CSE 122 Practice Final Exam
Summer 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 Hospital.java implementation included here.

```
public class Hospital {
    private String name;
    private int numDoctors;
    private int maxPatients;

    public Hospital (String name, int numDoctors, int maxPatients) {
        this.name = name;
        this.numDoctors = numDoctors;
        this.maxPatients = maxPatients;
    }

    public Hospital (String name) {
        this(name, 0, 0);
    }

    public void expand (int newNumDocs, int newMaxPatients) {
        this.numDoctors = newNumDocs;
        this.maxPatients = newMaxPatients;
    }

    public double patientDoctorRatio() {
        if (this.numDoctors == 0) {
            return 0;
        }
        return (double) maxPatients / numDoctors;
    }

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

    public int getNumDoctors() {
        return this.numDoctors;
    }

    public int getMaxPatients() {
        return this.maxPatients;
    }

    public String toString() {
        return this.name + " {Num Doc: " + this.numDoctors + ", Max Pat: " +
        this.maxPatients + "}";
    }
}
```

Part A: Consider the following code snippet.

```
Hospital h1 = new Hospital("Seattle Childrens", 200, 1200);
Hospital h2 = new Hospital("Overlake");
System.out.println(h1);
Hospital h3 = h1;
h3.expand(300, 1800);
h3 = h2;
h2 = h1;
h3.expand(500, 1500);
System.out.println(h2);
System.out.println(h3);
```

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

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

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

Part C: (Select one option) Suppose we were to write the following method. Which of the options below would make the best summary for this new method of our Hospital.java class?

```
public Hospital mystery(List<Hospital> list) {
    double a = 0.0;
    Hospital b;
    for (Hospital h : list) {
        double c = h.patientDoctorRatio();
        if (c >= a) {
            a = c;
            b = h;
        }
    }
    return b;
}
```

- ◯ Updates each Hospital in the list to have a higher Patient-Doctor Ratio
- ◯ Checks and returns the Hospital that is tied for the same Patient-Doctor Ratio
- ◯ Calculates the Hospital in the list that has the highest Patient-Doctor Ratio
- ◯ Calculates the Hospital in the list that has the lowest Patient-Doctor Ratio

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

```
public static List<Integer> mystery(int[][] data) {
    List<Integer> result = new ArrayList<>();
    for (int i = 1; i < data.length; i++) {
        for (int j = data[i].length - 1; j > 0; j--) {
            result.add(data[i][j] - 1);
        }
    }
    return result;
}
```

For each 2d array below, indicate in the right-hand column what values would be stored in the list returned by method mystery if the array in the left-hand column is passed as a parameter to mystery. List elements should be listed in proper order as a comma-separated bracketed list, as in [3, 18, 25].

Input 2D Array	**Contents of List Returned**
`[[0, 1],` `[2, 3]]`	_____

`[[0, 1, 2],`
`[3, 4, 5],`
`[6, 7, 8]]`
	_____

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

3. **Debugging:** Consider the following buggy implementation of **rotateLeftAndNegateEvens**. The intended behavior of this method is to take a list of integers and integer steps and modify that list so that the numbers are rotated left by the specified number of steps. Additionally, after the rotation, if the number is even and it moves from the front to the back, then it should be negated.

For example, if a variable called list stores this sequence of values and int steps = 2:

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

We want to rotate the numbers within this list to the left by two steps. Additionally, we want to negate the even numbers that move from the front to the back. If the method gets is passed a negative value for steps or if the list is empty, then the method should throw an IllegalArgumentException(). So, our expected output is:

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

Notice that 2 became -2 because it moved from the front to the back.

A TA wrote a buggy implementation of this method shown below. HINT: There are **2** bugs.

```
1.  public static void rotateLeftAndNegateEvens(List<Integer> list, int steps) {
2.      if (steps < 0 || list.length() == 0) {
3.          throw new IllegalArgumentException();
4.      }
5.      for (int i = 0; i < steps; i++) {
6.          int valFirst = list.remove(i);
7.          if (valFirst % 2 == 0) {
8.              list.add(valFirst * -1);
9.          } else {
10.             list.add(valFirst);
11.         }
12.     }
13. }
```

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).

4. **Collections Programming**: Write a method called **studentsTaught** that takes a map indicating each student's enrollment history and an instructor's name and returns a set indicating all students that the given instructor has taught.

The input map will have keys that are names of students (strings) and corresponding values which are maps representing the classes the student has taken. The value map maps the course code (integers) to the name of the instructor (strings) they took the class with. For example, if a variable called m stored the following map in the format described above:

```
{Colton={163=Kevin},
 Darel={416=Hunter, 373=Kevin, 143=Kevin},
 Ben={373=Kevin, 143=Stuart},
 Atharva={121=Miya, 122=Hunter, 123=Brett}}
```

Then a call to studentsTaught(m, "Kevin") should return a Set with the following elements:

```
[Ben, Colton, Darel]
```

A call to studentsTaught(m, "Elba") should return an empty Set, since none of the students in m took a course with her.

Notice that some instructors teach multiple courses and the same course may be taught by different instructors (in different quarters, for example).

The set you return should be sorted alphabetically. You may assume that the given map and none of its contents are null.

Your method should construct the new set to return but should otherwise not construct any other new data structures. Your method should not modify the given Map. You should use interface types and generics appropriately.

Write your solution on the next page

5. **Objects Programming:** Consider the following interface Restaurant. For this problem, you are to write a class called FastFoodRestaurant, which implements the Restaurant. The FastFoodRestaurant class should have two constructors. The first constructor should take a String name and the associated cuisine would be "N/A". The second constructor should take two parameters (String name and String cuisine).

```
public interface Restaurant {
    // Returns the name of the Restaurant
    public String getName();

    // Returns the cuisine of the Restaurant (For example: Thai or Indian)
    public String getCuisine();

    // Returns a list of items on the menu.
    public List<String> getMenu();

    // Adds a food item to the menu. If the food item is already present, then there is no
    // change.
    public void addFoodItem(String foodName);

    // Removes a food item from the menu.
    // Throws an IllegalArgumentException if the food item does not exist.
    public void removeFoodItem(String foodName);

    // Returns the number of food items on the menu.
    public int getNumFoodItems();

    // Makes a reservation at the particular restaurant. If a restaurant doesn't take
    // reservations (ex. Fast Food restaurants), then it returns "No reservation needed!".
    // If a restaurant accepts reservations, it returns "Success" or "Failure".
    public String makeReservation(String time);

    // Returns true if this restaurant has more items than the other restaurant
    // otherwise false.
    public boolean hasMoreOptions(Restaurant other);
}
```

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

```
Restaurant r1 = new FastFoodRestaurant("McDonalds");
r1.addFoodItem("Butter Paneer");
r1.addFoodItem("Cheeseburger");
r1.removeFoodItem("Butter Paneer");
Restaurant r2 = new FastFoodRestaurant("Chipotle", "Mexican");
r2.addFoodItem("Tacos");
r2.addFoodItem("PadThai");
r2.addFoodItem("Burrito");
r2.addFoodItem("Burrito");
r2.removeFoodItem("PadThai");
```

Then, the following methods would return...

```
r1.getName()                    // McDonalds
r1.getCuisine()                 // N/A
r1.getMenu()                    // [Cheeseburger]
r1.getNumFoodItems()            // 1
r2.getName()                    // Chipotle
r2.getCuisine()                 // Mexican
r2.getMenu()                    // [Tacos, Burrito] -> order does not matter
r2.getNumFoodItems()            // 2
r1.hasMoreOptions(r2)           // false
r1.makeReservation("8:00 pm")   // No reservation needed!
```

Your FastFoodRestaurant class should implement the Restaurant Interface. Your FastFoodRestaurant class should have private fields and should implement the above-outlined public methods. **Write your solution on the next page.**

6. **Stacks/Queues Programming:** Write a method called **mirrorSplit** that takes a stack of integers as a parameter and that splits each value into two halves, adding new values to the stack in a mirror position. For example, suppose that a stack s stores the following values:

```
bottom [14, 20, 8, 12] top
```

and we make the following call:

```
mirrorSplit(s);
```

Then s should store the following values after the call:

```
bottom [7, 10, 4, 6, 6, 4, 10, 7] top
        ^   ^  ^  ^  ^  ^   ^  ^
        |   |  |  +--+  |   |  |
        |   |  +--------+   |  |
        |   +---------------+  |
        +----------------------+
            mirror positions
```

The first value 14 has been split in half into two 7s which appear in mirror positions (first and last). The second value 20 has been split in half into two 10s which appear in mirror positions (second and second-to-last). And so on. This example included just even numbers in which case you get a true mirror image. If the stack contains odd numbers, they should be split so as to add up to the original with the larger value appearing closer to the bottom of the stack. For example, if the stack stores these values:

```
bottom [13, 5, 12] top
```

After the call, it would store the following values:

```
bottom [7, 3, 6, 6, 2, 6] top
```

The first value 13 has been split into 7 and 6 with the 7 included as the first value and 6 included as the last value. The value 5 has been split into 3 and 2 with 3 appearing as the second value and 2 appearing as the second-to-last value. And so on.

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.

(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();
}
```