Java SE 8 New Features

Table of Contents

Practices for Lesson 1: Interfaces and Lambda Expressions

Chapter 1

Practices for Lesson 1: Overview

Practices Overview

In these practices, you will use Java interfaces and lambda expressions.

.

.

Practice 1-3: Summary Level: Write Lambda Expressions

Overview

In this practice, write additional lambda expressions for the `StringAnalzyer` application.

Assumptions

You have reviewed the lambda expressions section of this lesson.

Summary

Use the `StringAnalyzer` project from the lecture to create 3 additional lambda expressions.

Tasks

1. Open the `LambdaBasics06-03Prac` project.

a. Select File > Open Project.

b. Browse to `D:/labs/06-interfaces/practices/practice3`.

c. Select `LambdaBasics06-03Prac` and click Open Project.

2. Expand the project directories.

3. Open the `LambdaTest.java` file.

4. Write a lambda expression that displays strings that end with the search string.

5. Write a lambda expression that displays strings that contain the search string and are 5 characters or less in length.

6. Write a lambda expression that displays strings that contain the search string and are greater than 5 characters in length.

7. Run the project. The output should be as follows:

```
Searching for: to
==Contains==
Match: tomorrow
Match: toto
Match: to
Match: timbukto
==Starts With==
Match: tomorrow
```

```
Match: toto
Match: to
==Equals==
Match: to
==Ends With==
Match: toto
Match: to
Match: timbukto
==Less than 5==
Match: toto
Match: to
==Greater than 5==
Match: tomorrow
Match: timbukto
```

.

Practice 1-3: Detailed Level: Write Lambda Expressions

Overview

In this practice, write additional lambda expressions for the `StringAnalzyer` application.

Assumptions

You have reviewed the lambda expressions section of this lesson.

Summary

Use the `StringAnalyzer` project from the lecture to create three additional lambda expressions.

Tasks

1. Open the `LambdaBasics06-03Prac` project.

a. Select File > Open Project.

b. Browse to `D:/labs/06-interfaces/practices/practice3`.

c. Select `LambdaBasics06-03Prac` and click Open Project.

2. Expand the project directories.

3. Open the `LambdaTest.java` file

4. Write a lambda expression that displays strings that end with the search string.

```
(t,s) -> t.endsWith(s));
```

5. Write a lambda expression that displays strings that contain the search string and are 5 characters or less in length.

```
(t,s) -> t.contains(s) && t.length() < 5);
```

6. Write a lambda expression that displays strings that contain the search string and are greater than five characters in length.

```
(t,s) -> t.contains(s) && t.length() > 5);
```

.

7. Run the project. The output should be as follows:

```
Searching for: to
==Contains==
Match: tomorrow
Match: toto
Match: to
Match: timbukto
==Starts With==
Match: tomorrow
Match: toto
Match: to
==Equals==
Match: to
==Ends With==
Match: toto
Match: to
Match: timbukto
==Less than 5==
Match: toto
Match: to
==Greater than 5==
Match: tomorrow
Match: timbukto
```

.

Collections Streams, and

Filters

Practices for Lesson 2: Overview

Practice Overview

In these practices, you use lambda expressions to improve an application.

The RoboCall App

The RoboCall app is an application for automating the communication with groups of people. The app can contact individuals by phone, email, or regular mail. In this example, the app will be used to contact three groups of people.

- Drivers: Persons over the age of 16
- Draftees: Male persons between the ages of 18 and 25
- Pilots (specifically commercial pilots): Persons between the ages of 23 and 65

Person

The `Person` class creates the master list of persons you want to contact. The class uses the builder pattern to create new object. The following are some key parts of the class.

First, private fields for each Person are as follows:

Person.java

```
9 public class Person {
10 private String givenName;
11 private String surName;
12 private int age;
13 private Gender gender;
14 private String eMail;
15 private String phone;
16 private String address;
17 private String city;
18 private String state;
19 private String code;
20
```

So these will be the fields that our application can search.

A static method is used to create a list of sample users. The code looks something like this:

Person.java

```
167 public static List<Person> createShortList(){
168 List<Person> people = new ArrayList<>();
169
170 people.add(
171 new Person.Builder()
172 .givenName("Bob")
173 .surName("Baker")
174 .age(21)
175 .gender(Gender.MALE)
176 .email("bob.baker@example.com")
177 .phoneNumber("201-121-4678")
178 .address("44 4th St")
179 .city("Smallville")
180 .state("KS")
181 .code("12333")
182 .build()
183 );
184
```

forEach

All collections have a new `forEach` method.

RoboCallTest06.java

```
9 public class RoboCallTest06 {
10
11 public static void main(String[] args){
12
13 List<Person> pl = Person.createShortList();
14
15 System.out.println("\n=== Print List ===");
16 pl.forEach(p -> System.out.println(p));
17
18 }
19 }
```

Notice that the `forEach` takes a method reference or a lambda expression as a parameter. In the example, the `toString` method is called to print out each `Person` object. Some form of

expression is needed to specify the output.

Stream and Filter

The following example shows how `stream()` and `filter()` methods are used with a collection in the RoboCall app.

RoboCallTest07.java

```
10 public class RoboCallTest07 {
11
12 public static void main(String[] args){
13
14 List<Person> pl = Person.createShortList();
15 RoboCall05 robo = new RoboCall05();
16
17 System.out.println("\n=== Calling all Drivers Lambda ===");
18 pl.stream()
19 .filter(p -> p.getAge() >= 23 && p.getAge() <= 65)
20 .forEach(p -> robo.roboCall(p));
21
22 }
23 }
```

The `stream` method creates a pipeline of immutable `Person` elements and access to methods
that can perform actions on those elements. The `filter` method takes a lambda expression as
a parameter and filters on the logical expression provide. This indicates that a `Predicate` is
.
the target type of the filter. The elements that meet the filter criteria are passed to the `forEach` method, which does a `roboCall` on matching elements.

The following example is functionally equivalent to the last. But in the case, the lambda expression is assigned to a variable, which is then passed to the stream and filter.

RoboCallTest08.java

```
10 public class RoboCallTest08 {
11
12 public static void main(String[] args){
13
```

```
14 List<Person> pl = Person.createShortList();
15 RoboCall05 robo = new RoboCall05();
16
17 // Predicates
18 Predicate<Person> allPilots =
19 p -> p.getAge() >= 23 && p.getAge() <= 65;
20
21 System.out.println("\n=== Calling all Drivers Variable ===");
22 pl.stream().filter(allPilots)
23 .forEach(p -> robo.roboCall(p));
24 }
25 }
```

Method References

In cases where a lambda expression just calls an instance method, a method reference can be used instead.

A03aMethodReference.java

```
9 public class A03aMethodReference {
10
11 public static void main(String[] args) {
12
13 List<SalesTxn> tList = SalesTxn.createTxnList();
14
15 System.out.println("\n== CA Transations Lambda ==");
16 tList.stream()
17 .filter(t -> t.getState().equals(State.CA))
18 .forEach(t -> t.printSummary());
19
20 tList.stream()
21 .filter(t -> t.getState().equals(State.CA))
22 .forEach(SalesTxn::printSummary);
23 }
24 }
```

So lines 18 and 22 are essentially equivalent. Method reference syntax uses the class name followed by "::" and then the method name.

Chaining and Pipelines

The final example compares a compound lambda statement with a chained version using multiple `filter` methods.

A04IterationTest.java

```
9 public class A04IterationTest {
10
11 public static void main(String[] args) {
12
13 List<SalesTxn> tList = SalesTxn.createTxnList();
14
15 System.out.println("\n== CA Transations for ACME ==");
16 tList.stream()
17 .filter(t -> t.getState().equals(State.CA) &&
18 t.getBuyer().getName().equals("Acme Electronics"))
19 .forEach(SalesTxn::printSummary);
20
21 tList.stream()
22 .filter(t -> t.getState().equals(State.CA))
23 .filter(t -> t.getBuyerName()
24 .equals("Acme Electronics"))
25 .forEach(SalesTxn::printSummary);
26 }
27 }
```

The two examples are essentially equivalent. The second example demonstrates how methods can be chained to possibly make the code a little easier to read. Both are examples of pipelines created by the `stream` method.

.

Practice 2-1: Update RoboCall to use Streams

Overview

In this practice, you have been given an old email mailing list program named `RoboMail`. It is used to send emails or text messages to employees at your company. Refactor `RoboMail` so that it uses lambda expressions instead of anonymous inner classes.

Assumptions

You have completed the lecture and reviewed the overview for this practice.

Tasks

1. Open the `EmployeeSearch08-01Prac` project.

 Select File > Open Project.

 Browse to `D:/labs/08-CollectionsStreamsFilters /practices/practice1`.

 Select `EmployeeSearch08-01Prac` and click Open Project.

2. Open the `RoboMail01.java` file and remove the `mail` and `text` methods. They are no longer needed since a `stream` will be used to filter the employees and a `forEach` will call the required communication task.

3. Open the `RoboMailTest01.java` file and review the code there.

4. Update `RoboMailTest01.java` to use `stream`, `filter`, and `forEach` to perform the mailing and texting tasks of the previous program.

5. Your program should continue to perform the following tasks to the following groups.

 Email all sales executives using `stream`, `filter`, and `forEach`.

 Text all sales executives using `stream`, `filter`, and `forEach`.

 Email all sales employees older than 50 using `stream`, `filter`, and `forEach`.

 Text all sales employees older than 50 using `stream`, `filter`, and `forEach`.

6. To mail or text a group in the `forEach` method, use a lambda expression for each task.

a. Mail example: `p -> robo.roboMail(p)`

b. Text example: `p -> robo.roboText(p)`

Your output should look similar to the following:

```
==== RoboMail 01
=== Sales Execs
Emailing: Betty Jones age 65 at betty.jones@example.com
Texting: Betty Jones age 65 at 211-33-1234
=== All Sales
Emailing: John Adams age 52 at john.adams@example.com
Emailing: Betty Jones age 65 at betty.jones@example.com
Texting: John Adams age 52 at 112-111-1111
Texting: Betty Jones age 65 at 211-33-1234
```

.

Practice 2-2: Mail Sales Executives using Method Chaining

Overview

In this practice, continue to work with the RoboMail app from the previous lesson.

Assumptions

You have completed the lecture and completed the previous practice.

Tasks

1. Open the `EmployeeSearch08-02Prac` project.

- Select File > Open Project.
- Browse to `D:/labs/08-CollectionsStreamsFilters/practices/practice2`.
- Select `EmployeeSearch08-02Prac` and click Open Project.

2. Open the `RoboMailTest01.java` file and review the code there.

3. Update the `RoboMailTest01.java` file to mail all sales executives. Use two filter methods to select the recipients of the mail.

The output from the program should look similar to the following:

```
==== RoboMail 01
=== Sales Execs
Emailing: Betty Jones age 65 at betty.jones@example.com
```

Practice 2-3: Mail Sales Employees over 50 Using Method Chaining

Overview

In this practice, continue to work with the RoboMail app from the previous lesson.

Assumptions

You have completed the lecture and completed the previous practice.

Tasks

1. Open the `EmployeeSearch08-03Prac` project.

- Select File > Open Project.
- Browse to `D:/labs/08-CollectionsStreamsFilters/practices/practice3`.
- Select `EmployeeSearch08-03Prac` and click Open Project.

2. Open the `RoboMailTest01.java` file and review the code there.

3. Update the `RoboMailTest01.java` file to mail all sales employees over 50. Use two filter methods to select the recipients of the mail.

The output from the program should look similar to the following:

```
==== RoboMail 01
=== All Sales 50+
Emailing: John Adams age 52 at john.adams@example.com
```

```
Emailing: Betty Jones age 65 at betty.jones@example.com
.
```

Practice 2-4: Mail Male Engineering Employees Under 65 Using Method Chaining

Overview

In this practice, continue to work with the RoboMail app from the previous lesson.

Assumptions

You have completed the lecture and completed the previous practice.

Tasks

1. Open the `EmployeeSearch08-04Prac` project.

- Select File > Open Project.
- Browse to `D:/labs/08-CollectionsStreamsFilters/practices/practice4`.
- Select `EmployeeSearch08-04Prac` and click Open Project.

2. Open the `RoboMailTest01.java` file and review the code there.

3. Update the `RoboMailTest01.java` file to mail all male engineering employees under 65.

Use three filter methods to select the recipients of the mail.

The output from the program should look similar to the following:

```
==== RoboMail 01
=== Male Eng Under 65
Emailing: James Johnson age 45 at james.johnson@example.com
Emailing: Joe Bailey age 62 at joebob.bailey@example.com
.
```

.

Lambda Built-in Functional Interfaces

Practices for Lesson 3: Overview

Practice Overview

In these practices, create lambda expressions using the built-in functional interfaces found in the `java.util.function` package.

The focus of this lesson and examples is to make you familiar with the built-in functional interfaces for use with lambda expressions. They are often used as parameters for method calls with streams. Familiarity with these interfaces makes working with streams much easier.

Predicate

The `Predicate` interface has already been covered in the last lesson. Essentially, it is a lambda expression that takes a generic type and returns a `boolean`.

A01Predicate.java

```
10 public class A01Predicate {
11
12 public static void main(String[] args){
13
14 List<SalesTxn> tList = SalesTxn.createTxnList();
15
16 Predicate<SalesTxn> massSales =
17 t -> t.getState().equals(State.MA);
18
19 System.out.println("\n== Sales - Stream");
20 tList.stream()
21 .filter(massSales)
22 .forEach(t -> t.printSummary());
23
24 System.out.println("\n== Sales - Method Call");
25 for(SalesTxn t:tList){
26 if (massSales.test(t)){
27 t.printSummary();
```

```
28 }
29 }
30 }
31 }
```

In the preceding code, the lambda expression is used in a `filter` for a stream. The second example also shows that the `test` method can be executed on any `SalesTxn` element using the functional interface that stores the `Predicate`.

To repeat, a `Predicate` takes in a generic type and returns a `boolean`.

.

Consumer

The `Consumer` interface specifies a generic type but returns nothing. Essentially, it is a `void` return type for lambdas. In the following example, the lambda expression specifies how a transaction should be printed.

A02Consumer.java

```
10 public class A02Consumer {
11
12 public static void main(String[] args){
13
14 List<SalesTxn> tList = SalesTxn.createTxnList();
15 SalesTxn first = tList.get(0);
16
17 Consumer<SalesTxn> buyerConsumer = t ->
18 System.out.println("Id: " + t.getTxnId()
19 + " Buyer: " + t.getBuyerName());
20
21 System.out.println("== Buyers - Lambda");
22 tList.stream().forEach(buyerConsumer);
23
24 System.out.println("== First Buyer - Method");
25 buyerConsumer.accept(first);
26 }
27 }
```

For the `forEach` method, the default argument is a `Consumer`. The lambda expression is basically just a print statement that is used in the two cases shown. In the second example, the

`accept` method is called along with a transaction. This prints the first transaction in the list. The key point here is that the `Consumer` takes a generic type and returns nothing. It is essentially a `void` return type for lambda expressions.

Function

The `Function` interface specifies two generic object types to be used in the expression. The first generic object is used in the lambda expression and the second is the return type from the lambda expression. The example uses a `SalesTxn` to return a `String`.

A03Function.java

```
10 public class A03Function {
11
12 public static void main(String[] args){
13
14 List<SalesTxn> tList = SalesTxn.createTxnList();
15 SalesTxn first = tList.get(0);
16
17 Function<SalesTxn, String> buyerFunction =
18 t -> t.getBuyerName();
19
20 System.out.println("\n== First Buyer");
21 System.out.println(buyerFunction.apply(first));
```

The `Function` has one method named `apply`. In this example, a `String` is returned to the print statement.

With a `Function` the key concept is that a `Function` takes in one type and returns another.

Supplier

The `Supplier` interface specifies one generic type, which is returned from the lambda expression. Nothing is passed in so this is similar to a `Factory`. The follow expression example creates and returns a `SalesTxn` and adds it to our existing list.

A04Supplier.java

```
13 public static void main(String[] args){
14
15 List<SalesTxn> tList = SalesTxn.createTxnList();
16 Supplier<SalesTxn> txnSupplier =
17 () -> new SalesTxn.Builder()
```

```
18 .txnId(101)
19 .salesPerson("John Adams")
20 .buyer(Buyer.getBuyerMap().get("PriceCo"))
21 .product("Widget")
22 .paymentType("Cash")
23 .unitPrice(20)
24 .unitCount(8000)
25 .txnDate(LocalDate.of(2013,11,10))
26 .city("Boston")
27 .state(State.MA)
28 .code("02108")
29 .build();
30
31 tList.add(txnSupplier.get());
32 System.out.println("\n== TList");
33 tList.stream().forEach(SalesTxn::printSummary);
34 }
```

Notice a `Supplier` has no input arguments, there is merely empty parentheses: `() ->`. The example uses a builder to create a new object. Notice Supplier has only one method `get`, which in this case returns a `SalesTxn`.

The key take away with a `Supplier` is that it has no input parameters but returns a generic type.

So that pretty much covers the basic function interfaces. However, there are a lot of variations.

.

Primitive Types - ToDoubleFunction and AutoBoxing

There are primitive versions of all the built-in lambda functional interfaces. The following code shows an example of the `ToDoubleFunction` interface.

A05PrimFunction.java

```
11 public class A05PrimFunction {
12
13 public static void main(String[] args){
14
15 List<SalesTxn> tList = SalesTxn.createTxnList();
16 SalesTxn first = tList.get(0);
```

```
17
18 ToDoubleFunction<SalesTxn> discountFunction =
19 t -> t.getTransactionTotal()
20 * t.getDiscountRate();
21
22 System.out.println("\n== Discount");
23 System.out.println(
24 discountFunction.applyAsDouble(first));
25
```

Remember a `Function` takes in one generic and return a different generic. However, the `ToDoubleFunction` interface has only one generic specified. That is because it takes a generic type as input and returns a `double`. Notice also that the method name for this functional interface is `applyAsDouble`. So to repeat, the `ToDoubleFunction` takes in a generic and returns a double. There are also `long` and `int` versions of this interface.

Why create these primitive variations? Consider this piece of code.

A05PrimFunction.java

```
26 // What's wrong here?
27 Function<SalesTxn, Double> taxFunction =
28 t -> t.getTransactionTotal() * t.getTaxRate();
29 double tax = taxFunction.apply(first); // What happerns here?
30 }
31 }
```

With object types, this would require the autoboxing and unboxing of primitive values. Not good for performance. These specialized primitive interfaces address this issue and allow for operations on primitive types.

Primitive Types -– DoubleFunction

What if you need to pass in a primitive to a lambda expression? Well, the `DoubleFunction` interface is a great example of that.

A06DoubleFunction.java

```
5 public class A06DoubleFunction {
6
7 public static void main(String[] args) {
8
9 A06DoubleFunction test = new A06DoubleFunction();
```

```
10
11 DoubleFunction<String> calc =
12 t -> String.valueOf(t * 3);
.
13
14 String result = calc.apply(20);
15 System.out.println("New value is: " + result);
16 }
17 }
```

Primitive interfaces like `DoubleFunction`, `IntFunction`, or `LongFunction` take a primitive as input and return a generic type. In this case, a double is passed to the lambda expression and a String is returned. Once again, this avoids any boxing issues.

Binary Intefaces – BiPredicate

A number of examples having the `Predicate` interface have been explored so far in this course. A `Predicate` takes a generic class and returns a `boolean`. But what if you want to compare two things? There is a binary specialization for that.

The `BiPredicate` interface allows two object types to be used in a lambda expression. Binary interfaces for the other main interface types are also available.

A07Binary.java

```
10 public class A07Binary {
11
12 public static void main(String[] args){
13
14 List<SalesTxn> tList = SalesTxn.createTxnList();
15 SalesTxn first = tList.get(0);
16 String testState = "CA";
17
18 BiPredicate<SalesTxn,String> stateBiPred =
19 (t, s) -> t.getState().equals(State.CA);
20
21 System.out.println("\n== First in CA?");
22 System.out.println(
23 stateBiPred.test(first, testState));
24 }
```

```
25 }
```

The example specifies a `SalesTxn` and a `String` as the generic types used in the lambda expression. Note that the types are specified with `t` and `s` and a `boolean` is still returned. It is

the same result as a `Predicate`, but with two input types.

UnaryOperator

The `Function` interface takes in one generic and returns a different generic. What if you want to return the same thing? Then the `UnaryOperator` interface is what you need.

A08Unary.java

```
10 public class A08Unary {
11
12 public static void main(String[] args){
13
14 List<SalesTxn> tList = SalesTxn.createTxnList();
15 SalesTxn first = tList.get(0);
16
17 UnaryOperator<String> unaryStr =
18 s -> s.toUpperCase();
.
19
20 System.out.println("== Upper Buyer");
21 System.out.println(
22 unaryStr.apply(first.getBuyerName()));
23 }
24 }
```

The example takes a `String` and returns an uppercase version of that `String`.

API Docs

As a reminder, it is difficult to remember all the variations of functional interfaces and what they do. Make liberal use of the API docs to remember your options or what is returned for the `java.util.function` package.

.

Practice 3-1: Create Consumer Lambda Expression

Overview

In this practice, create a `Consumer` lambda expression to print out employee data.

Note that `salary` and `startDate` fields were added to the Employee class. In addition, enumerations are included for `Bonus` and `VacAccrual`. The enums allow calculations for bonuses and vacation time.

Assumptions

You have completed the lecture portion of the course.

Tasks

1. Open the `EmployeeSearch09-01Prac` project.

- Select File > Open Project.
- Browse to `D:/labs/09-LambdaBuiltIns/practices/practice1`.
- Select `EmployeeSearch09-01Prac` and click Open Project.

2. Open the `Employee.java` file and become familiar with the code included in the file.
3. Open the `ConsumerTest.java` file and make the following updates.
4. Write a `Consumer` lambda expression to print data about the first employee in the list.

a. The data printed should be the following: `"Name: " + e.getSurName() + " Role: " + e.getRole() + " Salary: " + e.getSalary()`

5. Write a statement to execute the lambda expression on the `first` variable.
6. Your output should look similar to the following:

```
=== First Salary
Name: Baker Role: STAFF Salary: 40000.0
```

Practice 3-2: Create a Function Lambda Expression

Overview

In this practice, create a `ToDoubleFunction` lambda expression to calculate an employee bonus.

Assumptions

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

Tasks

1. Open the `EmployeeSearch09-02Prac` project.

- Select File > Open Project.
- Browse to `D:/labs/09-LambdaBuiltIns/practices/practice2`.
- Select `EmployeeSearch09-02Prac` and click Open Project.

2. Open the `Bonus.java` file and review the code included in the file.
3. Open the `FunctionTest.java` file and make the following updates.

4. Write a `ToDoubleFunction` lambda expression to calculate the bonus for the first employee in the list.

a. The bonus can be calculated as follows: `e.getSalary() * Bonus.byRole(e.getRole())`

5. Write a statement to execute the lambda expression on the `first` variable.

6. Your output should look similar to the following:

```
=== First Employee Bonus
Name: Bob Baker Role: STAFF Dept: ENG eMail: bob.baker@example.com
Salary: 40000.0
Bonus: 800.0
```

.

Practice 3-3: Create a Supplier Lambda Expression

Overview

In this practice, create a `Supplier` lambda expression to add a new employee to the employee
list.

Assumptions

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

Tasks

1. Open the `EmployeeSearch09-03Prac` project.

- Select File > Open Project.
- Browse to `D:/labs/09-LambdaBuiltIns/practices/practice3`.
- Select `EmployeeSearch09-03Prac` and click Open Project.

2. Open the `SupplierTest.java` file and make the following updates.

3. Write a `Supplier` lambda expression to add a new employee to the list. The employee data is as follows:

Given name: Jill
SurName: Doe
Age: 26
Gender: Gender.FEMALE
Role: Role.STAFF
Dept: Sales
StartDate: LocalDate.of(2012, 7, 14)

Salary: 45000

Email: jill.doe@example.com

PhoneNumber: 202-123-4678

Address: 33 3rd St

City: Smallville

State: KS

Code: 12333

Hint: Her data is almost exactly the same as her sister Jane and can be found in the `Employee.java` file.

4. Write a statement to add the new employee to the employee list.

.

5. Your output should look similar to the following after adding the new employee to the list:

```
=== Print employee list after
Name: Bob Baker Role: STAFF Dept: ENG eMail: bob.baker@example.com
Salary: 40000.0
Name: Jane Doe Role: STAFF Dept: Sales eMail: jane.doe@example.com
Salary: 45000.0
Name: John Doe Role: MANAGER Dept: Eng eMail: john.doe@example.com
Salary: 65000.0
Name: James Johnson Role: MANAGER Dept: Eng eMail:
james.johnson@example.com Salary: 85000.0
Name: John Adams Role: MANAGER Dept: Sales eMail:
john.adams@example.com Salary: 90000.0
Name: Joe Bailey Role: EXECUTIVE Dept: Eng eMail:
joebob.bailey@example.com Salary: 120000.0
Name: Phil Smith Role: EXECUTIVE Dept: HR eMail:
phil.smith@examp;e.com Salary: 110000.0
Name: Betty Jones Role: EXECUTIVE Dept: Sales eMail:
betty.jones@example.com Salary: 140000.0
Name: Jill Doe Role: STAFF Dept: Sales eMail: jill.doe@example.com
Salary: 45000.0
```

.

Practice 3-4: Create a BiPredicate Lambda Expression

Overview

In this practice, create a `BiPredicate` lambda expression to calculate an employee bonus.

Assumptions

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

Tasks

1. Open the `EmployeeSearch09-04Prac` project.

- Select File > Open Project.
- Browse to `D:/labs/09-LambdaBuiltIns/practices/practice4`.
- Select `EmployeeSearch09-04Prac` and click Open Project.

2. Open the `BiPredicateTest.java` file and make the following updates.

3. Write a `BiPredicate` lambda expression to compare a field in the employee class to a string.

a. The `searchState` variable should be compared to the state value in the employee element.

4. Write an expression to perform the logical test in the `for` loop.

5. Your output should look similar to the following:

```
=== Print matching list
Name: Bob Baker Role: STAFF Dept: ENG eMail: bob.baker@example.com
Salary: 40000.0
Name: Jane Doe Role: STAFF Dept: Sales eMail: jane.doe@example.com
Salary: 45000.0
Name: John Doe Role: MANAGER Dept: Eng eMail: john.doe@example.com
Salary: 65000.0
```

Practices for Lesson 4:

Lambda Operations

Practices for Lesson 4: 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

```
9 public class A01MapTest {
10
11 public static void main(String[] args) {
12
13 List<Employee> eList = Employee.createShortList();
14
15 System.out.println("\n== CO Bonuses ==");
16 eList.stream()
17 .filter(e -> e.getRole().equals(Role.EXECUTIVE))
18 .filter(e -> e.getState().equals("CO"))
19 .map(e -> e.getSalary() * Bonus.byRole(e.getRole()))
20 .forEach( s -> System.out.printf("Bonus paid: $%,6.2f %n", s));
21
```

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

.

```
15 System.out.println("\n== CO Bonuses ==");
16 eList.stream()
17 .filter(e -> e.getRole().equals(Role.EXECUTIVE))
18 .filter(e -> e.getState().equals("CO"))
19 .peek(e -> System.out.print("Name: "
20 + e.getGivenName() + " " + e.getSurName()))
21 .map(e -> e.getSalary() * Bonus.byRole(e.getRole()))
22 .forEach( s ->
23 System.out.printf(
24 " 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

```
10 public class A03FindFirst {
11
12 public static void main(String[] args) {
13
14 List<Employee> eList = Employee.createShortList();
15
16 System.out.println("\n== First CO Bonus ==");
17 Optional<Employee> result;
18
19 result = eList.stream()
```

```
20 .filter(e -> e.getRole().equals(Role.EXECUTIVE))
21 .filter(e -> e.getState().equals("CO"))
22 .findFirst();
23
24 if (result.isPresent()){
25 result.get().print();
26 }
27
28 }
```

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.

.

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

```
10 public class A04FindFirstLazy {
```

```
11
12 public static void main(String[] args) {
13
14 List<Employee> eList = Employee.createShortList();
15
16 System.out.println("\n== CO Bonuses ==");
17 eList.stream()
18 .peek(e -> System.out.println("Stream start"))
19 .filter(e -> e.getRole().equals(Role.EXECUTIVE))
20 .peek(e -> System.out.println("Executives"))
21 .filter(e -> e.getState().equals("CO"))
22 .peek(e -> System.out.println("CO Executives"))
23 .map(e -> e.getSalary() * Bonus.byRole(e.getRole()))
24 .forEach( s -> System.out.printf(
25 " Bonus paid: $%,6.2f %n", s));
26
27 System.out.println("\n== First CO Bonus ==");
28 Employee tempEmp = new Employee.Builder().build();
29 Optional<Employee> result = eList.stream()
30 .peek(e -> System.out.println("Stream start"))
31 .filter(e -> e.getRole().equals(Role.EXECUTIVE))
32 .peek(e -> System.out.println("Executives"))
33 .filter(e -> e.getState().equals("CO"))
34 .peek(e -> System.out.println("CO Executives"))
35 .findFirst();
36
37 if (result.isPresent()){
.
38 result.get().printSummary();
39 }
40 }
41 }
```

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 shortcircuiting
terminal operation.

A05AnyMatch.java

```
10 public class A05AnyMatch {
11
12 public static void main(String[] args) {
13
14 List<Employee> eList = Employee.createShortList();
15
16 System.out.println("\n== First CO Bonus ==");
17 Optional<Employee> result;
18
19 if (eList.stream().anyMatch(
20 e -> e.getState().equals("CO"))){
21
22 result = eList.stream()
23 .peek(e -> System.out.println("Stream"))
24 .filter(e -> e.getRole().equals(Role.EXECUTIVE))
25 .filter(e -> e.getState().equals("CO"))
26 .findFirst();
27
28 if (result.isPresent()){result.get().printSummary();}
29 }
```

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

```
15 List<Employee> eList = Employee.createShortList();
16
17 System.out.println("\n== Executive Count ==");
18 long execCount =
19 eList.stream()
20 .filter(e -> e.getRole().equals(Role.EXECUTIVE))
21 .count();
```

```
22
23 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

```
23 System.out.println("Exec count: " + execCount);
24
25 System.out.println("\n== Highest Paid Exec ==");
26 Optional highestExec =
27 eList.stream()
28 .filter(e -> e.getRole().equals(Role.EXECUTIVE))
29 .max(Employee::sortBySalary);
30
31 if (highestExec.isPresent()){
.
32 Employee temp = (Employee) highestExec.get();
33 System.out.printf(
34 "Name: " + temp.getGivenName() + " "
35 + temp.getSurName() + " Salary: $%,6.2f %n ",
36 temp.getSalary());
37 }
```

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

```
39 System.out.println("\n== Lowest Paid Staff ==");
40 Optional lowestStaff =
41 eList.stream()
42 .filter(e -> e.getRole().equals(Role.STAFF))
43 .min(Comparator.comparingDouble(e -> e.getSalary()));
44
45 if (lowestStaff.isPresent()){
46 Employee temp = (Employee) lowestStaff.get();
47 System.out.printf("Name: " + temp.getGivenName()
48 + " " + temp.getSurName() +
49 " Salary: $%,6.2f %n ", temp.getSalary());
50 }
```

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.

.

A07CalcSum.java

```
26 System.out.println("\n== Total CO Bonus Details ==");
27
```

```
28 result = eList.stream()
29 .filter(e -> e.getRole().equals(Role.EXECUTIVE))
30 .filter(e -> e.getState().equals("CO"))
31 .peek(e -> System.out.print("Name: "
32 + e.getGivenName() + " " + e.getSurName() + " "))
33 .mapToDouble(e -> e.getSalary() * Bonus.byRole(e.getRole()))
34 .peek(d -> System.out.printf("Bonus paid: $%,6.2f %n", d))
35 .sum();
36
37 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

```
28 System.out.println("\n== Average CO Bonus Details ==");
29
30 result = eList.stream()
31 .filter(e -> e.getRole().equals(Role.EXECUTIVE))
32 .filter(e -> e.getState().equals("CO"))
33 .peek(e -> System.out.print("Name: " + e.getGivenName()
34 + " " + e.getSurName() + " "))
35 .mapToDouble(e -> e.getSalary() * Bonus.byRole(e.getRole()))
36 .peek(d -> System.out.printf("Bonus paid: $%,6.2f %n", d))
37 .average();
38
39 if (result.isPresent()){
40 System.out.printf("Average Bonuses paid: $%,6.2f %n",
```

```
41 result.getAsDouble());
42 }
43 }
```

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
.
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

```
10 public class A09SortBonus {
11 public static void main(String[] args) {
12 List<Employee> eList = Employee.createShortList();
13
14 System.out.println("\n== CO Bonus Details ==");
15
16 eList.stream()
17 .filter(e -> e.getRole().equals(Role.EXECUTIVE))
18 .filter(e -> e.getState().equals("CO"))
19 .mapToDouble(e -> e.getSalary() * Bonus.byRole(e.getRole()))
20 .sorted()
21 .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.

.

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

```
23 System.out.println("\n== CO Bonus Details Reversed ==");
24
25 eList.stream()
```

```
26 .filter(e -> e.getRole().equals(Role.EXECUTIVE))
27 .filter(e -> e.getState().equals("CO"))
28 .sorted(Comparator.comparing(Employee::getSurName).reversed())
29 .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

```
31 System.out.println("\n== Two Level Sort, Dept then Surname ==");
32
33 eList.stream()
34 .sorted(
35 Comparator.comparing(Employee::getDept)
36 .thenComparing(Employee::getSurName))
37 .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
```

.

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

```
12 public class A11Collect {
13
14 public static void main(String[] args) {
15
16 List<Employee> eList = Employee.createShortList();
17
18 List<Employee> nList = new ArrayList<>();
19
20 // Collect CO Executives
21 nList = eList.stream()
22 .filter(e -> e.getRole().equals(Role.EXECUTIVE))
23 .filter(e -> e.getState().equals("CO"))
24 .sorted(Comparator.comparing(Employee::getSurName))
25 .collect(Collectors.toList());
26
27 System.out.println("\n== CO Bonus Details ==");
28
29 nList.stream()
30 .forEach(Employee::printSummary);
31
32 }
33
34 }
```

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

```
12 public class A12CollectMath {
13
14 public static void main(String[] args) {
15
16 List<Employee> eList = Employee.createShortList();
.
17
18 // Collect CO Executives
19 double avgSalary = eList.stream()
20 .filter(e -> e.getRole().equals(Role.EXECUTIVE))
21 .filter(e -> e.getState().equals("CO"))
22 .collect(
23 Collectors.averagingDouble(Employee::getSalary));
24
25 System.out.println("\n== CO Exec Avg Salary ==");
26 System.out.printf("Average: $%,9.2f %n", avgSalary);
27
28 }
29
30 }
```

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

```
12 public class A13CollectJoin {
13
14 public static void main(String[] args) {
15
16 List<Employee> eList = Employee.createShortList();
17
18 // Collect CO Executives
19 String deptList = eList.stream()
20 .map(Employee::getDept)
21 .distinct()
22 .collect(Collectors.joining(", "));
23
24 System.out.println("\n== Dept List ==");
25 System.out.println("Total: " + deptList);
26
27 }
28
29 }
```

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
```

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
```

.

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

```
12 public class A15CollectCount {
13
14 public static void main(String[] args) {
15
16 List<Employee> eList = Employee.createShortList();
17
18 Map<String, Long> gMap = new HashMap<>();
19
20 // Collect CO Executives
21 gMap = eList.stream()
22 .collect(
23 Collectors.groupingBy(
24 e -> e.getDept(), Collectors.counting()));
25
26 System.out.println("\n== Employees by Dept ==");
27 gMap.forEach((k,v) ->
28 System.out.println("Dept: " + k + " Count: " + v)
29 );
30
31 }
32
33 }
```

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`.

A16CollectPartition.java

```
12 public class A16CollectPartition {
13
14 public static void main(String[] args) {
15
16 List<Employee> eList = Employee.createShortList();
17
18 Map<Boolean, List<Employee>> gMap = new HashMap<>();
19
20 // Collect CO Executives
21 gMap = eList.stream()
22 .collect(
23 Collectors.partitioningBy(
24 e -> e.getRole().equals(Role.EXECUTIVE)));
25
26 System.out.println("\n== Employees by Dept ==");
27 gMap.forEach((k,v) -> {
28 System.out.println("\nGroup: " + k);
29 v.forEach(Employee::printSummary);
30 });
31
32 }
33
34 }
```

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
```

Practice 4-1: 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 `D:/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
```

.

Practice 4-2: 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 `D:/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
.
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.

.

Practice 4-3: 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 `D:/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.

.

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
```

.

Practice 4-4: 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 `D:/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
```

.

Practice 4-5: 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 `D:/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
.
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
.
```

Practice 4-6: 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 `D:/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
```

Practice 4-7: 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 `D:/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
```

Practice 4-8: 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 `D:/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.

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
.
.

.
```

Practices for Lesson 5:

Using the Date/Time API

Practices Overview

In these practices, you will work with the new date and time API.

Practice 5-1: Summary Level: Working with local dates and times

Overview

In this practice you work with `LocalDate`, `LocalTime`, and `LocalDateTime` objects to provide answers to the questions asked in the practice. Local objects have no concept of a time zone, so you can assume that all of the questions in the practice relate to the local time zone. Also, all of the dates utilize the ISO calendar.

Tasks

1. Open the `LocalDatesAndTimes12-01Prac` project in the `D:/labs/12-DateTime/practices/practice1` directory.
2. Open the Java class file, `LocalDatesAndTimes`, in the `com.example` package.
3. Read through the comments—these indicate what code you need to write to answer the questions provided.

 Consult the lecture slides and the documentation if you get stuck.

 When you have completed, the output from your class should look similar to the following output. (You do not need to format the print statements exactly the same way.)

```
Abe was 46 when he died.
Abe lived for 16863 days.
Bennedict was born in a leap year: true
Days in the year he was born: 366
Bennedict is 3 decades old.
It was a SATURDAY on his 21st birthday.
Planned Travel time: 340 minutes
Delayed arrival time: 20:44
The flight arrives in Miami: 2014-03-25T01:30
The delayed arrival time is: 2014-03-25T05:57
School starts: 2014-09-09
School ends: 2015-06-25
Number of school days: 183
The meeting time is: 2014-08-05T13:30
```

Practice 5-2: Detailed Level: Working with local dates and times

Overview

In this practice you work with `LocalDate`, `LocalTime` and `LocalDateTime` objects to provide answers to the questions asked in the practice. Local objects have no concept of a time zone, so you can assume that all of the questions in the practice relate to the local time zone. Also, all of the dates utilize the ISO calendar.

Tasks

1. Open the `LocalDatesAndTimes12-01Prac` project.

a. Select File > Open Project.

b. Browse to `D:/labs/12-DateTime/practices/practice1`.

c. Select `LocalDatesAndTimes12-01Prac` and click Open Project.

2. Open the Java class file, `LocalDatesAndTimes`, in the `com.example` package.

3. Given the scenario:

Abe Lincoln's Birthday: February 12, 1809, died April 15, 1855

How old was he when he died?

How many days did he live?

To implement this scenario, add the following code to `LocalDatesAndTimes.java`

```
LocalDate abeBorn = LocalDate.of(1809, FEBRUARY, 12);
LocalDate abeDies = LocalDate.of(1855, APRIL, 15);
System.out.println("Abe was " + abeBorn.until(abeDies, YEARS) +
" when he died.");
System.out.println("Abe lived for " + abeBorn.until(abeDies,
DAYS) + " days.");
System.out.println("");
```

4. Given the scenario:

Bennedict Cumberbatch, July 19, 1976

Born in a leap year?

How many days in the year he was born?

How many decades old is he?

What was the day of the week on his 21st birthday?

To implement this scenario, add the following code to `LocalDatesAndTimes.java`

```
LocalDate bennedict = LocalDate.of(1976, JULY, 19);
System.out.println("Bennedict was born in a leap year: " +
bennedict.isLeapYear());
System.out.println("Days in the year he was born: " +
bennedict.lengthOfYear());
LocalDate now = LocalDate.now();
System.out.println("Bennedict is " + bennedict.until(now,
DECADES) + " decades old.");
.
System.out.println("It was a " +
bennedict.plusYears(21).getDayOfWeek() + " on his 21st
birthday.");
System.out.println("");
```

5. Given the scenario:

Train departs Boston at 1:45PM and arrives New York 7:25PM

How many minutes long is the train ride?

If the train was delayed 1 hour 19 minutes, what is the actual arrival time?

To implement this scenario, add the following code to `LocalDatesAndTimes.java`

```
LocalTime depart = LocalTime.of(13, 45);
```

```
LocalTime arrive = LocalTime.of(19, 25);
System.out.println("Planned Travel time: " +
depart.until(arrive, MINUTES) + " minutes");
System.out.println("Delayed arrival time: " +
arrive.plusHours(1).plusMinutes(19));
System.out.println("");
```

6. Given the scenario:

Flight: Boston to Miami, leaves March 24th 9:15PM. Flight time is 4 hours 15 minutes

When does it arrive in Miami?

When does it arrive if the flight is delays 4 hours 27 minutes?

To implement this scenario, add the following code to `LocalDatesAndTimes.java`

```
LocalDateTime leaveBoston = LocalDateTime.of(2014, MARCH,
24, 21, 15);
LocalDateTime arriveMiami =
leaveBoston.plusHours(4).plusMinutes(15);
System.out.println("The flight arrives in Miami: " +
arriveMiami);
System.out.println("The delayed arrival time is: " +
arriveMiami.plusHours(4).plusMinutes(27));
System.out.println("");
```

.

7. Given the scenario:

School semester starts the second Tuesday of September of this year.

Hint: Look at the TemporalAdjusters class

What is the date?

School summer vacation starts June 25th

Assuming:

/* Two weeks off in December

* Two other vacation weeks

* School is taught Monday - Friday

How many days of school are there?

Hint: keep track of the short weeks also

To implement this scenario, add the following code to `LocalDatesAndTimes.java`

```
int excludeWeeks = 5;
LocalDate schoolStarts = LocalDate.of(2014, SEPTEMBER,
1).with(TemporalAdjusters.firstInMonth(TUESDAY)).with(TemporalAd
```

```
justers.next(TUESDAY));
LocalDate endOfFirstWeek =
schoolStarts.with(TemporalAdjusters.next(FRIDAY));
long firstWeekDays = schoolStarts.until(endOfFirstWeek,
DAYS) + 1;
System.out.println("School starts: " + schoolStarts);
LocalDate schoolEnds = LocalDate.of(2015, JUNE, 25);
System.out.println("School ends: " + schoolEnds);
long lastWeeksDays = 0;
if (schoolEnds.getDayOfWeek() != MONDAY) {
LocalDate lastWeekStart =
schoolEnds.with(TemporalAdjusters.previous(MONDAY));
lastWeeksDays = lastWeekStart.until(schoolEnds, DAYS) + 1;
excludeWeeks++;
}
long days = ((schoolStarts.until(schoolEnds, WEEKS) -
excludeWeeks) * 5); // 7 days per week, weekdays are 5/7 of a
week.
days = days + firstWeekDays + lastWeeksDays;
System.out.println("Number of school days: " + days);
System.out.println("");
```

8. Given the scenario:

A meeting is scheduled for 1:30 PM next Tuesday. If today is Tuesday, assume it is today.

What is the time of the week's meetings?

To implement this scenario, add the following code to `LocalDatesAndTimes.java`

```
LocalTime meetingTime = LocalTime.of(13, 30);
LocalDate meetingDate =
LocalDate.now().with(TemporalAdjusters.nextOrSame(TUESDAY));
LocalDateTime meeting = LocalDateTime.of(meetingDate,
meetingTime);
System.out.println("The meeting time is: " + meeting);
System.out.println("");
```

9. Run the project, the output should look similar to the following output.

```
Abe was 46 when he died.
```

```
Abe lived for 16863 days.
Bennedict was born in a leap year: true
Days in the year he was born: 366
Bennedict is 3 decades old.
It was a SATURDAY on his 21st birthday.
Planned Travel time: 340 minutes
Delayed arrival time: 20:44
The flight arrives in Miami: 2014-03-25T01:30
The delayed arrival time is: 2014-03-25T05:57
School starts: 2014-09-09
School ends: 2015-06-25
Number of school days: 183
The meeting time is: 2014-08-05T13:30
```

.

Practice 5-2: Summary Level: Working with dates and times across time zones

Overview

In this practice, you work with time zone classes to calculate dates and times across time zones.

Tasks

1. Open the NetBeans project `DepartArrive12-02Prac` in the `D:/labs/12-DateTime/practices/practice2` directory.

2. Open the class file, `DepartArrive.java` in the `com.example` package.

3. Read through the comments—these indicate what code you need to write to answer the questions provided.

 Consult the lecture slides and the documentation if you get stuck.

 When you are complete, the output from your class should look similar to this (note that you need not format the print statements exactly the same way).

```
Flight 123 departs SFO at: 2014-06-13T22:30-07:00[America/Los_Angeles]
Local time BOS at departure: 2014-06-14T01:30-04:00[America/New_York]
Flight time: 5 hours 30 minutes
Flight 123 arrives BOS: 2014-06-14T07:00-04:00[America/New_York]
Local time SFO at arrival: 2014-06-14T04:00-07:00[America/Los_Angeles]
Flight 456 leaves SFO at: 2014-06-28T22:30-07:00[America/Los_Angeles]
Local time BLR at departure: 2014-06-29T11:00+05:30[Asia/Calcutta]
```

```
Flight time: 22 hours
Flight 456 arrives BLR: 2014-06-30T09:00+05:30[Asia/Calcutta]
Local time SFO at arrival: 2014-06-29T20:30-07:00[America/Los_Angeles]
Flight 123 departs SFO at: 2014-11-01T22:30-07:00[America/Los_Angeles]
Local time BOS at departure: 2014-11-02T01:30-04:00[America/New_York]
Flight time: 5 hours 30 minutes
Flight 123 arrives BOS: 2014-11-02T06:00-05:00[America/New_York]
Local time SFO at arrival: 2014-11-02T03:00-08:00[America/Los_Angeles]
```

.

Practice 5-2: Detailed Level: Working with dates and times across time zones

Overview

In this practice, you work with time zone classes to calculate dates and times across time zones.

Tasks

1. Open the project `DepartArrive12-02Prac.`

a. Select File > Open Project.

b. Browse to `D:/labs/12-DateTime/practices/practice2`

c. Select `DepartArrive12-02Prac` and click Open Project.

2. Open the `DepartArrive.java` file in the `com.example` package and make the following changes:

a. Set the time zone for the three cities.

```
ZoneId SFO = ZoneId.of("America/Los_Angeles");
ZoneId BOS = ZoneId.of("America/New_York");
ZoneId BLR = ZoneId.of("Asia/Calcutta");
```

b. Given the scenario:

Flight 123, San Francisco to Boston, leaves SFO at 10:30 PM June 13, 2014

The flight is 5 hours 30 minutes

What is the local time in Boston when the flight takes off?

What is the local time at Boston Logan airport when the flight arrives?

What is the local time in San Francisco when the flight arrives?

Complete the following steps:

To compute the local time in Boston when the flight takes off, add the following code:

```
LocalDateTime departure = LocalDateTime.of(2014, JUNE, 13, 22,
30);
ZonedDateTime departSFO = ZonedDateTime.of(departure, SFO);
```

```
System.out.println("Flight 123 departs SFO at: " + departSFO);
ZonedDateTime departTimeAtBOS =
departSFO.toOffsetDateTime().atZoneSameInstant(BOS);
System.out.println("Local time BOS at departure: " +
departTimeAtBOS);
System.out.println("Flight time: 5 hours 30 minutes");
```

c. To compute local time at Boston Logan airport when the flight arrives, add the following code:

```
ZonedDateTime arriveBOS =
departSFO.plusHours(5).plusMinutes(30).toOffsetDateTime().atZone
SameInstant(BOS);
System.out.println("Flight 123 arrives BOS: " +
arriveBOS);
```

d. To compute the local time in San Francisco when the flight arrives, add the following code:

```
ZonedDateTime arriveTimeAtSFO =
arriveBOS.toOffsetDateTime().atZoneSameInstant(SFO);
System.out.println("Local time SFO at arrival: " +
arriveTimeAtSFO);
System.out.println("");
```

3. Given the scenario:

- Flight 456, San Francisco to Bangalore, India, leaves SFO at Saturday, 10:30 PM June 28, 2014
- The flight time is 22 hours
- Will the traveler make a meeting in Bangalore Monday at 9 AM local time?
- Can the traveler call her husband at a reasonable time?

Modify `DepartArrive.java`.

a. Compute the local departure time at SFO.

```
departure = LocalDateTime.of(2014, JUNE, 28, 22, 30);
departSFO = ZonedDateTime.of(departure, SFO);
System.out.println("Flight 456 leaves SFO at: " +
departSFO);
```

b. Compute the local departure time at Bangalore.

```
ZonedDateTime departTimeAtBLR =
departSFO.toOffsetDateTime().atZoneSameInstant(BLR);
```

```
System.out.println("Local time BLR at departure: " +
departTimeAtBLR);
System.out.println("Flight time: 22 hours");
```

c. Compute the local arrival time at Bangalore.

```
ZonedDateTime arriveBLR =
departSFO.plusHours(22).toOffsetDateTime().atZoneSameInstant(BLR
);
System.out.println("Flight 456 arrives BLR: " + arriveBLR);
```

d. Compute the local arrival time at SFO.

```
arriveTimeAtSFO =
arriveBLR.toOffsetDateTime().atZoneSameInstant(SFO);
System.out.println("Local time SFO at arrival: " +
arriveTimeAtSFO);
System.out.println("");
```

4. Given the scenario:

Flight 123, San Francisco to Boston, leaves SFO at 10:30 PM Saturday, November 1st, 2014

Flight time is 5 hours 30 minutes.

What day and time does the flight arrive in Boston?

What happened?

Modify `DepartArrive.java`.

a. Compute the local departure time at SFO.

```
departure = LocalDateTime.of(2014, NOVEMBER, 1, 22, 30);
departSFO = ZonedDateTime.of(departure, SFO);
System.out.println("Flight 123 departs SFO at: " +
departSFO);
```

b. Compute the local departure time at Boston.

```
departTimeAtBOS =
departSFO.toOffsetDateTime().atZoneSameInstant(BOS);
System.out.println("Local time BOS at departure: " +
departTimeAtBOS);
System.out.println("Flight time: 5 hours 30 minutes");
```

c. Compute the local arrival time at SFO with the delay of 5 hours.

```
arriveBOS =
departSFO.plusHours(5).plusMinutes(30).toOffsetDateTime().atZone
```

```
SameInstant(BOS);
System.out.println("Flight 123 arrives BOS: " +
arriveBOS);
arriveTimeAtSFO =
arriveBOS.toOffsetDateTime().atZoneSameInstant(SFO);
System.out.println("Local time SFO at arrival: " +
arriveTimeAtSFO);
System.out.println("");
```

.

5. Run the project, the output could be similar to this:

```
Flight 123 departs SFO at: 2014-06-13T22:30-07:00[America/Los_Angeles]
Local time BOS at departure: 2014-06-14T01:30-04:00[America/New_York]
Flight time: 5 hours 30 minutes
Flight 123 arrives BOS: 2014-06-14T07:00-04:00[America/New_York]
Local time SFO at arrival: 2014-06-14T04:00-07:00[America/Los_Angeles]
Flight 456 leaves SFO at: 2014-06-28T22:30-07:00[America/Los_Angeles]
Local time BLR at departure: 2014-06-29T11:00+05:30[Asia/Calcutta]
Flight time: 22 hours
Flight 456 arrives BLR: 2014-06-30T09:00+05:30[Asia/Calcutta]
Local time SFO at arrival: 2014-06-29T20:30-07:00[America/Los_Angeles]
Flight 123 departs SFO at: 2014-11-01T22:30-07:00[America/Los_Angeles]
Local time BOS at departure: 2014-11-02T01:30-04:00[America/New_York]
Flight time: 5 hours 30 minutes
Flight 123 arrives BOS: 2014-11-02T06:00-05:00[America/New_York]
Local time SFO at arrival: 2014-11-02T03:00-08:00[America/Los_Angeles]
```

.

Practice 5-3: Summary Level: Formatting Dates

Overview

In this practice, you work with the `DateTimeFormatter`.

Tasks

1. Open the project `TimeBetween12-03Prac` in NetBeans from the `D:/labs/12-DateTime/practices/practice3` directory.

2. Open the class, `TimeBetween.java`.

3. Modify the class to read a string from the console and correctly identify the delta between today and the entered date in years, months, and days.

 Use the appropriate method to ensure that the values for the year, month and days are always positive.

4. The output should look similar to this:

```
Enter a date: (MMMM d, yyyy): July 9, 2014
Date entered: July 9, 2014
There are 0 years, 4 months, 16 days between now and the date
entered.
```

Practice 5-3: Detailed Level : Formatting Dates

Overview

In this practice, you work with the `DateTimeFormatter`.

Tasks

1. Open the project `TimeBetween12-03Prac` in NetBeans.

a. Select File > Open Project.

b. Browse to `D:/labs/12-DateTime/practices/practice3`

c. Select `TimeBetween12-03Prac` and click Open Project.

2. Open the class, `TimeBetween.java`.

3. Modify the class to read a string from the console and correctly identify the delta between today and the entered date in years, months, and days.

 Use the appropriate method to ensure that the values for the year, month and days are always positive.

a. Declare two variables.

```
String dateFormat = "MMMM d, yyyy";
LocalDate aDate = null;
```

b. Create a formatter to accept date entries using the USA common standard(month day, year).

```
DateTimeFormatter formatter =
DateTimeFormatter.ofPattern(dateFormat);
```

c. Use the parse method with the formatter to create a date. Add the following statement within the `try` block.

```
aDate = LocalDate.parse(dateEntered, formatter);
```

d. To calculate the years, months, and days between now and the date entered, enter the following code:

```
Period between;
if (aDate.isBefore(now)) {
between = Period.between(aDate, now);
```

```
} else {
between = Period.between(now, aDate);
}
```

e. Obtain the value of day, month and year and assign it to the variables: `days`, `months`, and `years`.

```
int years = between.getYears();
int months = between.getMonths();
int days = between.getDays();
```

f. Print the values of the years, months, and days.

```
System.out.println("There are " + years + " years, "
+ months + " months, "
+ days + " days between now and the date entered.");
```

4. Run the project, the output could be similar to this:

Practices for Lesson 6:
Parallel Streams

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 6-1: 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 6-2: 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 6-3: 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
```