MPP Standardized Programming Exam
April, 2017

This 90-minute programming test measures the success of your MPP course by testing your new skill level in two core areas of the MPP curriculum: (1) Lambdas and streams, and (2) Implementation of UML in Java.You will need to demonstrate a basic level of competency in these areas in order to move past MPP.

Your test will be evaluated with marks "Pass" or "Fail." A "Pass" means that you have completed this portion of evaluation only; your professor will evaluate your work over the past month to determine your final grade in your MPP course, taking into account your work on exams and assignments. A "Fail" means you will need to repeat MPP, with your professor's approval.

There are two programming problems to solve on this test. You will use the Java classes that have been provided for you in an Eclipse workspace. You will complete the necessary coding in these classes, following the instructions provided below.

Problem 1. **[Lambdas/Streams]** In your `prob1` package, you will find two classes, `Employee` and `EmployeeAdmin`. A `Main` class has also been provided that will make it convenient to test your code.

The `Employee` class has been fully implemented. It has three fields: `name, salary`, and `ssn` (which stores a social security number). `Employee` provides getters and setters for each of these fields.

The `EmployeeAdmin` class is intended to provide reports about `Employee`s. For this problem, the `EmployeeAdmin` class has two static methods, `prepareSsnReport` and `prepareEmpReport,` each of which accepts a `HashMap table` and a `List socSecNums` as arguments. The `HashMap` matches employee social security numbers with `Employee` objects. The `List` contains some employee social security numbers, represented as `String`s.

The method `prepareSsnReport` should return a list of social security numbers, sorted in ascending order (from numerically smallest to numerically largest), which belong to an `Employee` in the input table but which are not on the `socSecNums` input list.

The method `prepareEmpReport` should return a list of all `Employee`s in the input `table` whose social security number is in the input list `socSecNums` and whose `salary` is greater than $80,000. This list of `Employee`s does not need to be in any particular sorted order.

The `main` method in the `Main` class provides test data that you can use to test your code.

Below are examples of how these two methods should behave. Suppose we are given the following input data. In the input `table`, there are four entries: The first entry associates with the `ssn` "`223456789`" the `Employee` object ["Jim", 90000, "`223456789`"]. There are three additional entries in table. The list `socSecNums,` also provided, contains four social security numbers.

`table`:

```
"223456789"  →  ["Jim", 90000, "223456789"]
"100456789"  →  ["Tom", 88000, "100456789"]

"630426389"  →  ["Don", 60000, "630426389"]
"777726389"  →  ["Obi", 60000, "777726389"]
```

socSecNums :

```
"630426389",  "223456789"  , "929333111",  "100456789"
```

In this case, the method `prepareSsnReport` will return (in sorted order) the following list of social security numbers:

```
["777726389"]
```

And the method `prepareEmpReport` will return the following list of Employee objects. (Note: The output in this case makes use of the `toString` method, provided in the Employee class.)

```
["Tom", 88000, "100456789"],  ["Jim", 90000, "223456789"]
```

Note that the `Employee`s could be output in a different order in this case, since there is no requirement concerning the ordering of output.

Hint. The following code shows how to create a `Stream` containing the keys of a given `HashMap h`:

```
h.keySet().stream()
```

Requirements for this problem.

(1) Each of the methods you implement, in the class `EmployeeAdmin`, must be a single `Stream` pipeline. You must not make use of instance variables or local variables declared in the body of either method. (Example of a local variable:

```
int myMethod() {
       int x = //computation
       return x;
}
```

Here, x is a local variable. Not allowed in this problem.)

(2) You must not modify the `Employee` class in any way. Also, you must not modify the signature of `EmployeeAdmin.prepareSsnReport` or `EmployeeAdmin.prepareEmpReport` or change any of its qualifiers (`public, static`).

(3) There must not be any compilation errors or runtime errors in the solution that you submit.

Problem 2. [UML → Code] In a company, the administrative office receives hourly formatted reports from the Billing, Sales, and Marketing Departments. Each of these departments provides a message queue, which the administrative office reads each hour. When it is time to read the department queues, the administrative office software reads each queue's message and then assembles all the messages into a report. A typical formatted report looks like this:

```
Billing: Number of hard-copy mailers sent yesterday: 10000
Marketing: Number of viewings of yesterday's informercial: 20,000
Sales: Yesterday's revenue: $20,000
```

For this problem, you will implement classes `BillingDept`, `SalesDept`, `MarketingDept`, and their superclass `Department`. Note from the class diagram (below) that each of these subclasses of `Department` implements the method `getName()`. In the table below, the return values of this method are provided:

Class	Return value for the getName() method
`BillingDept`	`"Billing"`
`SalesDept`	`"Sales"`
`MarketingDept`	`"Marketing"`

You will also implement an `Admin` class and the method `hourlyCompanyMessage`(), which reads the message in each of the `Department` queues and assembles them into a report, returned as a `String`. In order to assemble the messages and organize them into the correct format, the `format()` method in `Admin` must be called.

The `Department` queues are implementations of a special queue class that has been provided for you, called `StringQueue`. The `StringQueue` stores messages within each department class, and your `hourlyCompanyMessage()` implementation will read each of the departmental queues to get the current message from each.

It is possible that, when you access one of the Departmental queues, an `EmptyQueueException` (which is a class that has been implemented for you) could be thrown; your `hourlyCompanyMessage()` method must handle any such thrown exception.

There is a test class, `Main`, whose `main` method provides test data to test your code. The expected output of the `main` method (after commented sections have been uncommented) is:

```
Billing: Number of hard-copy mailers sent yesterday: 10000
Marketing: Number of viewings of yesterday's informercial: 20,000
Sales: Yesterday's revenue: $20,000

Billing: Number of overdue clients: 20
Marketing: Number of internal marketing meetings this week: 40
Sales: No updates
```

Important: Each of the departments `BillingDept, SalesDept, MarketingDept` has a method besides `getName()`, indicated in the class diagram below. These methods have already been declared for you – you should not implement these methods. The table below lists these methods and the class each belongs to; remember, these methods *do not* need to be implemented and they have already been declared for you.

Methods You Should *Not* Implement	
`BillingDept`	`monthlyReport()`
`SalesDept`	`requestMarketingMaterials()`
`MarketingDept`	`applyForJob()`

Tasks.

(1) Carefully implement the classes shown in the class diagram, with behavior shown in the sequence diagram (below), observing multiplicities, roles, and stereotypes.
(2) Implement all operations shown in the class diagram, except for those in the table above.
(3) The most important implementation you need to do is for the `Admin` method `hourlyCompanyMessage`. During evaluation of your code, the expected output of this method (shown above) will be compared with yours. To test your output, use the `main` method provided for you in the `Main` class.

Method You Need to Implement	Class	Description
`getName`	`BillingDept`, `SalesDept`, `MarketingDept`	Returns the name of the department (using values mentioned in table above)
`addMessage`	`Department`	Adds a message to the queue that is stored in Department
`nextMessage`	`Department`	Reads the queue stored in Department and handles any exception that could be thrown by the queue
`format(name, msg)`	`Admin`	Returns a string in the form `name: msg`
`hourlyCompanyMessage`	`Admin`	Reads all Department queues and formats each message using the format method.

Requirements for this problem.

(1) You must use the `StringQueue` class provided whenever a queue is needed in your code.

(2) The output of the `Admin` method `hourlyCompanyMessage` must be formatted as it has been done in the samples shown above.

(3) Your `hourlyCompanyMessage` must call the `format()` method to perform the necessary formatting of messages.

(4) The flow of your code, when `hourlyCompanyMessage` is executed, must match the sequence diagram shown below.

(5) The `nextMessage()` method in `Department` must read the next value in the `StringQueue` and return it. Since reading the `StringQueue` could cause an `EmptyQueueException` to be thrown, you must make use of a `try/catch` block. The body of the `catch` block can be left empty (you do not need to handle an `EmptyQueueException` in any special way).

(6) The `String` returned from `hourlyCompanyMessage()` in `Admin` must have the following format (which is produced by the `format(name, msg)` method):

```
<department_name>: <message>
```

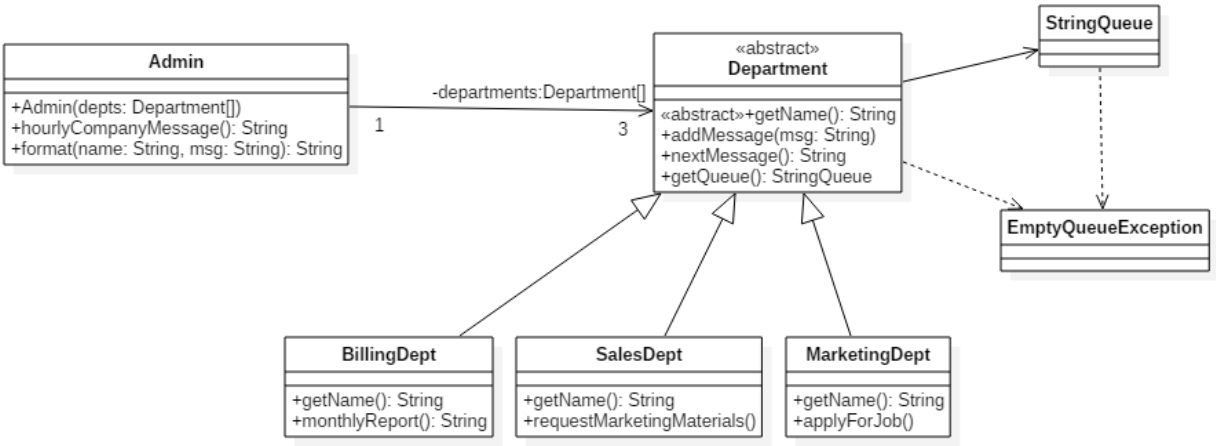
For example:

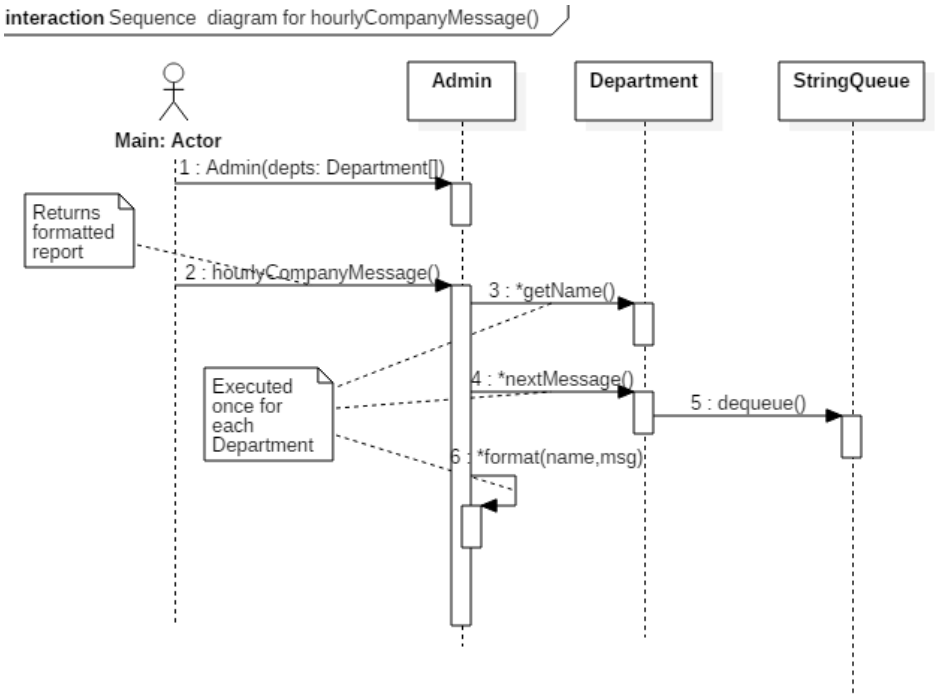
```
Billing: This is a message from the BillingDepartment
```

The department name that appears in the output is obtained by calling the `getName()` method.

(7) You must not modify the code in `StringQueue` or `EmptyQueueException`. (Note that these two classes are shown in the diagrams, but implementation details are not shown since they are already fully implemented.) And, you are strongly advised to avoid modifying any part of the `Main` class, except for uncommenting the commented part of the code when you are ready to use it . Note: The `Main` class plays the role of an actor in this piece of software, and is indicated this way in the sequence diagram. (For this reason, `Main` does not appear in the class diagram.)

(8) Your submitted code must not have compiler errors or runtime exceptions when executed.


Admin
+Admin(depts: Department[])
+hourlyCompanyMessage(): String
+format(name: String, msg: String): String
-departments:Department[]
1
3
«abstract»
Department
«abstract»+getName(): String
+addMessage(msg: String)
+nextMessage(): String
+getQueue(): StringQueue
StringQueue
EmptyQueueException
BillingDept
+getName(): String
+monthlyReport(): String
SalesDept
+getName(): String
+requestMarketingMaterials()
MarketingDept
+getName(): String
+applyForJob()


interaction Sequence diagram for hourlyCompanyMessage()
Main: Actor
Admin
Department
StringQueue
1 : Admin(depts: Department[])
Returns formatted report
2 : hourlyCompanyMessage()
3 : *getName()
Executed once for each Department
4 : *nextMessage()
5 : dequeue()
6 : *format(name,msg)