A modular program is easier to write, read, and modify

An ADT is a collection of data and a set of operations on that data

Data structures are part of an ADT's implementation

ADT and data structures are not same

The ADT sorted list maintains item in sorted order. It also inserts and deletes items by their values, not their positions

You can use ADT operations in an application without the distraction of implementation details

Encapsulation hides implementation details

A class's data fields may or may not be public. However, making them private is a recommended best practice. Also, by default all members in a class are private unless a programmer specifies otherwise.

Java package provides a way to group related classes together

Access to a package's classes can be public or restricted

"super" keyword -> Used in a constructor of a subclass to call the constructor of the superclass

ADT list contains some operations such as insert, delete, and retrieve

A reference variable as a data field of a class has the default value null

A local reference variable has no default value

The "new" is a Java keyword

An array of objects is actually an array of references to the objects

A traverse operation visits each node in the linked list

```java
//An example of shallow copy
public class Student implements Cloneable {

    // reference objects
    private Subject subj;
    private String name;

    // constructor
    public Student(String s, String sub) {
        name = s;
        subj = new Subject(sub);
    }

    public Subject getSubj() {
        return subj;
    }

    public String getName() {
        return name;
    }

    public void setName(String s) {
        name = s;
    }

    // override clone method
    public Object clone() {
        // shallow copy
        try {
            // directly call clone method from super class
            return super.clone();
        } catch (CloneNotSupportedException e) {
            return null; // garbage collection for null
        }
    }
}
```

```java
public class StudentDeepCopy implements Cloneable {

    // reference objects
    private Subject subj;
    private String name;

    public StudentDeepCopy(String s, String sub) {
        name = s;
        subj = new Subject(sub);
    }

    public Subject getSubj() {
        return subj;
    }

    public String getName() {
        return name;
    }

    public void setName(String s) {
        name = s;
    }

    // override clone method
    public Object clone() {
        // deep copy
        // create a new object, two object are independent
        StudentDeepCopy s = new StudentDeepCopy(name, subj.getName());
        return s;
    }
}
```

Equality operators compare values of reference variables, not the objects that they reference. The "equals" method compares objects field by field

A ".java" file cannot have more than one public class

A new node can be inserted into a linked list

A specified node can be deleted from a linked list

Modularity keeps the complexity of a large program manageable by systematically controlling the interaction of its components. Also, isolates errors and eliminates redundancies

Procedural abstraction separates the purpose and use of a module from its implementation

A module's specifications should detail how the module behaves and identify details that can be hidden within the module

Abstract Data Types

- Typical operations on data
  - Add data to a data collection
  - Remove data from a data collection
  - Ask questions about the data in a data collection
- Data abstraction
  - Asks you to think *what* you can do to a collection of data independently of *how* you do it
  - Allows you to develop each data structure in relative isolation from the rest of the solution
  - A natural extension of procedural abstraction

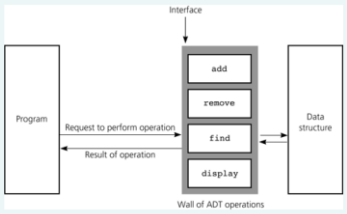


Figure 4-4
A wall of ADT operations isolates a data structure from the program that uses it

- ADT List operations
  - Create an empty list
  - Determine whether a list is empty
  - Determine the number of items in a list
  - Add an item at a given position in the list
  - Remove the item at a given position in the list
  - Remove all the items from the list
  - Retrieve (get) the item at a given position in the list
- Items are referenced by their position within the list

Axioms for the ADT List
- (aList.createList()).size() = 0
- (aList.add(i, x)).size() = aList.size() + 1
- (aList.remove(i)).size() = aList.size() – 1
- (aList.createList()).isEmpty() = true
- (aList.add(i, item)).isEmpty() = false
- (aList.createList()).remove(i) = error
- (aList.add(i, x)).remove(i) = aList
- (aList.createList()).get(i) = error
- (aList.add(i, x)).get(i) = x
- aList.get(i) = (aList.add(i, x).get(i+1)
- aList.get(i+1) = (aList.remove(i)).get(i)

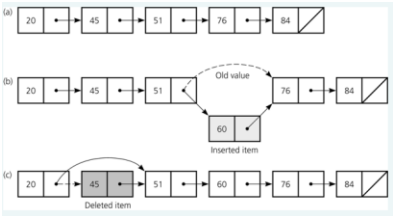


Figure 5-1
a) A linked list of integers; b) insertion; c) deletion

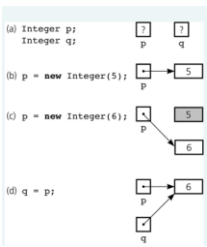


Figure 5-3a-d
a) Declaring reference variables; b) allocating an object; c) allocating another object, with the dereferenced object marked for garbage collection

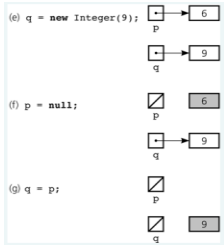


Figure 5-3e-g
e) allocating an object; f) assigning *null* to a reference variable; g) assigning a reference with a *null* value

Resizable Arrays

- The number of references in a Java array is of fixed size
- Resizable array
  - An array that grows and shrinks as the program executes
  - An illusion that is created by using an allocate and copy strategy with fixed-size arrays
- `java.util.Vector` class
  - Uses a similar technique to implement a growable array of objects

- Using the Node class

```
Node n = new Node (new Integer(6));
Node first = new Node (new Integer(9), n);
```

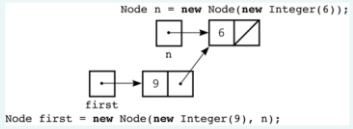


Figure 5-7
Using the *Node* constructor to initialize a data field and a link value

- Data field `next` in the last node is set to `null`
- `head` reference variable
  - References the list's first node
  - Always exists even when the list is empty

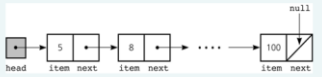


Figure 5-8
A *head* reference to a linked list

- `head` reference variable can be assigned `null` without first using `new`
  - Following sequence results in a lost node

```
head = new Node(); // Don't really need to use new here
head = null; // since we lose the new Node object here
```

Figure 5-9
A lost node

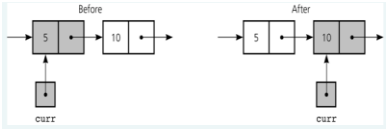


Figure 5-10
The effect of the assignment *curr = curr.next*

- Deleting the first node is a special case

```
head = head.next;
```

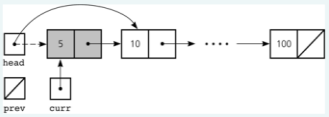


Figure 5-12
Deleting the first node

- To delete node N which `curr` references
  - Set `next` in the node that precedes N to reference the node that follows N

```
prev.next = curr.next;
```

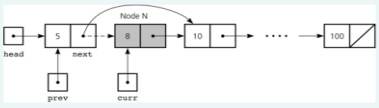


Figure 5-11
Deleting a node from a linked list

- To return a node that is no longer needed to the system

```
curr.next = null;
curr = null;
```

- Three steps to delete a node from a linked list
  - Locate the node that you want to delete
  - Disconnect this node from the linked list by changing references
  - Return the node to the system

- To create a node for the new item

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

- To insert a node between two nodes

```
newNode.next = curr;
prev.next = newNode;
```

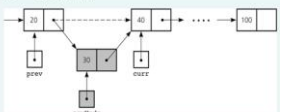


Figure 5-13
Inserting a new node into a linked list

- To insert a node at the beginning of a linked list

```
newNode.next = head;
head = newNode;
```

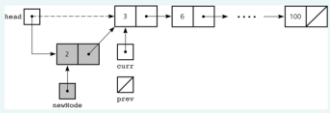


Figure 5-14
Inserting at the beginning of a linked list

- Inserting at the end of a linked list is not a special case if `curr` is `null`

```
newNode.next = curr;
prev.next = newNode;
```

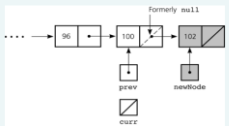


Figure 5-15
Inserting at the end of a linked list

- Three steps to insert a new node into a linked list
  - Determine the point of insertion
  - Create a new node and store the new data in it
  - Connect the new node to the linked list by changing references

- To display all the data items in a linked list

```
for (Node curr = head; curr != null; curr =
    curr.next) {
  System.out.println(curr.item);
} // end for
```

```
public class AList<T> implements ListInterface<T> {
    // Data
    private T[] list;
    private int numberOfEntries;
    private boolean integrityOK;
    private static final int DEFAULT_CAPACITY = 30;
    private static final int MAX_CAPACITY = 10000;
    // Operations
    public AList() {
        this(DEFAULT_CAPACITY);// call another constructor
    }
    public AList(int initialCapacity) {
        integrityOK = false;
        if (initialCapacity < DEFAULT_CAPACITY)
            initialCapacity = DEFAULT_CAPACITY;
        else
            checkCapacity(initialCapacity);
        T[] tempList = (T[])new Object[initialCapacity + 1];// terminate and return
        list = tempList;
        numberOfEntries = 0;
        integrityOK = true;
    }
    public void add(T newEntry) {
        add(numberOfEntries + 1, newEntry);
    }

    public void add(int givenPosition, T newEntry) {
        checkIntegrity();
        if ((givenPosition >= 1) && (givenPosition <= numberOfEntries + 1))
        {
            if (givenPosition <= numberOfEntries)
                makeRoom(givenPosition);// modular programming
            list[givenPosition] = newEntry;
            numberOfEntries++;
            ensureCapacity();//Ensure enough room for next add
        }
        else
            throw new IndexOutOfBoundsException("Given position of add's new entry is out
    }
```

1

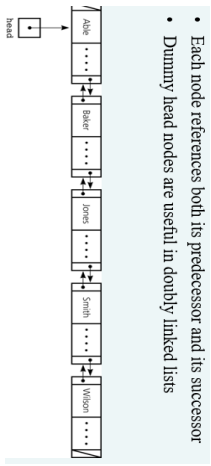
```
public T remove(int givenPosition) {
    checkIntegrity();
    if ((givenPosition >= 1) && (givenPosition <= numberOfEntries))
    {
        T result = list[givenPosition];
        if(givenPosition < numberOfEntries)
            removeGap(givenPosition);
        list[numberOfEntries] = null;
        numberOfEntries--;
        return result;
    }
    else
        throw new IndexOutOfBoundsException("Illegal position given to remove operati
}
public void clear() {
    checkIntegrity();
    for (int index = 1; index <= numberOfEntries; index++)
        list[index] = null;
    numberOfEntries = 0;
}
public T replace(int givenPosition, T newEntry) {
    checkIntegrity();
    if ((givenPosition >= 1) && (givenPosition <= numberOfEntries))
    {
        T originalEntry = list[givenPosition];
        list[givenPosition] = newEntry;
        return originalEntry;
    }
    else
        throw new IndexOutOfBoundsException("Illegal position given to replace operat
}
public T getEntry(int givenPosition) {
    checkIntegrity();
    if ((givenPosition >= 1) && (givenPosition <= numberOfEntries))
    {
        return list[givenPosition];
    }
    else
        throw new IndexOutOfBoundsException("Illegal position given to getEntry opera
```

2

```
public T[] toArray() {
    checkIntegrity();
    @SuppressWarnings("unchecked")
    T[] result = (T[])new Object[numberOfEntries];// downcast
    for (int index = 0; index < numberOfEntries; index++) {
        result[index] = list[index + 1];
    }

    return result;
}
public boolean contains(T anEntry) {
    checkIntegrity();
    boolean found = false;
    int index = 1;
    while (!found && (index <= numberOfEntries))
    {
        if(anEntry.equals(list[index]))
            found = true;
        index++;
    }
    return found;
}
public int getLength() {
    return numberOfEntries;
}
public boolean isEmpty() {
    return numberOfEntries == 0;
}
// Double the capacity of the array list if it is full
private void ensureCapacity() {
    int capacity = list.length - 1;
    if (numberOfEntries >= capacity)
    {
        int newCapacity = 2 * capacity;
        checkCapacity(newCapacity);
        list = Arrays.copyOf(list, newCapacity+1);
    }
}
```

3



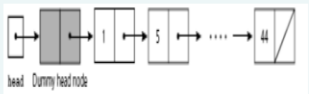
- Each node references both its predecessor and its successor
- Dummy head nodes are useful in doubly linked lists

```
private void makeRoom(int givenPosition) {
    int newIndex = givenPosition;
    int lastIndex = numberOfEntries;

    for (int index = lastIndex; index >= newIndex; index--)
        list[index + 1] = list[index];
}
private void removeGap(int givenPosition) {
    int removedIndex = givenPosition;
    for (int index = removedIndex; index < numberOfEntries; index++)
        list[index+1] = list[index];
}
private void checkIntegrity() {
    if (!integrityOK)
        throw new SecurityException("AList object is corrupt.");
}
private void checkCapacity(int capacity) {
    if(capacity > MAX_CAPACITY)
        throw new IllegalStateException("Attempt to create a list" + "whose capacity e
}
}
```

4

- Dummy head node
  - Always present, even when the linked list is empty
  - Insertion and deletion algorithms initialize `prev` to reference the dummy head node, rather than `null`



- To delete the node that `curr` references

```
curr.preceding.next = curr.next;
curr.next.preceding = curr.preceding;
```

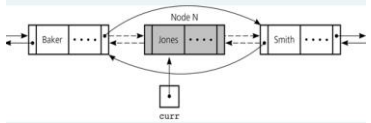


Figure 5-28
Reference changes for deletion

- To insert a new node that `newNode` references before the node referenced by `curr`

```
newNode.next = curr;
newNode.preceding = curr.preceding;
curr.preceding = newNode;
newNode.preceding.next = newNode;
```

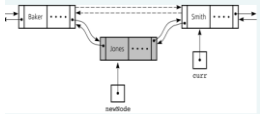


Figure 5-29
Reference changes for insertion

```
*LinkedList.java
1  public class LinkedList<T> implements ListInterface<T>
2  {  private Node firstNode;
3     private int  numberOfEntries;// 0 means the empty list
4     public LinkedList()
5     {
6        initializeDataFields();
7     }
8     public void clear()
9     {
10       initializeDataFields();
11    } // end clear
12    public void add(T newEntry)          // OutOfMemoryError possible
13    {
14       Node newNode = new Node(newEntry);
15
16       if (isEmpty())
17          firstNode = newNode;
18       else                              // Add to end of nonempty list
19       {
20          Node lastNode = getNodeAt(numberOfEntries);
21          lastNode.setNextNode(newNode); // Make last node reference new node
22       } // end if
23
24       numberOfEntries++;
25    } // end add
```

1

```
public void add(int givenPosition, T newEntry) // OutOfMemoryError possible
{
   // precondition
   if ((givenPosition >= 1) && (givenPosition <= numberOfEntries + 1))
   {
      Node newNode = new Node(newEntry);
      if (givenPosition == 1)                 // Case 1
      {
         newNode.setNextNode(firstNode);
         firstNode = newNode;
      }
      else                                    // Case 2: list is not empty
      {                                       // and givenPosition > 1
         Node nodeBefore = getNodeAt(givenPosition - 1);
         Node nodeAfter = nodeBefore.getNextNode();
         newNode.setNextNode(nodeAfter);
         nodeBefore.setNextNode(newNode);
      } // end if
      numberOfEntries++;
   }
   else
      throw new IndexOutOfBoundsException("Illegal position given to add operation.");
} // end add
```

2

```
public T remove(int givenPosition)
{
   T result = null;                           // Return value
   if ((givenPosition >= 1) && (givenPosition <= numberOfEntries))
   {
      // Assertion: !isEmpty()
      if (givenPosition == 1)                 // Case 1: Remove first entry
      {
         result = firstNode.getData();        // Save entry to be removed
         firstNode = firstNode.getNextNode(); // Remove entry
      }
      else                                    // Case 2: Not first entry
      {
         Node nodeBefore = getNodeAt(givenPosition - 1);
         Node nodeToRemove = nodeBefore.getNextNode();
         result = nodeToRemove.getData();     // Save entry to be removed
         Node nodeAfter = nodeToRemove.getNextNode();
         nodeBefore.setNextNode(nodeAfter);   // Remove entry
      } // end if
      numberOfEntries--;                      // Update count
      return result;                          // Return removed entry
   }
   else
      throw new IndexOutOfBoundsException("Illegal position given to remove operation.");
} // end remove
public T replace(int givenPosition, T newEntry)
{
   if ((givenPosition >= 1) && (givenPosition <= numberOfEntries))
   {
      // Assertion: !isEmpty()
      Node desiredNode = getNodeAt(givenPosition);
      T originalEntry = desiredNode.getData();
      desiredNode.setData(newEntry);
      return originalEntry;
   }
   else
      throw new IndexOutOfBoundsException("Illegal position given to replace operation.");
} // end replace
```

3

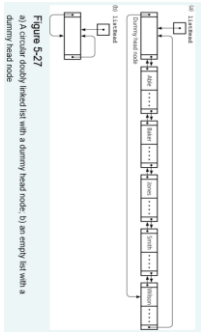


Figure 5-27
a) A circular doubly linked list with a dummy head node; b) an empty list with a dummy head node

```
public T getEntry(int givenPosition)
{  if ((givenPosition >= 1) && (givenPosition <= numberOfEntries))
   {  // Assertion: !isEmpty()
      return getNodeAt(givenPosition).getData();
   }
   else
      throw new IndexOutOfBoundsException("Illegal position given to getEntry operation.");
} // end getEntry
public T[] toArray()
{
   // The cast is safe because the new array contains null entries
   @SuppressWarnings("unchecked")
   T[] result = (T[])new Object[numberOfEntries];
   int index = 0;
   Node currentNode = firstNode;
   while ((index < numberOfEntries) && (currentNode != null))
   {
      result[index] = currentNode.getData();
      currentNode = currentNode.getNextNode();
      index++;
   }

   return result;
}
public boolean contains(T anEntry)
{
   boolean found = false;
   Node currentNode = firstNode;
   while (!found && (currentNode != null))
   {
      if (anEntry.equals(currentNode.getData()))
         found = true;
      else
         currentNode = currentNode.getNextNode();
   } // end while

   return found;
} // end contains
```

4

```
public int getLength()
{
   return numberOfEntries;
} // end getLength

public boolean isEmpty()
{
   boolean result;

   if (numberOfEntries == 0) // Or getLength() == 0
   {
      // Assertion: firstNode == null
      result = true;
   }
   else
   {
      // Assertion: firstNode != null
      result = false;
   } // end if

   return result;
} // end isEmpty

// Initializes the class's data fields to indicate an empty list.
private void initializeDataFields()
{
   firstNode = null;
   numberOfEntries = 0;
} // end initializeDataFields
```

5

```
// Returns a reference to the node at a given position.
// Precondition: The chain is not empty;
//               1 <= givenPosition <= numberOfEntries.
private Node getNodeAt(int givenPosition)
{
   // Assertion: (firstNode != null) &&
   //            (1 <= givenPosition) && (givenPosition <= numberOfEntries)
   Node currentNode = firstNode;

   // Traverse the chain to locate the desired node
   // (skipped if givenPosition is 1)
   for (int counter = 1; counter < givenPosition; counter++)
      currentNode = currentNode.getNextNode();
   // Assertion: currentNode != null
   return currentNode;
} // end getNodeAt

private class Node
{
   private T    data; // Entry in list
   private Node next; // Link to next node

   private Node(T dataPortion)
   {
      data = dataPortion;
      next = null;
   }

   private Node(T dataPortion, Node nextNode)
   {
      data = dataPortion;
      next = nextNode;
   }

   private T getData()
   {
      return data;
   }
```

6

```
   private void setData(T newData)
   {
      data = newData;
   }

   private Node getNextNode()
   {
      return next;
   }

   private void setNextNode(Node nextNode)
   {
      next = nextNode;
   }
}
```

7

- A reference-based implementation of the ADT list
  - Does not shift items during insertions and deletions
  - Does not impose a fixed maximum length on the list

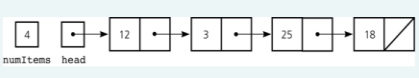


Figure 5-18
A reference-based implementation of the ADT list

- A method with access to a linked list's `head` reference has access to the entire list
- When head is an actual argument to a method, its value is copied into the corresponding formal parameter

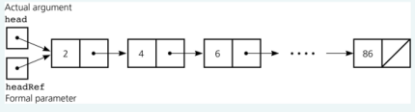


Figure 5-19
A head reference as an argument

- `tail` references
  - Remembers where the end of the linked list is
  - To add a node to the end of a linked list

```
tail.next = new Node(request, null);
```

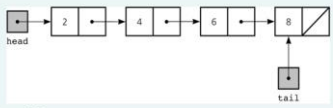
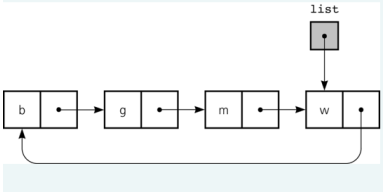


Figure 5-22
A linked list with `head` and `tail` references



igure 5-24
circular linked list with an external reference to the last node