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 using `ordering2`, "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 map) to eliminate all null tests in this code.

   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. You have a list of classes of type `Simple` which just contains a single boolean variable `flag`. You want to define a method that returns `true` if at least one instance of `Simple` in the list has `flag` set to `true`. Here is an imperative way of doing this:

```
public boolean someSimpleIsTrue(List<Simple> list) {
    boolean accum = false;
    for(Simple s: list) {
        accum = accum || s.flag;
    }
    return accum;
}
```

    See the startup code for this exercise. Rewrite the implementation of `someSimpleIsTrue` using the `reduce` operation on `Stream`.

    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. 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;
    }
}
```