CSE 122 Sample Final Solutions
Winter 2023

Name of Student: _____

Section (e.g., AA):_____ Student Number: _____

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 cheat 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: Consider the following code snippet. The following sub-parts will ask how many references to objects and how many objects exist in this program.

```
Set<Double> s = new HashSet<>();
Set<Double> s2 = s;
s.add(2.0);
s.add(-8.2);
Set<Double> s3 = s;
Set<Double> s4 = new HashSet<>();
s2.add(-3.8);
Set<Double> s5 = s;
s4 = s;
```

Part A-1: How many **objects** exist in this program? Write your answer in the box below.

2

Part A-2: How many **references to objects** exist in this program? Write in the box below

5

Part B: Consider the following method. For each of the following commented Points, 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 void mystery(List<Integer> list, int num) {
  int count = 0;
  // Point A

  if (list.isEmpty()) {
    throw new IllegalArgumentException();
  }
  // Point B

  for (int i = 0; i < list.size(); i++) {
    if (list.get(i) == num) {
      // Point C

      list.set(i, list.get(i) + 1);
      // Point D

      count++;
    }
  }
  // Point E
  return count;
}
```

An explanation for Point A's example answers

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

list.size() > 0: **S**
Can be given an empty list or an non-empty list. Therefore sometimes true.

count > 0: **N**
Initialized to 0 and is not modified at Point A. Never greater than 0 here.

	Point A	Point B	Point C	Point D	Point E
list.contains(num)	**S**	**S**	**A**	**S**	**N**
list.size() > 0	**S**	**A**	**A**	**A**	**A**
count > 0	**N**	**N**	**S**	**S**	**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? We use TreeSet here as the parameter because we require the parameter values to be sorted for this problem.

```
public List<String> method(TreeSet<String> s) {
    Iterator iter = s.iterator();
    List<String> l = new ArrayList<>();
    while (iter.hasNext()) {
        String str = iter.next();
        if (str.contains(" ")) {
            l.add(0, str);
            iter.remove();
        }
    }
}
```

◯ Copies all multiword strings to l in reverse alphabetical order.

◯ Using an iterator, goes through the TreeSet and checks whether each element contains a character and adds it to a List<String> if that is true.

◯ It produces a ConcurrentModificationException because the set is being modified while being iterated over.

● Moves all multiword strings from s to a returned list in reverse alphabetical order.

2. **Code Tracing:** Write the output that is printed when the given method below is passed each of the following maps as its parameter. We use TreeMap here as the parameter because we require the parameter values to be sorted for this problem. Your answer should display the right values in the right order.

```
public static void mystery(TreeMap<Integer, List<Double>> m) {
    for (int i = 0; i < 3; i++) {
        if (m.containsKey(i)) {
            List<Double> s = m.get(i);
            m.put(s.size(), s);
            s.add(i * 1.5);
        }
    }
    System.out.println(m);
}
```

For each call below, indicate what output is produced:

Method Call	Output Produced
{1=[9.0]}	{1=[9.0, 1.5]}
{0=[0.0, 1.0, 2.0], 3=[3.0], 4=[]}	{0=[0.0, 1.0, 2.0, 0.0], 3=[0.0, 1.0, 2.0, 0.0], 4=[]}
{1=[3.3, 2.5], 2=[0.0]}	{1=[3.3, 2.5 1.5, 3.0], 2=[3.3, 2.5 1.5, 3.0], 3=[3.3, 2.5 1.5, 3.0]}
{0=[3.3, -0.7], 1=[3.1, 3.99], 2=[4.3, 1.9]}	{0=[3.3, -0.7, 0.0], 1=[3.1, 3.99, 1.5, 3.0], 2=[3.1, 3.99, 1.5, 3.0], 3=[3.1, 3.99, 1.5, 3.0]}

3. **Debugging:** Consider the following buggy implementation of **bottomLeftSum**.

The intended behavior of this method is to take a 2D array of numbers, and return the sum of all the "bottom left" values in the 2D array. The bottom left values are the ones that are to the left of the line drawn from the diagonal going from left to right. Two method calls are shown as an example of what "bottom left" means (values highlighted as bold). You can assume that the 2D array has a non-zero number of rows and columns.

Example 1: Should return 66

```
[[1, 2, 3, 4],
 [2, 3, 4, 5],
 [3, 4, 5, 6],
 [4, 5, 6, 7],
 [5, 6, 7, 8]]
```

Example 2: Should return 18

```
[[1, 2, 3, 4, 5],
 [2, 3, 4, 5, 6],
 [3, 4, 5, 6, 7]]
```

```
 1. public int bottomLeftSum(int[][] nums) {
 2.     int sum = 0;
 3.     for (int i = 0; i < nums.length; i++) {
 4.         for (int j = 0; j < nums[i].length; j++) {
 5.             if (i < j) {
 6.                 sum += nums[i][j];
 7.             }
 8.         }
 9.     }
10. }
```

There is a *single* bug in this program that is your task to find and fix. As a hint, when running the code on the Example 1, it returns 24 instead of the expected 66.

Part A: Identify the *1 line of code* that causes the bug. Write your answer as a number in the box to the right.

Part A Answer Line 5 was the line causing the bug (including the wrong indices for the sum).

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.

One possible fix to line 5 is to change the statement to `if (j <= i) {` although there are other equivalent statements (e.g., `!(j > i)`).

4. **Collections Programming**: Write a method called **commonHobbies** that takes a map with keys that are TA names and values that are a list of favorite hobbies for that TA and that returns a new map that associates each hobby name with a list of TAs who share that hobby. TA names and hobby names are represented by strings.

For example, suppose a map called tas contains the following associations:

```
{Hitesh=[coding, hiking], Sara=[coding, hiking, reading],
 Sravani=[coding, reading, biking]}
```

The call commonHobbies(tas) should return a new map with the following associations:

```
{biking=[Sravani], coding=[Hitesh, Sara, Sravani], hiking=[Hitesh, Sara],
 reading=[Sara, Sravani]}
```

As in this example, the keys of the map returned by your method should appear in alphabetical order. The TA names can appear in any order in the lists that the method constructs. Your method should construct the new map and each of the lists contained in the map and can construct iterators but should otherwise not construct any other structured objects (no extra sets, lists, etc.). It should also not modify the map passed as a parameter and it should be reasonably efficient. You should use interface types and generics appropriately.

```
public Map<String, List<String>> commonHobbies(Map<String, List<String>>
tas) {
    Map<String, List<String>> result = new TreeMap<>();
    for (String ta : tas.keySet()) {
        for (String hobby : tas.get(ta)) {
            if (!result.containsKey(hobby)) {
                result.put(hobby, new ArrayList<>());
            }
            result.get(hobby).add(ta);
        }
    }
    return result;
}a
```

5. **Objects Programming:** Consider the following IceCream interface used by Molly Moons (a local ice cream shop). For this problem, write a class called IceCreamCone that implements the IceCream interface to implement the required methods. The IceCreamCone class should have a constructor that takes no arguments and sets up the necessary state.

```
// Represents an ice cream order that can contain various scoops of different flavors.
public interface IceCream {
    // Adds the given flavor of ice cream with the given quantity of scoops.
    // The same flavor can be added multiple times, which should increase that
    // flavor's scoop count. If the given number of scoops is non-positive,
    // this method throws an IllegalArgumentException.
    public void add(String flavor, int scoops);

    // Returns the number of scoops of the given flavor in this IceCream (0 if no scoops
    // the given flavor have been added))
    public int getFlavor(String flavor);

    // Returns the set of flavors in this IceCream (in no particular order)
    public Set<String> getFlavors();

    // Returns a string representation of this IceCream.
    // If there are no scoops of ice cream in IceCream, it returns
    //    "No ice cream :("
    // If scoops have been added, it will return a string
    //    "<scoops> scoops of ice cream with <flavors>"
    public String toString();

    // Returns true if the given IceCream contains the same set of flavors
    // as the other. They do not need to have the same number of scoops of each
    // flavor to have the same flavors.
    public boolean sameFlavors(IceCream other);
}
```

For example, if the following lines are executed:

```
    IceCream order0 = new IceCreamCone();
    IceCream order1 = new IceCreamCone();
    order1.add("vanilla", 1);
    order1.add("chocolate", 2);
    order1.add("vanilla", 2);
```

Then the following calls to toString would return:

```
    order0.toString();    "No ice cream :("
    order1.toString();    "5 scoops of ice cream with [chocolate, vanilla]"
```

In the description of the toString above, you should not include the < and > characters in the returned value as those are placeholders for the real value. The order of the flavors in the toString representation does not matter.

Your implementation of IceCreamCone should implement the IceCream interface. In terms of Code Quality, the IceCreamCone should have private fields and should implement all of the method behaviors as described above.

Write your solution on the next page.

```
public class IceCreamCone implements IceCream {
    private int totalScoops;
    private Map<String, Integer> scoops;

    public IceCreamCone() {
        totalScoops = 0;
        scoops = new HashMap<>();
    }

    public void add(String flavor, int n) {
        if (n < 1) { // OK to be n <= 1 for E/S
            throw new IllegalArgumentException();
        }

        if (scoops.containsKey(flavor)) {
            scoops.put(flavor, scoops.get(flavor) + n)
        } else {
            scoops.put(flavor, n);
        }
        totalScoops += n;
    }

    public int getFlavor(String flavor) {
        if (scoops.containsKey(flavor)) {
            return scoops.get(flavor);
        } else {
            return 0;
        }
    }

    public Set<String> getFlavors() {
        return scoops.keySet();
    }

    public String toString() {
        if (totalScoops == 0) {
            return "No ice cream :(";
        } else {
            return totalScoops + " scoops of ice of ice cream with " +
                scoops.keySet();
        }
    }

    public boolean sameFlavors(IceCream other) {
        return scoops.keySet().equals(other.getFlavors());
    }
}
```

6. **Stacks/Queues Programming:** Write a method named **separate** that accepts a queue of Strings as a parameter and that rearranges the values by their length. We will assume all of the Strings in the queue are length 1, 2, or 3. The queue should be rearranged so that the strings of length 1 appear first, followed by values that are length 2, followed by values that are length 3, otherwise preserving their relative order in the original queue.

For example, suppose a queue called q stored the following sequence of values:

front ["cat", "to", "c", "dog", "rat", "b", "am, "a", "d", "hat", "run", "of"] back

After the call separate(q), the queue should store the following sequence values (groups underlined for clarity):

front ["c", "b", "a", "d", "to", "am", "of", "cat", "dog", "rat", "hat", "run"] back
|----------------| |--------------| |-------------------------------|

Length 1 Length 2 Length 3

Notice that within the groups, the ordering is the same as the ordering of the original queue.

You may assume that the queue passed to your method is not null and that it does not contain any null values. You may assume all of the Strings in the queue are either length 1, 2, or 3. If the queue is empty, then no values should be added to the queue.

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 one stack 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 not 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 void s2q(Stack<String> s, Queue<String> q) {
    while (!s.isEmpty()) {
        q.add(s.pop());
    }
}
public void q2s(Queue<String> q, Stack<String> s) {
    while (!q.isEmpty()) {
        s.push(q.remove());
    }
}
```

You may use the rest of this page as scratch paper, but **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.

```
public static void separate(Queue<String> q) {
    Stack<String> s = new Stack<>();
    int size = q.size();
    // Move length 1 to stack
    for (int i = 0; i < size; i++) {
        String str = q.remove();
        if (str.length() == 1) {
            s.push(str);
        } else {
            q.add(str);
        }
    }
    size = q.size();
    // Move length 2 to stack
    for (int i = 0; i < size; i++) {
        String str = q.remove();
        if (str.length() == 2) {
            s.push(str);
        } else {
            q.add(str);
        }
    }
    // Move length 3 (the rest) to stack
    q2s(q, s);
    // Reverse contents of everything
    s2q(s, q);
    q2s(q, s);
    s2q(s, q);
}
```

(You may use the rest of this page as scratch paper if necessary)

^_^ 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 {
    private type field;
    public Example() {
       field = something;
    }
    public void method() {
       // do something
    }
}
```

```
public interface InterfaceExample {
    public void method();
}
```