Lab 9 – Part 2

7. In the code folder for this lab, prob7, there is a Main class with a main method that prepares some data and calls two (unimplemented) methods: `ordering1` and `ordering2`. Each of these methods is supposed to sort a given input list in a stream pipeline – using a non-standard ordering rule which must be specified using `comparing` and `thenComparing` – and then output as a sorted list, which is then to be printed to the console.

   `ordering1(List<Integer>)` : The ordering of integers to be used here is one that would sort the integers in the following way:

   0, -1, 1, -2, 2, -3, 3, . . .

   `ordering2(List<String>)` **:** The ordering of Strings to be used here is the following:

   s precedes t if and only if reverse(s) comes before reverse(t) in the usual ordering of strings.

   For example, in this new ordering, "cba" precedes "bbd" because, when the strings are reversed, we see that "abc" precedes "dbb" in the usual string ordering.

   In the main method, the expected outputs of each of these methods are shown.

8. In the prob8 package there is a Main class with a method findProduct:

```
private boolean findProduct(String prodName) {
    for(OrderItem item : orderItems) {
        if(item != null) {
            Product p=item.getProduct();
            if(p != null) {
                String name = p.getProductName();
                if(name != null) {
                    if(name.equals(prodName)) return true;
                }
            }
        }
    }
    return false;
}
```

   This method searches through a list of OrderItems (which is populated by another method loadOrderItemData) to determine whether any of the OrderItems in the list contains a product having a specified name (called `prodName`).

   As you can see, the code is very messy, with multiple null tests. Use the technique discussed in the slides for chaining Optionals (using flatMap) to eliminate all null tests in this code. You will need to refactor the entity classes in this package so that getters return Optionals instead of values.

   To get started, use the startup code provided in the code folder for this problem.

9. Implement a method

   `public static void printSquares(int num)`

   which creates an `IntStream` using the `iterate` method. The method prints to the console the first `num` squares. For instance, if `num` = 4, then your method would output 1, 4, 9, 16. Note:

You will need to come up with a function to be used in the second argument of `iterate`. Do not use the `map` or `filter` operations on `Stream`.

10. Short Answer:
    a. What happens when the following code is executed?

```
public static void main(String[] args) {
   IntStream ones = IntStream.generate(() -> 1).distinct();
   ones.forEach(System.out::println);
}
```

    Explain. What would be a quick way to fix this?

    b. You have a `Stream` of `String`s called `stringStream` consisting of the values "Bill", "Thomas", and "Mary". Write the one line of code necessary to print this stream to the console so that the output looks like this:

    Bill, Thomas, Mary

    c. You have a `Stream` of `Integer`s called `myIntStream` and you need to output both the maximum and minimum values somehow, making use of this stream only once. Write compact code that efficiently accomplishes this.

11. In the package lesson9.labs.prob11a, there is an `Employee` class and a `Main` class, which has a `main` method that loads up a `Stream` of `Employee` instances.
    a. In the final line of the `main` method, write a stream pipeline (using `filters` and `maps`) which prints, *in sorted order (comma-separated, on a single line),* the full names (first name + " " + last name) of all `Employee`s in the list whose salary is greater than $100,000 and whose last name begins with any of the letters in the alphabet *past* the letter 'M' (so, any letters in the range 'N'– 'Z').

    For the main method provided in your lab folder, expected output is:

```
Alice Richards, Joe Stevens, John Sims, Steven Walters
```

    b. Turn your lambda/stream pipeline from part (a) into a Lambda Library element, following the steps in the slides. First, create a class LambdaLibrary; this class will contain only public static final lambda expressions. Then, identify the parameters that need to be passed in so that your lambda/stream pipeline can operate properly. Finally, think of a function-style interface (Function, BiFunction, TriFunction, etc) that can be used to accommodate your parameters and then name your pipeline, with the function-type interface as its type (as in the slide example). Call your Library element in the main method instead of creating the pipeline there, as you did in part (a).

12. In the package lesson9.labs.prob12, a Main class is provided that is essentially the same as the one used in Problem 11. Comments appear in the main method that indicate two queries that need to be executed. As in Problem 11, create a class LambdaLibrary that will store

implementations of these queries as lambda pipeline expressions. Then call these expressions in the main method to verify they produce the expected results.

13. In the folder lesson9.labs.prob13 there are classes Book and BookCopy, as in the Library project. Use a lambda/stream pipeline to implement an `isAvailable()` method in Book that uses the stream operation reduce (Hint: a Book is available if copy1 is available OR copy2 is available OR…). To test your code, add a Main class to the package and run the following main method:

```
public static void main(String[] args) {
    //set up
    Book book = new Book("test", 3);
    List<BookCopy> copies = book.getCopies();
    copies.forEach(copy -> copy.changeAvailability());

    //test
    System.out.println(book.isAvailable());
}
```

14. Rewrite the lazy singleton implementation (shown below) using `Optional`, so that nulls are not tested. Hint. Use `ofNullable`. Create a main method in your class to test that your `getInstance` method really works.

```
/** Singleton with lazy initialization. Not threadsafe */
public class MySingletonLazy{
    private static MySingletonLazy instance = null;
    private MySingletonLazy() {}
    public static MySingletonLazy getInstance() {
        if(instance == null) {
            instance = new MySingletonLazy();
        }
        return instance;
    }
}
```