may not be thread safe. Lines 23 to 25 demonstrate the correct method for saving data from a pipeline using the `collect` method and `Collectors` class.

### Deterministic and Non-Deterministic Operations

Most stream pipelines are deterministic. That means that whether the pipeline is processed serially or in parallel the result will be the same.

**A06Determine.java**

```
10 public class A06Determine {
11
12     public static void main(String[] args) {
13
14         List<Employee> eList = Employee.createShortList();
15
16         double r1 = eList.stream()
17             .filter(e -> e.getState().equals("CO"))
18             .mapToDouble(Employee::getSalary)
19             .sequential().sum();
20
21         double r2 = eList.stream()
22             .filter(e -> e.getState().equals("CO"))
23             .mapToDouble(Employee::getSalary)
24             .parallel().sum();
25
26         System.out.println("The same: " + (r1 == r2));
```

```
27     }
28 }
```

The example shows that the result for a sum is the same that is processed using either highlighted method.

The output from the sample is as follows:

**Output**

```
The same: true
```

However, some operations are not deterministic. The `findAny()` method is a short-circuit terminal operation that may produce different results when processed in parallel.

**A07DetermineNot.java**

```
10 public class A07DetermineNot {
11
12     public static void main(String[] args) {
13
14         List<Employee> eList = Employee.createShortList();
15
16         Optional<Employee> e1 = eList.stream()
17             .filter(e -> e.getRole().equals(Role.EXECUTIVE))
18             .sequential().findAny();
19
20         Optional<Employee> e2 = eList.stream()
21             .filter(e -> e.getRole().equals(Role.EXECUTIVE))
22             .parallel().findAny();
23
24         System.out.println("The same: " +
25             e1.get().getEmail().equals(e2.get().getEmail()));
26
27     }
28 }
```

The data set used in the example is fairly small therefore the two different approaches will often produce the same result. However, with a larger data set, it becomes more likely that the results produced will not be the same.

### Reduction

The reduce method performs reduction operations for the stream libraries. The following example sums numbers 1 to 5.

**A08Reduction.java**

```
 9 public class A08Reduction {
10
11     public static void main(String[] args) {
12
13         int r1 = IntStream.rangeClosed(1, 5).parallel()
14             .reduce(0, (a, b) -> a + b);
15
16         System.out.println("Result: " + r1);
17
18         int r2 = IntStream.rangeClosed(1, 5).parallel()
19             .reduce(0, (sum, element) -> sum + element);
20
21         System.out.println("Result: " + r2);
22
23     }
```

Two examples are shown. The second example started on line 18 uses more description variables to show how the two variables are used. The left value is used as an accumulator. The value on the right is added to the value on the left. Reductions must be associative operations to get a correct result.

The output from both expressions should be the following:

**Output**

```
Result: 15
Result: 15
```

# Practice 17-1: Calculate Total Sales without a Pipeline

## Overview

In this practice, calculate the sales total for Radio Hut using the Stream class and normal Java statements.

## Assumptions

You have completed the lecture portion of this lesson and the previous practice.

## Tasks

1. Open the `SalesTxn17-01Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/ 17-ParallelStreams /practices/practice1`.
   - Select `SalesTxn17-01Prac` and click the Open Project button.
2. Expand the project directories.
3. Edit the `CalcTest` class to perform the steps in this practice.
4. Calculate the total sales for Radio Hut using the Stream class and Java statements.

Create a stream from `tList` and assign it to: `Stream<SalesTxn> s1`

Create a second stream and assign the results of the `filter` method for Radio Hut transactions: `Stream<SalesTxn> s2`

Create a third stream and assign the results from a `mapToDouble` method that returns the transaction total: `DoubleStream s3`

Sum the final stream and assign the result to: `double t1`.

5. Print the results.

   **Hint:** Be mindful of the method return types. Use the API doc to ensure that you are using the correct methods and classes to create and store results.
6. The output from your test class should be similar to the following:

```
=== Transactions Totals ===
Radio Hut Total: $3,840,000.00
```

## Practice 17-2: Calculate Sales Totals using Parallel Streams

### Overview

In this practice, calculate the sales totals from the collection of sales transactions.

### Assumptions

You have completed the lecture portion of this lesson and the previous practice.

### Tasks

1. Open the `SalesTxn17-02Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/ 17-ParallelStreams /practices/practice2`.
   - Select `SalesTxn17-02Prac` and click the Open Project button.
2. Expand the project directories.
3. Edit the `CalcTest` class to perform the steps in this practice.
4. Calculate the total sales for Radio Hut, PriceCo, and Best Deals.
   a. Calculate the Radio Hut total using the `parallelStream` method. The pipeline should contain the following methods: `parallelStream`, `filter`, `mapToDouble`, and `sum`.
   b. Calculate the PriceCo total using the `parallel` method. The pipeline should contain the following methods: `filter`, `mapToDouble`, `parallel`, and `sum`.
   c. Calculate the Best Deals total using the `sequential` method. The pipeline should contain the following methods: `filter`, `mapToDouble`, `sequential`, and `sum`.
5. Print the results.
6. The output from your test class should be similar to the following:

```
=== Transactions Totals ===
Radio Hut Total: $3,840,000.00
PriceCo Total: $1,460,000.00
Best Deals Total: $1,300,000.00
```

# Practice 17-3: Calculate Sales Totals Using Parallel Streams and Reduce

## Overview

In this practice, calculate the sales totals from the collection of sales transactions using the `reduce` method.

## Assumptions

You have completed the lecture portion of this lesson and the previous practice.

## Tasks

1. Open the `SalesTxn17-03Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/ 17-ParallelStreams /practices/practice3`.
   - Select `SalesTxn17-03Prac` and click the Open Project button.
2. Expand the project directories.
3. Edit the `CalcTest` class to perform the steps in this practice.
4. Calculate the total sales for PriceCo using the `reduce` method instead of `sum`.
   a. Your pipeline should consist of: `filter`, `mapToDouble`, `parallel`, and `reduce`.
   b. The reduce function can be defined as: `reduce(0, (sum, e) -> sum + e)`
5. In addition, calculate the total number of transactions for PriceCo using `map` and `reduce`.
   a. Your pipeline should consist of: `filter`, `mapToInt`, `parallel`, and `reduce`.
   b. To count the transactions, use: `mapToInt(t -> 1)`
   c. The `reduce` function can be defined as: `reduce(0, (sum, e) -> sum + e).`
6. Print the results.
7. The output from your test class should be similar to the following:

```
=== Transactions Totals ===

PriceCo Total: $1,460,000.00
PriceCo Transactions:      4
```