LinkedList.java

```java
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)
          {
            this.data = data;
            this.next = 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()
          {
            return nextElement != null;
          }


        /**
         * Returns the next element in the list.
         *
         * @throws NoSuchElementException if the iteration has no next element.
         */
        @Override
        public E next()
          {
            if(this.hasNext()) {
              E rtnval = nextElement.data;
              nextElement = nextElement.next;
              return rtnval;
            }
            else throw new NoSuchElementException();
          }

        // 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)
```

```java
      {
        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()
      {
        first = null;
        last = null;
        size = 0;
      }


    /**
     * 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)
      {
        Node<E> curr = first;
        Node<E> newNode = new Node<>(element);

        // If inserting at index 0, empty or not
        if(index == 0) {
          newNode.next = first;
          first = newNode;
          if(size == 0)
            last = newNode;
        }

        // If not empty and inserting in last index
        else if(index == this.size) {
          last.next = newNode;
          last = newNode;
        }

        // If inserting anywhere else
        else {
          checkIndex(index);
          curr = getNode(index - 1);
          newNode.next = curr.next;
          curr.next = newNode;
        }
        ++size;
      }


    /**
     * 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)
      {
        Node<E> curr = getNode(index);
        E rtnval = curr.data;
        curr.data = newValue;
        return rtnval;
      }


    /**
     * 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()
```

```java
      {
        return this.size == 0;
      }


    /**
     * 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)
      {
        checkIndex(index);
        E rtnval;
        // If removing first item, empty or not
        if(index == 0) {
          rtnval = first.data;
          if(size > 0)
            first = first.next;
          else {
            first = null;
            last = null;
          }
        }
        // If not empty and removing last item
        else if(index == this.size-1) {
          rtnval = last.data;
          last = this.getNode(index-1);
          last.next = null;
        }
        // If removing any other item
        else {
          Node<E> curr = getNode(index - 1);
          rtnval = curr.next.data;
          curr.next = curr.next.next;
        }
        --size;
        return rtnval;
      }


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


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


    /**
     * Returns a string representation of this list.
     */
    @Override
    public String toString()
      {
        String rtnval = "[";
        if(size > 0) {
          Node<E> curr = this.first;
          while(curr != null) {
            if(curr.data == null) {
              rtnval += "null, ";
              curr = curr.next;
            }
            else {
              rtnval += curr.data.toString() + ", ";
              curr = curr.next;
            }
          }
          rtnval = rtnval.substring(0, rtnval.length()-2);
        }
        rtnval += "]";
        return rtnval;
      }


    /*
     * 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
```

```java
              {
                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;
      }
  }
```