CS 101 Fall 2006 Final Exam Name: _____ Email ID: _____

You only need to write your name and e-mail ID on the first page.

This exam is CLOSED text book, closed-notes, closed-calculator, closed-neighbor, etc. Questions are worth different amounts, so be sure to look over all the questions and plan your time accordingly. Please sign the honor pledge here:

_____

_____

_____

_____

Section #	____ / 5
Question 1	____ / 15
Question 2	____ / 25
Question 3	____ / 20
Question 4	____ / 10
Question 5	____ / 25
Total	**____ / 100**

Note: When an integer type is required use `int`*, when a floating-point type is required use* `double`*. If we don't specify an aspect of a problem, you can make any choice consistent with the given constraints.*

Note: If you are still writing on the exam after "pens down" is called – even if it is just to write your name – then you will receive a zero on this exam. No exceptions!

1. [5 points] What lab section are you in?

____ CS 101-E

____ CS 101-2 (lab 7:00–8:30 p.m. Thu)

____ CS 101-3 (lab noon–1:30 p.m. Fri)

____ CS 101-4 (lab 2:00–3:30 p.m. Fri)

____ CS 101-5 (lab 10:00–11:30 a.m. Fri)

____ CS 101-6 (lab 2:00–3:30 p.m. Thu)

This exam involves the programming of a single class: a simplified version of the Java *Vector* class. A major benefit of a vector over an array is that the size of a vector is not fixed. Rather, vectors grow and shrink as elements are added and removed. The class you will implement, called *MyVector*, will support variable-sized *lists of Objects* using a technique known as a *linked list* (which we'll discuss in a moment). The *MyVector* class provides five public methods, with the following specifications:

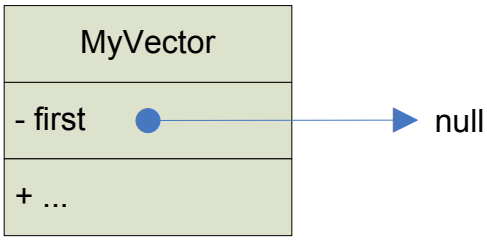
- MyVector() – construct an empty list, i.e., one with no elements
- int count() – return the number of elements in the list
- void add(Object o) – add the Object *o* as an element at the beginning of the list
- void clear() – clear the list, i.e., set it back to a list with no elements
- Object [] toArray() – return an *array* whose size is the length of the list and that contains the same elements as the list in the same order that they appear in the list

Here then is an outline of the MyVector class:

```
public class MyVector {
   private ListNode first;          // reference to a node storing the first element in the list, if any

   public MyVector()                { // your code here }
   public int count()               { // your code here }
   public void add(Object o)        { // your code here }
   public void clear()              { // your code here }
   public Object [] toArray()       { // your code here }
}
```

Class *MyVector* has a single private data member, *first*, of type *ListNode*. The value of this data member represents the list of objects stored by a given *MyVector* object. Here is some very important information. Read this carefully. (1) An empty list is represented by setting the value of *first* to *null*. Recall: in Java, *null* is the constant representing a null object reference. Here, then, is a picture of the representation of an empty list:



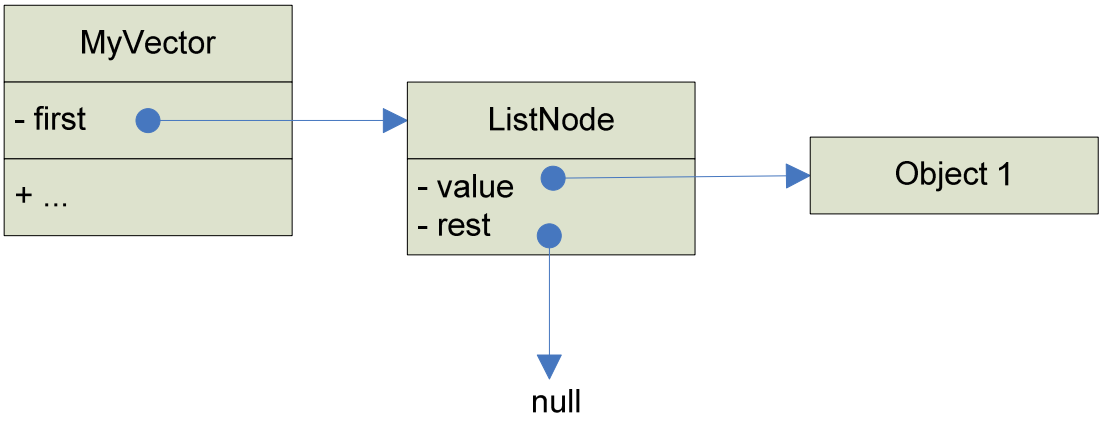
(2) A non-empty list is represented by setting the value of *first* to be a reference to a *ListNode* object that in turn represents two things: the *value* of the *first* element in the list, and a reference to another *ListNode* object that represents the beginning of the *rest* of the list (which, at the end of the list, is the empty list).

This approach to *chaining* together nodes through references is what we mean by the term *linked list*. To see how this will works, you have to first consider and understand the definition of the *ListNode* class, itself.

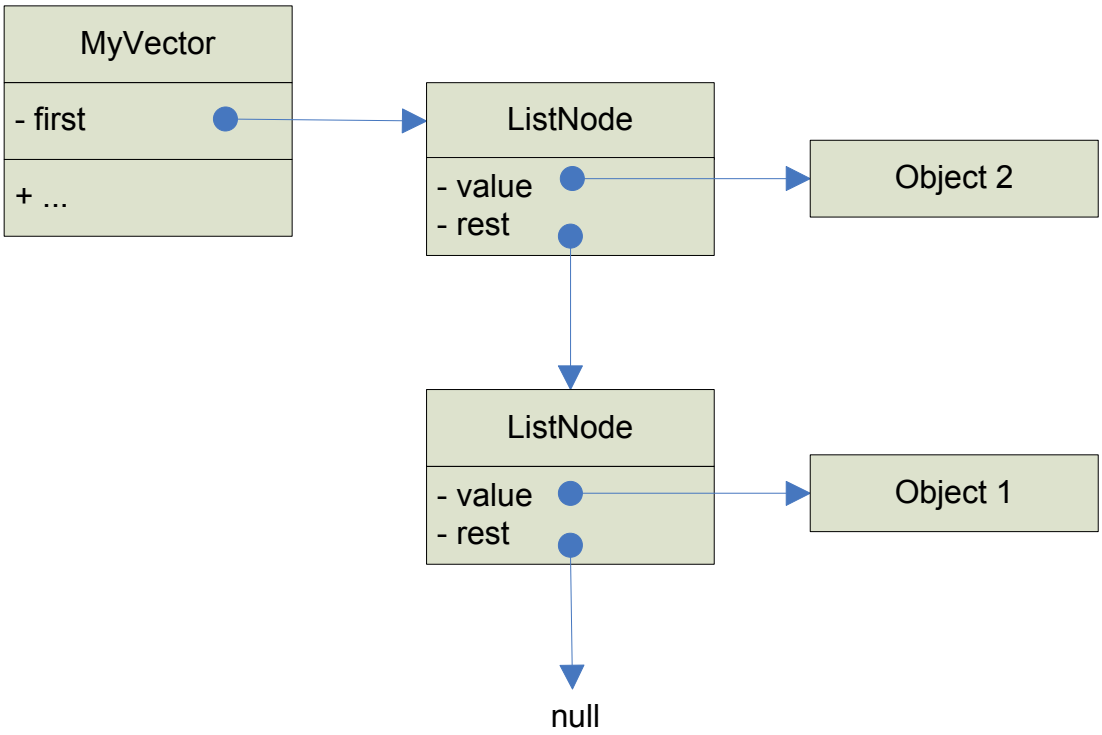
```
class ListNode {
   public Object value;        // reference to the Object stored by this node
   public ListNode rest;       // reference to the first ListNode in the rest of the list, if any (can be null)
}
```

Remember: An object of this class will store just *one* of the values in a *list* of values. A *MyVector* object representing a list of *N* objects will thus have *N ListNode* objects, one for each element in the list. As you can see, the *ListNode* class is simple. It has two public data members. The first, *value*, will holds a reference to one object in a given list. The second data member, *rest*, is the slightly tricky part. It stores a reference to the first ListNode in the rest of the list, which we represent as a reference, *rest*, to another *ListNode* object. All of this is much clearer if you consider a few simple pictures. In the following pictures, each rectangle represents an object. Its type is given at the top of the rectangle and its data members follow. Reference values are represented by arrows between objects. A *null* reference is represented by an arrow pointing to *null*.

The picture that follows is of a list with one element, called *Object 1*. The list as a whole is represented by an object of type *MyVector*. This object has a data member *first*, as described above. The value of this data member is a reference to a *ListNode* that in turn stores a reference to *Object 1*. The *ListNode* object also has a reference, *rest*, to the rest of the list. Because this list has only one element, the rest of the list is empty, and so the value of the *rest* data member is set to *null*. If the rest of the list were not empty (as would occur if the list had two or more elements), then the value of the *rest* data member would be a reference to another *ListNode* object representing the first element in the rest of the list!



Here, then, is a picture for a list with two elements, *Object 2* and *Object 1*. In particular, this is the list that we will get if we add *Object 2* to a list initially containing only *Object 1*. On this exam, we always add new elements at the beginning of a list. A list with two elements, as the picture below illustrates, is represented by having the *first* data member of class *MyVector* refer to a *ListNode* representing the first element in the list, and by having that *ListNode* refer to a second *ListNode* object, which in turn represents the second object in the list. Here then is a picture of a list with two elements, *Object 2* and *Object 1*, which, again, is the list we get if we *add Object 2* to the one-element list above.



A list with three elements would be represented by having the second *ListNode* object refer to a third. A list with *N* elements is thus represented by a *chain* of *ListNode* objects, each referring to the next in line, with the last one in line referring to *null*.

Your job on this exam is to program a set of *MyVector* methods that in effect maintain a variable-length list of objects as a variable-length chain, *linked list*, of ListNodes.

Question one [15 points]. Provide an implementation of the constructor, *MyVector()*. Remember, this method is required to initialize a new *MyVector* object to represent an *empty* list of objects. If you don't know what to do at this point, just go back and re-read the preceding directions. They state very clearly how we will represent an empty list.

```
public MyVector() { // your code goes here




}
```

Question two [25 points]. Implement the method *public void add(Object o)*. Think very carefully about what needs to happen. First, you need to create a new *ListNode* object to represent (and refer to) the object, *o*, being added to the list. Then you need to make this new *ListNode* object the first element in the chain of *ListNode* objects representing the updated list. The key insight is that the object being added will become the *first* element in the updated list, while the *first* element in the list before the new node is added will become the *rest* of the new list. Read that again. Now go back and look at the pictures of the empty and the one-element lists above, and between the one-element and two-element lists. The differences between these pairs of lists is precisely the difference that your *add* code must implement. Note in particular: the *first* node in the list *before* the *add* method executes becomes the first node in the *rest* of the list after the new element is added. Here, then, is an outline of what your code must do: (1) create a new ListNode; (2) make its *value* refer to the object being added; (3) make its *rest* field refer to the element that was, until now, first in the list (which will be null if the list was empty); (4) make the *first* element in the new list refer to the new *ListNode.*

```
public add(Object o) { // your code goes here

    // create a new ListNode


    // make its value refer to the object being added


    // make its rest field refer to the element that until now was first in the list


    // make the first element in the list refer to the new ListNode.

}
```

Question three [20 points]. This question tests your ability to use iteration in Java. The *count* method returns the number of elements in the list. The question is how to implement this method. Here what you need to do. Your code will (1) initialize a counter to zero and then execute the following loop: (a) load the reference to the *first ListNode* into a variable *cursor*; (b) if the end of the chain has been found (*cursor* is null) terminate the loop and return the current count (which will be zero if the list was empty to start with), (b) otherwise add one to the count, update *cursor* to refer to the next *ListNode* in the chain (*rest*), and repeat the loop, starting with the test. This *iterative* process will repeat until the end of the list is found, at which point *count* will have accumulated a value indicating the number of elements in the list. If you haven't understood these directions so far, just try "running" them by hand. Your finger is the *cursor*. See the picture of the empty list. *count* starts at zero. Point at *first.* What is its value? Null? Yes, the loop terminates and the "program" returns *count*, which is zero. Now do the same thing for the list with one element. *count* starts at zero. Point at first. Is it zero? No, so add one to *count,* and now point at *rest.* Is it zero? Keep going. Get it? It's a little tricky but really not so hard! Now you just need to express your understanding in Java code.

```
public int count() {

   // initialize an integer variable count to zero


   // initialize a ListNode variable, cursor, to the first element in the list (it could be null)


   // write the loop – if cursor is null, return count; otherwise (a) increment count, (b) update cursor to refer
   // to the rest ListNode (it could be null), and (c) repeat.




   // return the value of count

}
```

Question four [10 points]. This one is easy. Implement *public void clear().* When this method is applied to a *MyVector v*, the result is that *v* subsequently represents the empty list.

```
public void clear() {
   // your code goes here




}
```

Question five [25 points]. Okay, you now have to implement *public Object [] toArray().* This method is to return an array of the same length as the list and containing the same elements in exactly the same order. You may assume that this method will only be called for *MyVector* lists with at least one element. Conceptually the logic is pretty simple. First, create an array of *Objects* having the same number of elements as the given list. Hint: You already have a method that returns that information. *Use it.* Second, you need to set the value of each element in the array to the value of the corresponding element in the list. To do this, loop through the elements of the list, setting the corresponding elements of the array. You have already written code to loop through the elements of the list. Re-use the key parts of that code here.

```
public Object [] toArray() {

   // determine the number of elements in the list, store it as length



  // allocate an array arr of objects of that length



  // iterate over the list elements, storing each value in the corresponding array element.
  // You will need to keep track of the correct array index as you proceed, incrementing by
  // one on each pass through the loop.




}
```

This page unintentionally left blank