This exam is open-book -- you may refer to any book or any notes you have brought with you during the exam. However, you may not use a computer of any kind (including cell-phones) during the exam.

Score:

Problem 1: ______/4

Problem 2: ______/4

Problem 3: ______/12

Extra credit: ______

Total: ______/20
Problem 1: Short-answers -- 4 points

1. (1 point): Describe one advantage of linked lists over array-based lists.

2. (1 point): Describe one advantage of array-based lists over linked lists.

3. (1 point): Describe one advantage of separating a software interface from its implementation when developing software:

4. (1 points): Give the worst-case asymptotic time complexity of the addToFront(o) method of an array-based list. (Recall that, after addToFront(o) has finished, o will reside at index 0 of the array.) You must justify your answer to get credit.
Problem 2: Object-orientation in Java -- 4 points

The purpose of this problem is to make sure you understand the relationship between classes, abstract classes, interfaces, sub-interfaces, and implementations.

Consider the Java interfaces specified below. Write a (non-abstract) class C that extends D and implements interface B. **Your class C doesn’t have to do anything useful.** However, there are two requirements: (a) your code must compile without errors; and (b) none of your methods may return null -- for instance, a method with return-type A must return a valid object of type A. If you wish, you may define additional classes -- either inner-classes or “regular” classes -- to complete this task.

```java
interface A {
    int gimmeSomeInt ();
}

interface B extends A {
    A nextA (String yoSup);
}

abstract class D {
    public abstract void notMuch ();
    public abstract D yetAgain ();
}

class C extends D implements B {
    // Write your solution below. You may also
    // create additional classes if they help.
    // ** Make sure that all methods are public! **
```
Problem 3: CountingList -- 12 points

Create a Java class called CountingListImpl (along with a static inner-class Node) that implements the CountingList interface (shown below). A CountingList is a doubly-linked list that additionally keeps track of how many times an element has been added to the list (minus the number of times it was removed). Your static inner-class Node should include not only _next, _prev, and _data instance variables, but also an int _counter instance variable.

The user can add an object o to the list by calling add(o): If o is already in the list, then the counter associated with o is incremented. Otherwise, a new Node should be created, its counter set to 1, and the Node should be added to the tail of the list. You should test whether o is already contained in the list using the Java equals(o) method. You may assume that the user will never call add(o) with null as the argument.

The user can “remove” an object o by calling the remove(o) method. If o is not in the list, then this method should have no effect. If o is in the list, then remove(o) should decrement the counter associated with o by 1. If the counter reaches 0, then o (and its associated Node) should be removed from the list entirely. If the counter is positive, then o should remain in the list.

Some of the code is already written for you. Your solution does not need to be “generic”.

```
// CountingList: Doubly-linked list that additionally stores the
// number of times an element was added.
interface CountingList {
    // add: Either adds o to the list (if o was is already in the
    // list), or increments the counter associated with o by 1 (if o is
    // already in the list). o cannot be null.
    void add (Object o);

    // getCount: Returns the value of the counter associated with o. If
    // o is not in the list, then this method returns 0 (zero).
    int getCount (Object o);

    // remove: Decrements the counter associated with o by 1. If the
    // counter reaches 0 (zero), then o is removed from the list;
    // otherwise, o remains in the list. If o was not contained in the
    // list, then this method has no effect.
    void remove (Object o);
}
```
Problem 3: CountingList (continued)

class CountingListImpl implements CountingList {
    private static class Node {
        Node _next, _prev;
        Object _data;
        int _counter;
    }

    private Node _head, _tail;  // dummy head and tail

    CountingListImpl () {
        _head = new Node();
        _tail = new Node();
        _head._next = _tail;
        _tail._prev = _head;
    }
    // Insert your code below...
Problem 3: CountingList (continued)
Extra credit: Reverse a doubly-linked list in $O(1)$ space -- 3 points

Assume you have already implemented a doubly-linked list implementation in a Java class called DoublyLinkedList as shown below. Implement a method called `void reverse()` that reverses the order of data stored in the list. In other words, if the data stored in the list (ordered from head to tail) were “a”, “b”, and “c”, then the reversed list would be “c”, “b”, and “a”. Your algorithm may have $O(n)$ time complexity (for $n$ data stored in the list) but must have $O(1)$ space complexity, i.e., the amount of memory the algorithm requires should not depend on the length of the list. In fact, it is possible to write this method without creating a single additional object.

class DoublyLinkedList {
    private static class Node {
        Node _next, _prev;
        Object _data;
    }
    private Node _head, _tail; // Assume they point to dummy nodes

    ... 

    void reverse () {
        
    }
}