Chapter 12: Collections
Lab Exercises

Topic	Lab Exercises
Linked Lists	Linked List of Integers
	Recursive Processing of Linked List
	Linked List of Objects
	Doubly Linked Lists
Queues	An Array Queue Implementation
	A Linked Queue Implementation
	Queue Manipulation
Stacks	An Array Stack Implementation
	A Linked Stack Implementation
	Stack Manipulation
	Matching Parentheses

A Linked List of Integers

File *IntList.java* contains definitions for a linked list of integers. The class contains an inner class *IntNode* that holds information for a single node in the list (a node has a value and a reference to the next node) and the following *IntList* methods:

- public IntList()—constructor; creates an empty list of integers
- public void addToFront(int val)—takes an integer and puts it on the front of the list
- public void addToEnd(int val)—takes an integer and puts it on the end of the list
- public void removeFirst()—removes the first value from the list
- public void print()—prints the elements in the list from first to last

File *IntListTest.java* contains a driver that allows you to experiment with these methods. Save both of these files to your directory, compile and run IntListTest, and play around with it to see how it works. Then add the following methods to the IntList class. For each, add an option to the driver to test it.

1. public int length()—returns the number of elements in the list
2. public String toString()—returns a String containing the print value of the list.
3. public void removeLast()—removes the last element of the list. If the list is empty, does nothing.
4. public void replace(int oldVal, int newVal)—replaces all occurrences of oldVal in the list with newVal. Note that you can still use the old nodes; just replace the values stored in those nodes.

```
// ****************************************************************
// FILE:  IntList.java
//
// Purpose: Defines a class that represents a list of integers
//
// ****************************************************************
public class IntList
{
    private IntNode front;      //first node in list

    //----------------------------------------
    //  Constructor.  Initially list is empty.
    //----------------------------------------
    public IntList()
    {
      front = null;
    }

    //----------------------------------------
    //   Adds given integer to front of list.
    //----------------------------------------
    public void addToFront(int val)
    {
      front = new IntNode(val,front);
    }

    //---------------------------------------
    //   Adds given integer to end of list.
    //---------------------------------------
    public void addToEnd(int val)
    {
      IntNode newnode = new IntNode(val,null);

      //if list is empty, this will be the only node in it
      if (front == null)
          front = newnode;
```

```
      else
          {
            //make temp point to last thing in list
            IntNode temp = front;
            while (temp.next != null)
                temp = temp.next;
            //link new node into list
            temp.next = newnode;
          }
    }

    //-----------------------------------------
    //   Removes the first node from the list.
    //   If the list is empty, does nothing.
    //-----------------------------------------
    public void removeFirst()
    {
      if (front != null)
          front = front.next;
    }

    //-----------------------------------------------
    //   Prints the list elements from first to last.
    //-----------------------------------------------
    public void print()
    {
      System.out.println("---------------------");
      System.out.print("List elements: ");

      IntNode temp = front;

      while (temp != null)
          {
            System.out.print(temp.val + " ");
            temp = temp.next;
          }
      System.out.println("\n--------------------\n");
    }

    //*************************************************************
    // An inner class that represents a node in the integer list.
    // The public variables are accessed by the IntList class.
    //*************************************************************
    private class IntNode
    {
      public int val;         //value stored in node
      public IntNode next;    //link to next node in list

    //-----------------------------------------------------------------
    // Constructor; sets up the node given a value and IntNode reference
    //-----------------------------------------------------------------
      public IntNode(int val, IntNode next)
      {
          this.val = val;
          this.next = next;
      }
    }
}
```

```
// ************************************************************
//   IntListTest.java
//
//   Driver to test IntList methods.
// ************************************************************

import java.util.Scanner;

public class IntListTest
{
    private static Scanner scan;
    private static IntList list = new IntList();

    //-----------------------------------------------------------------
    // Creates a list, then repeatedly prints the menu and does what
    // the user asks until they quit.
    //-----------------------------------------------------------------
    public static void main(String[] args)
    {
      scan = new Scanner(System.in);
      printMenu();
      int choice = scan.nextInt();
      while (choice != 0)
          {
            dispatch(choice);
            printMenu();
            choice = scan.nextInt();

          }
    }

    //---------------------------------------
    //  Does what the menu item calls for.
    //---------------------------------------
    public static void dispatch(int choice)
    {
      int newVal;
      switch(choice)
          {
          case 0:
            System.out.println("Bye!");
            break;

          case 1:    //add to front
            System.out.println("Enter integer to add to front");
            newVal = scan.nextInt();
            list.addToFront(newVal);
            break;

          case 2:   //add to end
            System.out.println("Enter integer to add to end");
            newVal = scan.nextInt();
            list.addToEnd(newVal);
            break;

          case 3:  //remove first element
            list.removeFirst();
            break;

          case 4:  //print
```

```
            list.print();
            break;
          default:
            System.out.println("Sorry, invalid choice");
          }
    }

    //----------------------------------------
    //  Prints the user's choices
    //----------------------------------------
    public static void printMenu()
    {
      System.out.println("\n   Menu   ");
      System.out.println("   ====");
      System.out.println("0: Quit");
      System.out.println("1: Add an integer to the front of the list");
      System.out.println("2: Add an integer to the end of the list");
      System.out.println("3: Remove an integer from the front of the list");
      System.out.println("4: Print the list");

      System.out.print("\nEnter your choice: ");
    }
}
```

Recursive Processing of Linked Lists

File *IntList.java* contains definitions for a linked list of integers (see previous exercise). The class contains an inner class *IntNode*, which holds information for a single node in the list (a node has a value and a reference to the next node) and the following *IntList* methods:

- public IntList()—constructor; creates an empty list of integers
- public void addToFront(int val)—takes an integer and puts it on the front of the list
- public void addToEnd(int val)—takes an integer and puts it on the end of the list
- public void removeFirst()—removes the first value from the list
- public void print()—prints the elements in the list from first to last

File *IntListTest.java* contains a driver that allows you to experiment with these methods. Save both of these files to your directory. If you have not already worked with these files in a previous exercise, compile and run IntListTest and play around with it to see how it works. Then add the following methods to the IntList class. For each, add an option in the driver to test the method.

1. public void printRec()—prints the list from first to last using recursion. **Hint:** The basic idea is that you print the first item in the list then do a recursive call to print the rest of the list. This means you need to keep track of what hasn't been printed yet (the "rest" of the list). In particular, your recursive method needs to know where the first item is. Note that printRec() has no parameter so you need to use a helper method that does most of the work. It should have a parameter that lets it know where the part of the list to be printed starts.

2. public void printRecBackwards()—prints the list from last to first using recursion. **Hint:** Printing backward recursively is just like printing forward recursively except you print the rest of the list *before* printing this element. Simple!

A Linked List of Objects

Listing 12.2 in the text is an example of a linked list of objects of type Magazine; the file IntList.java contains an example of a linked list of integers (see previous exercise). A list of objects is a lot like a list of integers or a particular type of object such as a Magazine, except the value stored is an Object, not an int or Magazine. Write a class ObjList that contains arbitrary objects and that has the following methods:

- public void addToFront (Object obj)—puts the object on the front of the list
- public void addToEnd (Object obj)—puts the object on the end of the list
- public void removeFirst()—removes the first value from the list
- public void removeLast()—removes the last value from the list
- public void print()—prints the elements of the list from first to last

These methods are similar to those in IntList. Note that you won't have to write all of these again; you can just make very minor modifications to the IntList methods.

Also write an ObjListTest class that creates an ObjList and puts various different kinds of objects in it (String, array, etc) and then prints it.

Doubly Linked Lists

Sometimes it is convenient to maintain references to both the next node and the previous node in a linked list. This is called a *doubly linked list* and is illustrated in Figure 12.4 of the text. File *DoubleLinked.java* contains definitions for a doubly linked list of integers. This class contains an inner class *IntNode* that holds information for a single node in the list (its value and references to the next and previous nodes). The DoubleLinked class also contains the following methods:

- public DoubleLinked()—constructor; creates an empty list of integers
- public void addToFront(int val)—takes an integer and puts it on the front of the list
- public void print()—prints the elements in the list from first to last

File *DoubleLinkedTest.java* contains a driver that allows you to experiment with these methods. Save both of these files to your directory, compile and run DoubleLinkedTest, and play around with it to see how it works. Then add the following methods to the DoubleLinked class. For each, add an option to the driver to test it.

1. public void addToEnd(int val)—takes an integer and puts it on the end of the list
2. public void removeFirst()—removes the first value from the list. If the list is empty, does nothing.
3. public void removeLast()—removes the last element of the list. If the list is empty, does nothing.
4. public void remove(int oldVal)—removes the first occurrence of oldVal in the list.

```
// ************************************************************
//   DoubleLinked.java
//
//   A class using a doubly linked list to represent a list of integers.
//
// ************************************************************
public class DoubleLinked
{
    private IntNode list;

    // -----------------------------------------------
    // Constructor -- initializes list
    // -----------------------------------------------
    public DoubleLinked()
    {
      list = null;
    }

    // -----------------------------------------------
    // Prints the list elements
    // -----------------------------------------------
    public void print()
    {
      for (IntNode temp = list; temp != null; temp = temp.next)
          System.out.println(temp.val);
    }

    // -----------------------------------------------
    // Adds new element to front of list
    // -----------------------------------------------
    public void addToFront(int val)
    {
      IntNode newNode = new IntNode(val);
      newNode.next = list;
      if (list != null)
            list.prev = newNode;
      list = newNode;
    }
```

```
    //*************************************************************
    // An inner class that represents a list element.
    //*************************************************************

    private class IntNode
    {
        public int val;
        public IntNode next;
        public IntNode prev;

        public IntNode(int val)
        {
          this.val = val;
          this.next = null;
          this.prev = null;
        }
    }
}

// *************************************************************
//   DoubleLinkedTest.java
//
//   Driver to test DoubleLinked methods.
// *************************************************************

import java.util.Scanner;

public class DoubleLinkedTest
{
    private static Scanner scan;
    private static DoubleLinked list = new DoubleLinked();

    //-----------------------------------------------------------------
    // Creates a list, then repeatedly prints the menu and does what
    // the user asks until they quit.
    //-----------------------------------------------------------------
    public static void main(String[] args)
    {
      scan = new Scanner(System.in);
      printMenu();
      int choice = scan.nextInt();
      while (choice != 0)
          {
            dispatch(choice);
            printMenu();
            choice = scan.nextInt();

          }
    }

    //---------------------------------------
    //  Does what the menu item calls for.
    //---------------------------------------
    public static void dispatch(int choice)
    {
      int newVal;
      switch(choice)
          {
          case 0:
```

```
            System.out.println("Bye!");
            break;

          case 1:  //print
            System.out.println("** List elements **");
            list.print();
            break;

          case 2:    //add to front
            System.out.println("Enter integer to add to front");
            newVal = scan.nextInt();
            list.addToFront(newVal);
            break;

          default:
            System.out.println("Sorry, invalid choice");
          }
      }

      //---------------------------------------
      //  Prints the user's choices
      //---------------------------------------
      public static void printMenu()
      {
        System.out.println("\n   Menu   ");
        System.out.println("   ====");
        System.out.println("0: Quit");
        System.out.println("1: Print list");
        System.out.println("2: Add an integer to the front of the list");
        System.out.print("\nEnter your choice: ");
      }
}
```

An Array Queue Implementation

File *QueueADT.java* contains a Java interface representing a queue ADT. In addition to *enqueue(), dequeue(),* and *isEmpty(),* this interface contains two methods that are not described in the book – *isFull ()* and *size().* File *ArrayQueue.java* contains a skeleton for an array-based implementation of this interface; it also includes a *toString()* method that returns a string containing the queue elements, one per line. File *TestQueue.java* contains a simple test program.

Complete the method definitions in *ArrayQueue.java*. Some things to think about:

- A queue has activity at both ends -- elements are enqueued at one end and dequeued from the other end. In an array implementation this means that repeated enqueues and dequeues will shift the queue elements from the beginning to the end of the array, so the array may appear full (in that the last element is in the last slot) when there are actually spaces available at the beginning. To address this the next element can simply be placed in the first element of the array, so that the queue "wraps around" the array. This is called a *circular* array implementation, and is used because it allows the enqueue and dequeue methods to be implemented efficiently in both space and time.
- You'll need to use integers to keep track of the indices of the front and back of the queue. Think carefully about what initial values these variables (*front* and *back*) should get in the constructor and how they should be incremented given the circular nature of the implementation.
- The easiest way to implement the *size()* method is to keep track of the number of elements as you go with the *numElements* variable -- just increment this variable when you enqueue an element and decrement it when you dequeue an element.
- An easy way to tell if a queue is full in an array implementation is to check how many elements it contains (stored in *numElements*). If it's equal to the size of the array, the queue is full. You can also use *numElements* to check if the queue is empty.
- The test program given tries to enqueue more elements than will fit in default size of the queue. Be sure that you check if the queue is full before enqueueing, and if it is full just do nothing. It's safest to do this in the enqueue() method.

Study the code in *TestQueue.java* so you know what it is doing, then compile and run it. Correct any problems in your Linked Queue class.

```
//********************************************************
// QueueADT.java
// The classic FIFO queue interface.
//********************************************************
public interface QueueADT
{
    //-----------------------------------------
    // Puts item on end of queue.
    //-----------------------------------------
    public void enqueue(Object item);

    //-----------------------------------------
    // Removes and returns object from front of queue.
    //-----------------------------------------
    public Object dequeue();

    //-----------------------------------------
    // Returns true if queue is empty.
    //-----------------------------------------
    public boolean isEmpty();

    //-----------------------------------------
    // Returns true if queue is full.
    //-----------------------------------------
    public boolean isFull();
```

```
    //-----------------------------------------
    // Returns the number of elements in the queue.
    //-----------------------------------------
    public int size();
}


//*********************************************************
// ArrayQueue.java
// An array-based implementation of the classic FIFO queue interface.
//*********************************************************

public class ArrayQueue implements QueueADT
{
    private final int DEFAULT_SIZE = 5;
    private Object[] elements;
    private int numElements;
    private int front, back;

    //-----------------------------------------
    // Constructor; creates array of default size.
    //-----------------------------------------
    public ArrayQueue()
    {
    }


    //-----------------------------------------
    // Puts item on end of queue.
    //-----------------------------------------
    public void enqueue(Object item)
    {
    }


    //-----------------------------------------
    // Removes and returns object from front of queue.
    //-----------------------------------------
    public Object dequeue()
    {
    }


    //-----------------------------------------
    // Returns true if queue is empty.
    //-----------------------------------------
    public boolean isEmpty()
    {
    }


    //-----------------------------------------
    // Returns true if queue is full, but it never is.
    //-----------------------------------------
    public boolean isFull()
    {
    }


    //-----------------------------------------
```

```
    // Returns the number of elements in the queue.
    //-----------------------------------------
    public int size()
    {
    }


    //-----------------------------------------
    // Returns a string containing the elements of the queue
    // from first to last
    //-----------------------------------------
    public String toString()
    {
      String result = "\n";
      for (int i = front, count=0; count < numElements;
                        i=(i+1)%elements.length,count++)
            result = result elements[i]+ "\n";
      return result;

    }
}


//*********************************************************
// TestQueue
// A driver to test the methods of the QueueADT implementations.
//*********************************************************
public class TestQueue
{
    public static void main(String[] args)
    {
      QueueADT q = new ArrayQueue();

      System.out.println("\nEnqueuing chocolate, cake, pie, truffles:");
      q.enqueue("chocolate");
      q.enqueue("cake");
      q.enqueue("pie");
      q.enqueue("truffles");

      System.out.println("\nHere's the queue: " + q);
      System.out.println("It contains " + q.size() + " items.");

      System.out.println("\nDequeuing two...");
      System.out.println(q.dequeue());
      System.out.println(q.dequeue());

      System.out.println("\nEnqueuing cookies, profiteroles, mousse, cheesecake,
ice cream:");
      q.enqueue("cookies");
      q.enqueue("profiteroles");
      q.enqueue("mousse");
      q.enqueue("cheesecake");
      q.enqueue("ice cream");

      System.out.println("\nHere's the queue again: " + q);
      System.out.println("Now it contains "  + q.size() + " items.");

      System.out.println("\nDequeuing everything in queue");
      while (!q.isEmpty())
```

```
            System.out.println(q.dequeue());

        System.out.println("\nNow it contains "  + q.size() + " items.");
        if (q.isEmpty())
            System.out.println("Queue is empty!");
        else
            System.out.println("Queue is not empty -- why not??!!");
    }
}
```

A Linked Queue Implementation

File *QueueADT.java* contains a Java interface representing a queue ADT. In addition to *enqueue()*, *dequeue()*, and *isEmpty()*, this interface contains two methods that are not described in the book – *isFull ()* and *size()*. File *LinkedQueue.java* contains a skeleton for a linked implementation of this interface; it also includes a *toString()* method that returns a string containing the queue elements, one per line. It depends on the Node class in *Node.java*. (This could also be defined as an inner class.) File *TestQueue.java* contains a simple test program.

Complete the method definitions in *LinkedQueue.java*. Some things to think about:

- In *enqueue()* and *dequeue()* you have to maintain both the front and back pointers – this takes a little thought. In particular, in *enqueue* be careful of the case where the queue is empty and you are putting the first item in. This case requires special treatment (think about why).
- The easiest way to implement the *size()* method is to keep track of the number of elements as you go with the *numElements* variable -- just increment this variable when you enqueue an element and decrement it when you dequeue an element.
- A linked queue is never full, so *isFull()* always returns false. Easy!

Study the code in *TestQueue.java* so you know what it is doing, then compile and run it. Correct any problems in your Linked Queue class.

```
//********************************************************
// QueueADT.java
// The classic FIFO queue interface.
//********************************************************
public interface QueueADT
{
    //-----------------------------------------
    // Puts item on end of queue.
    //-----------------------------------------
    public void enqueue(Object item);

    //-----------------------------------------
    // Removes and returns object from front of queue.
    //-----------------------------------------
    public Object dequeue();

    //-----------------------------------------
    // Returns true if queue is empty.
    //-----------------------------------------
    public boolean isEmpty();

    //-----------------------------------------
    // Returns true if queue is full.
    //-----------------------------------------
    public boolean isFull();

    //-----------------------------------------
    // Returns the number of elements in the queue.
    //-----------------------------------------
    public int size();
}

//********************************************************
// LinkedQueue.java
// A linked-list implementation of the classic FIFO queue interface.
//********************************************************
public class LinkedQueue implements QueueADT
```

```
{
    private Node front, back;
    private int numElements;

    //-----------------------------------------
    // Constructor; initializes the front and back pointers
    // and the number of elements.
    //-----------------------------------------
    public LinkedQueue()
    {
    }

    //-----------------------------------------
    // Puts item on end of queue.
    //-----------------------------------------
    public void enqueue(Object item)
    {
    }

    //-----------------------------------------
    // Removes and returns object from front of queue.
    //-----------------------------------------
    public Object dequeue()
    {
      Object item = null;
    }

    //-----------------------------------------
    // Returns true if queue is empty.
    //-----------------------------------------
    public boolean isEmpty()
    {
    }

    //-----------------------------------------
    // Returns true if queue is full, but it never is.
    //-----------------------------------------
    public boolean isFull()
    {
    }

    //-----------------------------------------
    // Returns the number of elements in the queue.
    //-----------------------------------------
    public int size()
    {
    }

    //-----------------------------------------
    // Returns a string containing the elements of the queue
    // from first to last
    //-----------------------------------------
    public String toString()
    {
      String result = "\n";
      Node temp = front;
      while (temp != null)
      {
            result += temp.getElement() + "\n";
            temp = temp.getNext();
```

```
      }
      return result;
    }
}


//*************************************************************
//  Node.java
//  A general node for a singly linked list of objects.
//*************************************************************
public class Node
{
    private Node next;
    private Object element;

    //-----------------------------------------------------
    // Creates an empty node
    //-----------------------------------------------------
    public Node()
    {
      next = null;
      element = null;
    }

    //-----------------------------------------------------
    // Creates a node storing a specified element
    //-----------------------------------------------------
    public Node(Object element)
    {
      next = null;
      this.element = element;
    }

    //-----------------------------------------------------
    // Returns the node that follows this one
    //-----------------------------------------------------
    public Node getNext()
    {
      return next;
    }

    //-----------------------------------------------------
    // Sets the node that follows this one
    //-----------------------------------------------------
    public void setNext(Node node)
    {
      next = node;
    }

    //-----------------------------------------------------
    // Returns the element stored in this node
    //-----------------------------------------------------
    public Object getElement()
    {
      return element;
    }

    //-----------------------------------------------------
    // Sets the element stored in this node
    //-----------------------------------------------------
```

```
    public void setElement(Object element)
    {
      this.element = element;
    }
}


//********************************************************
// TestQueue
// A driver to test the methods of the QueueADT implementations.
//********************************************************
public class TestQueue
{
    public static void main(String[] args)
    {
      QueueADT q = new LinkedQueue();

      System.out.println("\nEnqueuing chocolate, cake, pie, truffles:");
      q.enqueue("chocolate");
      q.enqueue("cake");
      q.enqueue("pie");
      q.enqueue("truffles");

      System.out.println("\nHere's the queue: " + q);
      System.out.println("It contains " + q.size() + " items.");

      System.out.println("\nDequeuing two...");
      System.out.println(q.dequeue());
      System.out.println(q.dequeue());

      System.out.println("\nEnqueuing cookies, profiteroles, mousse, cheesecake,
ice cream:");
      q.enqueue("cookies");
      q.enqueue("profiteroles");
      q.enqueue("mousse");
      q.enqueue("cheesecake");
      q.enqueue("ice cream");

      System.out.println("\nHere's the queue again: " + q);
      System.out.println("Now it contains "  + q.size() + " items.");

      System.out.println("\nDequeuing everything in queue");
      while (!q.isEmpty())
          System.out.println(q.dequeue());

      System.out.println("\nNow it contains "  + q.size() + " items.");
      if (q.isEmpty())
          System.out.println("Queue is empty!");
      else
          System.out.println("Queue is not empty -- why not??!!");
    }
}
```

Queue Manipulation

The file *QueueTest.java* contains a printQueue method that takes an object of type QueueADT and prints its contents, restoring the queue before it returns. It uses a temporary queue that actually holds the same information as the original queue. If you know the number of elements in the queue, you can write a printQueue method that prints the queue and restores it to its original form without using an auxiliary data structure (stack, queue, etc.). Think about how, then do it! That is, modify the printQueue method in QueueTest so that it behaves exactly as it does now but does not require an auxiliary data structure. Note that this code uses a LinkedQueue implementation for the QueueADT (see previous exercises), but you could substitute an ArrayQueue if you like.

```
// ****************************************************************
//   QueueTest.java
//
//   A simple driver to manipulate a queue.
//
// ****************************************************************

public class QueueTest
{
    public static void main(String[] args)
    {

      QueueADT queue = new LinkedQueue();

      //put some stuff in the queue: 0,2,4,..,14
      for (int i=0; i<8; i++)
          queue.enqueue(i*2);
      System.out.println("\n\n** Initial queue **");
      printQueue(queue);

      //dequeue 4 items
      for (int i=0; i<4; i++)
          queue.dequeue();
      System.out.println("\n\n** After dequeueing 4 items **");
      printQueue(queue);

      //enqueue 7 more: 1,2,..,7
      for (int i=0; i<7; i++)
          queue.enqueue(i+1);
      System.out.println("\n\n** After enqueueing 7 more items **");
      printQueue(queue);

    }

    //-----------------------------------------------------------
    // Prints elements of queue, restoring it before returning
    //-----------------------------------------------------------
    public static void printQueue(QueueADT queue)
    {
      QueueADT temp = new LinkedQueue();

      //print everything in the queue, putting elements
      //back into a temporary queue
      while (!queue.isEmpty())
          {
            int val = queue.dequeue();
            temp.enqueue(val);
            System.out.print(val + " ");
```

```
            }
        System.out.println ();

        //restore the original queue
        while (!temp.isEmpty())
            {
              int val = temp.dequeue();
              queue.enqueue(val);
            }
    }

}
```

An Array Stack Implementation

Java has a Stack class that holds elements of type Object. However, many languages do not provide stack types, so it is useful to be able to define your own. File *StackADT.java* contains an interface representing the ADT for a stack of objects and *ArrayStack.java* contains a skeleton for a class that uses an array to implement this interface. Fill in code for the following public methods:

- void push(Object val)
- int pop()
- boolean isEmpty()
- boolean isFull()

In writing your methods, keep in mind the following:

- The bottom of an array-based stack is always the first element in the array. In the skeleton given, variable *top* holds the index of the location where the next value pushed will go. So when the stack is empty, *top* is 0; when it contains one element (in location 0 of the array), *top* is 1, and so on.
- Make *push* check to see if the array is full first, and do nothing if it is. Similarly, make *pop* check to see if the array is empty first, and return null if it is.
- Popping an element removes it from the stack, but not from the array—only the value of *top* changes.

File *StackTest.java* contains a simple driver to test your stack. Save it to your directory, compile it, and make sure it works. Note that it tries to push more things than will fit on the stack, but your *push* method should deal with this.

```
// ****************************************************************
//   StackADT.java
//   The classic Stack interface.
// ****************************************************************
public interface StackADT
{
    // -------------------------------------------------
    // Adds a new element to the top of the stack.
    // -------------------------------------------------
    public void push(Object val);

    // -------------------------------------------------
    // Removes and returns the element at the top of the stack.
    // -------------------------------------------------
    public Object pop();

    // -------------------------------------------------
    // Returns true if stack is empty, false otherwise.
    // -------------------------------------------------
    public boolean isEmpty();

    // -------------------------------------------------
    // Returns true if stack is full, false otherwise.
    // -------------------------------------------------
    public boolean isFull();
}
```

```
// ****************************************************************
//   ArrayStack.java
//
//   An array-based Object stack class with operations push,
//   pop, and isEmpty and isFull.
//
// ****************************************************************
```

```
public class ArrayStack implements StackADT
{
    private int stackSize = 5;   // capacity of stack
    private int top;             // index of slot for next element
    private Object[] elements;

    // ------------------------------------------------
    // Constructor -- initializes top and creates array
    // ------------------------------------------------
    public ArrayStack()
    {

    }

    // ------------------------------------------------
    // Adds element to top of stack if it's not full, else
    // does nothing.
    // ------------------------------------------------
    public void push(Object val)
    {

    }

    // ------------------------------------------------
    // Removes and returns value at top of stack.  If stack
    // is empty returns null.
    // ------------------------------------------------
    public Object pop()
    {

    }

    // ------------------------------------------------
    // Returns true if stack is empty, false otherwise.
    // ------------------------------------------------
    public boolean isEmpty()
    {

    }

    // ------------------------------------------------
    // Returns true if stack is full, false otherwise.
    // ------------------------------------------------
    public boolean isFull()
    {

    }

}
```

```
// ******************************************************
// StackTest.java
//
// A simple driver that exercises push, pop, isFull and isEmpty.
// Thanks to autoboxing, we can push integers onto a stack of Objects.
//
// ******************************************************

public class StackTest
{
    public static void main(String[] args)
    {
      StackADT stack = new ArrayStack();

      //push some stuff on the stack
      for (int i=0; i<6; i++)
          stack.push(i*2);

      //pop and print
      //should print 8 6 4 2 0
      while (!stack.isEmpty())
          System.out.print(stack.pop() + " ");
      System.out.println();


      //push a few more things
      for (int i=1; i<=6; i++)
          stack.push(i);

      //should print 5 4 3 2 1
      while (!stack.isEmpty())
          System.out.print(stack.pop() + " ");
      System.out.println();

    }
}
```

A Linked Stack Implementation

Java has a Stack class that holds elements of type Object. However, many languages do not provide stack types, so it is useful to be able to define your own. File *StackADT.java* contains an interface representing the ADT for a stack of objects and *LinkedStack.java* contains a skeleton for a class that uses a linked list to implement this interface. It depends on the Node class in *Node.java*. (This could also be defined as an inner class.) Fill in code for the following public methods:

- void push(Object val)
- int pop()
- boolean isEmpty()
- boolean isFull()

In writing your methods, keep in mind that in a linked implementation of a stack, the top of stack is always at the front of the list. This makes it easy to add (push) and remove (pop) elements.

File *StackTest.java* contains a simple driver to test your stack. Save it to your directory, compile it, and make sure it works.

```
// *************************************************************
//   StackADT.java
//   The classic Stack interface.
// *************************************************************
public interface StackADT
{
    // ------------------------------------------------
    // Adds a new element to the top of the stack.
    // ------------------------------------------------
    public void push(Object val);

    // ------------------------------------------------
    // Removes and returns the element at the top of the stack.
    // ------------------------------------------------
    public Object pop();

    // ------------------------------------------------
    // Returns true if stack is empty, false otherwise.
    // ------------------------------------------------
    public boolean isEmpty();

    // ------------------------------------------------
    // Returns true if stack is full, false otherwise.
    // ------------------------------------------------
    public boolean isFull();
}
```

```
// *************************************************************
//   LinkedStack.java
//
//   An linked implementation of an Object stack class with operations push,
//   pop, and isEmpty and isFull.
//
// *************************************************************
public class LinkedStack implements StackADT
{
    private Node top;             // reference to top of stack

    // ------------------------------------------------
```

```
    // Constructor -- initializes top
    // -----------------------------------------------
    public LinkedStack()
    {
    }

    // -----------------------------------------------
    // Adds element to top of stack if it's not full, else
    // does nothing.
    // -----------------------------------------------
    public void push(Object val)
    {
    }

    // -----------------------------------------------
    // Removes and returns value at top of stack.  If stack
    // is empty returns null.
    // -----------------------------------------------
    public Object pop()
    {
    }

    // -----------------------------------------------
    // Returns true if stack is empty, false otherwise.
    // -----------------------------------------------
    public boolean isEmpty()
    {
    }

    // -----------------------------------------------
    // Returns true if stack is full, false otherwise.
    // -----------------------------------------------
    public boolean isFull()
    {
    }
}


//*************************************************************
// Node.java
// A general node for a singly linked list of objects.
//*************************************************************
public class Node
{
    private Node next;
    private Object element;

    //-------------------------------------------------------
    // Creates an empty node
    //-------------------------------------------------------
    public Node()
    {
      next = null;
      element = null;
    }

    //-------------------------------------------------------
    // Creates a node storing a specified element
    //-------------------------------------------------------
    public Node(Object element)
```

```
    {
      next = null;
      this.element = element;
    }

    //-----------------------------------------------------
    // Returns the node that follows this one
    //-----------------------------------------------------
    public Node getNext()
    {
      return next;
    }

    //-----------------------------------------------------
    // Sets the node that follows this one
    //-----------------------------------------------------
    public void setNext(Node node)
    {
      next = node;
    }

    //-----------------------------------------------------
    // Returns the element stored in this node
    //-----------------------------------------------------
    public Object getElement()
    {
      return element;
    }

    //-----------------------------------------------------
    // Sets the element stored in this node
    //-----------------------------------------------------
    public void setElement(Object element)
    {
      this.element = element;
    }
}


// ********************************************************
// StackTest.java
//
// A simple driver that exercises push, pop, isFull and isEmpty.
// Thanks to autoboxing, we can push integers onto a stack of Objects.
//
// ********************************************************

public class StackTest
{
    public static void main(String[] args)
    {
      StackADT stack = new LinkedStack();

      //push some stuff on the stack
      for (int i=0; i<10; i++)
          stack.push(i*2);

      //pop and print
      //should print 18 16 14 12 10 8 6 4 2 0
      while (!stack.isEmpty())
```

```
            System.out.print(stack.pop() + " ");
        System.out.println();


        //push a few more things
        for (int i=1; i<=6; i++)
            stack.push(i);

        //should print 5 4 3 2 1
        while (!stack.isEmpty())
            System.out.print(stack.pop() + " ");
        System.out.println();

    }
}
```

Stack Manipulation

Sometimes it's useful to define operations on an ADT without changing the type definition itself. For example, you might want to print the elements in a stack without actually adding a method to the Stack ADT (you may not even have access to it). To explore this, use either the Stack class provided by Java (in java.util) or one of the stack classes that you wrote in an earlier lab exercise and the test program *StackTest.java*. Add the following static methods to the StackTest class (the signature for these methods and the declaration in StackTest assumes you are using a stack class named Stack—modify them to use the name of your class):

- void printStack(Stack s)—prints the elements in stack s from top to bottom. When printStack returns, s should be unchanged.
- Stack reverseStack(Stack s)—returns a new stack whose elements are backwards from those in s. Again, s is unchanged.
- Stack removeElement(Stack s, int val)—returns a new stack whose elements are the same as those in s (and in the same order) except that all occurrences of val have been removed. Again, s is unchanged.

Modify the main method to test these methods. Be sure you print enough information to see if they're working!

```
// ****************************************************************
//   StackTest.java
//
//   A simple driver to test a stack.
//
// ****************************************************************
import java.util.Stack;
public class StackTest
{
    public static void main(String[] args)
    {
      // Declare and instantiate a stack
      Stack stack = new Stack();

      //push some stuff on the stack
      for (int i=0; i<10; i++)
          stack.push(i);
      stack.push(5);

      // call printStack to print the stack

      // call reverseStack to reverse the stack

      // call printStack to print the stack again

      // call removeElement to remove all occurrences of the value 5 - save the
      // stack returned from this call

      // call printStack to print the original stack and the new stack.

    }
}
```

Matching Parentheses

One application of stacks is to keep track of things that must match up such as parentheses in an expression or braces in a program. In the case of parentheses when a left parenthesis is encountered it is pushed on the stack and when a right parenthesis is encountered its matching left parenthesis is popped from the stack. If the stack has no left parenthesis, that means the parentheses don't match—there is an extra right parenthesis. If the expression ends with at least one left parenthesis still on the stack then again the parentheses don't match—there is an extra left parenthesis.

File *ParenMatch.java* contains the skeleton of a program to match parentheses in an expression. It uses the Stack class provided by Java (in java.util). Complete the program by adding a loop to process the line entered to see if it contains matching parentheses. Just ignore characters that are neither left nor right parentheses. Your loop should stop as soon as it detects an error. After the loop print a message indicating what happened—the parentheses match, there are too many left parentheses, or there are too many right parentheses. Also print the part of the string up to where the error was detected.

```
// ********************************************************************
//   ParenMatch.java
//
//   Determines whether or not a string of characters contains
//   matching left and right parentheses.
// ********************************************************************

import java.util.*;
import java.util.Scanner;

public class ParenMatch
{
    public static void main (String[] args)
    {
      Stack s = new Stack();
      String line;             // the string of characters to be checked
      Scanner scan = new Scanner(System.in);

      System.out.println ("\nParenthesis Matching");
      System.out.print ("Enter a parenthesized expression: ");
      line = scan.nextLine();


      // loop to process the line one character at a time

      // print the results


    }
}
```