```
public class DoublyLinkedList<E>
implements Cloneable {
   //------------nested Node
class------------
   private static class Node<E> {
      private E element;      //
reference to stored element
      private Node<E> prev;    //
reference to previous element
      private Node<E> next;    //
reference to next element

   /** The constructor that creates
a node */
   public Node(E e, Node<E> p,
Node<E> n) {
      element = e;
      prev = p;
      next = n;
   }

   // methods
   /** getter for the element */
   public E getElement() {
      return element;
   }

   /** getter for previous node in list
*/
   public Node<E> getPrev() {
      return prev;
   }

   /** getter for next node in list */
```

```
    public Node<E> getNext() {
        return next;
    }

    /** setter for previous node */
    public void setPrev(Node<E> p) {
        prev = p;
    }

    /** setter for the next node */
    public void setNext(Node<E> n) {
        next = n;
    }
} //------------end of nested node
class------------

// instance variables of
DoublyLinkedList
private Node<E> header;     // head
sentinel
private Node<E> trailer;    // tail
sentinel
private int size = 0;       // number of
elements in list

/** List constructor */
public DoublyLinkedList() {
    header = new Node<E>(null, null,
null);      // create header
    trailer = new Node<E>(null,
header, null);   // header precedes
trailer

header.setNext(trailer);
// trailer follows header
```

```
}

// access methods
/** Returns the size of the doubly
linked list */
public int getSize() {
    return size;
}

/** Tests whether the linked list is
empty */
public boolean isEmpty() {
    return size == 0;
}

/** Returns but does not remove
the first element in the list */
public E first() {
    if (isEmpty()) {
        return null;
    } else {
        return
header.getNext().getElement();  //
return first node's element
    }
}

/** Returns but does not remove
the last element in the list */
public E last() {
    if (isEmpty()) {
        return null;
    } else {
        return
trailer.getPrev().getElement();  //
```

```
return last node's element
    }
}

//update methods
/** Adds element e to the front of
the list */
public void addFirst(E e) {
    addBetween(e, header,
header.getNext());
}

/** Adds element e to the back of
the list */
public void addLast(E e) {
    addBetween(e, trailer.getPrev(),
trailer);
}

/** Removes and returns the first
element of the list */
public E removeFirst() {
    if (isEmpty()) {
        return null;
    } else {
        return
remove(header.getNext());
    }
}

/** Removes and returns the last
element of the list */
public E removeLast() {
    if (isEmpty()) {
        return null;
```

```
    } else {
        return
remove(trailer.getPrev());
    }
}

// private update helpers
/** Does the heavy lifting for
adding an element to the list */
private void addBetween(E e,
Node<E> predecessor, Node<E>
successor) {
    // create and link a new node
    Node<E> newest = new
Node<>(e, predecessor,
successor);
    predecessor.setNext(newest);
    successor.setPrev(newest);
    size++;
}

/** Does the heavy lifting for
removing an element from the list */
private E remove(Node<E> node) {
    Node<E> predecessor =
node.getPrev();
    Node<E> successor =
node.getNext();
    predecessor.setNext(successor);
    successor.setPrev(predecessor);
    size--;
    return node.getElement();
}
// equals and clone methods
    /** Equals method currently
```

```
assumes that the list must be of
the same
     *  type in order to be equal. This
means that a doubly linked list will
     *  not be equal to a circularly
linked list or a singly linked list even
     *  if the elements are identical.
Because of type erasure in Java,
we have
     *  to use Objects and casts to
handle any type rather than
generics. */

@SuppressWarnings({ "rawtypes" }
)
   public boolean equals(Object o) {
       if (o == null) {
           return false;
       }

       // at this point, the classes
have to be the same.
       if (getClass() != o.getClass()) {
           return false;
       }

       DoublyLinkedList other =
(DoublyLinkedList) o;  // use non-
parameterized type (erasure)

       // the size must be the same
for them to be equal
       if (size != other.size) {
           return false;
       }
```

```
        Node walkA =
header;                         // traverse
primary list
        Node walkB =
other.header;                   //
traverse secondary list

        // We don't want to compare
the trailers, so size - 1
        for(int i = 0; i < size; i++) {
            if (!
walkA.getElement().equals(walkB.g
etElement())) {
                return false; // mismatch
            }
            walkA = walkA.getNext();
            walkB = walkB.getNext();
        }
        return true;            // if we
reach this, then they are equal.
    }

    /** The clone method that
performs a deep clone of the list */

@SuppressWarnings("unchecked")
    public DoublyLinkedList<E>
clone() throws
CloneNotSupportedException {
        // always use inherited
Object.clone() to create initial copy
        DoublyLinkedList<E> other =
(DoublyLinkedList<E>)
super.clone(); // safe cast
```

```
        if (size > 0) {                    // we
need independent node chain
            other.header = new
Node<>(null, null, null);
            other.trailer = new
Node<>(null, other.header, null);

other.header.setNext(other.trailer);
            Node<E> walk =
header.getNext();   // walk through
remainder of original list
            Node<E> otherWalk =
other.header;
            for(int i = 0; i < size; i++)
{           // make new node storing
same element
                Node<E> newest = new
Node<>(walk.getElement(), null,
null);

otherWalk.setNext(newest);   // link
previous node to this one
                otherWalk = newest;

otherWalk.setPrev(newest);   // link
node back to the previous one
                walk = walk.getNext();
            }
        }
        return other;
    }

    /** Test driver for the circularly
linked list class */
```

```
@SuppressWarnings({ "rawtypes",
"unchecked" })
   public static void main(String
args[]) {
      DoublyLinkedList theList =
new DoublyLinkedList();
      DoublyLinkedList clonedList;
      theList.addFirst(1);
      theList.addFirst(2);
      theList.addLast(3);
      try {
         clonedList = theList.clone();
         System.out.println("Original
List values");
         while(theList.first() != null) {

System.out.println(theList.removeFi
rst());
         }

         System.out.println("Cloned
List values");
         while(clonedList.first() !=
null) {

System.out.println(clonedList.remo
veFirst());
         }

System.out.println(theList.equals(cl
onedList));
      } catch
(CloneNotSupportedException e) {
         System.err.println("I AM
ERROR: List didn't clone.");
```

```
        e.printStackTrace();
    }
```