Programming Assignment
Linked List with Iterator

1. A list (also known as a sequence or vector) is an ordered collection. Elements can be accessed by their integer index. There are several alternatives for implementing a list, but most commonly a list is implemented using either arrays or a linked data structure. This assignment uses a singly linked data structure for implementing a list.

2. Implement the class `LinkedList` whose skeleton is given on the following pages, and write one or more Java programs that tests your `LinkedList` class. Your work for this assignment should be contained in at least two separate files, a Java source file (.java) for the `LinkedList` class and one or more Java source files (.java) that test the linked list. You may modify the private parts of the `LinkedList` class as desired, but you should not modify the package name or public parts as specified.

3. Note that the `LinkedList` class implements interface `Iterable` (i.e., it has a method named `iterator()` that returns an `Iterator` object. This allows the `LinkedList` class to be used in a Java enhanced for-loop.

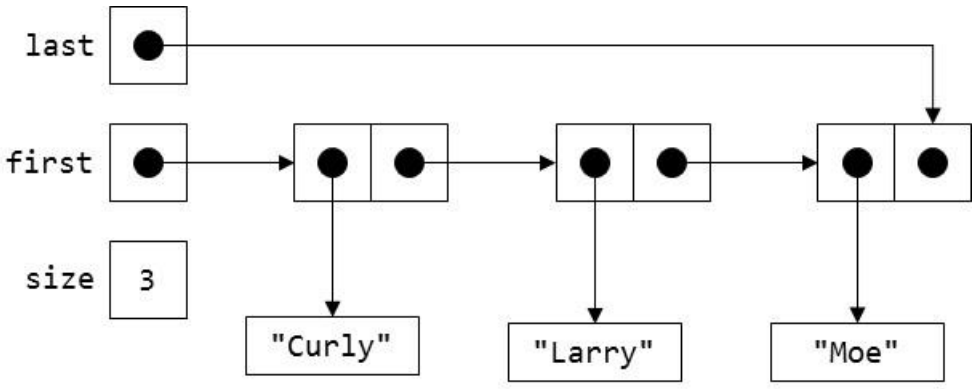
```
LinkedList<String> strings = new LinkedList<String>();

...  // add some strings to the list

for (String s : strings)
    System.out.println(s);
```

4. Also note that implementing interface `Iterable` gives you a `forEach()` method for free (default method in interface `Iterable`) that allows a lambda expressions as the parameter.

```
names.forEach(name -> System.out.println(name));
```

Turn in printed copies of your Java source file for the linked list plus one or more screen shots of the program running on your computer.

Implementation Notes: A linked list is implemented using two references, one for the first node in the list and one for the last node. Each node in the list contains a reference to the node's value and a reference to the next node in the list. The last node contains a null reference indicating that there is no "next" node. An integer value keeps track of the size of the list. For example, for a list of strings, after adding values "Curly," "Larry," and "Moe" in that order to an initially empty list, the list could be visualized as follows:



```
package edu.citadel.util;


import java.util.Iterator;
import java.util.NoSuchElementException;


/**
 * This class implements a List by means of a linked data structure.
 * A List (also known as a <i>sequence</i>) is an ordered collection.
 * Elements in the list can be accessed by their integer index.  The
 * index of the first element in the list is zero.
 */
public class LinkedList<E> implements Iterable<E>
  {
    private Node<E> first;   // reference to the first node
    private Node<E> last;    // reference to the last node
    private int size;        // number of elements in the list


    /**
     * A list node contains the data value and a link to the next
     * node in the linked list.
     */
    private static class Node<E>
      {
        private E data;
        private Node<E> next;


        /**
         * Construct a node with the specified data value and link.
         */
        public Node(E data, Node<E> next)
          {
            ...
          }


        /**
         * Construct a node with the given data value
         */
        public Node(E data)
          {
            this(data, null);
          }
      }


    /**
     *  An iterator for this singly-linked list.
     */
    private static class LinkedListIterator<E> implements Iterator<E>
      {
        private Node<E> nextElement;


        /**
```

```
     * Construct an iterator initialized to the first element in the list.
     */
    public LinkedListIterator(Node<E> head)
      {
        nextElement = head;
      }


    /**
     * Returns true if the iteration has more elements.
     */
    @Override
    public boolean hasNext()
      {
        ...
      }


    /**
     * Returns the next element in the list.
     *
     * throws NoSuchElementException if the iteration has no next element.
     */
    @Override
    public E next()
      {
        ...
      }

    // Note: Do not have to implement other methods in interface
    // Iterator since they have default implementations. The following
    // is provided for versions of Java prior to version 8.

    /**
     * Remove operation is not supported by this iterator.
     *
     * @throws UnsupportedOperationException always.
     */
    @Override
    public void remove()
      {
        throw new UnsupportedOperationException("remove");
      }
  }


/**
 * Helper method: Checks that the specified index is between 0 and size - 1.
 *
 * @throws IndexOutOfBoundsException if the index is out of range
 *         (<tt>index &lt; 0 || index &gt;= size()</tt>)
 */
private void checkIndex(int index)
  {
    if (index < 0 || index >= size)
        throw new IndexOutOfBoundsException(Integer.toString(index));
  }
```

```
/**
 * Helper method: Find the node at a specified index.
 *
 * @return a reference to the node at the specified index
 *
 * @throws IndexOutOfBoundsException if the index is out of range
 *         (<tt>index &lt; 0 || index &gt;= size()</tt>)
 */
private Node<E> getNode(int index)
  {
    checkIndex(index);
    Node<E> node = first;

    for (int i = 0;  i < index;  ++i)
        node = node.next;

    return node;
  }


/**
 * Constructs an empty list.
 */
public LinkedList()
  {
    ...
  }


/**
 * Appends the specified element to the end of the list.
 */
public void add(E element)
  {
    if (isEmpty())
      {
        first = new Node<E>(element);
        last = first;
      }
    else
      {
        last.next = new Node<E>(element);
        last = last.next;
      }

    ++size;
  }


/**
 * Inserts the specified element at the specified position in the list.
 *
 * @throws IndexOutOfBoundsException if the index is out of range
 *         (<tt>index &lt; 0 || index &gt; size()</tt>)
 */
public void add(int index, E element)
  {
```

```
    ...
  }


/**
 * Removes all of the elements from this list.
 */
public void clear()
  {
    while (first != null)
      {
         Node<E> temp = first;
         first = first.next;

         temp.data = null;
         temp.next = null;
      }

    last = null;
    size = 0;
  }


/**
 * Returns the element at the specified position in this list.
 *
 * @throws IndexOutOfBoundsException if the index is out of range
 *         (<tt>index &lt; 0 || index &gt;= size()</tt>)
 */
public E get(int index)
  {
    // do not need explicit index check since getNode() does it for us
    Node<E> node = getNode(index);
    return node.data;
  }


/**
 * Replaces the element at the specified position in this list
 * with the specified element.
 *
 * @returns The data value previously at index
 * @throws IndexOutOfBoundsException if the index is out of range
 *         (<tt>index &lt; 0 || index &gt;= size()</tt>)
 */
public E set(int index, E newValue)
  {
    ...
  }


/**
 * Returns the index of the first occurrence of the specified element
 * in this list, or -1 if this list does not contain the element.
 */
public int indexOf(Object obj)
  {
    int index = 0;
```

```
    if (obj == null)
      {
        for (Node<E> node = first;  node != null;  node = node.next)
          {
            if (node.data == null)
                return index;
            else
                index++;
          }
      }
    else
      {
        for (Node<E> node = first;  node != null;  node = node.next)
          {
            if (obj.equals(node.data))
                return index;
            else
                index++;
          }
      }

    return -1;
}


/**
 * Returns <tt>true</tt> if this list contains no elements.
 */
public boolean isEmpty()
  {
    ...
  }


/**
 * Removes the element at the specified position in this list.  Shifts
 * any subsequent elements to the left (subtracts one from their indices).
 *
 * @returns the element previously at the specified position
 *
 * @throws IndexOutOfBoundsException if the index is out of range
 *         (<tt>index &lt; 0 || index &gt;= size()</tt>)
 */
public E remove(int index)
  {
    ...
  }


/**
 * Returns the number of elements in this list.
 */
public int size()
  {
    ...
  }
```

```
/**
 * Returns an iterator over the elements in this list in proper sequence.
 */
@Override
public Iterator<E> iterator()
  {
    ...
  }


/**
 * Returns a string representation of this list.
 */
@Override
public String toString()
  {
    ...
  }


/*
 * Compares the specified object with this list for equality. Returns true
 * if and only if both lists contain the same elements in the same order.
 */
@Override
@SuppressWarnings("rawtypes")
public boolean equals(Object obj)
  {
    if (obj == this)
        return true;

    if (!(obj instanceof LinkedList))
        return false;

    // cast obj to a linked list
    LinkedList listObj = (LinkedList) obj;

    // compare elements in order
    Node<E> node1 = first;
    Node    node2 = listObj.first;

    while (node1 != null && node2 != null)
      {
        // check to see if data values are equal
        if (node1.data == null)
          {
            if (node2.data != null)
                return false;
          }
        else
          {
            if (!node1.data.equals(node2.data))
                return false;
          }

        node1 = node1.next;
        node2 = node2.next;
```

```
        }

      return node1 == null && node2 == null;
    }


  /**
   * Returns the hash code value for this list.
   */
  @Override
  public int hashCode()
    {
      int hashCode = 1;
      Node<E> node = first;

      while (node != null)
        {
          E obj = node.data;
          hashCode = 31*hashCode + (obj == null ? 0 : obj.hashCode());
          node = node.next;
        }

      return hashCode;
    }
}
```