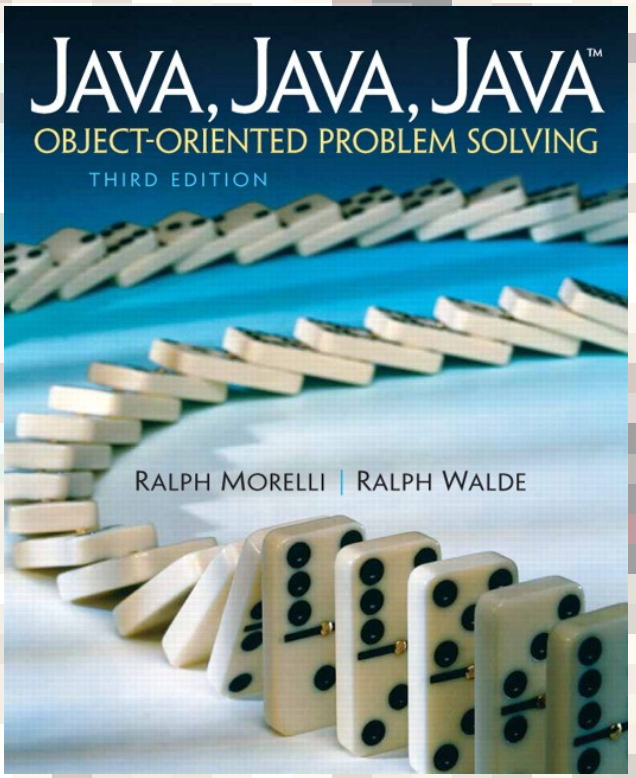


presentation slides for

Object-Oriented Problem Solving

JAVA, JAVA, JAVA

Third Edition

Ralph Morelli | Ralph Walde
Trinity College
Hartford, CT

published by Prentice Hall

Java, Java, Java

Object Oriented Problem Solving

Lecture 10: Abstract Data Structures (ADT): Lists, Stacks, and Linked List - Generics in Java

Objectives

- Understand the concepts of a dynamic data structure and an Abstract Data Type (ADT).
- Be able to create and use dynamic data structures such as linked lists
- Be able to use inheritance to define extensible data structures.
- Be able to use the Java generic type construct.

 Copyright 2006. All rights reserved.

Outline

- The Linked List Data Structure
- Object-Oriented Design: The List Abstract Data Type (ADT)
- The Stack ADT
- From the Java Library: The Java Collections Framework and Generic Types

 Copyright 2006. All rights reserved.

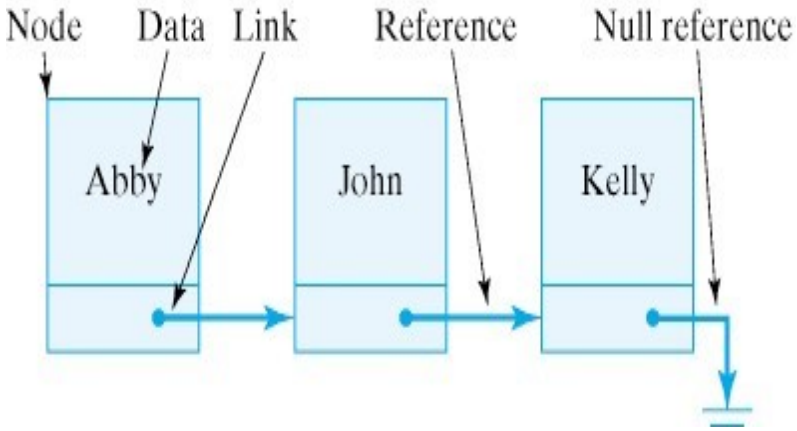
The Linked List Data Structure

- A *dynamic* data structure is one that can grow or shrink. A linked list is an example.
- A *static* data structure is one whose size is fixed during a program's execution.
- A *self-referential object* is one that contains a reference to an object of the same type.
- A *linked list* is a list in which a collection of nodes are linked together by references from one node to the next.
- The *nodes* of a linked list are self-referential.

 Copyright 2006. All rights reserved.

A Node Class

- Each node stores some data and a reference (or link) to another node.
- A linked list is terminated with a null reference.

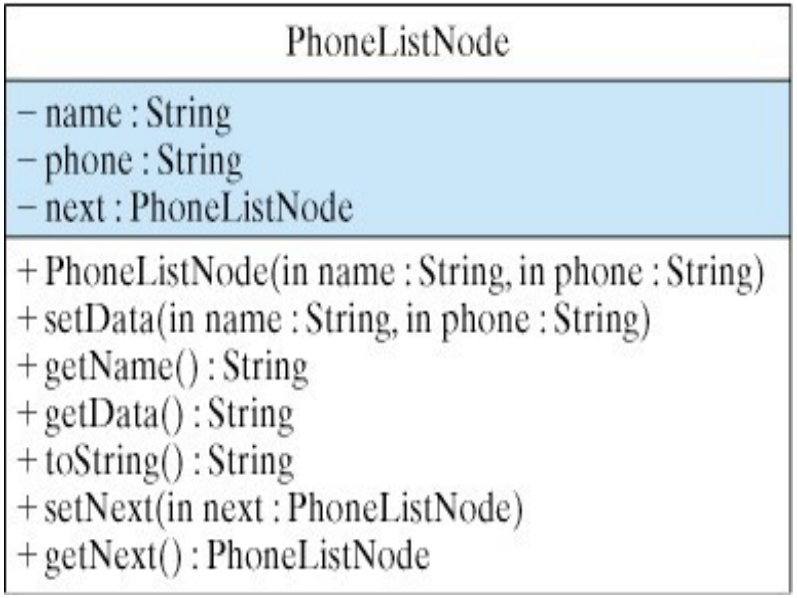


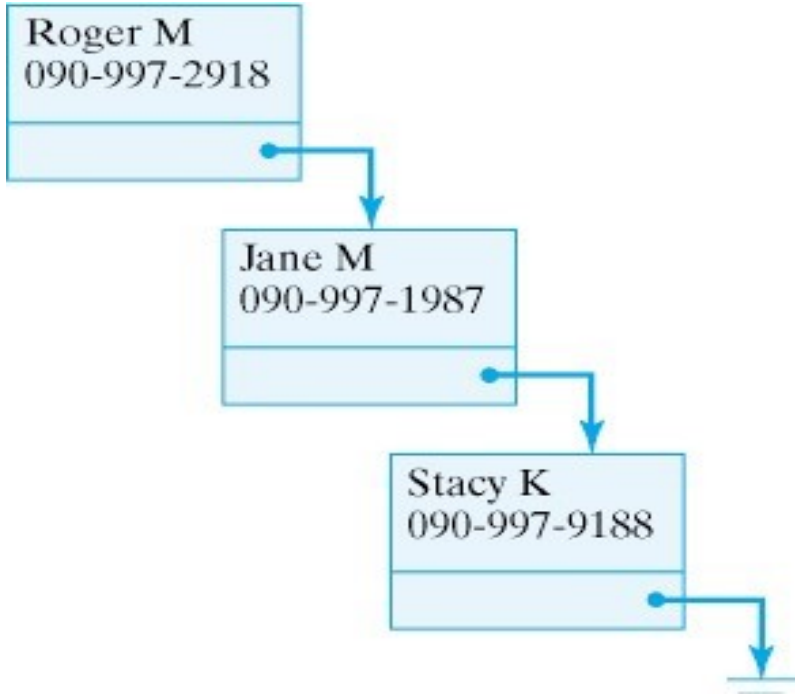
Node
− data : Object − next : Node
+ Node(in o : Object) + setData(in o : Object) + getData() : Object + setNext(in link : Node) + getNext() : Node

 Copyright 2006. All rights reserved.

Example: The Dynamic Phone List

- The definition of a *PhoneListNode* is a specialization of the generic node definition.
- Data in each node is a name and a phone number.

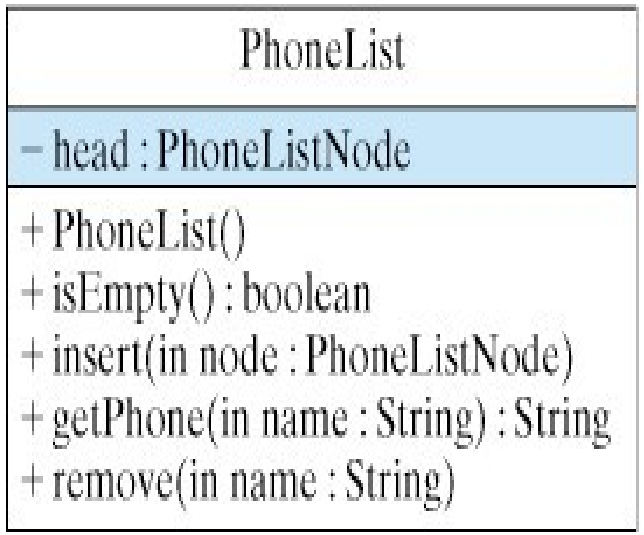


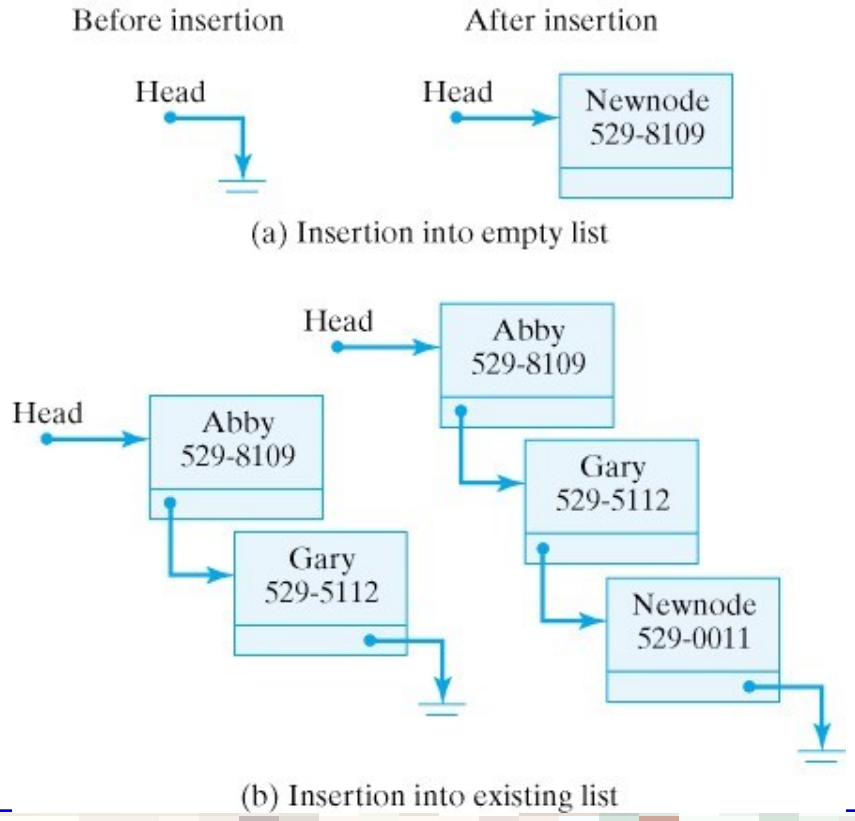


 Copyright 2006. All rights reserved.

Manipulating the Phone List

- PhoneList object contains PhoneListNode reference.
- This example inserts new node at the end of the list.





 Copyright 2006. All rights reserved.

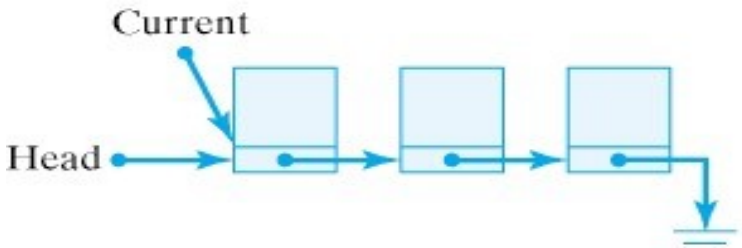
Inserting Nodes into a List

- If list is empty set head to newNode.
- Else traverse list and insert newNode at end.

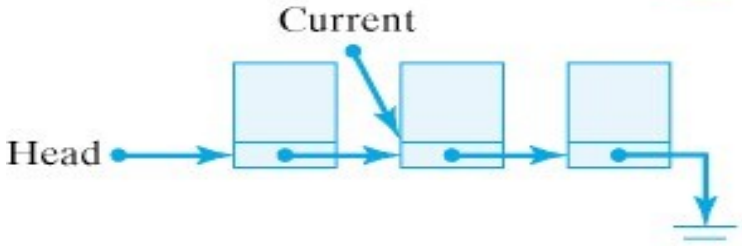
```
public void insert(PhoneListNode newNode){
   if (isEmpty())
     head = newNode;                          // Insert at head of list
   else {
     PhoneListNode current = head;            // Start traversal at head
     while (current.getNext() != null)        // While not the last node
       current = current.getNext();           // go to next node.
     current.setNext(newNode);                // Do the insertion
   } // else
} // insert()

// Note: the newNode's 'next' field is null.
```

 Copyright 2006. All rights reserved.

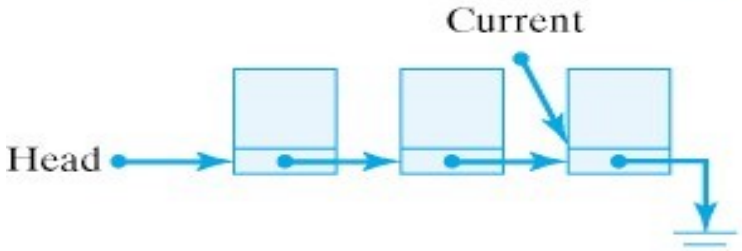
Traversing a List to Find its End



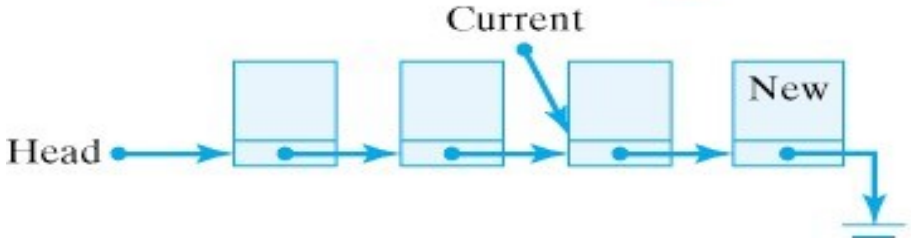
(a) Start at the head of the list



(b) Traverse by following the links



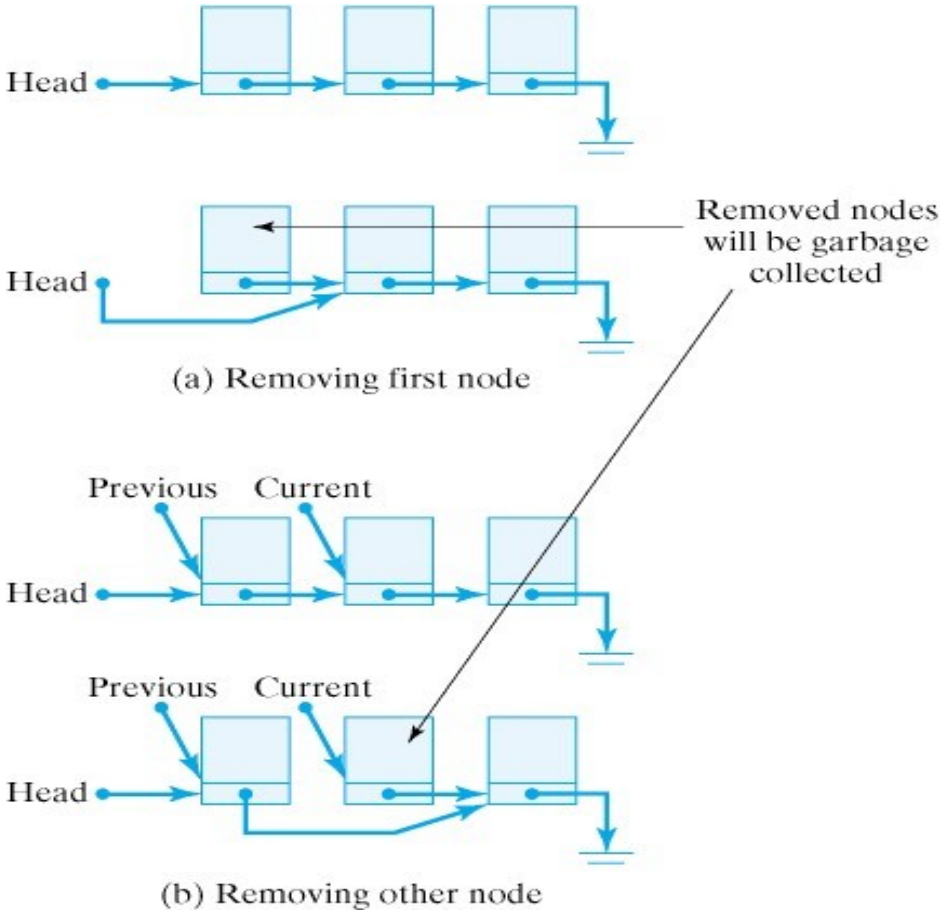
(c) Find the end of the list



(d) Insert the new node

 Copyright 2006. All rights reserved.

Removing a Node from a List



 Copyright 2006. All rights reserved.

Java Code for Removing a Node

```
public String remove(String name) {          // Remove an entry by name
   if (isEmpty())                            // Case 1: empty list
     return "Phone list is empty";
   PhoneListNode current = head;
   PhoneListNode previous = null;
   if (current.getName().equals(name)) {     // Case 2: remove first node
     head = current.getNext();
     return "Removed " + current.toString() ;
   }// if
   while((current.getNext()!= null)&&(!current.getName().equals(name))){
     previous = current;
     current = current.getNext();
   }// while
   if (current.getName().equals(name)) {     // Case 3: remove named node
     previous.setNext(current.getNext());
     return "Removed " + current.toString();
   } else
     return ("Sorry. No entry for " + name); // Case 4: node not found
} // remove()
```

 Copyright 2006. All rights reserved.

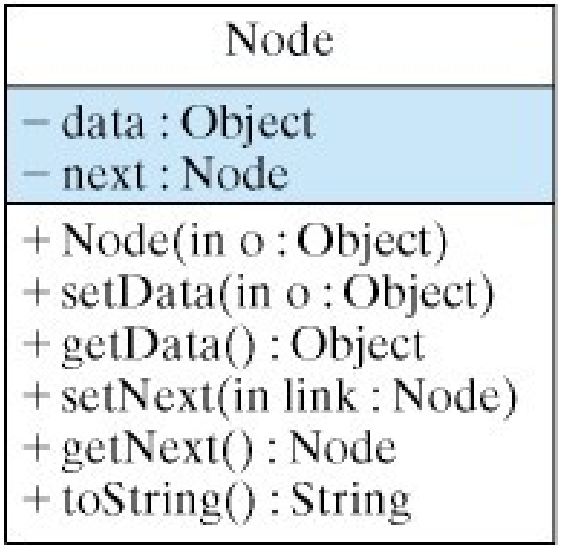
OOD: The List Abstract Data Type(ADT)

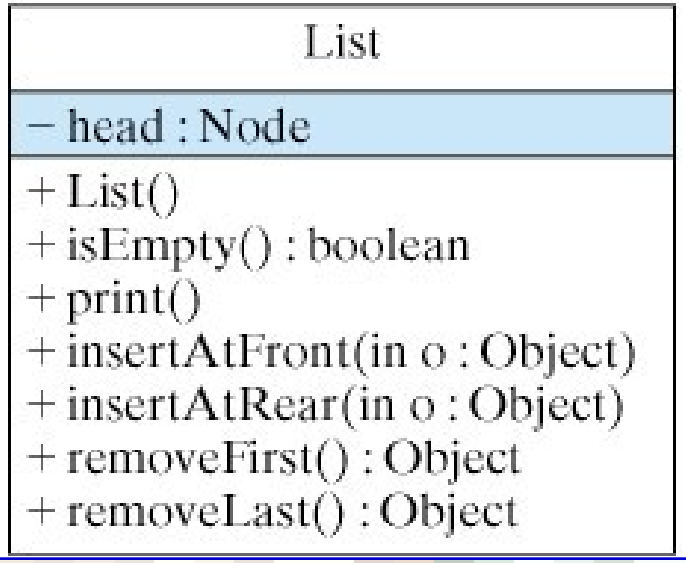
- OOD: Object-Oriented Design
- An Abstract Data Type (ADT) involves two components: the data being stored and manipulated and the methods and operations that can be performed on the data.
- A List ADT is like the Phone List except that it can be used to store and manipulated any kind of data.
- An ADT should hide implementation details and provide a collection of public methods as its interface with users.

 Copyright 2006. All rights reserved.

The List ADT Node and List Classes

- Access methods (e.g., setData(), getData(), setNext(), getNext()) use any kind of Objects
- The List class has methods for inserting or removing from either end of the List (e.g., the Front/Rear, First/Last.)

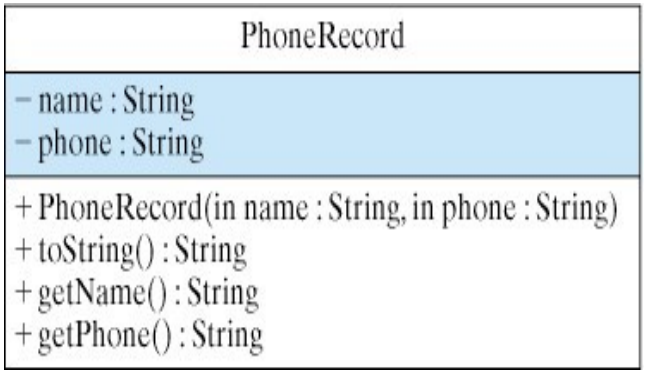




 Copyright 2006. All rights reserved.

A PhoneRecord Class

- This class will be used to test the List ADT
- It has the same data as PhoneListNode.




```
public class PhoneRecord {
  private String name;
  private String phone;
  public PhoneRecord(String s1,String s2){
    name = s1;
    phone = s2;
  }// PhoneRecord() constructor
  public String toString() {
    return name + " " + phone;
  }// toString()
  public String getName( ) {
    return name;
  }// getName()
  public String getPhone( ) {
    return phone;
  }// getPhone()
} // PhoneRecord
```

 Copyright 2006. All rights reserved.

Testing the List ADT

```
public static void main( String argv[] ) {
    // Create list and insert heterogeneous nodes
    List list = new List();
    list.insertAtFront(new PhoneRecord("Roger M", "997-0020"));
    list.insertAtFront(new Integer(8647));
    list.insertAtFront(new String("Hello World"));
    list.insertAtRear(new PhoneRecord("Jane M", "997-2101"));
    list.insertAtRear(new PhoneRecord("Stacy K", "997-2517"));
    System.out.println("Generic List");
    list.print();           // Print the list
    Object o;               // Remove objects and print resulting list
    o = list.removeLast();
    System.out.println(" Removed " + o.toString());
    System.out.println("Generic List:");
    list.print();
    o = list.removeLast();
    System.out.println(" Removed " + o.toString());
    System.out.println("Generic List:");
    list.print();
    o = list.removeFirst();
    System.out.println(" Removed " +o.toString());
    System.out.println("Generic List:");
    list.print();
} // main()
```

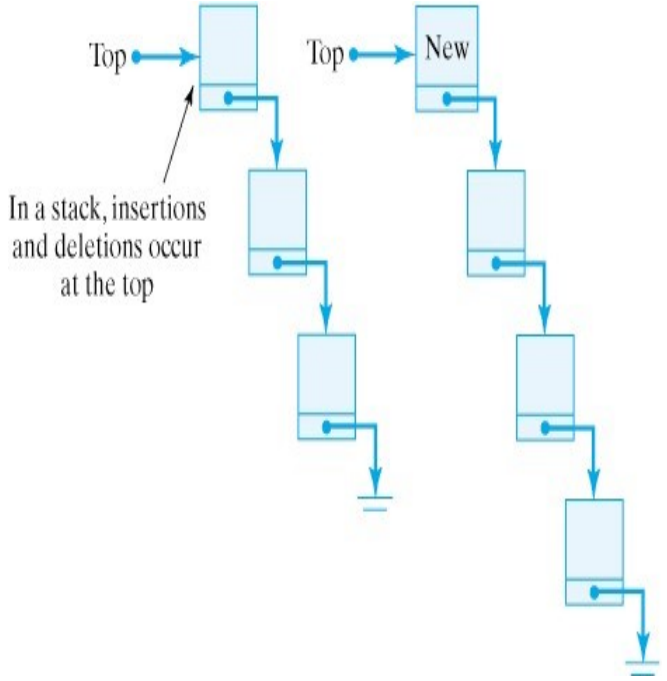
The Stack ADT

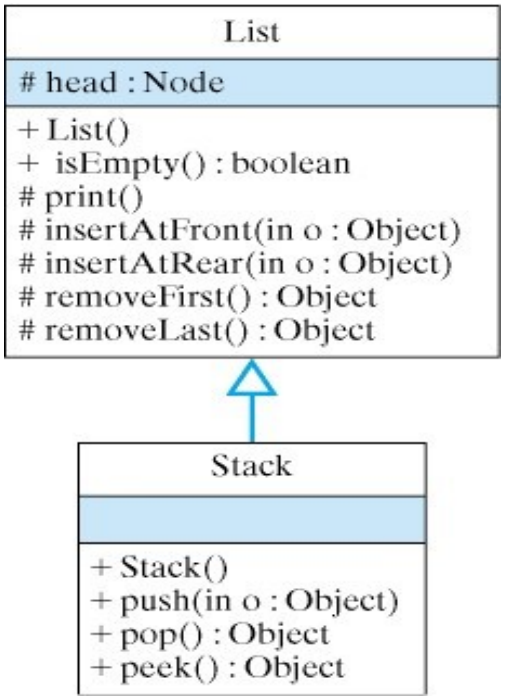
- A *Stack* is a special type of List, which extends List.
- It allows insertions and removals only at the front of the list.
- Thus, it enforces *last-in/first-out (LIFO)* behavior.
- The stack insert operation is called *push*.
- The stack remove operation is called *pop*.
- Stacks behave like stacks of dishes.
- Stacks are used in several computing tasks like managing calls to methods.

 Copyright 2006. All rights reserved.

The Stack class

- The *Stack* class is a subclass of List.
- The *peek()* method returns the top element, without removing it.





 Copyright 2006. All rights reserved.

Implementation of the Stack class

- Stack class methods simply call List methods insertAtFront() and removeFirst().

```
public class Stack extends List {

    public Stack() {
        super();        // Initialize the list
    }

    public void push( Object obj ) {
        insertAtFront( obj );
    }

    public Object pop() {
        return removeFirst();
    }

} // Stack
```

 Copyright 2006. All rights reserved.

Testing the Stack class

- This program reverses the letters of a string.
- Note how Objects are converted to chars.

```
public static void main( String argv[] ) {
        Stack stack = new Stack();
        String string = "Hello this is a test string";

        System.out.println("String: " + string);
        for (int k = 0; k < string.length(); k++)
            stack.push(new Character( string.charAt(k)));

        Object o = null;
        String reversed = "";
        while (!stack.isEmpty()) {
            o  = stack.pop();
            reversed = reversed + o.toString();
        }
        System.out.println("Reversed String: " + reversed);
    } // main()
```

 Copyright 2006. All rights reserved.

The Java Collections Framework and Generic Types

- The *Java Collections Framework* is a group of classes and interfaces in java.util.* that implement abstract data types.
  See https://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html
- The *Generic Type* construct allows a programmer to specify a type for the objects stored in a data structure.
- Java 5.0 has reimplemented the Java Collections Framework using generic types.
- The Java Collections Framework includes implementations of LinkedList, Stack, Queue, and numerous other abstract data types.

 Copyright 2016. All rights reserved.

Generic Types in Java

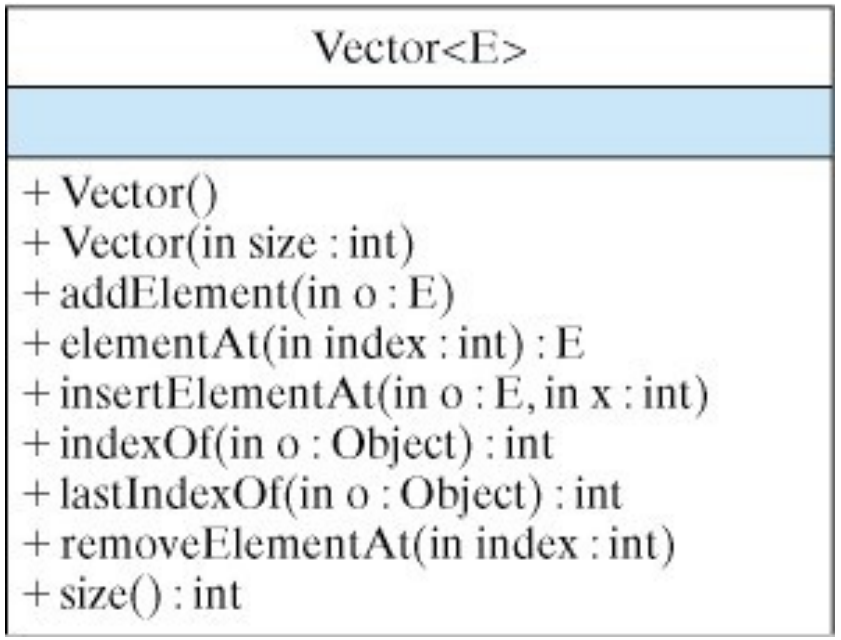
- The generic type syntax is **ClassName<E>**
  - Example: List<String>, List<Student>, Vector<Person>, etc.
- A generic type is a generic class or interface that is parameterized over types.
- ClassName is a class or interface from the Java 5.0 Collections Framework.
- E is an unspecified class or interface name.

 Copyright 2016. All rights reserved.

Using the Generic Vector Type

- The following Java code is an example of using a generic type. It creates a Vector object which can store String objects where Vector is in the Java 5.0 Collections Framework.

```
// java.util.Vector<E>;
Vector<String> strVec =
    new Vector<String>();
strVec.addElement("alpha");
strVec.addElement("beta");
String str =
    strVec.elementAt(0);
```

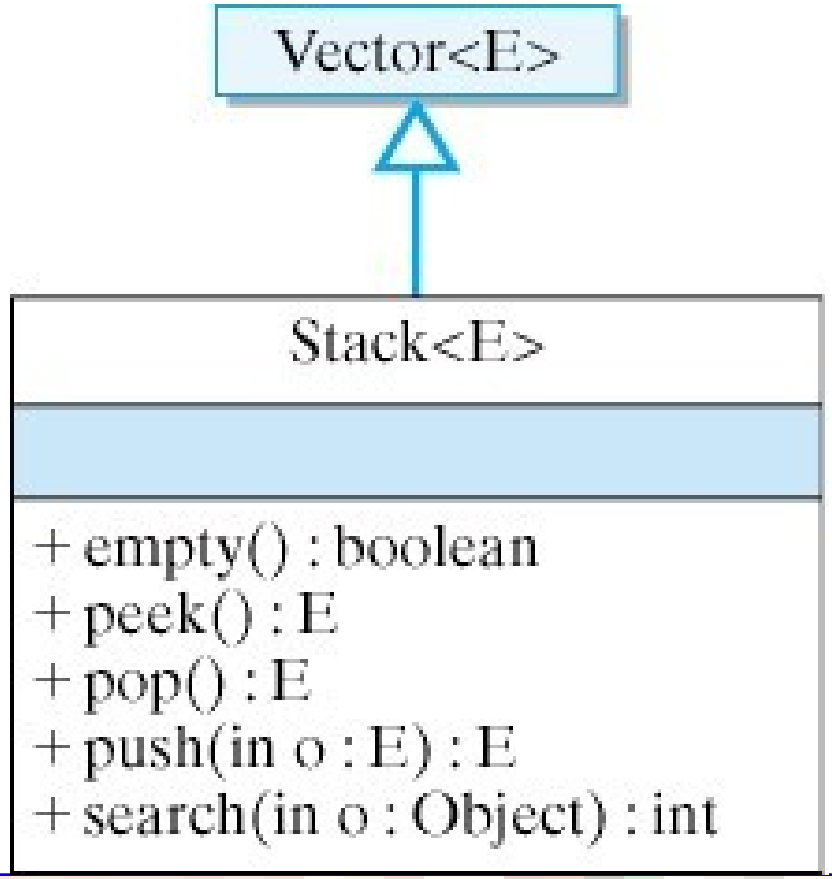


 Copyright 2016. All rights reserved.

Using Methods of Stack<E>

- Stack<E> is a subclass of Vector<E> in the Java 5.0 Collections Framework.

```
Stack<String> stk =
    new Stack<String>();
stk.push("alpha");
stk.push("beta");
String str =  stk.pop();
```



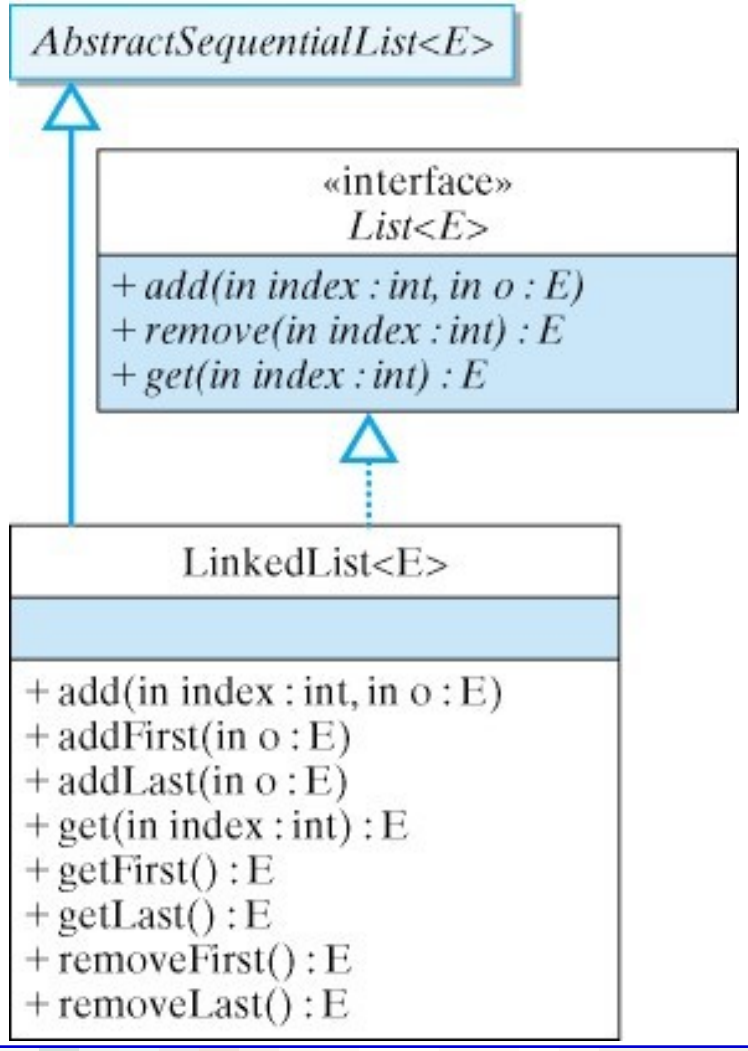
 Copyright 2016. All rights reserved.

List<E> Interface and LinkedList<E> Class

- The **LinkedList<E>** and **ArrayList<E>** classes both implement the **List<E>** interface.
- A LinkedList<E> is appropriate for use when data will always be traversed in the same order that it is entered, and a relatively small amount of data is involved.
- An ArrayList<E> is appropriate for use when data will be traversed in an order different from how it is entered, or a large amount of data is involved.

 Copyright 2016. All rights reserved.

Methods of List<E> and LinkedList<E>



- ArrayList<E> also implements the methods of List<E>

 Copyright 2016. All rights reserved.

Using List<E> and LinkedList<E>

- The following code demonstrates how a variable of type List<PhoneRecord> can be instantiated with an object of type LinkedList<PhoneRecord>.

```
public static void testList() {
  List<PhoneRecord> theList =
            new LinkedList<PhoneRecord>();
  // new ArrayList<PhoneRecord>() could also be used.
  theList.add(new PhoneRecord("Roger M", "090-997-2918"));
  theList.add(new PhoneRecord("Jane M", "090-997-1987"));
  theList.add(new PhoneRecord("Stacy K", "090-997-9188"));
  theList.add(new PhoneRecord("Gary G", "201-119-8765"));
  theList.add(new PhoneRecord("Jane M", "090-997-1987"));
  System.out.println("Testing a LinkedList List");
  for (PhoneRecord pr : theList)
    System.out.println(pr);
} // testList()
```

 Copyright 2016. All rights reserved.

Technical Terms

- Abstract Data Type (ADT)
- binary search tree
- data structure
- dequeue
- dynamic structure
- enqueue
- first-in/first-out (FIFO)
- generic type
- Java collections framework
- key
- last-in/first-out (LIFO)
- link
- linked list
- list
- pop
- push
- queue
- reference
- self-referential object
- stack
- static structure
- traverse
- value

 Copyright 2006. All rights reserved.

Summary Of Important Points (part 1)

- A *data structure* is used to organize data and make them more efficient to process. An array is an example of a *static structure* because its size does not change during a program's execution. A vector is an example of a *dynamic structure,* one whose size can grow and shrink during a program's execution.
- A *linked list* is a linear structure in which the individual nodes of the list are joined together by references. A *reference* is a variable that refers to an object. Each node of the list has a *link* variable that refers to another node. An object that can refer to the same kind of object is said to be *self-referential*.

 Copyright 2006. All rights reserved.

Summary Of Important Points (part 2)

- The **Node** class is an example of a self-referential class. It contains a link variable that refers a **Node**. By assigning references to the link variable , **Node**s can be chained together into a linked linked list. In addition to their link variables, **Node**s contain data variables that should be accessed through public methods.
- Depending on the use of a linked list, nodes can be inserted at various locations in the list: at the head, the end, or in the middle of the list.

 Copyright 2006. All rights reserved.

Summary Of Important Points (part 3)

- Traversal algorithms must be used to access the elements of a singly linked list. To traverse a list, you start at the first node and follow the links of the chain until you reach the desired node.
- Depending on the application, nodes can be removed from the front, rear, or middle of a linked list. Except for the front node, traversal algorithms are used to locate the desired node.

 Copyright 2006. All rights reserved.

Summary Of Important Points (part 4)

- In developing list algorithms, it is important to test them thoroughly. Ideally, you should test every possible combination insertions and removals that the list supports. Practically, you should test every independent case of insertions and removals that the list supports.
- An *Abstract Data Type* (ADT) combines two elements: a collection of data, and the operations that can be performed on the data. For a list ADT, the data are the values (**Object**s or **int**s) contained in the nodes that make up the list, and the operations are insertion, removal, and tests of whether the list is empty.

 Copyright 2006. All rights reserved.

Summary Of Important Points (part 5)

- In designing an ADT, it is important to provide a public interface that can be used to access the ADTs data. The ADTs implementation should not matter to the user and therefore should be hidden. A java class definition, with its **public** and **private** aspects, is perfectly suited to implement and ADT.
- A *stack* is a list that allows insertions and removals only at the front of the list. A stack insertion is called a *push*, and a stack removal is called a *pop*. The first element in a stack is usually called the top of the stack. The **Stack ADT** can easily be defined as a subclass of **List**. Stacks are used for managing method call and returns in programming languages.

 Copyright 2006. All rights reserved.