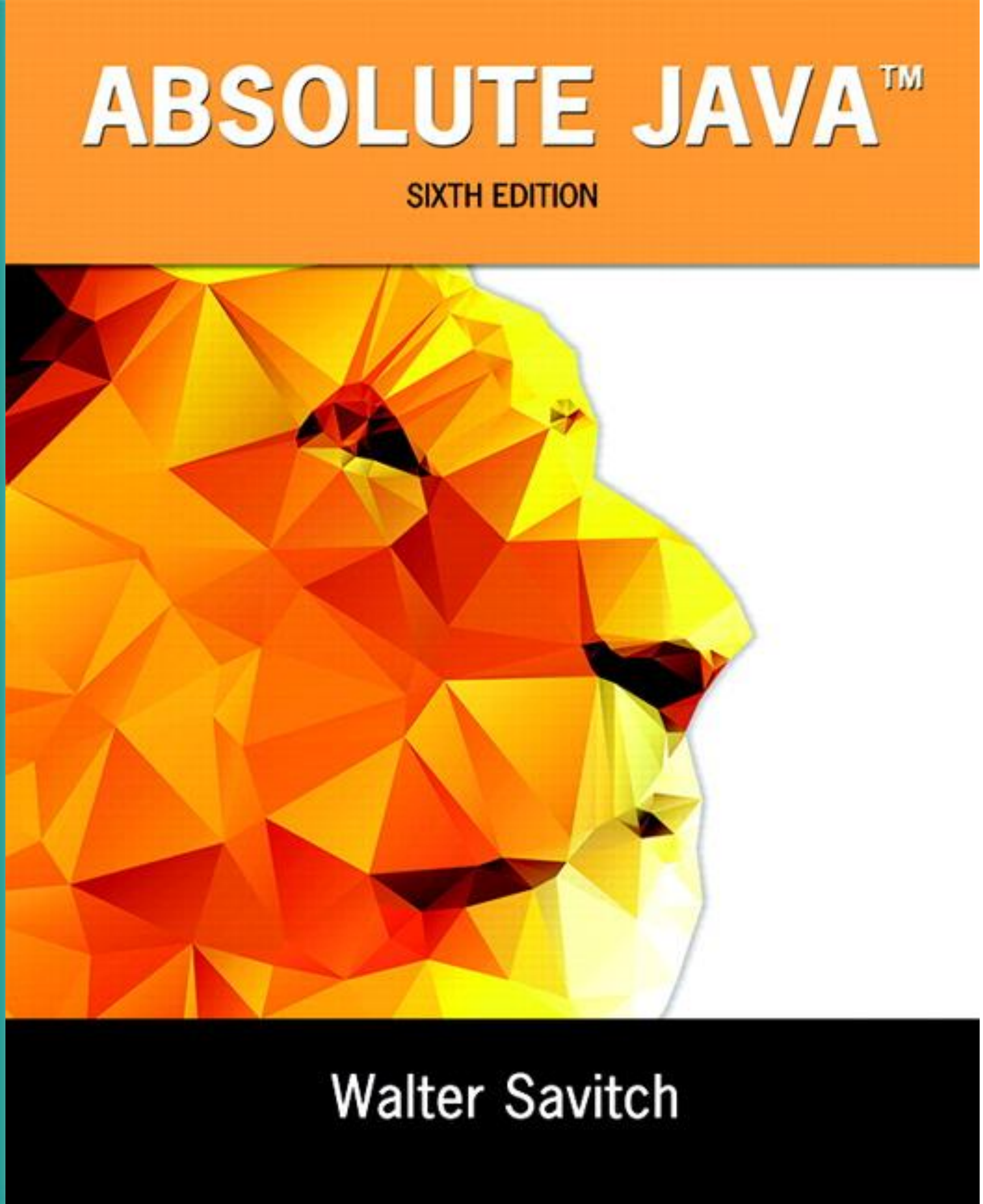


Chapter 15

Linked Data Structures

Introduction to Linked Data Structures

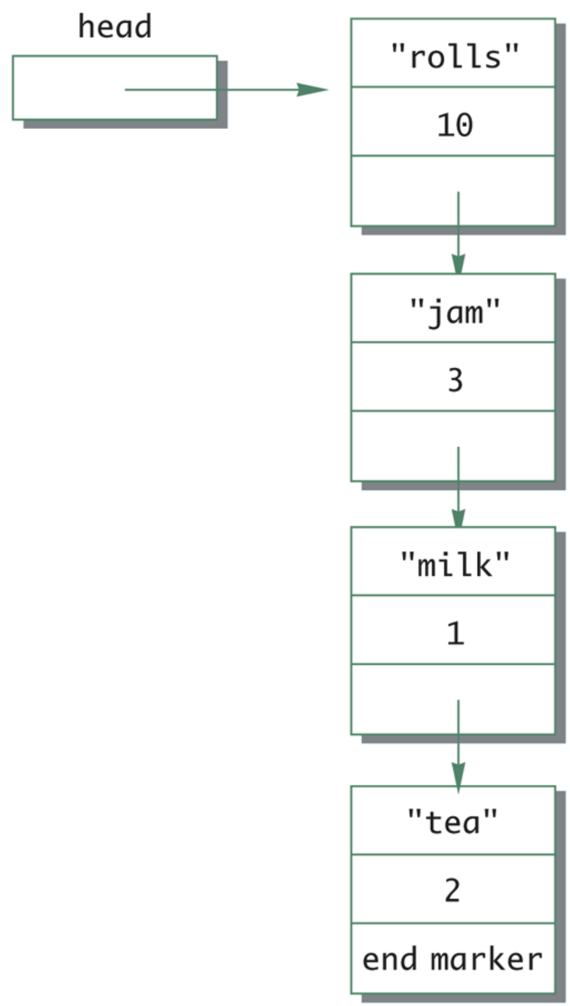
- A *linked data structure* consists of capsules of data known as *nodes* that are connected via *links*
  - Links can be viewed as arrows and thought of as one way passages from one node to another
- In Java, nodes are realized as objects of a node class
- The data in a node is stored via instance variables
- 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"

A Node Class (Part 1 of 3)

Display 15.2 A Node Class

```
public class Node1
{
    private String item;
    private int count;
    private Node1 link;

    public Node1()
    {
        link = null;
        item = null;
        count = 0;
    }

    public Node1(String newItem, int newCount, Node1 linkValue)
    {
        setData(newItem, newCount);
        link = linkValue;
    }
```

A node contains a reference to another node. That reference is the link to the next node.

We will define a number of node classes so we numbered the names, as in Node1.

(continued)

A Node Class (Part 2 of 3)

Display 15.2 A Node Class

```
public void setData(String newItem, int newCount)
{
    item = newItem;
    count = newCount;
}

public void setLink(Node1 newLink)
{
    link = newLink;
}
```

We will give a better definition of a node class later in this chapter.

(continued)

A Node Class (Part 3 of 3)

Display 15.2 A Node Class

```
    public String getItem()
    {
        return item;
    }

    public int getCount()
    {
        return count;
    }

    public Node1 getLink()
    {
        return link;
    }
}
```

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

A Linked List Class (Part 1 of 6)

Display 15.3 A Linked List Class

```
public class LinkedList1
{
    private Node1 head;

    public LinkedList1()
    {
        head = null;
    }

    /**
     Adds a node at the start of the list with the specified data.
     The added node will be the first node in the list.
    */
    public void addToStart(String itemName, int itemCount)
    {
        head = new Node1(itemName, itemCount, head);
    }
```

We will define a better linked list class later in this chapter.

(continued)

A Linked List Class (Part 2 of 6)

Display 15.3 A Linked List Class

```java
    /**
     Removes the head node and returns true if the list contained at least
     one node. Returns false if the list was empty.
    */
    public boolean deleteHeadNode()
    {
        if (head != null)
        {
            head = head.getLink();
            return true;
        }
        else
            return false;
    }
```

(continued)

A Linked List Class (Part 3 of 6)

Display 15.3 A Linked List Class

```
    /**
     Returns the number of nodes in the list.
    */
    public int size()
    {
        int count = 0;
        Node1 position = head;

```

(continued)

A Linked List Class (Part 4 of 6)

Display 15.3 A Linked List Class

```
        while (position != null)
        {
            count++;
            position = position.getLink();
        }
        return count;
    }

    public boolean contains(String item)
    {
        return (find(item) != null);
    }
```

The last node is indicated by the `link` *field being equal to* `null`.

(continued)

A Linked List Class (Part 5 of 6)

Display 15.3 A Linked List Class

```
    /**
     Finds the first node containing the target item, and returns a
     reference to that node. If target is not in the list, null is returned.
    */
    private Node1 find(String target)
    {
        Node1 position = head;
        String itemAtPosition;
        while (position != null)
        {
```

(continued)

A Linked List Class (Part 6 of 6)

Display 15.3 A Linked List Class

```
            itemAtPosition = position.getItem();
            if (itemAtPosition.equals(target))
                return position;
            position = position.getLink();
        }
        return null; //target was not found
    }

    public void outputList()
    {
        Node1 position = head;
        while (position != null)
        {
            System.out.println(position.getItem() + " "
                                + position.getCount());
            position = position.getLink();
        }
    }
}
```

This is the way you traverse an entire linked list.

Indicating the End of a Linked List

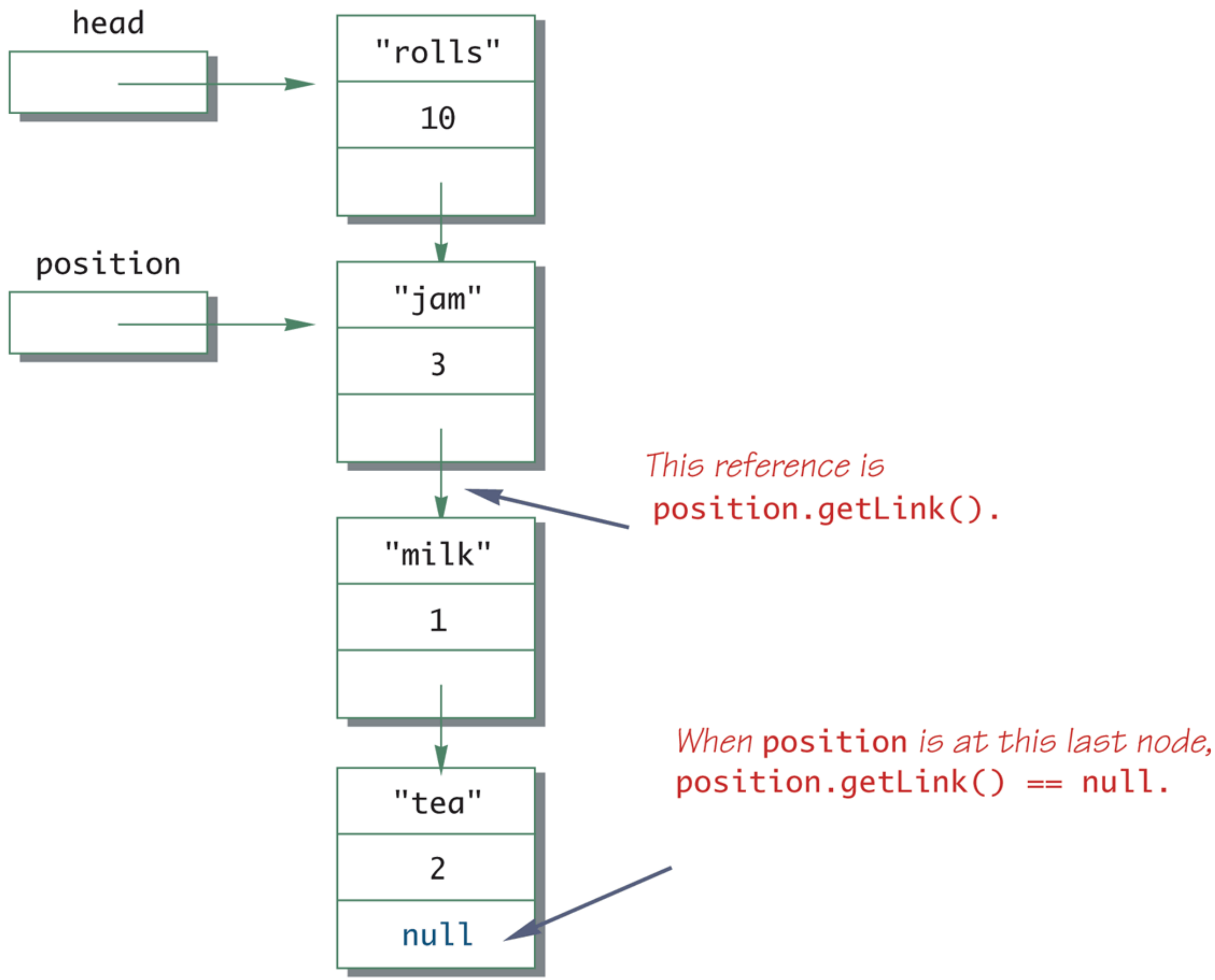
- The last node in a linked list should have its link instance variable set to `null`
  - That way the code can test whether or not a node is the last node
  - Note: This is tested using `==`, not the `equals` method

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

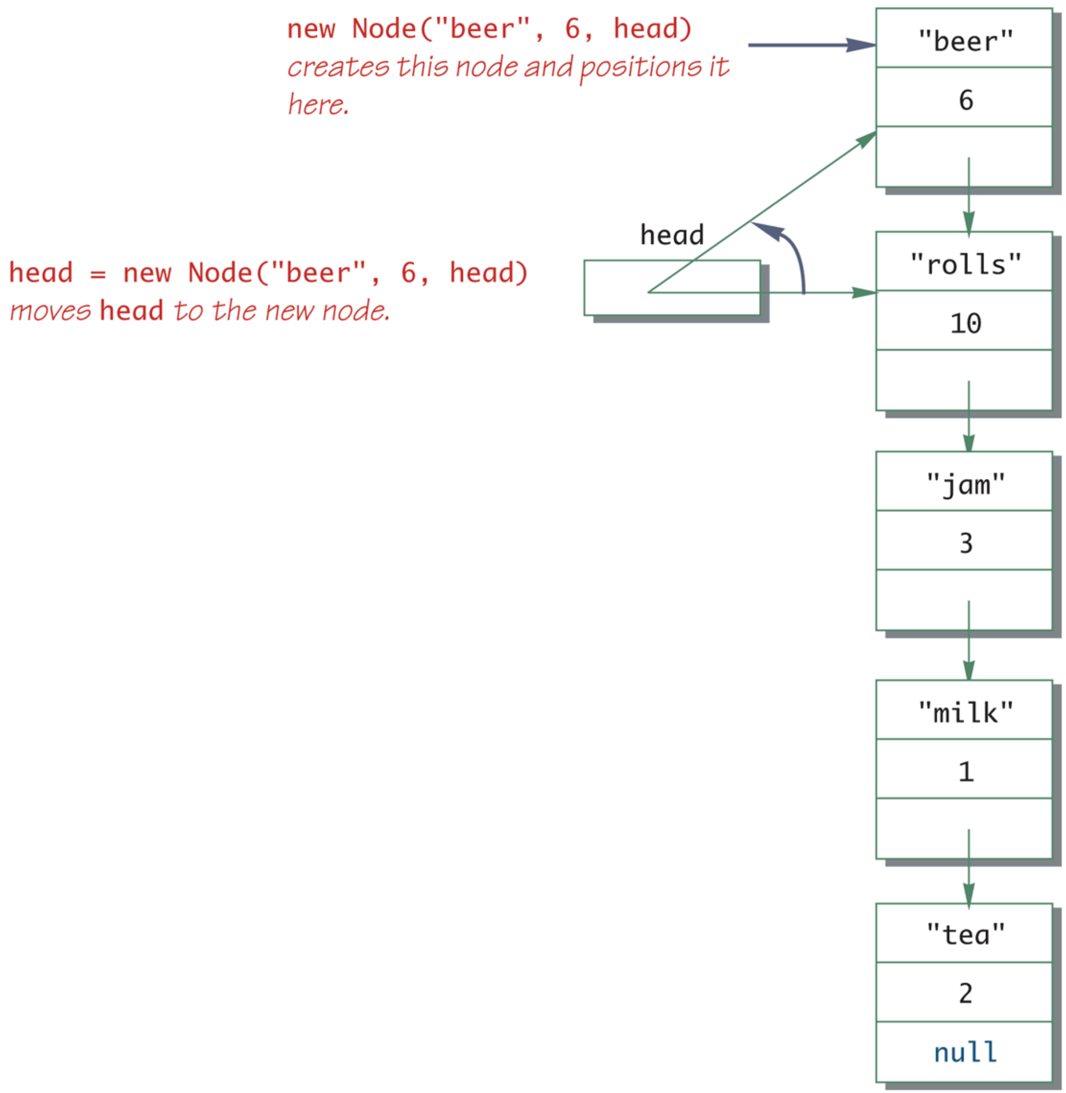


Adding a Node to a Linked List

- The method **`add`** adds a node to the start of the linked list
  - This makes the new node become the first node on the list
- The variable **`head`** gives the location of the current first node of the list
  - Therefore, when the new node is created, its **`link`** field is set equal to **`head`**
  - Then **`head`** is set equal to the new node

Adding a Node at the Start

Display 15.5 Adding a Node at the Start



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*

A Linked List Demonstration (Part 1 of 3)

Display 15.6 A Linked List Demonstration

```
public class LinkedList1Demo
{
    public static void main(String[] args)
    {
        LinkedList1 list = new LinkedList1();
        list.addToStart("Apples", 1);
        list.addToStart("Bananas", 2);
        list.addToStart("Cantaloupe", 3);
        System.out.println("List has " + list.size()
                                + " nodes.");
        list.outputList();
        if (list.contains("Cantaloupe"))
            System.out.println("Cantaloupe is on list.");
```

Cantaloupe *is now in the head node.*

(continued)

A Linked List Demonstration (Part 2 of 3)

Display 15.6 A Linked List Demonstration

```
        else
            System.out.println("Cantaloupe is NOT on list.");

        list.deleteHeadNode();

        if (list.contains("Cantaloupe"))
            System.out.println("Cantaloupe is on list.");
        else
            System.out.println("Cantaloupe is NOT on list.");

        while (list.deleteHeadNode())
            ; //Empty loop body

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

Removes the head node.

Empties the list. There is no loop body because the method **deleteHeadNode** *both performs an action on the list and returns a Boolean value.*

(continued)

A Linked List Demonstration (Part 3 of 3)

Display 15.6 A Linked List Demonstration

SAMPLE DIALOGUE

```
List has 3 entries.
Cantaloupe 3
Bananas 2
Apples 1
Cantaloupe is on list.
Cantaloupe is NOT on list.
Start of list:
End of list.
```

Node Inner Classes

- Note that the linked list class discussed so far is dependent on an external node class
- A linked list or similar data structure can be 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

Pitfall: Privacy Leaks

- The original node and linked list classes examined so far have a dangerous flaw
  - The node class accessor method returns a reference to a node
  - Recall that if a method returns a reference to an instance variable of a mutable class type, 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

A Linked List Class with a Node Inner Class (Part 1 of 6)

Display 15.7 A Linked List Class with a Node Inner Class

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

        public Node()
        {
             item = null;
             link = null;
        }

        public Node(String newItem, Node linkValue)
        {
            item = newItem;
            link = linkValue;
        }
     }//End of Node inner class
```

It makes no difference whether we make the instance variables of **Node** *public or private.*

An inner class for the node class

(continued)

A Linked List Class with a Node Inner Class (Part 2 of 6)

Display 15.7 A Linked List Class with a Node Inner Class

```
    private Node head;

    public LinkedList2()
    {
        head = null;
    }
    /**
     Adds a node at the start of the list with the specified data.
     The added node will be the first node in the list.
    */
    public void addToStart(String itemName)
    {
        head = new Node(itemName, head);
    }

    /**
     Removes the head node and returns true if the list contained at least
     one node. Returns false if the list was empty.
    */
```

We have simplified this class and the previous linked list class to keep them relatively short. Among other things, these classes should have a copy constructor, an **equals** *method, and a* **clone** *method. Our next linked list example includes these items.*

(continued)

A Linked List Class with a Node Inner Class (Part 3 of 6)

Display 15.7 A Linked List Class with a Node Inner Class

```
    public boolean deleteHeadNode()
    {
        if (head != null)
        {
            head = head.link;
            return true;
        }
        else
            return false;
    }

    /**
     Returns the number of nodes in the list.
    */
    public int size()
    {
        int count = 0;
        Node position = head;
```

(continued)

A Linked List Class with a Node Inner Class (Part 4 of 6)

Display 15.7 A Linked List Class with a Node Inner Class

```
        while (position != null)
        {
            count++;
            position = position.link;
        }
        return count;
    }

    public boolean contains(String item)
    {
        return (find(item) != null);
    }
```

Note that the outer class has direct access to the inner class's instance variables, such as `link`.

(continued)

A Linked List Class with a Node Inner Class (Part 5 of 6)

Display 15.7 A Linked List Class with a Node Inner Class

```java
    /**
     Finds the first node containing the target item, and returns a
     reference to that node. If target is not in the list, null is returned.
    */
    private Node find(String target)
    {
        Node position = head;
        String itemAtPosition;
        while (position != null)
        {
            itemAtPosition = position.item;
            if (itemAtPosition.equals(target))
                return position;
            position = position.link;
        }
        return null; //target was not found
    }
```

(continued)

A Linked List Class with a Node Inner Class (Part 6 of 6)

Display 15.7 A Linked List Class with a Node Inner Class

```java
    public void outputList()
    {
        Node position = head;
        while (position != null)
        {
            System.out.println(position.item );
            position = position.link;
        }
    }

    public boolean isEmpty()
    {
        return (head == null);
    }

    public void clear()
    {
        head = null;
    }
}
```

Running Times

- How fast is program?
  - "Seconds"?
  - Consider: large input? .. small input?
- Produce "table"
  - Based on input size
  - Table called "function" in math
    - With arguments and return values!
  - Argument is input size:
    T(10), T(10,000), …
- Function T is called "running time"

Table for Running Time Function: **Display 15.31** Some Values of a Running Time Function

Some Values of a Running Time Function

INPUT SIZE	RUNNING TIME
10 numbers	2 seconds
100 numbers	2.1 seconds
1,000 numbers	10 seconds
10,000 numbers	2.5 minutes

Consider Sorting Program

- Faster on smaller input set?
  - Perhaps
  - Might depend on "state" of set
    - "Mostly" sorted already?
- Consider worst-case running time
  - T(N) is time taken by "hardest" list
    - List that takes longest to sort

Counting Operations

- T(N) given by formula, such as:
  T(N) = 5N + 5
  - "On inputs of size N program runs for 5N + 5 time units"
- Must be "computer-independent"
  - Doesn’t matter how "fast" computers are
  - Can’t count "time"
  - Instead count "operations"

Counting Operations Example

- ```
  int i = 0;
  Boolean found = false;
  while (( i < N) && !found)
          if (a[I] == target)
                  found = true;
          else
                  i++;
  ```
- 5 operations per loop iteration:
  <, &&, !, [ ], ==, ++
- After N iterations, final three: <, &&, !
- So: 6N+5 operations when target not found
  ```

Big-O Notation

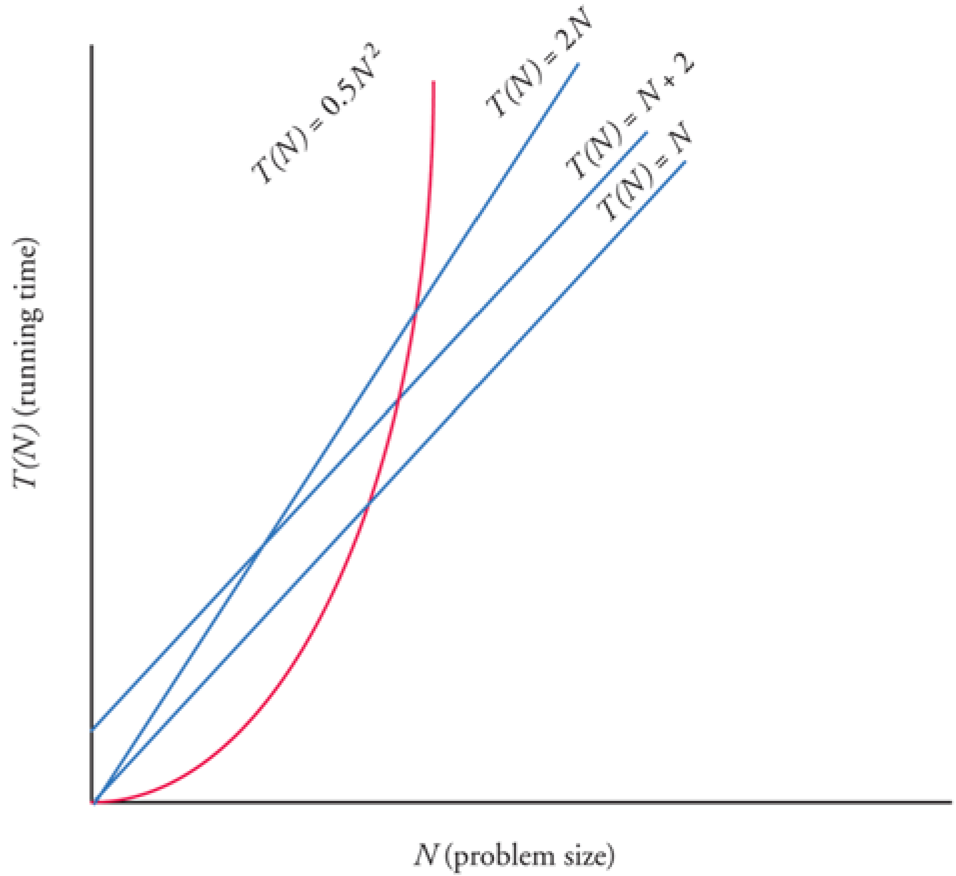
- Recall: 6N+5 operations in "worst-case"
- Expressed in "Big-O" notation
  - Some constant "c" factor where c(6N+5) is actual running time
    - c different on different systems
  - We say code runs in time O(6N+5)
  - But typically only consider "highest term"
    - Term with highest exponent
  - O(N) here

Big-O Terminology

- Linear running time:
  - O(N)—directly proportional to input size N
- Quadratic running time:
  - $O(N^2)$
- Logarithmic running time:
  - O(log N)
    - Typically "log base 2"
    - Very fast algorithms!

Display 15.32
Comparison of Running Times

Comparison of Running Times



Efficiency of Linked Lists

- Find method for linked list
  - May have to search entire list
  - On average would expect to search half of the list, or n/2
  - In big-O notation, this is O(n)
- Adding to a linked list
  - When adding to the start we only reassign some references
  - Constant time or O(1)