Lists

- References:
  - Text book : Chapters 12, 13, 14
  - Oracle/Sun Java Tutorial :
    http://java.sun.com/j2se/1.5.0/docs/guide/collections/index.html
    http://download.oracle.com/javase/6/docs/api/java/util/List.html

1. ADT Lists

- consider a list containing items of the same type in specific order, example : dates, addresses, flight numbers, phone numbers, to do list, grocery list, customer records, student records etc. // Unlike ADT bags, order of items are important in lists
- Possible operations: add new item, remove an item, remove all items, replace an item, look at an item, find an item, count how many items, display all items and etc
- ListInterface

```
/* Entries in the list have positions that begin with 1. */

public interface ListInterface < T >
{
   /* Adds a new entry to the end of this list.*/
   public void add (T newEntry);

   /* Adds a new entry at a specified position within this list. */
   public boolean add (int newPosition, T newEntry);

   /* Removes the entry at a given position from this list. */
   public T remove (int givenPosition);
```

```
   /* Removes all entries from this list. */
   public void clear ();

   /* Replaces the entry at a given position in this list. */
   public boolean replace (int givenPosition, T newEntry);

   /* Retrieves the entry at a given position in this list. */
   public T getEntry (int givenPosition);

   /* Sees whether this list contains a given entry. */
   public boolean contains (T anEntry);

   /* Gets the length of this list. */
   public int getLength ();

   /* Sees whether this list is empty. */
   public boolean isEmpty ();

   /* Retrieves all entries that are in this list in the order in which
   they occur in the list. */
   public T [] toArray ();

} // end ListInterface
```

- Examples

Example 1 : assume we have a list MyList

MyList.add(a), MyList.add(b), MyList.add(c),
MyList.add(2,d), MyList.add(1,e), MyList.remove(3)

Empty → a → a,b → a,b,c → a,d,b,c → e,a,d,b,c → e,a,b,c

Note the content of MyList and position of data

Example 2:

```
public class ListClient
{
        public static void main(String[] args)
        {
                testList();
        }  // end main

        public static void testList()
        {
                ListInterface<String> runnerList = new AList<String>();
                //  runnerList has only methods in ListInterface

                runnerList.add("16"); // winner
                runnerList.add(" 4"); // second place
                runnerList.add("33"); // third place
                runnerList.add("27"); // fourth place
                System.out.println (runnerList.toArray());
        } // end testList
} // end ListClient
```

- Java Class Library: The Interface **List**

  - The standard package contains a list interface – called List
  - Position start with 0
  - Several methods provided:

boolean	**add**(E e) Appends the specified element to the end of this list (optional operation).
void	**add**(int index, E element) Inserts the specified element at the specified position in this list (optional operation).
void	**clear**() Removes all of the elements from this list (optional operation).
boolean	**contains**(Object o) Returns true if this list contains the specified element.
E	**get**(int index) Returns the element at the specified position in this list.
boolean	**remove**(Object o) Removes the first occurrence of the specified element from this list, if it is present (optional operation).
boolean	**isEmpty**() Returns true if this list contains no elements.
int	**size**() Returns the number of elements in this list.

Complete listing :
http://download.oracle.com/javase/6/docs/api/java/util/List.html

2. List Implementation That Use Arrays

- Use fixed size array + dynamic expansion array to implement List
- See text book for using Java API class Vector to implement list
- Strategy: always fill slots from beginning and no "empty" slot in between 2 used slots

  Suppose I have the following letters from position 1 to 10 in alphabetical order, and some available spaces:

  A D G I J K O P Q S _ _ _ _ _ _ _ _

  If I want to add a new letter E to 3rd position, then I need shift down all letters from position 3 and onward by one position :

  A D E G I J K O P Q S _ _ _ _ _ _ _

  If I want to remove letter I from 5th position, then I need shift up all letters from position 6 and onward by one position :

  A D E G J K O P Q S _ _ _ _ _ _ _ _

- Array List source code:

  Assertions (a good, simple way to check program logic in runtime)

  - An assertion is a Boolean statement that is true (or should be true) If it is false, something is wrong with your program
  - It can be a simple comment, which allows reader to understand your program. In Java, you may use an "assert" statement.
  - By default, assert statements are disabled during execution. You may turn on using –ea option, i.e. java –ea myprogram

```
import java.util.Arrays;
/**
A class that implements a list of objects by using an array.
The list is never full.
@author Frank M. Carrano
*/

public class AList < T > implements ListInterface < T >
{
      // same as Bags in blue color
      private T [] list; // array of list entries
      private int numberOfEntries;
      private static final int DEFAULT_INITIAL_CAPACITY = 25;

   public AList ()
   {
      this (DEFAULT_INITIAL_CAPACITY); // call next constructor
   } // end default constructor

   public AList (int initialCapacity)
   {
      numberOfEntries = 0;
      // the cast is safe because the new array contains null entries
      @ SuppressWarnings ("unchecked")
         T [] tempList = (T []) new Object [initialCapacity];
      list = tempList;
   } // end constructor

   public void add (T newEntry)
   {    // same as in Bag , use dynamic expansion if current array is full } // end add
```

```
public boolean add (int newPosition, T newEntry)
{
   boolean isSuccessful = true;
   if ((newPosition >= 1) && (newPosition <= numberOfEntries + 1))
   {
      ensureCapacity ();   //double size if it is needed
      makeRoom (newPosition);
      list [newPosition - 1] = newEntry;
      numberOfEntries++;
   }
   else
      isSuccessful = false;
   return isSuccessful;
} // end add

// Makes room for a new entry at newPosition.
// Precondition: 1 <= newPosition <= numberOfEntries+1;
// numberOfEntries is lists length before addition.
private void makeRoom (int newPosition)
{
   assert (newPosition >= 1) && (newPosition <= numberOfEntries + 1);
   int newIndex = newPosition - 1;
   int lastIndex = numberOfEntries - 1;
   // move each entry to next higher index, starting at end of
   // list and continuing until the entry at newIndex is moved
   for (int index = lastIndex ; index >= newIndex ; index--)
      list [index + 1] = list [index];
} // end makeRoom
```

```
public T remove (int givenPosition)
{
   T result = null; // return value
   if ((givenPosition >= 1) && (givenPosition <= numberOfEntries))
   {
      assert !isEmpty ();
      result = list [givenPosition - 1]; // get entry to be removed
      // move subsequent entries toward entry to be removed,
      // unless it is last in list
      if (givenPosition < numberOfEntries)
         removeGap (givenPosition);
      numberOfEntries--;
   } // end if
   return result; // return reference to removed entry, or
   // null if either list is empty or givenPosition
   // is invalid
} // end remove

// Shifts entries that are beyond the entry to be removed to the
// next lower position.
// Precondition: 1 <= givenPosition < numberOfEntries;
// numberOfEntries is lists length before removal.
private void removeGap (int givenPosition)
{
   assert (givenPosition >= 1) && (givenPosition < numberOfEntries);
   int removedIndex = givenPosition - 1;
   int lastIndex = numberOfEntries - 1;
   for (int index = removedIndex ; index < lastIndex ; index++)
      list [index] = list [index + 1];
} // end removeGap

public void clear ()
{  // same as bag }

public boolean contains (T anEntry)
{   // same as bag } // end contains

public int getLength ()
{  // same as bag }
```

```
  public boolean replace (int givenPosition, T newEntry)
  {
     boolean isSuccessful = true;
     if ((givenPosition >= 1) && (givenPosition <= numberOfEntries))
     {
        assert !isEmpty ();
        list [givenPosition - 1] = newEntry;
     }
     else
        isSuccessful = false;
     return isSuccessful;
  } // end replace

  public T getEntry (int givenPosition)
  {
     T result = null; // result to return
     if ((givenPosition >= 1) && (givenPosition <= numberOfEntries))
     {
        assert !isEmpty ();
        result = list [givenPosition - 1];
     } // end if
     return result;
  } // end getEntry

  public boolean isEmpty ()
  {  // same as bag }

  public T [] toArray ()
  {   // same as bag }

  // Doubles the size of the array list if it is full.
  private void ensureCapacity ()   // this is defined before, in ADT Bags
  {
     if (numberOfEntries == list.length)
        list = Arrays.copyOf (list, 2 * list.length);
  } // end ensureCapacity

  // This class will define two more private methods that will be discussed later.
} // end AList
```

- Java Class Library: ArrayList and Vector

  - Has two classes that use dynamic array expansion
    ArrayList, Vector

    Found in **java.util**
    Implement interface **List**
    Defined in terms of a generic type

  - Ref:
    http://download.oracle.com/javase/6/docs/api/java/util/Vector.html

    http://download.oracle.com/javase/6/docs/api/java/util/ArrayList.html

  - ArrayList is a newer class than Vector. Both are about the same

```
import java.util.*;

public class ArrayListDemo{
   public static void main(String[] args) {

        ArrayList<Object> arl=new ArrayList<Object>();
        System.out.println("The content of arraylist is: " + arl);
        System.out.println("The size of an arraylist is: " + arl.size());

        Integer i1=new Integer(10);
        Integer i2=new Integer(20);
        Integer i3=new Integer(30);
        String st=new String("hello");
        arl.add(i1);
        arl.add(i2);
        arl.add(st);
        arl.add(i2);
        arl.add(i3);
        System.out.println("The content of arraylist is: " + arl);
        System.out.println("The size of an arraylist is: " + arl.size());

        arl.remove(3);
        System.out.println("The content of arraylist is: " + arl);
        System.out.println("The size of an arraylist is: " + arl.size());
```

```
        Object arl2=arl.clone();
        System.out.println("The content of arraylist is: " + arl);
        System.out.println("The size of an arraylist is: " + arl.size());
        System.out.println("The content of arl2 is: " + arl2);
      }
    }
```

Output:

```
The content of arraylist is: []
The size of an arraylist is: 0

The content of arraylist is: [10, 20, hello, 20, 30]
The size of an arraylist is: 5

The content of arraylist is: [10, 20, hello, 30]
The size of an arraylist is: 4

The content of arraylist is: [10, 20, hello, 30]
The size of an arraylist is: 4
The content of arl2 is: [10, 20, hello, 30]
```

- Pros and Cons of Array Use for the ADT List

  When using an array or vector

  - Retrieving an entry is fast
  - Adding an entry at the end of the list is fast
  - Adding or removing an entry that is between other entries requires shifting elements in the array
  - Increasing the size of the array or vector requires copying elements

3. List Implementation That Links Data (Linked List)

References: Oracle/Sun Java Tutorial :
http://java.sun.com/j2se/1.5.0/docs/guide/collections/index.html
http://download.oracle.com/javase/6/docs/api/java/util/LinkedList.html

- Explain program in next few slides

```
public class LList < T > implements ListInterface < T >
{
   private Node firstNode; // reference to first node
   private int numberOfEntries;     // number of entries in list

  // constructor
   public LList ()
   {
      clear ();
   } // end default constructor

  // clear list. Set firstNode to NULL
   public final void clear ()
   {
      firstNode = null;
      numberOfEntries = 0;
   } // end clear

   < Implementations of methods go here….. >

   private class Node // private inner class
   {
     // as given previously
   } // end Node
} // end LList
```

- Add a new node to various positions

Possibilities for placement of a new node Node:

i. Before all current node // first position
ii. Between two existing nodes // need to know NodeBefore, NodeAfter
iii. After all current nodes // last position

i. Add a new node to the beginning (covered in ADT Bags)

Two cases:
Case 1: The list is empty. First = NewNode
Case 2: The list is not empty, i.e. one or more
NewNode.Next = FirstNode
First = NewNode

iii. Add a new node to the end

Two cases:
Case 1: The list is empty. First = NewNode
Case 2: The list is not empty
Steps:

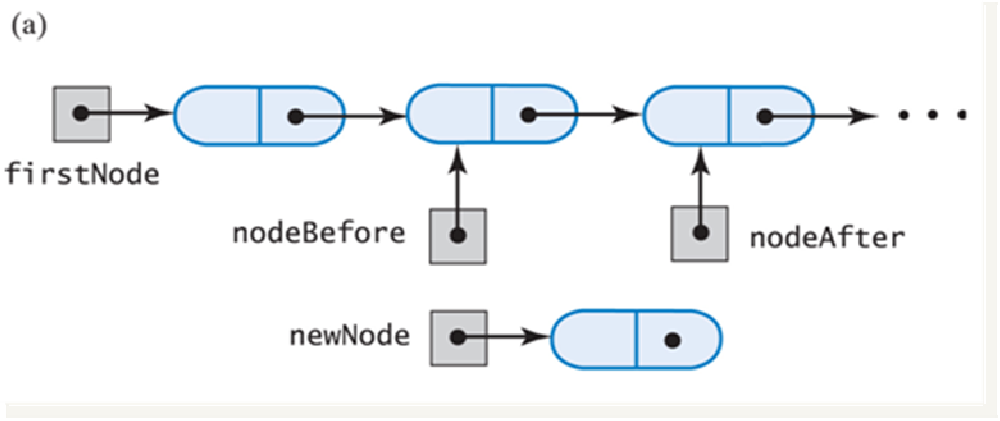
Start from First node // get id from First
Get Next node ** // following links to get last node
Goto next node
Repeat ** until you reach Last node

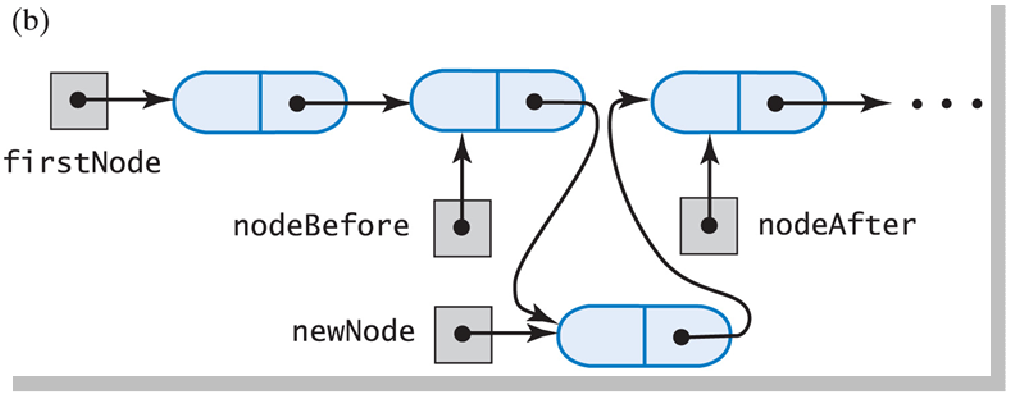
Set LastNode.next = NewNode
// now, new node is the last node

ii. Between two existing nodes // need to know NodeBefore, NodeAfter

Steps to place node in between two existing nodes:

locate the id of NodeBefore // similar steps as ** above
// NodeBefore.next is NodeAfter
NewNode.next = NodeAfter // new node links to NodeAfter
NodeBefore.next = NewNode





Main private method

```
/** Task: Returns a reference to the node at a given position.
   * Precondition: List is not empty; 1 <= givenPosition <= length. */
   private Node getNodeAt (int givenPosition)
   {
      assert !isEmpty () &&
         (1 <= givenPosition) && (givenPosition <= length);

      // use temporary node currentNode to traverse
      // list starting at firstNode
      Node currentNode = firstNode;
      // traverse the list to locate the desired node
      for (int counter = 1 ; counter < givenPosition ; counter++)
         currentNode = currentNode.next;

      // currentList now refers to node at givenPosition
      assert currentNode != null;
      return currentNode;
   } // end getNodeAt
```

- Add to the end of list

```
   public void add (T newEntry)
   {
      // get new node to hold newEntry
      Node newNode = new Node (newEntry);
      // case I
      if (isEmpty ())
         firstNode = newNode;
      else // Case II: add to end of nonempty list
      {
         Node lastNode = getNodeAt (length);
         lastNode.next = newNode; // make last node reference new node
      } // end if
      numberOfEntries ++;
   } // end add
```

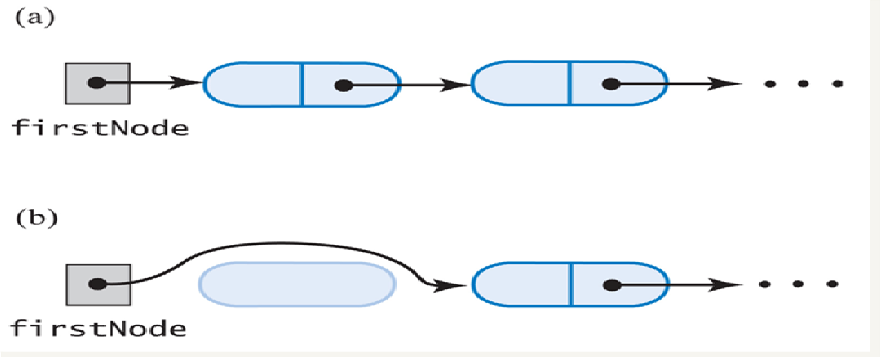
- Add to any position

```
public boolean add (int newPosition, T newEntry)
{
   boolean isSuccessful = true;
   if ((newPosition >= 1) && (newPosition <= length + 1))
   {
      // get new node to hold newEntry
      Node newNode = new Node (newEntry);

      if (isEmpty () || (newPosition == 1))  // empty or 1st position
      {
         newNode.next = firstNode;
         firstNode = newNode;
      }
      else // not empty and newPosition > 1
         {
            Node nodeBefore = getNodeAt (newPosition - 1);
            Node nodeAfter = nodeBefore.next;
            newNode.next = nodeAfter;
            nodeBefore.next = newNode;
         } // end if
      numberOfEntries ++;
   }
   else
      isSuccessful = false;
   return isSuccessful;
} // end add
```

- Remove a node from various positions

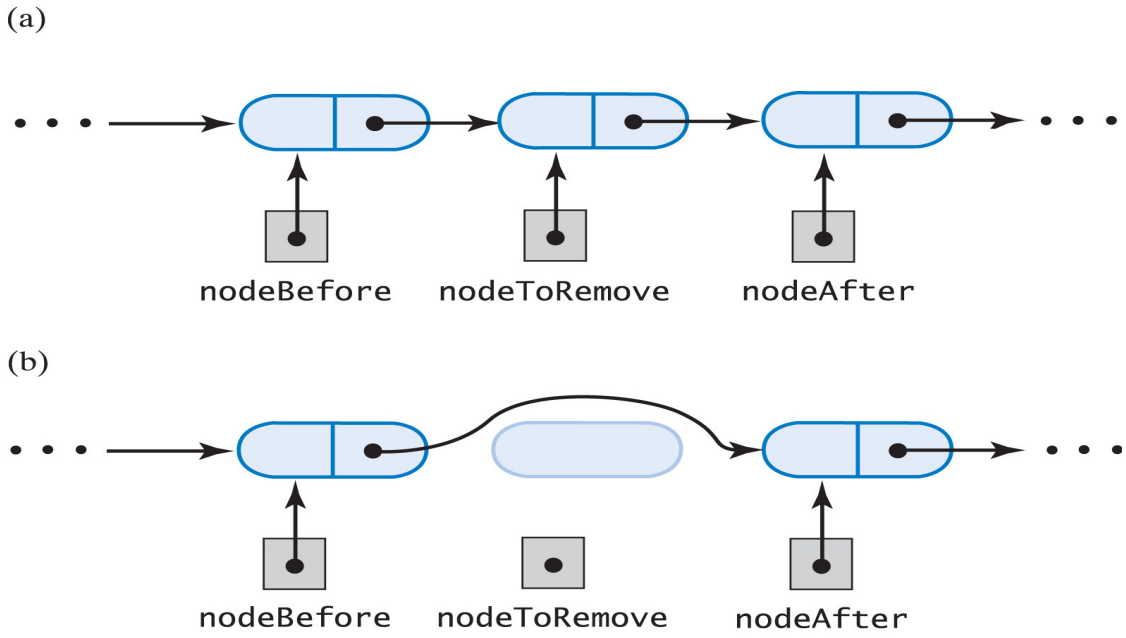
i. Remove first node

First = First.Next // set First to second node.



ii. Remove in between two nodes, nodeBefore & nodeAfter
Remove last node

nodeBefore.Next = nodeToRemove.Next



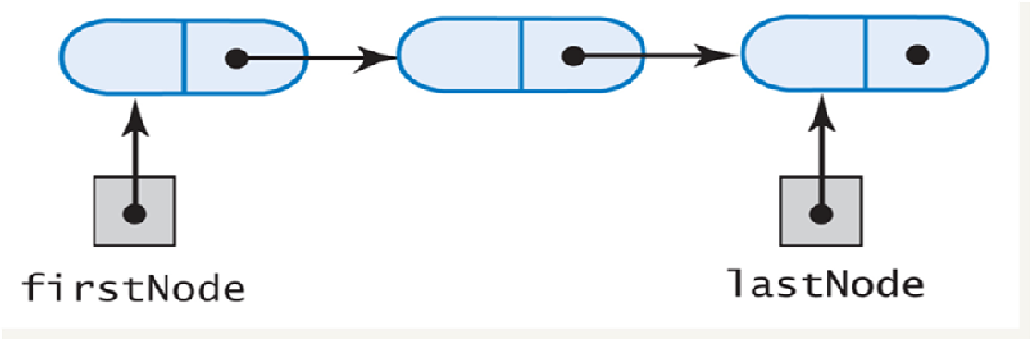
```
// remove a node at givenPosition and return its entry (data)
public T remove (int givenPosition)
{
   T result = null; // return value
   if ((givenPosition >= 1) && (givenPosition <= length))
   {
      assert !isEmpty ();
      if (givenPosition == 1) // case 1: remove first entry
      {
         result = firstNode.data; // save entry to be removed
         firstNode = firstNode.next;
      }
      else // case 2: givenPosition > 1
      {
         Node nodeBefore = getNodeAt (givenPosition - 1);
         Node nodeToRemove = nodeBefore.next;
         Node nodeAfter = nodeToRemove.next;
         nodeBefore.next = nodeAfter; // disconnect the node to be removed
         result = nodeToRemove.data; // save entry to be removed
      } // end if
      numberOfEntries --;
   } // end if

   return result; // return removed entry, or
   // null if operation fails
} // end remove
```

- Other methods are listed in text book, some are similar to ADT Bags, others use getNodeAt() methods to accomplish tasks

- Variation of linked lists

In general, previous implementation is called singly linked list, i.e. each node has one reference to next node.

(i) Singly linked list: maintain a pointer that always keeps track of the end of the chain, i.e. tail reference (used in ADT Queues)



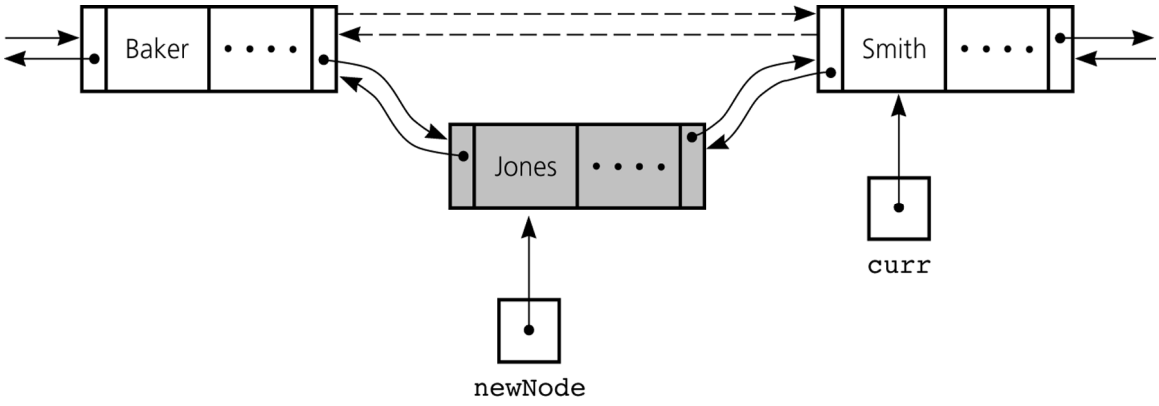
(ii) Circular linked list, i.e. last node next refers to first node (instead of NULL)

(iii) List with dummy node, i.e. physical first node is a dummy node. It never change and do not contain valid data. So, first "real data" is stored in 2nd physical node

(iv) Doubly linked list, i.e. each node has two references, one to precede node (prev) and one to next node (next). Java class library: LinkedList class is implemented using this strategy.

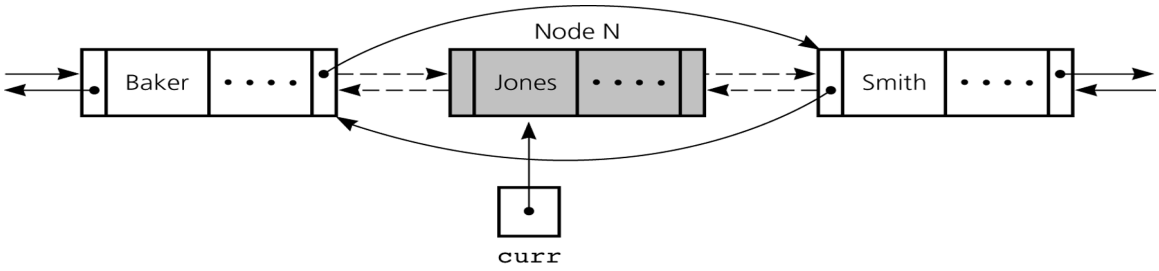
Outline of adding a newNode (Jones) before currentNode (Smith):

```
newNode.setNext(curr);
newNode.setPrev(curr.getPrev());
curr.setPrev(newNode);
newNode.getPrev().setNext(newNode);
```



Outline of deleting a currentNode (Jones):

```
curr.getPrev().setNext(curr.getNext());
curr.getNext().setPrev(curr.getPrev());
```



- Java Class Library: The Class **LinkedList**
  - The standard java package java.util contains the class LinkedList

    http://download.oracle.com/javase/6/docs/api/java/util/LinkedList.html

  - This class implements the interface List
  - Contains additional methods

```
• addFirst()
• addLast()
• removeFirst()
• removeLast()
• getFirst()
• getLast()
```

  - Sample usage:

```
import java.util.*;

class ListTest
{
        private static String colors[] =
        { "black", "yellow", "green", "blue", "violet", "silver" };

        private static String colors2[] =
        { "gold", "white", "brown", "blue", "gray", "silver" };

        public static void printList( List<String> listRef )
        {
           System.out.println( "\nlist: " );
           for ( int k = 0; k < listRef.size(); k++ )
                System.out.print( listRef.get( k ) + " " );
           System.out.println();
        }

        public static void uppercaseStrings( List<String> listRef2 )
        {
           //This is a another way to traverse list items, will cover in next chapter
           ListIterator<String> listIt = listRef2.listIterator();
           while ( listIt.hasNext() )
           {
```

```
            String s = listIt.next(); // get item
            listIt.set( s.toUpperCase() );
        }
    }

    public static void removeItems( List<String> listRef3, int start, int end )
    {
            // sublist is a special methods, it does not create nodes
            List<String> sl = listRef3.subList( start, end );
            sl.clear(); // remove nodes from listRef3
    }

    public static void main( String args[] )
    {
        LinkedList<String> link = new LinkedList<String>();
        LinkedList<String> link2 = new LinkedList<String>();

        for ( int k = 0; k < colors.length; k++ )
        {
            link.add( colors[ k ] );
            link2.add( colors2[ k ] ); // same length as colors
        }

        link.addAll( link2 ); // concatenate lists
        link2 = null; // release resources
        printList( link );
        uppercaseStrings( link );
        printList( link );
        System.out.print( "\nDeleting elements 4 to 6..." );
        removeItems( link, 4, 7 );
        printList( link );
    }
}
```

Output:

```
$ java ListTest1

list:
black yellow green blue violet silver gold white brown blue gray silver

list:
BLACK YELLOW GREEN BLUE VIOLET SILVER GOLD WHITE BROWN
BLUE GRAY SILVER

Deleting elements 4 to 6...
list:
BLACK YELLOW GREEN BLUE WHITE BROWN BLUE GRAY SILVER
```