Exam 2 for Com S 228 Spring 2017 on March 23, 2017

Name:

University ID:

Recitation section (please circle one):

```
Section 1  R 10:00am-10:50am    SWEENEY 1126    Andrew & Ryan K.
Section 2  R 2:10pm-3:00pm      SWEENEY 1126    Anthony
Section 3  R 1:10pm-2:00pm      SWEENEY 1134    Lige & Alex
Section 4  R 4:10am-5:00am      SWEENEY 1126    Michael & Xinxin
Section 5  R 3:10pm-4:00pm      SWEENEY 1126    Ryan Y.
Section 6  T 9:00am-9:50am      SWEENEY 1126    Blake
Section 7  T 2:10pm-3:00pm      SWEENEY 1126    Trang & Alex
Section 8  T 10:00am-10:50am    SWEENEY 1126    Kelsey & Blake
```

There are 8 pages and 4 questions. Please start with easy questions. We will give partial credit for a partially correct solution. You are not allowed to use any books, notes, calculators, or electronic devices. The exam is 60 minutes long.

Question	Points	Your Score
1	20	
2	20	
3	20	
4	40	
Total	100	

1. (20 pts) On the next page, you will see several snippets of code. Assume the code executes within a class with two instance variables and a method `init()` as shown:

```
public class Exam2
{
   List<String> aList;
   ListIterator<String> iter;
   void init()
   {
      aList = new ArrayList<String>();
      aList.add("A");
      aList.add("B");
      aList.add("C");
      aList.add("D");
   }
  // ...
}
```

For each snippet,

(a) show what the output from the `println` statement is, if any, and

(b) draw the state of the list and iterator after the code executes.

However, if the code throws an exception, print any output that occurs before the exception, but ***do not*** draw the list; instead write down the exception that is thrown.

Use a bar (`|`) symbol to indicate the iterator's logical cursor position. For example, right after the statements

```
init();
iter = aList.listIterator();
```

the list would be drawn as follows.

```
|A B C D
```

(the first one has been done for you as an example).

Code snippet	**Output**	**`List` iterator and state or exception thrown**
`init();` `iter = aList.listIterator();`	(none)	`|A B C D`

`// 3pts`
`init();`
`iter = aList.listIterator();`
`iter.previous();`
`System.out.println(iter.nextIndex());`		

`// 5pts`
`init();`
`iter = aList.listIterator();`
`iter.next();`
`iter.add("X");`
`iter.previous();`
`iter.set("S");`
`System.out.println(iter.next());`		

`//4pts`
`init();`
`iter = aList.listIterator();`
`iter.add("X");`
`System.out.println(iter.next());`
`System.out.println(iter.previous());`
`iter.remove();`		

`//4pts`
`init();`
`iter = aList.listIterator(4);`
`iter.add("X");`
`System.out.println(iter.nextIndex());`
`iter.remove();`		

`//4pts`
`init();`
`iter = aList.listIterator(aList.size()/2);`
`while` `(iter.hasNext()){`
`  iter.set("X");`
`  iter.next();`
`}`		

2. (20 pts) Complete the implementation of the `static` method `isSorted()` below. If List `list` is `null`, throw a `NullPointerException`. The method should return `true` if the number of elements in List `list` is 0 or 1, or if its elements are in strictly increasing order according to the comparator `comp` (The element at index 0 is less than the element at index 1, the element at index 1 less than the element at index 2, and so on). Otherwise, it returns `false`.

```
import java.util.Comparator;
import java.util.List;
import java.util.ListIterator;

public static <T> boolean isSorted(List<T> list,
                           Comparator<           > comp)     // <- ANSWER
{

  if ( list == null )
                                                             // <- ANSWER

  if (                           )                           // <- ANSWER
       return true;

  ListIterator<T> iter = list.listIterator();
  T prev = iter.next();
  while ( iter.hasNext() )
  {
    T curr = iter.next();

    if (                                      )              // <- ANSWER
          turn false;

                                                             // <- ANSWER
  }

                                                             // <- ANSWER
}
```

3. (20 pts) Give the big-O time complexity of the following API operations from the Java Collections framework. Be sure to list the worst case performance of each operation and use formal notation! Assume that the number of elements in the list is n. If the operation also involves a separate collection, assume the number of elements in the collection is m. You may assume that Java's `LinkedList` implementation maintains a tail reference. You may assume that the element type has a constant-time `equals()` method in all cases.

   (a) (5 pts)

   ```
   LinkedList.containsAll(Collection<?> c);
   ```

   (b) (5 pts)

   ```
   LinkedList.addFirst(E e)
   ```

   (c) (5 pts)

   ```
   List.remove(Object obj)
   ```

   (d) (5 pts)

   ```
   ArrayList.listIterator(int index)
   ```

4. (40 pts). `HeadTail` is a genetic class implementing a limited version of a doubly linked list (a limited list), where data items can be added, removed or accessed only at the ends of the list. A limited list is more powerful than a stack or queue. Each method in the class `HeadTail` takes O(1) time. Your task is to fill in the blanks to complete the functionality. Pay attention to the comments for additional instructions and clues.

```
import java.util.NoSuchElementException;

public class HeadTail<E>
{
  private class Node
  {
    public E data;
    public Node next;
    public Node prev;

    public Node(E item)
    {
      data = item;
      next = prev = null;
    }
  }

  private Node head; // a dummy node for one end of the list
  private Node tail; // a dummy node for the other end
  private int size;  // number of elements in the list

  public HeadTail()
  {
    head = new Node(null);
    tail = new Node(null);
    head.next = tail;
    tail.prev = head;
    size = 0;
  }

  // Returns the number of elements in the list.
  public int size()
  {
                                                          // <- ANSWER
  }

  // Returns true if the list is empty.
  public boolean isEmpty()
  {
                                                          // <- ANSWER
  }
```

```
// Removes the node toRemove from the list.
private void unlink(Node toRemove)
{
  if ( toRemove == head || toRemove == tail )
    throw new RuntimeException("An␣attempt␣to␣remove␣head␣or␣tail");
  toRemove.prev.next = toRemove.next;
                                                              // <- ANSWER
}

// Connects the new node toAdd immediately after the old node current.
private void link(Node current, Node toAdd)
{
  if ( current == tail )
    throw new RuntimeException("An␣attempt␣to␣link␣after␣tail");
  if ( toAdd == head || toAdd == tail )
    throw new RuntimeException("An␣attempt␣to␣add␣head/tail");
  toAdd.next = current.next;
                                                              // <- ANSWER
  toAdd.prev = current;
                                                              // <- ANSWER
}

// Creates a new node with the given element and places
// it immediately after the head.
public void addAtHead(E element)
{
  Node toAdd = new Node(element);
                                                              // <- ANSWER
  size++;
}

// Removes the non-dummy node immediately after the head if it exits.
public E removeAtHead()
{
  if (                        )                               // <- ANSWER
    throw new NoSuchElementException();
  E returnVal = head.next.data;
                                                              // <- ANSWER
  if ( size <= 0 )
    throw new RuntimeException("an␣incorrect␣number␣of␣elements");
                                                              // <- ANSWER
  return returnVal;
}
```

```
  // Returns the data field in the non-dummy node
  // immediately after the head if it exits.
  public E head()
  {
    if (                          )                              // <- ANSWER
      throw new NoSuchElementException();
                                                                 // <- ANSWER
  }

  // Creates a new node with the given element and places
  // it immediately before the tail.
  public void addAtTail(E element)
  {
    Node toAdd = new Node(element);
                                                                 // <- ANSWER
    size++;
  }

  // Removes the non-dummy node immediately before the tail if it exits.
  public E removeAtTail()
  {
    if (                          )                              // <- ANSWER
      throw new NoSuchElementException();
    E returnVal = tail.prev.data;
                                                                 // <- ANSWER
    if ( size <= 0 )
      throw new RuntimeException("an␣incorrect␣number␣of␣elements");
                                                                 // <- ANSWER
    return returnVal;
  }

  // Returns the data field in the non-dummy node
  // immediately before the tail if it exits.
  public E tail()
  {
    if (                          )                              // <- ANSWER
      throw new NoSuchElementException();
                                                                 // <- ANSWER
  }

} // HeadTail
```