Practices for Lesson 9: Lambda Built-in Functional Interfaces

Chapter 9

Practices for Lesson 9: 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 9-1: 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 `/home/oracle/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 9-2: 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 `/home/oracle/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 9-3: 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 `/home/oracle/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 9-4: 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 `/home/oracle/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
```