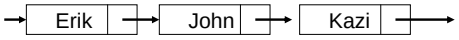
Chapter 7

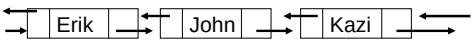
Linked Lists

A linked list is a List **object (that is, an object in a class that implements the** List **interface) in which the following property is satisfied:**

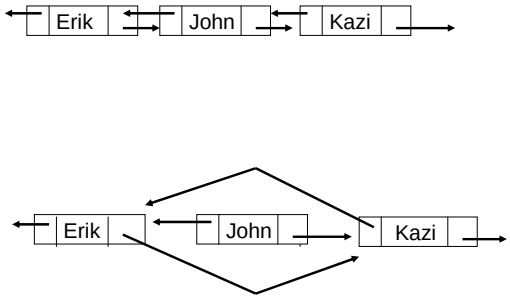
Each element is contained in an object, called an Entry **object, that also includes a reference, called a *link*, to the** Entry **object that holds the next element in the list.**



If each Entry object also includes a link to the Entry object that holds the previous element in the list, we have a *doubly linked* list.



The beauty of a linked list is that insertions and removals can be made without moving any elements: Only the links are altered.



We will devote most of this chapter to the study of the LinkedList class, a doubly-linked data structure that is part of the Java collections framework. As a warm-up to that class, we start with a toy class, SinglyLinkedList.

The Entry **class will be a nested class within** SinglyLinkedList**:**

```
protected class Entry<E>
{
   E element;
   Entry<E> next;
} // class Entry
```

Then methods in SinglyLinkedList **can access the** Entry **fields.**

```
public class SinglyLinkedList<E>
                        implements List<E>
{

    // We'll fill this part in shortly


   protected class Entry<E>
   {
      E element;
      Entry<E> next;
   } // class Entry

} // class SinglyLinkedList
```

We will specify and define just enough methods for you to get a feel for the SinglyLinkedList **class**

```
/**
 * Initializes this SinglyLinkedList object to be empty,
 *  with elements to be of type E.
 *
 */
public SinglyLinkedList()
```

```
/**
 *  Determines if this SinglyLinkedList object has no
 *  elements.

 *  @return true – if this SinglyLinkedList object has
 *                 no elements; otherwise, false.
 *
 */
public boolean isEmpty ()
```

```
/**
 *  Inserts a specified element at the front of this
 *  SinglyLinkedList object.
 *
 *  @param element – the element to be inserted (at
 *              the front).
 *
 *  @return true.
 *
 */
public boolean add (E element)
```

If true is always returned, why bother??

```
/**
 *  Determines the number of elements in this
 *  SinglyLinkedList object.
 *  The worstTime(n) is O(n).
 *
 *  @return the number of elements.
 *
 */
public int size ()
```

```
/**
 *  Determines if this SinglyLinkedList object contains
 *  a specified element. The worstTime(n) is O(n).
 *
 *  @param obj – the specified element being sought.
 *
 *  @return true - if this SinglyLinkedList object
 *              contains obj; otherwise, false.
 */
public boolean contains (Object obj)
```

Warning: Make sure the element class implements an equals method.

```
SinglyLinkedList<String> linked =
                new SinglyLinkedList<String>();

linked.add ("yes");
linked.add ("no");
if (linked.size() == 2)
   if (linked.contains ("maybe"))
      linked.add ("true");
  else
      linked.add ("maybe");
```

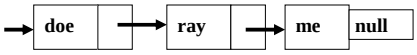
What does linked **consist of now?**

Fields and implementation of the SinglyLinkedList **class:**

How can we indicate the end of a SinglyLinkedList **object?**

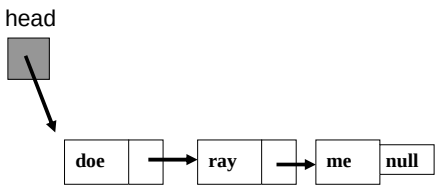
How can we indicate the beginning of a SinglyLinkedList **object?**

To indicate the end of a SinglyLinkedList object, the next field in the last Entry should be null.



To indicate the beginning of a SinglyLinkedList **object, we need a reference to the first** Entry:

```
protected Entry<E> head;
```

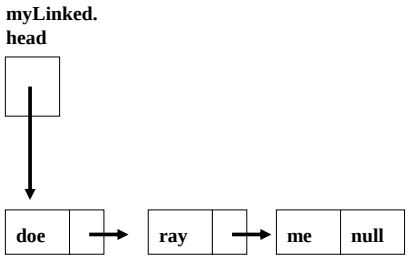



```
public SinglyLinkedList( )
{
      head == null;
} // default constructor
```

```
public boolean isEmpty( )
{
      return head == null;
} // method isEmpty
```

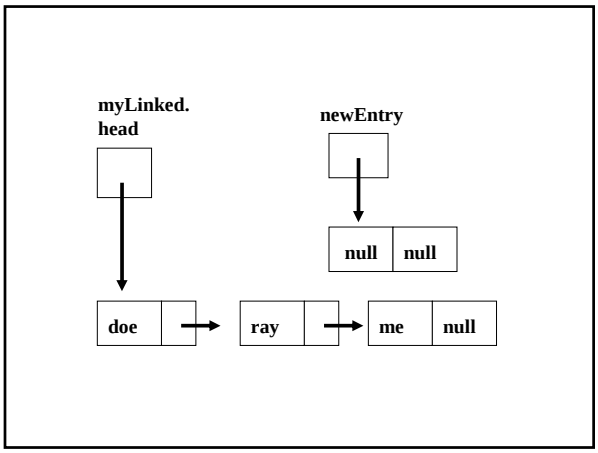
For the add (E element) **method, let's start with some examples:**

```
SinglyLinkedList<String> myLinked =
                  new SinglyLinkedList<String> ( );

myLinked.add ("me");
myLinked.add ("ray");
myLinked.add ("doe");
```

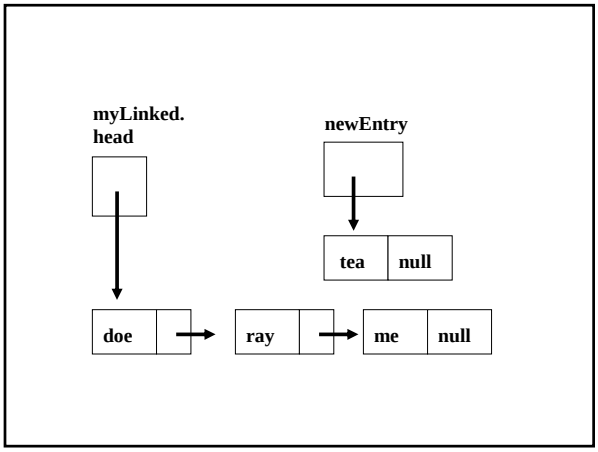



```
myLinked.add ("tea");
```

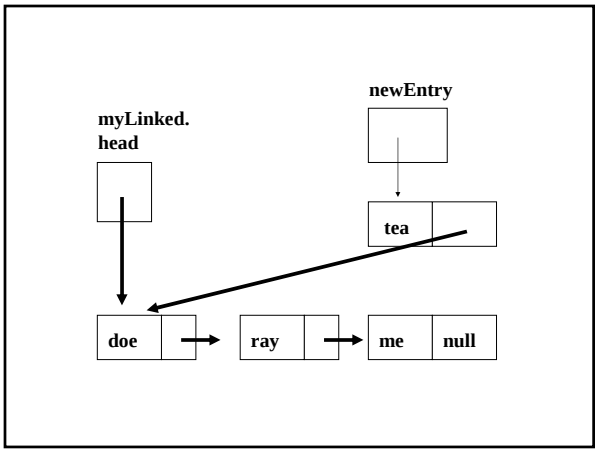
```
// Construct a new Entry:
Entry<E> newEntry = new Entry<E>( );
```



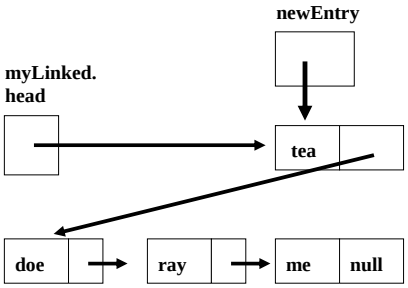
newEntry.element = element;



newEntry.next = head;



head = newEntry;



Here is the complete definition of add**:**

```
public boolean add (E element)
{
      Entry<E> newEntry = new Entry<E>();
      newEntry.element = element;
      newEntry.next = head;
      head = newEntry;
      return true;
} // method add
```

```
public int size( )
{
      ???

int count = 0;
```

```
Entry<E> current;

for loop:
```

Initialization:

```
current = head;
```

Continuation condition:

```
current != null
```

Incrementation:

```
current = current.next;
```

```
public int size( )
{
   int count = 0;

   for (Entry<E> current = head; current != null;
                    current = current.next)
          count++;
   return count;
} // method size
```

```
public boolean contains (Object obj)
{
     if (obj instanceof E)
          for (Entry<E> current = head; current != null;
                         current = current.next)
               if (obj.equals (current.element))
                    return true;
     return false;
} // method contains
```

The actual definition is slightly more complicated because it is legal for an element to be null.

If the element's class does not implement equals**, there could be trouble. Why?**

The Object **class's** equals **method tests for equality of references, not objects!**

Exercise:

Define the get **method:**

```
/**
 * Returns the element at a specified index.
 * The worstTime(n) is O(n).
 *
 *  @param index – the specified index.
 *  @return – the element at index.
 *  @throws IndexOutOfBoundsException – if index
 *               is less than 0 or greater than size() – 1.
 *
 */
public E get (int index)
```

Iterators

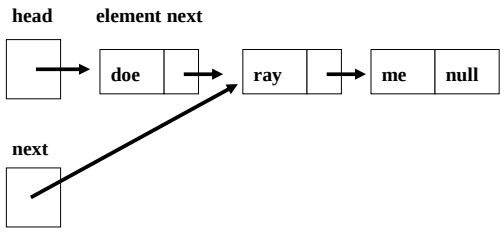
An *iterator* is an object that enables a user to loop through a collection without accessing the collection's fields.

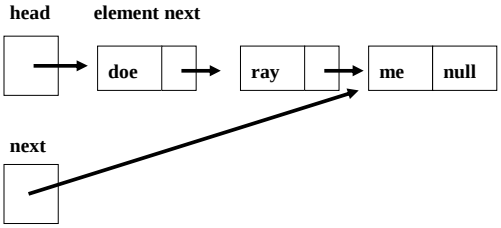
```
public class SinglyLinkedList<E> implements List<E>
                    extends AbstractCollection<E>
{
        // We did this part earlier

        protected class Entry<E>
        {
            E element;
            Entry<E> next;
        } // class Entry

        protected class SinglyLinkedListIterator
                    implements Iterator<E>
        {
                protected Entry<E> next;
                …
        } // class SinglyLinkedListIterator
} // class SinglyLinkedList
```

```
public SinglyLinkedListIterator( )
{
    next = head;
} // default constructor
```

```
public boolean hasNext( )
{
        return next != null;
} // method hasNext
```

To motivate the definition of the next method, consider the following example:





Returned: "ray"

So we need to save next.element**, advance** next**, and return the saved element.**

```
public E next( )
{
        E theElement = next.element;
        next = next.next;   // ???
        return theElement;
} // method next
```

The next **field in the** SinglyLinkedListIterator **object is a reference to an** Entry **object that has a** next **field.**

```
public E next( )
{
        E theElement = next.element;
        next = next.next;
        return theElement;
} // method next
```

The next **field in the** SinglyListIterator **object is a reference to an** Entry **object that has a** next **field.**

Finally, we define an iterator **method in the** SinglyLinkedList **class:**

```
/**
 *  Returns a SinglyLinkedListIterator object to iterate
 *  over this SinglyLinkedList object.
 *
 */
public Iterator<E> iterator()
{
    return new SinglyLinkedListIterator();
} // method iterator
```

What is returned?

A reference to an object in the SinglyLinkedListIterator **class, which implements the** Iterator **interface.**

Example: Print each element of myLinked **whose value is greater than 5.0:**

```
Iterator<Double> itr = myLinked.iterator();
while (itr.hasNext())
{
    double d = itr.next();  // unboxing
    if (d > 5.0)
        System.out.println (d);
}
```

Exercise: In the preceding example, use an enhanced for **statement instead of an iterator.**

Now, onto the class

LinkedList**!**

For the most part, the LinkedList **class has the same method headings as the** ArrayList **class, but those classes have different time estimates for some methods.**

For example,

public E get (int index)

public E set (int index, E element)

worstTime(*n*) is linear in *n*

versus constant for an ArrayList**.**

Sometimes LinkedList **versions are faster:**

public boolean add (E element)

The worstTime(*n*) is constant, versus linear in *n* for an ArrayList **object because of the possibility of re-sizing.**

Basically, to get *to* a position in a LinkedList **takes linear-in-*n* time, but once you get there, you can remove or insert in constant time.**

That magnifies the importance of iterators, because once an iterator is positioned somewhere in the collection, you can insert or remove in constant time.

Here are the method headings for all of the methods in the LinkedList class:

1. **public** LinkedList( )
2. **public** LinkedList (Collection<? **extends** E> c)
3. **public boolean** add (E element)
4. **public void** add (**int** index, E element)
5. **public void** addAll (Collection<? **extends** E> c)
6. **public boolean** addAll (**int** index, Collection c)
7. **public boolean** addFirst (E element)
8. **public boolean** addLast (E element)
9. **public void** clear( ) // worstTime(n) is constant
10. **public** Object clone( )
11. **public boolean** contains (Object obj)
12. **public boolean** containsAll (Collection<?> c)

13. **public boolean** equals (Object obj)
14. **public** E get (**int** index)
15. **public** E getFirst ( )
16. **public** E getLast ( )
17. **public int** hashCode( )
18. **public int** indexOf (Object obj)
19. **public boolean** isEmpty( )
20. **public** Iterator<E> iterator( )
21. **public int** lastIndexOf (Object obj)
22. **public** ListIterator<E> listIterator( )
23. **public** ListIterator<E> listIterator (**final int** index)
24. **public boolean** remove (Object obj)

25. **public** E remove (**int** index)
26. **public boolean** removeAll (Collection<?> c)
27. **public** E removeFirst()
28. **public** E removeLast()
29. **public boolean** retainAll (Collection<?> c)
30. **public** E set (**int** index, E element)
31. **public int** size( )
32. **public** List<E> subList (**int** fromIndex, **int** toIndex)
33. **public** Object[ ] toArray( )
34. **public** <T> T[ ] toArray (T[ ] a)
35. **public** String toString()

Example: Here is a processInput (String s) **method that starts by converting** s **to int** n **and then**

0. **Constructs a** LinkedList **of** Double **objects.**
   public LinkedList( )
1. **In a loop with** i **going from** 0 **to** n – 1**, appends new** Double **(**i**) to the** LinkedList**.**
   public boolean add (E element)
2. **Inserts new** Double (1.4) **at index** n / 3**.**
   public void add (**int** index, E element)
3. **Removes the element at index** 2n / 3.
   public E remove (**int** index)
4. **Multiplies the middle element by** 3.5.
   public E get (**int** index)
   public E set (**int** index, E element)
5. **Prints out the** LinkedList**; public** String toString()

```
public void processInput (String s)
{
    int n = Integer.parseInt (s);
    List<Double> myList = new LinkedList<Double>();
    for (int i = 0; i < n; i++)
        myList.add (i + 0.0);
    myList.add (n / 3, 1.4);
    myList.remove (2 * n / 3);
   double d = (myList.get (n / 2)) * 3.5;
    myList.set (n / 2, d);
    System.out.println (myList) ;
} // method processInput
```

Does this look familiar?

The output, just as before, is

[0.0, 1.0, 2.0, 1.4, 3.0, 14.0, 6.0, 7.0, 8.0, 9.0]

Methods in the embedded ListItr **class:**

IT1. **public void** add (E element)
IT2. **public boolean** hasNext( )
IT3. **public boolean** hasPrevious( )
IT4. **public** E next( )
IT5. **public int** nextIndex( )
IT6. **public** E previous( )
IT7. **public int** previousIndex( )
IT8. **public void** remove( )
IT9. **public void** set (E element)

For each method, worstTime (n) is constant!

In the LinkedList **class:**

```
/**
 * Returns a ListIterator object that is positioned
 * at the beginning of this LinkedList.
 */
public ListIterator<E> listIterator( )


/**
 *  Returns a ListIterator object that is positioned at index
 *  in this LinkedList, or beyond the last element if
 *  index = size( ). The worstTime(n) is O(n).
 *
 * @throws IndexOutOfBoundsException – if index
 *              is less than 0 or greater than size()
 *
 */
public ListIterator<E> listIterator (final int index)
```

In the ListItr **class:**

```
/**
 * Retreats this ListIterator to the previous element,
 * and returns that element.
 *
 *@return the element that was at the index one less
 *              than where this ListIterator was positioned
 *              when this call was made.
 *
 * @throws NoSuchElementException – if this ListIterator
 *              was positioned at index 0 when this call
 *              was made.
 */
public E previous( )
```

```
LinkedList<String> myList = new LinkedList<String>( );
myList.add ("Brian");
myList.add ("Clayton");
myList.add ("Eric");

ListIterator<String> itr = myList.listIterator( );
System.out.println (itr.next( ) + " " + itr.next( ) + " " +
                        itr.previous( ));
```

Recall that the next( ) **method returns the element where the iterator is currently positioned, and advances to the next position.**

But the previous( ) **method first retreats to the previous position, and returns that element.**

So the output is

Brian Clayton Clayton

To print myList **in reverse order:**

```
itr = myList.listIterator (myList.size( ));
while (itr.hasPrevious( ))
      System.out.println (itr.previous( ));
```

Another ListIterator **method:**

```
/**
 *  Inserts an element into the LinkedList in front of
 *  the element that would be returned by next() and
 *  in back of the element that would be returned by
 *  previous().
 *
 */
public void add (E element);
```

```
LinkedList<Double>myList = new LinkedList<Double>();
myList.add (0.0);
myList.add (1.0);

ListIterator<Double> itr = myList.listIterator();
itr.next();
itr.add (0.8);
```

The LinkedList **would now have**

0.0, 0.8, 1.0

Another ListIterator **method:**

```
/**
 *  Removes the last returned element.
 *
 */
public void remove();
```

```
List<String> myList = new LinkedList<String>();
 myList.add ("Kimotho");
 myList.add ("King");
 myList.add ("Kleinbach");
 myList.add ("Kolba");
 ListIterator<String> itr = myList.listIterator();
 itr.add ("zero-th");
 itr.next();
 itr.add ("second");
 itr.next();
 itr.remove();
 itr.previous();
 itr.remove();
 System.out.println (myList);
```

User's guide for choosing ArrayList **or** LinkedList**:**

If the application entails a lot of accessing and/or modifying elements at widely varying indexes, an ArrayList **will be much faster than a** LinkedList**.**

If a large part of the application consists of iterating through a list and making insertions and/or removals during the iterations, a LinkedList **will be much faster than an** ArrayList**.**

Fields and implementation of the LinkedList **class:**

There are two fields:

```
private transient int size = 0;

private transient Entry<E> header =
                              new Entry (null, null, null);
```

Fields and implementation of the LinkedList **class:**

There are two fields

Field not saved if object is serialized

```
private transient int size = 0;

private transient Entry header =
                              new Entry (null, null, null);
```

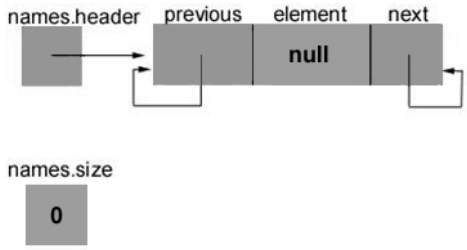
```
private static class Entry<E>
{
  E element;
  Entry<E> next;
  Entry<E> previous;

  Entry (<E> element, Entry<E> next, Entry<E> previous)
  {
        this.element = element;
        this.next = next;
        this.previous = previous;
  } // constructor

} // class Entry
```

```
public LinkedList()
{
        header.next = header.previous = header;
}
```

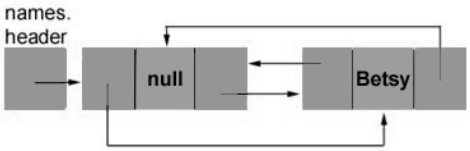
For example,

```
List<String> names = new LinkedList<String>();
```

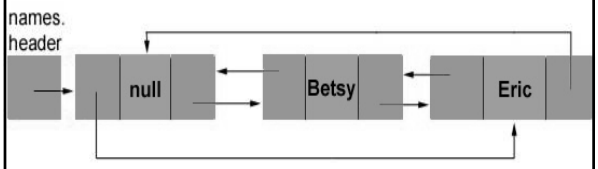
Empty LinkedList



LinkedList **with on element**



LinkedList **with two elements**



The significance of the header entry is that there is always an entry in back of an in front of any entry.

That simplifies insertions and removals.

Details of insertion into a LinkedList**:**

names.add (1, "Don");

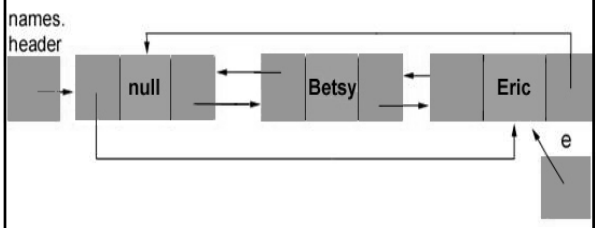
The method heading for this method is:
public void add (**int** index, E element)

This add **method calls**
addBefore (element, entry (index));

and the heading for addBefore **is**
private Entry<E> addBefore (E element, Entry<E> e)

So "Don" will be inserted in front of the entry at index 1.

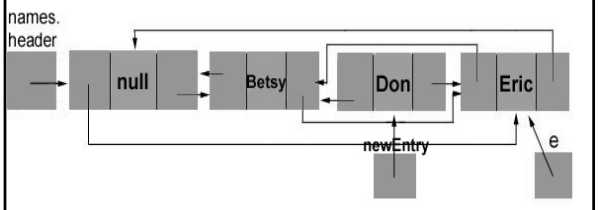
Start



Step 1: Code

```
// insert newEntry in front of e
Entry<E> newEntry =
            new Entry(element, e, e.previous);
```

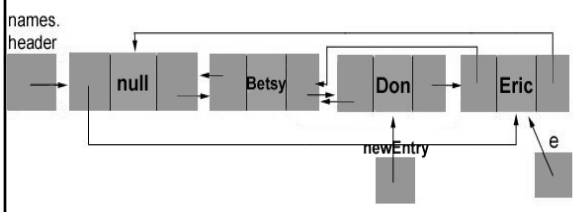
Step 1



Step 2:

```
// make newEntry follow its predecessor
newEntry.previous.next = newEntry;
```

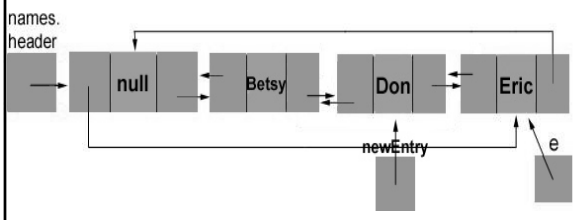
Step 2



Step 3: Code

```
// make newEntry precede its successor, e
newEntry.next.previous = newEntry;
```

Step 3



The same strategy works for

names.add (0, “Kalena”);
// inserts between header entry and entry at // index 0

names.add (names.size(), “Hana”);
// inserts between entry at index size() – 1
// and header entry

Group exercise:

Determine the output from the following:

```
LinkedList<String> myList = new LinkedList<String>();
myList.add ("a");
myList.add ("b");
myList.add ("c");
myList.add ("d");
myList.add ("e");
myList.add (2, "r");
myList.remove (4);
ListIterator<String> itr = myList.listIterator (3);
 itr.previous();
 itr.add ("x");
 itr.next();
 itr.remove();
 itr = myList.listIterator (myList.size());
 while (itr.hasPrevious())
        System.out.println (itr.previous());
```

Application of

Linked Lists

A Line Editor

Line editor: **A program that manipulates test, line by line.**

- **First line = line 0**
- **One line is designated the current line.**
- **Each editing command begins with $.**

For now, there are only four editing commands:

1. $Insert

Each subsequent line, up to the next editing command, is inserted into the test *in front of* the current line (at back of text if no current line)

Example: Suppose the text is empty

```
$Insert
Mairzy Doats and Dozy Doats
And Liddle Lamzy Divy
A Kiddle Edivy Too, Wouldn't You?

Now the text is:
    Mairzy Doats and Dozy Doats
    And Liddle Lamzy Divy
    A Kiddle Edivy Too, Wouldn't You?
>
```

Another example: Suppose the text is

```
    In Xanadu did Kubla Khan
    A stately pleasure dome decree,
>   Down to a sunless sea.

$Insert
Where Alph the sacred river ran,
Through caverns measureless to man,
```

Now the text is

In Xanadu did Kubla Khan
A stately pleasure dome decree,
Where Alph the sacred river ran,
Through caverns measureless to man,
> Down to a sunless sea.

2. $Delete m n

Each line in the text between lines m and n, inclusive, will be deleted. The current line is now just *after* the last line deleted.

Example: Suppose the text is

I must go down to the sea again,
To the lonely sea and the sky.
And all I ask is a tall ship,
> And a star to steer her by.
And the wheel's kick and the wind's song,
And the white sails shaking,
And a grey mist on the sea's face,
And a grey dawn breaking.

$Delete 2 4

Now the text is

I must go down to the sea again,
To the lonely sea and the sky.
> And the white sails shaking,
And a grey mist on the sea's face,
And a grey dawn breaking.

Suppose the next command is

$Delete 3 3

Now the text is

I must go down to the sea again,
To the lonely sea and the sky.
And the white sails shaking,
> And a grey dawn breaking.

Possible errors in the command line:

Error: The first line number is greater the second.

Error: The first line number is less than 0.

Error: The 2nd line number is greater than the last line number.

Error: The command is not followed by two integers.

3. $Line m

Line m becomes the current line in the text.

Example: Suppose the text is

I must go down to the sea again,
To the lonely sea and the sky.
And all I ask is a tall ship,
> And a star to steer her by.
And the wheel's kick and the wind's song,
And the white sails shaking,
And a grey mist on the sea's face,
And a grey dawn breaking.

$Line 8

Now the text is

I must go down to the sea again,
To the lonely sea and the sky.
And all I ask is a tall ship,
And a star to steer her by.
And the wheel's kick and the wind's song,
And the white sails shaking,
And a grey mist on the sea's face,
And a grey dawn breaking.
>

Possible errors in the command line?

4. $Done

The text is printed and the execution of the editor is finished.

Line Editor Applet

http://www.cs.lafayette.edu/~collinsw/lineeditor/line.html

For flexibility, we will separate editing from input/output.

Then, for example, the input could come from the keyboard, or from a file. And that choice would not affect the editing.

So we will create two classes:

EditorDriver**: To handle input and output**

Editor**: To handle editing**

Error message will be thrown as exceptions. For example,

```
throw new RuntimeException (M_LESS_THAN_ZERO);
```

Error messages will be thrown in Editor **methods and caught in** EditorDriver **methods.**

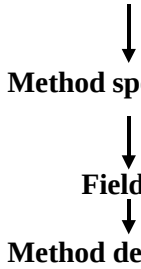
For the Editor **class, how do we start?**

Fields or methods?

The chicken or the egg?

Proceed as follows:

Responsibilities of the class (what the class will provide to users)



Responsibilities:

--To determine whether the line contains a legal command, an illegal command, or a line of text

--To carry out each of the four commands

```
/**
 * Interprets a given line, and the result of carrying out
 * (if a command) or inserting that line has been returned.
 *
 *  @param s – the line to be interpreted.
 *  @return the result of carrying out or inserting s
 */
public String interpret (String s)

/**
 *  Inserts a line into the text in front of the current line.
 *
 * @param s – the line to be inserted.
 * @throws RuntimeException – if s has more than
 *              MAX_LINE_LENGTH characters.
protected void insert (String s)
```

```
/**
 *  Deletes lines, in a range specified by line numbers,
 *  from the text.
 *
 *  @param m – the first index in the range
 *  @param n – the last index in the range
 *  @throws RuntimeException – if m is less than 0 or
 *              greater than n or if n is greater than or
 *              equal to the number of lines in the text.
 */
protected void delete (int m, int n)

/**
 *  Makes the line specified by an index the current line.
 *  @param m – the line number specified
 *  @throws RuntimeException – if m is less than 0 or
 *              greater than the number of lines of text.
 */
protected void line (int m)
```

```
/**
 *  Terminates the editor and returns the text.
 *
 *  @return the text
 */
protected String done( )
```

Fields?

```
protected LinkedList<String> text;

protected ListIterator<String> current;

protected boolean inserting;
```

```
public Editor( )
{
      text = new LinkedList<String>();
      current = text.listIterator( );
      inserting = false;
} // default constructor
```

```
public String interpret (String s)
{
      // If the line s doesn't start with a $, insert the
      // line if inserting is true, and otherwise throw an
      // exception. If the line does start with a $,
      // perform the appropriate command, or throw
      // an exception if there is no such command.

 } // method interpret
```

To insert a line:

```
protected void insert (String s)
{
    if (s.length( ) > MAX_LINE_LENGTH)
       throw new RuntimeException (LINE_TOO_LONG +
                                   MAX_LINE_LENGTH);
    current.add (s);
} // insert
```

```
protected void tryToDelete (StringTokenizer tokens) {

    // Try to tokenize m and n into integers: throw an
    // exception if appropriate.  Otherwise, delete lines m
    // through n.

} // method tryToDelete
```

```
protected void delete (int m, int n)
{
  if (m > n)
       throw new RuntimeException (FIRST_GREATER);
  if (m < 0)
       throw new RuntimeException
                   (FIRST_LESS_THAN_ZERO);
  if (n >= text.size( ))
       throw new RuntimeException
                   (SECOND_TOO_LARGE);
  current = text.listIterator (m);
  for (int i = m; i <= n; i++)
  {
     current.next();
     current.remove ();
  } // for
} // method delete
```

The tryToSetLine **and** line **methods are similar to, but simpler than, the** tryToDelete **and** delete **methods. For example, here is the** line **method:**

```
protected void line (int m)
{
      if (m < 0)
            throw new RuntimeException
                                    (M_LESS_THAN_ZERO);
      if (m > text.size())
            throw new RuntimeException
                                    (M_TOO_LARGE);
      current = text.listIterator (m);
} // method line
```

```
protected String done ( ) {

      // Iterate through the text, and print each line.

} // method done
```

What about '>'? To determine whether itr **is positioned at the current line, we cannot check either** itr == current **or** itr.next().equals (current.next())

Why not?

itr == current **won't work because these are references to iterators, not to elements, nor even to entries.**

itr.next( ).equals (current.next( )) **won't work because there may be copies of the current element.**

```
public String done ( )
{
      final String FINAL_TEXT_MESSAGE =
            "\n\nHere is the final text:\n";
      String s = FINAL_TEXT_MESSAGE;

      ListIterator itr = text.listIterator( );

      while (itr.hasNext( ))
            if (itr.nextIndex( ) == current.nextIndex( ))
                  s = s + ">  " + itr.next() + '\n';
            else
                  s = s + "   " + itr.next() + '\n';
      if (!current.hasNext())
            s = s + ">   " + '\n';
      return s;
} // method done
```

The EditorDriver **class**

The EditorDriver **class has** openFiles() **and** editText() **methods. These are virtually identical to the** openFiles() **and** testVeryLongInt() **methods from the** VeryLongDriver **class in Chapter 6.**

The editText() **method reads in each line in the input file. The line is interpreted, and exceptions are caught and printed. For the $Done command, the text is printed.**

```
public void editText(){
        Editor editor = new Editor();
        String line = new String(),
               result = new String();
        while (true) {
            try {
                line = fileReader.readLine();
                if (line == null)
                    break;
                fileWriter.println (line);
                result = editor.interpret (line);
            } // try
            catch (RuntimeException e)
            {
                    fileWriter.println (e);
            } // catch RuntimeException
            catch (IOException e)
            {
                    System.out.println (e);
            } // catch IOException
            if (line.equals (Editor.DONE_COMMAND))
                    fileWriter.println (result);
        } // while
} // method editText
```

Group exercise: In the previous slide, we had

if (line.equals (Editor.DONE_COMMAND))

What is the complete declaration of the identifier DONE_COMMAND **in the** Editor **class?**