CSE 122 Practice Final
Spring 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 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 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 like 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 the these problems should be considered independent of the others

Part A: (Select all that apply) Which of these statements are true about Sets in Java?

- ☐ A Set **does not** allow duplicate elements
- ☐ A LinkedList is an implementation of a Set
- ☐ A Set **does not** preserve the insertion order of its elements
- ☐ A Set **will** preserve the the insertion order of its elements if the TreeSet implementation is used

Part B: (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 int method(int[][] a, int n) {
    int count = 0;
    for (int i = 0; i < a.length; i++) {
        for (int j = 0; j < a[i].length; j++) {
            if (a[i][j] + a[a.length - i - 1][j] == n) {
                count++;
            }
        }
    }
    return count;
}
```

- ◯ Returns the numbered of “mirrored pairs” in a that sum up to the given value n. A “mirrored pair” of a value is the one in the opposite side of the same **row.**
- ◯ Returns the numbered of “mirrored pairs” in a that sum up to the given value n. A “mirrored pair” of a value is the one in the opposite side of the same **column.**
- ◯ Uses a nested for loop to find the number of values in the 2D array a where a[i][j] + a[a.length - i - 1][j] == n.
- ◯ Finds the number of values in a 2D array a that match some condition based on n.

Part C: Consider the following method

```
 1. public void method(Map<String, String> m) {
 2.     for (String key : m.keySet()) {
 3.         if (m.get(key).equals(key)) {
 4.             System.out.println(m.get(key));
 5.         } else if (m.get(key).startsWith(key)) {
 6.             System.out.println(key);
 7.         } else {
 8.             System.out.println(m.get(key));
 9.         }
10.     }
11. }
```

Part C-1: Provide a map that, if passed to the above method, would result in line 4 being executed at least once and line 8 never being executed.

Part C-2: Provide a map that, if passed to the above method, would result in line 8 being executed at least once and line 4 and 6 never being executed.

2. **<u>Code Tracing:</u>** Write the output that is printed when the given method below is passed each of the following maps as its parameter. Assume that each parameter map stores its key/value pairs in exactly the order shown, and that is the order in which a for-each loop would examine them. Recall that maps print in a *{key1=value1, key2=value2, ..., keyN=valueN}* format. Your answer should display the right values in the right order.

```
public void mapMystery(Map<String, String> m) {
    Set<String> s = new TreeSet<>();
    for (String key : m.keySet()) {
        if (!m.get(key).equals(key)) {
            s.add(m.get(key));
        } else {
            s.remove(m.get(key));
        }
    }
    System.out.println(s);
}
```

For each call below, indicate what output is produced:

Method Call	**Output Produced**
{sheep=wool, house=brick, cast=plaster, wool=wool}	______________________
{munchkin=blue, winkie=yellow, corn=yellow, grass=green, emerald=green}	______________________
{pumpkin=peach, corn=apple, apple=apple, pie=fruit, peach=peach}	______________________
{lab=ipl, lion=cat, terrier=dog, cat=cat, platypus=animal, nyan=cat}	______________________

3. **<u>Debugging:</u>** Consider the following buggy implementation of **mostFrequentLength.**

The intended behavior of this method is to take a list of strings, and return the most frequent length of the strings in the list. In the case of a tie, it should return the smaller length. If the list of words is empty, should throw an IllegalArgumentException;

```
 1.  public int mostFrequentLength(List<String> words) {
 2.     if (words.isEmpty()) {
 3.         throw new IllegalArgumentException();
 4.     }
 5.     Map<Integer, Integer> counts = new TreeMap<>();
 6.     for (String word : words) {
 7.         counts.put(word.length(), counts.get(word.length()) + 1);
 8.     }
 9.     int maxLength = -1;
10.     int maxCount = -1;
11.     for (int length : counts.keySet()) {
12.         int count = counts.get(length);
13.         if (count > maxCount) {
14.             maxLength = length;
15.             maxCount = count;
16.         }
17.     }
18.     return maxLength;
19.  }
```

There is a *single* bug in this program that is your task to find and fix. As a hint, when running this code on a simple test case

```
["dogs", "are", "better", "than", "cats"]
```

The method crashes and produces the following error:

```
Exception java.lang.NullPointerException: Cannot invoke "java.lang.Integer.intValue()"
because the return value of "java.util.Map.get(Object)" is null
```

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

Part A Answer

Part B: Fix the error in the method above. Since there is only one bug, this should take very few lines of code to fix. Specifically mention which lines 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 numbers. Write your answer for **Part B** in the box below.

4. **Collections Programming**: Write a method **removeShorterStrings** that takes a list of Strings as a parameter and that removes from each successive pair of values the shorter string in the pair. For example, suppose that an list called list contains the following values:

```
    ["four", "score", "and", "seven", "years", "ago"]
```

In the first pair, "four" and "score", the shorter string is "four". In the second pair, "and" and "seven", the shorter string is "and". In the third pair, "years" and "ago", the shorter string is "ago".

Therefore, the call:

```
    removeShorterStrings(list);
```

Should remove these shorter strings, leaving the list as follows:

```
    ["score", "seven", "years"]
```

If there is a tie (both strings have the same length), your method should remove the first string in the pair. If there is an odd number of strings in the list, the final value should be kept in the list.

5. **Objects Programming:** Consider the following ItemListing interface used at Amazon to represent an item they have in their inventory. For this problem, you should write a class called GeneralItem that implements the ItemListing interface to implement the required methods. The GeneralItem should also have a constructor that takes a name (String) and a brand name (String). In the description of the toSTring below, you should not include the < and > characters in the returned value.

```
// Represents an item represented on Amazon's store. Provides details about the name
// and brand of the item, as well as a way to rate the item and get its average rating.
public interface ItemListing {
    // Adds the given rating (0.0 - 5.0) to this item
    public void review(double rating);

    // Returns the name of this item.
    public String getItemName();

    // Returns the brand of this item.
    public String getBrand();

    // Returns the average of all ratings this item has received.
    // If this item has received no ratings, returns 0.0.
    public double getRating();

    // Returns a string representation of this ItemListing.
    // If there are zero or multiple reviews for this item, will return:
    //    <itemName> (<brand>) rates <averageRating> (<numberOfReviews> reviews)
    // If there is only a single review, will return:
    //    <itemName> (<brand>) rates <averageRating> (<numberOfReviews> review)
    // Does not round or truncate the average rating. If there aren't any ratings for this
    // item, then the average rating should be 0 and it should still return (0 reviews) as
    // part of the string.
    public String toString();

    // Returns true if the given ItemLiting is considered the same item.
    // Two ItemListings are the same if they both have the same name and
    // same brand.
    public boolean isDuplicate(ItemListing other);

}
```

Below shows an example usage of this class.

```
    ItemListing item = new GeneralItem("Vacuum", "Dyson");
    item.review(4.7);
    item.review(5);
    item.review(4.9);
    item.review(4.9);

    item.getRating();  // Should return 4.875
    item.toString();   // Should return "Vacuum (Dyson) rates 4.875 (4 reviews)"
```

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

Write your solution on the next page.

6. **Stacks/Queues Programming:** Write a method **reorder** that takes a queue of integers as a parameter and that puts the integers into sorted (nondecreasing) order assuming that the queue is already sorted by absolute value. For example, suppose that a variable called q stores the following sequence of values:

```
front [1, 2, -2, 4, -5, 8, -8, 12, -15, 23] back
```

Notice that the values appear in sorted order if you ignore the sign of the numbers. The following call:

```
reorder(q);
```

should reorder the values so that the queue stores this sequence of values:

```
front [-15, -8, -5, -2, 1, 2, 4, 8, 12, 23] back
```

Notice that the values now appear in sorted order taking into account the sign of the numbers.

You are to use one stack as auxiliary storage to solve this problem. You may not use any other auxiliary data structures to solve this problem, although you can have as many simple variables as you like. You should use the Queue interface and LinkedList and Stack implementations as discussed in class, and limit your solution to only use the methods shown on the provided reference sheet. You should not use any advanced material to solve this problem such as recursion.

You may assume the following s2q and q2s helper methods have already been implemented as shown below (flattened to one line body to save space).

```
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 (!s.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.

(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 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();
}
```