Practices for Lesson 10: Lambda Operations

Chapter 10

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practices for Lesson 10: Overview

Practice Overview

In these practices, create lambda expressions and streams to process data in collections.

Employee List

Here is a short list of Employees and their data that will be used for the examples that follow.

```
Name: Bob Baker Role: STAFF Dept: Eng St: KS Salary: $40,000.00
Name: Jane Doe Role: STAFF Dept: Sales St: KS Salary: $45,000.00
Name: John Doe Role: MANAGER Dept: Eng St: KS Salary: $65,000.00
Name: James Johnson Role: MANAGER Dept: Eng St: MA Salary: $85,000.00
Name: John Adams Role: MANAGER Dept: Sales St: MA Salary: $90,000.00
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
```

Map

The `map` method in the `Stream` class allows you to extract a field from a stream and perform some operation or calculation on that value. The resulting values are then passed to the next stream in the pipeline.

A01MapTest.java

```
public class A01MapTest {

    public static void main(String[] args) {

        List<Employee> eList = Employee.createShortList();

        System.out.println("\n== CO Bonuses ==");
        eList.stream()
            .filter(e -> e.getRole().equals(Role.EXECUTIVE))
            .filter(e -> e.getState().equals("CO"))
            .map(e -> e.getSalary() * Bonus.byRole(e.getRole()))
            .forEach( s -> System.out.printf("Bonus paid: $%,6.2f %n", s));

```

The example prints out the bonuses for two different groups. The `filter` methods select the groups and then `map` is used to compute a result.

Output

```
== CO Bonuses ==
Bonus paid: $7,200.00
Bonus paid: $6,600.00
Bonus paid: $8,400.00
```

Peek

The `peek` method of the `Stream` class allows you to perform an operation on an element in the stream. The elements are returned to the stream and are available to the next stream in the pipeline. The `peek` method can be used to read or change data in the stream. Any changes will be made to the underlying collection.

A02MapPeekTest.java

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

```
        System.out.println("\n== CO Bonuses ==");
        eList.stream()
            .filter(e -> e.getRole().equals(Role.EXECUTIVE))
            .filter(e -> e.getState().equals("CO"))
            .peek(e -> System.out.print("Name: "
              + e.getGivenName() + " " + e.getSurName()))
            .map(e -> e.getSalary() * Bonus.byRole(e.getRole()))
            .forEach( s ->
              System.out.printf(
                "  Bonus paid: $%,6.2f %n", s));
```

In this example, after filtering the data, `peek` is used to print data from the current stream to the console. After the `map` method is called, only the data returned from `map` is available for output.

Output

```
== CO Bonuses ==
Name: Joe Bailey  Bonus paid: $7,200.00
Name: Phil Smith  Bonus paid: $6,600.00
Name: Betty Jones  Bonus paid: $8,400.00
```

Find First

The `findFirst` method of the `Stream` class finds the first element in the stream specified by the filters in the pipeline. The `findFirst` method is a terminal short-circuit operation. This means intermediate operations are performed in a lazy manner resulting in more efficient processing of the data in the stream. A terminal operation ends the processing of a pipeline.

A03FindFirst.java

```
public class A03FindFirst {

    public static void main(String[] args) {

        List<Employee> eList = Employee.createShortList();

        System.out.println("\n== First CO Bonus ==");
        Optional<Employee> result;

        result = eList.stream()
            .filter(e -> e.getRole().equals(Role.EXECUTIVE))
            .filter(e -> e.getState().equals("CO"))
            .findFirst();

        if (result.isPresent()){
            result.get().print();
        }

    }
```

The code filters the pipeline for executives in the state of Colorado. The first element in the collection that meets this criterion is returned and printed out. Notice that the type of the result variable is `Optional<Employee>`. This is a new class that allows you to determine if a value is present before trying to retrieve a result. This has advantages for concurrent applications.

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Output

```
== First CO Bonus ==

Name: Joe Bailey
Age: 62
Gender: MALE
Role: EXECUTIVE
Dept: Eng
Start date: 1992-01-05
Salary: 120000.0
eMail: joebob.bailey@example.com
Phone: 112-111-1111
Address: 111 1st St
City: Town
State: CO
Code: 11111
```

Find First Lazy

The following example compares a pipeline, which filters and iterates through an entire collection to a pipeline with a short-circuit terminal operation (`findFirst`). The peek method is used to print out a message associated with each operation.

A04FindFirstLazy.java

```
public class A04FindFirstLazy {

    public static void main(String[] args) {

        List<Employee> eList = Employee.createShortList();

        System.out.println("\n== CO Bonuses ==");
        eList.stream()
            .peek(e -> System.out.println("Stream start"))
            .filter(e -> e.getRole().equals(Role.EXECUTIVE))
            .peek(e -> System.out.println("Executives"))
            .filter(e -> e.getState().equals("CO"))
            .peek(e -> System.out.println("CO Executives"))
            .map(e -> e.getSalary() * Bonus.byRole(e.getRole()))
            .forEach( s -> System.out.printf(
              "  Bonus paid: $%,6.2f %n", s));

        System.out.println("\n== First CO Bonus ==");
        Employee tempEmp = new Employee.Builder().build();
        Optional<Employee> result = eList.stream()
            .peek(e -> System.out.println("Stream start"))
            .filter(e -> e.getRole().equals(Role.EXECUTIVE))
            .peek(e -> System.out.println("Executives"))
            .filter(e -> e.getState().equals("CO"))
            .peek(e -> System.out.println("CO Executives"))
            .findFirst();

        if (result.isPresent()){
```

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

```
            result.get().printSummary();
        }
    }
}
```

The pipeline prints out 17 different options. The second, with a short-circuit operator, prints 8. This demonstrates how lazy operations can really improve the performance of iteration through a collection.

Output

```
== CO Bonuses ==
Stream start
Stream start
Stream start
Stream start
Stream start
Stream start
Executives
CO Executives
  Bonus paid: $7,200.00
Stream start
Executives
CO Executives
  Bonus paid: $6,600.00
Stream start
Executives
CO Executives
  Bonus paid: $8,400.00

== First CO Bonus ==
Stream start
Stream start
Stream start
Stream start
Stream start
Stream start
Executives
CO Executives
Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary:
$120,000.00
```

anyMatch

The `anyMatch` method returns a `boolean` based on the specified `Predicate`. This is a short-circuiting terminal operation.

A05AnyMatch.java

```
public class A05AnyMatch {

    public static void main(String[] args) {

      List<Employee> eList = Employee.createShortList();

      System.out.println("\n== First CO Bonus ==");
```

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

```
Optional<Employee> result;

if (eList.stream().anyMatch(
    e -> e.getState().equals("CO"))){

  result = eList.stream()
    .peek(e -> System.out.println("Stream"))
    .filter(e -> e.getRole().equals(Role.EXECUTIVE))
    .filter(e -> e.getState().equals("CO"))
    .findFirst();

  if (result.isPresent()){result.get().printSummary();}
}
```

The example shows how the `anyMatch` method could be used to check for a value before executing a more detailed query.

Count

The `count` method returns the number of elements in the current stream. This is a terminal operation.

A06StreamData.java

```
List<Employee> eList = Employee.createShortList();

System.out.println("\n== Executive Count ==");
long execCount =
    eList.stream()
      .filter(e -> e.getRole().equals(Role.EXECUTIVE))
      .count();

System.out.println("Exec count: " + execCount);
```

The example returns the number of executives in Colorado and prints the result.

Output

```
== Executive Count ==
Exec count: 3
```

Max

The `max` method returns the highest matching value given a `Comparator` to rank elements. The `max` method is a terminal operation.

A06StreamData.java

```
System.out.println("Exec count: " + execCount);

System.out.println("\n== Highest Paid Exec ==");
Optional highestExec =
  eList.stream()
    .filter(e -> e.getRole().equals(Role.EXECUTIVE))
    .max(Employee::sortBySalary);

if (highestExec.isPresent()){
```

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

```
      Employee temp = (Employee) highestExec.get();
      System.out.printf(
          "Name: " + temp.getGivenName() + " "
          + temp.getSurName() + "   Salary: $%,6.2f %n ",
          temp.getSalary());
    }
```

The example shows `max` being used with a `Comparator` that has been written for the class. The `sortBySalary` method is called using a method reference. Notice the return type of `Optional`. This is not the generic version used in previous examples. Therefore, a cast is required when the object is retrieved.

Output

```
== Highest Paid Exec ==
Name: Betty Jones   Salary: $140,000.00
```

Min

The `min` method returns the lowest matching value given a `Comparator` to rank elements. The `min` method is a terminal operation.

A06StreamData.java

```
    System.out.println("\n== Lowest Paid Staff ==");
    Optional lowestStaff =
        eList.stream()
         .filter(e -> e.getRole().equals(Role.STAFF))
         .min(Comparator.comparingDouble(e -> e.getSalary()));

    if (lowestStaff.isPresent()){
      Employee temp = (Employee) lowestStaff.get();
      System.out.printf("Name: " + temp.getGivenName()
        + " " + temp.getSurName() +
        "   Salary: $%,6.2f %n ", temp.getSalary());
    }
```

In this example, a different `Comparator` is used. The `comparingDouble` static method is called to make the comparison. Notice that the example uses a lambda expression to specify the comparison field. If you look at the code closely, a method reference could be substituted instead: `Employee::getSalary`. More discussion on this subject follows in the `Comparator` section.

Output

```
== Lowest Paid Staff ==
Name: Bob Baker   Salary: $40,000.00
```

Sum

The `sum` method calculates a sum based on the stream passed to it. Notice the `mapToDouble` method is called before the stream is passed to `sum`. If you look at the `Stream` class, no `sum` method is included. Instead, a `sum` method is included in the primitive version of the `Stream` class, `IntStream`, `DoubleStream`, and `LongStream`. The `sum` method is a terminal operation.

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A07CalcSum.java

```
    System.out.println("\n== Total CO Bonus Details ==");

    result = eList.stream()
      .filter(e -> e.getRole().equals(Role.EXECUTIVE))
      .filter(e -> e.getState().equals("CO"))
      .peek(e -> System.out.print("Name: "
        + e.getGivenName() + " " + e.getSurName() + " "))
      .mapToDouble(e -> e.getSalary() * Bonus.byRole(e.getRole()))
      .peek(d -> System.out.printf("Bonus paid: $%,6.2f %n", d))
      .sum();

    System.out.printf("Total Bonuses paid: $%,6.2f %n", result);
```

Looking at the example, can you tell the type of `result`? If the API documentation is examined, the `mapToDouble` method returns a `DoubleStream`. The `sum` method for `DoubleStream` returns a `double`. Therefore, the result variable must be a `double`.

Output

```
== Total CO Bonus Details ==
Name: Joe Bailey Bonus paid: $7,200.00
Name: Phil Smith Bonus paid: $6,600.00
Name: Betty Jones Bonus paid: $8,400.00
Total Bonuses paid: $22,200.00
```

Average

The `average` method returns the average of a list of values passed from a stream. The `avg` method is a terminal operation.

A08CalcAvg.java

```
    System.out.println("\n== Average CO Bonus Details ==");

    result = eList.stream()
      .filter(e -> e.getRole().equals(Role.EXECUTIVE))
      .filter(e -> e.getState().equals("CO"))
      .peek(e -> System.out.print("Name: " + e.getGivenName()
          + " " + e.getSurName() + " "))
      .mapToDouble(e -> e.getSalary() * Bonus.byRole(e.getRole()))
      .peek(d -> System.out.printf("Bonus paid: $%,6.2f %n", d))
      .average();

      if (result.isPresent()){
        System.out.printf("Average Bonuses paid: $%,6.2f %n",
          result.getAsDouble());
      }
  }
```

Once again, the return type for `avg` can be inferred from the code shown in this example. Note the check for `isPresent()` in the if statement and the call to `getAsDouble()`. In this case an `OptionalDouble` is returned.

Output

```
== Average CO Bonus Details ==
Name: Joe Bailey Bonus paid: $7,200.00
Name: Phil Smith Bonus paid: $6,600.00
```

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

```
Name: Betty Jones Bonus paid: $8,400.00
Average Bonuses paid: $7,400.00
```

Sorted

The sorted method can be used to sort stream elements based on their natural order. This is an intermediate operation.

A09SortBonus.java

```
public class A09SortBonus {
  public static void main(String[] args) {
    List<Employee> eList = Employee.createShortList();

    System.out.println("\n== CO Bonus Details ==");

    eList.stream()
      .filter(e -> e.getRole().equals(Role.EXECUTIVE))
      .filter(e -> e.getState().equals("CO"))
      .mapToDouble(e -> e.getSalary() * Bonus.byRole(e.getRole()))
      .sorted()
      .forEach(d -> System.out.printf("Bonus paid: $%,6.2f %n", d));
```

In this example, the bonus is computed and those values are used to sort the results. So a list for `double` values is sorted and printed out.

Output

```
== CO Bonus Details ==
Bonus paid: $6,600.00
Bonus paid: $7,200.00
Bonus paid: $8,400.00
```

Sorted with Comparator

The `sorted` method can also take a `Comparator` as a parameter. Combined with the `comparing` method, the `Comparator` class provides a great deal of flexibility when sorting a stream.

A10SortComparator.java

```
11 public class A10SortComparator {
12   public static void main(String[] args) {
13     List<Employee> eList = Employee.createShortList();
14
15     System.out.println("\n== CO Bonus Details Comparator ==");
16
17     eList.stream()
18       .filter(e -> e.getRole().equals(Role.EXECUTIVE))
19       .filter(e -> e.getState().equals("CO"))
20       .sorted(Comparator.comparing(Employee::getSurName))
21       .forEach(Employee::printSummary);
```

In this example, notice on line 20 that a method reference is passed to the `comparing` method. In this case, the stream is sorted by surname. However, clearly the implication is any of the `get` methods from the `Employee` class could be passed to this method. So with one simple expression, a stream can be sorted by any available field.

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Output

```
== CO Bonus Details Comparator ==
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
```

Reversed

The `reversed` method can be appended to the `comparing` method thus reversing the sort order of the elements in the stream. The example and output demonstrate this using surname.

A10SortComparator.java

```
    System.out.println("\n== CO Bonus Details Reversed ==");

    eList.stream()
      .filter(e -> e.getRole().equals(Role.EXECUTIVE))
      .filter(e -> e.getState().equals("CO"))
      .sorted(Comparator.comparing(Employee::getSurName).reversed())
      .forEach(Employee::printSummary);
```

Output

```
== CO Bonus Details Reversed ==
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
Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary: $120,000.00
```

Two Level Sort

In this example, the `thenComparing` method has been added to the `comparing` method. This allows you to do a multilevel sort on the elements in the stream. The `thenComparing` method takes a `Comparator` as a parameter just like the `comparing` method.

A10SortComparator.java

```
    System.out.println("\n== Two Level Sort, Dept then Surname ==");

    eList.stream()
      .sorted(
        Comparator.comparing(Employee::getDept)
          .thenComparing(Employee::getSurName))
      .forEach(Employee::printSummary);
```

In the example, the stream is sorted by department and then by surname. The output is as follows.

Output

```
== Two Level Sort, Dept then Surname ==
Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary: $120,000.00
Name: Bob Baker Role: STAFF Dept: Eng St: KS Salary: $40,000.00
Name: John Doe Role: MANAGER Dept: Eng St: KS Salary: $65,000.00
Name: James Johnson Role: MANAGER Dept: Eng St: MA Salary: $85,000.00
Name: Phil Smith Role: EXECUTIVE Dept: HR St: CO Salary: $110,000.00
Name: John Adams Role: MANAGER Dept: Sales St: MA Salary: $90,000.00
Name: Jane Doe Role: STAFF Dept: Sales St: KS Salary: $45,000.00
Name: Betty Jones Role: EXECUTIVE Dept: Sales St: CO Salary: $140,000.00
```

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Collect

The `collect` method allows you to save the results of all the filtering, mapping, and sorting that takes place in a pipeline. Notice how the `collect` method is called. It takes a `Collectors` class as a parameter. The `Collectors` class provides a number of ways to return the elements left in a pipeline.

A11Collect.java

```
public class A11Collect {

    public static void main(String[] args) {

        List<Employee> eList = Employee.createShortList();

        List<Employee> nList = new ArrayList<>();

        // Collect CO Executives
        nList = eList.stream()
            .filter(e -> e.getRole().equals(Role.EXECUTIVE))
            .filter(e -> e.getState().equals("CO"))
            .sorted(Comparator.comparing(Employee::getSurName))
            .collect(Collectors.toList());

        System.out.println("\n== CO Bonus Details ==");

        nList.stream()
            .forEach(Employee::printSummary);

    }

}
```

In this example, the `Collectors` class simply returns a new `List`, which consists of the elements selected by the filter methods. In addition to a `List`, a `Set` or a `Map` may be returned as well. Plus there are a number of other options to save the pipeline results. Below are the three `Employee` elements that match the filter criteria in sorted order.

Output

```
== CO Bonus Details ==
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
```

Collectors and Math

The `Collectors` class includes a number of math methods including `averagingDouble` and `summingDouble` along with other primitive versions.

A12CollectMath.java

```
public class A12CollectMath {

    public static void main(String[] args) {

        List<Employee> eList = Employee.createShortList();
```

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

```

        // Collect CO Executives
        double avgSalary = eList.stream()
            .filter(e -> e.getRole().equals(Role.EXECUTIVE))
            .filter(e -> e.getState().equals("CO"))
            .collect(
                Collectors.averagingDouble(Employee::getSalary));

        System.out.println("\n== CO Exec Avg Salary ==");
        System.out.printf("Average: $%,9.2f %n", avgSalary);

    }

}
```

In this example, an average salary is computed based on the filters provided. A double `primitive` value is returned.

Output

```
== CO Exec Avg Salary ==
Average: $123,333.33
```

Collectors and Joining

The `joining` method of the `Collectors` class allows you to join together elements returned from a stream.

A13CollectJoin.java

```
public class A13CollectJoin {

    public static void main(String[] args) {

        List<Employee> eList = Employee.createShortList();

        // Collect CO Executives
        String deptList = eList.stream()
            .map(Employee::getDept)
            .distinct()
            .collect(Collectors.joining(", "));

        System.out.println("\n== Dept List ==");
        System.out.println("Total: " + deptList);

    }

}
```

In this example, the values for department are extracted from the stream using a `map`. A call is made to the `distinct` method, which removes any duplicate values. The resulting values are joined together using the `joining` method. The output is shown in the following.

Output

```
== Dept List ==
Total: Eng, Sales, HR
```

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Collectors and Grouping

The groupingBy method of the Collectors class allows you to generate a Map based on the elements contained in a stream.

A14CollectGrouping.java

```
12 public class A14CollectGrouping {
13
14     public static void main(String[] args) {
15
16         List<Employee> eList = Employee.createShortList();
17
18         Map<String, List<Employee>> gMap = new HashMap<>();
19
20         // Collect CO Executives
21         gMap = eList.stream()
22             .collect(Collectors.groupingBy(Employee::getDept));
23
24         System.out.println("\n== Employees by Dept ==");
25         gMap.forEach((k,v) -> {
26             System.out.println("\nDept: " + k);
27             v.forEach(Employee::printSummary);
28         });
29
30     }
31
32 }
```

In this example, the groupingBy method is called with a method reference to getDept. This created a Map with the department names used as key and a list of elements that match that key become the value for the Map. Notice how the Map is specified on line 18. In addition, starting on line 25 the code iterates through the resulting Map. The output from the Map is shown in the following.

Output

```
== Employees by Dept ==

Dept: Sales
Name: Jane Doe Role: STAFF Dept: Sales St: KS Salary: $45,000.00
Name: John Adams Role: MANAGER Dept: Sales St: MA Salary: $90,000.00
Name: Betty Jones Role: EXECUTIVE Dept: Sales St: CO Salary: $140,000.00

Dept: HR
Name: Phil Smith Role: EXECUTIVE Dept: HR St: CO Salary: $110,000.00

Dept: Eng
Name: Bob Baker Role: STAFF Dept: Eng St: KS Salary: $40,000.00
Name: John Doe Role: MANAGER Dept: Eng St: KS Salary: $65,000.00
Name: James Johnson Role: MANAGER Dept: Eng St: MA Salary: $85,000.00
Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary: $120,000.00
```

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Collectors, Grouping, and Counting

Another version of the `groupingBy` function takes a `Function` and `Collector` as parameters and returns a `Map`. This example builds on the last and instead of returning matching elements, it counts them.

A15CollectCount.java

```
public class A15CollectCount {

    public static void main(String[] args) {

        List<Employee> eList = Employee.createShortList();

        Map<String, Long> gMap = new HashMap<>();

        // Collect CO Executives
        gMap = eList.stream()
            .collect(
                Collectors.groupingBy(
                    e -> e.getDept(), Collectors.counting()));

        System.out.println("\n== Employees by Dept ==");
        gMap.forEach((k,v) ->
            System.out.println("Dept: " + k + " Count: " + v)
        );

    }

}
```

Note how the method once again creates the `Map` based on department. But this time, `Collectors.counting` is used to return `long` values to the `Map`. The output from the `Map` is shown in the following.

Output

```
== Employees by Dept ==
Dept: Sales Count: 3
Dept: HR Count: 1
Dept: Eng Count: 4
```

Collectors and Partitioning

The `partitioningBy` method offers an interesting way to create a `Map`. The method takes a `Predicate` as an argument and creates a `Map` with two `Boolean` keys. One key is `true` and includes all the elements that met the true criteria of the Predicate. The other key, `false`, contains all the elements that resulted in false values as determined by the `Predicate`.

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A16CollectPartition.java

```java
public class A16CollectPartition {

    public static void main(String[] args) {

        List<Employee> eList = Employee.createShortList();

        Map<Boolean, List<Employee>> gMap = new HashMap<>();

        // Collect CO Executives
        gMap = eList.stream()
            .collect(
                Collectors.partitioningBy(
                    e -> e.getRole().equals(Role.EXECUTIVE)));

        System.out.println("\n== Employees by Dept ==");
        gMap.forEach((k,v) -> {
            System.out.println("\nGroup: " + k);
            v.forEach(Employee::printSummary);
        });

    }

}
```

This example creates a `Map` based on role. All executives will be in the `true` group, and all other employees will be in the `false` group. Here is a printout of the map.

Output

```
== Employees by Dept ==

Group: false
Name: Bob Baker Role: STAFF Dept: Eng St: KS Salary: $40,000.00
Name: Jane Doe Role: STAFF Dept: Sales St: KS Salary: $45,000.00
Name: John Doe Role: MANAGER Dept: Eng St: KS Salary: $65,000.00
Name: James Johnson Role: MANAGER Dept: Eng St: MA Salary: $85,000.00
Name: John Adams Role: MANAGER Dept: Sales St: MA Salary: $90,000.00

Group: true
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
```

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 10-1: Using Map and Peek

Overview

In this practice, use lambda expressions and the `stream` method along with the `map` and `peek` methods to print a report on all the Widget Pro sales in the state of California (CA).

Assumptions

You have completed the lecture portion of this course.

Tasks

1. Open the `SalesTxn10-01Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/10-LambdaOperations /practices/practice1`.
   - Select `SalesTxn10-01Prac` and click Open Project.
2. Review the code for the `SalesTxn` class. Note that enumerations exist for `BuyerClass`, `State`, and `TaxRate`.
3. Modify the `MapTest` class to create a sales tax report.
   a. Filter the transactions for the following.
      - Transactions from the state of CA: `t.getState().equals(State.CA)`
      - Transactions for the Widget Pro product: `t.getProduct().equals("Widget Pro")`

   b. Use the `map` method to calculate the sales tax. The calculation is as follows: `t.getTransactionTotal() * TaxRate.byState(t.getState())`

   c. Print a report similar to the following:

```
=== Widget Pro Sales Tax in CA ===
Txn tax: $36,000.00
Txn tax: $180,000.00
```

Note: To get the comma-separated currency, use something like this:

`System.out.printf("Txn tax: $%,9.2f%n", amt)`

4. Copy the main method from the `MapTest` class to the `PeekTest` class.
5. Update your code to print more detailed information about the matching transaction using the `peek` method. A `Consumer` is provided for you that adds the following:
   - Transaction ID
   - Buyer
   - Total Transaction amount
   - Sales tax amount
6. The output should look similar to the following:

```
=== Widget Pro Sales Tax in CA ===
Id: 12 Buyer: Acme Electronics Txn amt: $400,000.00 Txn tax:
$36,000.00
Id: 13 Buyer: Radio Hut Txn amt: $2,000,000.00 Txn tax: $180,000.00
```

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 10-2: FindFirst and Lazy Operations

Overview

In this practice, compare a `forEach` loop to a `findFirst` short-circuit terminal operation and see how the two differ in number of operations.

The following Consumer lambda expressions have been written for you to save you from some typing. The variables are: `quantReport`, `streamStart`, `stateSearch`, and `productSearch`.

Assumptions

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

Tasks

1. Open the `SalesTxn10-02Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/10-LambdaOperations /practices/practice2`.
   - Select `SalesTxn10-02Prac` and click Open Project.
2. Edit the `LazyTest` class to perform the steps in this practice.
3. Using `stream` and lambda expressions print out a list of transactions that meet the following criteria.
   a. Create a filter to select all "Widget Pro" sales.
   b. Create a filter to select transactions in the state of Colorado (CO).
   c. Iterate through the matching transactions and print a report similar to the following using `quantReport` in the `forEach`.

```
=== Widget Pro Quantity in CO ===
Seller: Betty Jones-- Buyer: Radio Hut -- Quantity:    20,000
Seller: Dave Smith-- Buyer: PriceCo -- Quantity:     6,000
Seller: Betty Jones-- Buyer: Best Deals -- Quantity:    20,000
```

4. Perform the same search as in the previous step. This time use the `peek` method to display each step in the process. Put a `peek` method call in the following places.
   a. Add a `peek` method after the `stream()` method that uses the `streamStart` as its parameter.
   b. Add a `peek` method after the `filter` for state that uses `stateSearch` as its parameter.
   c. Add a `peek` method after the `filter` for product that uses `productSearch` as its parameter.
   d. Print the final result using `forEach` as in the previous step.
   e. The output should look similar to the following.

```
=== Widget Pro Quantity in CO ===
Stream start: Jane Doe ID: 11
Stream start: Jane Doe ID: 12
Stream start: Jane Doe ID: 13
Stream start: John Smith ID: 14
Stream start: Betty Jones ID: 15
```

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

```
State Search: Betty Jones St: CO
Product Search
Seller: Betty Jones-- Buyer: Radio Hut -- Quantity:    20,000
Stream start: Betty Jones ID: 16
State Search: Betty Jones St: CO
Stream start: Dave Smith ID: 17
State Search: Dave Smith St: CO
Product Search
Seller: Dave Smith-- Buyer: PriceCo -- Quantity:     6,000
Stream start: Dave Smith ID: 18
State Search: Dave Smith St: CO
Stream start: Betty Jones ID: 19
State Search: Betty Jones St: CO
Product Search
Seller: Betty Jones-- Buyer: Best Deals -- Quantity:    20,000
Stream start: John Adams ID: 20
Stream start: John Adams ID: 21
Stream start: Samuel Adams ID: 22
Stream start: Samuel Adams ID: 23
```

5. Copy the code from the previous step so you can modify it.
6. Replace the `forEach` with a `findFirst` method.
7. Add the following code:
   a. Use an `Optional<SalesTxn>` named `ft` to store the result.
   b. Write an `if` statement to check to see if `ft.isPresent()`.
   c. If a value is returned, call the `accept` method of `quantReport` to display the result.
   d. Your output should look similar to the following:

```
=== Widget Pro Quantity in CO (FindFirst)===
Stream start: Jane Doe ID: 11
Stream start: Jane Doe ID: 12
Stream start: Jane Doe ID: 13
Stream start: John Smith ID: 14
Stream start: Betty Jones ID: 15
State Search: Betty Jones St: CO
Product Search
Seller: Betty Jones-- Buyer: Radio Hut -- Quantity:    20,000
```

Take a moment to consider the difference between terminal and short-circuit terminal operations.

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 10-3: Analyze Transactions with Stream Methods

Overview

In this practice, count the number of transactions and determine the min and max values in the collection for transactions involving Radio Hut.

Assumptions

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

Tasks

1. Open the `SalesTxn10-03Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/10-LambdaOperations /practices/practice3`.
   - Select `SalesTxn10-03Prac` and click Open Project.
2. Edit the `RadioHutTest` class to perform the steps in this practice.
3. Using `stream` and lambda expressions print out all the transactions involving Radio Hut.
   a. Use a `filter` to select all "Radio Hut" transactions.
   b. Use the `radioReport` variable to print the matching transactions.
   c. Your output should look similar to the following:

```
=== Radio Hut Transactions ===
ID: 13  Seller: Jane Doe-- Buyer: Radio Hut -- State: CA -- Amt:
$2,000,000
ID: 15  Seller: Betty Jones-- Buyer: Radio Hut -- State: CO -- Amt:
$  800,000
ID: 23  Seller: Samuel Adams-- Buyer: Radio Hut -- State: MA -- Amt:
$1,040,000
```

4. Use `stream`, `filter`, and lambda expressions to calculate and print out the total number of transactions involving Radio Hut. (**Hint:** Use the `count` method.)
5. Use `stream` and lambda expressions to calculate and print out the largest transaction based on the total transaction amount involving Radio Hut. Use the `max` function with a `Comparator`, for example:

```
.max(Comparator.comparing(SalesTxn::getTransactionTotal))
```

6. Using `stream` and lambda expressions calculate and print out the smallest transaction based on the total transaction amount involving Radio Hut. Use the `min` method in a manner similar to the previous method.

   Hint: Remember to check the API documentation for the return types for the specified methods.

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

7. When complete, your output should look similar to the following.

```
=== Radio Hut Transactions ===
ID: 13  Seller: Jane Doe-- Buyer: Radio Hut -- State: CA -- Amt: $2,000,000
ID: 15  Seller: Betty Jones-- Buyer: Radio Hut -- State: CO -- Amt: $  800,000
ID: 23  Seller: Samuel Adams-- Buyer: Radio Hut -- State: MA -- Amt: $1,040,000
Total Transactions: 3
=== Radio Hut Largest ===
ID: 13  Seller: Jane Doe-- Buyer: Radio Hut -- State: CA -- Amt: $2,000,000
=== Radio Hut Smallest ===
ID: 15  Seller: Betty Jones-- Buyer: Radio Hut -- State: CO -- Amt: $  800,000
```

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 10-4: Perform Calculations with Primitive Streams

Overview

In this practice, calculate the sales totals and average units sold from the collection of sales transactions.

Assumptions

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

Tasks

1. Open the `SalesTxn10-04Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/10-LambdaOperations /practices/practice4`.
   - Select `SalesTxn10-04Prac` and click Open Project.
2. Edit the `CalcTest` class to perform the steps in this practice.
3. Calculate the total sales for "Radio Hut", "PriceCo", and "Best Deals" and print the results.
   - For example, filter Radio Hut with a lambda like this:
     `t -> t.getBuyerName().equals("Radio Hut")`
   - For example, get the transaction total with:
     `.mapToDouble( t -> t.getTransactionTotal())`
4. Calculate the average number of units sold for the "Widget" and "Widget Pro" products and print the results.
   - For example, the Widget Pro code looks like the following:

```
.filter(t -> t.getProduct().equals("Widget Pro"))
.mapToDouble( t-> t.getUnitCount())
```

   Hint: Be mindful of the method return types. Use to the API doc to ensure you are using the correct methods and classes to create and store results.

5. 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
=== Average Unit Count ===
Widget Pro Avg:    21,143
Widget Avg:    12,400
```

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 10-5: Sort Transactions with Comparator

Overview

In this practice, sort transactions using the `Comparator` class, the `comparing` method, and the `sorted` method.

Assumptions

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

Tasks

1. Open the `SalesTxn10-05Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/10-LambdaOperations /practices/practice5`.
   - Select `SalesTxn10-05Prac` and click Open Project.
2. Edit the `SortTest` class to perform the steps in this practice.
3. Use streams and lambda expressions to print out all the PriceCo transactions by transaction total in ascending order.
   - The sorted method should look something like this:

   ```
   .sorted(Comparator.comparing(SalesTxn::getTransactionTotal))
   ```
   - Use the `transReport` variable to print the results.
4. Use the same data from the previous step to print out the PriceCo transactions in descending order.
5. Print out all the transactions sorted using the following sort keys.
   - Buyer name
   - Sales person
   - Transaction total
6. When complete, the output should look similar to the following:

```
=== PriceCo Transactions ===
Id: 17 Seller: Dave Smith Buyer: PriceCo Amt: $240,000.00
Id: 20 Seller: John Adams Buyer: PriceCo Amt: $280,000.00
Id: 18 Seller: Dave Smith Buyer: PriceCo Amt: $300,000.00
Id: 21 Seller: John Adams Buyer: PriceCo Amt: $640,000.00

=== PriceCo Transactions Reversed ===
Id: 21 Seller: John Adams Buyer: PriceCo Amt: $640,000.00
Id: 18 Seller: Dave Smith Buyer: PriceCo Amt: $300,000.00
Id: 20 Seller: John Adams Buyer: PriceCo Amt: $280,000.00
Id: 17 Seller: Dave Smith Buyer: PriceCo Amt: $240,000.00

=== Triple Sort Transactions ===
Id: 11 Seller: Jane Doe Buyer: Acme Electronics Amt: $60,000.00
Id: 12 Seller: Jane Doe Buyer: Acme Electronics Amt: $400,000.00
Id: 16 Seller: Betty Jones Buyer: Best Deals Amt: $500,000.00
Id: 19 Seller: Betty Jones Buyer: Best Deals Amt: $800,000.00
Id: 14 Seller: John Smith Buyer: Great Deals Amt: $100,000.00
```

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

```
Id: 22 Seller: Samuel Adams Buyer: Mom and Pops Amt: $60,000.00
Id: 17 Seller: Dave Smith Buyer: PriceCo Amt: $240,000.00
Id: 18 Seller: Dave Smith Buyer: PriceCo Amt: $300,000.00
Id: 20 Seller: John Adams Buyer: PriceCo Amt: $280,000.00
Id: 21 Seller: John Adams Buyer: PriceCo Amt: $640,000.00
Id: 15 Seller: Betty Jones Buyer: Radio Hut Amt: $800,000.00
Id: 13 Seller: Jane Doe Buyer: Radio Hut Amt: $2,000,000.00
Id: 23 Seller: Samuel Adams Buyer: Radio Hut Amt: $1,040,000.00
```

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 10-6: Collect Results with Streams

Overview

In this practice, use the `collect` method to store the results from a `stream` in a new list.

Assumptions

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

Tasks

1. Open the `SalesTxn10-06Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/10-LambdaOperations /practices/practice6`.
   - Select `SalesTxn10-06Prac` and click Open Project.
2. Edit the `CollectTest` class to perform the steps in this practice.
3. Filter the transaction list to only include transactions greater than $300,000 sorted in ascending order.
4. Store the results in a new list using the `collect` method. For example:

```
.collect(Collectors.toList())
```

5. Print out the transactions in the new list. The output should look similar to the following:

```
=== Transactions over $300k ===
Id: 12 Seller: Jane Doe Buyer: Acme Electronics Amt: $400,000.00
Id: 16 Seller: Betty Jones Buyer: Best Deals Amt: $500,000.00
Id: 21 Seller: John Adams Buyer: PriceCo Amt: $640,000.00
Id: 15 Seller: Betty Jones Buyer: Radio Hut Amt: $800,000.00
Id: 19 Seller: Betty Jones Buyer: Best Deals Amt: $800,000.00
Id: 23 Seller: Samuel Adams Buyer: Radio Hut Amt: $1,040,000.00
Id: 13 Seller: Jane Doe Buyer: Radio Hut Amt: $2,000,000.00
```

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 10-7: Join Data with Streams

Overview

In this practice, use the `joining` method to combine data returned from a stream.

Assumptions

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

Tasks

1. Open the `SalesTxn10-07Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/10-LambdaOperations /practices/practice7`.
   - Select `SalesTxn10-07Prac` and click Open Project.
2. Edit the `JoinTest` class to perform the steps in this practice.
3. Get a list of unique buyer names in a sorted order. Follow these steps to accomplish the task:
   a. Use `map` to get all the buyer names.
   b. Use `distinct` to remove duplicates.
   c. Use `sorted` to sort the names.
   d. Use `joining` to join the names together in the output you see in the following.
4. When complete, your output should look similar to the following:

```
=== Sorted Buyer's List ===
Buyer list: Acme Electronics, Best Deals, Great Deals, Mom and Pops,
PriceCo, Radio Hut
```

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 10-8: Group Data with Streams

Overview

In this practice, create a `Map` of transaction data using the `groupingBy` method from the `Collectors` class.

Assumptions

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

Tasks

1. Open the `SalesTxn10-08Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/10-LambdaOperations /practices/practice8`.
   - Select `SalesTxn10-08Prac` and click Open Project.
2. Edit the `GroupTest` class to perform the steps in this practice.
3. Populate the `Map` by using the stream `collect` method to return the list elements grouped by buyer name.
   a. Use `Collectors.groupingBy()` to group the results.
   b. Use `SalesTxn::getBuyerName` to determine what to group by.
4. Print out the result.
5. Use the `printSummary` method of the `SalesTxn` class to print individual transactions.

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

6. Your output should look similar to the following:

```
=== Transactions Grouped by Buyer ===

Buyer: PriceCo
ID: 17 - Seller: Dave Smith - Buyer: PriceCo - Product: Widget Pro - ST: CO - Amt:
240000.0 - Date: 2013-03-20
ID: 18 - Seller: Dave Smith - Buyer: PriceCo - Product: Widget - ST: CO - Amt:
300000.0 - Date: 2013-03-30
ID: 20 - Seller: John Adams - Buyer: PriceCo - Product: Widget - ST: MA - Amt:
280000.0 - Date: 2013-07-14
ID: 21 - Seller: John Adams - Buyer: PriceCo - Product: Widget Pro - ST: MA - Amt:
640000.0 - Date: 2013-10-06

Buyer: Acme Electronics
ID: 11 - Seller: Jane Doe - Buyer: Acme Electronics - Product: Widgets - ST: CA -
Amt: 60000.0 - Date: 2013-01-25
ID: 12 - Seller: Jane Doe - Buyer: Acme Electronics - Product: Widget Pro - ST: CA
- Amt: 400000.0 - Date: 2013-04-05

Buyer: Radio Hut
ID: 13 - Seller: Jane Doe - Buyer: Radio Hut - Product: Widget Pro - ST: CA - Amt:
2000000.0 - Date: 2013-10-03
ID: 15 - Seller: Betty Jones - Buyer: Radio Hut - Product: Widget Pro - ST: CO -
Amt: 800000.0 - Date: 2013-02-04
ID: 23 - Seller: Samuel Adams - Buyer: Radio Hut - Product: Widget Pro - ST: MA -
Amt: 1040000.0 - Date: 2013-12-08

Buyer: Mom and Pops
ID: 22 - Seller: Samuel Adams - Buyer: Mom and Pops - Product: Widget - ST: MA -
Amt: 60000.0 - Date: 2013-10-02

Buyer: Best Deals
ID: 16 - Seller: Betty Jones - Buyer: Best Deals - Product: Widget - ST: CO - Amt:
500000.0 - Date: 2013-03-21
ID: 19 - Seller: Betty Jones - Buyer: Best Deals - Product: Widget Pro - ST: CO -
Amt: 800000.0 - Date: 2013-07-12

Buyer: Great Deals
ID: 14 - Seller: John Smith - Buyer: Great Deals - Product: Widget - ST: CA - Amt:
100000.0 - Date: 2013-10-10
```

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.