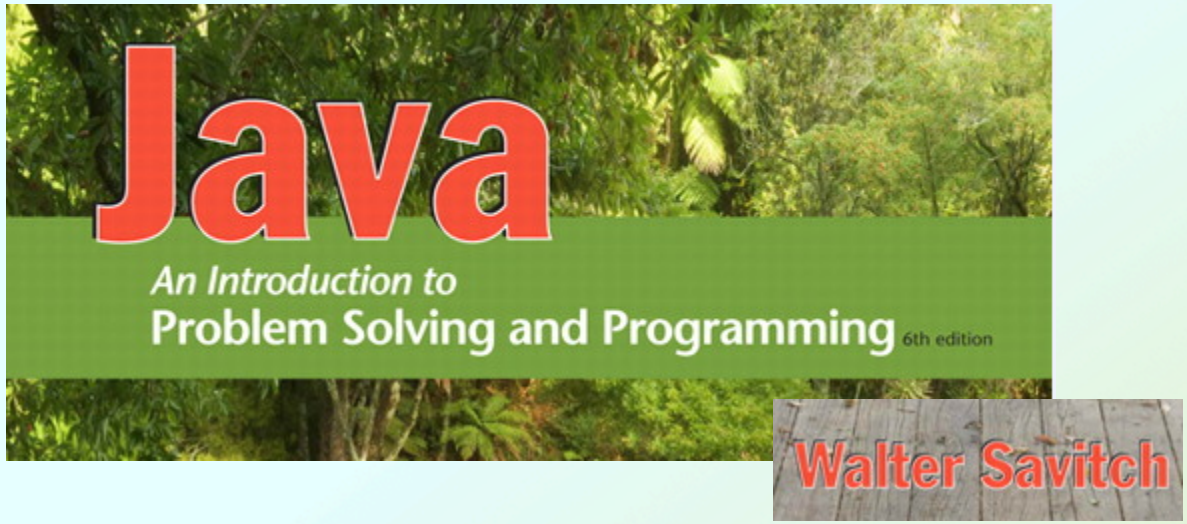



# Dynamic Data Structures and Generics

## Chapter 12

*JAVA: An Introduction to Problem Solving & Programming, 6*th Ed. By Walter Savitch
ISBN 0132162709 

# Objectives

- Define and use an instance of **`ArrayList`**
- Introduction to the Java Collections Framework
- Describe general idea of linked list data structures and implementation
- Manipulate linked lists
- Use inner classes in defining linked data structures
- Describe, create, use iterators
- Define, us classes with generic types

# Array-Based Data Structures: Outline

- The Class **`ArrayList`**
- Creating an Instance of **`ArrayList`**
- Using Methods of **`ArrayList`**
- Programming Example: A To-Do List
- Parameterized Classes and Generic Data Types

# Class `ArrayList`

- Consider limitations of Java arrays
  - Array length is not dynamically changeable
  - Possible to create a new, larger array and copy elements – but this is awkward, contrived
- More elegant solution is use instance of `ArrayList`
  - Length is changeable at run time

# Class `ArrayList`

- Drawbacks of using `ArrayList`
  - Less efficient than using an array
  - Can only store objects
  - Cannot store primitive types
- Implementation
  - Actually does use arrays
  - Expands capacity in manner previously suggested

# Class `ArrayList`

- Class ArrayList is an implementation of an **A**bstract **D**ata **T**ype (ADT) called a *list*
- Elements can be added
  - At end
  - At beginning
  - In between items
- Possible to edit, delete, access, and count entries in the list

# Class `ArrayList`

- Figure 12.1a Methods of class `ArrayList`

| |
|---|
| `public ArrayList<`*Base_Type*`>(int initialCapacity)`<br>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*`>()`<br>Behaves like the previous constructor, but the initial capacity is ten. |
| `public boolean add(`*Base_Type* `newElement)`<br>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)`<br>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()`. |

# Class `ArrayList`

- Figure 12.1b Methods of class `ArrayList`

| |
|---|
| `public` *Base_Type* `get(int index)`<br>Returns the element at the position specified by `index`. Throws `IndexOutOfBounds-Exception` if `index` < 0 or `index` ≥ `size()`. |
| `public` *Base_Type* `set(int index,` *Base_Type* `element)`<br>Replaces the element at the position specified by `index` with the given element. Returns the element that was replaced. Throws `IndexOutOfBoundsException` if `index` < 0 or `index` ≥ `size()`. |
| `public` *Base_Type* `remove(int index)`<br>Removes and returns the element at the specified index. Shifts elements at subsequent positions toward position `index` by decreasing their indices by 1. Decreases the list's size by 1. Throws `IndexOutOfBoundsException` if `index` < 0 or `index` ≥ `size()`. |
| `public boolean remove(Object element)`<br>Removes the first occurrence of `element` 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 `element` was removed; otherwise returns false and does not alter the list. |

# Creating Instance of `ArrayList`

- Necessary to
  `import java.util.ArrayList;`
- Create and name instance
  ```
  ArrayList<String> list =
    new ArrayList<String>(20);
  ```
- This list will
  - Hold `String` objects
  - Initially hold up to 20 elements

# Using Methods of `ArrayList`

- Object of an ArrayList used like an array
  - But methods must be used
  - Not square bracket notation
- Given

```
ArrayList<String> aList =
    new ArrayList<String>(20);
```

  - Assign a value with

```
aList.add(index, "Hi Mom");
aList.set(index, "Yo Dad");
```

# Programming Example

- A To-Do List
  - Maintains a list of everyday tasks
  - User enters as many as desired
  - Program displays the list
- View source code, listing 12.1

  **`class ArrayListDemo`**

# Programming Example

```
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
```

Sample screen output

# Programming Example

- When accessing all elements of an **`ArrayList`** object
  - Use a For-Each loop
- Use the **`trimToSize`** method to save memory
- To copy an **`ArrayList`**
  - Do not use just an assignment statement
  - Use the **`clone`** method

# Parameterized Classes, Generic Data Types

- Class **`ArrayList`** is a *parameterized class*
  - It has a parameter which is a type
- Possible to declare our own classes which use types as parameters
- Note earlier versions of Java had a type of **`ArrayList`** that was not parameterized

# The Java Collections Framework

- A collection of interfaces and classes that implement useful data structures and algorithms
- The `Collection` interface specifies how objects can be added, removed, or accessed from a `Collection`
- Brief introduction to just two implementations, `HashSet` and `HashMap`
  - See other references for more info

# `HashSet` Class

- Used to store a set of objects
- Uses the same `<>` notation as an `ArrayList` to specify the data type
- View source code, listing 12.2 `class HashSetDemo`
- If you use `HashSet` of your own class, it must override `hashCode()` and `equals()`

# `HashMap` Class

- Used like a database to efficiently map from a **key** to an **object**
- Uses the same `<>` notation as an `ArrayList` to specify the data type of both the key and object
- View source code, listing 12.3
  `class HashMapDemo`
  - If you use `HashMap` of your own class as key, it must override `hashCode()` and `equals()`

# Linked Data Structures:Outline

- The Class `LinkedList`
- Linked Lists
- Implementing Operations of a Linked List
- A Privacy Leak
- Inner Classes

# Linked Data Structures:Outline

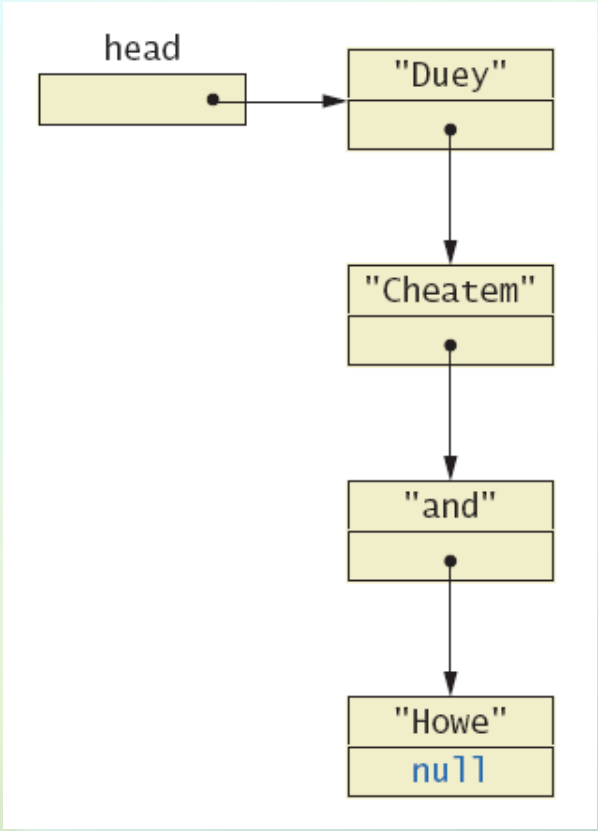
- Node Inner Classes
- Iterators
- The Java `Iterator` Interface
- Exception Handling with Linked Lists
- Variations on a Linked List
- Other Linked Data Structures

# Class `LinkedList`

- Linked data structure
  - Collection of objects
  - Each object contains data and a reference to another object in the collection
- Java provides a class to do this, `LinkedList`
  - More efficient memory use than `ArrayList`
- We will write our own version to learn the concepts of a linked list

# Linked Lists

- A dynamic data structure
- Links items in a list to one another
- Figure 12.4, a linked list



# Linked Lists

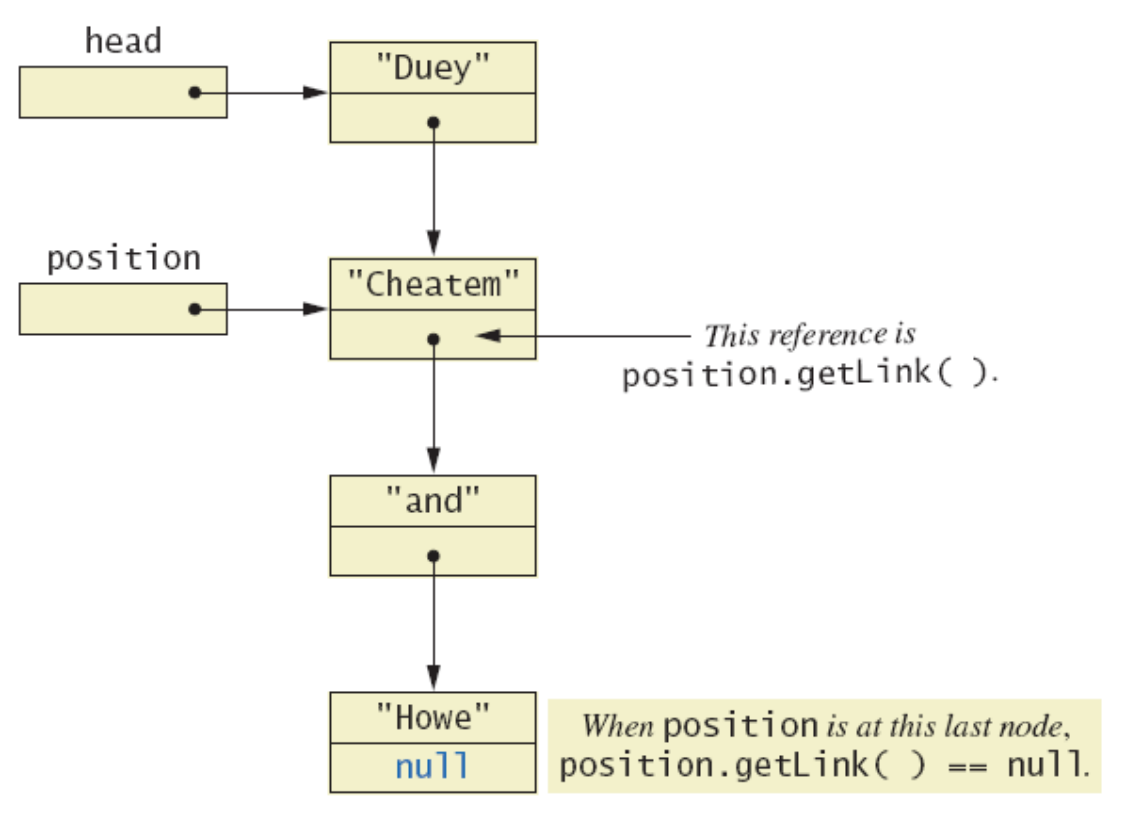
- Node of a linked list object requires two instance variables
  - Data
  - Link
- View sample class, listing 12.4
  `class ListNode`
  - This example has `String` data
  - Note the link, a reference to the type which is the class

# Implementing Operations of Linked Lists

- Now we create a linked list class which uses the node class
- View class, listing 12.5 class **`StringLinkedList`**
- Note the single instance variable of type **`ListNode`**
- Note method to traverse and print the contents of the list

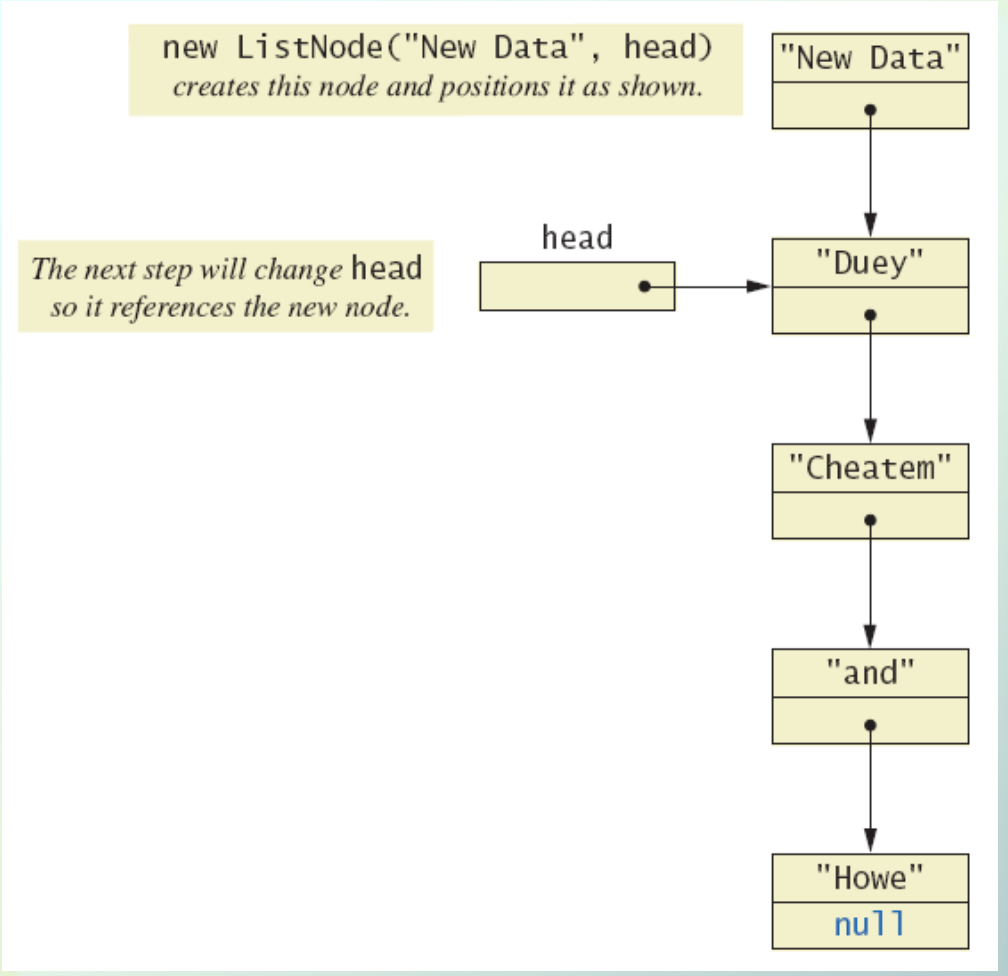
# Implementing Operations of Linked Lists

- Figure 12.5 Moving down a linked list



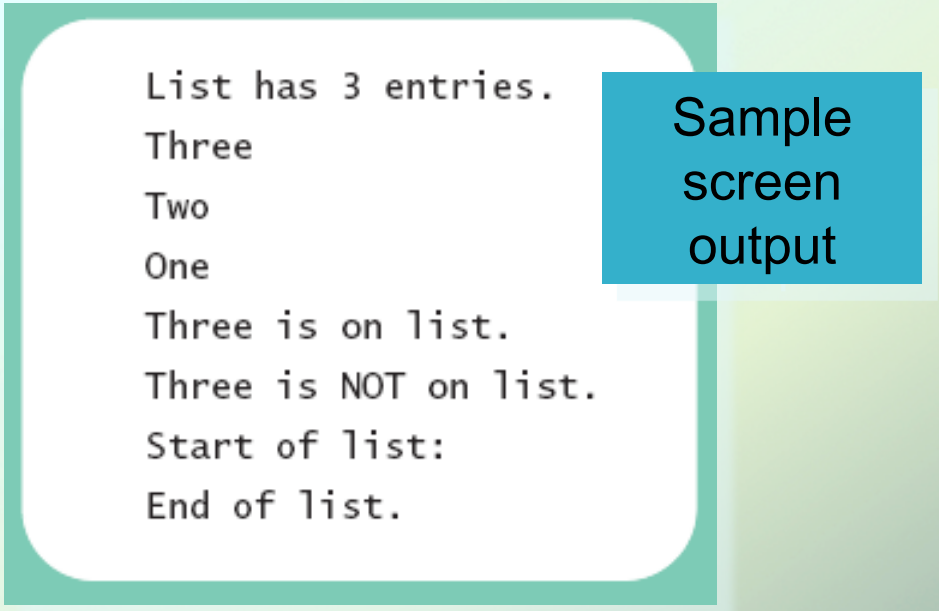
# Implementing Operations of Linked Lists

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



# Implementing Operations of Linked Lists

- View linked-list demonstration, listing 12.6
  **`class StringLinkedListDemo`**



# Implementing Operations of Linked Lists

- Java automatically returns memory used by deleted node to the operating system.
  - Called automatic *garbage collecting*
- In this context, note the significance of `NullPointerException` messages
- Consider the fact that our program has a reference (name) to only the head node

# A Privacy Leak

- Note results of `getLink` in class `ListNode`
  - Returns reference to `ListNode`
  - This is a reference to an instance variable of a class type … which is supposed to be private
- Typical solution is to make `ListNode` a private inner class of `StringLinkedList`

# Inner Classes

- Defined within other classes
- Example

```
public class OuterClass
{
    Declarations_of_OuterClass_Instance_Variables
    Definitions_of_OuterClass_Methods

    private class InnerClass
    {
        Declarations_of_InnerClass_Instance_Variables
        Definitions_of_InnerClass_Methods
    }
}
```

# Inner Classes

- Inner class definition local to the outer-class definition
  - Inner-class definition usable anywhere within definition of outer class
- Methods of inner and outer classes have access to each other's methods, instance variables

# Node Inner Classes

- We show `ListNode` as a private inner class
  - This is safer design
  - Hides method getLink from world outside `StringLinkedList` definition
- View new version, listing 12.7 class `StringLinkedListSelfContained`

# Iterators

- A variable that allows you to step through a collection of nodes in a linked list
  - For arrays, we use an integer
- Common to place elements of a linked list into an array
  - For display purposes, array is easily traversed
- View method to do this, listing 12.8

**`method toArray`**

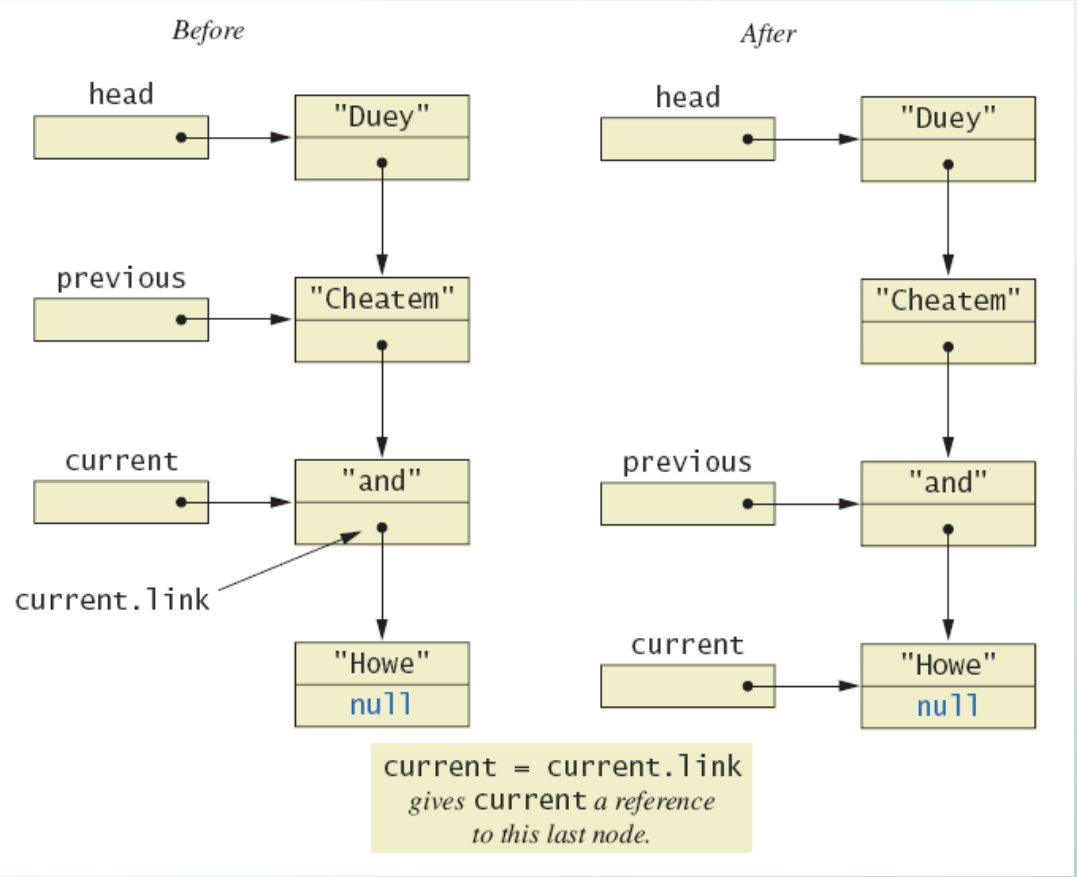
# Iterators

- Consider an iterator that will move through a linked list
  - Allow manipulation of the data at the nodes
  - Allow insertion, deletion of nodes
- View sample code, listing 12.9

  **`class StringLinkedListWithIterator`**

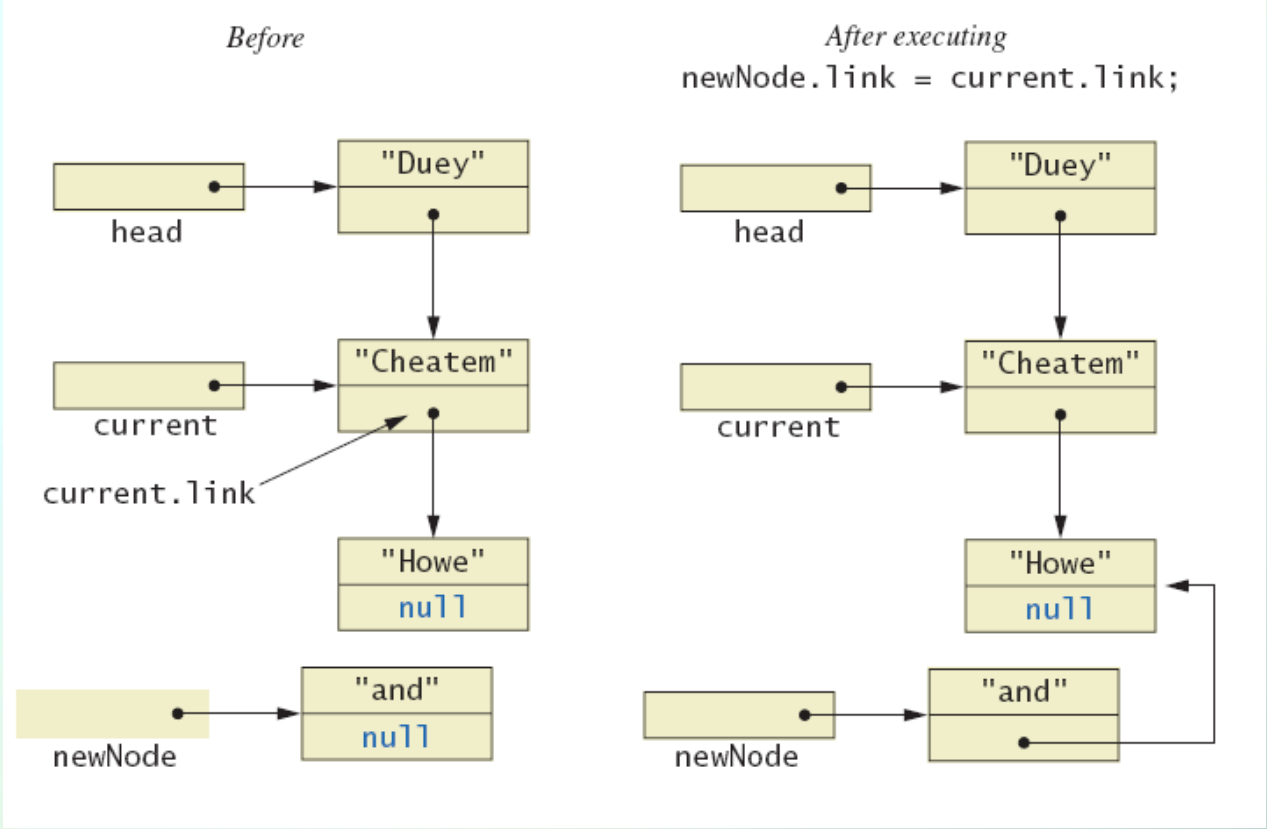
# Iterators

- Figure 12.7 The effect of `goToNext` on a linked list



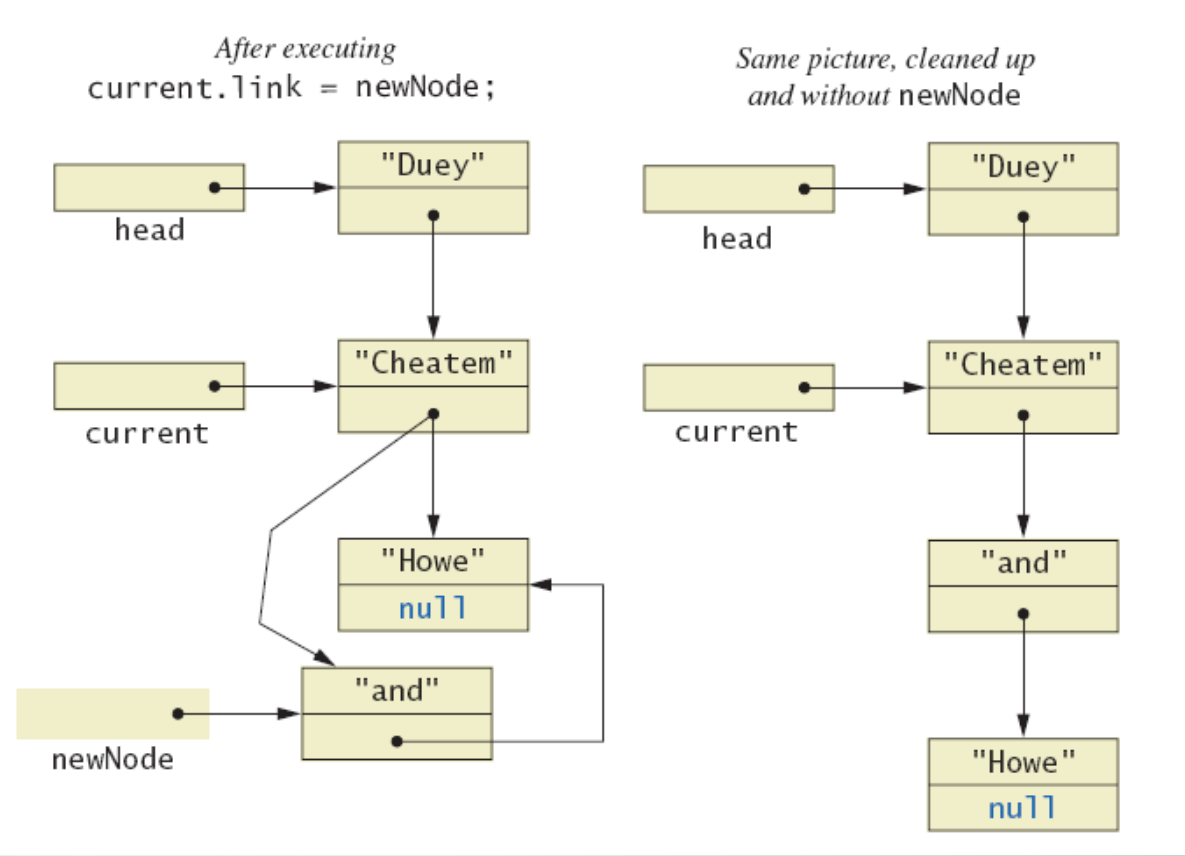
# Iterators

- Figure 12.8a Adding node to linked list using **`insertAfterIterator`**



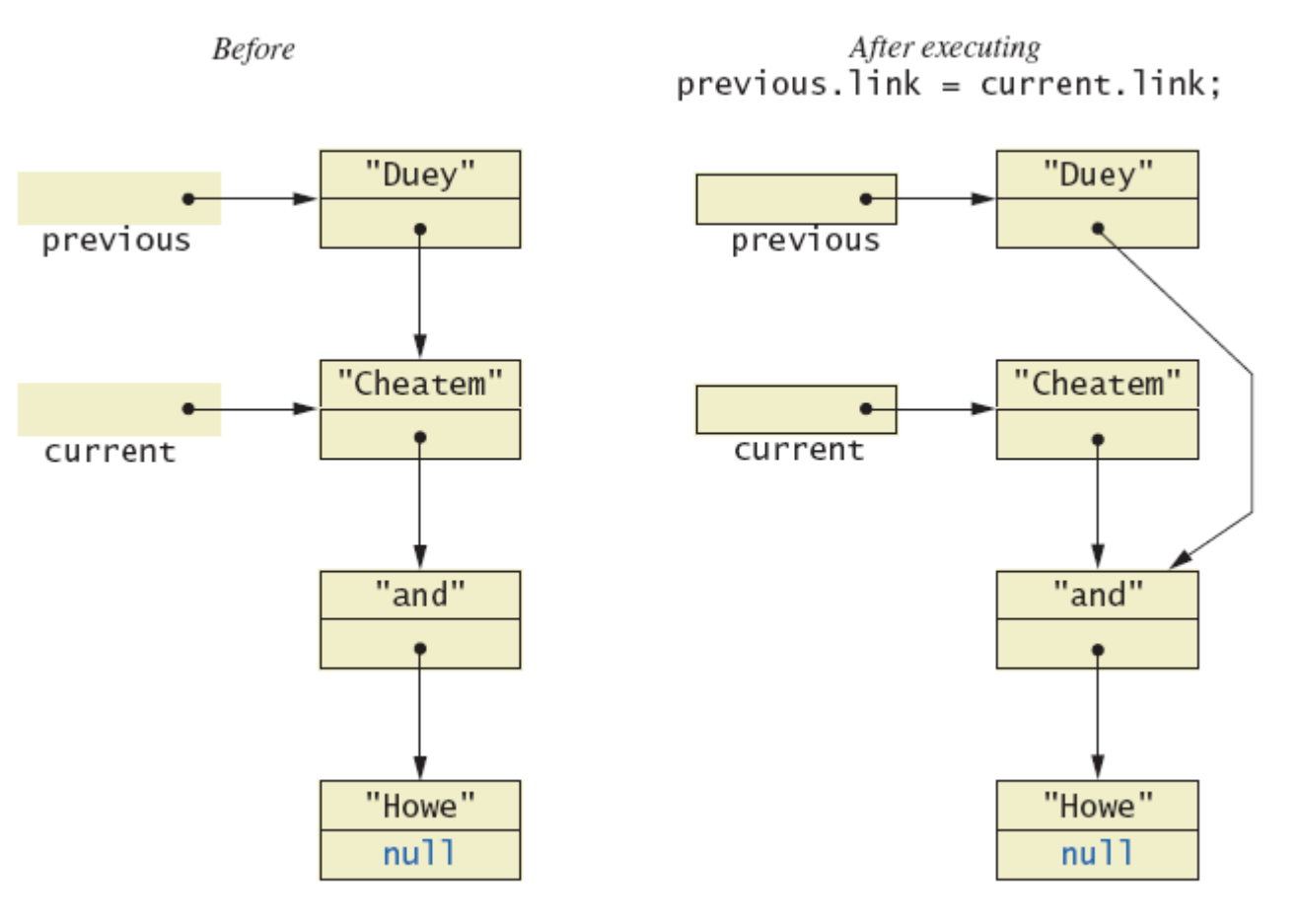
# Iterators

- Figure 12.8b Adding node to linked list using **`insertAfterIterator`**



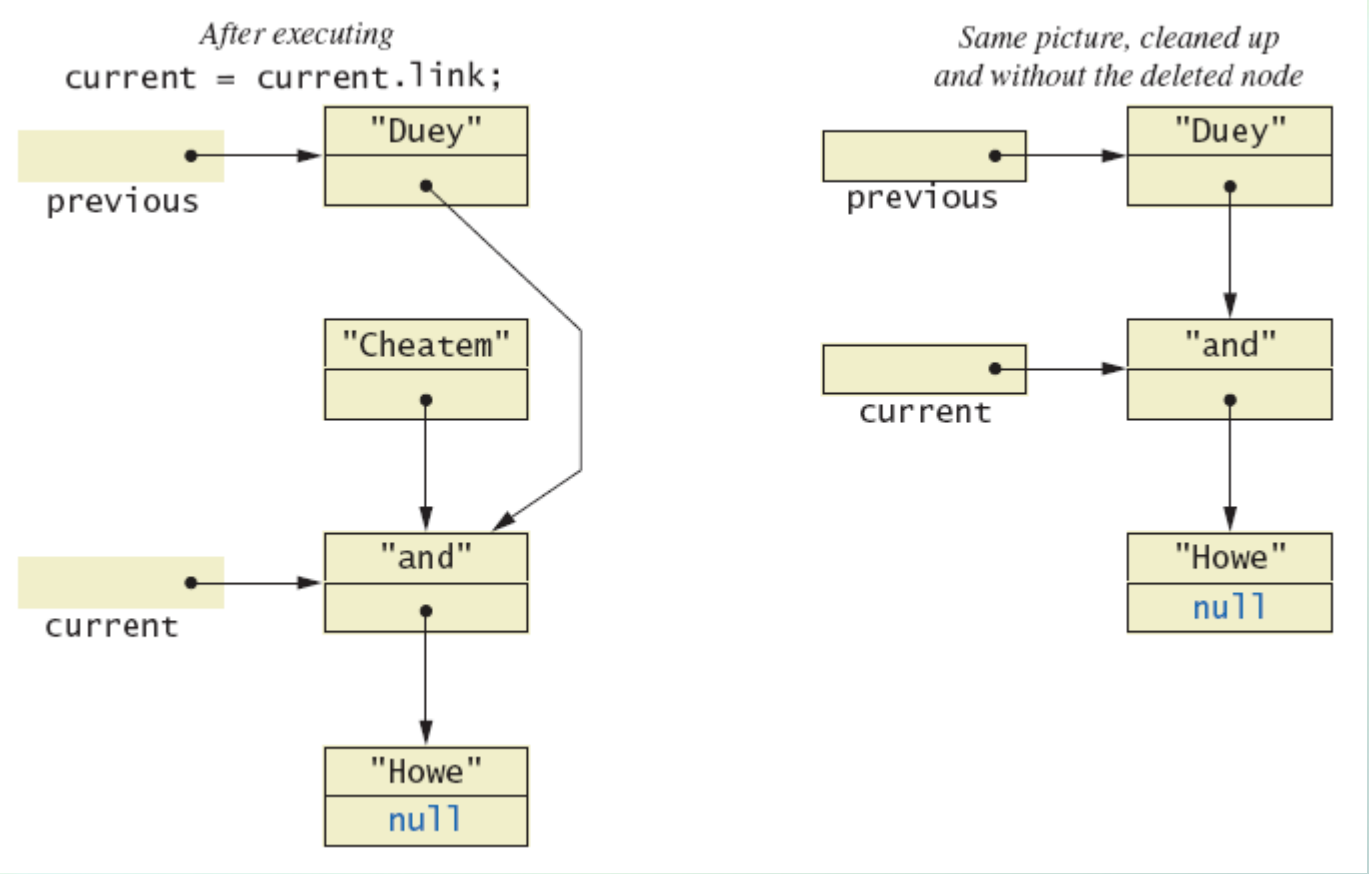
# Iterators

- Figure 12.9a Deleting a node



# Iterators

- Figure 12.9b Deleting a node



# Java `Iterator` Interface

- Java formally considers an iterator to be an object
- Note interface named Iterator with methods
  - **`hasNext`** – returns boolean value
  - **`next`** – returns next element in iteration
  - **`remove`** – removes element most recently returned by **`next`** method

# Exception Handling with Linked Lists

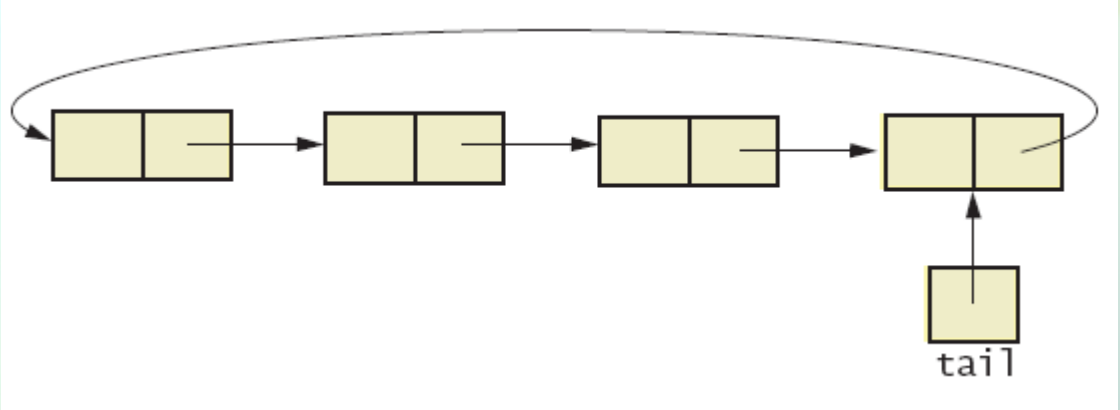
- Recall class **`StringLinkedListWithIterator`**
  - Methods written so that errors caused screen message and program end
- More elegant solution is to have them throw exceptions
  - Programmer decides how to handle
- Note class which does this, listing 12.10 class **`LinkedListException`**

# Variations on a Linked List

- Possible to make a linked where data element is of any type
  - Replace type String in definition of node class with desired data type
- Consider keeping a reference to last node in list
  - Called the *tail* of the list
  - Constructors, methods modified to accommodate new reference

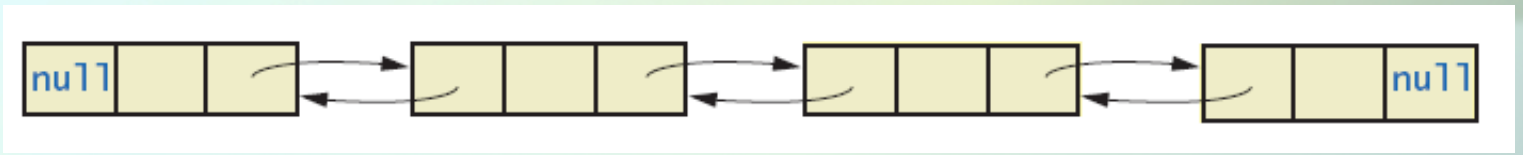
# Variations on a Linked List

- Consider having last link point back to head
  - Creates a circular linked list
- Figure 12.10 Circular linked list



# Variations on a Linked List

- Also possible to have backward as well as forward links in the list
  - Doubly linked list
  - Possible to traverse in either direction
- Figure 12.11 Doubly linked list

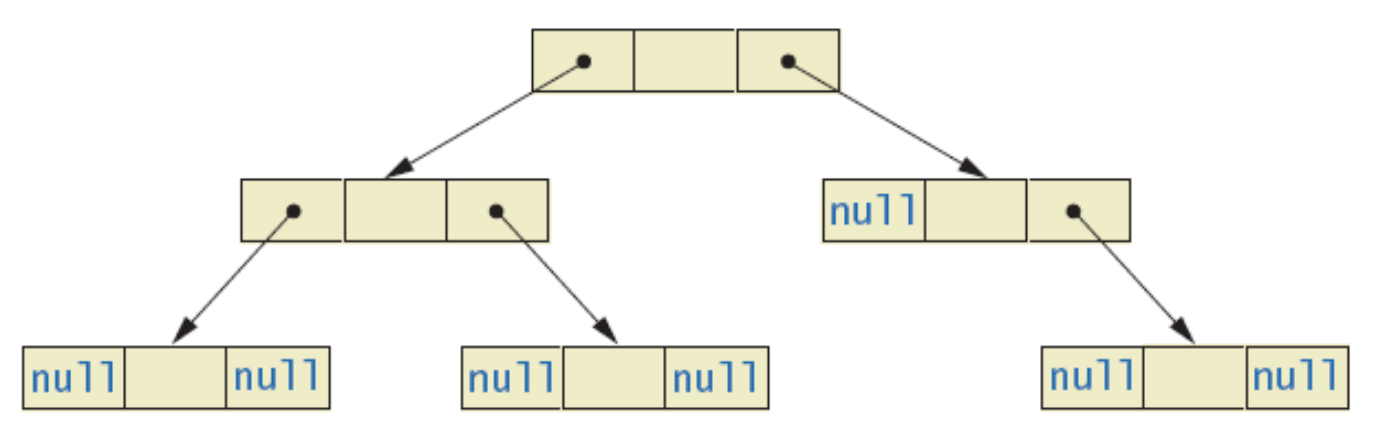


# Other Linked Data Structures

- Stack
  - Elements removed from ADT in reverse order of initial insertion
  - Can be implemented with linked list
- Tree
  - Each node leads to multiple other nodes
  - Binary tree leads to at most two other nodes

# Other Linked Data Structures

- Figure 12.12 Binary tree



# Generics: Outline

- The Basics
- Programming Example: A Generic Linked List

# Basics of Generics

- Beginning with Java 5.0, class definitions may include parameters for types
  - Called *generics*
- Programmer now can specify any class type for the type parameter
- View class definition, listing 12.11
  `class Sample<T>`
- Note use of `<T>` for the type parameter

# Basics of Generics

- Legal to use parameter T almost anywhere you can use class type
  - Cannot use type parameter when allocating memory such as `anArray = new T[20];`
- Example declaration

```
Sample <String> sample1 =
    new Sample<String>();
```

  - Cannot specify a primitive type for the type parameter

# Programming Example

- Generic linked list
  - Revision of listing 12.5
  - Use type parameter E instead of String
- Note similarities and differences of parameterized class with non-paramaterized classes
- View generic class, listing 12.12

```
class LinkedList <E>
```

# Programming Example

- View demo program, listing 12.13
  **`class LinkedListDemo`**

# Summary

- Java Class Library includes **`ArrayList`**
  - Like an array that can grow in length
  - Includes methods to manipulate the list
- Java Collections Framework contains many classes to store and manipulate objects
- Linked list data structure contains nodes (objects)
- Linked data structure is self-contained by making the node class an inner class

# Summary

- Variable or object which allows stepping through linked list called an iterator
- Class can be declared with type parameter
- Object of a parameterized class replaces type parameter with an actual class type
- Classes **`ArrayList`**, **`HashSet`**, **`HashMap`** and **`LinkedList`** are parameterized classes