CSE 122 Final Exam **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; write each answer in the provided boxes.

```
ArrayList<Integer> list = new ArrayList<>();
ArrayList<Integer> another = new ArrayList<>();
list.add(3);
list.add(-14);
another.add(0);
list = another;
ArrayList<Integer> list2 = another;
list2.remove(0);
```

Part A1: How many ArrayList **objects** are created in this program? Answer in the box below

2

Part A2: How many **references to objects** exist in this program? Answer in the box below

3

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 Set<String> mystery(Set<String> s, int n) {
    Set<String> s2 = new HashSet<>();
    // Point A
    if (n <= 0) {
        throw new IllegalArgumentException();
    }
    // Point B
    if (!s.isEmpty()) {
        Iterator<String> it = s.iterator();
        // Point C
        while (it.hasNext()) {
            // Point D
            if (it.next().length() > n) {
                s2.add(it.remove());
                n--;
            }
        }
    }
    // Point E
    return s2;
}
```

An explanation for Point A's answers

s.isEmpty(): **S**
True for mystery([], 3) and false for mystery(["hi"], 3). Therefore sometimes true.

s2.isEmpty(): **A**
s2 has just been created with no elements and no other methods have been called on it.

n > 0: **S**
True for mystery(["hi"], 3) and false for mystery(["hi"], -19). Therefore sometimes true.

	Point A	Point B	Point C	Point D	Point E
s.isEmpty()	**S**	**S**	**N**	**N**	**S**
s2.isEmpty()	**A**	**A**	**A**	**S**	**S**
n > 0	**S**	**A**	**A**	**S**	**S**

Part C: (Select one option) Consider the following code where m is a Map<String, Set<Integer>>. Which of the following options is the best "plain-English" explanation of what the code is doing?

```
List<String> l = new ArrayList<>();
for (String str : m.keySet()) {
      Iterator<Integer> iter = m.get(str).iterator();
      while (iter.hasNext()) {
            if (iter.next() < 0) {
                  iter.remove();
            }
      }
      if (m.get(str).isEmpty()) {
            l.add(str);
      }
}
for (String str : l) {
      m.remove(str);
}
```

◯ It produces a NoSuchElementException because next() is called on the iterator after all of the elements have been processed.

◯ Uses for each loops and iterators to go through each mapping of String to Set of ints in the Map, removes some of the ints in each set if they are negative, and adds some of the key Strings to another List.

✅ Removes all negative numbers from value Sets in the Map and removes all Map entries where the corresponding value Set only contained negative numbers.

◯ Removes all negative from each of the Set values in the Map and adds all empty Sets to a List.

2. **Code Tracing:** Consider the method below.

```
public static void mystery(int[][] a) {
    for (int i = a.length - 1; i > 0; i--) {
        for (int j = 0; j < a[i].length; j++) {
            a[i][j] = Math.max(a[i][j], a[i - 1][j]);
        }
    }
}
```

For each 2D array below, indicate what it would hold after a call to mystery where it was passed as a parameter. Notice that each 2D array is rectangular.

Before mystery call	**After mystery call**
[[0, 1, 2], [3, 4, 5]]	[[0, 1, 2], [3, 4, 5]]
[[4, 8], [23, 42], [15, 16]]	[[4, 8], [23, 42], [23, 42]]
[[-9, 8, 14]]	[[-9, 8, 14]]
[[5, 10, 15, 20], [12, 9, 6, 3], [6, 12, 18, 24]]	[[5, 10, 15, 20], [12, 10, 15, 20], [12, 12, 18, 24]]

3. **Debugging:** Consider the following buggy implementation of the **Inventory** class.

The intended functionality of the Inventory class is to represent an inventory of items, keeping tracking of the count of each item that is "in stock" with the following methods:

- A **constructor** that takes a map of items to their counts as a parameter and initializes the inventory with these counts of items.
- **getItemCount(String item)** which returns the number of the given item in the inventory, 0 if the item does not exist in the inventory.
- **remove(String item, int count)** which removes count items from the inventory and throws an IllegalArgumentException if the item does not exist in the inventory, if count is negative, or if there are not enough items to fulfill the remove request.
- **restockList()** which returns a list of items in the inventory whose count is 0.

```
 1. public class Inventory {
 2.     private Map<String, Integer> items;
 3.
 4.     public Inventory(Map<String, Integer> stuff) {
 5.         items = new HashMap<>();
 6.         for (String s : stuff.keySet()) {
 7.             items.put(s, stuff.get(s));
 8.         }
 9.     }
10.
11.     public int getItemCount(String item) {
12.         if (!items.containsKey(item)) {
13.             return 0;
14.         } else {
15.             return items.get(item);
16.         }
17.     }
18.
19.     public void remove(String item, int count) {
20.         if (!items.containsKey(item) || count < 0 || items.get(item) < count) {
21.             throw new IllegalArgumentException();
22.         }
23.         int numLeft = items.get(item) - count;
24.         items.put(item, numLeft);
25.     }
26.
27.     public List<String> restockList() {
28.         List<String> result = new ArrayList<>();
29.         for (String s : items.keySet()) {
30.             if (items.get(s) == 0) {
31.                 result.add(s);
32.             }
33.         }
34.         return result;
35.     }
36. }
```

There is a *single* bug in this program that is your task to find and fix.

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 20

Part B: Specifically mention which line(s) you will change and how to fix the error in the code above. 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.

Need to add another check in remove() so that an IllegalArgumentException is thrown if items.get(item) < count (there are not enough of an item to fulfill the remove request). This can be done by adding another test to the existing if statement with ||, or it could be a completely separate if statement *after* the first one if students are unsure about short-circuiting (we need to check that items.containsKey(item) before checking the item's count).

4. **Collections Programming**: Write a method called **favoriteStudySpots** that takes a map indicating how each person rates various study spots and a target rating and returns a map indicating all the study spots each person has rated with at least the target rating.

The input map will have keys that are people's names (strings) and values which are maps with keys that are study spots (strings) and values which are numbers in the range of 0.0 to 5.0 for the rating that person has given that study spot. An example would be if we had a variable called ratings that stored the following map in the format just described:

```
{"Karen"={"CSE2"=5.0, "ALB"=5.0, "OUG"=0.0},
 "Poojitha"={"NAN"=4.3, "MGH"=4.2},
 "Steven"={"SAV"=2.4},
 "Evelyn"={}}
```

In this example, we see that Karen has rated CSE2 and ALB as a 5.0 each and OUG as a 0.0, while Steven has only rated SAV as a 2.4.

The favoriteStudySpots method you are writing should take a ratings map described above and a target rating and should return a map indicating all the study spots each person has rated with at least the target rating. The map you are to return should use the people's names as keys and the set of all the study spots that person rated with at least the target value as values.

For example, suppose the following call is made:

```
favoriteStudySpots(ratings, 4.3);
```

Given this call, the following map would be returned:

```
{"Poojitha"=["NAN"],
 "Evelyn"=[],
 "Karen"=["ALB", "CSE2"],
 "Steven"=[]}
```

Notice that the value for the key "Karen" is the set ["CSE2", "ALB"] because she rated only those study spots with at least a rating of 4.3. The value for the key "Steven" is [] because Steven rated no study spots with a rating of at least 4.3. Note that study spots rated with a 4.3 should be included (see Poojitha).

The map you return should have keys sorted alphabetically and the study spots in the values should appear in alphabetical order as well. You may assume that the map and none of its contents are null.

Your method should construct the new map and each of the sets 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, Set<String>> favoriteStudySpots (Map<String,
Map<String, Double>> ratings, double target) {
   Map<String, Set<String>> result = new TreeMap<>();
   for (String person : ratings.keySet()) {
      result.add(person, new TreeSet<>());
      Map<String, Double> ratingsFor = ratings.get(person);
      for (String locn : ratingsFor.keySet()) {
         if (ratingsFor.get(locn) >= target) {
            result.get(person).add(locn);
         }
      }
   }
   return result;
}
```

5. **Objects Programming:** Consider the following Appointment interface. For this problem, write a class called GroomingAppointment that implements the Appointment interface and implements the required methods. The GroomingAppointment class should have a constructor that takes no parameters and sets up the necessary state.

```
// Represents an appointment with a date, time, and customer name
public interface Appointment {
    // Sets the appointment date/time to the given month, day, hour, and minute (e.g.,
    // setTime(1, 6, 12, 30) would be setting the appointment date to Jan 6 and
    // appointment time to 12:30. Throws an IllegalArgumentException if any of the
    // parameters are invalid (according to these simplified rules)
    // - month should be between 1 and 12 inclusive
    // - day should be between 1 and 31 inclusive
    // - hour should be between 1 and 12 inclusive
    // - min should be between 0 and 59 inclusive
    public void setDateTime(int month, int day, int hour, int min);

    // Sets the customer's name for the appointment to the given name.
    public void setName(String name);

    // Adds the given service to this appointment, including the amount of time this
    // service takes.
    public void addService(String service, int mins);

    // Returns the total amount of time for this appointment.
    public int totalAppointmentTime();

    // Returns a string representation of this appointment.
    // If the date/time or the customer name hasn't been set yet, returns
    //     "Appointment not yet scheduled"
    // Otherwise returns a String in the form of
    //     "<customer's name>'s appointment scheduled for <month>/<day> at <hour>:<min>"
    public String toString();

    // Returns true if the given Appointment has the same date and start time as this
    // Appointment; otherwise returns false.
    public boolean conflictsWith(Appointment other);
}
```

For example, if the following lines are executed:

```
GroomingAppointment appt1 = new GroomingAppointment();
appt1.setName("Gumball");
appt1.setDateTime(1, 25, 5, 0); // Jan 25, 5:00
appt1.addService("Bath", 25);
appt1.addService("Trim", 30);
GroomingAppointment appt2 = new GroomingAppointment();
appt2.setName("Willow");
appt2.addService("Nail trim", 10);
```

Then the following calls to toString would return:

```
appt1.toString();    "Gumball's appointment scheduled for 1/25 at 5:0"
appt2.toString();    "Appointment not yet scheduled"
```

Note that the time for appt1 is listed as "5:0" not "5:00" (so no special handling is required to make sure the minutes' value is shown with two digits in this case).

The following calls to totalAppointmentTime would return:

```
appt1.totalAppointmentTime();        55
appt2.totalAppointmentTime();        10
```

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

Write your solution on the next page.

```
public class GroomingAppointment implements Appointment {
    private Map<String, Integer> services;
    private int month;
    private int day;
    private int hour;
    private int minute;
    private String customerName;

    public GroomingAppointment(String name, int month, int day, int hour, int minute) {
        if (month < 1 || month > 12
            || day < 1 || day > 31
            || hour < 1 || hour > 24
            || minute < 0 || min > 59) {
            throw new IllegalArgumentException();
        }
        services = new HashMap<>();
        this.month = month;
        this.day = day;
        this.hour = hour;
        this.minute = minute
    }

    public void addService(String service, int mins) {
        services.put(service, mins);
    }

    public int totalAppointmentTime() {
        int total = 0;
        for (String service : services.keySet()) {
                total += services.get(service);
        }
        return total;
    }

    public String getDate() {
        return this.month + "/" + this.day;
    }

    public String getTime() {
        return this.hour + ":" + this.minutes;
    }

    public Set<String> getServices() {
        return services.keySet();
    }

    public String toString() {
        return this.name + "'s appointment scheduled for "
                + this.getDate() + " at " + this.getTime();
    }

    public boolean conflictsWith(Appointment other) {
        String otherDate = other.getDate();
        String otherTime = other.getTime();
        return otherDate.equals(this.getDate()) && otherTime.equals(this.getTime());
    }
}
```

6. **Stacks/Queues Programming:** Write a method named **expand** that accepts two stacks of integers of the same length. For each pair of numbers between the stacks, we will make copies of the values in the first stack, using the values in the second stack to determine the number of copies to make. This is clearer with an example.

Assume we had the following stacks, s1 and s2. In the rest of this description, bottom is denoted with a B while the top is denoted with a T.

```
s1:     B [4, 5, 6] T                    s2:     B [1, 2, 3] T
```

After the call expand(s1, s2), s1 and s2 should store the following values.

```
s1:     B [4, 5, 5, 6, 6, 6] T           s2:     B [1, 2, 3] T
```

The pairs are determined by being at the same position in each stack. In this example, the pairs are (6, 3), (5, 2), (4, 1) since they are in the same position in each stack. For each pair, we use the value in the second stack to determine how many values of the first stack to make; for the bottom pair, this means there should be one 4 in the result. This is why the result above shows the method modified s1 to store one 4, two 5s, and three 6s. The method should only modify the first stack and the second one is unchanged.

Alternatively, assume we had the original s1 and s2 but made the call expand(s2, s1). In this case the pairs will be (3, 6), (2, 5), (1, 4). This means the method should modify s2 to have four 1s, five 2s, and six 3s. The resulting state of both stacks is shown below (s1 is unchanged in this example since it is the second parameter).

```
s1:   B [4, 5, 6] T          s2:  B [1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3] T
```

The values should appear in the stack in the same relative order they appeared originally. The second stack parameter should not be modified after the call finishes. You may assume that the stacks passed to your method are not null and they do not contain any null values. You may assume all values in the second stack will be at least 1.

Your method should throw an IllegalArgumentException if the stacks are not the same length.

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 queue 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<Integer> s, Queue<Integer> q) {
    while (!s.isEmpty()) {
        q.add(s.pop());
    }
}

public void q2s(Queue<Integer> q, Stack<Integer> 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.

```
// Merge and split solution
public static void expand(Stack<Integer> s1, Stack<Integer> s2) {
```

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

```
// Merge and "decode" solution
public static void expand(Stack<Integer> s1, Stack<Integer> s2) {
    if (s1.size() != s2.size()) {
        throw new IllegalArgumentException();
    }
    Queue<Integer> q = new LinkedList<>();
    // Expand first and add everything into queue
    while (!s2.isEmpty()) {
        int n1 = s1.pop();
        int n2 = s2.pop();
        q.add(n2);
        for (int i = 0; i < n2; i++) {
            q.add(n1);
        }
    }
    // [3, 6, 6, 6, 2, 5, 5, 1, 4]
    while (!q.isEmpty()) {
        int val = q.remove();
        for (int i = 0; i < val; i++) {
            s1.push(q.remove());
        }
        s2.push(val);
    }
    s2q(s1, q);
    q2s(q, s1);
    s2q(s2, q);
    q2s(q, s2);
}

// "Juggling" solution
public static void expand(Stack<Integer> s1, Stack<Integer> s2) {
    if (s1.size() != s2.size()) {
        throw new IllegalArgumentException();
    }
    Queue<Integer> q = new LinkedList<>();
    // Start by putting copy counts in q
    s2q(s2, q);

    // Don't need to merge them into same structure now since we can loop over a Q + S1, using S2
as storage
    int size = q.size();
    for (int i = 0; i < size; i++){
        int n1 = s1.pop();
        int n2 = q.remove();
        for (int j = 0; j < n2; j++) {
            s2.push(n1);
        }
        // could also merge in Stack
        q.add(n2);
    }

    // Put everything back in the right structure in the right order
    while (!s2.isEmpty()) {
        s1.add(s2.pop());
    }
    q2s(q, s2);
    s2q(s2, q);
    q2s(q, s2);
}
```

^_^ 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();
}
```