Indiana University
CSCI-C212: Introduction to Software Systems (SP24)
Joshua Crotts & Muazzam Siddiqui

Assignment 4
February 28, 2024

Streams, Objects, Classes

Important Dates:

- Assigned: February 28, 2024
- Deadline: March 27, 2024 at 11:59 PM EST

Objectives:

- Students learn to use the Stream API and its methods to solve a problem.
- Students design classes as blueprints for objects.

What To Do:

Because we're introducing classes with this assignment, it is no longer appropriate to use the class name `ProblemX`. Instead, design classes with the given specification in each problem, along with the appropriate test suite. **Do not round your solutions!**

You must write sufficient tests and adequate documentation.

1. This problem has three parts.

   Design the `StreamMethods` class with the following static methods. Write tests for each in the `StreamMethodsTester` class. For each of these, you may only use the methods from the `Stream` API as discussed in class. Do not use loops, recursion, or anything else aside from the `Stream` API. ***DO NOT MODIFY THE GIVEN LIST OF VALUES!***

   (a) Design the `List<String> removeLonger(List<String> los, int n)` method that receives a list of strings, and removes all strings that contain more characters than a given integer n. Return this result as a new list.

   (b) Design the `List<Integer> sqAddFiveOmit(List<Integer> lon)` that receives a list of numbers, returns a list of those numbers squared and adds five to the result, omitting any of the resulting numbers that end in 5 or 6.

   (c) Design the `Map<Integer, Integer> groupLength(List<String> los)` that, when given a list of strings, groups the words by their length and counts how many words are there for each length. This means that the return value should be a `Map<Integer, Integer>`. There are a couple of ways to do this, and any way that correctly utilizes the Stream API is fine.

2. This problem has three parts.

   (a) Design the `Employee` class, which stores the employee's first and last names as strings, their birthyear as an integer, their yearly salary as a double (we will assume that all employees are paid some value greater than zero), and their employee ID as a string. To make things interesting, assume that an employee's ID is not alterable and must be set in the constructor. The employee ID is constructed using the first five characters of their last name, the first letter of their first name, and the last two digits of their birthyear. For instance, if the employee's name is `Joshua Crotts` and their birthyear is `1999`, their employee ID is `CrottJ99`. Its constructor should receive the name, birthyear, and salary as parameters, but build the employee ID from the name and birthyear. Be sure to design the relevant accessor and mutator methods. Hint: you wrote this method in one of your labs; just go grab it!

   (b) As part of the `Employee` class, design the `void bonus()` method, which updates the salary of an employee. Calling `bonus` on an employee increases their salary by ten percent.

   (c) As part of the `Employee` class, override the `equals` and `toString` methods from the `Object` class to compare two employees by their employee ID and to print the employee's name, birthyear, salary, and employee ID respectively separated by commas.

3. This problem has six parts.

   In this exercise you will design a class for storing employees. This relies on completing the `Employee` class exercise.

   (a) Design the `Job` class, which stores a list of employees `ArrayList<Employee>` as an instance variable. Its constructor should receive no arguments.

   (b) Design the `void addEmployee(Employee e)` method, which adds an employee to the `Job`.

   (c) Design the `void removeEmployee(Employee e)` method, which removes an employee from the `Job`.

   (d) Design the `Optional<Double> computeAverageSalary()` method, which returns the average salary of all employees in the `Job`. If there are no employees, return an empty `Optional`.

   (e) Design the `Optional<Employee> highestPaid()` method, which returns the employee whose salary is the highest of all employees in the `Job`. If there are no employees, return an empty `Optional`.

   (f) Override the `toString` method to print out the list of employees in the `Job`. You can use the default `toString` implementation of the `ArrayList` class.

4. This problem has seven parts.

   (a) Design the `Matrix` class, which stores a two-dimensional array of integers. Its constructor should receive two integers m and n representing the number of rows and columns respectively, as well as a two-dimensional array of integers. Copy the integers from the passed array into an instance variable array.

   (b) Design the `void set(int i, int j, int val)` method, which sets the value at row i and column j to `val`.

   (c) Design the `void add(Matrix m)` method, which adds the values of the passed matrix to the current matrix. If the dimensions of the passed matrix do not match the dimensions of the current matrix, do nothing.

   (d) Design the `void multiply(Matrix m)` method, which multiplies the values of the passed matrix to the current matrix. If we cannot multiply m with this matrix, do nothing.

   (e) Design the `void transpose()` method, which transposes the matrix. That is, the rows become the columns and the columns become the rows. You may need to alter the dimensions of the matrix.

   (f) Design the `void rotate()` method, rotates the matrix 90 degrees clockwise. To rotate a matrix, compute the transposition and then reverse the rows. You may need to alter the dimensions of the matrix.

   (g) Override the `String toString()` method to print out the matrix in a boxed format.

5. This exercise has seven parts.

   A *persistent data structure* is one that saves intermittent data structures after applying operations that would otherwise alter the contents of the data structure. Take, for instance, a standard FIFO queue. When we invoke its 'enqueue' method, we modify the underlying data structure to now contain the new element. If this were a persistent queue, then enqueueing a new element would, instead, return a new queue that contains all elements and the newly-enqueued value, thereby leaving the original queue unchanged.

   (a) First, design the generic, private, and static class `Node` inside a generic `PQueue` class skeleton. It should store, as instance variables, a pointer to its next element as well as its associated value.

   (b) Then, design the `PQueue` class, which represents a persistent queue data structure. As instance variables, store "first" and "last" pointers as `Node` objects, as well as an integer to represent the number of existing elements. In the constructor, instantiate the pointers to `null` and the number of elements to zero.

   (c) Design the `PQueue<T> enqueue(T t)` method that enqueues a value onto the end of a new queue containing all the old elements, in addition to the new value.

   (d) Design the `PQueue<T> dequeue()` method that removes the first element of the queue, returning a new queue without this first value.

   (e) Design the `T peek()` method that returns the first element of the queue.

   (f) Design the `static PQueue<T> of(T...vals)` method that creates a queue with the values passed as `vals`. Note that this must be a variadic method. Do not create a series of `PQueue` objects by enqueueing each element into a distinct queue; this is incredibly inefficient. Instead, allocate each `Node` one-by-one, thereby never calling `enqueue`.

   (g) Design the `int size()` method that returns the number of elements in the queue. You should not traverse the queue to compute this value.

   (h) Override the `.equals` method to compare two `PQueue` objects for equality, which will traverse the lists and determine if they contain the same elements in the same order.