

**Dynamic Data Structures and Generics 12**

**FIGURE 12.1** Some Methods in the Class ArrayList

`public ArrayList<Base_Type>(int initialCapacity)`

Creates an empty list with the specified *Base\_Type* and initial capacity. The *Base\_Type* must be a class type; it cannot be a primitive type such as `int` or `double`. When the list needs to increase its capacity, the capacity doubles.

`public ArrayList<Base_Type>()`

Behaves like the previous constructor, but the initial capacity is ten.

`public boolean add(Base_Type newElement)`

Adds the specified element to the end of this list and increases the list's size by 1. The capacity of the list is increased if that is required. Returns true if the addition is successful.

`public void add(int index, Base_Type newElement)`

Inserts the specified element at the specified index position of this list. Shifts elements at subsequent positions to make room for the new entry by increasing their indices by 1. Increases the list's size by 1. The capacity of the list is increased if that is required. Throws `IndexOutOfBoundsException` if `index < 0` or `index > size()`.

`public Base_Type get(int index)`

Returns the element at the position specified by `index`. Throws `IndexOutOfBoundsException` if `index < 0` or `index ≥ size()`.

<pre>public Base_Type set(int index, Base_Type element)</pre> <p>Replaces the element at the position specified by <code>index</code> with the given element. Returns the element that was replaced. Throws <code>IndexOutOfBoundsException</code> if <code>index &lt; 0</code> or <code>index ≥ size()</code>.</p>
<pre>public Base_Type remove(int index)</pre> <p>Removes and returns the element at the specified index. Shifts elements at subsequent positions toward position <code>index</code> by decreasing their indices by 1. Decreases the list's size by 1. Throws <code>IndexOutOfBoundsException</code> if <code>index &lt; 0</code> or <code>index ≥ size()</code>.</p>
<pre>public boolean remove(Object element)</pre> <p>Removes the first occurrence of <code>element</code> in this list, and shifts elements at subsequent positions toward the removed element by decreasing their indices by 1. Decreases the list's size by 1. Returns true if <code>element</code> was removed; otherwise returns false and does not alter the list.</p>
<pre>public void clear()</pre> <p>Removes all elements from this list.</p>
<pre>public int size()</pre> <p>Returns the number of elements in this list.</p>
<pre>public boolean contains(Object element)</pre> <p>Returns true if <code>element</code> is in this list; otherwise, returns false.</p>
<pre>public int indexOf(Object element)</pre> <p>Returns the index of the first occurrence of <code>element</code> in this list. Returns -1 if <code>element</code> is not on the list.</p>
<pre>public boolean isEmpty()</pre> <p>Returns true if this list is empty; otherwise, returns false.</p>

## LISTING 12.1 Using ArrayList to Maintain a List (part 1 of 2)

---

```
import java.util.ArrayList;
import java.util.Scanner;

public class ArrayListDemo
{
    public static void main(String[] args)
    {
        ArrayList<String> toDoList = new ArrayList<String>();
        System.out.println("Enter items for the list, when "+
                           "prompted.");
        boolean done = false;
        Scanner keyboard = new Scanner(System.in);
        while (!done)
        {
            System.out.println("Type an entry:");
            String entry = keyboard.nextLine( );
            toDoList.add(entry);
            System.out.print("More items for the list? ");
            String ans = keyboard.nextLine( );
            if (!ans.equalsIgnoreCase("yes"))
                done = true;
        }
        System.out.println("The list contains:");
        int listSize = toDoList.size( );
        for (int position = 0; position < listSize;
            position++)
            System.out.println(toDoList.get(position));
    }
}
```

### *Sample Screen Output*

```
Enter items for the list, when prompted.  
Type an entry:  
Buy milk  
More items for the list? yes  
Type an entry:  
Wash car  
More items for the list? yes  
Type an entry:  
Do assignment  
More items for the list? no  
The list contains:  
Buy milk  
Wash car  
Do assignment
```

**FIGURE 12.2 Selected Methods in the Collection Interface**

<pre>public boolean add(Base_Type newElement)</pre> <p>Adds the specified element to the collection. Returns <code>true</code> if the collection is changed as a result of the call.</p>
<pre>public void clear()</pre> <p>Removes all of the elements from the collection.</p>
<pre>public boolean remove(Object o)</pre> <p>Removes a single instance of the specified element from the collection if it is present. Returns <code>true</code> if the collection is changed as a result of the call.</p>
<pre>public boolean contains(Object o)</pre> <p>Returns <code>true</code> if the specified element is a member of the collection.</p>
<pre>public boolean isEmpty()</pre> <p>Returns <code>true</code> if the collection is empty.</p>
<pre>public int size()</pre> <p>Returns the number of elements in the collection.</p>
<pre>public Object[] toArray()</pre> <p>Returns an array containing all of the elements in the collection. The array is of a type <code>Object</code> so each element may need to be typecast back into the original base type.</p>

## LISTING 12.2 A HashSet Demonstration (part 1 of 2)

---

```
import java.util.HashSet;
public class HashSetDemo
{
    public static void main(String[] args)
    {
        HashSet<Integer> intSet = new HashSet<Integer>();
        intSet.add(2);
        intSet.add(7);
        intSet.add(7); ← Ignored since 7 is already in the set
        intSet.add(3);
        printSet(intSet);
        intSet.remove(3);
        printSet(intSet);
        System.out.println("Set contains 2: " +
            intSet.contains(2));
        System.out.println("Set contains 3: " +
            intSet.contains(3));
    }
    public static void printSet(HashSet<Integer> intSet)
    {
        System.out.println("The set contains:");
        for (Object obj : intSet.toArray())
        {
            Integer num = (Integer) obj;
            System.out.println(num.intValue());
        }
    }
}
```

### *Sample Screen Output*

```
The set contains:  
2  
3  
7  
The set contains:  
2  
7  
Set contains 2: true  
Set contains 3: false
```



**FIGURE 12.3 Selected Methods in the Map Interface**

---

<pre>public Base_Type_Value put(Base_Type_Key k, Base_Type_Value v)</pre> <p>Associates the value <i>v</i> with the key <i>k</i>. Returns the previous value for <i>k</i> or <code>null</code> if there was no previous mapping</p>
<pre>public Base_Type_Value get(Object k)</pre> <p>Returns the value mapped to the key <i>k</i> or <code>null</code> if no mapping exists.</p>
<pre>public void clear()</pre> <p>Removes a single instance of the specified element from the collection if it is present. Returns <code>true</code> if the collection is changed as a result of the call.</p>
<pre>public Base_Type_Value remove(Object k)</pre> <p>Removes the mapping of key <i>k</i> from the map if present. Returns the previous value for the key <i>k</i> or <code>null</code> if there was no previous mapping.</p>
<pre>public boolean containsKey(Object k)</pre> <p>Returns <code>true</code> if the key <i>k</i> is a key in the map.</p>
<pre>public boolean containsValue(Object v)</pre> <p>Returns <code>true</code> if the value <i>v</i> is a value in the map.</p>
<pre>public boolean isEmpty()</pre> <p>Returns <code>true</code> if the map contains no mappings.</p>
<pre>public int size()</pre> <p>Returns the number of mappings in the map.</p>
<pre>public Set &lt;Base_Type_Key&gt; keySet()</pre> <p>Returns a set containing all of the keys in the map.</p>
<pre>public Collection &lt;Base_Type_Value&gt; values()</pre> <p>Returns a collection containing all of the values in the map.</p>

### LISTING 12.3 A HashMap Demonstration (part 1 of 2)

---

```
import java.util.HashMap;
public class HashMapDemo
{
    public static void main(String[] args)
    {
        HashMap<String, Integer> mountains =
            new HashMap<String, Integer>();
        mountains.put("Everest", 29029);
        mountains.put("K2", 28251);
        mountains.put("Kangchenjunga", 28169);
        mountains.put("Denali", 20335);
        printMap(mountains);
        System.out.println("Denali in the map: " +
            mountains.containsKey("Denali"));
        System.out.println();
    }
}
```

```
System.out.println("Changing height of Denali.");  
mountains.put("Denali", 20320);  
printMap(mountains);
```

*Overwrites the old  
value for Denali*

```
System.out.println("Removing Kangchenjunga.");  
mountains.remove("Kangchenjunga");  
printMap(mountains);
```

```
}
```

```
public static void printMap(HashMap<String, Integer> map)
```

```
{
```

```
System.out.println("Map contains:");
```

```
for (String keyMountainName : map.keySet())
```

```
{
```

```
Integer height = map.get(keyMountainName);
```

```
System.out.println(keyMountainName + " --> " +  
height.intValue() + " feet.");
```

```
}
```

```
System.out.println();
```

```
}
```

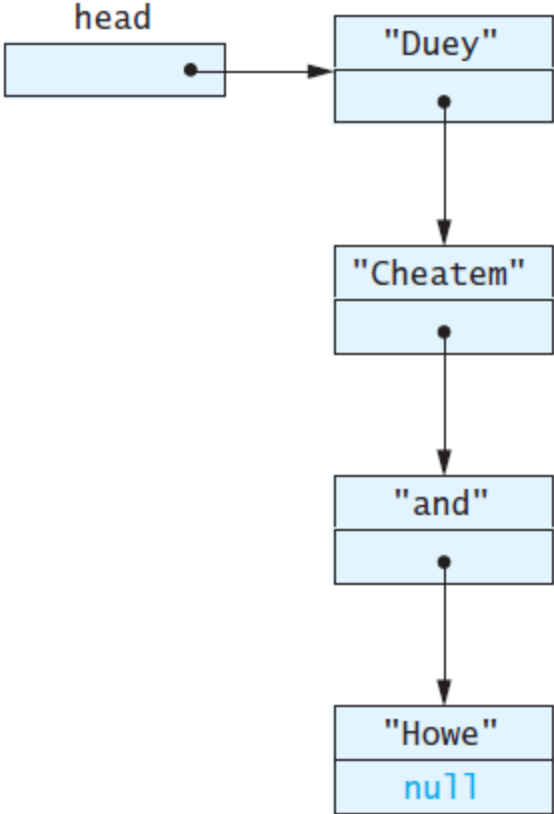
```
}
```

## Sample Screen Output

```
Map contains:
K2 --> 28251 feet.
Denali --> 20355 feet.
Kangchenjunga --> 28169 feet.
Everest --> 29029 feet.
Denali in the map: true
Changing height of Denali.
Map contains:
K2 --> 28251 feet.
Denali --> 20320 feet.
Kangchenjunga --> 28169 feet.
Everest --> 29029 feet.
Removing Kangchenjunga.
Map contains:
K2 --> 28251 feet.
Denali --> 20320 feet.
Everest --> 29029 feet.
```

**FIGURE 12.4 A Linked List**

---



## LISTING 12.4 A Node Class

---

```
public class ListNode
{
    private String data;
    private ListNode link;

    public ListNode()
    {
        link = null;
        data = null;
    }

    public ListNode(String newData, ListNode linkValue)
    {
        data = newData;
        link = linkValue;
    }

    public void setData(String newData)
    {
        data = newData;
    }

    public String getData()
    {
        return data;
    }

    public void setLink(ListNode newLink)
    {
        link = newLink;
    }

    public ListNode getLink()
    {
        return link;
    }
}
```

*Later in this chapter, we will hide this class by making it private.*

## LISTING 12.5 A Linked-List Class (part 1 of 2)

---

```
public class StringLinkedList
{
    private ListNode head;
    public StringLinkedList()
    {
        head = null;
    }
    /**
    Displays the data on the list.
    */
    public void showList()
    {
        ListNode position = head;
        while (position != null)
        {
            System.out.println(position.getData());
            position = position.getLink();
        }
    }
}
```

We will give another definition of this class later in this chapter.

```

/**
Returns the number of nodes on the list.
*/
public int length()
{
    int count = 0;
    ListNode position = head;
    while (position != null)
    {
        count++;
        position = position.getLink();
    }
    return count;
}
/**
Adds a node containing the data addData at the
start of the list.
*/
public void addANodeToStart(String addData)
{
    head = new ListNode(addData, head);
}

```



```

/**
Deletes the first node on the list.
*/
public void deleteHeadNode()
{
    if (head != null)
        head = head.getLink();
    else
    {
        System.out.println("Deleting from an empty list.");
        System.exit(0);
    }
}
/**
Sees whether target is on the list.
*/
public boolean onList(String target)
{
    return find(target) != null;
}

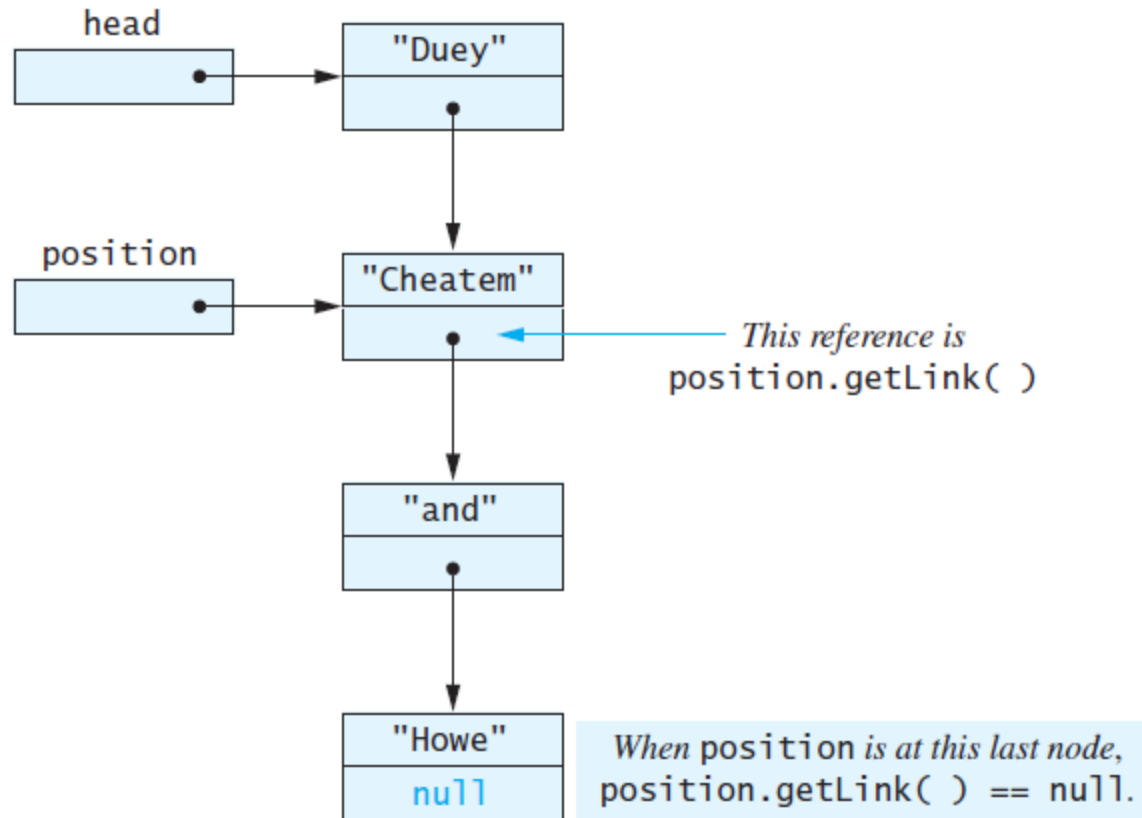
```

```

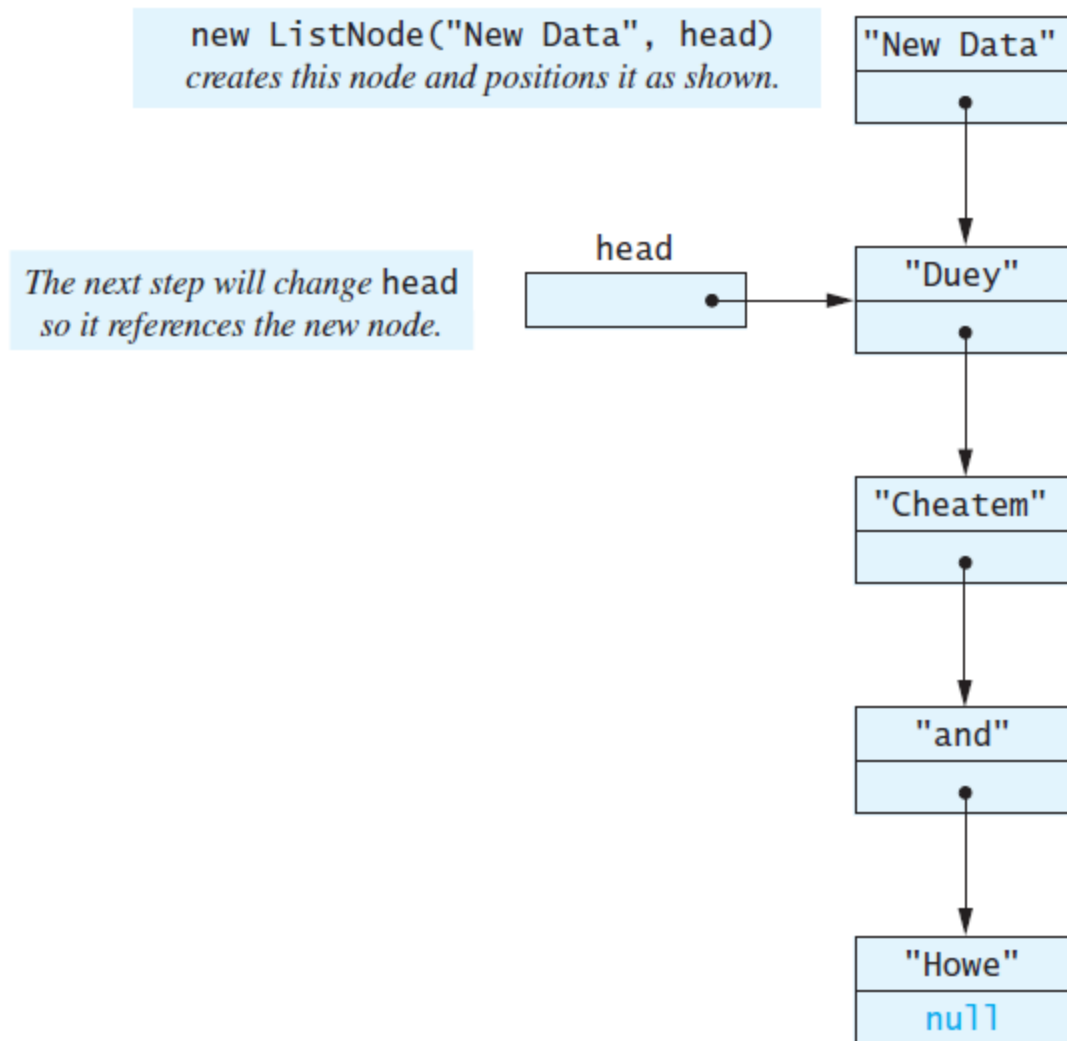
//Returns a reference to the first node containing the
//target data. If target is not on the list, returns null.
private ListNode find(String target)
{
    boolean found = false;
    ListNode position = head;
    while ((position != null) && !found)
    {
        String dataAtPosition = position.getData();
        if (dataAtPosition.equals(target))
            found = true;
        else
            position = position.getLink();
    }
    return position
}
}

```

**FIGURE 12.5** Moving Down a Linked List



**FIGURE 12.6 Adding a Node at the Start of a Linked List**



## LISTING 12.6 A Linked-List Demonstration

---

```
public class StringLinkedListDemo
{
    public static void main(String[] args)
    {
        StringLinkedList list = new StringLinkedList();
        list.addANodeToStart("One");
        list.addANodeToStart("Two");
        list.addANodeToStart("Three");
        System.out.println("List has " + list.length() +
                           " entries.");
        list.showList();

        if (list.onList("Three"));
            System.out.println("Three is on list.");
        else
            System.out.println("Three is NOT on list.");

        list.deleteHeadNode();

        if (list.onList("Three"));
            System.out.println("Three is on list.");
        else
            System.out.println("Three is NOT on list.");

        list.deleteHeadNode();
        list.deleteHeadNode();
        System.out.println("Start of list:");
        list.showList();
        System.out.println("End of list");
    }
}
```

### *Screen Output*


```
List has 3 entries.  
Three  
Two  
One  
Three is on list.  
Three is NOT on list.  
Start of list:  
End of list.
```

## LISTING 12.7 A Linked List with a Node Class as an Inner Class (part 1 of 2)

---

```
public class StringLinkedListSelfContained
{
    private ListNode head;
    public StringLinkedListSelfContained()
    {
        head = null;
    }
    /**
    Displays the data on the list.
    */
    public void showList()
    {
        ListNode position = head;
        while (position != null)
        {
            System.out.println(position.data);
            position = position.link;
        }
    }
}
```

Note that the outer class has direct access to the inner-class instance variables `data` and `link`.



```

/**
Returns the number of nodes on the list.
*/
public int length()
{
    int count = 0;
    ListNode position = head;
    while (position != null)
    {
        count++;
        position = position.link;
    }
    return count
}
/**
Adds a node containing the data addData at the
start of the list.
*/
public void addANodeToStart(String addData)
{
    head = new ListNode(addData, head);
}

```



```
/**
Deletes the first node on the list.
*/
public void deleteHeadNode()
{
    if (head != null)
        head = head.link;
    else
    {
        System.out.println("Deleting from an empty list.");
        System.exit(0);
    }
}
/**
Sees whether target is on the list.
*/
public boolean onList(String target)
{
    return (find(target) != null);
}
```

```

//Returns a reference to the first node containing the
//target data. If target is not on the list, returns null.
private ListNode find(String target)
{
    boolean found = false;
    ListNode position = head;
    while ((position != null) && !found)
    {
        String dataAtPosition = position.data;
        if (dataAtPosition.equals(target))
            found = true;
        else
            position = position.link;
    }
    return position;
}

private class ListNode
{
    private String data;
    private ListNode link;

    public ListNode()
    {
        link = null;
        data = null;
    }

    public ListNode(String newData, ListNode linkValue)
    {
        data = newData;
        link = linkValue;
    }
}
}

```

An inner class

End of outer-class definition

## LISTING 12.8 Placing the Linked-List Data into an Array

---

*This method can be added to the linked-list class in Listing 12.7.*

```
/**
 *Returns an array of the elements on the list.
 */
public String[] toArray()
{
    String[] anArray = new String[length()];
    ListNode position = head;
    int i = 0;
    while (position != null)
    {
        anArray[i] = position.data;
        i++;
        position = position.link;
    }
    return anArray;
}
```

---

## LISTING 12.9 A Linked List with an Iterator (part 1 of 4)

---

```
/**
 *Linked list with an iterator. One node is the "current node."
 *Initially, the current node is the first node. It can be changed
 *to the next node until the iteration has moved beyond the end
 *of the list.
 */
public class StringLinkedListWithIterator
{
    private ListNode head;
    private ListNode current;
    private ListNode previous;

    public StringLinkedListWithIterator()
    {
        head = null;
        current = null;
        previous = null;
    }
    public void addANodeToStart(String addData)
    {
        head = new ListNode(addData, head);
        if ((current == head.link) && (current != null))
            //if current is at old start node
            previous = head;
    }
}
```

```

/**
Sets iterator to beginning of list.
*/
public void resetIteration()
{
    current = head;
    previous = null;
}
/**
Returns true if iteration is not finished.
*/
public boolean moreToIterate()
{
    return current != null;
}
/**
Advances iterator to next node.
*/
public void goToNext()
{
    if (current != null)
    {
        previous = current;
        current = current.link;
    }
}

```

```

else if (head != null)
{
    System.out.println(
        "Iterated too many times or uninitialized
        iteration.");
    System.exit(0);
}
else
{
    System.out.println("Iterating with an empty list.");
    System.exit(0);
}
}
/**
Returns the data at the current node.
*/
public String getDataAtCurrent()
{
    String result = null;
    if (current != null)
        result = current.data;
}

```

```

    else
    {
        System.out.println(
            "Getting data when current is not at any node.");
        System.exit(0);
    }
    return result;
}
/**
Replaces the data at the current node.
*/
public void setDataAtCurrent(String newData)
{
    if (current != null)
    {
        current.data = newData;
    }
    else
    {
        System.out.println(
            "Setting data when current is not at any node.");
        System.exit(0);
    }
}

```

```

/**
Inserts a new node containing newData after the current node.
The current node is the same after invocation as it is before.
Precondition: List is not empty; current node is not
beyond the entire list.
*/
public void insertNodeAfterCurrent(String newData)
{
    ListNode newNode = new LisNode();
    newNode.data = newData;
    if (current != null)
    {
        newNode.link = current.link;
        current.link = newNode;
    }

    else if (head != null)
    {
        System.out.println(
            "Inserting when iterator is past all " +
            "nodes or is not initialized.");
        System.exit(0);
    }
}

```



```

else
{
    System.out.println(
        "Using insertNodeAfterCurrent with empty list.");
    System.exit(0);
}
}
/**
Deletes the current node. After the invocation,
the current node is either the node after the
deleted node or null if there is no next node.
*/
public void deleteCurrentNode()
{
    if ((current != null) && (previous == null))
    {
        previous.link = current.link;
        current = current.link;
    }
}

```

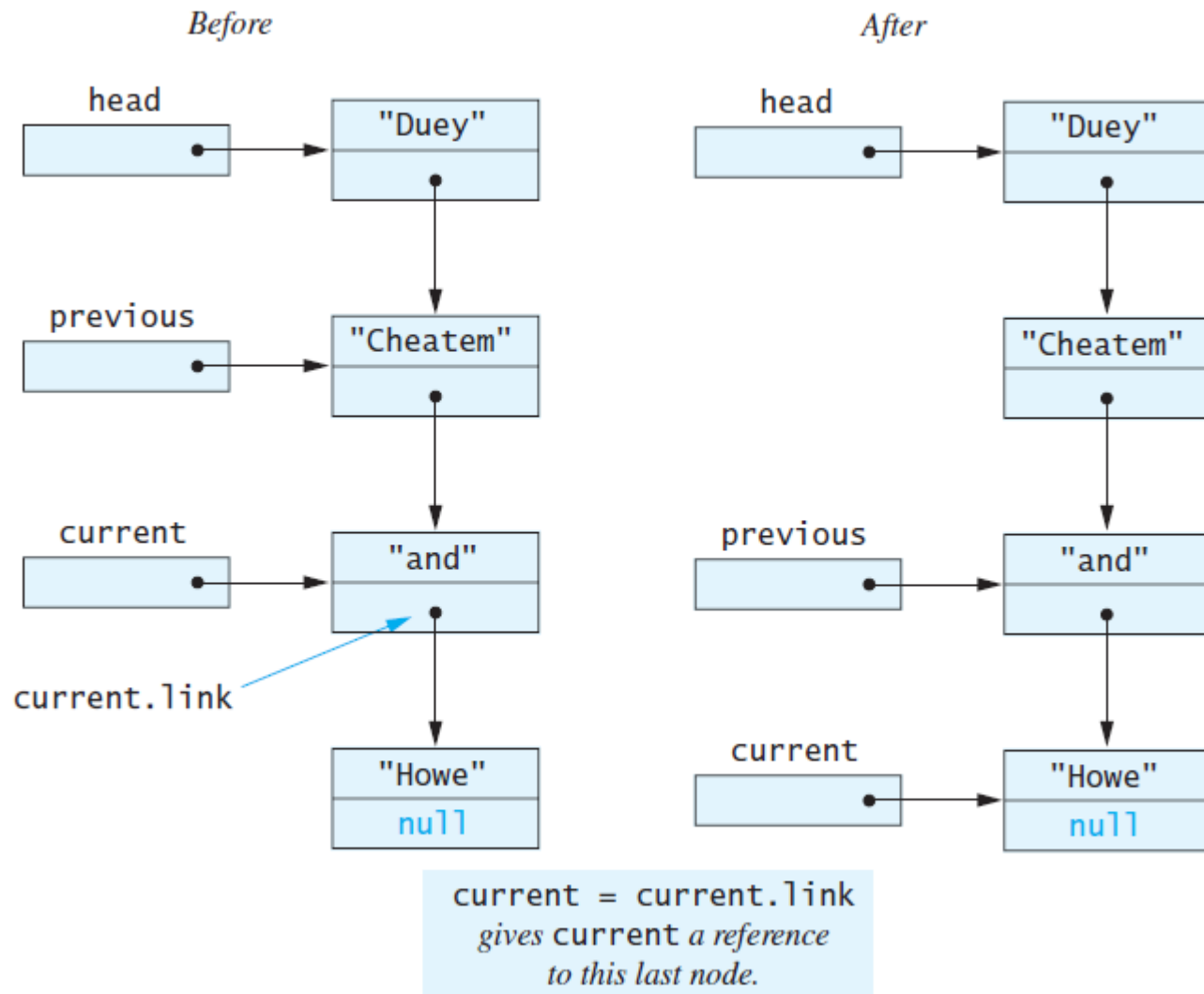
```

else if ((current != null) && (previous == null))
{ //At head node
    head = current.link;
    current = head;
}
else //current==null
{
    System.out.println(
        "Deleting with uninitialized current or an empty " +
        "list.");
    System.exit(0);
}
}
}
<The methods length, onList, find, and showList, as well as the
private inner class ListNode are the same as in Listing 12.7.>
<The method toArray is the same as in Listing 12.8.>
}

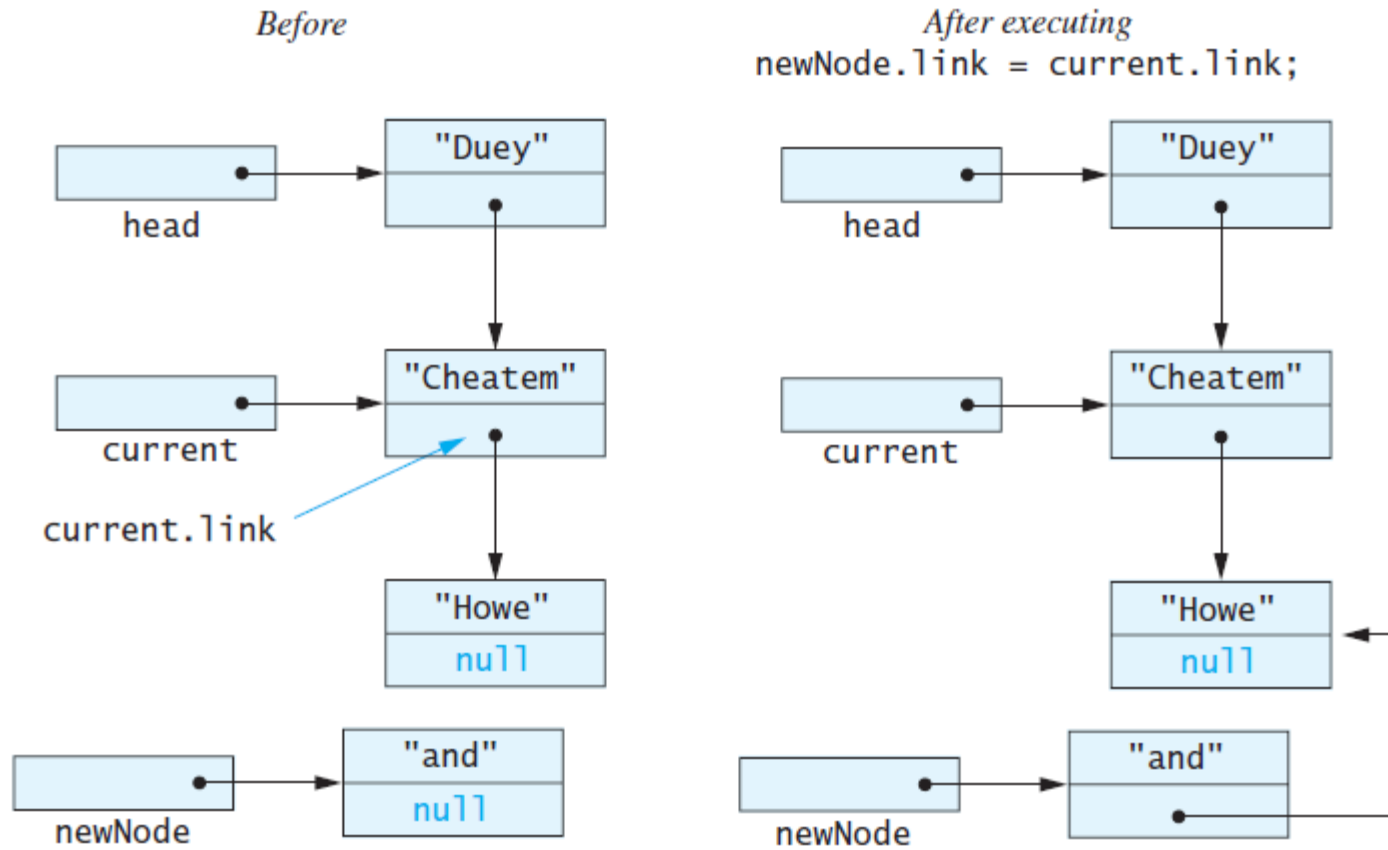
```

*deleteHeadNode is no longer needed, since you have deleteCurrentNode, but if you want to retain deleteHeadNode, it must be redefined to account for current and previous.*

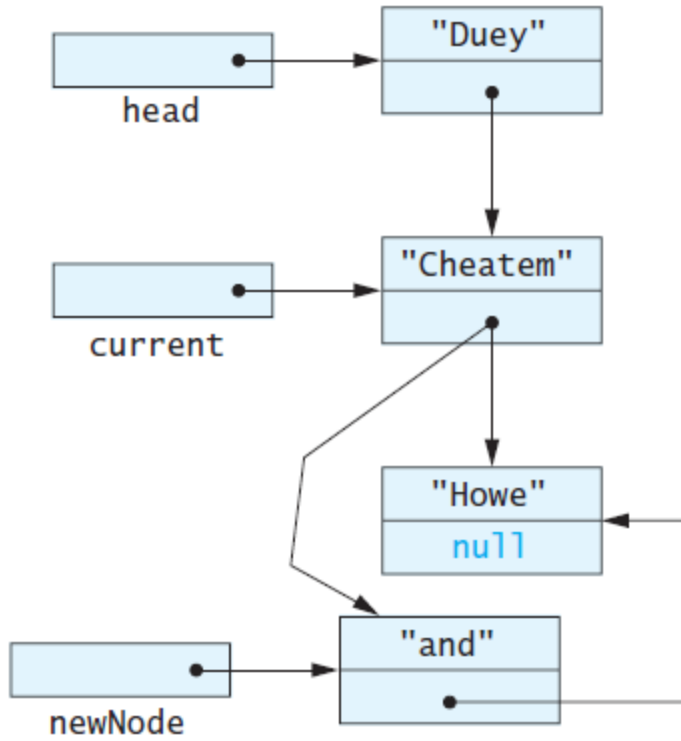
**FIGURE 12.7** The Effect of `goToNext` on a Linked List



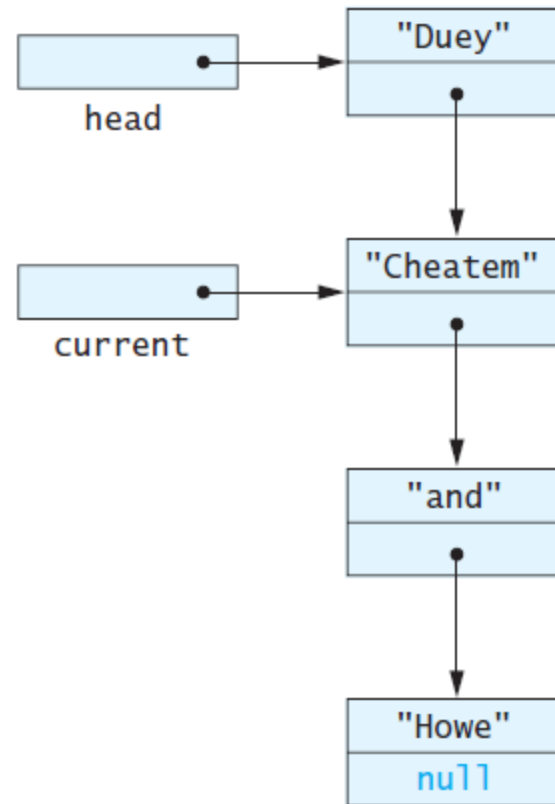
**FIGURE 12.8** Adding a Node to a Linked List, Using `insertAfterIterator`



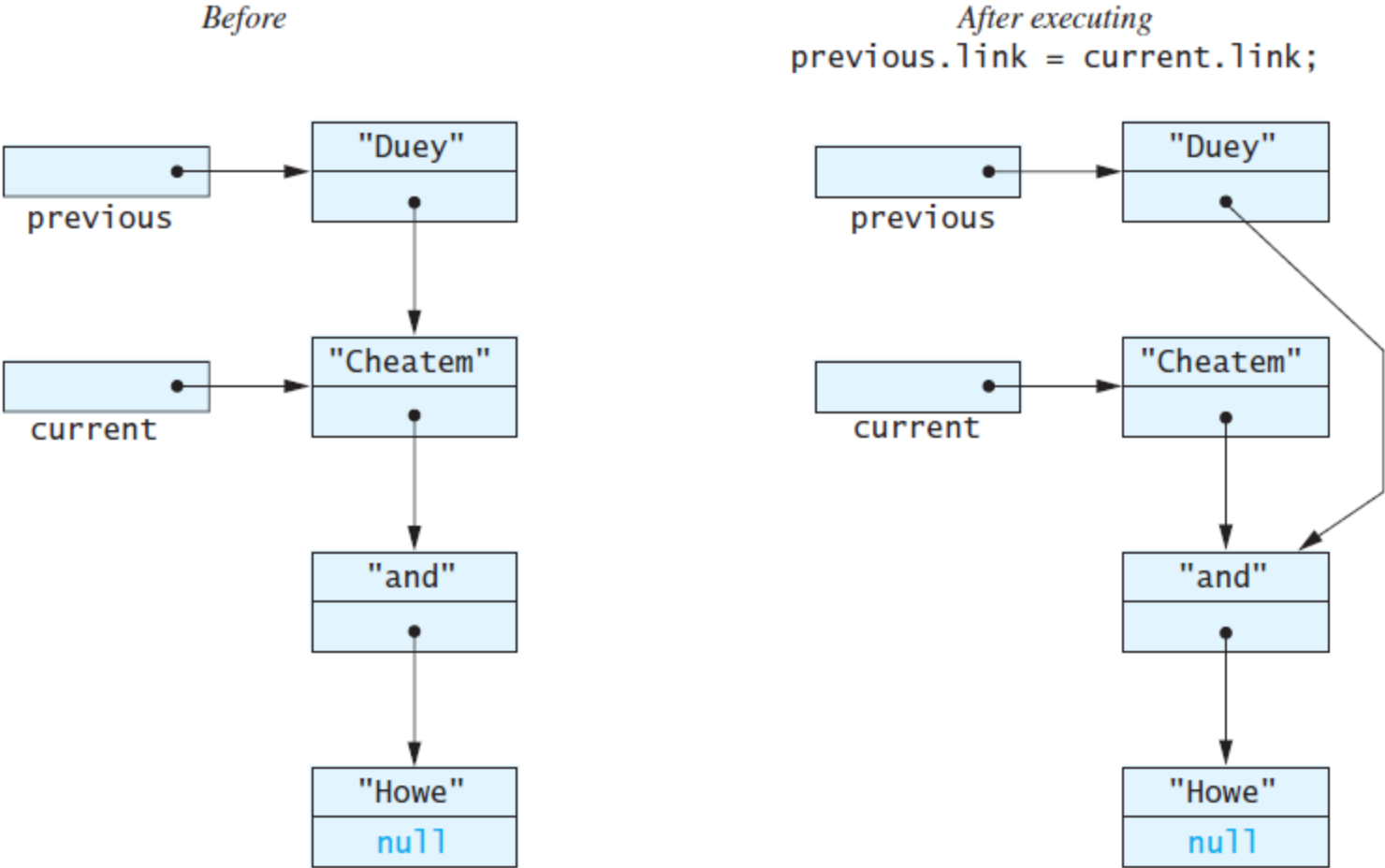
After executing  
`current.link = newNode;`



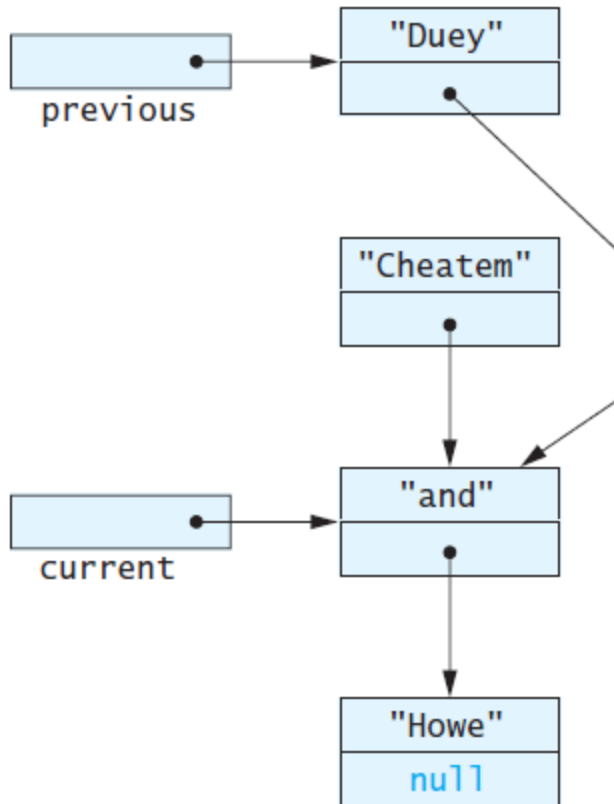
Same picture, cleaned up  
and without newNode



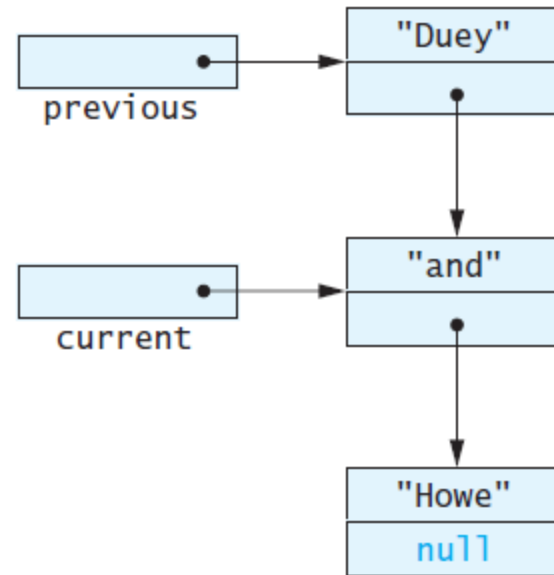
**FIGURE 12.9** Deleting a Node



After executing  
`current = current.link;`



Same picture, cleaned up  
and without the deleted node



### **LISTING 12.10** The LinkedListException Class

---

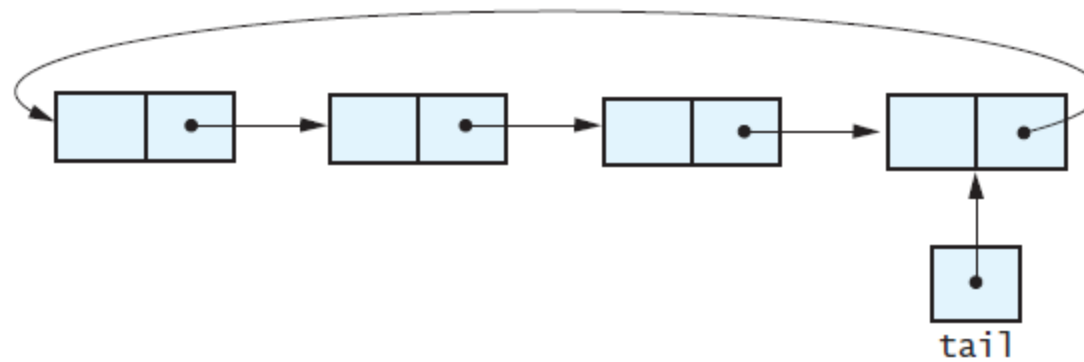
```
public class LinkedListException extends Exception
{
    public LinkedListException()
    {
        super("Linked List Exception");
    }
    public LinkedListException(String message)
    {
        super(message);
    }
}
```

---

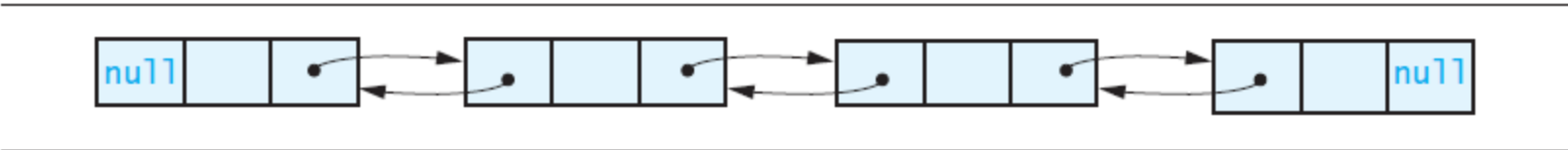


**FIGURE 12.10** A Circular Linked List

---

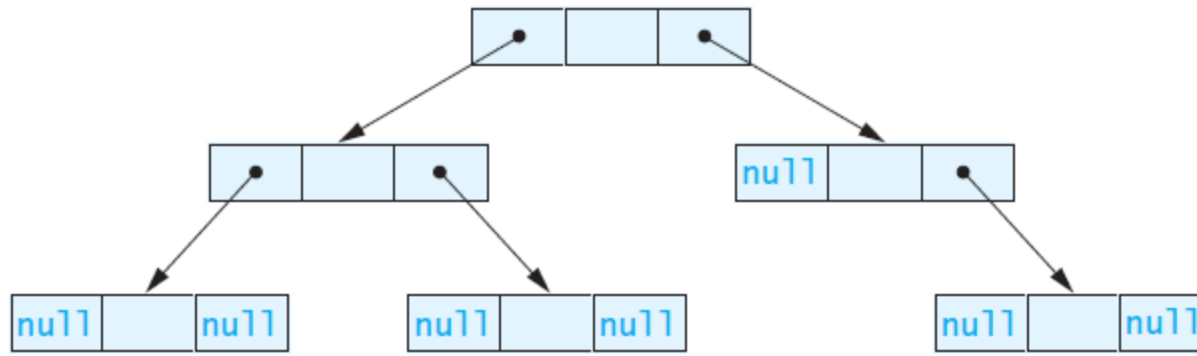


**FIGURE 12.11** A Doubly Linked List



**FIGURE 12.12 A Binary Tree**

---



## LISTING 12.11 A Class Definition That Uses a Type Parameter

---

```
public class Sample<T>
{
    private T data;

    public void setData(T newValue)
    {
        data = newValue;
    }

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

---

## LISTING 12.12 A Generic Linked-List Class (part 1 of 2)

---

```
import java.util.ArrayList;
public class LinkedList2<E>
{
    private ListNode head;
    public LinkedList2( )
    {
        head = null;
    }
    public void showList( )
    {
        ListNode position = head;
        while (position != null)
        {
            System.out.println(position.getData( ));
            position = position.getLink( );
        }
    }
    public int length( )
    {
        int count = 0;
        ListNode position = head;
        while (position != null)
        {
            count++;
            position = position.getLink( );
        }
        return count;
    }
}
```

*Constructor headings do not include the type parameter.*

```

public void addANodeToStart(E addData)
{
    head = new ListNode(addData, head);
}
public void deleteHeadNode( )
{
    if (head != null)
    {
        head = head.getLink( );
    }
    else
    {
        System.out.println("Deleting from an empty list.");
        System.exit(0);
    }
}
public boolean onList(E target)
{
    return find(target) != null;
}

```

```

private ListNode find(E target)
{
    boolean found = false;
    ListNode position = head;
    while (position != null)
    {
        E dataAtPosition = position.getData();
        if (dataAtPosition.equals(target))
            found = true;
        else
            position = position.getLink();
    }
    return position;
}
private ArrayList<E> toArrayList()
{
    ArrayList<E> list = new ArrayList<E>(length());
    ListNode position = head;
    while (position != null)
    {
        list.add(position.data);
        position = position.link;
    }
    return list;
}

```

```
private class ListNode
{
    private E data;
    private ListNode link;
    public ListNode()
    {
        link = null;
        data = null;
    }
    public ListNode(E newData, ListNode linkValue)
    {
        data = newData;
        link = linkValue;
    }
    public E getData()
    {
        return data;
    }
    public ListNode getLink()
    {
        return link;
    }
}
}
```

The inner class heading has no type parameter.

However, the type parameter is used within the definition of the inner class.



## LISTING 12.13 Using the Generic Linked list

---

```
public class LinkedList2Demo
{
    public static void main(String[] args)
    {
        LinkedList2<String> stringList = new LinkedList2<String>( );
        stringList.addANodeToStart("Hello");
        stringList.addANodeToStart("Good-bye");
        stringList.showList( );
        LinkedList2<Integer> numberList = new LinkedList2<Integer>( );
        for (int i = 0; i < 5; i++)
            numberList.addANodeToStart(i);
        numberList.deleteHeadNode();
        numberList.showList( );
    }
}
```

---

### Screen Output

```
Good-bye
Hello
3
2
1
0
```