12.3 Linked Lists

- 1) Node Class 가 LinkedList Class 밖에 있는 경우
- 2) Node Class 가 LinkedList Class 안에 있는 경우
- 3) Iterator 기능을 가진 LinkedList Class

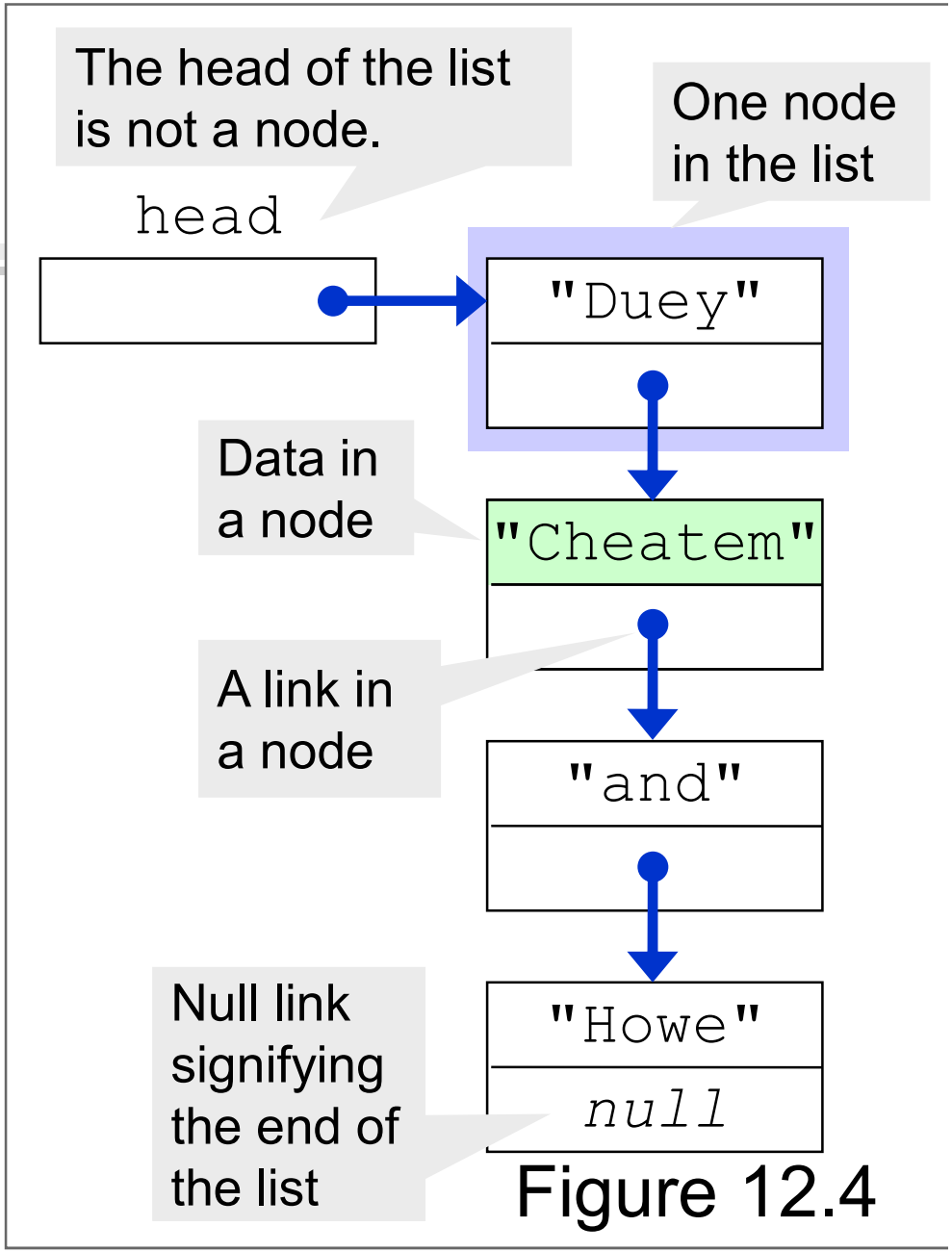


Figure 12.4

1) NODE CLASS 가 LINKEDLIST CLASS 밖에 있는 경우

12.3 Linked Lists

- *Linked lists* consists of objects known as nodes
- Each node has a place for data and a link to another node
- Links are shown as arrows

- Each node is an object of a class that has two instance variables: one for the data and one for the link

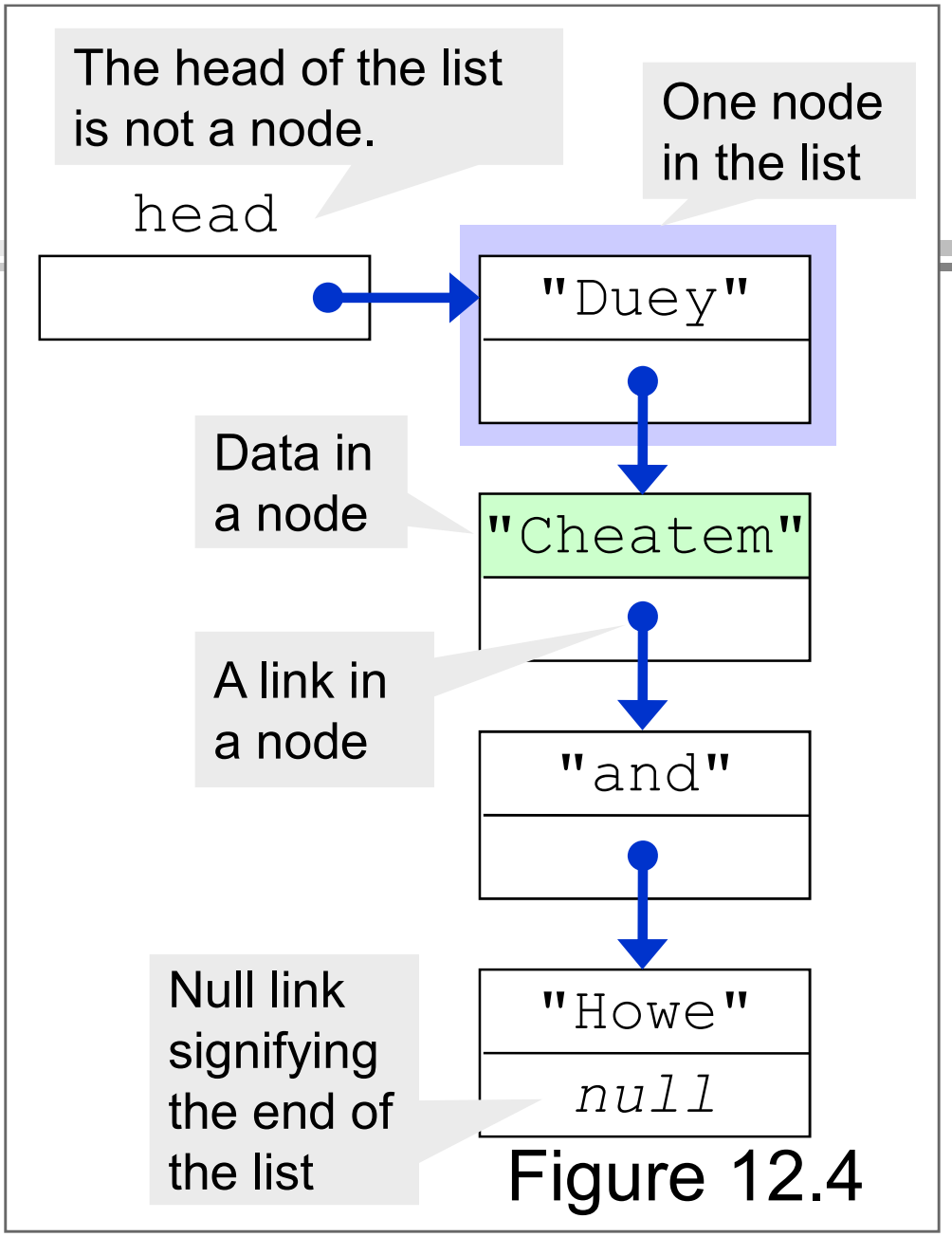
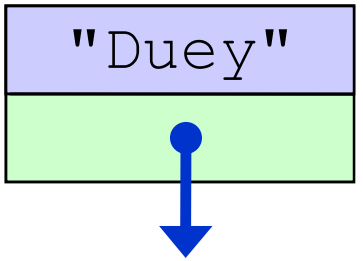


Figure 12.4

ListNode Class: Instance Variables and Constructor

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

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



Two parameters for the constructor:

- data value for the new node
- Link value for the new node

Listing 12.4 A Node Class - ListNode.java

```
// Listing 12.4  A Node Class
// we will give a better definition of this class later in this chapter

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;
    }

// this method works, but has a problem.
// we will discuss the problem later and provide a better alternative
    public ListNode getLink( )
    {
       return link;
    }

}
```

Listing 12.5 A linked List Class - StringLinkedList.java

```
//Listing 12.5 A linked List Class

public class StringLinkedList
{
   private ListNode head;

   public StringLinkedList( )
   {
      head = null;
   }

   /**
    Returns the number of nodes in the list.
   */
   public int length( )
   {
      int count = 0;
      ListNode position = head;
      while (position != null)
      {
         count++;
         position = position.getLink( );
      }
      return count;
   }
```

```
/**
 Adds a node at the start of the list. The added node has addData
 as its data. The added node will be the first node in the list.
*/
public void addANodeToStart(String addData)
{
    head = new ListNode(addData, [      ]);
}

public void deleteHeadNode( )
{
    if (head != null)
    {
        head = head.getLink( );
    }
    else
    {
        System.out.println("Deleting from an empty list.");
        System.exit(0);
    }
}

public boolean onList(String target)
{
    return (Find(target) != null);
}
```

```
/**
   Finds the first node containing the target data, and
   returns a reference to that node.
   If target is not in the list, null is returned
  */
  private ListNode Find(String target)
  {
     ListNode position;
     position = head;
     String dataAtPosition;
     while (position != null)
     {
        dataAtPosition = position.getData( );
        if (dataAtPosition.equals(target))
           return position;
        position = position.getLink( );
     }
     //target was not found,
     return null;
  }
```

```
  public void showList( )
    {
        ListNode position;
        position = head;
        while (position != null)
        {
            System.out.println(position.getData( ));
            position = position.getLink( );
        }
    }

}
```

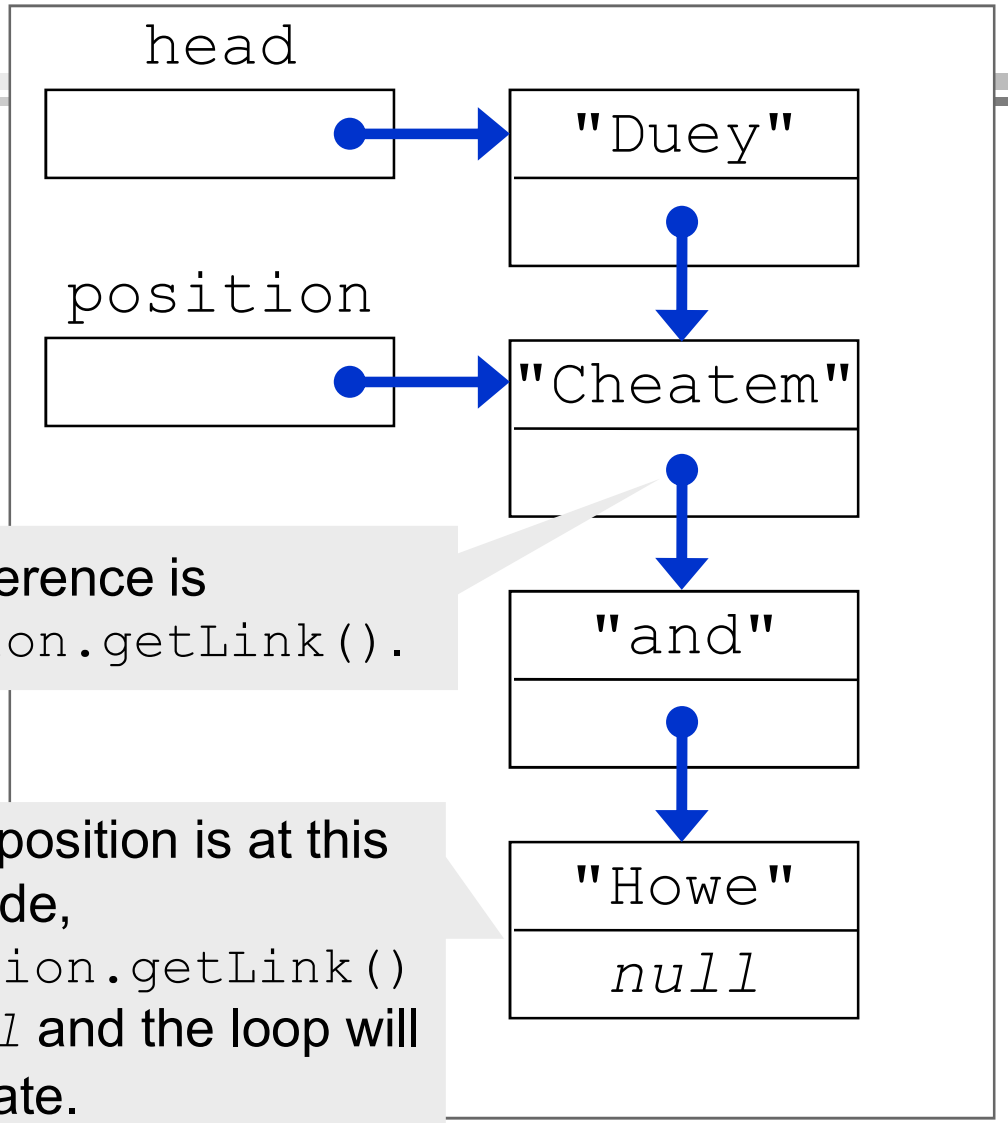
Stepping through a List

Excerpt from `showList` in `StringLinkedList`

Start at beginning of list

```
ListNode position;
position = head;
while (position != null)
{
    ...
    position =
      position.getLink();
}
```

Moves to next node in the list.

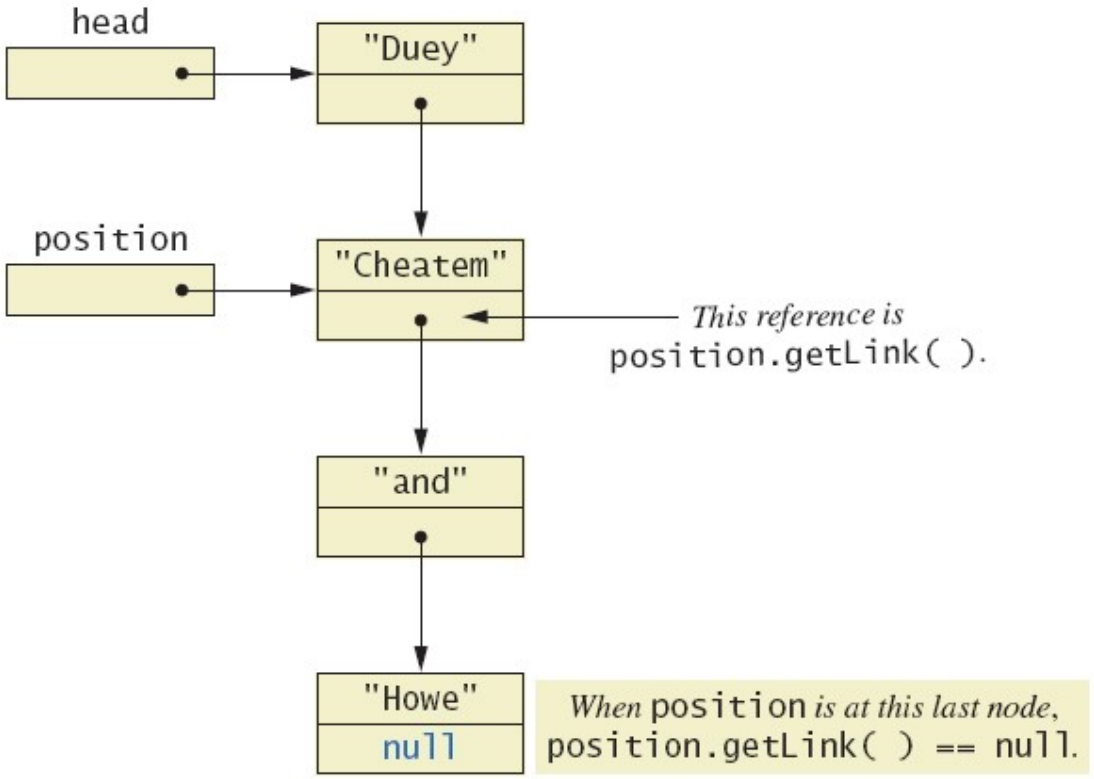


This reference is `position.getLink()`.

When position is at this last node, `position.getLink()` is *null* and the loop will terminate.

Implementing Operations of Linked Lists

- Figure 12.5 Moving down a linked list



Empty List

- Empty List
  - » head instance variable contains null if the linked list is null

Adding a Node

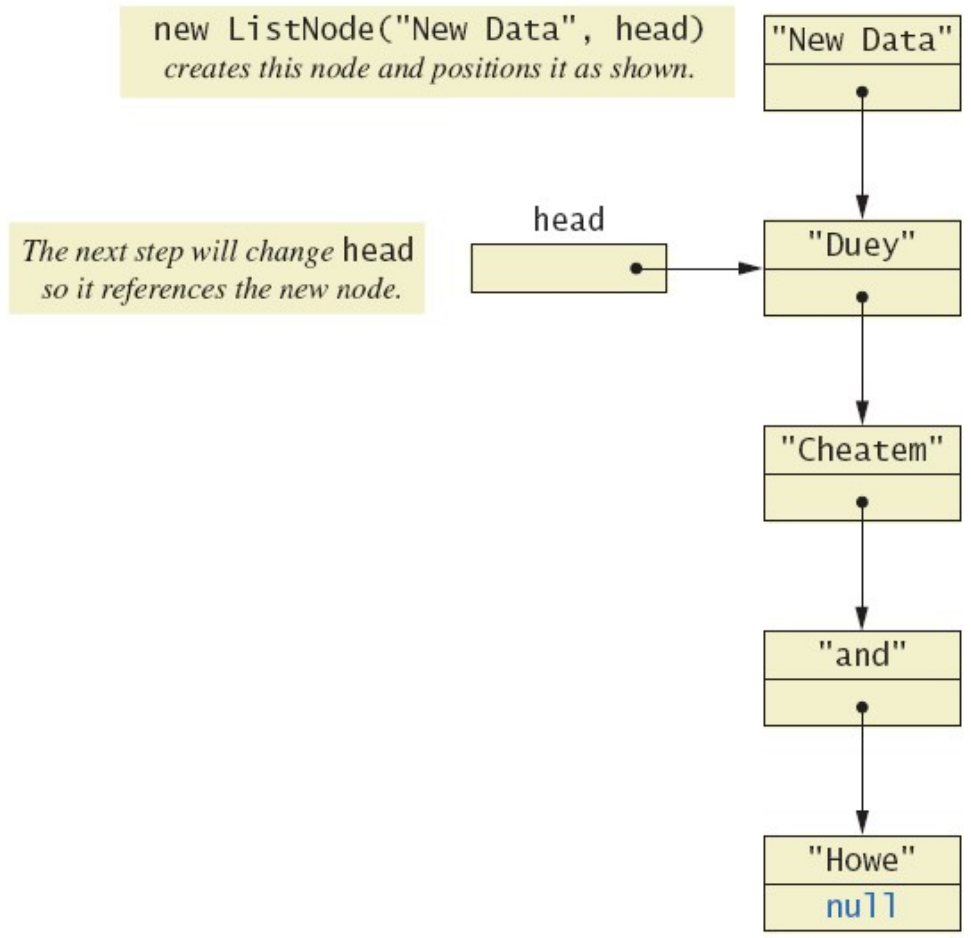
To add a node at the beginning of the list:

```
public void addANodeToStart(String addData)
{
   head = new ListNode(addData,      );
}
```

- The new node will point to the old start of the list, which is what `head` pointed to.
- The value of `head` is changed to point to the new node, which is now the first node in the list.

Implementing Operations of Linked Lists

- Figure 12.6 Adding a node at the start of a linked list



Deleting a Node

To delete a node from the beginning of the list:

```
public void deleteHeadNode()
{
   if (head != null)
   {
      head =
   }
   else
        // prints an error message and exits
...
```

- Doesn't try to delete from an empty list.
- Removes first element and sets head to point to the node that was second but is now first.

Moving Down a Linked List

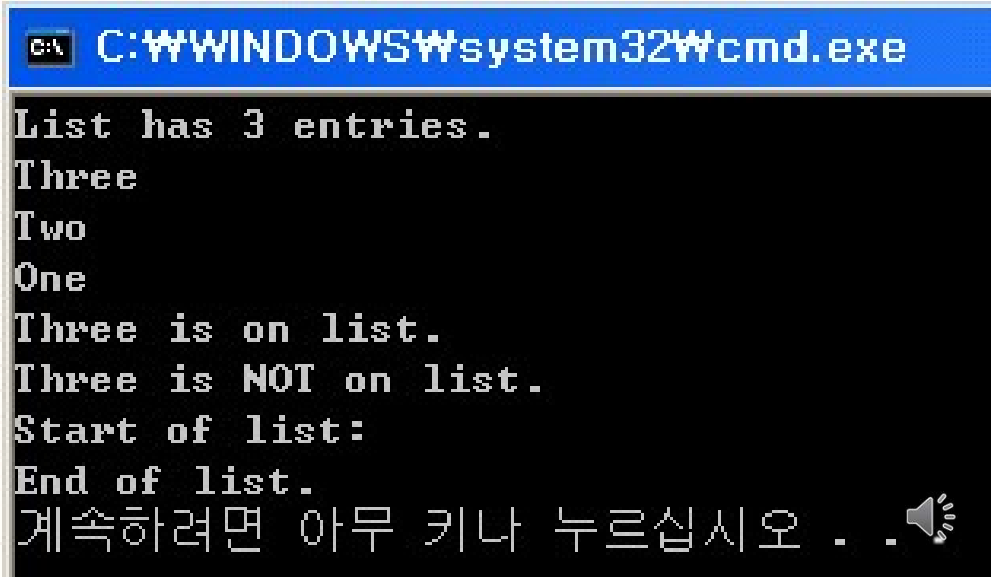
- Moving Down a Linked List

```
private ListNode Find(String target)
  {
     ListNode position;
     position = head;
     String dataAtPosition;
     while (position != null)
     {
        dataAtPosition = position.getData( );
        if (dataAtPosition.equals(target))
           return position;
     }
     //target was not found,
     return null;
  }
```

Listing 12.6 A Linked List Demonstration - StringLinkedListDemo.java

```
// 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.");
    }
}
```



Gotcha: Null Pointer Exception

- Null pointer exception occurs when your code tries to access some member of a class variable and the class variable does not name an object.
- List nodes use `null` to indicate a link instance variable contains no reference.
- `NullPointerException` is not an exception that has to be caught or declared.
  - » Usually indicates you need to fix your code, not add a `catch` block.

2) INNER NODE

Node Inner Classes

```
public class StringLinkedList
{
    private ListNode head;
        <methods for StringLinkedList inserted here>

    private class ListNode
    {
        <Define ListNode instance variables and methods here>
    }
}
```

- Using an inner class makes `StringLinkedList` self-contained because it doesn't depend on a separate file
- Making the inner class **private** makes it safer from the point of view of information hiding

```
// Listing 12.7

public class StringLinkedListSelfContained
{
    private ListNode head;
     ……….
    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;
        }
    }
}
```

```
// Listing 12.7

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;
        }
    }
```

```
public void showList( )
  {
      ListNode position;
      position = head;
      while (position != null)
      {
System.out.println(position.getData( ));
          position = position.getLink( );
      }
  }
```

```
/**
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);
}
```

```
// A linked List Class

public class StringLinkedList
{
  ……..
   public int length( )
   {
      int count = 0;
      ListNode position = head;
      while (position != null)
      {
         count++;
         position = position.getLink( );
      }
      return count;
   }
```

```
/**
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;
        }
    }
}
```

Listing12.8 Placing the Linked-List Data into an Array

```
public String[] arrayCopy( )
{
   String[] a = new String[length( )];

   ListNode position;
   position = head;
   int i = 0;
   while (position != null)
   {
      a[i] = position.data;
      i++;
      position = position.link;
   }

   return a;
}
```

3) ITERATOR

Iterators

- An object that allows a program to step through all the objects and do some action on each one is called an ***iterator***.
- For arrays, an index variable can be used as an iterator, with the action of going to the next thing in the list being something like:

```
index++;
```

- In a linked list, a reference to the node can be used as an iterator.
- `StringLinkedListWithIterator` has an instance variable called `current` that is used keep track of where the iteration is.
- The `goToNext` method moves to the next node in the list by using the statement:

```
current = current.link;
```

// Listing 12.9

```
/** 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;
    }
    /**    Returns true if iteration is not finished.    */
    public boolean moreToIterate ()
    {
        return
    }
```

```
/**
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);
    }
}
```

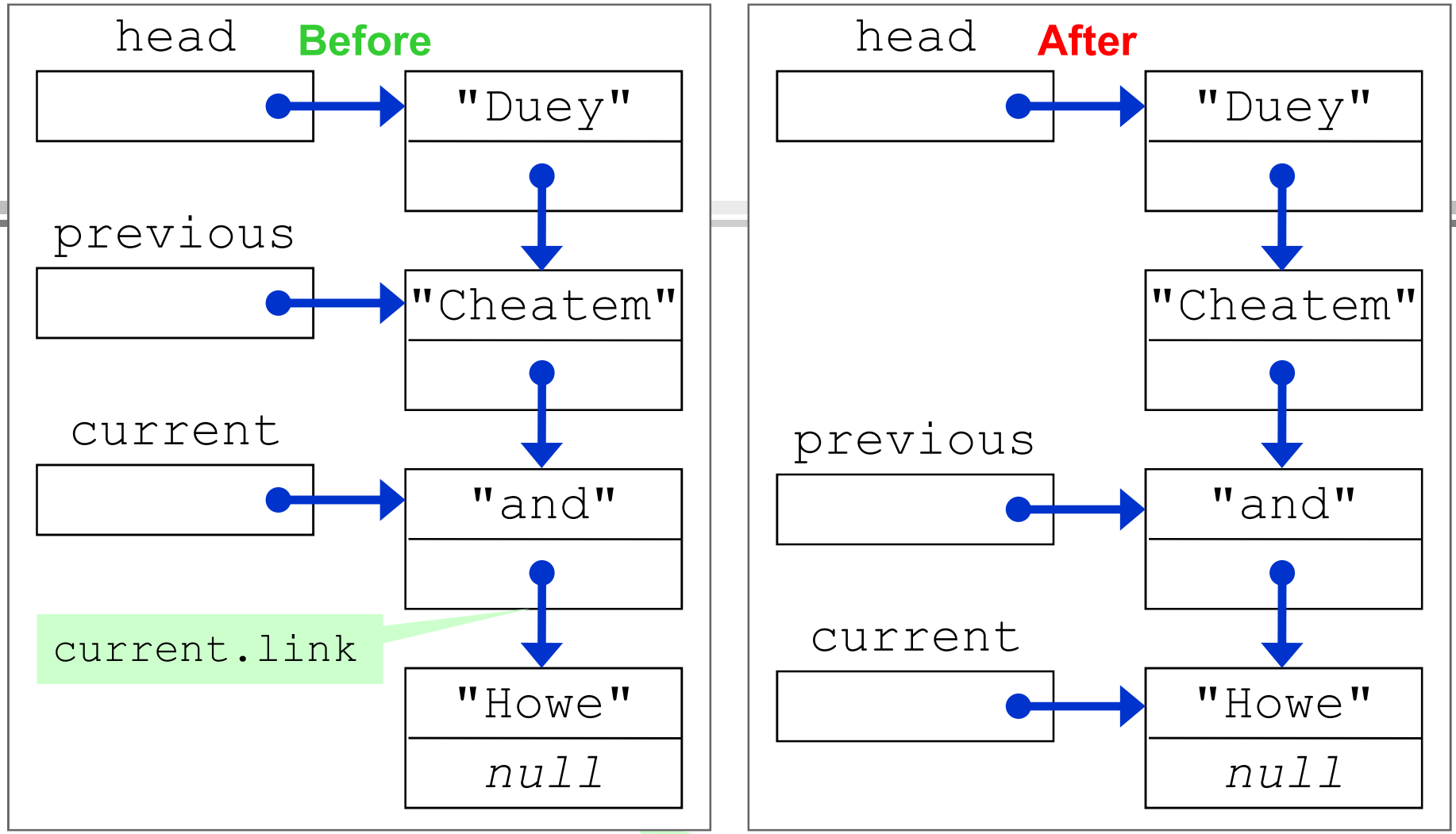
```
public void addANodeToStart (String addData)
{
    head = new ListNode (addData, head);
    if ((current == head.link) && (current != null))
        //if current is at old start node
        previous = head;
}

/**    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;
}
```

```
/**
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 ListNode ();
    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);
    }
}
```



goToNext

Figure 12.7

`current = current.link` gives `current` a reference to this node

```
/**    Advances iterator to next node.    */
public void goToNext ()
{
    if (current != null)
    {
        previous =
        current =
    }
    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);
    }
}
```

Other Methods in the Linked List with Iterator

- `getDataAtCurrent`—returns the data part of the node that the iterator (`current`) is at
- `moreToIterate`—returns a `boolean` value that will be true if the iterator is not at the end of the list
- `resetIteration`—moves the iterator to the beginning of the list

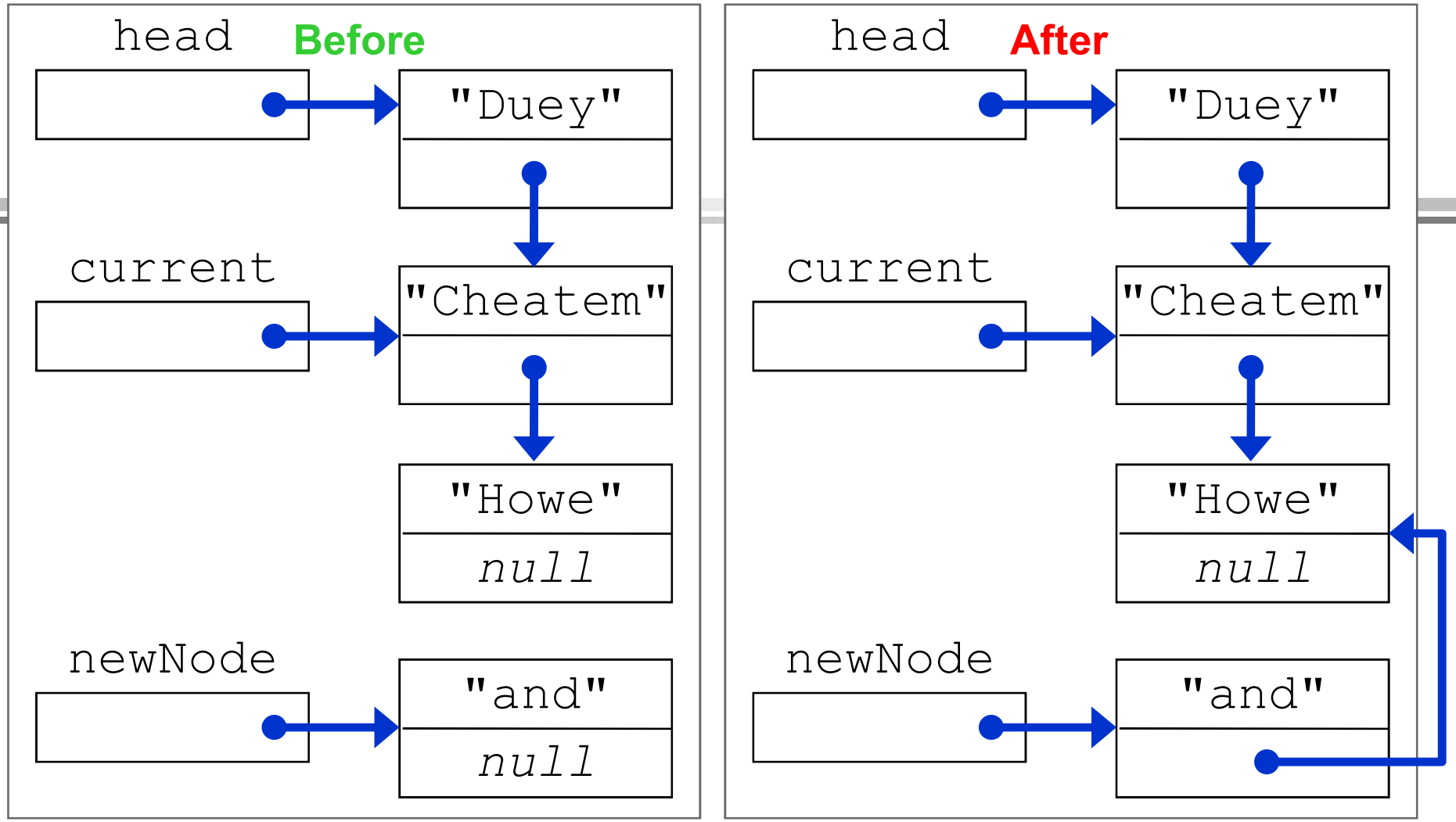
- Can write methods to add and delete nodes at the iterator instead of only at the head of the list.
  - » Following slides show diagrams illustrating the add and delete methods.

Adding a Node (After Current node)

- public void insertNodeAfterCurrent(String newData)
- After creating the node, the two statements used to add the node to the list are:

```
newNode.link =
current.link =
```

- What would happen if these two steps were done in reverse order?



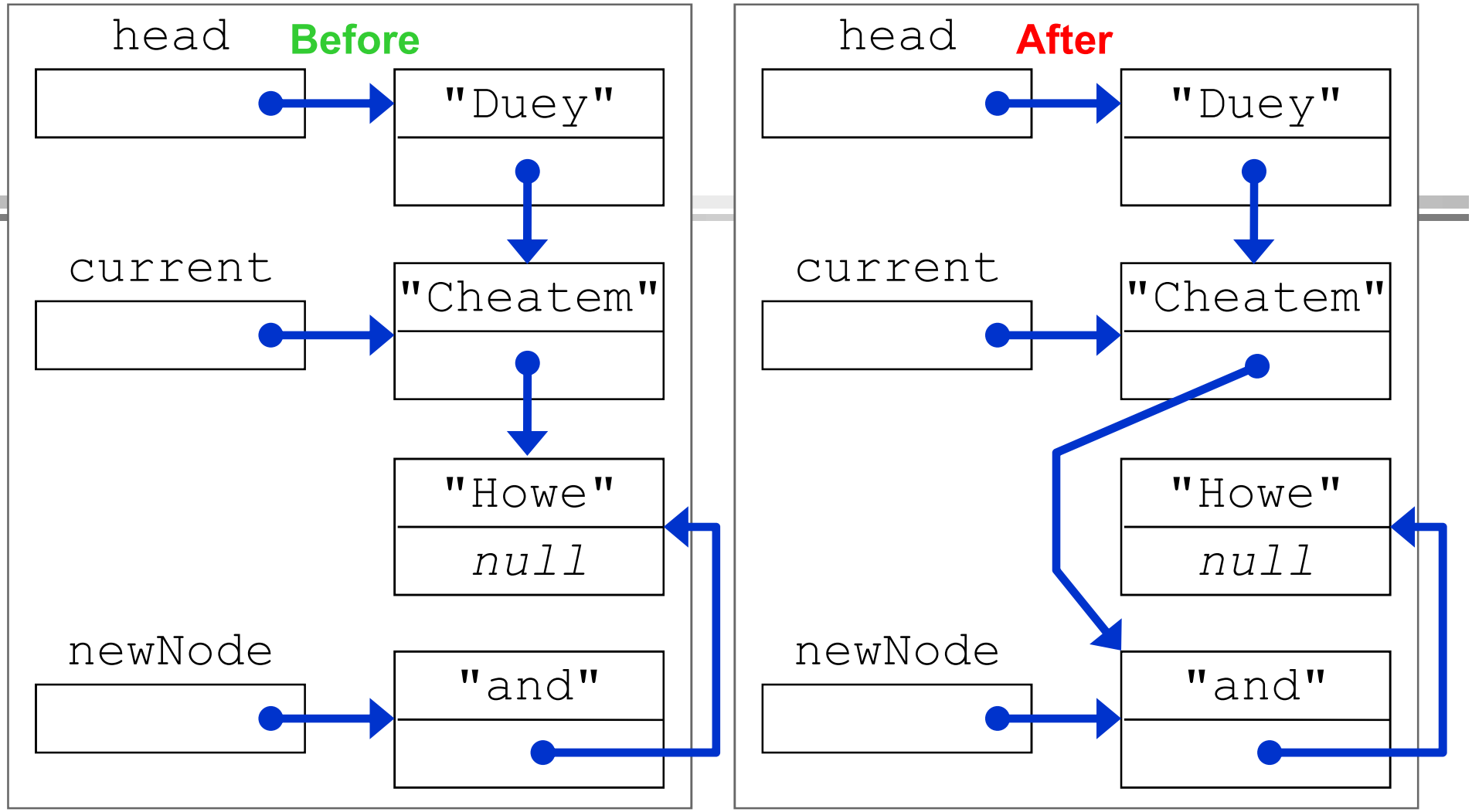
Adding a Node

Step 1()

Create the node with reference `newNode`

Add data to the node

```
newNode.link =
```



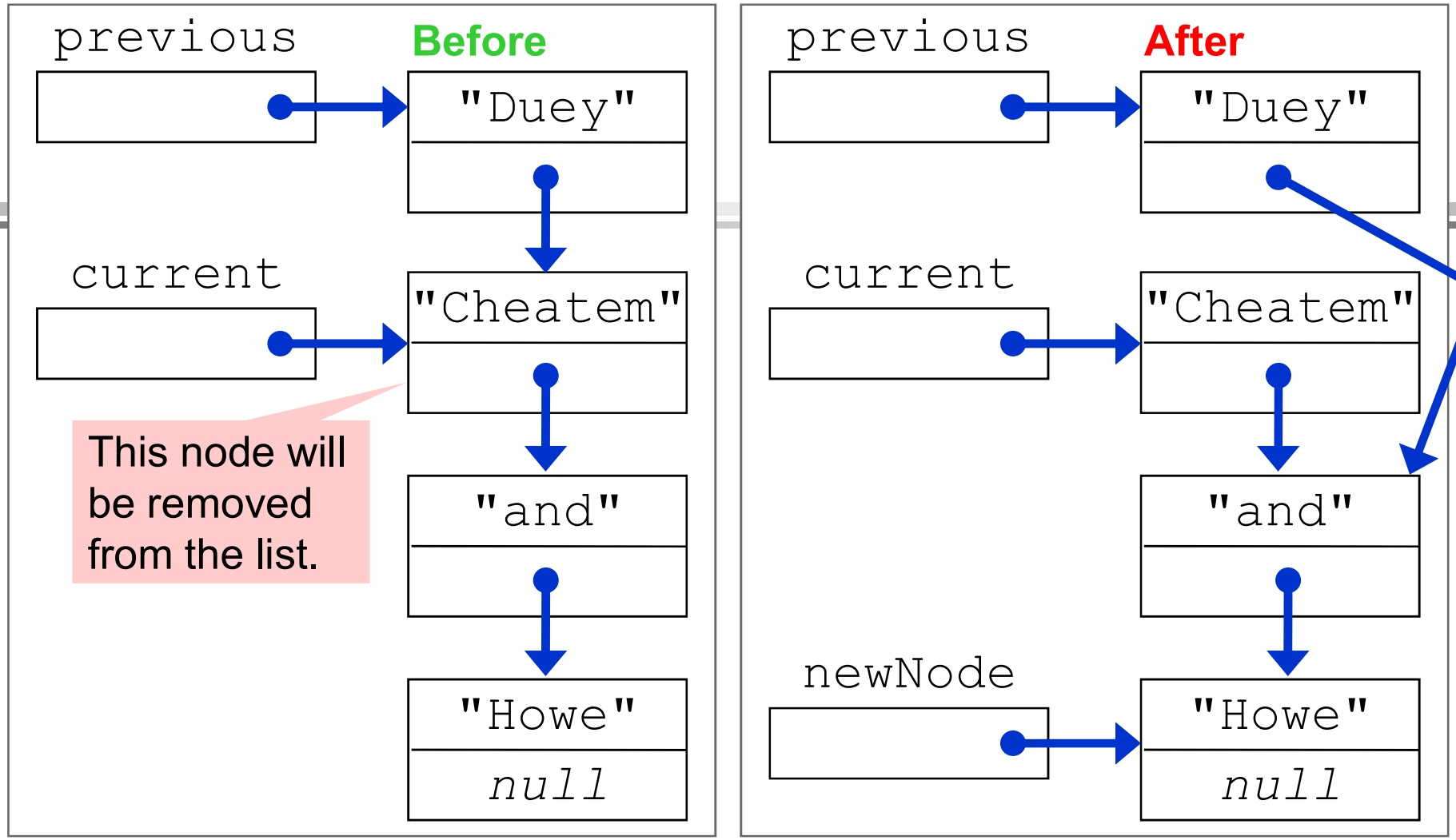
Adding a Node

Step 2

`current.link =`

The node has been added to the list although it might appear out of place in this diagram.

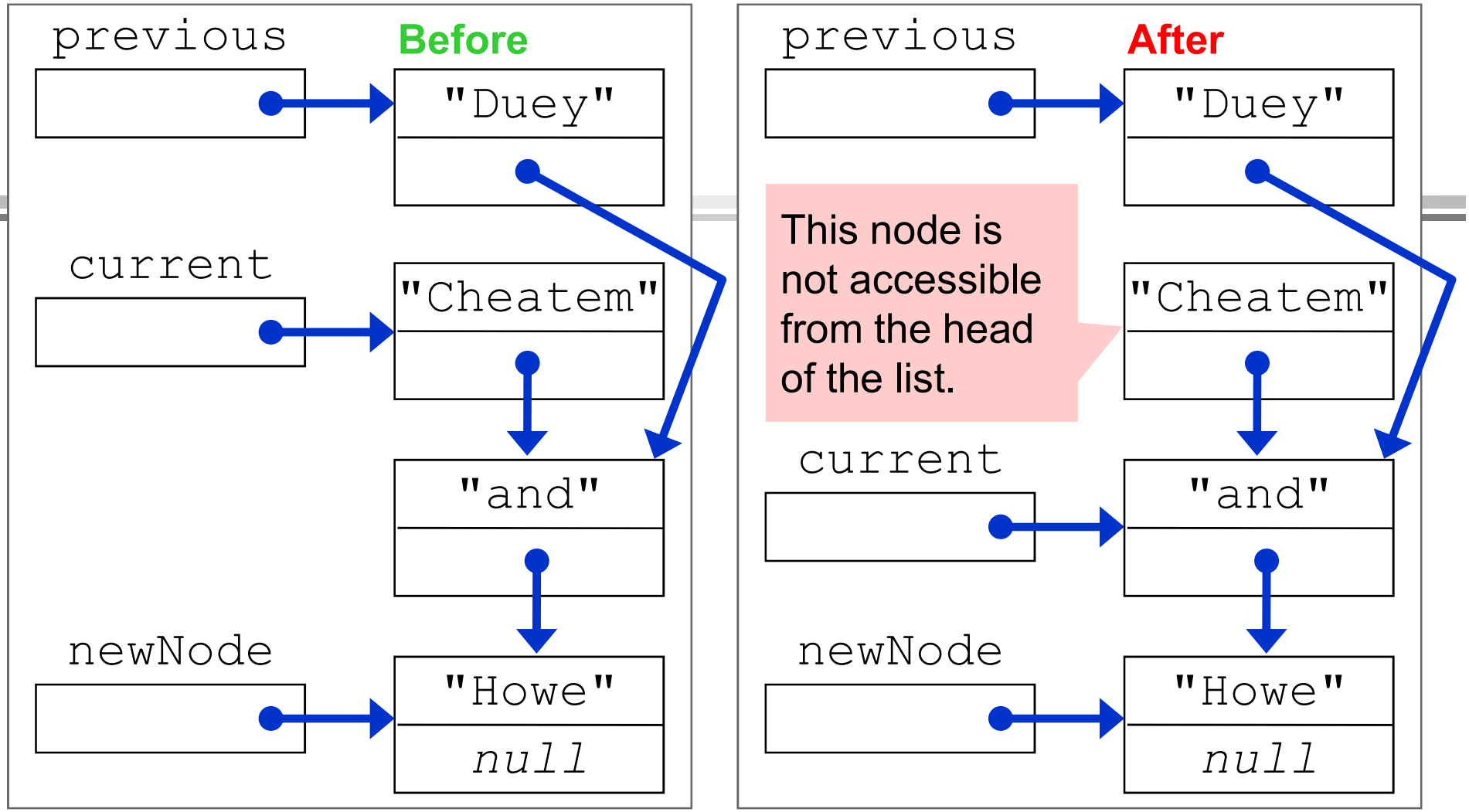
```
/**    Inserts a new node containing newData after the current node.
The current node is the same after invocation as it is before.
Precondition: List is not empty; current node is not  beyond the entire list. */
public void insertNodeAfterCurrent (String newData)
{
    ListNode newNode = new ListNode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



Deleting a Node

What should be done next?

Step 1.



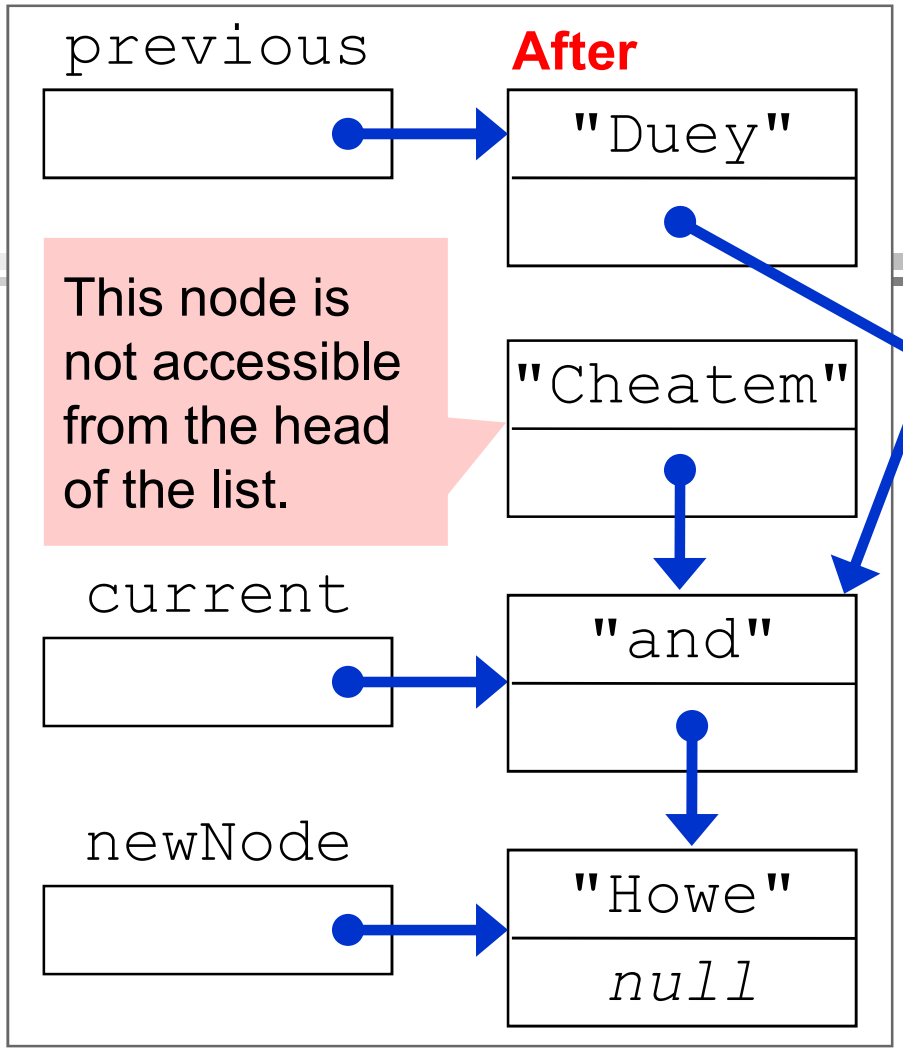
Deleting a Node

Step 2

The node has been deleted from the list although it is still shown in this picture.

```
/**    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);
    }
}
```

FAQ: What Happens to a Deleted Node?



- The `Cheatem` node has been deleted from the list.
- If there are no other references to the deleted node, the storage should be released for other uses.
  - » Some programming languages make the programmer responsible for ***garbage collection***.
  - » Java provides ***automatic garbage collection***.

    Storage used by the `Cheatem` node will be available for other uses without the programmer having to do anything.

Exception Handling with Linked Lists

- handling errors in `StringLinkedListWithIterator` class
  - » original version prints error message and ends program
  - » allow more options by using exceptions

- `LinkedListException` Class
  - » programmer can still end program if desired
  - » programmer can use exceptions for things like detecting end of list

- Java `Iterator` interface uses exceptions
  - » see Appendix 7 for information about the `Iterator` interface

Listing 12.10 The LinkedListException Class - The LinkedListException.java

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

StringLinkedListWithIterator2.java

```
public void insertNodeAfterCurrent(String newData)
                              throws LinkedListException
  {
     ListNode newNode = new ListNode( );
     newNode.data = newData;
     if (current != null)
     {
        newNode.link = current.link;
        current.link = newNode;
     }
     else if (head != null)
     {
        throw new LinkedListException(
                    "Inserting when iterator is past all"
                  + " nodes or uninitialized iterator.");
     }
     else
     {
        throw new LinkedListException(
           "Using insertNodeAfterCurrent with empty list");
     }
  }
```

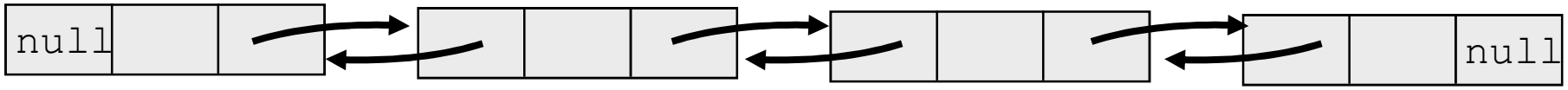
```
/**    Inserts a new node containing newData after the current node.
The current node is the same after invocation as it is before.
Precondition: List is not empty; current node is not  beyond the entire list. */
public void insertNodeAfterCurrent (String newData)
{
    ListNode newNode = new ListNode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

A Doubly Linked List

- A doubly linked list allows the program to move backward as well as forward in the list.
- The beginning of the node class for a doubly-linked list would look something like this:

```
private class ListNode
{
    private Object data
    private ListNode next;
    private ListNode previous;
```

Declaring the data reference as class `Object` allows any kind of data to be stored in the list.



Other Linked Data Structures

- ***tree*** data structure
  - » each node leads to multiple other nodes
- ***binary tree***
  - » each node leads to at most two other nodes
- ***root***—top node of tree
  - » normally keep a reference to root, as for head node of list