Indiana University
CSCI-C212: Introduction to Software Systems (S25)
Joshua Crotts, Muazzam Siddiqui

Problem Set 4
March 5, 2025

Streams, Objects, Classes

Important Dates:

- Assigned: March 5, 2025
- Deadline: April 2, 2025 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 problem set, 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!**

This problem set contains eight required problems, meaning the maximum possible score is 100%/100%.

What You Cannot Use:

You cannot use any content beyond Chapter 4.2. Namely, do not use interfaces, inheritance, or anything that trivializes the problem. If you have questions, contact a staff member.

Note: for problems 1-4, create the `StreamMethods` class, and place the methods that you design, in this class. You must use the Stream API for all four of these problems.

Problem 1

Design the `static boolean containsHigh(List<int[]> lop)` method that, when given a list of two-element arrays representing x, y coordinate pairs, returns whether or not any of the y-coordinates are greater than 450.

Problem 2

Design the `static 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. You must use the Stream API.

Problem 3

Design the `static 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. You must use the Stream API.

Problem 4

Design the `static int filterSumChars(String s)` method that, when given a string s, removes all non-alphanumeric characters, converts all letters to uppercase, and computes the sum of the ASCII values of the letters. Digits should also be added, but use the digit itself and not its ASCII value. You must use the Stream API.

Problem 5

The *bag*, or multi-set, data structure is a simple set-like data structure, with the exception that elements can be inserted multiple times into a bag. Order does not matter with a bag, nor does its internal structure. With bags, we can add items, remove items, query how many of an item there are, and return the number of items in the bag.

(a) Design the generic `Bag<T>` class. It would be inefficient to simply store a list of all the items in the bag. So, your bag will employ a `Map<T, Integer>`, which associates an item T with its count in the map. Instantiate the map as an instance variable as a `HashMap`.

(b) Design the `void insert(T t)` method, which inserts an item t into the bag. If the item already exists, its associated count is incremented by one.

(c) Design the `boolean remove(T t)` method, which removes an item t from the bag, meaning its associated count is decremented by one. If t does not exist, the method returns `false`, and otherwise returns `true`. If decrmenting the count results in 0, then remove the key t.

(d) Design the `int count(T t)` method that returns the respective quantity of item t in the bag.

(e) Design the `int size()` method that returns the number of items in the bag. The method should *not* traverse over the keys and sum the values; this should be a "constant-time" operation. Don't overthink how to do this!

(f) Design the `boolean contains(T t)` method that returns whether the bag contains `t`.

(g) Design the `boolean isSubBag(Bag b)` method that determines whether b is a sub-bag of `this` bag. A sub-bag b_1 is a sub-bag of b_2 if, for every element $i \in b_1$, then $i \in b_2$ and $\texttt{count}(b_1[i]) \leq \texttt{count}(b_2[i])$.

Problem 6

(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 (you may assume that the number of rows and columns of the passed array are equal to m and n). Copy the integers from the passed array into an instance variable array. Do *not* simply assign the provided array to the instance variable!

Note: your constructor must look like `Matrix(int rows, int columns, int[][] M)`. *(The names of the formal parameters does not matter, but the order does matter.) Also, be sure to include three methods for accessing the rows, columns, and the underlying array. Call them* `int getRows()`, `int getCols()`, *and* `int[][] getMatrix()`.

(b) Design the `void set(int i, int j, int val)` method, which sets the value at row i and column j to `val`. If the row or column is out of bounds, do nothing.

(c) Design the `boolean 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, return false and do not add the matrix.

(d) Design the `boolean multiply(Matrix m)` method, which multiplies the values of the passed matrix to the current matrix. If we cannot multiply m with this matrix, return false and do not multiply the matrix.

(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 `public String toString()` method to return a stringified version of the matrix. As an example, `"[[1, 2, 3], [4, 5, 6]]"` represents the following matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Problem 7

Repeated string concatenation is a common performance issue in Java. As we know, Java `String` objects are immutable, which means that concatenation creates a new `String` objects. This is fine for small strings, but for larger strings (or concatenation operations performed in a loop), this can be a performance bottleneck. Each concatenation requires copying the entire string. Java provides the `StringBuilder` class to alleviate the issue. In this exercise, you will design a `MiniStringBuilder` class that mimics the behavior of `StringBuilder`. You cannot use `StringBuilder` or the older `StringBuffer` classes in your implementation.

(a) Design the `MiniStringBuilder` class, which stores a `char[]` as an instance variable. The class should also store a variable to keep track of the number of "logical characters" that are in-use by the buffer.

(b) Design two constructors for the `MiniStringBuilder` class: one that receives no arguments and initializes the default capacity of the underlying `char[]` array to 20, and another that receives a `String` *s* and initializes the `char[]` array to the characters of *s*.

(c) Override the `public boolean equals(Object o)` method to return whether two `MiniStringBuilder` objects represent the same string.

(d) Override the `public int hashCode()` method to return the hash code of the `MiniStringBuilder` object. The hash code is defined as the hash code of the underlying array of characters. Use `Arrays.hashCode` rather than `Objects.hash`.

(e) Override the `public String toString()` method, which returns the `char[]` array as a `String` object. The resulting string should contain only the logical characters in the buffer, and not the entire array. Output the characters without any additional characters, such as brackets or commas.

(f) Design the `void append(String s)` method, which appends the given string *s* onto the end of the current string stored in the buffer. The given string should not simply be appended onto the end of the buffer, but rather added to the end of the previous string in the buffer. If the buffer runs out of space, reallocate the array to be twice its current size, similar to how we reallocate the array in the `MiniArrayList` example class.

(g) Design the `void clear()` method, which resets the `char[]` array to the default size of 20 and clears the character buffer.

Problem 8

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 `private PQueue<T> copy()` method that returns a new queue with the same elements as the current queue. You should divide this method into a case analysis: one where `this` queue is empty and another where it is not. In the former case, return a new queue with no elements. In the latter case, iterate over the elements of the queue, enqueuing each element into a new queue. You will need instantiate a new `Node` (reference) for each element.

(d) Override the `public boolean equals(Object o)` method that returns whether the elements of this queue are equal to the provided queue's elements. You will need to traverse over the queues in a fashion similar to how you do it in `copy`. Again, break this up into a case analysis: (1) if the provided object is not a `PQueue<?>`, return `false`. (2) Otherwise, if they do not have the same number of elements, return `false`. Otherwise, compare each element sequentially.

(e) 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. You should use the `copy` method to your advantage.

(f) Design the `PQueue<T> dequeue()` method that removes the first element of the queue, returning a new queue without this first value. You should use the `copy` method to your advantage.

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

(h) Design the `static <T> PQueue<T> of(T... vals)` method that creates a queue with the values passed as `vals`. Note that this must be a variadic method. **Warning:** 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`.

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