```
package linkedLists;

import java.util.AbstractSequentialList;
import java.util.Iterator;
import java.util.ListIterator;
import java.util.NoSuchElementException;

/**
 * Partial implementation of the List interface based on doubly-linked nodes with
 * dummy nodes at the head and tail. This sample code includes
 * a complete implementation of the ListIterator.
 *
 * Overridden versions of add(item), add(pos, item), contains(obj) and get(pos),
 * are also provided as examples.
 */
public class DoublyLinkedList<E> extends AbstractSequentialList<E>
{
  /**
   * Reference to dummy node at the head.
   */
  private Node head;

  /**
   * Reference to dummy node at the tail.
   */
  private Node tail;

  /**
   * Number of elements in the list.
   */
  private int size;

  /**
   * Constructs an empty list.
   */
  public DoublyLinkedList()
  {
    head = new Node(null);
    tail = new Node(null);
    head.next = tail;
    tail.previous = head;
    size = 0;
  }

  @Override
  public boolean add(E item)
  {
    Node temp = new Node(item);
    link(tail.previous, temp);
    ++size;
    return true;
  }

  @Override
  public void add(int pos, E item)
  {
    if (pos < 0 || pos > size) throw new IndexOutOfBoundsException("" + pos);

    Node temp = new Node(item);
```

```
    Node predecessor = findNodeByIndex(pos - 1);
    link (predecessor, temp);
    ++size;
  }

  @Override
  public E get(int pos)
  {
    // Note ">="
    if (pos < 0 || pos >= size) throw new IndexOutOfBoundsException("" + pos);
    return findNodeByIndex(pos).data;
  }

  // alternate version of get using iterator
//  @Override
//  public E get(int pos)
//  {
//    return listIterator(pos).next();
//  }


  @Override
  public boolean contains(Object obj)
  {
    Node current = head.next;
    while (current != tail)
    {
      E e = current.data;
      if (e == obj || e != null && e.equals(obj))
      {
        return true;
      }
      current = current.next;
    }
    return false;
  }

  // alternate version of contains using iterator
//  @Override
//  public boolean contains(Object obj)
//  {
//    for (E e : this)
//    {
//      if (e == obj || e != null && e.equals(obj))
//      {
//        return true;
//      }
//    }
//    return false;
//  }


  @Override
  public Iterator<E> iterator()
  {
    return new DoublyLinkedIterator();
  }
```

```
@Override
public ListIterator<E> listIterator()
{
  return new DoublyLinkedIterator();
}

@Override
public ListIterator<E> listIterator(int pos)
{
  return new DoublyLinkedIterator(pos);
}

@Override
public int size()
{
  return size;
}

/**
 * Inserts newNode into the list after current without
 * updating size.
 * Precondition: current != null, newNode != null
 */
private void link(Node current, Node newNode)
{
  newNode.previous = current;
  newNode.next = current.next;
  current.next.previous = newNode;
  current.next = newNode;
}

/**
 * Removes current from the list without
 * updating size.
 */
private void unlink(Node current)
{
  current.previous.next = current.next;
  current.next.previous = current.previous;
}

/**
 * Returns the Node whose index is pos, which
 * will be head if pos = -1 and tail if pos = size
 * Precondition: size >= pos >= -1
 */
private Node findNodeByIndex(int pos)
{
  if (pos == -1) return head;
  if (pos == size) return tail;

  // inv: position of current is count
  Node current = head.next;
  int count = 0;
  while (count < pos)
  {
    current = current.next;
    ++count;
  }
```

```
  return current;
}

/**
 * Doubly-linked node type for this class.
 */
private class Node
{
  public E data;
  public Node next;
  public Node previous;

  public Node(E data)
  {
    this.data = data;
  }
}

/**
 * Implementation of ListIterator for this class
 */
private class DoublyLinkedIterator implements ListIterator<E>
{
  // Class invariants:
  // 1) logical cursor position is always between cursor.previous and cursor
  // 2) after a call to next(), cursor.previous refers to the node just returned
  // 3) after a call to previous() cursor refers to the node just returned
  // 4) index is always the logical index of node pointed to by cursor
  // 5) direction is BEHIND if last operation was next(),
  //      AHEAD if last operation was previous(), NONE otherwise

  // direction for remove() and set()
  private static final int BEHIND = -1;
  private static final int AHEAD = 1;
  private static final int NONE = 0;

  private Node cursor;
  private int index;
  private int direction;

  public DoublyLinkedIterator(int pos)
  {
    if (pos < 0 || pos > size) throw new IndexOutOfBoundsException("" + pos);

    cursor = findNodeByIndex(pos);
    index = pos;
    direction = NONE;
  }

  public DoublyLinkedIterator()
  {
    this(0);
  }

  @Override
  public void add(E item)
  {
    Node temp = new Node(item);
    link(cursor.previous, temp);
```

```
  ++index;
  ++size;
  direction = NONE;
}

@Override
public boolean hasNext()
{
  return index < size;
}

@Override
public boolean hasPrevious()
{
  return index > 0;
}

@Override
public E next()
{
  if (!hasNext()) throw new NoSuchElementException();

  E ret = cursor.data;
  cursor = cursor.next;
  ++index;
  direction = BEHIND;
  return ret;
}

@Override
public int nextIndex()
{
  return index;
}

@Override
public E previous()
{
  if (!hasPrevious()) throw new NoSuchElementException();

  cursor = cursor.previous;
  --index;
  direction = AHEAD;
  return cursor.data;
}

@Override
public int previousIndex()
{
  return index - 1;
}

@Override
public void remove()
{
  if (direction == NONE)
  {
    throw new IllegalStateException();
  }
```

```
      else
      {
        if (direction == AHEAD)
        {
          // remove node at cursor and move to next node
          Node n = cursor.next;
          unlink(cursor);
          cursor = n;
        }
        else
        {
          // remove node behind cursor and adjust index
          unlink(cursor.previous);
          --index;
        }
      }
      --size;
      direction = NONE;
    }

    @Override
    public void set(E item)
    {
      if (direction == NONE)
      {
        throw new IllegalStateException();
      }

      if (direction == AHEAD)
      {
        cursor.data = item;
      }
      else
      {
        cursor.previous.data = item;
      }
    }
  }
}
```