CSE 122 Final Exam Solutions
Spring 2023

Name of Student: ____

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 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 uses the Point.java implementation from class, also included here for convenience:

```
public class Point {
    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public Point() {
        this(0, 0);
    }

    public void translate(int dx, int dy) {
        x += dx;
        y += dy;
    }

    public double distanceToOrigin() {
        return Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2));
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public String toString() {
        return "(" + this.x + ", " + this.y + ")";
    }
}
```

Part A: Consider the following code snippet.

```
    Point p1 = new Point(2, 3);
    Point p2 = new Point();
    System.out.println(p1);
    Point p3 = p1;
    p3.translate(2, 4);
    p3 = p2;
    p2 = p1;
    p3.translate(1, 1);
    System.out.println(p2);
    System.out.println(p3);
```

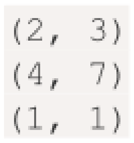
Part A1: How many Point **objects** are created in this snippet? Answer in box below

2

Part A2: How many **references to objects** are created in this snippet? Answer in box below

3

Part B: What is printed to the console by the code snippet from Part A?



```
(2, 3)
(4, 7)
(1, 1)
```

Part C: (Select one option) We plan to add the following method to the Point.java class listed above. Which of the options below would make the best comment for this new method of our Point.java class?

```
public int mystery(List<Point> list) {
   Iterator<Point> iter = list.iterator();
   int n = 0;
   while (iter.hasNext()) {
      Point p = iter.next();
      if (p.distanceToOrigin() < this.distanceToOrigin()) {
          iter.remove();
          n++;
      }
   }
   this.x += n;
   return n;
}
```

○ Removes some points from the given list.

○ Removes all points from the given list which are further from the origin than this point. Returns the number of removed points and moves this point right by the same number.

[✓ Removes all points from the given list which are closer to the origin than this point. Returns the number of removed points and moves this point right by the same number.]

○ Uses an iterator to remove all points from the given list which are closer to the origin than this point. Updates this.x to be larger by the number of removed points, also returns the number of removed points.

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

```
public static void mystery3(int[][] arr) {
    for (int i = 0; i < arr.length; i++) {
        for (int j = 0; j < arr[i].length; j++) {
            if (arr[i][j] <= arr[j][i]) {
                arr[i][j] *= 2;
            }
        }
    }
}
```

For each 2d array below, indicate what it would hold after a call to mystery3 where it was passed as a parameter.

```
Before mystery3 call                        After mystery3 call


[[2]]                                                 [[4]]


[                                                     [[2, 4],
 [1, 2],                                              [5, 12]]
 [5, 6]
]


[
 [1, 2, 3],
 [2, 0, 1],                                           [[2, 4, 6],
 [5, -2, 0]                                            [4, 0, 1],
]                                                      [10, -4, 0]]
```

3. **Debugging:** Consider the following buggy implementation of **divide.** The intended behavior of this method is to take a list of non-negative integers, and modify that list so that each number is divided into two numbers that sum up to the original. If the number is even it should be split into halves, if a number is odd, the larger half should appear first.

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

    [18, 7, 4, 24, 11]

The number 18 is divided into the pair (9, 9), the number 7 is divided into (4, 3), the number 4 is divided into (2, 2), the number 24 is divided into (12, 12) and the number 11 is divided into (6, 5). Thus, the call: divide(list); should cause list to store the following sequence of values afterwards:

    [9, 9, 4, 3, 2, 2, 12, 12, 6, 5]

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

```
1.  public static void split(List<Integer> list) {
2.     for (int i = 0; i < list.size(); i++) {
3.          int n = list.get(i);
4.          list.add(i, n / 2 + n % 2);
5.          list.add(i + 1, n / 2);
6.      }
7.  }
```

Your task is to fix this implementation so that it behaves as described above. If you are making significant changes to the structure of the method, it may be helpful to write your whole solution from scratch. However, if you are only making minor edits to the code that you can clearly explain, you can also write out just the edits below. If writing edits, specifically mention which line(s) you will change and write out the code you would replace them with. You will need to write correct code on the lines you change/add. 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).

Bug 1: The loop increment is incorrect as the loop body adds new elemtns to the list. One potential fix: Change loop bound.

```
for (int i = 0; i < list.size(); i += 2) {
```

Bug 2: Adds two elements to the list without removing any.

One potential fix: Change the first `add` call to a `set` call.

```
list.set(i, n / 2 + n % 2);
list.add(i + 1, n / 2);
```

Another potential fix is to remove the value at index `i` after adding the split values.

```
list.add(i, n / 2 + n % 2);
list.add(i + 1, n / 2);
   list.remove(i + 2);
```

4. **Collections Programming**: Write a method called **split** that takes a set of strings as a parameter and that returns the result of splitting the strings into different sets based on the length of the strings. In particular, your method should return a map whose keys are integers and whose values are sets of strings of that length. For example, if a variable called words contains the following set of strings:

```
    [to, be, or, not, that, is, the, question]
```

then the call split(words) should return a map whose values are sets of strings of equal length and whose keys are the string lengths:

```
    {2=[be, is, or, to], 3=[not, the], 4=[that], 8=[question]}
```

Notice that strings of length 2 like "be" and "is" appear in a set whose key is 2. If the words set had instead stored these strings:

```
    [four, score, and, seven, years, ago, our, fathers, brought, forth]
```

Then the method would return this map:

```
    {3=[ago, and, our], 4=[four], 5=[forth, score, seven, years],
     7=[brought, fathers]}
```

The set of strings passed to your method will not necessarily be in order, but the map returned by your method should be ordered numerically by key and each set contained in the map should be ordered alphabetically, as in the examples above.

Your method should construct the new map and each of the sets contained in the map but should otherwise not construct any new data structures. It should also not modify the set of words passed as a parameter. You should use interface types and generics appropriately.

Write your solution on the next page

One possible solution is shown below.

```
public static Map<Integer, Set<String>> split(Set<String> set) {
    Map<Integer, Set<String>> result = new TreeMap<>();
    for (String s : set) {
        if (!result.containsKey(s.length())) {
            result.put(s.length(), new TreeSet<>());
        }
        result.get(s.length()).add(s);
    }
    return result;
  }
```

5. **Objects Programming:** Consider the following interface Media. For this problem you are to write a class called Book which implements the Media interface. The Book class should have a constructor which takes one String parameter, the book’s title.

```
public interface Media {
    // Returns the name of this piece of media, eg: the title of a movie or book.
    public String getName();

    // Adds a new rating of this media. Reviewer is the name of the person giving this
    // rating. Reviewers may only give a single rating to a piece of media.
    // Throws an IllegalArgumentException if stars is not between 1 and 5 inclusive, or
    // if reviewer has already reviewed this media.
    public void addRating(String reviewer, int stars);

    // Change the rating given by a reviewer who has already reviewed this media.
    // Throws an IllegalArgumentException if newStars isn’t between 1 and 5 inclusive, or
    // if reviewer hasn’t reviewed this media before. (see addRating for first-time
    // reviewers).
    public void changeRating(String reviewer, int newStars);

    // Returns the average of all ratings this media has received.
    public double averageRating();

    // Returns true if this piece of media has an equal or higher average rating than
    // other. Returns false if other has a higher average rating.
    public boolean isRatedHigher(Media other);

    // Returns a string representation of this piece of media and it’s reviews. The format
    // is as follows:
    //     <media name> - rated <average rating> out of 5 stars
    // Eg, for a piece of media titled “Sharknado” with a 2 star review and a 5 star
    // review, the resulting toString would look like:
    //     Sharknado - rated 3.5 out of 5 stars
    // Note that the average value is not rounded.
    public String toString();
}
```

For example, if the following lines were executed using your Book class...

```
    Media book1 = new Book(“All Systems Red”);
    book1.addRating(“Tristan”, 5);
    book1.addRating(“Hunter”, 3);
    book1.changeRating(“Hunter”, 2);
    Media book2 = new Book(“Deadly Education”);
    book2.addRating(“Miya”, 4);
    book2.addRating(“Tristan”, 2);
    book2.changeRating(“Tristan”, 5);
```

Then the following methods would return...

```
    book1.getName();             // All Systems Red
    book1.averageRating();       // 3.5
    book2.averageRating();       // 4.5
    book1.toString();            // All Systems Red - rated 3.5 out of 5 stars
    book2.toString();            // Deadly Education - rated 4.5 out of 5 stars
    book1.isRatedHigher(book2);  // false
```

Your Book class should implement the Media interface. Your Book class should have private fields and implement the methods described above.

Write your solution on the next page.

One possible solution is shown below.

```
public class Book implements Media {
    private Map<String, Integer> ratings; // Or other structure is okay
    private String name;

    public Book(String name) {
        this.name = name;
        this.ratings = new HashMap<>();
    }

    public String getName() {
        return this.name;
    }

    public void addRating(String reviewer, int stars) {
        if (stars < 1 || stars > 5 || this.ratings.containsKey(reviewer)) {
            throw new IllegalArgumentException();
        }
        ratings.put(reviewer, stars);
    }

    public void changeRating(String reviewer, int stars) {
        if (stars < 1 || stars > 5 || !this.ratings.containsKey(reviewer)) {
            throw new IllegalArgumentException();
        }
        ratings.put(reviewer, stars);
    }

    public double averageRating() {
        // Can also store in fields
        int total = 0;
        for (int rating : ratings.values()) {
            total += rating;
        }
        return (double) total / ratings.size();  // Ignore 0 case
    }

    public boolean isRatedHigher(Media other) {
        return this.averageRating() >= other.averageRating();
    }

    public String toString() {
        return this.name + " - " + this.averageRating() + " out of 5 stars";
    }
  }
```

6. **Stacks/Queues Programming:** Write a method called **compressDuplicates** that takes a stack of integers as a parameter and that replaces each sequence of duplicates with a pair of values representing a count of the number of duplicates followed by the number. For example, suppose a variable called s stores the following sequence of values:

```
    bottom [2, 2, 2, 2, 2, -5, -5, 3, 3, 3, 3, 4, 4, 1, 0, 17, 17] top
```

and we make the following call:

```
    compressDuplicates(s);
```

Then s should store the following values after the call:

```
    bottom [5, 2, 2, -5, 4, 3, 2, 4, 1, 1, 1, 0, 2, 17] top
```

This new stack indicates that the original had 5 occurrences of 2 at the bottom of the stack followed by 2 occurrences of -5 followed by 4 occurrences of 3, and so on. This process works best when there are many duplicates in a row. For example, if the stack instead had stored:

```
    bottom [10, 20, 10, 20, 10, 20] top
```

Then the resulting stack ends up being longer than the original:

```
    bottom [1, 10, 1, 20, 1, 10, 1, 20, 1, 10, 1, 20] top
```

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

    public static 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.

Two possible solution are shown below.

```java
// Nested loop solution:
public static void compressDuplicates(Stack<Integer> s) {
    Queue<Integer> q = new LinkedList<>();
    while (!s.isEmpty()) {
        int count = 1;
        int val = s.pop();
        while (!s.isEmpty() && s.peek() == val) {
            count++;
            s.pop();
        }
        q.add(val);
        q.add(count);
    }

    q2s(q, s);
    s2q(s, q);
    q2s(q, s);
}

// Single loop solution:
public static void compressDuplicates(Stack<Integer> s) {
    if (s.isEmpty()) {
        return;
    }
    Queue<Integer> q = new LinkedList<>();
    int prior = s.pop();
    int count = 1;
    while (!s.isEmpty()) {
        int curr = s.pop();
        if (curr == prior) {
            count++;
        } else {
            q.add(prior);
            q.add(count);
            prior = curr;
            count = 1;
        }
    }
    q.add(prior);
    q.add(count);

    q2s(q, s);
    s2q(s, q);
    q2s(q, s);
  }
```

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