

## CIS 1100 — Fall 2023 — Exam 2

Full Name: \_\_\_\_\_

Recitation #: \_\_\_\_\_

PennID (e.g. 12345678): \_\_\_\_\_

My signature below certifies that I have complied with the University of Pennsylvania's Code of Academic Integrity in completing this examination.

\_\_\_\_\_  
Signature

\_\_\_\_\_  
Date

Instructions are below. Not complying will lead to a 0% score on the exam.

- Do not open this exam until told by the proctor.
- You will have exactly 60 minutes to take this exam.
- Make sure your phone is turned OFF (not on vibrate!) before the exam starts.
- Food and gum are not permitted—don't be noisy or messy.
- You may not use your phone or open your bag for any reason, including to retrieve or put away pens or pencils, until you have left the exam room.
- This exam is closed-book, closed-notes, and closed computational devices.
- If you get stuck on a problem, it may be to your benefit to move on to another question and come back later.
- All code must be written in proper Java format, including all curly braces and semicolons.
- Do not separate the exam pages. Do not take any exam pages with you. The entire exam packet must be turned in as is.
- Only answers on the FRONT of pages will be graded. There are two blank pages at the end of the exam if you need extra space for any graded answers.
- Use a pencil, or blue or black pen to complete the exam.
- If you have any questions, raise your hand and a proctor will come to you.
- When you turn in your exam, you may be required to show your PennCard. If you forgot to bring your ID, talk to an exam proctor immediately.
- We wish you the best of luck!

Q1	Q2	Q3	Q4	Q5	Bonus

## Q1. Records

### Q1.1 Acceptable Record Syntax

Consider the following record definition written in **ExamScore.java**:

```
public record ExamScore(String studentName, double score, int examNumber) {}
```

In a separate file, **ScoreAnalysis.java**, I define a new ExamScore record like so:

```
ExamScore es = new ExamScore("Goia", 95.4, 2);
```

I want to analyze this and other ExamScore records in ScoreAnalysis.java. For each line, identify if it compiles or does not compile. There is at least one line that compiles and at least one line that doesn't compile.

Line	Compiles Successfully?
es.score = es.score() + 1.0;	no
es.score() = es.score() + 1.0;	no
System.out.println(es.studentName());	yes
System.out.println(es.getExamNumber());	no
ExamScore copy = new ExamScore(es);	no
ExamScore copy = new ExamScore(es.studentName(), es.score(), es.examNumber());	yes

**Q1.2** Reading & Summarizing Functions with Records

Here is a function called **mystery** that is written inside of `ScoreAnalysis.java`. Answer some questions about it.

```
public static ExamScore[] mystery(ExamScore[] all, double threshold) {
    int count = 0;
    for (int i = 0; i < all.length; i++) {
        if (all[i].score() <= threshold) {
            count++;
        }
    }
    ExamScore[] out = new ExamScore[count];
    int idx = 0;
    for (int i = 0; i < all.length; i++) {
        if (all[i].score() <= threshold) {
            out[idx] = all[i];
            idx++;
        }
    }
    return out;
}
```

Does this function modify its inputs in any way?

- 
- Yes
- 
- 
- No

Describe the output of **mystery** in 30 words or fewer. You do not need to explain how it works, just describe what the output is in terms of the inputs.

*The function **mystery** returns...*

*an array of ExamScore records that contains the ExamScore records in *all* with score fields less than or equal to the *threshold**

Complete the following unit test so that it correctly passes for the specified inputs to **mystery**.

```
@Test    // todo: complete rest of test & fill in assertion statement
public void testMystery() {
    ExamScore one = new ExamScore("Harry", 88, 1);
    ExamScore two = new ExamScore("Harry", 74, 2);
    ExamScore finalExam = new ExamScore("Harry", 30.1, 3);
    ExamScore[] inputs = {one, two, finalExam};
    double inputThreshold = 74.0;
    ExamScore[] output = ScoreAnalysis.mystery(inputs, inputThreshold);

    ExamScore[] expected = {two, finalExam};

    assertEquals(expected, output);
}
```

### Q1.3 Replacing Arrays with Lists

Rewrite **mystery()** so that it returns an `ArrayList` of `ExamScore` records instead of an array. The updated signature is written here for you. You can do this without being able to answer Q1.2. An overview of List methods can be found at the end of the exam in the Appendix. Assume that all correct import statements have been made at the top of the file.

```
public static ArrayList<ExamScore> mysteryRewrite(ArrayList<ExamScore> all, double threshold) {
    ArrayList<ExamScore> out = new ArrayList<ExamScore>();

    for (ExamScore es : all) {
        if (es.score() <= threshold) {
            out.add(es);
        }
    }

    return out;
}
```

## Q2. Unit Tests & Debugging

### Q2.1

Here is a buggy implementation of a function inside the file **Exam.java** that is supposed to find and return the longest **String** in an array. If the array is null or empty, then the function *should* return the empty string (""). If there are multiple Strings that share the longest length, the function should return the one with the lowest index in the array.

```
1 public static String findLongest(String[] arr) {
2     if (arr == null || arr.length <= 1) {
3         return "";
4     }
5     String longest = arr[0];
6     for (int i = 1; i < arr.length; i++) {
7         if (arr[i].length() > longest.length()) {
8             longest = arr[i];
9         }
}
```

```

10    }
11    return longest;
12 }

```

Consider the following four unit tests and **mark the names of the ones that fail.**

<input type="checkbox"/> testOne	<input checked="" type="checkbox"/> testTwo	<input checked="" type="checkbox"/> testThree	<input type="checkbox"/> testFour
<pre> @Test public void testOne() {     String[] inputs = {"int", "double", "char"};     String expected = "double";     String actual = Exam.findLongest(inputs);     assertEquals(expected, actual); } </pre>		<pre> @Test public void testThree() {     String[] inputs = {"one", "two", "three"};     String expected = "one";     String actual = Exam.findLongest(inputs);     assertEquals(expected, actual); } </pre>	
	<pre> @Test public void testTwo() {     String[] inputs = {"int"};     String expected = "int";     String actual = Exam.findLongest(inputs);     assertEquals(expected, actual); } </pre>	<pre> @Test public void testFour() {     String[] inputs = {""};     String expected = "";     String actual = Exam.findLongest(inputs);     assertEquals(expected, actual); } </pre>	

**Q2.2** Finally, fix the implementation of findLongest. Write the number of the **single** line that needs to be fixed and then write the fixed version below.

Line Number	Fixed Line
2	<code>if (arr == null    arr.length &lt; 1) {</code>

### Q3. Writing Objects

The following is an implementation of the class **Stitches.java**, which is written to be part of an app to help a person keep track of their progress while knitting. Knitting patterns require a person to track the number of stitches they have completed in a row and the number of rows they have completed in the project. The stitch counter will know that a finished row consists of a fixed number of stitches, represented by the instance variable **stitchesPerRow**.

By calling **addStitch()**, the number of stitches in a row will be incremented. If this last stitch added means that the row is complete, then the number of **stitches** is set to 0 and the number of **rows** is incremented by 1.

Provided below are a few unit tests that formalize the expected behavior of the object's **constructor** and **addStitch** method. Your job will be to implement both of these methods matching the behavior laid out in the writing above & the unit tests.

<pre>@Test public void testConstructSPR() {     Stitches sc = new Stitches(12);     int expected = 12;     int actual = sc.getSPR();     assertEquals(expected, actual); }</pre>	<pre>@Test public void testConstructStart() {     Stitches sc = new Stitches(4);     String expected = "Row 0, Stitch 0";     String actual = sc.toString();     assertEquals(expected, actual); }</pre>
<pre>@Test public void testFirstStitch() {     Stitches sc = new Stitches(4);     sc.addStitch();     String expected = "Row 0, Stitch 1";     String actual = sc.toString();     assertEquals(expected, actual); }</pre>	<pre>@Test public void testFirstRow() {     Stitches sc = new Stitches(3);     sc.addStitch();     sc.addStitch();     sc.addStitch();     String expected = "Row 1, Stitch 0";     String actual = sc.toString();     assertEquals(expected, actual); }</pre>
<pre>@Test public void testFirstRowFirstStitch() {     Stitches sc = new Stitches(2);     sc.addStitch();     sc.addStitch();     sc.addStitch();     String expected = "Row 1, Stitch 1";     String actual = sc.toString();     assertEquals(expected, actual); }</pre>	<pre>@Test public void testSecondRow() {     Stitches sc = new Stitches(2);     sc.addStitch();     sc.addStitch();     sc.addStitch();     sc.addStitch();     String expected = "Row 2, Stitch 0";     String actual = sc.toString();     assertEquals(expected, actual); }</pre>

Finish the class implementation below by completing the **constructor** and **addStitch()**. The instance variables have been declared for you and **toString()** and **getSPR()** are completed for you.

```
public class Stitches {
    private int rows, stitches;
    private String projectSize;
    private int stitchesPerRow;
    public Stitches(int spr) { }      \\TODO
    public void addStitch() { }      \\TODO
    public String toString() {
        return "Row " + rows + ", Stitch " + stitches;
    }
    public int getSPR() {
        return stitchesPerRow;
    }
}
```

### 3.1 Constructor

```
// spr is short for "Stitches Per Row" and should be used to initialize
// the stitchesPerRow variable of the object. If the project has more than 10
// stitches per row, the variable projectSize should be initialized to "large";
// otherwise, it should be initialized to "small".
public Stitches(int spr) {

    stitchesPerRow = spr;
    rows = 0;
    stitches = 0;
    if (spr > 10) {
        projectSize = "large";
    } else {
        projectSize = "small";
    }
}
```

### 3.2 addStitch()

```
// Increment stitches by one. If this would complete the row, then reset
// stitches and increment row by one instead.
public void addStitch() {

    stitches++;
    if (stitches == stitchesPerRow) {
        stitches = 0;
        rows++;
    }
}
```

```
}

```

## Q4. Recursion

In this section, you will write a function **isSorted()** that calls a recursive helper **isSortedHelper()** that returns **true** if an array of ints is in ascending sorted order, meaning that each element is less than or equal to all elements at a higher index. Otherwise, return **false**.

Q4.1 Before writing code, complete/evaluate these statements:

An array containing one element or no elements is always in ascending sorted order.	<input checked="" type="checkbox"/> true	<input type="checkbox"/> false
An array A is definitely <b>not</b> in ascending sorted order if there is any index i such that _____.	<input checked="" type="checkbox"/> $A[i] > A[i + 1]$	<input type="checkbox"/> $A[i] \leq A[i + 1]$

Q4.2 Now, complete the implementation of the function **isSortedHelper()**, which takes in a `double[] arr` and an **index** representing the current position of the array to be inspected.

```
private static boolean isSortedHelper(int[] arr, int index) {
    // Base case: If the array has one or zero elements left to inspect
    // based on the value of index and the length of arr
    if (arr.length - index <= 1) {
        return true;
    }

    // Base case: Case when the array is not sorted because the current
    // element and the following one are in the wrong order.
    if (arr[index] > arr[index+1]) {
        return false;
    }

    // Recursive case for the remaining elements
    return isSortedHelper(arr, index + 1);
}
```

Q4.3 Finally, complete **isSorted()** by providing the values for the initial call to the recursive helper. You should not need any additional space to write this; it can be done in one line.



```
private static boolean isSorted(int[] arr) {
    return isSortedHelper(arr, 0);
}
}
```

## Q5. Nodes

For Question 5, assume we have the following Node class:

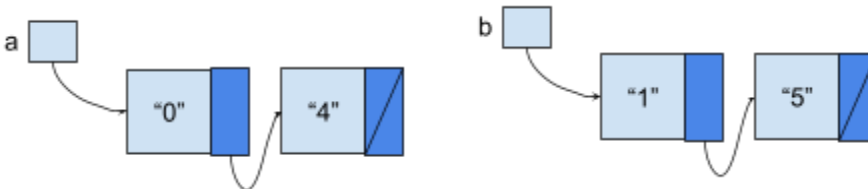
```
/**
 * This node class will store String values.
 */
public class Node {

    // public instance variables
    public String data;
    public Node next;

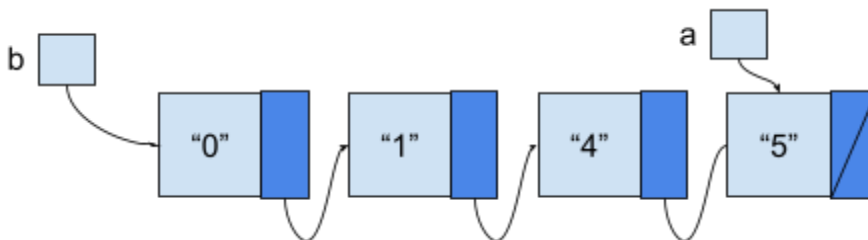
    public Node(String data, Node next) {
        this.data = data;
        this.next = next;
    }
}
}
```

### Question 5.1:

Given the following state of nodes:



rearrange the references so that the **variables & nodes** look like:



To be clear, you do not have to generalize this to work on any sequence of Nodes. You are given the chains drawn in the first image and you have to transform it into the chain drawn in the second image.

**NOTE:** you **are not allowed to** access or modify the value in any node's data instance variable, and you are not allowed to allocate a new node (e.g. the code you write should not contain *new Node(..., ...)* in it.)

You **are allowed to** create Node reference variables with something like: `Node temp = a;`

```
public static void main(String[] args) {
    Node a = new Node("0", null);
    a.next = new Node("4", null);

    Node b = new Node("1", null);
    b.next = new Node("5", null);
```

```
// your solution here
```

```
Node four = a.next; // 4
Node five = b.next; // 5
a.next = b; // 0 points to 1
a.next.next = four; // 1 points to 4
four.next = five; // 4 points to 5
b = a; // b points to 0
a = b.next.next.next; // a points to 5 this can also be a = five;
a.next = null; // a should be tail
```

```
}
```

## Question 5.2:

Complete the following function **countNulls()**:

```
/**
 * Given a reference to the start of a chain of linked nodes.
 * Iterate or recurse through each node reachable from head and check
 * if the data in each node is null. (Can use next page to define recursive
 * helper function if using.)
 * Return the number of nodes visited where the data instance
 * variable is null.
 */
public static int countNulls(Node head) {
    // TODO

    ITERATIVE SOLUTION

    int numNulls = 0;
    for (Node curr = head; curr != null; curr = curr.next) {
        if (curr.data == null) {
            numNulls++;
        }
    }
    return numNulls;

    RECURSIVE SOLUTION

    NO HELPER
    if (head == null) {
        return 0;
    }
    if (head.data != null) {
        return countNulls(head.next);
    }
    if (head.data == null) {
        return 1 + countNulls(head.next);
    }

    YES HELPER
    return countNullsHelper(head, 0);
}

HELPER

private static int countNullsHelper(Node curr, int numNulls) {
    if (curr == null) {
        return numNulls;
    }
    if (curr.data != null) {
        return countNullsHelper(curr.next, numNulls);
    }
}
```

```
    }  
    if (curr.data == null) {  
        return countNullsHelper(curr.next, numNulls + 1);  
    }  
}
```

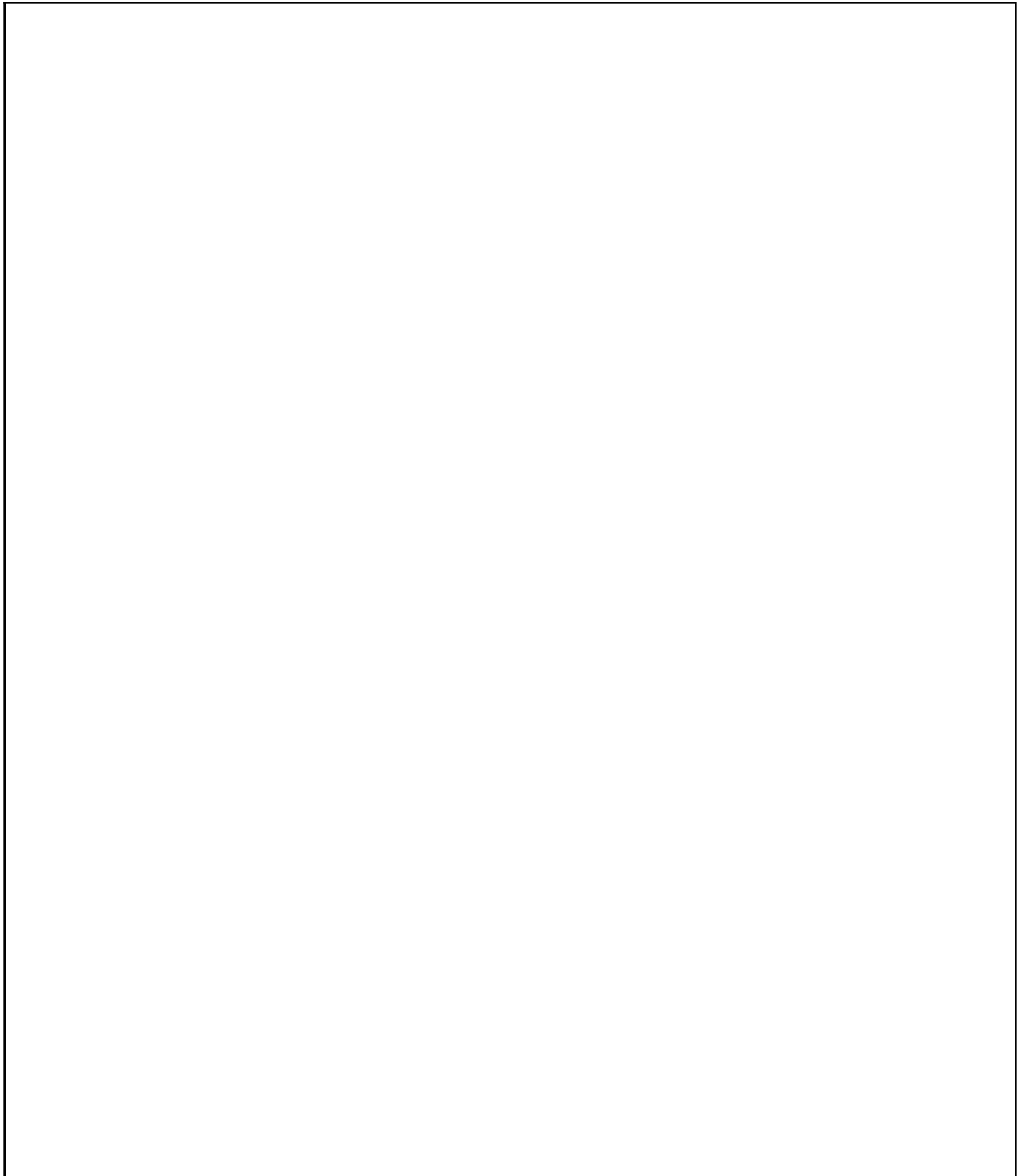
### Bonus:

What is one piece of advice you'd give to a new student just starting out CIS 1100?

All exams will get full credit for this question, answer as seriously as you would like!

**Room for Extra Answers**

You may use this page for additional space for answers; keep it attached to this exam. Clearly note on the original question page that your answer is on this extra page, and clearly note on this page what question you are answering.

A large, empty rectangular box with a thin black border, occupying most of the page below the instructions. It is intended for students to write their answers to exam questions.

## Appendix

### Lists

<b>method signature</b>	<b>purpose</b>
get(int i)	return the value at position i in the List.
set(int i, E e)	set the value at position i in the List to be e.
size()	return the number of values stored in the List.
remove(int i)	remove and return the value stored at position i in the List.
add(E e)	insert the value e at the end of the List.
add(int i, E e)	insert the value e at position i in the List. i must be between 0 and size().