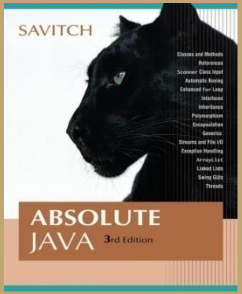
Comp 249
Programming Methodology
Chapter 15

Linked Data Structure – Part A

Dr. Aiman Hanna

Department of Computer Science & Software Engineering
Concordia University, Montreal, Canada

These slides has been extracted, modified and updated from original slides of Absolute Java 3rd Edition by Savitch; which has originally been prepared by Rose Williams of Binghamton University. Absolute Java is published by Pearson Education / Addison-Wesley.



Introduction to Linked Data Structures

- A *linked data structure* consists of capsules of data known as *nodes* that are connected via *links*
  - Links can be viewed as arrows and thought of as one way passages from one node to another
- In Java, nodes are realized as objects of a node class
- The data in a node is stored via instance variables
- The links are realized as references
  - A reference is a memory address, and is stored in a variable of a class type
  - Therefore, a link is an instance variable of the node class type itself

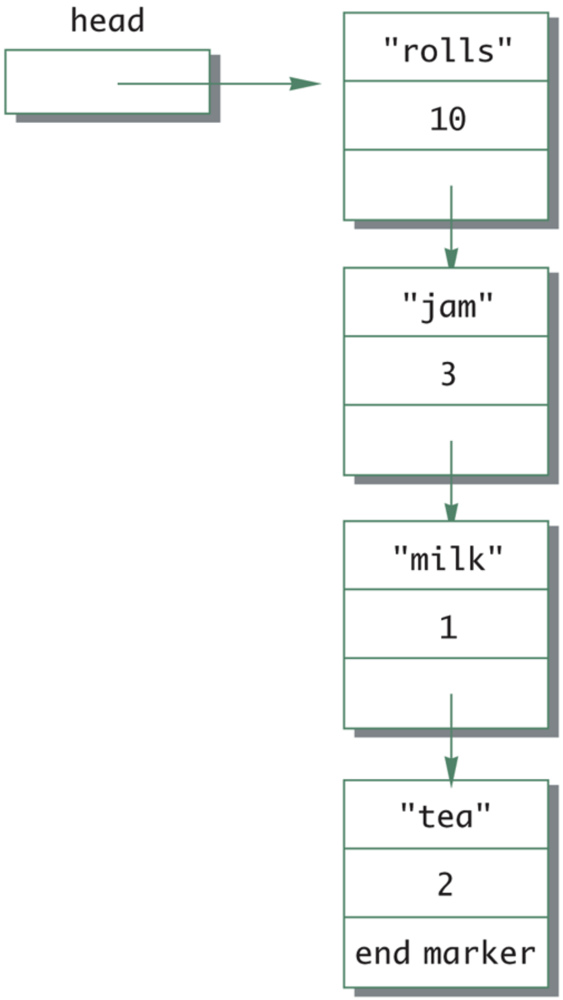
Java Linked Lists

- The simplest kind of linked data structure is a *linked list*

- A linked list consists of a single chain of nodes, each connected to the next by a link
  - The first node is called the *head* node
  - The last node serves as a kind of end marker

Nodes and Links in a Linked List

Display 15.1 **Nodes and Links in a Linked List**



A Simple Linked List Class

- In a linked list, each node is an object of a node class
  - Note that each node is typically illustrated as a box containing one or more pieces of data
- Each node contains data and a link to another node
  - A piece of data is stored as an instance variable of the node
  - Data is represented as information contained within the node "box"
  - Links are implemented as references to a node stored in an instance variable of the node type
  - Links are typically illustrated as arrows that point to the node to which they "link“

See LinkedList1.java

See LinkedList2.java

See LinkedList3.java

See LinkedList4.java

A Simple Linked List Class

- The first node, or start node in a linked list is called the head node
  - The entire linked list can be traversed by starting at the head node and visiting each node exactly once
- There is typically a variable of the node type (e.g., **`head`**) that contains a reference to the first node in the linked list
  - However, it is not the head node, nor is it even a node
  - It simply contains a reference to the head node

A Simple Linked List Class

- A linked list object contains the variable **`head`** as an instance variable of the class
- A linked list object does not contain all the nodes in the linked list directly
  - Rather, it uses the instance variable **`head`** to locate the head node of the list
  - The head node and every node of the list contain a link instance variable that provides a reference to the next node in the list
  - Therefore, once the head node can be reached, then every other node in the list can be reached

An Empty List Is Indicated by `null`

- The **`head`** instance variable contains a reference to the first node in the linked list
  - If the list is empty, this instance variable is set to **`null`**
  - Note: This is tested using **`==`**, not the **`equals`** method

- The linked list constructor sets the head instance variable to **`null`**
  - This indicates that the newly created linked list is empty

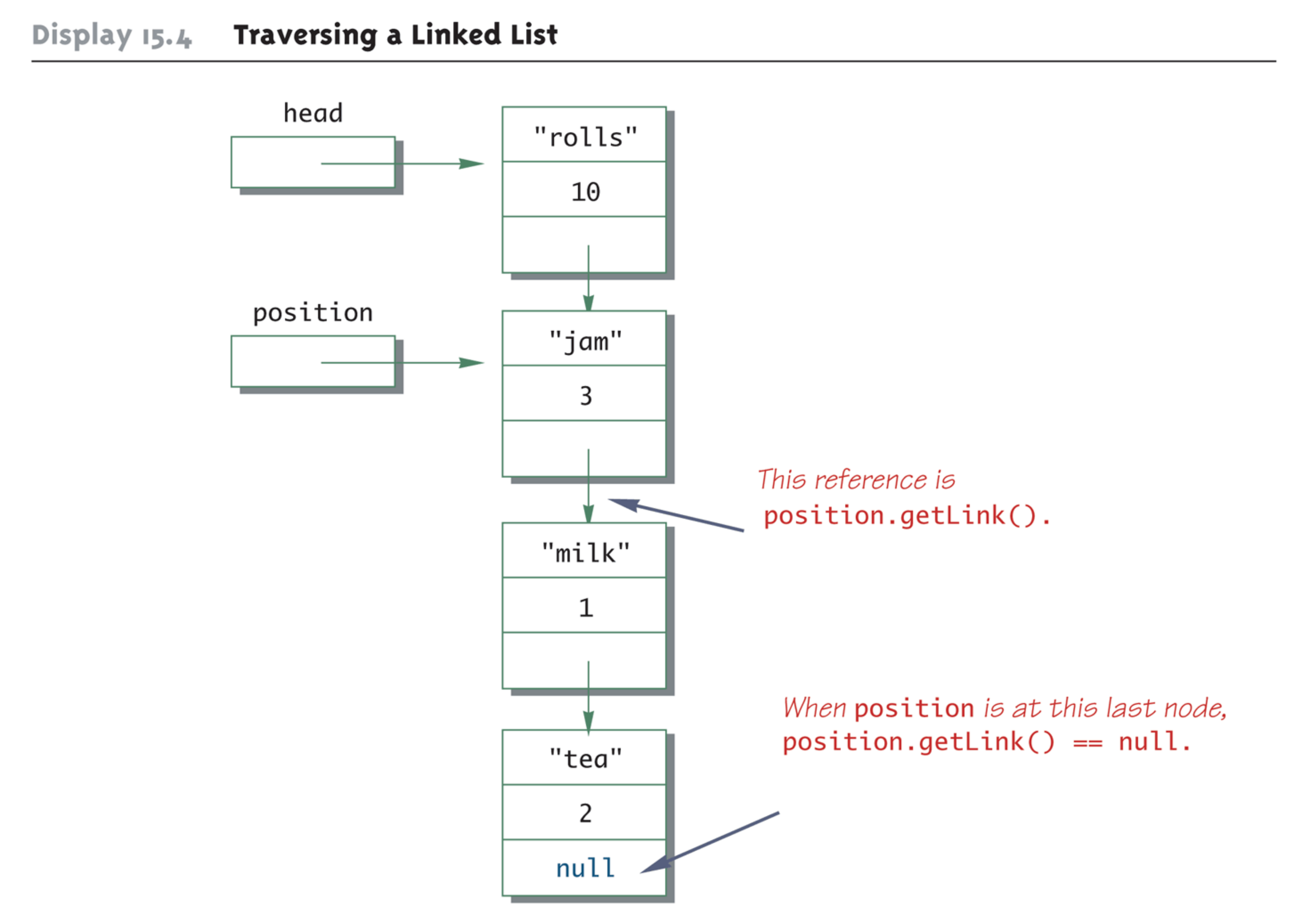
Indicating the End of a Linked List

- The last node in a linked list should have its link instance variable set to **`null`**
  - That way the code can test whether or not a node is the last node
  - Note: This is tested using **`==`**, not the **`equals`** method

Traversing a Linked List

- If a linked list already contains nodes, it can be traversed as follows:
  - Set a local variable equal to the value stored by the head node (its reference)
  - This will provides the location of the first node
  - After accessing the first node, the accessor method for the link instance variable will provide the location of the next node
  - Repeat this until the location of the next node is equal to **`null`**

Traversing a Linked List

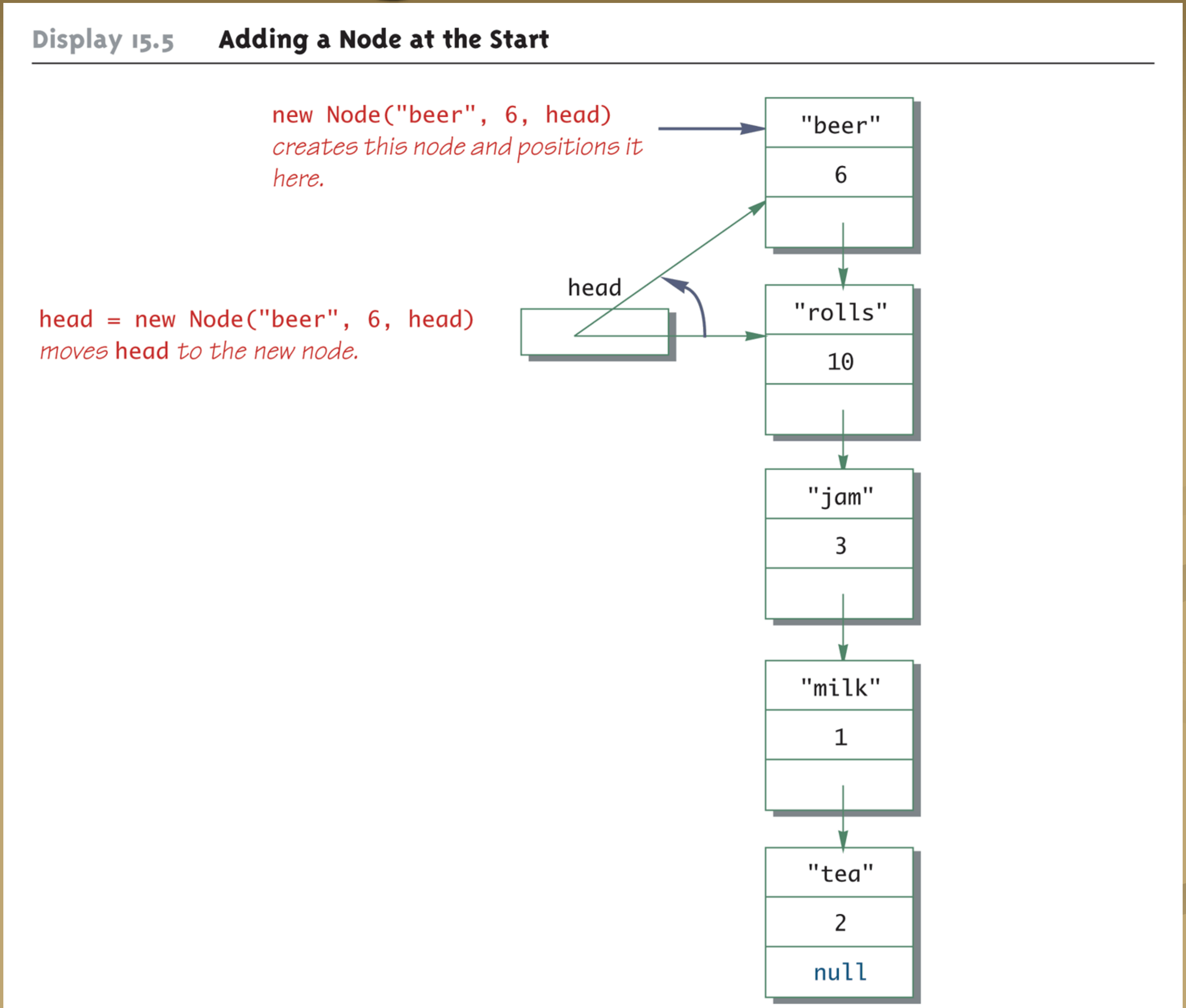


Display 15.4 **Traversing a Linked List**

This reference is `position.getLink().`

When `position` *is at this last node,* `position.getLink() == null.`

Adding a Node at the Start



Display 15.5 **Adding a Node at the Start**

`new Node("beer", 6, head)` *creates this node and positions it here.*

`head = new Node("beer", 6, head)` *moves* **head** *to the new node.*

Deleting the Head Node from a Linked List

- The method **`deleteHeadNode`** removes the first node from the linked list
  - It leaves the **`head`** variable pointing to (i.e., containing a reference to) the old second node in the linked list

- The deleted node will automatically be collected and its memory recycled, along with any other nodes that are no longer accessible
  - In Java, this process is called *automatic garbage collection*

Linked Lists Copy Constructors and `clone` Methods

- There is a simple way to define copy constructors and the **`clone`** method for data structures such as linked lists
  - Unfortunately, this approach produces only shallow copies
- Further coding is needed by the programmer in order to create copy constructor and clone() methods that perform deep copy

Pitfall: Privacy Leaks

- You should be careful with privacy leak.
  - If the node class accessor method returns a reference to a node, then the `private` restriction on the instance variables can be easily defeated
  - The easiest way to fix this problem would be to make the node class a private inner class in the linked list class

See LinkedList5.java

Node Inner Class vs. Node External Class

- Note that the linked list class discussed so is designed to have the node class as an inner class
- In that case, the linked list, or similar data structure, is made self-contained by making the node class an inner class
- A node inner class so defined should be made private, unless used elsewhere
  - This can simplify the definition of the node class by eliminating the need for accessor and mutator methods
  - Since the instance variables are private, they can be accessed directly from methods of the outer class without causing a privacy leak

Node Inner Class vs. Node External Class

- However, it is possible that the list can be made dependent on an external node class

See LinkedList6.java

Adding and Deleting Nodes

- An iterator is normally used to add or delete a node in a linked list

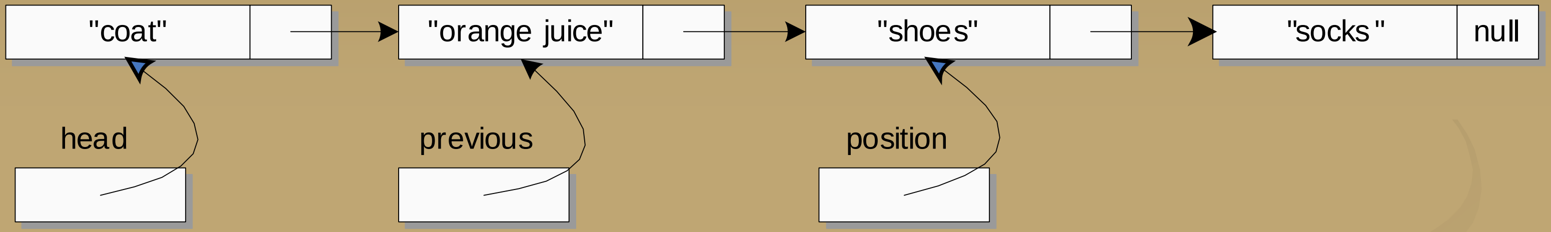
- Given iterator variables **`position`** and **`previous`**, the following two lines of code will delete the node at location **`position`**:

```
previous.link = position.link;
position = position.link;
```

  - Note: **`previous`** points to the node before **`position`**

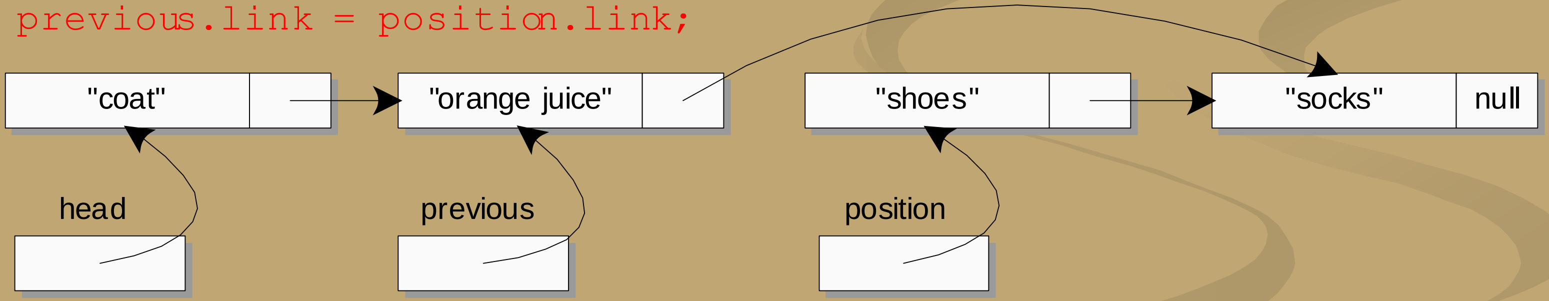
Deleting a Node (Part 1 of 2)

1. Existing list with the iterator positioned at “shoes”



2. Bypass the node at `position` from `previous`

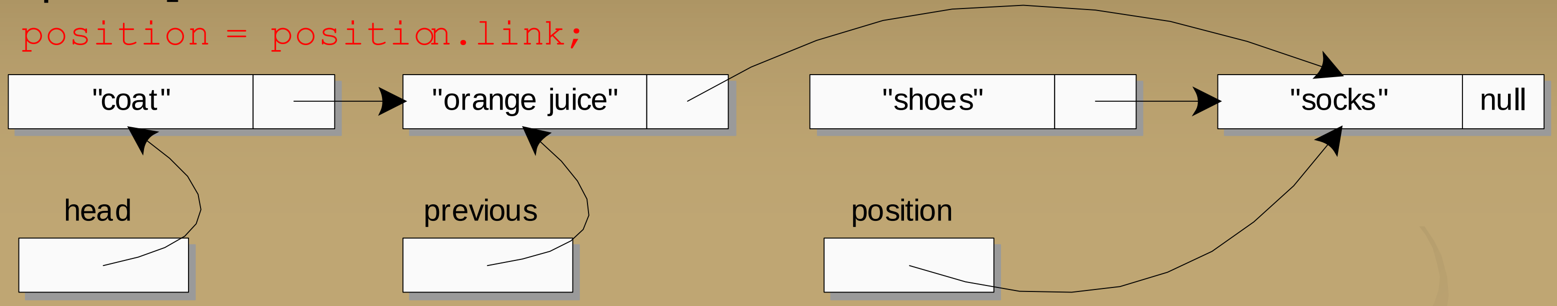
```
previous.link = position.link;
```



Deleting a Node (Part 2 of 2)

3. Update `position` to reference the next node

```
position = position.link;
```



"coat"

"orange juice"

"shoes"

"socks"

null

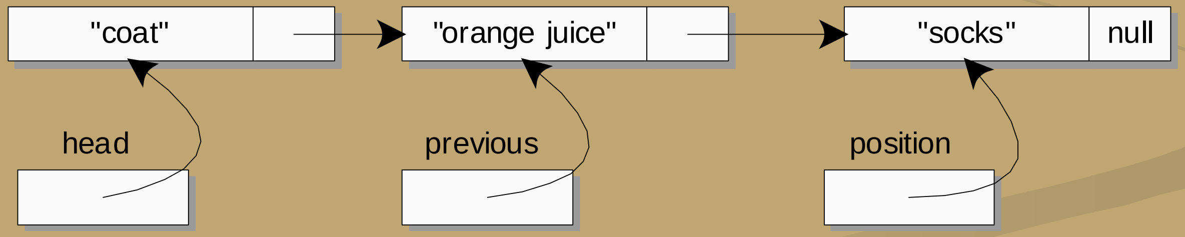
head

previous

position

Since no variable references the node "shoes" Java will automatically recycle the memory allocated for it.

4. Same picture with deleted node not shown



"coat"

"orange juice"

"socks"

null

head

previous

position

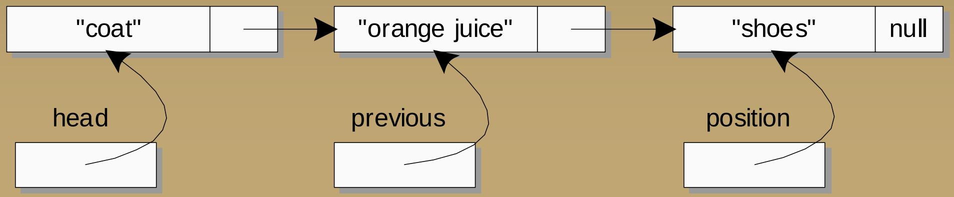
Adding and Deleting Nodes

- Note that Java has *automatic garbage collection*
  - In many other languages the programmer has to keep track of deleted nodes and explicitly return their memory for recycling
  - This procedure is called *explicit memory management*
- The iterator variables `position` and `previous` can be used to add a node as well
  - `previous` will point to the node before the insertion point, and `position` will point to the node after the insertion point

```
Node temp = new Node(newData,position);
previous.link = temp;
```

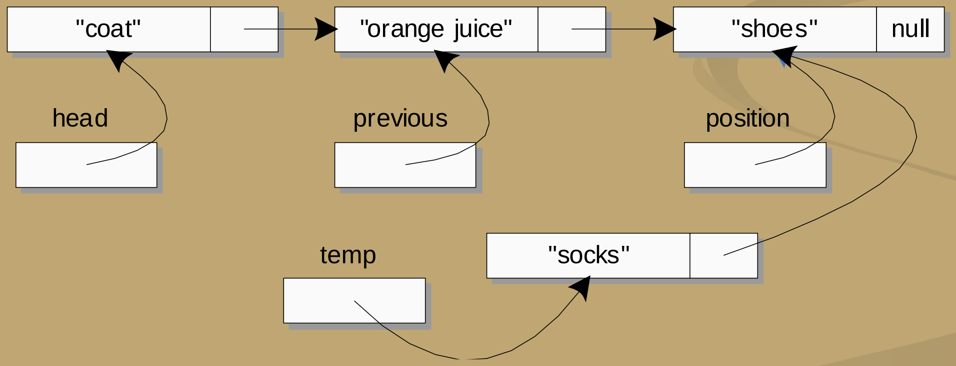
Adding a Node between Two Nodes (Part 1 of 2)

1. Existing list with the iterator positioned at “shoes”



2. Create new Node with "socks" linked to "shoes"

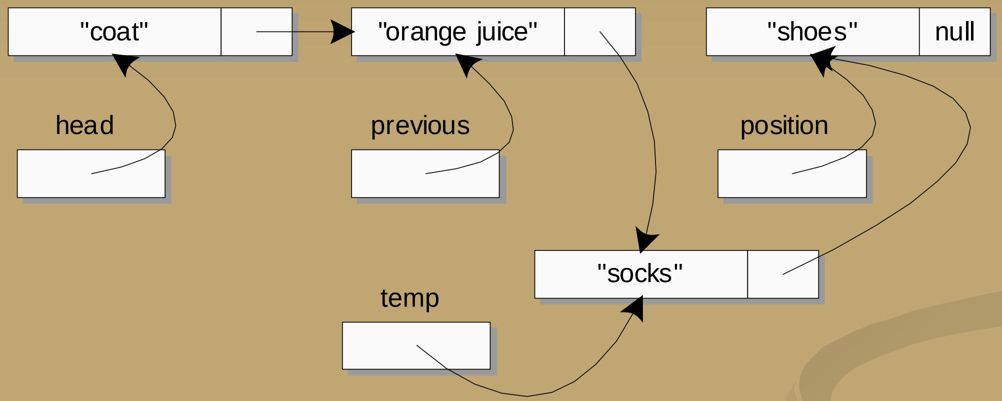
```
temp = new Node(newData, position);  // newData is "socks"
```



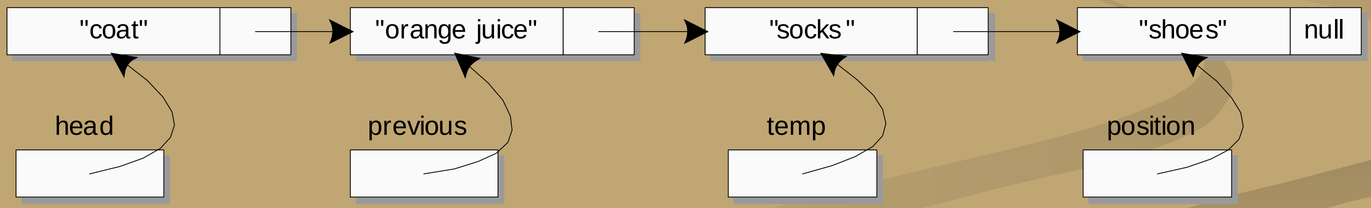
Adding a Node between Two Nodes (Part 2 of 2)

3. Make `previous` link to the Node `temp`

```
previous.link = temp;
```



4. Picture redrawn for clarity, but structurally identical to picture 3



Variations on a Linked List

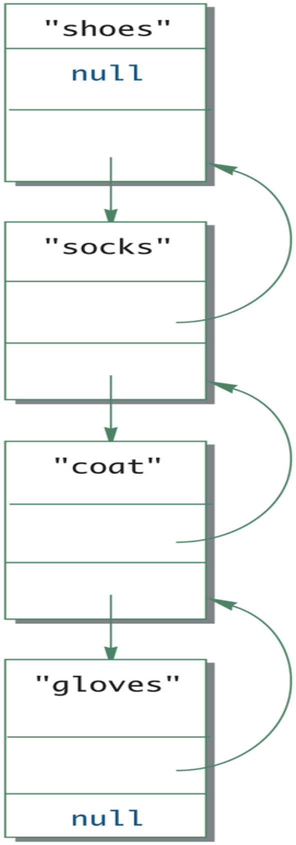
- An ordinary linked list allows movement in one direction only
- However, a doubly linked list has one link that references the next node, and one that references the previous node
- The node class for a doubly linked list can begin as follows:

```
private class TwoWayNode
{
   private String item;
   private TwoWayNode previous;
   private TwoWayNode next;
   . . .
```

- In addition, the constructors and methods in the doubly linked list class would be modified to accommodate the extra link

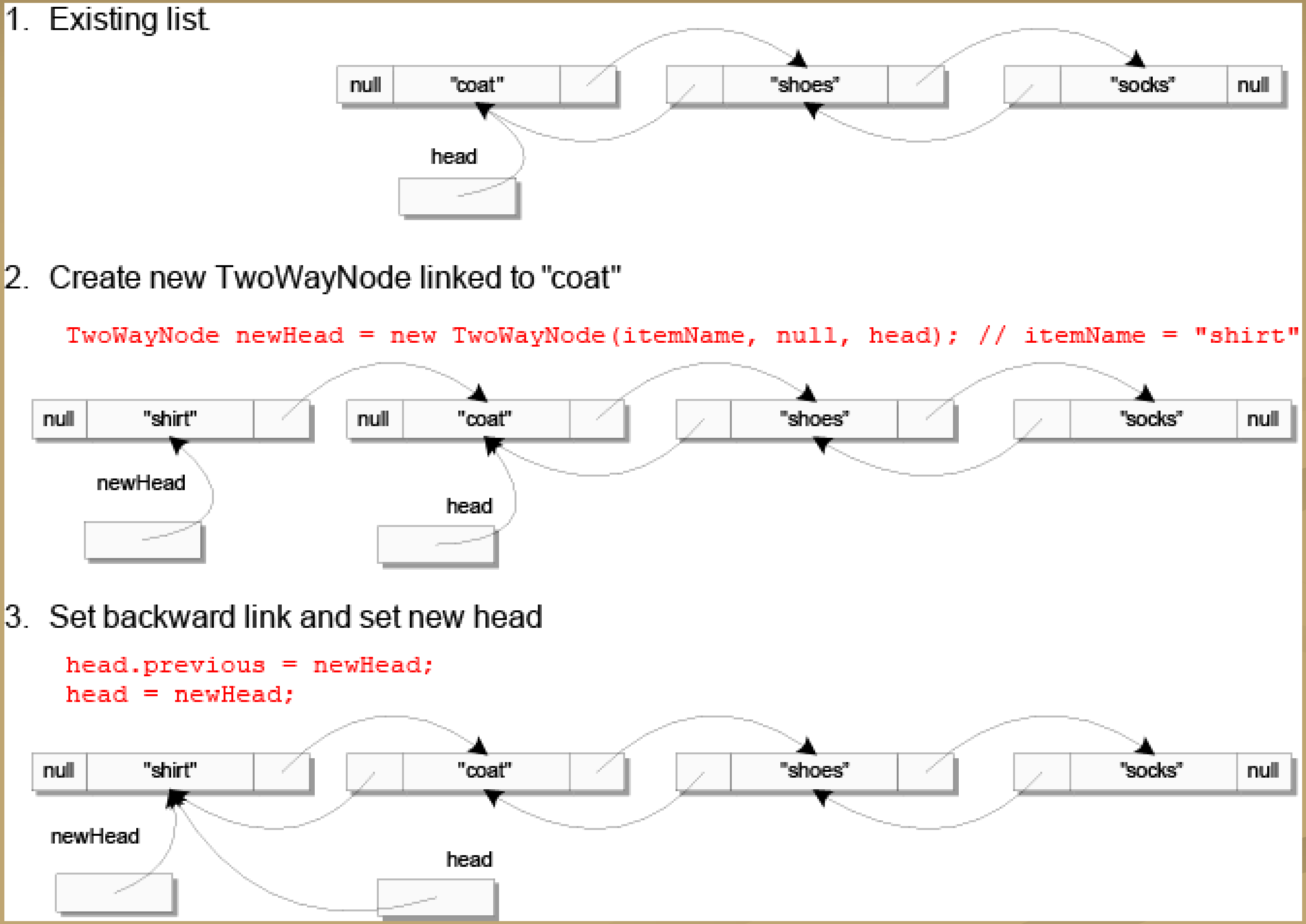
A Doubly Linked List

Display 15.21 **A Doubly Linked List**



See LinkedList10.java

Adding a Node to the Front of a Doubly Linked List



1. Existing list

2. Create new TwoWayNode linked to "coat"

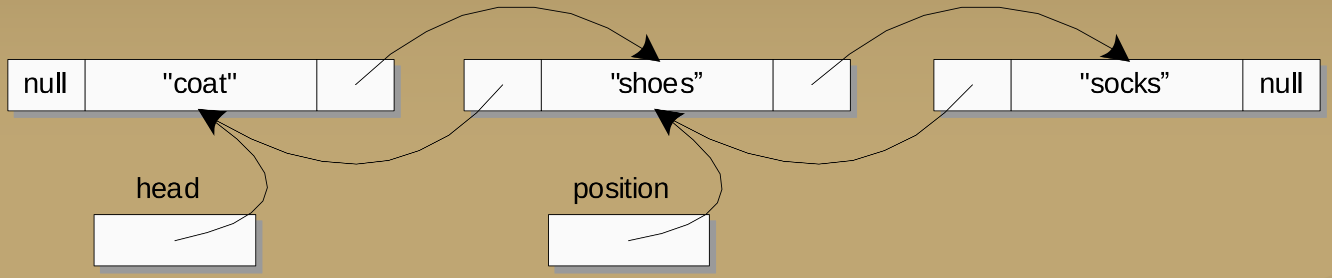
```
TwoWayNode newHead = new TwoWayNode(itemName, null, head); // itemName = "shirt"
```

3. Set backward link and set new head

```
head.previous = newHead;
head = newHead;
```

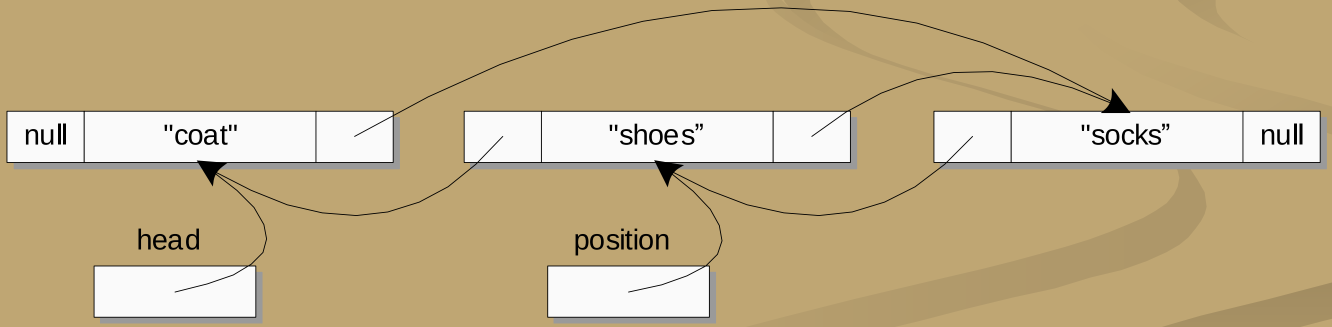
Deleting a Node from a Doubly Linked List (1 of 2)

1. Existing list with an iterator referencing "shoes"



2. Bypass the "shoes" node from the next link of the previous node

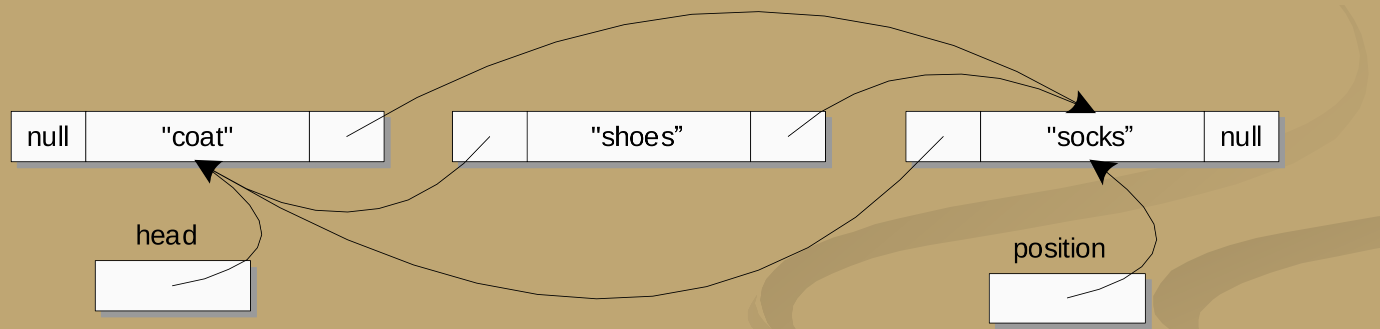
```
position.previous.next = position.next;
```



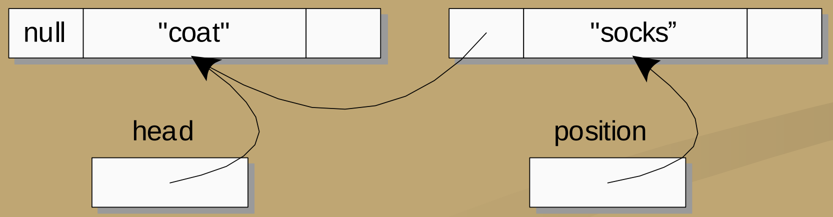
Deleting a Node from a Doubly Linked List (2 of 2)

3. Bypass the "shoes" node from the previous link of the next node and move position off the deleted node

```
position.next.previous = position.previous;
position = position.next;
```

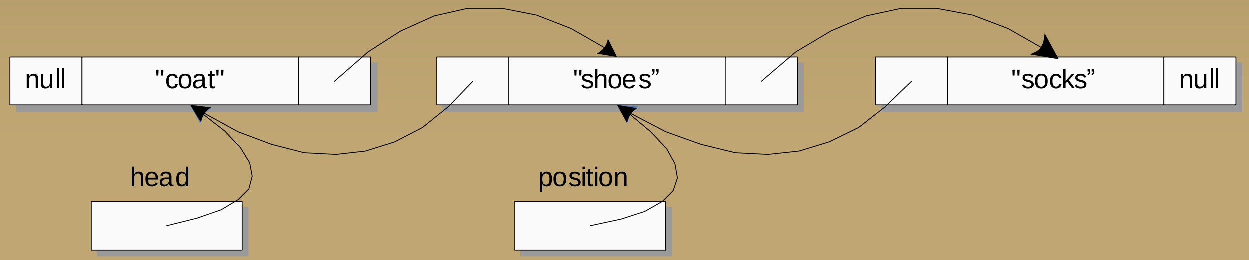


4. Picture redrawn for clarity with the "shoes" node removed since there are no longer references pointing to this node .



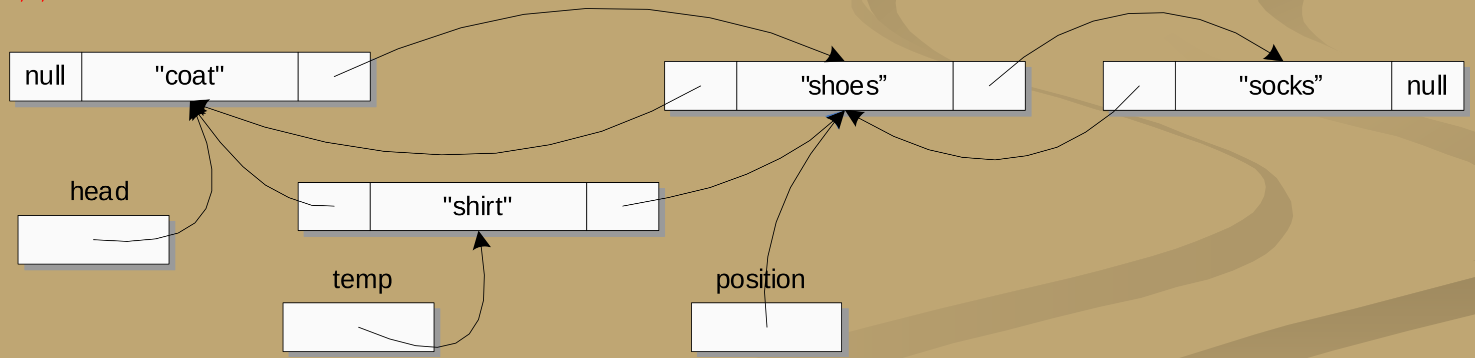
Inserting a Node Into a Doubly Linked List (1 of 2)

1. Existing list with an iterator referencing "shoes"



2. Create new TwoWayNode with previous linked to "coat" and next to "shoes"

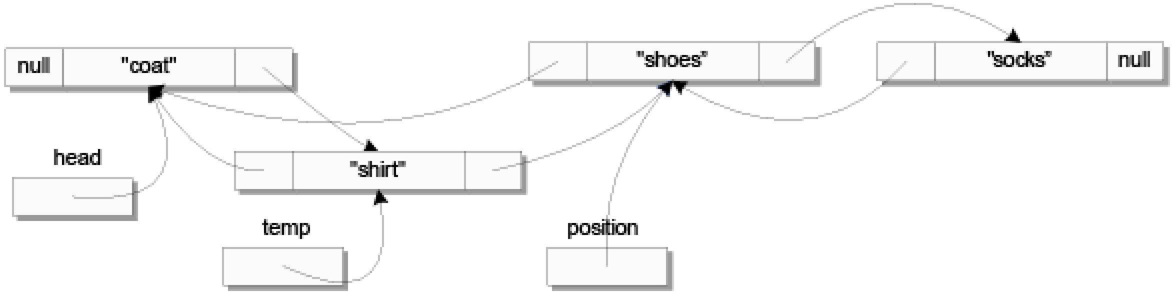
```
TwoWayNode temp = new TwoWayNode(newData, position.previous, position);
// newData = "shirt"
```



Inserting a Node Into a Doubly Linked List (2 of 2)

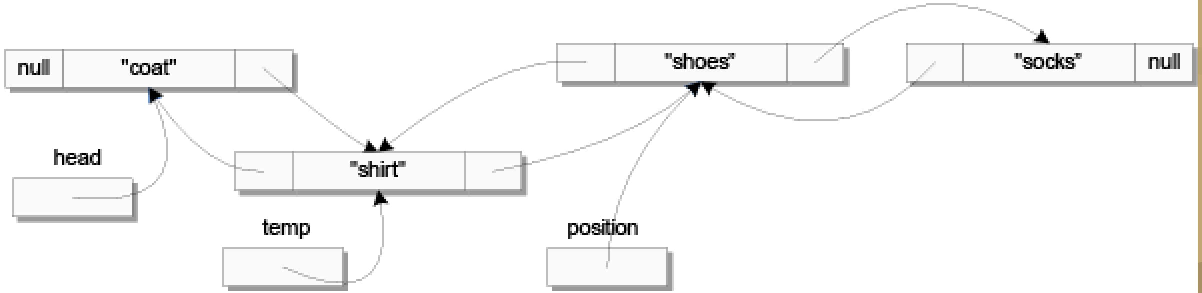
3. Set next link from "coat" to the new node of "shirt"

```
position.previous.next = temp;
```



4. Set previous link from "shoes" to the new node of "shirt"

```
position.previous = temp;
```



A Generic Linked List

- A linked list can be created whose **`Node`** class has a *type parameter* ***`T`*** for the type of data stored in the node
  - Therefore, it can hold objects of any class type, including types that contain multiple instance variable
  - The type of the actual object is plugged in for the type parameter ***`T`***

- For the most part, this class can have the same methods, coded in basically the same way, however some major difference can be there, and adjustment to the code may hence be necessary

- One of such differences is the use of the clone() method

See LinkedList7.java

See LinkedList8.java

Pitfall: The `clone` Method Is Protected in Object

- It would have been preferable to clone the data belonging to the list being copied in the `copyOf` method as follows:

```
nodeReference = new
   Node((T)(position.data).clone(), null);
```

- However, this is not allowed, and this code will not compile
  - The error message generated will state that `clone` is protected in `Object`
  - Although the type used is `T`, not `Object`, any class can be plugged in for `T`
  - When the class is compiled, all that Java knows is that `T` is a descendent class of Object

Tip: Use a Type Parameter Bound for a Better `clone`

- One solution to this problem is to place a bound on the type parameter **`T`** so that it must satisfy a suitable interface
  - Although there is no standard interface that does this, it is easy to define one
- For example, a **`Cloneable2`** interface could be defined

Tip: Use a Type Parameter Bound for a Better `clone`

- Any class that implements the **`Cloneable2`** interface would have these three properties:
  1. It would implement the **`Cloneable`** interface because **`Cloneable2`** extends **`Cloneable`**
  2. It would have to implement a public **`clone`** method
  3. Its **`clone`** method would have to make a deep copy

Tip: Cloning is an "All or Nothing" Affair

- If a `clone` method is defined for a class, then it should follow the official Java guidelines
    - In particular, it should implement the `Cloneable` interface

Exceptions

- A generic data structure is likely to have methods that throw exceptions

- Situations such as a **`null`** argument to the copy constructor may be handled differently in different situations
  - If this happens, it is best to throw a **`NullPointerException`**, and let the programmer who is using the linked list handle the exception, rather than take some arbitrary action
  - A **`NullPointerException`** is an *unchecked* exception: it need not be caught or declared in a throws clause

Pitfall: Using Node instead of Node<T>

- Any names can be substituted for the node **Node** and its parameter **<T>**

- When defining the **List<T>** class, the type for a node is **Node<T>**, not **Node**
  - If the **<T>** is omitted, this is an error for which the compiler may or may not issue an error message (depending on the details of the code), and even if it does, the error message may be quite strange
  - Look for a missing **<T>** when a program that uses nodes with type parameters gets a strange error message or doesn't run correctly

A Generic Linked List: the `equals` Method

- Like other classes, a linked list class should normally have an `equals` method

- The `equals` method can be defined in a number of reasonable ways
  - Different definitions may be appropriate for different situations

- Two such possibilities are the following:
  1. They contain the same data entries (possibly in different orders)
  2. They contain the same data entries in the same order

- Of course, the type plugged in for `T` must also have redefined the `equals` method

Iterators

- A collection of objects, such as the nodes of a linked list, must often be traversed in order to perform some action on each object
  - An *iterator* is any object that enables a list to be traversed in this way

- A linked list class may be created that has an iterator inner class
  - If iterator variables are to be used outside the linked list class, then the iterator class would be made public
  - The linked list class would have an **`iterator`** method that returns an iterator for its calling object
  - Given a linked list named **`list`**, this can be done as follows:

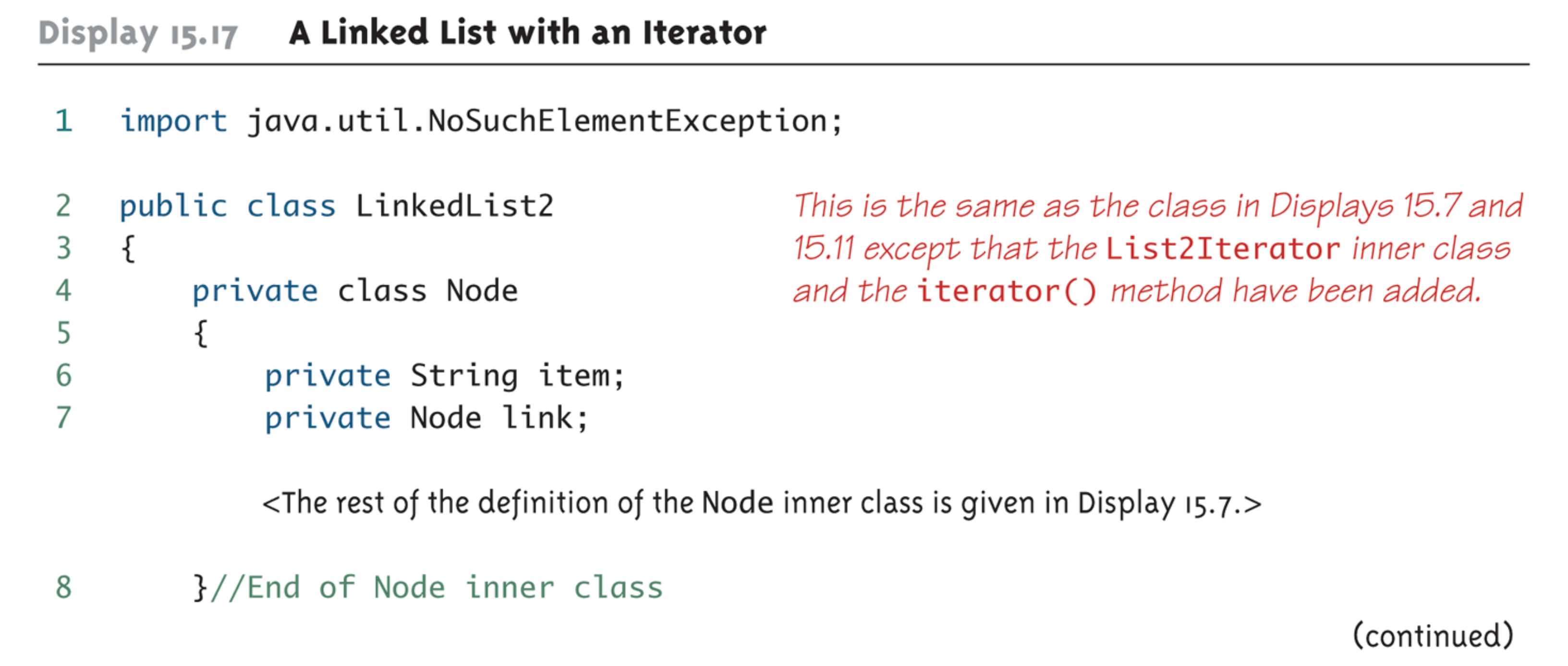
```
LinkedList2.List2Iterator i = list.iterator();
```

Iterators

- The basic methods used by an iterator are as follows:
  - **`restart`**: Resets the iterator to the beginning of the list
  - **`hasNext`**: Determines if there is another data item on the list
  - **`next`**: Produces the next data item on the list

See LinkedList9.java

A Linked List with an Iterator (Part 1 of 6)



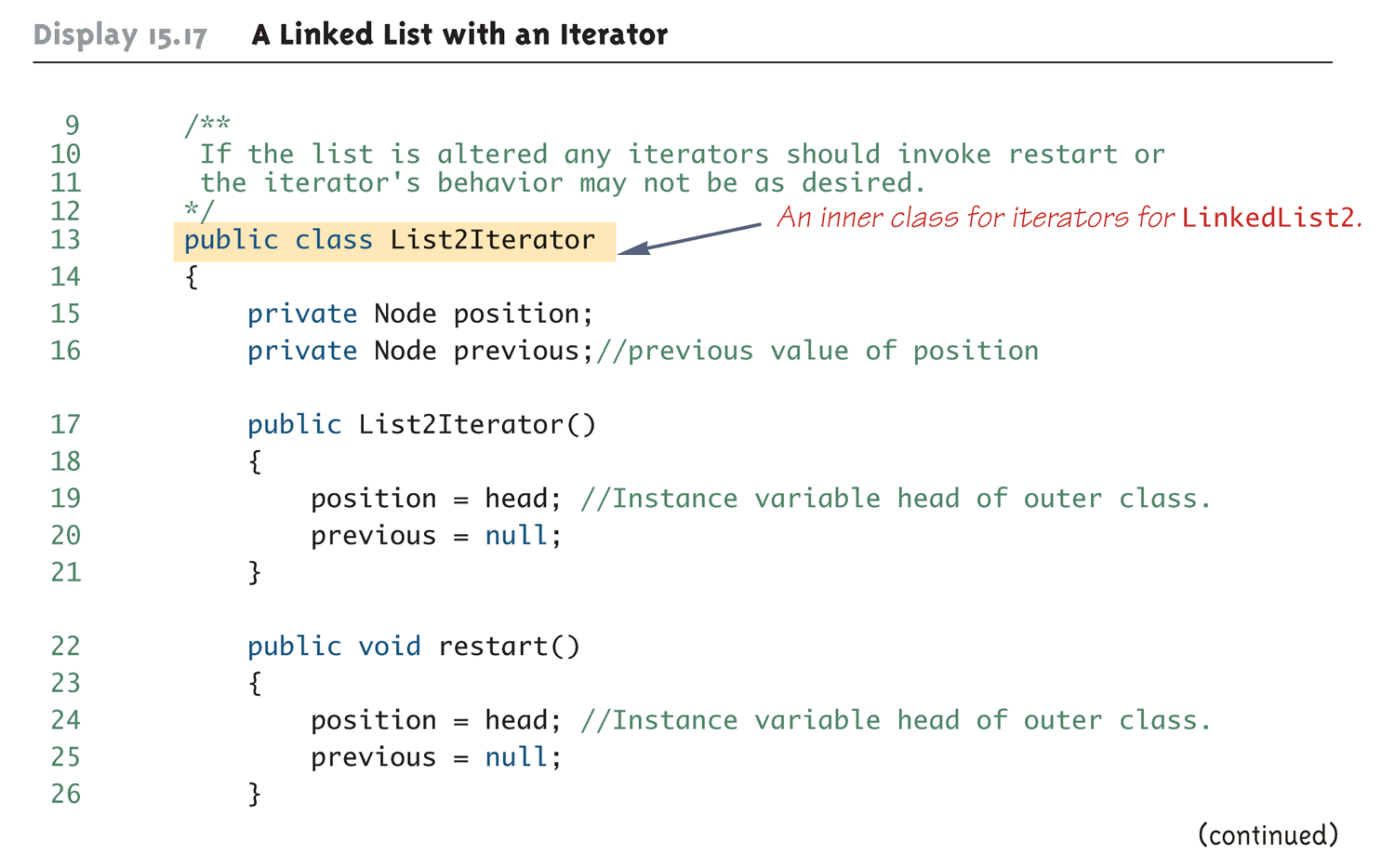
Display 15.17 A Linked List with an Iterator

```
import java.util.NoSuchElementException;

public class LinkedList2
{
    private class Node
    {
        private String item;
        private Node link;

        <The rest of the definition of the Node inner class is given in Display 15.7.>

    }//End of Node inner class
```

This is the same as the class in Displays 15.7 and 15.11 except that the `List2Iterator` *inner class and the* `iterator()` *method have been added.*

(continued)

A Linked List with an Iterator (Part 2 of 6)



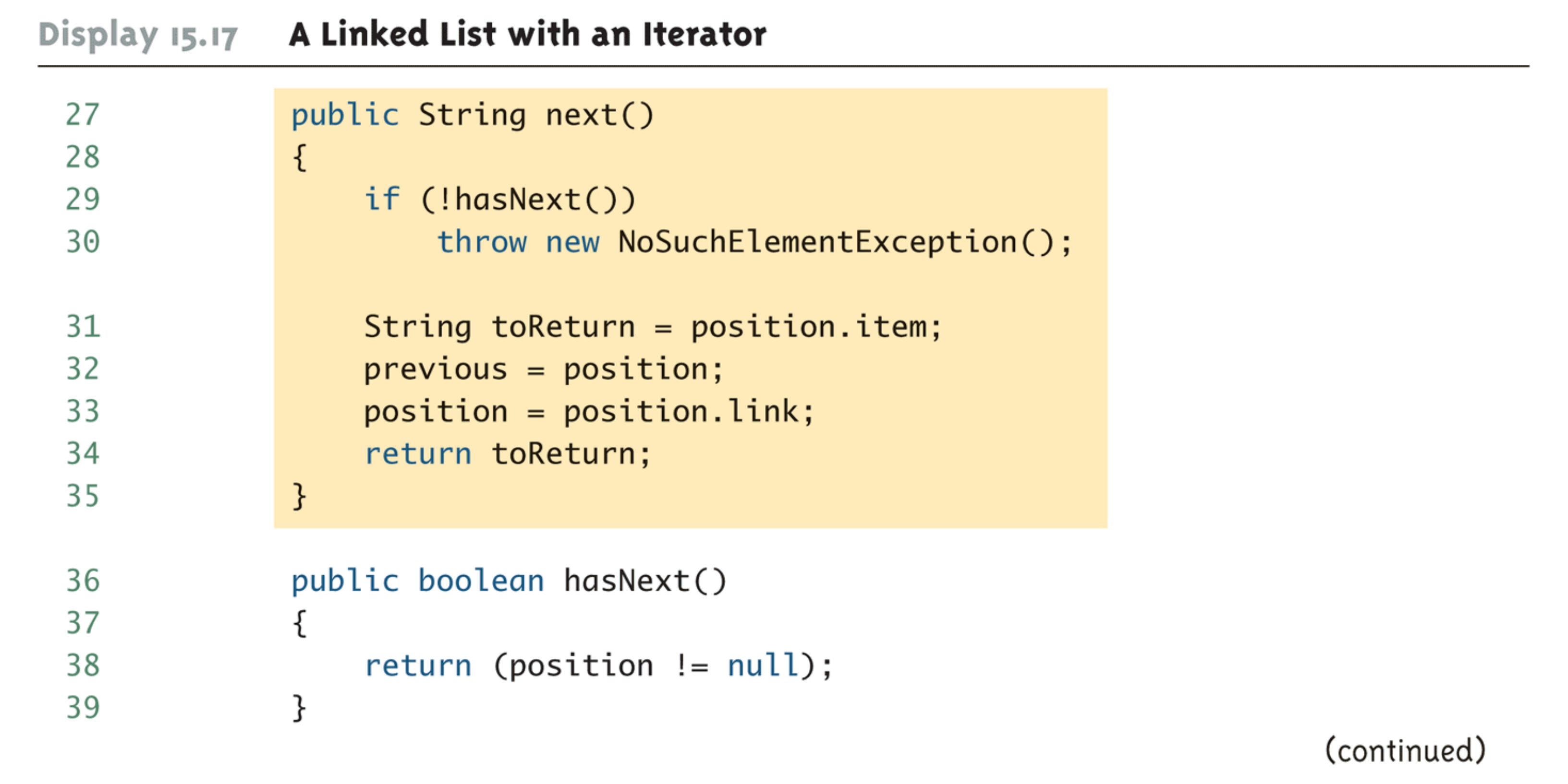
Display 15.17 **A Linked List with an Iterator**

```
    /**
     If the list is altered any iterators should invoke restart or
     the iterator's behavior may not be as desired.
    */
    public class List2Iterator
    {
        private Node position;
        private Node previous;//previous value of position

        public List2Iterator()
        {
            position = head; //Instance variable head of outer class.
            previous = null;
        }

        public void restart()
        {
            position = head; //Instance variable head of outer class.
            previous = null;
        }
```

An inner class for iterators for `LinkedList2`.

(continued)

A Linked List with an Iterator (Part 3 of 6)

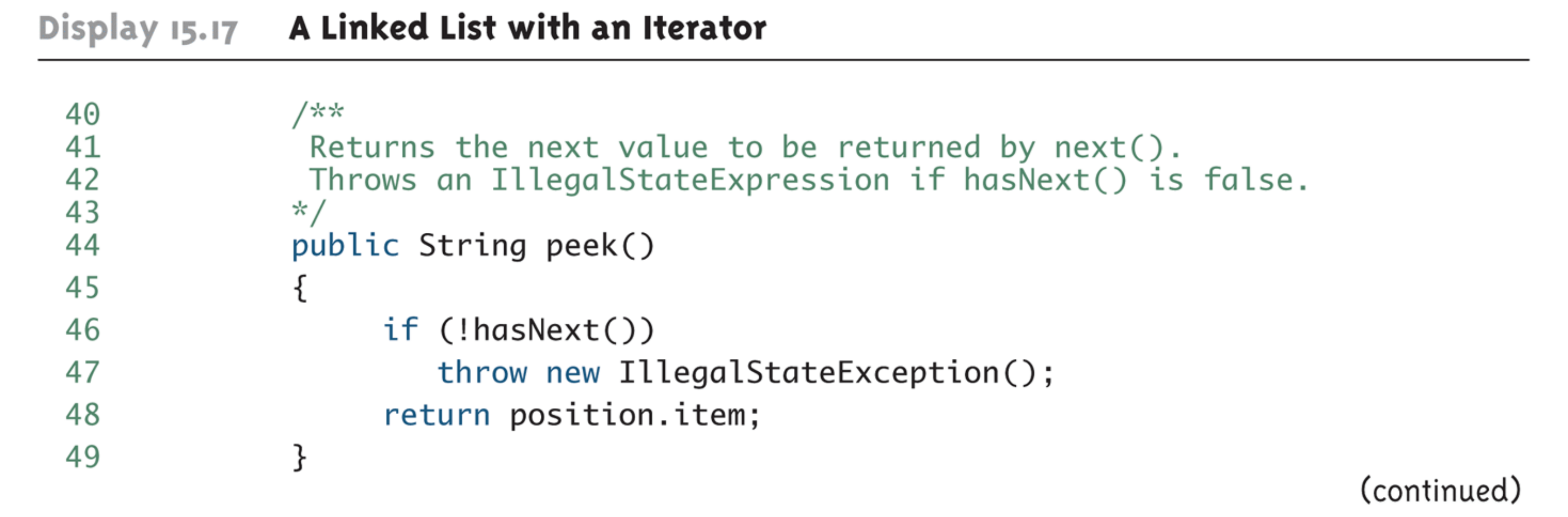


Display 15.17 **A Linked List with an Iterator**

```
public String next()
{
    if (!hasNext())
        throw new NoSuchElementException();

    String toReturn = position.item;
    previous = position;
    position = position.link;
    return toReturn;
}

public boolean hasNext()
{
    return (position != null);
}
```

(continued)

A Linked List with an Iterator (Part 4 of 6)

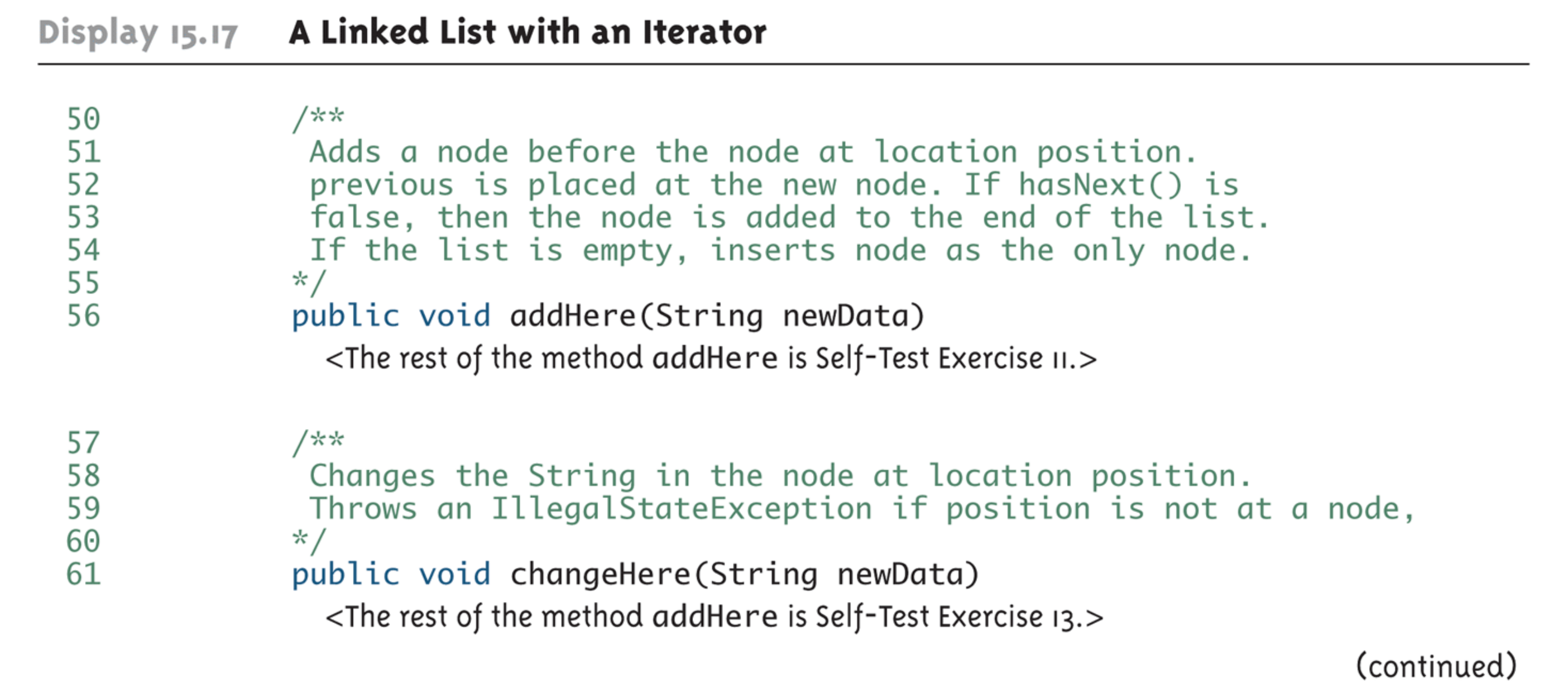


Display 15.17 A Linked List with an Iterator

```
        /**
         Returns the next value to be returned by next().
         Throws an IllegalStateExpression if hasNext() is false.
        */
        public String peek()
        {
            if (!hasNext())
               throw new IllegalStateException();
            return position.item;
        }
```

(continued)

A Linked List with an Iterator (Part 5 of 6)

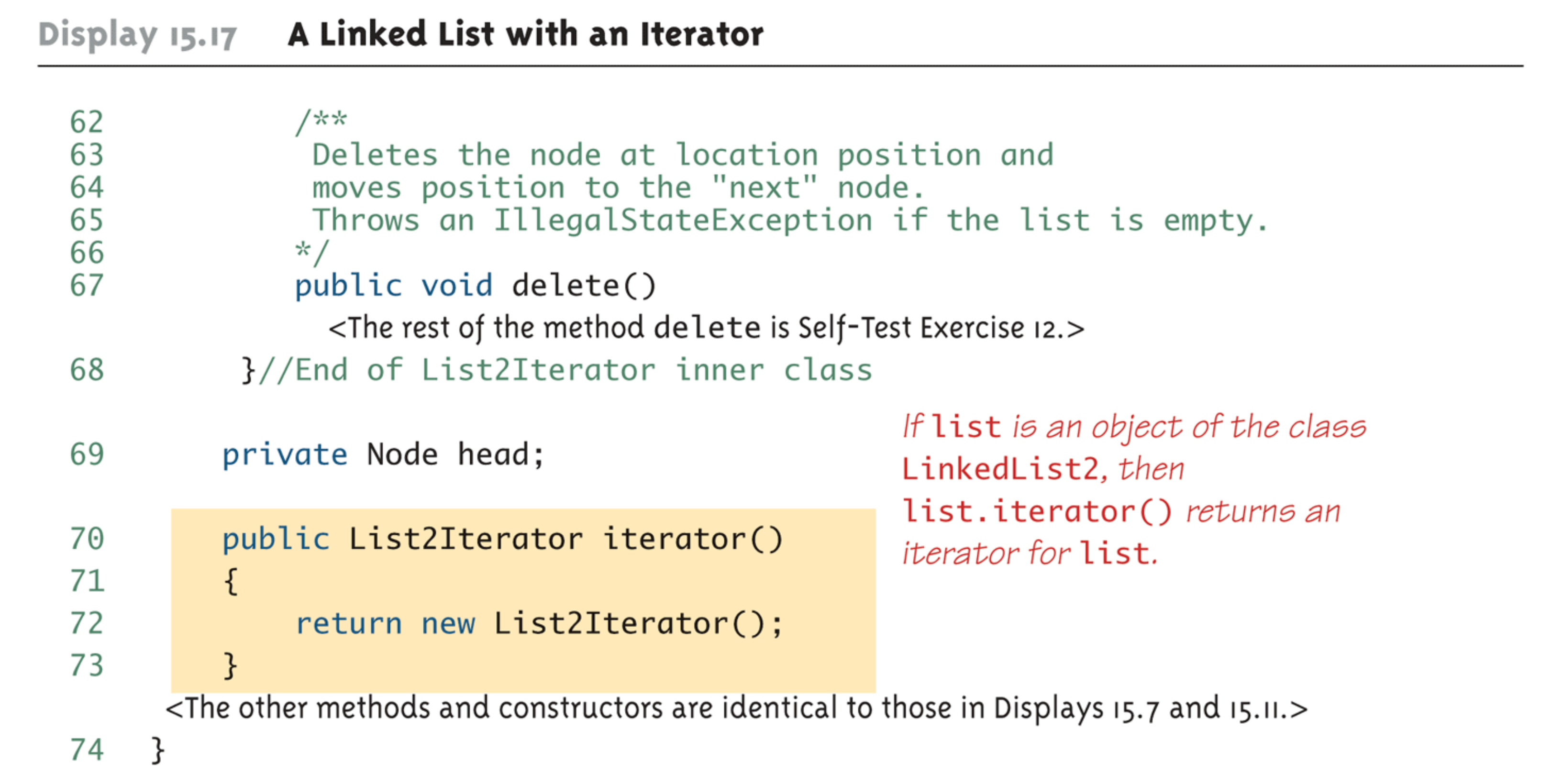


Display 15.17 **A Linked List with an Iterator**

```
    /**
     Adds a node before the node at location position.
     previous is placed at the new node. If hasNext() is
     false, then the node is added to the end of the list.
     If the list is empty, inserts node as the only node.
    */
    public void addHere(String newData)
       <The rest of the method addHere is Self-Test Exercise 11.>

    /**
     Changes the String in the node at location position.
     Throws an IllegalStateException if position is not at a node,
    */
    public void changeHere(String newData)
       <The rest of the method addHere is Self-Test Exercise 13.>
```

(continued)

A Linked List with an Iterator (Part 6 of 6)

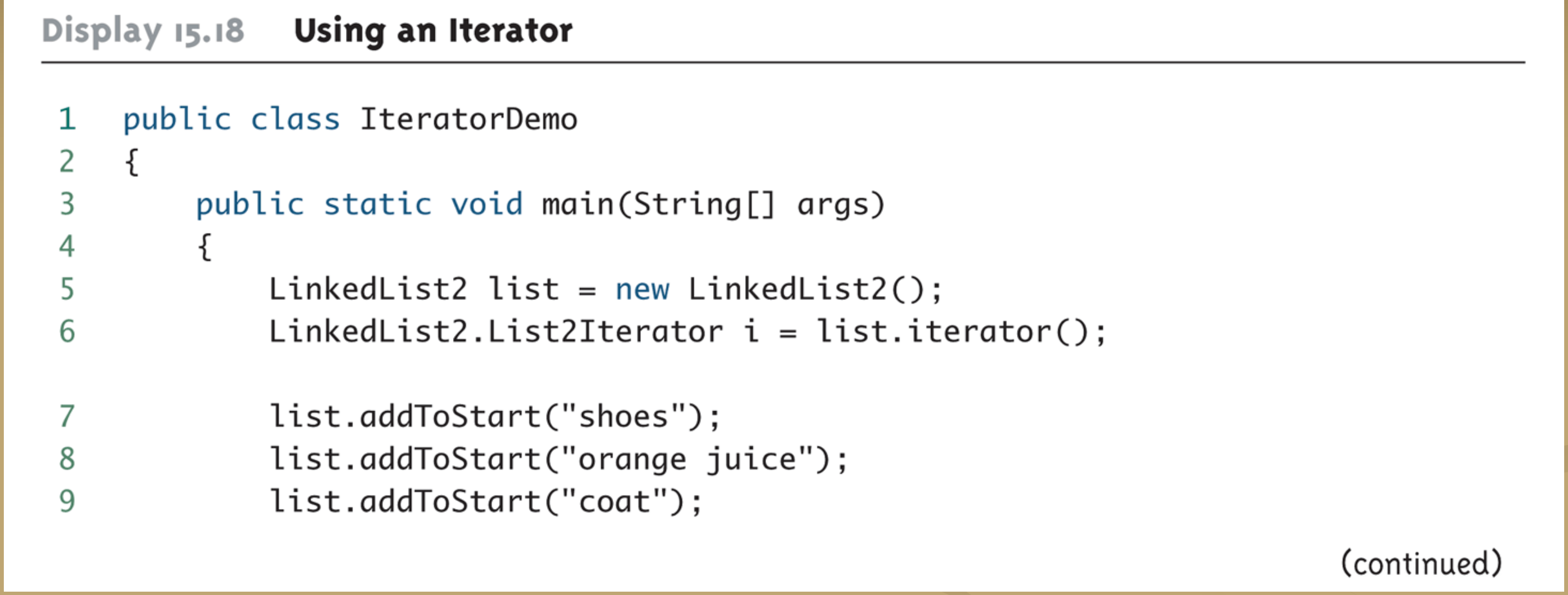


Display 15.17 **A Linked List with an Iterator**

```
        /**
         Deletes the node at location position and
         moves position to the "next" node.
         Throws an IllegalStateException if the list is empty.
        */
        public void delete()
          <The rest of the method delete is Self-Test Exercise 12.>
     }//End of List2Iterator inner class

    private Node head;

    public List2Iterator iterator()
    {
        return new List2Iterator();
    }
 <The other methods and constructors are identical to those in Displays 15.7 and 15.11.>
}
```

If `list` *is an object of the class* `LinkedList2`*, then* `list.iterator()` *returns an iterator for* `list`*.*

Using an Iterator (Part 1 of 6)

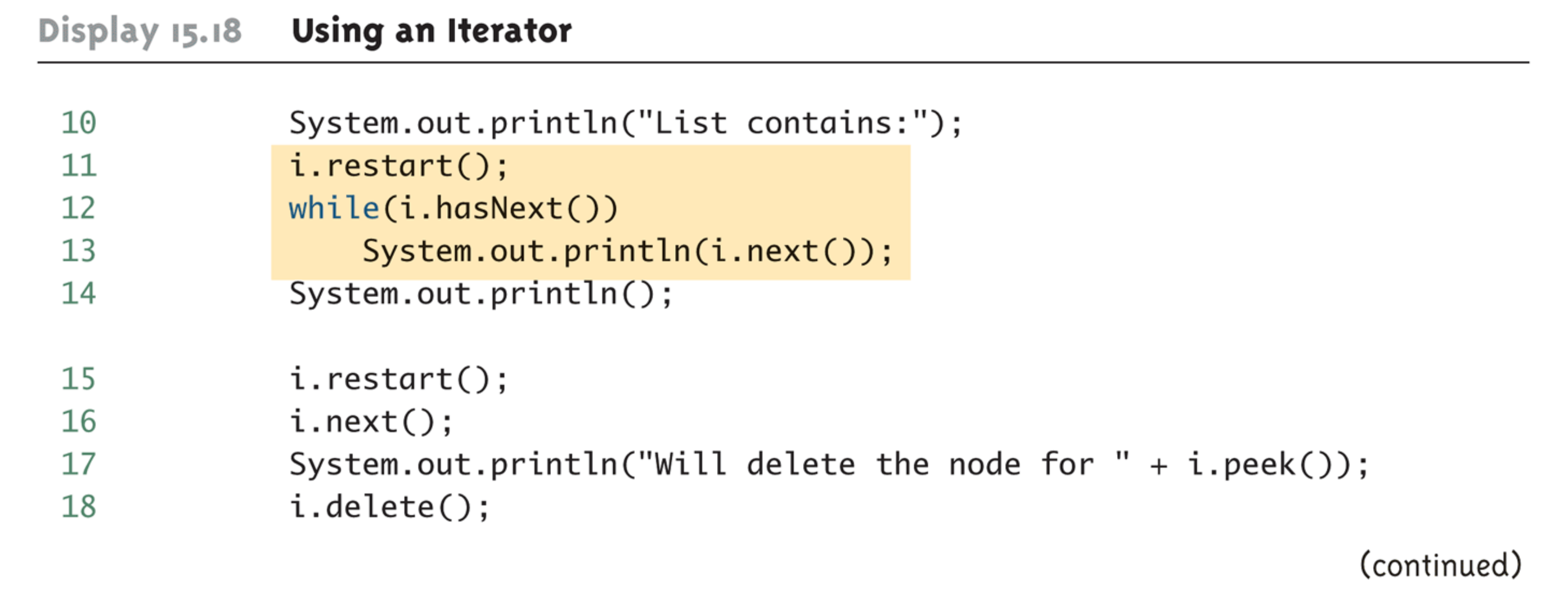


Display 15.18 **Using an Iterator**

```
public class IteratorDemo
{
    public static void main(String[] args)
    {
        LinkedList2 list = new LinkedList2();
        LinkedList2.List2Iterator i = list.iterator();

        list.addToStart("shoes");
        list.addToStart("orange juice");
        list.addToStart("coat");
```

(continued)

Using an Iterator (Part 2 of 6)

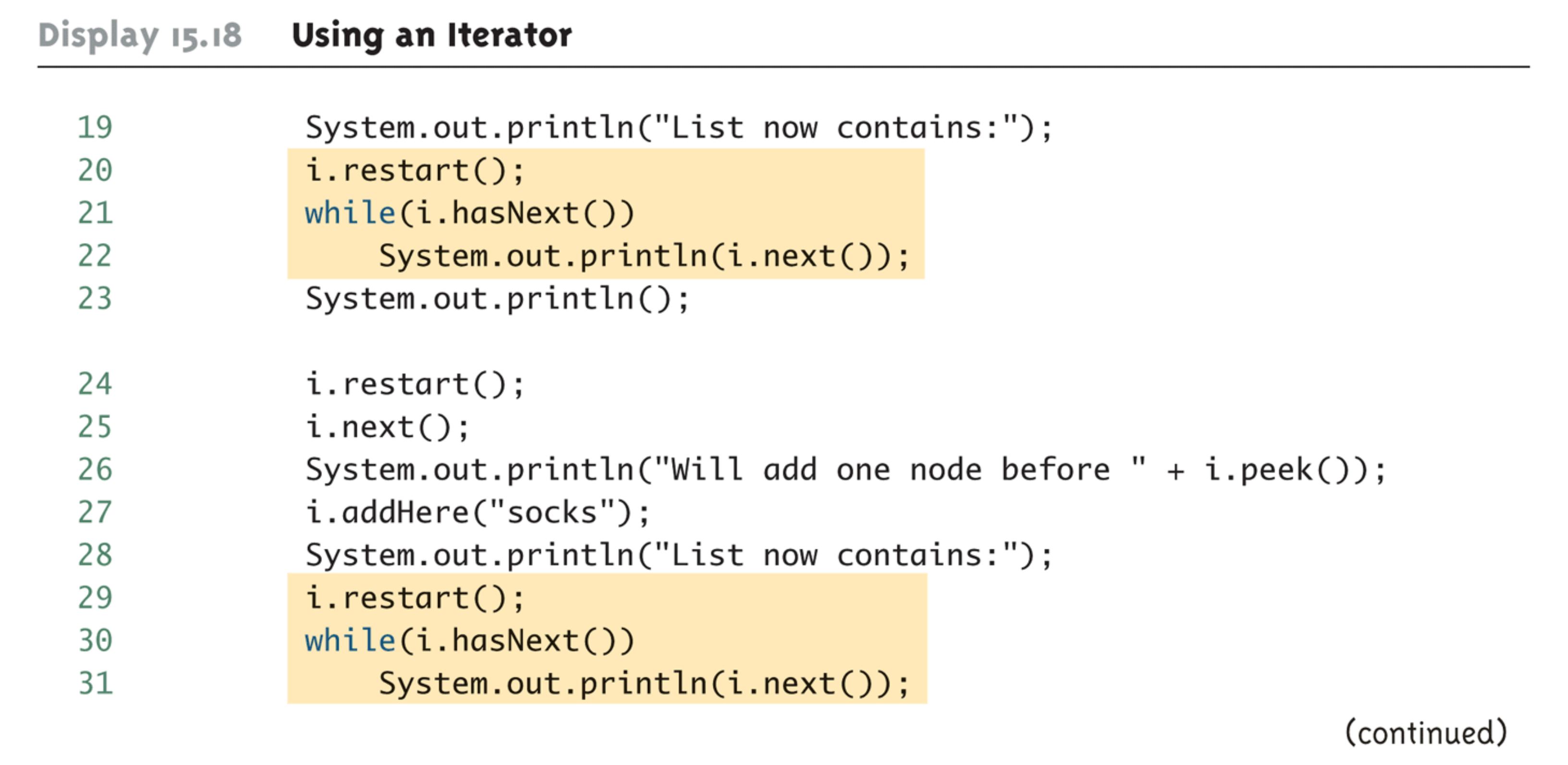


Display 15.18 Using an Iterator

```
System.out.println("List contains:");
i.restart();
while(i.hasNext())
    System.out.println(i.next());
System.out.println();

i.restart();
i.next();
System.out.println("Will delete the node for " + i.peek());
i.delete();
```

(continued)

Using an Iterator (Part 3 of 6)

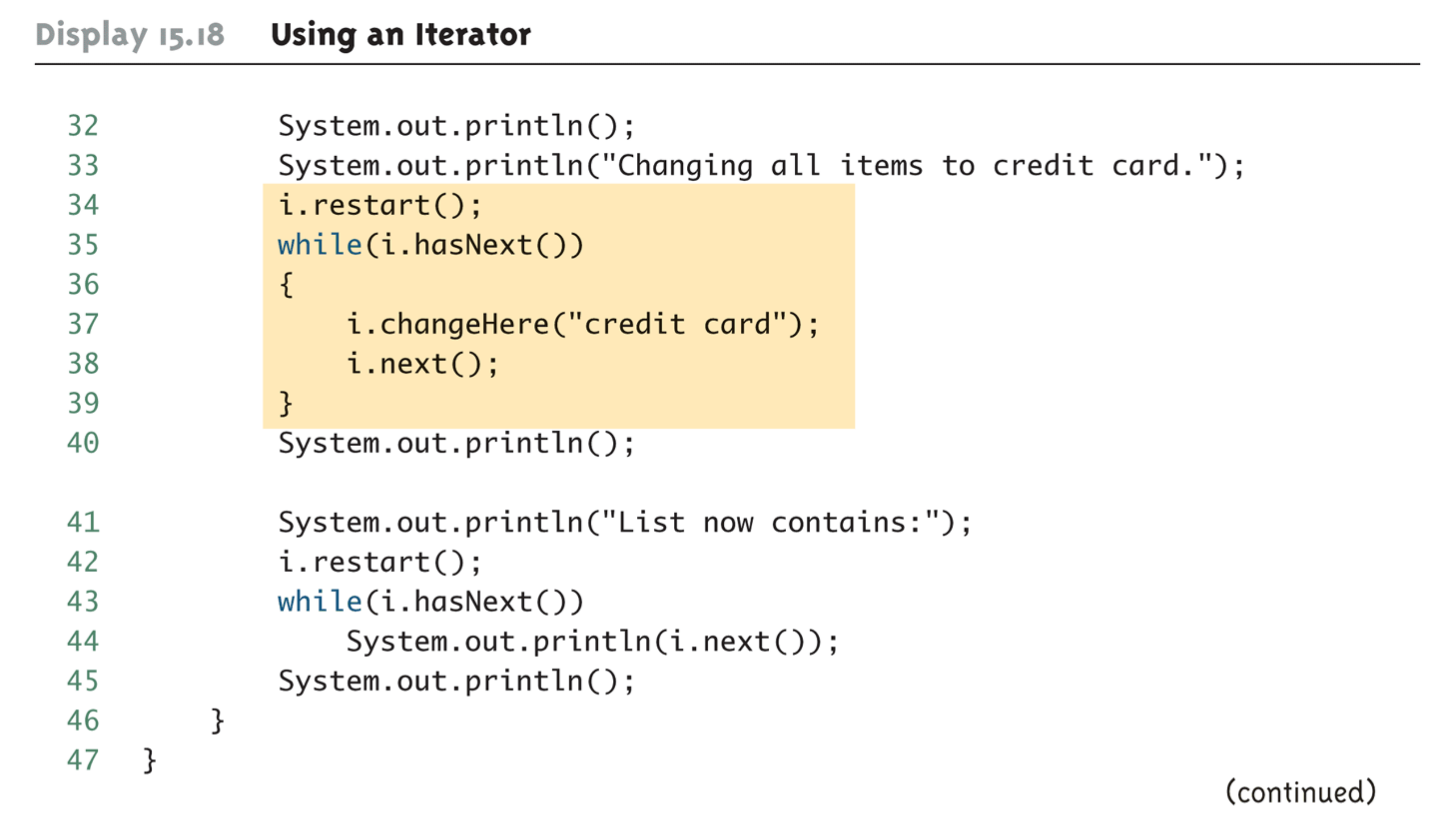


Display 15.18 Using an Iterator

```
System.out.println("List now contains:");
i.restart();
while(i.hasNext())
    System.out.println(i.next());
System.out.println();

i.restart();
i.next();
System.out.println("Will add one node before " + i.peek());
i.addHere("socks");
System.out.println("List now contains:");
i.restart();
while(i.hasNext())
    System.out.println(i.next());
```

(continued)

Using an Iterator (Part 4 of 6)

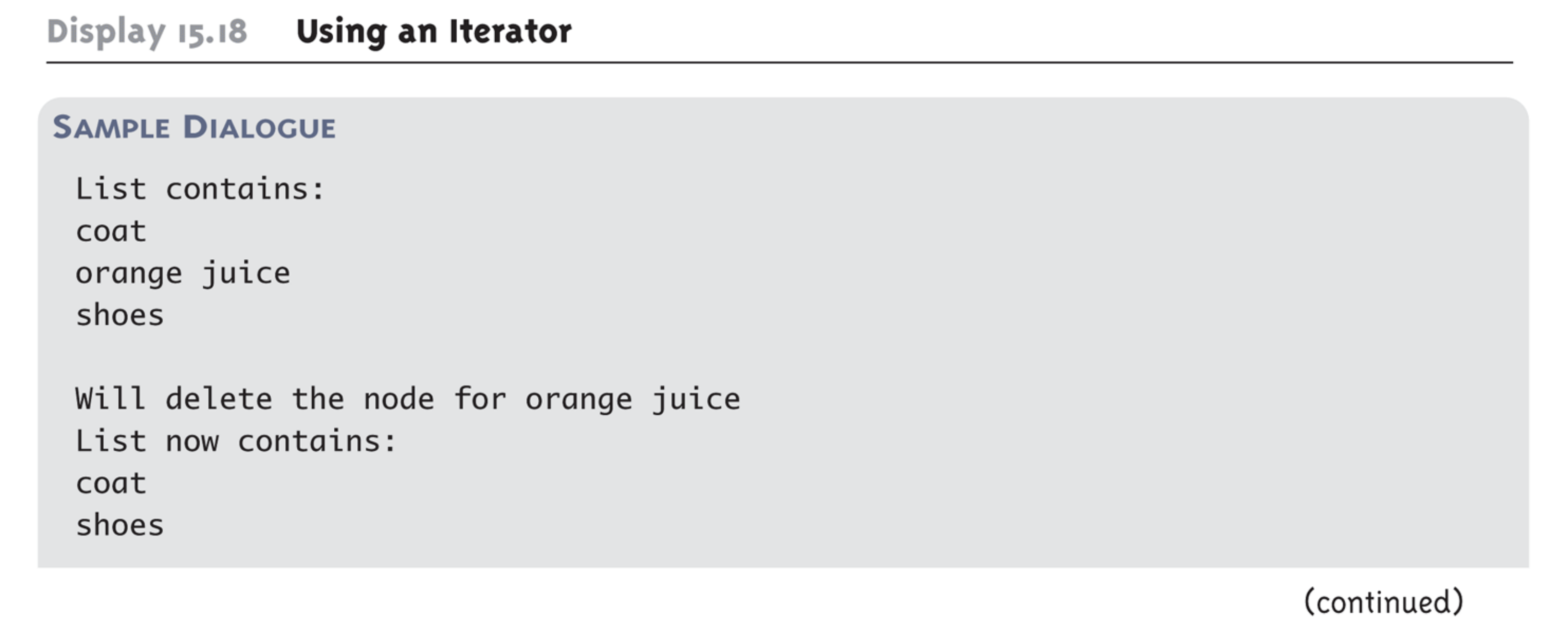


Display 15.18 **Using an Iterator**

```
        System.out.println();
        System.out.println("Changing all items to credit card.");
        i.restart();
        while(i.hasNext())
        {
            i.changeHere("credit card");
            i.next();
        }
        System.out.println();

        System.out.println("List now contains:");
        i.restart();
        while(i.hasNext())
            System.out.println(i.next());
        System.out.println();
    }
}
```

(continued)

Using an Iterator (Part 5 of 6)



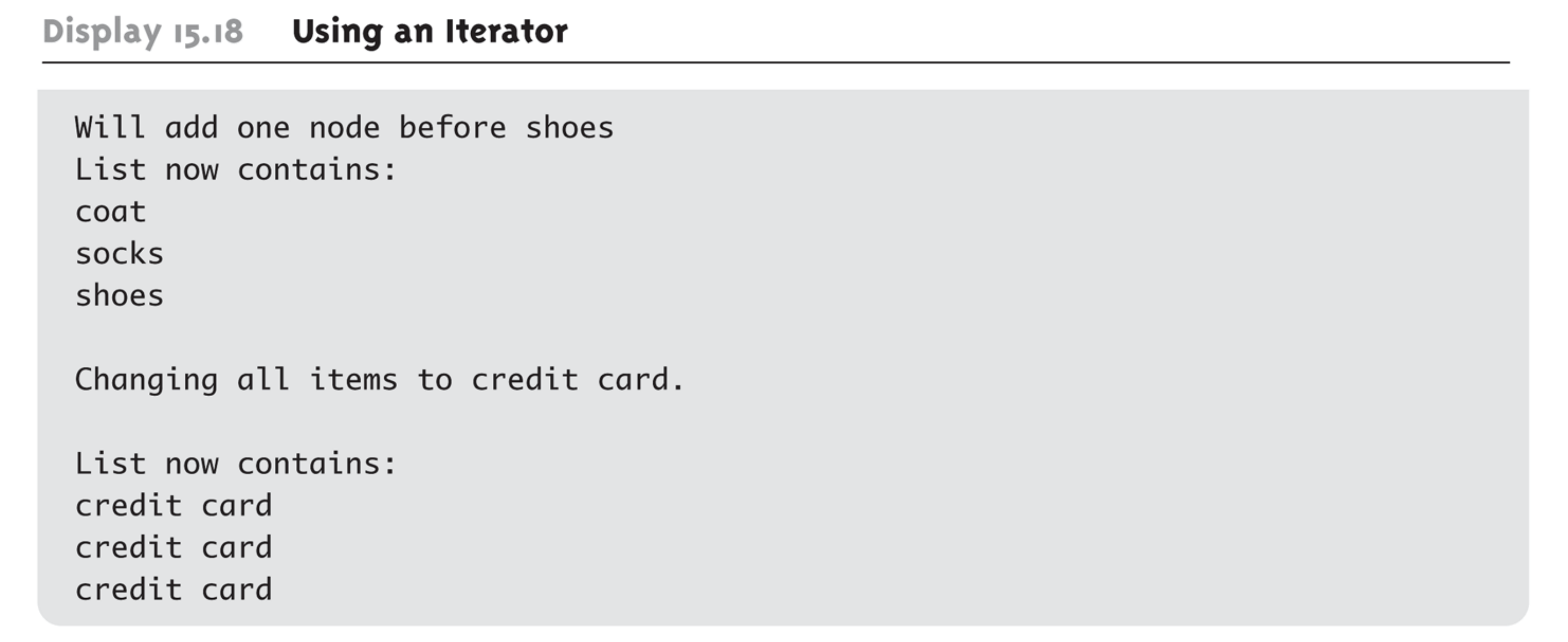
Display 15.18 **Using an Iterator**

Sample Dialogue

```
List contains:
coat
orange juice
shoes

Will delete the node for orange juice
List now contains:
coat
shoes
```

(continued)

Using an Iterator (Part 6 of 6)



Display 15.18 **Using an Iterator**

```
Will add one node before shoes
List now contains:
coat
socks
shoes

Changing all items to credit card.

List now contains:
credit card
credit card
credit card
```

The Java Iterator Interface

- Java has an interface named **`Iterator`** that specifies how Java would like an iterator to behave
  - Although the iterators examined so far do not satisfy this interface, they could be easily redefined to do so