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.

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??!!");
    }
}
```

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();
    }
}
```

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

    }
}
```