DECLARATIONS

```
public class MyLinkedList<E>  {

    private Node<E> head, tail;

    public MyLinkedList() {
        head = null;
        tail = null;
    }

    private static class Node<E> {
        E element;
        Node<E> next;

        public Node(E element) {
            this.element = element;
            next = null;
        }
    }
```

contains()

```
    public boolean contains(Object o) {
        for (Node<E> ptr = head; ptr != null; ptr = ptr.next) {
            if (ptr.element.equals(o))
                return true;
        }
        return false;

        //   ARRAY EQUIVALENT FOR CONTAINS
        //   for (int i = 0; i < size; i++) {
        //       if (data[i].equals(o))
        //          return true;
        //   }
        //   return false;

    }
```

getFirst()

```
    /** Return the head element in the list */
    public E getFirst() {
        if (head == null) {
            return null;
        }
        else {
            return head.element;
        }
    }

    /** Return the last element in the list */
```

getLast()

```
public E getLast() {
     if (head==null) {
         return null;
     }
     else {
         return tail.element;
     }
 }
```

prepend()

```
/** Add an element to the beginning of the list */
public void prepend(E e) {
    Node<E> newNode = new Node<>(e);        // Create a new node
    newNode.next = head;                    // link the new node with the head
    head = newNode;                         // head points to the new node

    if (tail == null)                       // the new node is the only node in list
        tail = head;
}
```

append()

```
/** Add an element to the end of the list */
public void append(E e) {

    Node<E> newNode = new Node<>(e);

    if (head == null) {
        head = tail = newNode;
    }
    else {
        tail.next = newNode;
        tail = newNode;
    }
}
```

removeFirst()

```
/** Remove the head node and
 *  return the object that is contained in the removed node. */
public E removeFirst() {
    if (head == null) {
        return null;
    }
    else {
        E temp = head.element;
        head = head.next;
        if (head == null) {
            tail = null;
        }
        return temp;
        }
    }
```

delete()

```
    public boolean delete(E item) {

        if (head == null)
            return false;

        Node<E> ptr = head;
        Node<E> prvPtr = null;

        while (ptr != null && (!ptr.element.equals(item))) {
            prvPtr = ptr;
            ptr = ptr.next;
        }

        if (ptr == null)
            return false;


        if (ptr == head)
            head = head.next;
        else
            prvPtr.next = ptr.next;

        if (ptr == tail)
            tail = prvPtr;

        return true;
    }
```

merge()

```
    public MyLinkedList merge(MyLinkedList paramlist)
    {
        Node<E> ptrCall, ptrParam;
        ptrCall = this.head;
        ptrParam = paramlist.head;

        MyLinkedList returnlist = new MyLinkedList();

        // calling list is empty - set this list to param list
        if(head==null) {
            return paramlist;
        }
        // param list is empty - make no changes
        if(paramlist.head == null){
            return this;
        }
        // traverse both list until one list is completely done
        while((ptrCall != null) && (ptrParam != null))
        {
            if (((Comparable)ptrCall.element).compareTo(ptrParam.element) <= 0)
            {
                returnlist.append(ptrCall.element);
                ptrCall = ptrCall.next;
            }
            else
            {
                returnlist.append(ptrParam.element);
                ptrParam = ptrParam.next;
            }
        }

        if(ptrCall == null)     // copy rest of param list
        {
           for (ptrParam = ptrParam; ptrParam != null; ptrParam = ptrParam.next)
               returnlist.append(ptrParam.element);
        }

        if(ptrParam == null)    // copy rest of calling list
        {
            for (ptrCall = ptrCall; ptrCall != null; ptrCall = ptrCall.next)
                returnlist.append(ptrCall.element);
        }

        return returnlist;
    }
```

isSublist()

```
    public boolean isSublist(MyLinkedList<E> paramList) {

        if (paramList.head == null || this.head == null)
            return false;

        Node<E> ptr = this.head;
        Node<E> ptrParam = paramList.head;

        while (ptr != null) {
            if ( ((Comparable)ptr.element).compareTo(ptrParam.element) == 0) {
                ptrParam = ptrParam.next;

                if (ptrParam == null)
                    return true;
            }
            ptr = ptr.next;
        }
        return false;
    }
```

toString()

```
      public String toString() {
        String result = "[";

        Node<E> ptr = head;
        for (ptr= head;ptr!=null; ptr=ptr.next)
        {
            result = result +  ptr.element.toString();
            if (ptr.next != null)
                result = result + ","; // add commas but not to the final 1
        }
        result += "]"; // Insert the closing ] in the string
        return result;
    }
```

clear()

```
    public void clear() {
        head = tail = null;
    }
```

Tutoring session

```
    // The method we wrote that returns a list of elements that only occur ONCE in the
list
    public MyLinkedList<E> getSingletons() {

        MyLinkedList<E> returnList = new MyLinkedList<>();

        for (Node<E> ptr = head; ptr != null; ptr = ptr.next) {

            boolean foundBefore = false;
            for (Node<E> ptrBefore = head; ptrBefore != ptr; ptrBefore = ptrBefore.next) {
                if (ptr.element.equals(ptrBefore.element)){
                    foundBefore = true;
                    break;
                }
            }

            boolean foundAfter = false;
            for (Node<E> ptrAfter = ptr.next; ptrAfter != null; ptrAfter = ptrAfter.next)
{
                if (ptr.element.equals(ptrAfter.element)) {
                    foundAfter = true;
                    break;
                }
            }

            if ((!foundBefore && !foundAfter)){
                returnList.append(ptr.element);
            }
        }
        return returnList;
    }

} // end myLinkedList class
```