CSE 122 Final Exam
Autumn 2023

Name of Student: Pratham

Section (e.g., AA):__________________ Student Number (eg., 1234567): ____________________

The exam is divided into six questions with the following points:

#	Problem Area
1	Conceptual
2	Code Tracing
3	Debugging
4	Collections Programming
5	Objects Programming
6	Stacks/Queues Programming

Do not begin work on this exam until instructed to do so. Any student who starts early or who continues to work after time is called will receive U's on some problems as a penalty.

You are allowed one page of a reference sheet, front and back, as notes during the exam. Space is provided for your answers. There is also a reference sheet at the end that you should use. You are not allowed to access any other papers during the exam. You are NOT to use any electronic devices while taking the test, including calculators. Anyone caught using an electronic device will receive U's on some problems as a penalty.

The exam is not, in general, graded on code quality and you do not need to include comments. For the stack/queue and collections questions, however, you are expected to use generics properly and to declare variables using interfaces when possible. You may only use the methods on the reference sheet for the data structures listed. For objects programming, you should declare all fields to be private. Problems may specify more specific requirements. You are not allowed to use programming constructs we haven't discussed in class such as break, continue, or returns from a void method on this exam.

Do not abbreviate code, such as "ditto" marks or dot-dot-dot ... marks.

You are allowed to ask for scratch paper to use as additional space when writing answers, but you must indicate on the original page for the problem that part of the solution is on scratch paper. Failure to do so may result in your work on scratch paper not being graded.

If you finish the exam early, please hand your exam to the instructor and exit quietly through the front door. During the last 5 minutes of the exam, please stay in your seats to avoid disrupting others during the end of the exam.

Each problem is graded on an E/S/N scale. In general, to earn an E on a problem your solution must work without error and meet all the problem requirements. To earn an S, there is allowance for minor errors in the solution, but the problem requirements must still be met to earn an S. Unless specified by the problem, we do not grade on code quality.

Initial here to indicate you have read and agreed to these rules: ________________________

1. **Conceptual:** Each of these parts should be considered independent of the others.

Part A: (Select all that apply) Which of the following are true about Sets?

- [✓] A TreeSet stores elements in sorted order
- [ ] A HashSet preserves the insertion order of its elements
- [✓] A Set does not allow for duplicate values
- [ ] A HashSet is generally more efficient than a TreeSet

Part B: Consider the following method. For each of the following commented checkpoints, fill in the table for which conditions are always true (under any circumstance), only sometimes true, or never true at each comment. You can abbreviate **A**=always, **S**=sometimes and **N**=never.

```
public Set<Integer> mystery1(List<Integer> list, int num) {
    Set<Integer> result = new TreeSet<>();
    int n = 1;
    // Checkpoint A
    if (list.isEmpty()) {
        throw new IllegalArgumentException();
    }
    // Checkpoint B
    for (int i = 0; i < list.size(); i++) {
        n = list.get(i);
        // Checkpoint C
        if (n != 0 && num % n == 0) {
            list.remove(i);
            i--;
            result.add(n);
            // Checkpoint D
        }
    }
    // Checkpoint E
    return result;
}
```

```
An explanation for
Checkpoint A's example answers

list.isEmpty(): S
true for mystery([], 3) and false
for mystery([1,2,3], 4). Therefore
sometimes true.

num % n == 0: A
Any number mod 1 is 0, so this is
always true.

result.isEmpty(): A
result was just initialized and
nothing has been added yet.
Therefore it is always empty here.
```

	Checkpoint A	Checkpoint B	Checkpoint C	Checkpoint D	Checkpoint E
list.isEmpty()	**S**	N	N	S	S
num % n == 0	**A**	A	S	A	N
result.isEmpty()	**A**	A	S	N	S

Part C: (Select one option) Consider the following method. Which of the following options is the best "plain-English" explanation of what the code is doing?

```
public static Set<Integer> mystery2(Map<Integer, Set<Integer>> m) {
    Set<Integer> result = new HashSet<>();
    for (int num : m.keySet()) {
        Set<Integer> val = m.get(num);
        int sum = 0;
        for (int num2 : val) {
            sum += num2;
        }
        if (sum == num) {
            result.add(num);
        }
    }
    return result;
}
```

- ○ Returns a set containing a subset of the keys of m
- ○ Returns a set containing the sums of the inner sets in m
- [✓ Returns a set of the keys of m that equal the sum of their inner sets in m]
- [✓ Uses a for-each loop to look at each key from m to sum the numbers in the inner sets, adding the key to a set if the sum equals the associated key]

2. <u>**Code Tracing:**</u> Consider the method below.

```
public static Set<Integer> superSecretMystery(int[][] arr) {
    Set<Integer> result = new TreeSet<>();
    for (int i = arr.length - 1; i >= 0; i--) {
        for (int j = 0; j < arr[i].length; j++) {
            result.add((j-i) * 10 + arr[i][j]);
        }
    }
    return result;
}
```

For each 2d array below, indicate the contents of the Set that superSecretMystery would return where it is passed as a parameter. Write the elements of the Set comma-separated with square brackets (for example: [1, 2, 0, 5])

Input	**Returned Set**
[[3]]	[3]
[[7], [7], [7]]	[-13, -3, 7]
[[2, 0, 4], [12, 10, 0]]	[2, 10, 4]
[[5, -5, 8, 0, 6]]	[5, 5, 28, 30, 46]

3. **Debugging:** Consider the following buggy implementation of **removeEvensInRange.** The intended behavior of this method is to take a list of integers, as well as integers start and end, and modify that list so that it removes the even numbers between indices start and end (both exclusive), and return the number of elements that were removed.

For example, if a variable called list stores this sequence of values:

```
[3, -1, 0, 2, 5, -10, 8]
```

Calling removeEvensInRange(list, 1, 5); should cause list to store the following sequence of values afterwards and the value to be returned is 2 because there were two elements that were removed (0 and 2):

```
[3, -1, 5, -10, 8]
```

Notice that the order from the original list is maintained. Assume that the given list is not empty and the inputs for start and end are valid indices of the original list.

A TA wrote a buggy implementation of this method shown below.

```
1.  public static int removeEvensInRange(List<Integer> list, int start, int end) {
2.      int count = 0;
3.      for (int i = start + 1; i < end; i++) {
4.          int num = list.get(i);
5.          if (num % 2 == 0) {
6.              count++;
7.              list.remove(i);
8.              i++;
9.              end--;
10.         }
11.     }
12.     return count;
13. }
```

Part A: There is a single bug in this program that is your task to find and fix. Identify the *1 line of code* that causes the bug. Write your answer as a line number in the box to the right. For example, if you think line 13 has a bug, write **13** in the box.

Part A Answer i--;//changed this from i++ for part A

Part B: Fix the error in the method above. Since there is only one bug, this should not take a lot of code to fix. Specifically mention which line(s) you will change and how. If you are deleting some code, make sure it's clear what parts are being removed. If you are inserting new code, make sure it is unambiguous where this new code belongs. Mention specific line number(s). Write your answer for **Part B** in the box below.

```
public static int removeEvensInRange(List<Integer> list, int start, int end) {
      int count = 0;
      int i = start + 1;
      while(i<end) {
         int num = list.get(i);
         if (num % 2 == 0) {
            count++;
            list.remove(i);
            end--;
         }
         else{
            i++;// as we only have to increment it if we dont remove the element
         }
      }
      return count;
   }
```

4. **Collections Programming**: Write a method called **mostPopularHobby** that takes in a Map with keys that are TA names and values that are sets of hobbies of that TA. For example, if a variable called m contains the following:

```
{
 Atharva = [board games, comedy shows, hiking, video gaming],
 Chaafen = [Formula 1, reading, traveling, video gaming],
 Jaylyn = [hiking, traveling, video gaming],
 Shivani = [cafes, music shows]
}
```

then the call mostPopularHobby(m) should return the String "video gaming" because three of the TAs enjoy this hobby while the other hobbies are less common.

If there is a tie between hobbies, you should break them alphabetically. For instance, if Shivani had an additional hobby of hiking, then the method should return "hiking" since it would be enjoyed by three TAs and comes alphabetically before "video gaming".

You may assume that the given map is not empty and that none of the inner sets are empty.

Your method should not construct any new data structures other than a single Map. It should also not modify the input map or any of the inner sets. You should use interface types and generics appropriately.

Write your solution on the next page

5. **Objects Programming:** Consider the following interface Team. For this problem you are to write a class called RelayTeam which implements the Team interface and represents a team of people on a relay racing team and their times for running 400m. The RelayTeam should have a constructor which takes one String parameter, the team's mascot.

```
public interface Team {
    // Returns the mascot of this team.
    public String getMascot();

    // Adds a runner to this relay team with the given time in seconds.
    // Throws an IllegalArgumentException if the runner is already on the team
    // or if time is negative
    public void addRunner(String runner, double time);

    // Substitutes newRunner for oldRunner on the relay team with newTime
    // Throws an IllegalArgumentException if oldRunner is not on the team or
    // if newRunner is already on this team, or if newTime is negative.
    public void substituteRunner(String oldRunner, String newRunner, double newTime);

    // Returns the name of the runner on this team that is the fastest (shortest time).
    // If two runners are tied for fastest, this method should return the name that
    // is alphabetically first.
    // Throws an IllegalStateException if there are no runners on this team.
    public String getFastestRunner();

    // Returns the average running time across all runners on the team,
    // or 0.0 if there are no runners.
    public double getAverageTime();

    // Returns true if this team has a faster average time than the given other team,
    // and false otherwise.
    public boolean hasFasterAverage(Team other);

    // Returns a string representation of this team. The format should be
    // as follows:
    //     <team mascot>'s average time: <average time> s
    // Eg, for the Cheetahs with an average time of 52.0 seconds,
    // the resulting toString would look like:
    //     Cheetahs's average time: 52.0 s
    // You do not need to round the resulting average
    public String toString();
}
```

For example, if the following lines were executed using your RelayTeam class…

```
    Team team1 = new RelayTeam("Sloths");
    team1.addRunner("Simon", 40);
    team1.addRunner("Subhash", 51);
    team1.substituteRunner("Subhash", "Samira", 40);

    Team team2 = new RelayTeam("Turtles");
    team2.addRunner("Thuy", 45);
    team2.addRunner("Tanya", 55);
```

Then the following method calls would return…

```
    team1.getMascot();                    // Sloths
    team1.getFastestRunner();             // Samira
    team2.getAverageTime();               // 50.0
    team1.hasFasterAverage(team2);        // true
    team2.toString();                     // Turtles's average time: 50.0 s
```

Your RelayTeam class should implement the Team interface. Your RelayTeam class should have private fields and you should use interface types and generics appropriately.

Write your solution on the next page.

6. **<u>Stacks/Queues Programming:</u>** Write a method called **alphabetize** that takes a queue of Strings as a parameter and modifies the queue to sort the Strings based on their first letter. We will assume that all Strings in the queue begin with either 'a', 'b', or 'c'. For example, suppose a queue called q stores the following sequence of values:

```
    front ["august", "cornelia street", "bejeweled", "cardigan", "afterglow"] back
```

and we make the following call:

```
    alphabetize(q);
```

Then q should store the following values after the call:

```
    front ["august", "afterglow", "bejeweled", "cornelia street", "cardigan"] back
```

Notice that the Strings that start with 'a' are at the front, followed by the Strings that start with 'b', followed by the Strings that start with 'c'.

Also, notice that the queue is **<u>not</u>** fully alphabetized, rather, the ordering within a letter group is maintained. For example, "august" is <u>before</u> "afterglow" in both the original queue and the modified queue.

If the input queue is empty, then calling this method should not modify it.

For an E, your solution must obey the following restrictions. A solution that disobeys them may get an S, but it is not guaranteed.

* You may use <u>one stack</u> as auxiliary storage. You may not use other structures (arrays, lists, etc.), but you can have as many simple variables as you like.
* Use the Queue interface and Stack/LinkedList classes discussed in class.
* Use stacks/queues in stack/queue-like ways only. Do <u>not</u> use index-based methods such as get, search, or set, or for-each loops or iterators. You may call add, remove, push, pop, peek, isEmpty, and size.
* Do not use advanced material such as recursion to solve the problem.

You have access to the following two methods and may call them as needed to help you solve the problem:

```
    public static void s2q(Stack<String> s, Queue<String> q) {
        while (!s.isEmpty()) {
            q.add(s.pop());
        }
    }

    public static void q2s(Queue<String> q, Stack<String> s) {
        while (!q.isEmpty()) {
            s.push(q.remove());
        }
    }
```

You should write your solution in the box on the next page. If you need additional space, please indicate that your solution is continued on scratch paper.

^_^ CSE 122 Final Exam Reference Sheet ^_^

(DO NOT WRITE ANY WORK YOU WANTED GRADED ON THIS REFERENCE SHEET. IT WILL NOT BE GRADED)

Examples of Constructing Various Collections

```
List<Integer> list = new ArrayList<Integer>();
Queue<Double> queue = new LinkedList<Double>();
Stack<String> stack = new Stack<>();  // Diamond operator also permitted
Set<String> words = new HashSet<>();
Map<String, Integer> counts = new TreeMap<String, Integer>();
```

Methods Found in ALL collections (Lists, Stacks, Queues, Sets, Maps)

`equals(`**collection**`)`	Returns `true` if the given other collection contains the same elements
`isEmpty()`	Returns `true` if the collection has no elements
`size()`	Returns the number of elements in a collection
`toString()`	Returns a string representation such as `"[10, -2, 43]"`

Methods Found in both Lists and Sets (ArrayList, LinkedList, HashSet, TreeSet)

`add(`**value**`)`	Adds value to collection (appends at end of list)
`addAll(`**collection**`)`	Adds all the values in the given collection to this one
`contains(`**value**`)`	Returns `true` if the given value is found somewhere in this collection
`iterator()`	Returns an Iterator object to traverse the collection's elements
`clear()`	Removes all elements of the collection
`remove(`**value**`)`	Finds and removes the given value from this collection
`removeAll(`**collection**`)`	Removes any elements found in the given collection from this one
`retainAll(`**collection**`)`	Removes any elements *not* found in the given collection from this one

`List<Type>` Methods

`add(`**index**`, `**value**`)`	Inserts given value at given index, shifting subsequent values right
`indexOf(`**value**`)`	Returns first index where given value is found in list (-1 if not found)
`get(`**index**`)`	Returns the value at given index
`lastIndexOf(`**value**`)`	Returns last index where given value is found in list (-1 if not found)
`remove(`**index**`)`	Removes/returns value at given index, shifting subsequent values left
`set(`**index**`, `**value**`)`	Replaces value at given index with given value

`Stack<Type>` Methods (only allowed methods plus `size` and `isEmpty`)

`pop()`	Removes the top value from the stack and returns it; `pop` throw an `EmptyStackException` if the stack is empty
`push(`**value**`)`	Places the given value on top of the stack
`peek()`	Returns the value at the top from the stack without removing it; throws a `EmptyStackException` if the stack is empty

`Queue<Type>` Methods (only allowed methods plus `size` and `isEmpty`)

`add(`**value**`)`	Places the given value at the back of the queue
`remove()`	Removes the value from the front of the queue and returns it; throws a `NoSuchElementException` if the queue is empty
`peek()`	Returns the value at the front of the queue without removing it; throws a `NoSuchElementException` if the queue is empty

`Map<KeyType, ValueType>` Methods

`containsKey(`**key**`)`	`true` if the map contains a mapping for the given key
`get(`**key**`)`	The value mapped to the given key (`null` if none)
`keySet()`	Returns a `Set` of all keys in the map
`put(`**key**`, `**value**`)`	Adds a mapping from the given key to the given value
`putAll(`**map**`)`	Adds all key/value pairs from the given map to this map
`remove(`**key**`)`	Removes any existing mapping for the given key
`toString()`	Returns a string such as `"{a=90, d=60, c=70}"`
`values()`	Returns a `Collection` of all values in the map

`Iterator<Type>` Methods

`hasNext()`	Returns `true` if there is another element in the iterator
`next()`	Returns the next value in the iterator and progresses the iterator forward one element
`remove()`	Removes the previous value returned by the next. Can only call once after each call to `next()`

`String` Methods

`charAt(`**i**`)`	The character in this String at a given index
`contains(`**str**`)`	`true` if this String contains the other's characters inside it
`endsWith(`**str**`)`	`true` if this String ends with the other's characters
`equals(`**str**`)`	`true` if this String is the same as *str*
`equalsIgnoreCase(`**str**`)`	`true` if this String is the same as *str*, ignoring capitalization
`indexOf(`**str**`)`	First index in this String where given String begins (-1 if not found)
`lastIndexOf(`**str**`)`	Last index in this String where given String begins (-1 if not found)
`length()`	Number of characters in this String
`isEmpty()`	`true` if this String is the empty string
`startsWith(`**str**`)`	`true` if this String begins with the other's characters
`substring(`**i, j**`)`	Characters in this String from index *i* (inclusive) to *j* (exclusive)
`substring(`**i**`)`	Characters in this String from index *i* (inclusive) to the end
`toLowerCase()`, `toUpperCase()`	A new String with all lowercase or uppercase letters

`Math` Methods

`abs(`**x**`)`	Returns the absolute value of `x`
`max(`**x, y**`)`	Returns the larger of `x` and `y`
`min(`**x, y**`)`	Returns the smaller of `x` and `y`
`pow(`**x, y**`)`	Returns the value of `x` to the `y` power
`random()`	Returns a random number between `0.0` and `1.0`
`round(`**x**`)`	Returns `x` rounded to the nearest integer

Object/Interface Syntax

```
public class Example implements InterfaceExample {  |     public interface InterfaceExample {
    private type field;                              |         public void method();
    public Example() {                               |     }
       field = something;                            |
    }                                                |
    public void method() {                           |
       // do something                               |
    }                                                |
}                                                    |
```

Extra fun: Draw a picture of your TA as you imagine them when on vacation! We'll give these drawings to your TA as thanks for all their hard work this quarter! :D