17

Data Structures

Much that I bound, I could not free;
Much that I freed returned to me.
—Lee Wilson Dodd

'Will you walk a little faster?' said a whiting to a snail,
'There's a porpoise close behind us, and he's treading on my tail.'
—Lewis Carroll

There is always room at the top.
—Daniel Webster

Push on—keep moving.
—Thomas Morton

I'll turn over a new leaf.
—Miguel de Cervantes

OBJECTIVES

In this chapter you will learn:

- To form linked data structures using references, self-referential classes and recursion.
- The type-wrapper classes that enable programs to process primitive data values as objects.
- To use autoboxing to convert a primitive value to an object of the corresponding type-wrapper class.
- To use auto-unboxing to convert an object of a type-wrapper class to a primitive value.
- To create and manipulate dynamic data structures, such as linked lists, queues, stacks and binary trees.
- Various important applications of linked data structures.
- How to create reusable data structures with classes, inheritance and composition.

Self-Review Exercises

17.1 Fill in the blanks in each of the following statements:

a) A self-_____ class is used to form dynamic data structures that can grow and shrink at execution time.

ANS: referential

b) A(n) _____ is a constrained version of a linked list in which nodes can be inserted and deleted only from the start of the list.

ANS: stack

c) A method that does not alter a linked list, but simply looks at it to determine whether it is empty, is referred to as a(n) _____ method.

ANS: predicate

d) A queue is referred to as a(n) _____ data structure because the first nodes inserted are the first ones removed.

ANS: first-in, first-out (FIFO)

e) The reference to the next node in a linked list is referred to as a(n) _____.

ANS: link

f) Automatically reclaiming dynamically allocated memory in Java is called _____.

ANS: garbage collection

g) A(n) _____ is a constrained version of a linked list in which nodes can be inserted only at the end of the list and deleted only from the start of the list.

ANS: queue

h) A(n) _____ is a nonlinear, two-dimensional data structure that contains nodes with two or more links.

ANS: tree

i) A stack is referred to as a(n) _____ data structure because the last node inserted is the first node removed.

ANS: last-in, first-out (LIFO)

j) The nodes of a(n) _____ tree contain two link members.

ANS: binary

k) The first node of a tree is the _____ node.

ANS: root

l) Each link in a tree node refers to a(n) _____ or _____ of that node.

ANS: child or subtree

m) A tree node that has no children is called a(n) _____ node.

ANS: leaf

n) The three traversal algorithms we mentioned in the text for binary search trees are _____, _____ and _____.

ANS: inorder, preorder, postorder

17.2 What are the differences between a linked list and a stack?

ANS: It is possible to insert a node anywhere in a linked list and remove a node from anywhere in a linked list. Nodes in a stack may be inserted only at the top of the stack and removed only from the top.

17.3 What are the differences between a stack and a queue?

ANS: A queue is a FIFO data structure that has references to both its head and its tail, so that nodes may be inserted at the tail and deleted from the head. A stack is a LIFO data structure that has a single reference to the top of the stack, where both insertion and deletion of nodes are performed.

17.4 Perhaps a more appropriate title for this chapter would have been "Reusable Data Structures." Comment on how each of the following entities or concepts contributes to the reusability of data structures:

a) classes

ANS: Classes allow us to instantiate as many data structure objects of a certain type (i.e., class) as we wish.

b) inheritance

ANS: Inheritance enables a subclass to reuse the functionality from a superclass. Public methods of a superclass can be accessed through a subclass to eliminate duplicate logic.

c) composition

ANS: Composition enables a class to reuse code by storing an instance of another class in a field. Public methods of the member class can be called by methods in the composite class.

17.5 Manually provide the inorder, preorder and postorder traversals of the binary search tree of Fig. 17.20.

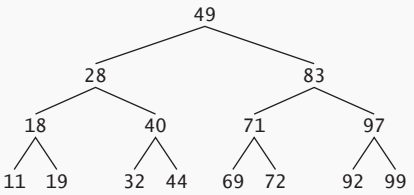

Fig. 17.20 | Binary search tree with 15 nodes.

ANS: The inorder traversal is

```
11 18 19 28 32 40 44 49 69 71 72 83 92 97 99
```

The preorder traversal is

```
49 28 18 11 19 40 32 44 83 71 69 72 97 92 99
```

The postorder traversal is

```
11 19 18 32 44 40 28 69 72 71 92 99 97 83 49
```

Exercises

17.6 Write a program that concatenates two linked list objects of characters. Class `ListConcatenate` should include a method `concatenate` that takes references to both list objects as arguments and concatenates the second list to the first list.

ANS:

```
1  // Exercise 17.6 Solution: List.java
2  // ListNode and List class definitions.
3  package com.deitel.jhtp7.ch17;
4
5  // class to represent one node in a list
6  class ListNode
7  {
8     // package access members; List can access these directly
```

```
9     Object data;
10    ListNode nextNode;
11
12    // constructor creates a ListNode that refers to object
13    ListNode( Object object )
14    {
15       this( object, null );
16    } // end ListNode one-argument constructor
17
18    // constructor creates ListNode that refers to
19    // Object and to next ListNode
20    ListNode( Object object, ListNode node )
21    {
22       data = object;
23       nextNode = node;
24    } // end ListNode two-argument constructor
25
26    // return reference to data in node
27    Object getObject()
28    {
29       return data; // return Object in this node
30    } // end method getObject
31
32    // return reference to next node in list
33    ListNode getNext()
34    {
35       return nextNode; // get next node
36    } // end method getNext
37 } // end class ListNode
38
39 // class List definition
40 public class List
41 {
42    private ListNode firstNode;
43    private ListNode lastNode;
44    private String name; // string like "list" used in printing
45
46    // constructor creates empty List with "list" as the name
47    public List()
48    {
49       this( "list" );
50    } // end List no-argument constructor
51
52    // constructor creates an empty List with a name
53    public List( String listName )
54    {
55       name = listName;
56       firstNode = lastNode = null;
57    } // end List one-argument constructor
58
59    // insert Object at front of List
60    public void insertAtFront( Object insertItem )
61    {
62       if ( isEmpty() ) // firstNode and lastNode refer to same object
```

```
63            firstNode = lastNode = new ListNode( insertItem );
64         else // firstNode refers to new node
65            firstNode = new ListNode( insertItem, firstNode );
66      } // end method insertAtFront
67
68      // insert Object at end of List
69      public void insertAtBack( Object insertItem )
70      {
71         if ( isEmpty() ) // firstNode and lastNode refer to same Object
72            firstNode = lastNode = new ListNode( insertItem );
73         else // lastNode's nextNode refers to new node
74            lastNode = lastNode.nextNode = new ListNode( insertItem );
75      } // end method insertAtBack
76
77      // remove first node from List
78      public Object removeFromFront() throws EmptyListException
79      {
80         if ( isEmpty() ) // throw exception if List is empty
81            throw new EmptyListException( name );
82
83         Object removedItem = firstNode.data; // retrieve data being removed
84
85         // update references firstNode and lastNode
86         if ( firstNode == lastNode )
87            firstNode = lastNode = null;
88         else
89            firstNode = firstNode.nextNode;
90
91         return removedItem; // return removed node data
92      } // end method removeFromFront
93
94      // remove last node from List
95      public Object removeFromBack() throws EmptyListException
96      {
97         if ( isEmpty() ) // throw exception if List is empty
98            throw new EmptyListException( name );
99
100        Object removedItem = lastNode.data; // retrieve data being removed
101
102        // update references firstNode and lastNode
103        if ( firstNode == lastNode )
104           firstNode = lastNode = null;
105        else // locate new last node
106        {
107           ListNode current = firstNode;
108
109           // loop while current node does not refer to lastNode
110           while ( current.nextNode != lastNode )
111              current = current.nextNode;
112
113           lastNode = current; // current is new lastNode
114           current.nextNode = null;
115        } // end else
116
```

```
117        return removedItem; // return removed node data
118     } // end method removeFromBack
119
120     // determine whether list is empty
121     public boolean isEmpty()
122     {
123        return firstNode == null; // return true if List is empty
124     } // end method isEmpty
125
126     // output List contents
127     public void print()
128     {
129        if ( isEmpty() )
130        {
131           System.out.printf( "Empty %s\n", name );
132           return;
133        } // end if
134
135        System.out.printf( "The %s is: ", name );
136        ListNode current = firstNode;
137
138        // while not at end of list, output current node's data
139        while ( current != null )
140        {
141           System.out.printf( "%s ", current.data );
142           current = current.nextNode;
143        } // end while
144
145        System.out.println( "\n" );
146     } // end method print
147
148     // concatenates two lists and stores the results in the first list
149     public void concatenate( List two )
150     {
151        while ( !two.isEmpty() )
152           insertAtBack( two.removeFromFront() );
153     } // end method concatenate
154  } // end class List
```

```
1  // Exercise 17.6 Solution: ListConcatenate.java
2  // Program concatenates two lists
3  import com.deitel.jhtp7.ch17.List;
4
5  public class ListConcatenate
6  {
7     public static void main( String args[] )
8     {
9        // create two linked lists
10       List list1 = new List();
11       List list2 = new List();
12
13       // use List insert methods to add characters to list1
14       System.out.println( "List 1:" );
```

```
15         list1.insertAtFront( '5' );
16         list1.print();
17         list1.insertAtFront( '@' );
18         list1.print();
19         list1.insertAtBack( 'V' );
20         list1.print();
21         list1.insertAtBack( '+' );
22         list1.print();
23
24         // use List insert methods to add character to list2
25         System.out.println( "List 2:" );
26         list2.insertAtFront( 'P' );
27         list2.print();
28         list2.insertAtFront( 'c' );
29         list2.print();
30         list2.insertAtBack( 'M' );
31         list2.print();
32         list2.insertAtBack( '&' );
33         list2.print();
34
35         // concatenate lists using method concatenate
36         list1.concatenate( list2 );
37         System.out.println( "Concatenated list is:" );
38         list1.print();
39      } // end main
40   } // end class ListConcatenate
```

```
List 1:
The list is: 5

The list is: @ 5

The list is: @ 5 V

The list is: @ 5 V +

List 2:
The list is: P

The list is: c P

The list is: c P M

The list is: c P M &

Concatenated list is:
The list is: @ 5 V + c P M &
```

17.7 Write a program that merges two ordered list objects of integers into a single ordered-list object of integers. Method `merge` of class `ListMerge` should receive references to each of the list objects to be merged and return a reference to the merged list object.

ANS:

```
1  // Exercise 17.7 Solution: MergeList.java
2  // Program merges two ordered integer list.
3  package com.deitel.jhtp7.ch17;
4
5  public class MergeList extends List
6  {
7     List list1; // list1 and list2 are two
8     List list2; // lists to be merged
9     List mergedList; // merged list of list1 and list2
10
11    // constructor merges two lists
12    public MergeList( List first, List second )
13    {
14       list1 = first;
15       list2 = second;
16       mergedList = new List();
17       merge();
18    } // end three-argument MergeList constructor
19
20    // merge two ordered list
21    private void merge()
22    {
23       // get min of two list element
24       Integer smallElement = findMin();
25
26       // insert into return list
27       while ( smallElement != null )
28       {
29          mergedList.insertAtBack( smallElement );
30          smallElement = findMin();
31       } // end while
32    } // end method merge
33
34    // find min of two list
35    private Integer findMin()
36    {
37       int list1First, list2First;
38
39       // list1 and list2 both are empty
40       if ( list1.isEmpty() && list2.isEmpty() )
41          return null;
42       else if ( list1.isEmpty() ) // list1 is empty
43          return ( Integer ) list2.removeFromFront();
44       else if ( list2.isEmpty() ) // list2 is empty
45          return ( Integer ) list1.removeFromFront();
46       else // neither list1 nor list2 is empty
47       {
48          list1First = ( Integer ) list1.removeFromFront();
49          list2First = ( Integer ) list2.removeFromFront();
```

```
50
51          // list1 element is smaller
52          if ( list1First <= list2First )
53          {
54             list2.insertAtFront( list2First ); // put item back to list2
55             return list1First;
56          } // end if
57          else
58          {
59             list1.insertAtFront( list1First ); // put item back to list1
60             return list2First;
61          } // end else
62       } // end else
63    } // end method findMin
64
65    // return merged list
66    public List getMergedList()
67    {
68       return mergedList;
69    } // end method getMergedList
70 } // end class MergeList
```

```
1  // Exercise 17.7 Solution: MergeListTest.java
2  // Program inserts sorted numbers in two lists and merges them
3  import com.deitel.jhtp7.ch17.MergeList;
4  import com.deitel.jhtp7.ch17.List;
5
6  public class MergeListTest
7  {
8     public static void main( String args[] )
9     {
10       List list1 = new List();
11       List list2 = new List();
12       int[] first = { 3, 12, 15, 19, 24, 38, 50, 61, 70, 99 };
13       int[] second = { 5, 7, 10, 25, 33, 67, 78, 100 };
14
15       // add list1 elements to fist
16       for ( int intValue : first )
17          list1.insertAtBack( intValue );
18
19       // add list2 elements to second
20       for ( int intValue : second )
21          list2.insertAtBack( intValue );
22
23       System.out.println( "First list is: " );
24       list1.print();
25
26       System.out.println( "Second list is: " );
27       list2.print();
28
29       // merge two list and display merged list
30       MergeList mergeList = new MergeList( list1, list2 );
31       List mergedList = mergeList.getMergedList();
```

```
32         System.out.println( "Merged list is: " );
33         mergedList.print();
34      } // end main
35   } // end class MergeListTest
```

```
First list is:
The list is: 3 12 15 19 24 38 50 61 70 99

Second list is:
The list is: 5 7 10 25 33 67 78 100

Merged list is:
The list is: 3 5 7 10 12 15 19 24 25 33 38 50 61 67 70 78 99 100
```

17.8 Write a program that inserts 25 random integers from 0 to 100 in order into a linked-list object. The program should calculate the sum of the elements and the floating-point average of the elements.

ANS:

```
1  // Exercise 17.8 Solution: List.java
2  // ListNode and List class definitions.
3  package com.deitel.jhtp7.ch17;
4
5  // class to represent one node in a list
6  class ListNode
7  {
8     // package access members; List can access these directly
9     Object data;
10    ListNode nextNode;
11
12    // constructor creates a ListNode that refers to object
13    ListNode( Object object )
14    {
15       this( object, null );
16    } // end ListNode one-argument constructor
17
18    // constructor creates ListNode that refers to
19    // Object and to next ListNode
20    ListNode( Object object, ListNode node )
21    {
22       data = object;
23       nextNode = node;
24    } // end ListNode two-argument constructor
25
26    // return reference to data in node
27    Object getObject()
28    {
29       return data; // return Object in this node
30    } // end method getObject
31
32    // return reference to next node in list
33    ListNode getNext()
```

```
34       {
35          return nextNode; // get next node
36       } // end method getNext
37    } // end class ListNode
38
39    // class List definition
40    public class List
41    {
42       private ListNode firstNode;
43       private ListNode lastNode;
44       private String name; // string like "list" used in printing
45
46       // constructor creates empty List with "list" as the name
47       public List()
48       {
49          this( "list" );
50       } // end List no-argument constructor
51
52       // constructor creates an empty List with a name
53       public List( String listName )
54       {
55          name = listName;
56          firstNode = lastNode = null;
57       } // end List one-argument constructor
58
59       // insert Object at front of List
60       public void insertAtFront( Object insertItem )
61       {
62          if ( isEmpty() ) // firstNode and lastNode refer to same object
63             firstNode = lastNode = new ListNode( insertItem );
64          else // firstNode refers to new node
65             firstNode = new ListNode( insertItem, firstNode );
66       } // end method insertAtFront
67
68       // insert Object at end of List
69       public void insertAtBack( Object insertItem )
70       {
71          if ( isEmpty() ) // firstNode and lastNode refer to same Object
72             firstNode = lastNode = new ListNode( insertItem );
73          else // lastNode's nextNode refers to new node
74             lastNode = lastNode.nextNode = new ListNode( insertItem );
75       } // end method insertAtBack
76
77       // remove first node from List
78       public Object removeFromFront() throws EmptyListException
79       {
80          if ( isEmpty() ) // throw exception if List is empty
81             throw new EmptyListException( name );
82
83          Object removedItem = firstNode.data; // retrieve data being removed
84
85          // update references firstNode and lastNode
86          if ( firstNode == lastNode )
87             firstNode = lastNode = null;
```

```
88          else
89             firstNode = firstNode.nextNode;
90
91          return removedItem; // return removed node data
92       } // end method removeFromFront
93
94       // remove last node from List
95       public Object removeFromBack() throws EmptyListException
96       {
97          if ( isEmpty() ) // throw exception if List is empty
98             throw new EmptyListException( name );
99
100         Object removedItem = lastNode.data; // retrieve data being removed
101
102         // update references firstNode and lastNode
103         if ( firstNode == lastNode )
104            firstNode = lastNode = null;
105         else // locate new last node
106         {
107            ListNode current = firstNode;
108
109            // loop while current node does not refer to lastNode
110            while ( current.nextNode != lastNode )
111               current = current.nextNode;
112
113            lastNode = current; // current is new lastNode
114            current.nextNode = null;
115         } // end else
116
117         return removedItem; // return removed node data
118      } // end method removeFromBack
119
120      // determine whether list is empty
121      public boolean isEmpty()
122      {
123         return firstNode == null; // return true if List is empty
124      } // end method isEmpty
125
126      // output List contents
127      public void print()
128      {
129         if ( isEmpty() )
130         {
131            System.out.printf( "Empty %s\n", name );
132            return;
133         } // end if
134
135         System.out.printf( "The %s is: ", name );
136         ListNode current = firstNode;
137
138         // while not at end of list, output current node's data
139         while ( current != null )
140         {
141            System.out.printf( "%s ", current.data );
```

```
142          current = current.nextNode;
143       } // end while
144
145       System.out.println( "\n" );
146    } // end method print
147
148    // calculates and returns sum of every value in list
149    public int sum()
150    {
151       int sum = 0;
152       ListNode current = firstNode;
153
154       // cycle through list adding values to total
155       while ( current != null )
156       {
157          sum += ( Integer ) current.getObject();
158          current = current.getNext();
159       } // end while
160
161       return sum;
162    } // end method add
163 } // end class List
```

```
1  // Exercise 17.8 Solution: ListTest.java
2  // Program inserts random numbers in a list,
3  // prints the sum, and displays the average.
4  import java.util.Random;
5  import com.deitel.jhtp7.ch17.List;
6
7  public class ListTest
8  {
9     public static void main( String args[] )
10    {
11       List list = new List();
12       int newNumber;
13       Random randomNumber = new Random();
14
15       // create Integers to store in list
16       for ( int k = 1; k <= 25; k++ )
17       {
18          newNumber = randomNumber.nextInt( 101 );
19          list.insertAtFront( newNumber );
20       } // end for
21
22       list.print();
23
24       int sum = list.sum();
25       System.out.printf(
26          "Sum is: %d\nAverage: %.3f", sum, sum / 25.0f );
27    } // end main
28 } // end class ListTest
```

```
The list is: 36 39 53 94 19 86 52 47 96 75 84 53 78 19 20 2 41 100 10 23 78
21 68 33 77

Sum is: 1304
Average: 52.160
```

17.9 Write a program that creates a linked list object of 10 characters, then creates a second list object containing a copy of the first list, but in reverse order.

ANS:

```
1  // Exercise 17.9 Solution: ListReverse.java
2  // Program creates a list, then creates a reverse of the list
3  import com.deitel.jhtp7.ch17.List;
4
5  public class ListReverse
6  {
7     public static void main( String args[] )
8     {
9        // create two linked lists
10       List list1 = new List();
11       List list2 = new List();
12
13       // use List insert methods
14       list1.insertAtFront( '5' );
15       list1.insertAtFront( '@' );
16       list1.insertAtBack( 'V' );
17       list1.insertAtBack( '+' );
18       list1.insertAtFront( 'P' );
19       list1.insertAtFront( 'c' );
20       list1.insertAtBack( 'M' );
21       list1.insertAtBack( '&' );
22       list1.insertAtFront( 'y' );
23       list1.insertAtBack( 'X' );
24
25       System.out.println( "List: " );
26       list1.print();
27
28       list2 = reverse( list1 ); // reverse lists
29       System.out.println( "Reversed list is:" );
30       list2.print();
31    } // end main
32
33    // reverses a list and returns it to the caller
34    public static List reverse( List one )
35    {
36       List reversed = new List();
37
38       while ( !one.isEmpty() )
39          reversed.insertAtFront( one.removeFromFront() );
40
41       return reversed;
```

```
42      } // end method
43   } // end class ListReverse
```

```
List:
The list is: y c P @ 5 V + M & X

Reversed list is:
The list is: X & M + V 5 @ P c y
```

17.10 Write a program that inputs a line of text and uses a stack object to print the words of the line in reverse order.

ANS:

```
1   // Exercise 17.10 Solution: StackTest.java
2   // Program prints the words of a line in reverse
3   import java.util.Scanner;
4   import java.util.StringTokenizer;
5   import com.deitel.jhtp7.ch17.StackComposition;
6
7   public class StackTest
8   {
9      public static void main( String args[] )
10     {
11        StackComposition stack = new StackComposition();
12
13        // get input text
14        Scanner scanner = new Scanner( System.in );
15        System.out.println( "Please enter a string" );
16        String input = scanner.nextLine();
17        StringTokenizer tokenizer = new StringTokenizer( input );
18
19        // take each word from tokenizer and push on stack
20        while ( tokenizer.hasMoreTokens() )
21           stack.push( tokenizer.nextToken() );
22
23        System.out.println( "\nInput string in reverse order:" );
24
25        // build reverse string by popping words from stack.
26        while ( !stack.isEmpty() )
27        {
28           Object removedObject = stack.pop();
29           System.out.printf( "%s ", removedObject );
30        } // end while
31     } // end main
32  } // end class StackTest
```

```
Please enter a string
print this string in reverse

Input string in reverse order:
reverse in string this print
```

17.11 Write a program that uses a stack to determine whether a string is a palindrome (i.e., the string is spelled identically backward and forward). The program should ignore spaces and punctuation.

ANS:

```
1  // Exercise 17.11 Solution: StackTest2.java
2  // Program tests for a palindrome.
3  import java.util.Scanner;
4  import com.deitel.jhtp7.ch17.StackComposition;
5  import com.deitel.jhtp7.ch17.EmptyListException;
6
7  public class StackTest2
8  {
9     public static void main( String args[] )
10    {
11       StackComposition stack = new StackComposition();
12
13       // get input string
14       Scanner scanner = new Scanner( System.in );
15       System.out.println( "Please enter a string:" );
16       String input = scanner.nextLine();
17
18       char letter; // character in string
19       Object removedObject = null; // object removed from stack
20       boolean flag = false; // true if string is palindrome
21
22       // cycle through input one char at a time to
23       // create stack of all relevant characters
24       for ( int i = 0; i < input.length(); i++ )
25       {
26          letter = input.charAt( i );
27
28          if ( Character.isLetterOrDigit( letter ) )
29             stack.push( letter );
30       } // end for
31
32       // test for palindrome
33       try
34       {
35          for ( int count = 0;
36             count < input.length() && !stack.isEmpty(); count++ )
37          {
38             letter = input.charAt( count );
39
40             // ignore spaces and punctuation
41             if ( !Character.isLetterOrDigit( letter ) )
42                continue;
43
44             removedObject = stack.pop();
45
46             // not palindrome
47             if ( letter != ( ( Character ) removedObject ) )
48             {
49                flag = true;
```

```
                break;
             } // end if
          } // end for

          // display output
          if ( flag == false )
             System.out.println( "\nThe input string is a palindrome" );
          else
             System.out.println(
                "\nThe input string is not a Palindrome" );
       } // end try
       catch ( EmptyListException exception )
       {
           System.err.printf( "\n%s", exception.toString() );
       } // end catch
    } // end main
 } // end class StackTest2
```

```
Please enter a string:
madam i'm adam

The input string is a palindrome
```

17.12 Stacks are used by compilers to help in the process of evaluating expressions and generating machine-language code. In this and the next exercise, we investigate how compilers evaluate arithmetic expressions consisting only of constants, operators and parentheses.

Humans generally write expressions like `3 + 4` and `7 / 9` in which the operator (`+` or `/` here) is written between its operands—this is called *infix notation*. Computers "prefer" *postfix notation*, in which the operator is written to the right of its two operands. The preceding infix expressions would appear in postfix notation as `3 4 +` and `7 9 /`, respectively.

To evaluate a complex infix expression, a compiler would first convert the expression to postfix notation and evaluate the postfix version. Each of these algorithms requires only a single left-to-right pass of the expression. Each algorithm uses a stack object in support of its operation, but each uses the stack for a different purpose.

In this exercise, you will write a Java version of the infix-to-postfix conversion algorithm. In the next exercise, you will write a Java version of the postfix expression evaluation algorithm. In a later exercise, you will discover that code you write in this exercise can help you implement a complete working compiler.

Write class `InfixToPostfixConverter` to convert an ordinary infix arithmetic expression (assume a valid expression is entered) with single-digit integers such as

```
(6 + 2) * 5 - 8 / 4
```

to a postfix expression. The postfix version of the preceding infix expression is (note that no parentheses are needed)

```
6 2 + 5 * 8 4 / -
```

The program should read the expression into `StringBuffer infix` and use one of the stack classes implemented in this chapter to help create the postfix expression in `StringBuffer postfix`. The algorithm for creating a postfix expression is as follows:

a) Push a left parenthesis `'('` on the stack.
b) Append a right parenthesis `')'` to the end of `infix`.
c) While the stack is not empty, read `infix` from left to right and do the following:
    If the current character in `infix` is a digit, append it to `postfix`.
    If the current character in `infix` is a left parenthesis, push it onto the stack.
    If the current character in `infix` is an operator:
        Pop operators (if there are any) at the top of the stack while they have equal or higher precedence than the current operator, and append the popped operators to `postfix`.
        Push the current character in `infix` onto the stack.
    If the current character in `infix` is a right parenthesis:
        Pop operators from the top of the stack and append them to `postfix` until a left parenthesis is at the top of the stack.
        Pop (and discard) the left parenthesis from the stack.

The following arithmetic operations are allowed in an expression:

- `+` addition
- `-` subtraction
- `*` multiplication
- `/` division
- `^` exponentiation
- `%` remainder

The stack should be maintained with stack nodes that each contain an instance variable and a reference to the next stack node. Some of the methods you may want to provide are as follows:

a) Method `convertToPostfix`, which converts the infix expression to postfix notation.
b) Method `isOperator`, which determines whether `c` is an operator.
c) Method `precedence`, which determines whether the precedence of `operator1` (from the infix expression) is less than, equal to or greater than the precedence of `operator2` (from the stack). The method returns `true` if `operator1` has lower precedence than `operator2`. Otherwise, `false` is returned.
d) Method `stackTop` (this should be added to the stack class), which returns the top value of the stack without popping the stack.

ANS:

```
1  // Exercise 17.12 Solution:InfixToPostfixConverter.java
2  // Infix to postfix conversion
3  import com.deitel.jhtp7.ch17.EmptyListException;
4  import com.deitel.jhtp7.ch17.StackComposition;
5
6  public class InfixToPostfixConverter
7  {
8     // take out the infix and change it into postfix
9     public static StringBuffer convertToPostfix( StringBuffer infix )
10    {
11       CharacterStack charStack = new CharacterStack();
12       StringBuffer temporary = new StringBuffer( "" );
13
14       // push a left paren onto the stack and add a right paren to infix
15       charStack.pushChar( '(' );
16       infix.append( ')' );
17
18       // convert the infix expression to postfix
```

```
19          for ( int infixCount = 0; !charStack.isEmpty(); ++infixCount )
20          {
21             if ( Character.isDigit( infix.charAt( infixCount ) ) )
22                temporary.append( infix.charAt( infixCount ) + " " );
23             else if ( infix.charAt( infixCount ) == '(' )
24                charStack.pushChar( '(' );
25             else if ( isOperator( infix.charAt( infixCount ) ) )
26             {
27                while (  isOperator( charStack.stackTop() ) &&
28                   precedence( charStack.stackTop(),
29                      infix.charAt( infixCount ) ) )
30                   temporary.append( charStack.popChar() + " " );
31
32                charStack.pushChar( infix.charAt( infixCount ) );
33             } // end else if
34             else if ( infix.charAt( infixCount ) == ')' )
35             {
36                while ( charStack.stackTop() != '(' )
37                   temporary.append( charStack.popChar() + " " );
38
39                charStack.popChar();
40             } // end else if
41          } // end for
42
43          return temporary;
44       } // end method convertToPostfix
45
46       // check if c is an operator
47       private static boolean isOperator( char c )
48       {
49          if ( c == '+' || c == '-' || c == '*' || c == '/' || c == '^' )
50             return true;
51          else
52             return false;
53       } // end method isOperator
54
55       // ensure proper order of operations
56       private static boolean precedence( char operator1, char operator2 )
57       {
58          if ( operator1 == '^' )
59             return true;
60          else if ( operator2 == '^' )
61             return false;
62          else if ( operator1 == '*' || operator1 == '/' )
63             return true;
64          else if ( operator1 == '+' || operator1 == '-' )
65          {
66             if ( operator2 == '*' || operator2 == '/' )
67                return false;
68             else
69                return true;
70          } // end else if
71
72          return false;
```

```
   } // end method precedence
} // end class InfixToPostfixConverter

class CharacterStack extends StackComposition
{
   public char stackTop()
   {
      char temp = popChar();
      pushChar( temp );

      return temp;
   } // end method stackTop

   public char popChar()
   {
      return ( ( Character )super.pop() );
   } // end method popChar

   public void pushChar( char c )
   {
      super.push( c );
   } // end method pushChar
} // end class CharacterStack
```

```
// Exercise 17.12 Solution:InfixToPostfixConverterTest.java
// Infix to postfix conversion
import java.util.Scanner;

public class InfixToPostfixConverterTest
{
   public static void main( String args[] )
   {
      // get infix expression
      Scanner scanner = new Scanner( System.in );
      System.out.println( "Please enter an infix expression:" );
      StringBuffer infix = new StringBuffer( scanner.nextLine() );
      System.out.printf(
         "\nThe original infix expression is:\n%s\n", infix );

      // change from infix notation into postfix notation
      StringBuffer postfix =
         InfixToPostfixConverter.convertToPostfix( infix );

      System.out.printf(
         "The expression in postfix notation is:\n%s\n", postfix );
   } // end main
} // end class InfixToPostfixConverterTest
```

```
Please enter an infix expression:
5+3*6-2

The original infix expression is:
5+3*6-2
The expression in postfix notation is:
5 3 6 * + 2 -
```

```
Please enter an infix expression:
(6+2)*5-8/4

The original infix expression is:
(6+2)*5-8/4
The expression in postfix notation is:
6 2 + 5 * 8 4 / -
```

17.13 Write class `PostfixEvaluator`, which evaluates a postfix expression such as

```
6 2 + 5 * 8 4 / -
```

The program should read a postfix expression consisting of digits and operators into a `StringBuffer`. Using modified versions of the stack methods implemented earlier in this chapter, the program should scan the expression and evaluate it (assume it is valid). The algorithm is as follows:

a) Append a right parenthesis `')'` to the end of the postfix expression. When the right-parenthesis character is encountered, no further processing is necessary.

b) When the right-parenthesis character has not been encountered, read the expression from left to right.

If the current character is a digit, do the following:

Push its integer value on the stack (the integer value of a digit character is its value in the computer's character set minus the value of `'0'` in Unicode).

Otherwise, if the current character is an *operator*:

Pop the two top elements of the stack into variables `x` and `y`.
Calculate `y` *operator* `x`.
Push the result of the calculation onto the stack.

c) When the right parenthesis is encountered in the expression, pop the top value of the stack. This is the result of the postfix expression.

[*Note*: In b) above (based on the sample expression at the beginning of this exercises), if the operator is `'/'`, the top of the stack is `2` and the next element in the stack is `8`, then pop `2` into `x`, pop `8` into `y`, evaluate `8 / 2` and push the result, `4`, back on the stack. This note also applies to operator `'-'`.] The arithmetic operations allowed in an expression are:

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`^`	exponentiation
`%`	remainder

The stack should be maintained with one of the stack classes introduced in this chapter. You may want to provide the following methods:

a) Method `evaluatePostfixExpression`, which evaluates the postfix expression.

b) Method `calculate`, which evaluates the expression `op1 operator op2`.
c) Method `push`, which pushes a value onto the stack.
d) Method `pop`, which pops a value off the stack.
e) Method `isEmpty`, which determines whether the stack is empty.
f) Method `printStack`, which prints the stack.

ANS:

```
// Exercise 17.13 Solution:PostfixEvaluator.java
// Using a stack to evaluate an expression in postfix notation
import com.deitel.jhtp7.ch17.StackInheritance;

public class PostfixEvaluator
{
   // evaluate the postfix notation
   public static int evaluatePostfixExpression( StringBuffer expr )
   {
      int i, popVal1, popVal2, pushVal;
      IntegerStack intStack = new IntegerStack();
      char c;

      expr.append( ")" );

      // until it reaches ")"
      for ( i = 0; expr.charAt( i ) != ')'; ++i )
      {
         if ( Character.isDigit( expr.charAt( i ) ) )
         {
            pushVal = expr.charAt( i ) - '0';
            intStack.pushInt( pushVal );
            intStack.print();
         } // end if
         else if ( !Character.isWhitespace( expr.charAt( i ) ) )
         {
            popVal2 = intStack.popInt();
            intStack.print();
            popVal1 = intStack.popInt();
            intStack.print();
            pushVal = calculate( popVal1, popVal2, expr.charAt( i ) );
            intStack.pushInt( pushVal );
            intStack.print();
         } // end else if
      } // end for

      return intStack.popInt();
   } // end method evaluatePostfixExpression

   // do the calculation
   private static int calculate( int op1, int op2, char oper )
   {
      switch( oper )
      {
         case '+':
            return op1 + op2;
         case '-':
```

```
48             return op1 - op2;
49          case '*':
50             return op1 * op2;
51          case '/':
52             return op1 / op2;
53          case '^':   // exponentiation
54             return ( int )Math.pow( op1, op2 );
55       } // end switch
56
57       return 0;
58    } // end method calculate
59 } // end class PostfixEvaluator
60
61 class IntegerStack extends StackInheritance
62 {
63    public int stackTop()
64    {
65       int temp = popInt();
66       pushInt( temp );
67
68       return temp;
69    } // end method stackTop
70
71    public int popInt()
72    {
73       return ( Integer ) super.pop();
74    } // end method popInt
75
76    public void pushInt( int c )
77    {
78       super.push( c );
79    } // end method pushInt
80 } // end class IntegerStack
```

```
1  // Exercise 17.13 Solution:PostfixEvaluatorTest.java
2  // Using a stack to evaluate an expression in postfix notation
3  import java.util.Scanner;
4
5  public class PostfixEvaluatorTest
6  {
7     public static void main( String args[] )
8     {
9        // get postfix expression
10       Scanner scanner = new Scanner( System.in );
11       System.out.println( "Please enter a postfix expression:" );
12       StringBuffer postfix = new StringBuffer( scanner.nextLine() );
13       System.out.printf(
14          "\nThe original postfix expression is:\n%s\n", postfix );
15
16       // evaluate postfix expression
17       int answer =
18          PostfixEvaluator.evaluatePostfixExpression( postfix );
19
```

```
20         System.out.printf( "The value of the expression is: %d", answer );
21      } // end main
22   } // end class PostfixEvaluatorTest
```

```
Please enter a postfix expression:
5 3 6 * + 2 -

The original postfix expression is:
5 3 6 * + 2 -
The stack is: 5

The stack is: 3 5

The stack is: 6 3 5

The stack is: 3 5

The stack is: 5

The stack is: 18 5

The stack is: 5

Empty stack
The stack is: 23

The stack is: 2 23

The stack is: 23

Empty stack
The stack is: 21

The value of the expression is: 21
```

17.14 Modify the postfix evaluator program of Exercise 17.13 so that it can process integer operands larger than 9.

17.15 *(Supermarket Simulation)* Write a program that simulates a checkout line at a supermarket. The line is a queue object. Customers (i.e., customer objects) arrive in random integer intervals of from 1 to 4 minutes. Also, each customer is serviced in random integer intervals of from 1 to 4 minutes. Obviously, the rates need to be balanced. If the average arrival rate is larger than the average service rate, the queue will grow infinitely. Even with "balanced" rates, randomness can still cause long lines. Run the supermarket simulation for a 12-hour day (720 minutes), using the following algorithm:

a) Choose a random integer between 1 and 4 to determine the minute at which the first customer arrives.

b) At the first customer's arrival time, do the following:
   Determine customer's service time (random integer from 1 to 4).
   Begin servicing the customer.
   Schedule arrival time of next customer (random integer 1 to 4 added to the current time).

c) For each minute of the day, consider the following:
If the next customer arrives, proceed as follows:
Say so.
Enqueue the customer.
Schedule the arrival time of the next customer.
If service was completed for the last customer, do the following:
Say so.
Dequeue next customer to be serviced.
Determine customer's service completion time (random integer from 1 to 4 added to the current time).

Now run your simulation for 720 minutes and answer each of the following:
a) What is the maximum number of customers in the queue at any time?
b) What is the longest wait any one customer experiences?
c) What happens if the arrival interval is changed from 1 to 4 minutes to 1 to 3 minutes?

17.16 Modify Figs. 17.17 and 17.18 to allow the binary tree to contain duplicates.
ANS:

```
1  // Exercise 17.16 Solution: Tree.java
2  // Definition of class TreeNode and class Tree.
3  package com.deitel.jhtp7.ch17;
4
5  // class TreeNode definition
6  class TreeNode
7  {
8     // package access members
9     TreeNode leftNode; // left node
10    int data; // node value
11    TreeNode rightNode; // right node
12
13    // constructor initializes data and makes this a leaf node
14    public TreeNode( int nodeData )
15    {
16       data = nodeData;
17       leftNode = rightNode = null;  // node has no children
18    } // end TreeNode no-argument constructor
19
20    // locate insertion point and insert new node; ignore duplicate values
21    public void insert( int insertValue )
22    {
23       // insert in left subtree
24       if ( insertValue < data )
25       {
26          // insert new TreeNode
27          if ( leftNode == null )
28             leftNode = new TreeNode( insertValue );
29          else // continue traversing left subtree
30             leftNode.insert( insertValue );
31       } // end if
32       else // insert in right subtree
33       {
34          // insert new TreeNode
35          if ( rightNode == null )
```

```
36              rightNode = new TreeNode( insertValue );
37           else // continue traversing right subtree
38              rightNode.insert( insertValue );
39        } // end else if
40     } // end method insert
41  } // end class TreeNode
42
43  // class Tree definition
44  public class Tree
45  {
46     private TreeNode root;
47
48     // constructor initializes an empty Tree of integers
49     public Tree()
50     {
51        root = null;
52     } // end Tree no-argument constructor
53
54     // insert a new node in the binary search tree
55     public void insertNode( int insertValue )
56     {
57        if ( root == null )
58           root = new TreeNode( insertValue ); // create the root node here
59        else
60           root.insert( insertValue ); // call the insert method
61     } // end method insertNode
62
63     // begin preorder traversal
64     public void preorderTraversal()
65     {
66        preorderHelper( root );
67     } // end method preorderTraversal
68
69     // recursive method to perform preorder traversal
70     private void preorderHelper( TreeNode node )
71     {
72        if ( node == null )
73           return;
74
75        System.out.printf( "%d ", node.data ); // output node data
76        preorderHelper( node.leftNode );       // traverse left subtree
77        preorderHelper( node.rightNode );      // traverse right subtree
78     } // end method preorderHelper
79
80     // begin inorder traversal
81     public void inorderTraversal()
82     {
83        inorderHelper( root );
84     } // end method inorderTraversal
85
86     // recursive method to perform inorder traversal
87     private void inorderHelper( TreeNode node )
88     {
89        if ( node == null )
```

```
90           return;
91
92        inorderHelper( node.leftNode );        // traverse left subtree
93        System.out.printf( "%d ", node.data ); // output node data
94        inorderHelper( node.rightNode );       // traverse right subtree
95     } // end method inorderHelper
96
97     // begin postorder traversal
98     public void postorderTraversal()
99     {
100       postorderHelper( root );
101    } // end method postorderTraversal
102
103    // recursive method to perform postorder traversal
104    private void postorderHelper( TreeNode node )
105    {
106       if ( node == null )
107          return;
108
109       postorderHelper( node.leftNode );      // traverse left subtree
110       postorderHelper( node.rightNode );     // traverse right subtree
111       System.out.printf( "%d ", node.data ); // output node data
112    } // end method postorderHelper
113 } // end class Tree
```

```
1  // Exercise 17.16 Solution: TreeTest.java
2  // This program tests the Tree class. The solution to 20.16 is mostly
3  // found in the code of TreeNode, in package com.deitel.jhtp7.ch17
4  import java.util.Random;
5  import com.deitel.jhtp7.ch17.Tree;
6
7  public class TreeTest
8  {
9     public static void main( String args[] )
10    {
11       Tree tree = new Tree();
12       int intVal;
13       Random randomNumber = new Random();
14       System.out.println( "Inserting the following values: " );
15
16       // randomly generate numbers and insert in the tree
17       for ( int i = 1; i <= 10; i++ )
18       {
19          intVal = randomNumber.nextInt( 100 );
20          System.out.print( intVal + " " );
21          tree.insertNode( intVal );
22       } // end for
23
24       // print each of the traversals
25       System.out.println ( "\n\nPreorder traversal" );
26       tree.preorderTraversal();
27
28       System.out.println ( "\n\nInorder traversal" );
```

```
29         tree.inorderTraversal();
30
31         System.out.println ( "\n\nPostorder traversal" );
32         tree.postorderTraversal();
33         System.out.println();
34      } // end main
35   } // end class TreeTest
```

```
Inserting the following values:
53 31 6 97 77 8 8 69 3 39

Preorder traversal
53 31 6 3 8 8 39 97 77 69

Inorder traversal
3 6 8 8 31 39 53 69 77 97

Postorder traversal
3 8 8 6 39 31 69 77 97 53
```

```
Inserting the following values:
98 7 66 4 67 66 19 51 64 49

Preorder traversal
98 7 4 66 19 51 49 64 67 66

Inorder traversal
4 7 19 49 51 64 66 66 67 98

Postorder traversal
4 49 64 51 19 66 67 66 7 98
```

17.17 Write a program based on the program of Figs. 17.17 and 17.18 that inputs a line of text, tokenizes the sentence into separate words (you might want to use the `StreamTokenizer` class from the `java.io` package), inserts the words in a binary search tree and prints the inorder, preorder and postorder traversals of the tree.

ANS:

```
1  // Exercise 17.17 Solution: Tree2.java
2  // Class Tree2 definition.
3  package com.deitel.jhtp7.ch17;
4
5  class TreeNode2
6  {
7     // public access members
8     public TreeNode2 leftNode;
9     public String data;
10    public TreeNode2 rightNode;
11
12    // initialize data and make this a leaf node
13    public TreeNode2( String value )
```

```
14     {
15        data = value;
16        leftNode = rightNode = null; // node has no children
17     } // end TreeNode2 one-argument constructor
18
19     // insert node
20     public void insert( String string )
21     {
22        // insert in left subtree
23        if ( string.compareTo( data ) < 0 )
24        {
25           // insert new TreeNode2
26           if ( leftNode == null )
27              leftNode = new TreeNode2( string );
28           else // continue traversing left subtree
29              leftNode.insert( string );
30        } // end if
31        else // insert in right subtree
32        {
33           // insert new TreeNode2
34           if ( rightNode == null )
35              rightNode = new TreeNode2( string );
36           else // continue traversing right subtree
37              rightNode.insert( string );
38        } // end else
39     } // end method insert
40  } // end class TreeNode2
41
42  // class Tree2 definition
43  public class Tree2
44  {
45     private TreeNode2 root;
46
47     public Tree2()
48     {
49        root = null;
50     } // end Tree2 no-argument constructor
51
52     // begin preorder traversal
53     public void preorderTraversal()
54     {
55        preorderHelper( root );
56     } // end method preorderTraversal
57
58     // recursive method to perform preorder traversal
59     private void preorderHelper( TreeNode2 node )
60     {
61        if ( node == null )
62           return;
63
64        System.out.printf( "%s ", node.data ); // output node data
65        preorderHelper( node.leftNode );       // traverse left subtree
66        preorderHelper( node.rightNode );      // traverse right subtree
67     } // end method preorderHelper
```

```

   // begin inorder traversal
   public void inorderTraversal()
   {
      inorderHelper( root );
   } // end method inorderTraversal

   // recursive method to perform inorder traversal
   private void inorderHelper( TreeNode2 node )
   {
      if ( node == null )
         return;

      inorderHelper( node.leftNode );        // traverse left subtree
      System.out.printf( "%s ", node.data ); // output node data
      inorderHelper( node.rightNode );       // traverse right subtree
   } // end method inorderHelper

   // begin postorder traversal
   public void postorderTraversal()
   {
      postorderHelper( root );
   } // end method postorderTraversal

   // recursive method to perform postorder traversal
   private void postorderHelper( TreeNode2 node )
   {
      if ( node == null )
         return;

      postorderHelper( node.leftNode );      // traverse left subtree
      postorderHelper( node.rightNode );     // traverse right subtree
      System.out.printf( "%s ", node.data ); // output node data
   } // end method postorderHelper

   public void insertNode( String string )
   {
      // tree is empty
      if ( root == null )
         root = new TreeNode2( string );
      else // call TreeNode2 method insert on root
         root.insert( string );
   } // end method insertNode
} // end class Tree2
```

```
// Exercise 17.17 Solution: Tree2Test.java
// Program tests the Tree2 class.
import java.util.Scanner;
import java.util.StringTokenizer;
import com.deitel.jhtp7.ch17.Tree2;


public class Tree2Test
```

```
{
   public static void main( String args[] )
   {
      // get input string
      Scanner scanner = new Scanner( System.in );
      System.out.println( "Please enter a string:" );
      String input = scanner.nextLine();
      StringTokenizer tokens = new StringTokenizer( input );
      Tree2 tree = new Tree2();

      // insert input string into tree
      while ( tokens.hasMoreTokens() )
         tree.insertNode( tokens.nextToken() );

      System.out.println( "\n\nPreorder traversal" );
      tree.preorderTraversal();

      System.out.println( "\n\nInorder traversal" );
      tree.inorderTraversal();

      System.out.println( "\n\nPostorder traversal" );
      tree.postorderTraversal();
   } // end main
} // end class Tree2Test
```

```
Please enter a string:
I have been inserted into a tree


Preorder traversal
I have been a inserted into tree

Inorder traversal
I a been have inserted into tree

Postorder traversal
a been tree into inserted have I
```

17.18 In this chapter, we saw that duplicate elimination is straightforward when creating a binary search tree. Describe how you would perform duplicate elimination when using only a one-dimensional array. Compare the performance of array-based duplicate elimination with the performance of binary-search-tree-based duplicate elimination.

17.19 Write a method `depth` that receives a binary tree and determines how many levels it has.

ANS:

```
// Exercise 17.19 Solution: Tree.java
// Definition of class TreeNode and class Tree.
package com.deitel.jhtp7.ch17;

// class TreeNode definition
class TreeNode
{
```

```
8      // package access members
9      TreeNode leftNode; // left node
10     int data; // node value
11     TreeNode rightNode; // right node
12
13     // constructor initializes data and makes this a leaf node
14     public TreeNode( int nodeData )
15     {
16        data = nodeData;
17        leftNode = rightNode = null;  // node has no children
18     } // end TreeNode no-argument constructor
19
20     // locate insertion point and insert new node; ignore duplicate values
21     public void insert( int insertValue )
22     {
23        // insert in left subtree
24        if ( insertValue < data )
25        {
26           // insert new TreeNode
27           if ( leftNode == null )
28              leftNode = new TreeNode( insertValue );
29           else // continue traversing left subtree
30              leftNode.insert( insertValue );
31        } // end if
32        else // insert in right subtree
33        {
34           // insert new TreeNode
35           if ( rightNode == null )
36              rightNode = new TreeNode( insertValue );
37           else // continue traversing right subtree
38              rightNode.insert( insertValue );
39        } // end else if
40     } // end method insert
41  } // end class TreeNode
42
43  // class Tree definition
44  public class Tree
45  {
46     private TreeNode root;
47
48     // constructor initializes an empty Tree of integers
49     public Tree()
50     {
51        root = null;
52     } // end Tree no-argument constructor
53
54     // insert a new node in the binary search tree
55     public void insertNode( int insertValue )
56     {
57        if ( root == null )
58           root = new TreeNode( insertValue ); // create the root node here
59        else
60           root.insert( insertValue ); // call the insert method
61     } // end method insertNode
```

```
62
63     // begin preorder traversal
64     public void preorderTraversal()
65     {
66        preorderHelper( root );
67     } // end method preorderTraversal
68
69     // recursive method to perform preorder traversal
70     private void preorderHelper( TreeNode node )
71     {
72        if ( node == null )
73           return;
74
75        System.out.printf( "%d ", node.data ); // output node data
76        preorderHelper( node.leftNode );       // traverse left subtree
77        preorderHelper( node.rightNode );      // traverse right subtree
78     } // end method preorderHelper
79
80     // begin inorder traversal
81     public void inorderTraversal()
82     {
83        inorderHelper( root );
84     } // end method inorderTraversal
85
86     // recursive method to perform inorder traversal
87     private void inorderHelper( TreeNode node )
88     {
89        if ( node == null )
90           return;
91
92        inorderHelper( node.leftNode );        // traverse left subtree
93        System.out.printf( "%d ", node.data ); // output node data
94        inorderHelper( node.rightNode );       // traverse right subtree
95     } // end method inorderHelper
96
97     // begin postorder traversal
98     public void postorderTraversal()
99     {
100       postorderHelper( root );
101    } // end method postorderTraversal
102
103    // recursive method to perform postorder traversal
104    private void postorderHelper( TreeNode node )
105    {
106       if ( node == null )
107          return;
108
109       postorderHelper( node.leftNode );      // traverse left subtree
110       postorderHelper( node.rightNode );     // traverse right subtree
111       System.out.printf( "%d ", node.data ); // output node data
112    } // end method postorderHelper
113
114    // recursively determine the depth
115    public int depth()
```

```
116    {
117       return calculateLevel( root );
118    } // end method depth
119
120    // recursively determine the depth
121    private int calculateLevel( TreeNode node )
122    {
123       // if node is a leaf
124       if ( node.rightNode == null && node.leftNode == null )
125          return 0;
126       else
127       {
128          // search each of the subtrees
129          int left = 0;
130          int right = 0;
131
132          if ( node.leftNode != null )
133             left = calculateLevel( node.leftNode );
134
135          if ( node.rightNode != null )
136             right = calculateLevel( node.rightNode );
137
138          // pass on the depth of the deeper subtree
139          if ( left > right )
140             return left + 1;
141          else
142             return right + 1;
143       } // end else
144    } // end method calculateLevel
145 } // end class Tree
```

```
1  // Exercise 17.19 Solution: TreeTest3.java
2  // Program tests method depth.
3  import java.util.Random;
4  import com.deitel.jhtp7.ch17.Tree;
5
6  public class TreeTest3
7  {
8     public static void main( String args[] )
9     {
10       Tree tree = new Tree();
11       int number;
12       Random randomNumber = new Random();
13       System.out.println( "Inserting the following values: " );
14
15       // create tree of random numbers
16       for ( int i = 1; i <= 10; i++ )
17       {
18          number = randomNumber.nextInt( 100 );
19          System.out.print( number + " " );
20          tree.insertNode( number );
21       } // end for
22
```

```
23         System.out.println ( "\n\nPreorder traversal" );
24         tree.preorderTraversal();
25
26         System.out.println ( "\n\nInorder traversal" );
27         tree.inorderTraversal();
28
29         System.out.println ( "\n\nPostorder traversal" );
30         tree.postorderTraversal();
31
32         System.out.printf( "\n\nTree has a depth of: %d", tree.depth() );
33      } // end main
34   } // end class TreeTest3
```

```
Inserting the following values:
50 81 35 65 45 32 82 5 85 71

Preorder traversal
50 35 32 5 45 81 65 71 82 85

Inorder traversal
5 32 35 45 50 65 71 81 82 85

Postorder traversal
5 32 45 35 71 65 85 82 81 50

Tree has a depth of: 3
```

17.20 (*Recursively Print a List Backward*) Write a method `printListBackward` that recursively outputs the items in a linked list object in reverse order. Write a test program that creates a sorted list of integers and prints the list in reverse order.

ANS:

```
1  // Exercise 17.20 Solution: List.java
2  // Class List2 definition.
3  package com.deitel.jhtp7.ch17;
4
5  // class to represent one node in a list
6  class ListNode
7  {
8     // package access members; List can access these directly
9     Object data;
10    ListNode nextNode;
11
12    // constructor creates a ListNode that refers to object
13    ListNode( Object object )
14    {
15       this( object, null );
16    } // end ListNode one-argument constructor
17
18    // constructor creates ListNode that refers to
19    // Object and to next ListNode
20    ListNode( Object object, ListNode node )
```

```
21      {
22         data = object;
23         nextNode = node;
24      } // end ListNode two-argument constructor
25
26      // return reference to data in node
27      Object getObject()
28      {
29         return data; // return Object in this node
30      } // end method getObject
31
32      // return reference to next node in list
33      ListNode getNext()
34      {
35         return nextNode; // get next node
36      } // end method getNext
37   } // end class ListNode
38
39   // class List definition
40   public class List
41   {
42      private ListNode firstNode;
43      private ListNode lastNode;
44      private String name; // string like "list" used in printing
45
46      // constructor creates empty List with "list" as the name
47      public List()
48      {
49         this( "list" );
50      } // end List no-argument constructor
51
52      // constructor creates an empty List with a name
53      public List( String listName )
54      {
55         name = listName;
56         firstNode = lastNode = null;
57      } // end List one-argument constructor
58
59      // insert Object at front of List
60      public void insertAtFront( Object insertItem )
61      {
62         if ( isEmpty() ) // firstNode and lastNode refer to same object
63            firstNode = lastNode = new ListNode( insertItem );
64         else // firstNode refers to new node
65            firstNode = new ListNode( insertItem, firstNode );
66      } // end method insertAtFront
67
68      // insert Object at end of List
69      public void insertAtBack( Object insertItem )
70      {
71         if ( isEmpty() ) // firstNode and lastNode refer to same Object
72            firstNode = lastNode = new ListNode( insertItem );
73         else // lastNode's nextNode refers to new node
74            lastNode = lastNode.nextNode = new ListNode( insertItem );
```

```
75      } // end method insertAtBack
76
77      // remove first node from List
78      public Object removeFromFront() throws EmptyListException
79      {
80         if ( isEmpty() ) // throw exception if List is empty
81            throw new EmptyListException( name );
82
83         Object removedItem = firstNode.data; // retrieve data being removed
84
85         // update references firstNode and lastNode
86         if ( firstNode == lastNode )
87            firstNode = lastNode = null;
88         else
89            firstNode = firstNode.nextNode;
90
91         return removedItem; // return removed node data
92      } // end method removeFromFront
93
94      // remove last node from List
95      public Object removeFromBack() throws EmptyListException
96      {
97         if ( isEmpty() ) // throw exception if List is empty
98            throw new EmptyListException( name );
99
100        Object removedItem = lastNode.data; // retrieve data being removed
101
102        // update references firstNode and lastNode
103        if ( firstNode == lastNode )
104           firstNode = lastNode = null;
105        else // locate new last node
106        {
107           ListNode current = firstNode;
108
109           // loop while current node does not refer to lastNode
110           while ( current.nextNode != lastNode )
111              current = current.nextNode;
112
113           lastNode = current; // current is new lastNode
114           current.nextNode = null;
115        } // end else
116
117        return removedItem; // return removed node data
118     } // end method removeFromBack
119
120     // determine whether list is empty
121     public boolean isEmpty()
122     {
123        return firstNode == null; // return true if List is empty
124     } // end method isEmpty
125
126     // output List contents
127     public void print()
128     {
```

```
129        if ( isEmpty() )
130        {
131           System.out.printf( "Empty %s\n", name );
132           return;
133        } // end if
134
135        System.out.printf( "The %s is: ", name );
136        ListNode current = firstNode;
137
138        // while not at end of list, output current node's data
139        while ( current != null )
140        {
141           System.out.printf( "%s ", current.data );
142           current = current.nextNode;
143        } // end while
144
145        System.out.println( "\n" );
146     } // end method print
147
148     // print list backwards
149     public void printListBackwards()
150     {
151        System.out.print( "Reverse ordered list: " );
152        reverse( firstNode );
153        System.out.println();
154     } // end method printListBackwards
155
156     // reverse node
157     private void reverse( ListNode currentNode )
158     {
159        if ( currentNode == null )
160           return;
161        else
162           reverse( currentNode.getNext() );
163
164        System.out.printf( "%s ", currentNode.data );
165     } // end method reverse
166  } // end class List
```

```
1  // Exercise 17.20 Solution: ListTest.java
2  // Program recursively prints a list of random numbers backwards.
3  import java.util.Random;
4  import com.deitel.jhtp7.ch17.List;
5
6  public class ListTest
7  {
8     public static void main( String args[] )
9     {
10       List list = new List();
11       int number;
12       Random randomNumber = new Random();
13
14       // create objects to store in the List
```

```
15         for ( int i = 1; i <= 25; i++ )
16         {
17            number = randomNumber.nextInt( 101 );
18            list.insertAtFront( number );
19         } // end for
20
21         list.print();
22         list.printListBackwards();
23      } // end main
24   } // end class ListTest
```

```
The list is: 100 42 43 85 6 81 42 71 90 14 24 60 71 36 0 72 22 76 17 59 15 13
90 70

Reverse ordered list: 70 0 9 13 15 59 17 76 22 72 0 36 71 60 24 14 90 71 42
81 6  85 43 42 100
```

17.21 (*Recursively Search a List*) Write a method `searchList` that recursively searches a linked list object for a specified value. Method `searchList` should return a reference to the value if it is found; otherwise, `null` should be returned. Use your method in a test program that creates a list of integers. The program should prompt the user for a value to locate in the list.

ANS:

```
1  // Exercise 17.21 Solution: List.java
2  // Class List and ListNode declarations.
3  package com.deitel.jhtp7.ch17;
4
5  // class to represent one node in a list
6  class ListNode
7  {
8     // package access members; List can access these directly
9     Object data;
10    ListNode nextNode;
11
12    // constructor creates a ListNode that refers to object
13    ListNode( Object object )
14    {
15       this( object, null );
16    } // end ListNode one-argument constructor
17
18    // constructor creates ListNode that refers to
19    // Object and to next ListNode
20    ListNode( Object object, ListNode node )
21    {
22       data = object;
23       nextNode = node;
24    } // end ListNode two-argument constructor
25
26    // return reference to data in node
27    Object getObject()
28    {
29       return data; // return Object in this node
```

```
30      } // end method getObject
31
32      // return reference to next node in list
33      ListNode getNext()
34      {
35         return nextNode; // get next node
36      } // end method getNext
37   } // end class ListNode
38
39   // class List definition
40   public class List
41   {
42      private ListNode firstNode;
43      private ListNode lastNode;
44      private String name; // string like "list" used in printing
45
46      // constructor creates empty List with "list" as the name
47      public List()
48      {
49         this( "list" );
50      } // end List no-argument constructor
51
52      // constructor creates an empty List with a name
53      public List( String listName )
54      {
55         name = listName;
56         firstNode = lastNode = null;
57      } // end List one-argument constructor
58
59      // insert Object at front of List
60      public void insertAtFront( Object insertItem )
61      {
62         if ( isEmpty() ) // firstNode and lastNode refer to same object
63            firstNode = lastNode = new ListNode( insertItem );
64         else // firstNode refers to new node
65            firstNode = new ListNode( insertItem, firstNode );
66      } // end method insertAtFront
67
68      // insert Object at end of List
69      public void insertAtBack( Object insertItem )
70      {
71         if ( isEmpty() ) // firstNode and lastNode refer to same Object
72            firstNode = lastNode = new ListNode( insertItem );
73         else // lastNode's nextNode refers to new node
74            lastNode = lastNode.nextNode = new ListNode( insertItem );
75      } // end method insertAtBack
76
77      // remove first node from List
78      public Object removeFromFront() throws EmptyListException
79      {
80         if ( isEmpty() ) // throw exception if List is empty
81            throw new EmptyListException( name );
82
83         Object removedItem = firstNode.data; // retrieve data being removed
```

```
84
85          // update references firstNode and lastNode
86          if ( firstNode == lastNode )
87             firstNode = lastNode = null;
88          else
89             firstNode = firstNode.nextNode;
90
91          return removedItem; // return removed node data
92       } // end method removeFromFront
93
94       // remove last node from List
95       public Object removeFromBack() throws EmptyListException
96       {
97          if ( isEmpty() ) // throw exception if List is empty
98             throw new EmptyListException( name );
99
100         Object removedItem = lastNode.data; // retrieve data being removed
101
102         // update references firstNode and lastNode
103         if ( firstNode == lastNode )
104            firstNode = lastNode = null;
105         else // locate new last node
106         {
107            ListNode current = firstNode;
108
109            // loop while current node does not refer to lastNode
110            while ( current.nextNode != lastNode )
111               current = current.nextNode;
112
113            lastNode = current; // current is new lastNode
114            current.nextNode = null;
115         } // end else
116
117         return removedItem; // return removed node data
118      } // end method removeFromBack
119
120      // determine whether list is empty
121      public boolean isEmpty()
122      {
123         return firstNode == null; // return true if List is empty
124      } // end method isEmpty
125
126      // output List contents
127      public void print()
128      {
129         if ( isEmpty() )
130         {
131            System.out.printf( "Empty %s\n", name );
132            return;
133         } // end if
134
135         System.out.printf( "The %s is: ", name );
136         ListNode current = firstNode;
137
```

```
138        // while not at end of list, output current node's data
139        while ( current != null )
140        {
141           System.out.printf( "%s ", current.data );
142           current = current.nextNode;
143        } // end while
144
145        System.out.println( "\n" );
146     } // end method print
147
148     // search list for a specifed value
149     public Object search( Object input )
150     {
151        return searchHelper( input, firstNode );
152     } // end method search
153
154     // helper method for searching
155     private Object searchHelper( Object input, ListNode node )
156     {
157        if ( node == null )
158           return null;
159        else if ( input.equals( node.data ) )
160           return node.data;
161        else
162           return searchHelper( input, node.nextNode );
163     } // end method searchHelper
164  } // end class List
```

```
1  // Exercise 17.21 Solution: ListTest.java
2  // Program recursively searches a list of numbers.
3  import java.util.Random;
4  import com.deitel.jhtp7.ch17.List;
5
6  public class ListTest
7  {
8     public static void main( String args[] )
9     {
10       List list = new List();
11       int number;
12       Random randomNumber = new Random();
13
14       // create objects to store in the List
15       for ( int i = 1; i <= 25; i++ )
16       {
17          number = randomNumber.nextInt( 101 );
18          list.insertAtFront( number );
19       } // end for
20
21       list.print();
22
23       Object searchResult = list.search( 34 );
24
25       // display result of searching 34
```

```
26         if ( searchResult != null )
27            System.out.println( "Value found: 34" );
28         else
29            System.out.println( "Value not found: 34" );
30
31         searchResult = list.search( 50 );
32
33         // display result of searching 50
34         if ( searchResult != null )
35            System.out.println( "Value found: 50" );
36         else
37            System.out.println( "Value not found: 50" );
38
39         searchResult = list.search( 72 );
40
41         // display result of searching 72
42         if ( searchResult != null )
43            System.out.println( "Value found: 72" );
44         else
45            System.out.println( "Value not found: 72" );
46      } // end main
47   } // end class ListTest
```

```
The list is: 82 41 9 13 72 40 56 88 79 92 33 96 62 47 84 65 83 92 41 11 57 25
43 77 18

Value not found: 34
Value not found: 50
Value found: 72
```

17.22 (*Binary Tree Delete*) In this exercise, we discuss deleting items from binary search trees. The deletion algorithm is not as straightforward as the insertion algorithm. Three cases are encountered when deleting an item—the item is contained in a leaf node (i.e., it has no children), the item is contained in a node that has one child or the item is contained in a node that has two children.

If the item to be deleted is contained in a leaf node, the node is deleted and the reference in the parent node is set to null.

If the item to be deleted is contained in a node with one child, the reference in the parent node is set to reference the child node and the node containing the data item is deleted. This causes the child node to take the place of the deleted node in the tree.

The last case is the most difficult. When a node with two children is deleted, another node in the tree must take its place. However, the reference in the parent node cannot simply be assigned to reference one of the children of the node to be deleted. In most cases, the resulting binary search tree would not embody the following characteristic of binary search trees (with no duplicate values): *The values in any left subtree are less than the value in the parent node, and the values in any right subtree are greater than the value in the parent node.*

Which node is used as a *replacement node* to maintain this characteristic? It is either the node containing the largest value in the tree less than the value in the node being deleted, or the node containing the smallest value in the tree greater than the value in the node being deleted. Let us consider the node with the smaller value. In a binary search tree, the largest value less than a parent's value is located in the left subtree of the parent node and is guaranteed to be contained in the rightmost node of the subtree. This node is located by walking down the left subtree to the right

until the reference to the right child of the current node is null. We are now referencing the replacement node, which is either a leaf node or a node with one child to its left. If the replacement node is a leaf node, the steps to perform the deletion are as follows:

a) Store the reference to the node to be deleted in a temporary reference variable.
b) Set the reference in the parent of the node being deleted to reference the replacement node.
c) Set the reference in the parent of the replacement node to `null`.
d) Set the reference to the right subtree in the replacement node to reference the right subtree of the node to be deleted.
e) Set the reference to the left subtree in the replacement node to reference the left subtree of the node to be deleted.

The deletion steps for a replacement node with a left child are similar to those for a replacement node with no children, but the algorithm also must move the child into the replacement node's position in the tree. If the replacement node is a node with a left child, the steps to perform the deletion are as follows:

a) Store the reference to the node to be deleted in a temporary reference variable.
b) Set the reference in the parent of the node being deleted to reference the replacement node.
c) Set the reference in the parent of the replacement node to reference the left child of the replacement node.
d) Set the reference to the right subtree in the replacement node to reference the right subtree of the node to be deleted.
e) Set the reference to the left subtree in the replacement node to reference the left subtree of the node to be deleted.

Write method `deleteNode`, which takes as its argument the value to delete. Method `deleteNode` should locate in the tree the node containing the value to delete and use the algorithms discussed here to delete the node. If the value is not found in the tree, the method should print a message that indicates whether the value is deleted. Modify the program of Figs. 17.17 and 17.18 to use this method. After deleting an item, call the methods `inorderTraversal`, `preorderTraversal` and `postorderTraversal` to confirm that the delete operation was performed correctly.

ANS:

```
1  // Exercise 17.22 Solution: Tree5.java
2  // Deleting items from a tree.
3  package com.deitel.jhtp7.ch17;
4
5  // class TreeNode declaration
6  class TreeNode
7  {
8     // package access members
9     TreeNode leftNode; // left node
10    int data; // node value
11    TreeNode rightNode; // right node
12
13    // constructor initializes data and makes this a leaf node
14    public TreeNode( int nodeData )
15    {
16       data = nodeData;
17       leftNode = rightNode = null;  // node has no children
18    } // end TreeNode no-argument constructor
19
```

```
20     // locate insertion point and insert new node; ignore duplicate values
21     public void insert( int insertValue )
22     {
23        // insert in left subtree
24        if ( insertValue < data )
25        {
26           // insert new TreeNode
27           if ( leftNode == null )
28              leftNode = new TreeNode( insertValue );
29           else // continue traversing left subtree
30              leftNode.insert( insertValue );
31        } // end if
32        else // insert in right subtree
33        {
34           // insert new TreeNode
35           if ( rightNode == null )
36              rightNode = new TreeNode( insertValue );
37           else // continue traversing right subtree
38              rightNode.insert( insertValue );
39        } // end else if
40     } // end method insert
41  } // end class TreeNode
42
43  // class Tree5 declaration
44  public class Tree5
45  {
46     protected TreeNode root;
47
48     public Tree5()
49     {
50        root = null;
51     } // end Tree5 no-argument constructor
52
53     // Insert new node in the binary tree. If the root is null, create the
54     // root here. Otherwise, call the insert method of class TreeNode.
55     public void insertNode( int d )
56     {
57        if ( root == null )
58           root = new TreeNode( d );
59        else
60           root.insert( d );
61     } // end method insertNode
62
63     // Preorder Traversal
64     public void preorderTraversal()
65     {
66        preorderHelper( root );
67     } // end method preorderTraversal
68
69     // Recursive method to perform preorder traversal
70     private void preorderHelper( TreeNode node )
71     {
72        if ( node == null )
73           return;
```

```
74
75          System.out.printf( "%d ", node.data );
76          preorderHelper( node.leftNode );
77          preorderHelper( node.rightNode );
78       } // end method preorderHelper
79
80       // Inorder Traversal
81       public void inorderTraversal()
82       {
83          inorderHelper( root );
84       } // end method inorderTraversal
85
86       // Recursive method to perform inorder traversal
87       private void inorderHelper( TreeNode node )
88       {
89          if ( node == null )
90             return;
91
92          inorderHelper( node.leftNode );
93          System.out.printf( "%d ", node.data );
94          inorderHelper( node.rightNode );
95       } // end method inorderHelper
96
97       // Postorder Traversal
98       public void postorderTraversal()
99       {
100         postorderHelper( root );
101      } // end method postorderTraversal
102
103      // Recursive method to perform postorder traversal
104      private void postorderHelper( TreeNode node )
105      {
106         if ( node == null )
107            return;
108
109         postorderHelper( node.leftNode );
110         postorderHelper( node.rightNode );
111         System.out.printf( "%d ", node.data );
112      } // end method postorderHelper
113
114      // top level method call
115      public void deleteItem( int d )
116      {
117         // if the tree is empty
118         if ( root == null )
119            return;
120
121         int test = root.data;
122
123         // if the root is the value to be deleted
124         if ( test == d )
125         {
126            // if the left child is null, set root to the right
127            if ( root.leftNode == null )
```

```
128             root = root.rightNode;
129
130          // if the right child is null, set root to the left
131          else if ( root.rightNode == null )
132             root = root.leftNode;
133          else // both children have values
134          {
135             boolean later = false;
136             TreeNode parent = root;
137
138             // find the largest value smaller than the root
139             TreeNode replacement = root.leftNode;
140
141             while ( replacement.rightNode != null )
142             {
143                later = true;
144                parent = replacement;
145                replacement = replacement.rightNode;
146             } // end while
147
148             // create a temporary value
149             TreeNode temp = root;
150
151             // set the root to the replacement node
152             root = replacement;
153
154             // break the old link to the replacement node
155             if ( later )
156                parent.rightNode = replacement.leftNode;
157             else // only occurs if replacement is immediately to the left
158                parent.leftNode = replacement.leftNode;
159
160             replacement.rightNode = temp.rightNode;
161             replacement.leftNode = temp.leftNode;
162          } // end else
163       } // end if
164       else if ( test > d ) // node is to the left of root
165          root.leftNode = deleteNode( root.leftNode, d );
166       else if ( test < d ) // node is to the right of root
167          root.rightNode = deleteNode( root.rightNode, d );
168    } // end method deleteItem
169
170    private TreeNode deleteNode( TreeNode top, int d )
171    {
172       // if the tree is empty
173       if ( top == null )
174          return top;
175
176       int test = top.data;
177
178       // if the top is the value to be deleted
179       if ( test == d )
180       {
181          // if the left child is null, set top to its right
```

```
182          if ( top.leftNode == null )
183             return top.rightNode;
184          else if ( top.rightNode == null ) // right child is null
185             return top.leftNode; // set top to its left child
186          else // both children have values
187          {
188             boolean later = false;
189             TreeNode parent = root;
190
191             // find the largest value smaller than the top
192             TreeNode replacement = top.leftNode;
193
194             while ( replacement.rightNode != null )
195             {
196                later = true;
197                parent = replacement;
198                replacement = replacement.rightNode;
199             } // end while
200
201             // create a temporary value
202             TreeNode temp = top;
203
204             // set the top to the replacement node
205             top = replacement;
206
207             // break the old link to the replacement node
208             if ( later )
209                parent.rightNode = replacement.leftNode;
210             else // only occurs if replacement is immediately to the left
211                parent.leftNode = replacement.leftNode;
212
213             replacement.rightNode = temp.rightNode;
214             replacement.leftNode = temp.leftNode;
215          } // end else
216       } // end if
217       else if ( test > d ) // node is to the left of root
218          top.leftNode = deleteNode( top.leftNode, d );
219       else if ( test < d ) // node is to the right of root
220          top.rightNode = deleteNode( top.rightNode, d );
221
222       return top;
223    } // end method deleteNode
224 } // end class Tree5
```

```
1  // Exercise 17.22 Solution: TreeTest3.java
2  // Program extends a binary tree to allow deletion.
3  import java.util.Random;
4  import com.deitel.jhtp7.ch17.Tree5;
5
6  public class TreeTest3
7  {
8     public static void main( String args[] )
9     {
```

```
10         Tree5 tree = new Tree5();
11         int number;
12         Random randomNumber = new Random();
13         System.out.println( "Inserting the following values: " );
14
15         // create tree of random numbers
16         for ( int i = 1; i <= 20; i++ )
17         {
18            number = randomNumber.nextInt( 100 );
19            System.out.print( number + " " );
20            tree.insertNode( new Integer( number ) );
21         } // end for
22
23         System.out.println ( "\n\nPreorder traversal" );
24         tree.preorderTraversal();
25
26         System.out.println ( "\n\nInorder traversal" );
27         tree.inorderTraversal();
28
29         System.out.println ( "\n\nPostorder traversal" );
30         tree.postorderTraversal();
31
32         int deleteValue = randomNumber.nextInt( 100 );
33
34         System.out.printf( "\n\nDeleted value: %d", deleteValue );
35         tree.deleteItem( new Integer( deleteValue ) );
36
37         System.out.println ( "\n\nPreorder traversal" );
38         tree.preorderTraversal();
39
40         System.out.println ( "\n\nInorder traversal" );
41         tree.inorderTraversal();
42
43         System.out.println ( "\n\nPostorder traversal" );
44         tree.postorderTraversal();
45      } // end method main
46   } // end class TreeTest3
```

```
Inserting the following values:
96 57 55 91 35 59 2 3 87 70 45 23 33 77 13 93 80 69 37 57

Preorder traversal
96 57 55 35 2 3 23 13 33 45 37 91 59 57 87 70 69 77 80 93

Inorder traversal
2 3 13 23 33 35 37 45 55 57 57 59 69 70 77 80 87 91 93 96

Postorder traversal
13 33 23 3 2 37 45 35 55 57 69 80 77 70 87 59 93 91 57 96

Deleted value: 80

Preorder traversal
96 57 55 35 2 3 23 13 33 45 37 91 59 57 87 70 69 77 93

Inorder traversal
2 3 13 23 33 35 37 45 55 57 57 59 69 70 77 87 91 93 96

Postorder traversal
13 33 23 3 2 37 45 35 55 57 69 77 70 87 59 93 91 57 96
```

17.23 (*Binary Tree Search*) Write method `binaryTreeSearch`, which attempts to locate a specified value in a binary-search-tree object. The method should take as an argument a search key to be located. If the node containing the search key is found, the method should return a reference to that node; otherwise, it should return a null reference.

ANS:

```
1  // Exercise 17.23 Solution: Tree.java
2  // Class Tree4 definition.
3  package com.deitel.jhtp7.ch17;
4
5  // class TreeNode definition
6  class TreeNode
7  {
8     // package access members
9     TreeNode leftNode; // left node
10    int data; // node value
11    TreeNode rightNode; // right node
12
13    // constructor initializes data and makes this a leaf node
14    public TreeNode( int nodeData )
15    {
16       data = nodeData;
17       leftNode = rightNode = null;  // node has no children
18    } // end TreeNode no-argument constructor
19
20    // locate insertion point and insert new node; ignore duplicate values
21    public void insert( int insertValue )
22    {
23       // insert in left subtree
24       if ( insertValue < data )
```

```
25          {
26             // insert new TreeNode
27             if ( leftNode == null )
28                leftNode = new TreeNode( insertValue );
29             else // continue traversing left subtree
30                leftNode.insert( insertValue );
31          } // end if
32          else if ( insertValue > data ) // insert in right subtree
33          {
34             // insert new TreeNode
35             if ( rightNode == null )
36                rightNode = new TreeNode( insertValue );
37             else // continue traversing right subtree
38                rightNode.insert( insertValue );
39          } // end else if
40       } // end method insert
41    } // end class TreeNode
42
43    // class Tree definition
44    public class Tree
45    {
46       private TreeNode root;
47
48       // constructor initializes an empty Tree of integers
49       public Tree()
50       {
51          root = null;
52       } // end Tree no-argument constructor
53
54       // insert a new node in the binary search tree
55       public void insertNode( int insertValue )
56       {
57          if ( root == null )
58             root = new TreeNode( insertValue ); // create the root node here
59          else
60             root.insert( insertValue ); // call the insert method
61       } // end method insertNode
62
63       // begin preorder traversal
64       public void preorderTraversal()
65       {
66          preorderHelper( root );
67       } // end method preorderTraversal
68
69       // recursive method to perform preorder traversal
70       private void preorderHelper( TreeNode node )
71       {
72          if ( node == null )
73             return;
74
75          System.out.printf( "%d ", node.data ); // output node data
76          preorderHelper( node.leftNode );       // traverse left subtree
77          preorderHelper( node.rightNode );      // traverse right subtree
78       } // end method preorderHelper
```

```
79
80    // begin inorder traversal
81    public void inorderTraversal()
82    {
83       inorderHelper( root );
84    } // end method inorderTraversal
85
86    // recursive method to perform inorder traversal
87    private void inorderHelper( TreeNode node )
88    {
89       if ( node == null )
90          return;
91
92       inorderHelper( node.leftNode );        // traverse left subtree
93       System.out.printf( "%d ", node.data ); // output node data
94       inorderHelper( node.rightNode );       // traverse right subtree
95    } // end method inorderHelper
96
97    // begin postorder traversal
98    public void postorderTraversal()
99    {
100      postorderHelper( root );
101   } // end method postorderTraversal
102
103   // recursive method to perform postorder traversal
104   private void postorderHelper( TreeNode node )
105   {
106      if ( node == null )
107         return;
108
109      postorderHelper( node.leftNode );      // traverse left subtree
110      postorderHelper( node.rightNode );     // traverse right subtree
111      System.out.printf( "%d ", node.data ); // output node data
112   } // end method postorderHelper
113
114   // begin binary tree search
115   public boolean contains( int key )
116   {
117      if ( containsHelper( root, key ) != null )
118         return true;
119      else
120         return false;
121   } // end method search
122
123   // recursive method to perform binary tree search
124   private TreeNode containsHelper( TreeNode currentNode, int key )
125   {
126      // key not found
127      if ( currentNode == null )
128         return null;
129
130      // key found
131      if ( key == currentNode.data )
132         return currentNode;
133      else if ( key < currentNode.data ) // search left node
```

```
134          return containsHelper( currentNode.leftNode, key );
135       else //  traverse down right child subtree
136          return containsHelper( currentNode.rightNode, key );
137    } // end method searchHelper
138 } // end class Tree
```

```
 1  // Exercise 17.23 Solution: TreeTest.java
 2  // Program performs a binary tree search.
 3  import java.util.Random;
 4  import com.deitel.jhtp7.ch17.Tree;
 5
 6  public class TreeTest
 7  {
 8     public static void main( String args[] )
 9     {
10        Tree tree = new Tree();
11        int number;
12        Random randomNumber = new Random();
13        System.out.println( "Inserting the following values: " );
14
15        // create Objects to store in tree
16        for ( int i = 1; i <= 10; i++ )
17        {
18           number = randomNumber.nextInt( 100 );
19           System.out.printf( "%d ", number );
20           tree.insertNode( number );
21        } // end for
22
23        // create Object to search for in tree
24        int searchNumber = randomNumber.nextInt( 100 );
25
26        // search
27        boolean result = tree.contains( searchNumber );
28
29        // searchNumber not in tree
30        if ( result == true )
31           System.out.printf( "\nThe tree contains %d\n", searchNumber );
32        else // searchNumber found in tree
33           System.out.printf(
34              "\nThe tree does not contain %d\n", searchNumber );
35     } // end main
36  } // end TreeTest
```

```
Inserting the following values:
23 45 99 42 41 85 63 29 23 41
The tree does not contain 36
```

```
Inserting the following values:
80 30 83 62 56 55 34 40 96 43
The tree contains 30
```

17.24 (*Level-Order Binary Tree Traversal*) The program of Figs. 17.17 and 17.18 illustrated three recursive methods of traversing a binary tree—inorder, preorder and postorder traversals. This exercise presents the *level-order traversal* of a binary tree, in which the node values are printed level by level, starting at the root node level. The nodes on each level are printed from left to right. The level-order traversal is not a recursive algorithm. It uses a queue object to control the output of the nodes. The algorithm is as follows:

a) Insert the root node in the queue.
b) While there are nodes left in the queue, do the following:
    Get the next node in the queue.
    Print the node's value.
    If the reference to the left child of the node is not null:
        Insert the left child node in the queue.
    If the reference to the right child of the node is not null:
        Insert the right child node in the queue.

Write method `levelOrder` to perform a level-order traversal of a binary tree object. Modify the program of Figs. 17.17 and 17.18 to use this method. [*Note*: You will also need to use queue-processing methods of Fig. 17.13 in this program.]

ANS:

```
1  // Exercise 17.24 Solution: Tree.java
2  // Class Tree definition
3  package com.deitel.jhtp7.ch17;
4
5  // class TreeNode declaration
6  class TreeNode
7  {
8     // package access members
9     TreeNode leftNode; // left node
10    int data; // node value
11    TreeNode rightNode; // right node
12
13    // constructor initializes data and makes this a leaf node
14    public TreeNode( int nodeData )
15    {
16       data = nodeData;
17       leftNode = rightNode = null;  // node has no children
18    } // end TreeNode no-argument constructor
19
20    // locate insertion point and insert new node; ignore duplicate values
21    public void insert( int insertValue )
22    {
23       // insert in left subtree
24       if ( insertValue < data )
25       {
26          // insert new TreeNode
27          if ( leftNode == null )
28             leftNode = new TreeNode( insertValue );
29          else // continue traversing left subtree
30             leftNode.insert( insertValue );
31       } // end if
32       else if ( insertValue > data ) // insert in right subtree
33       {
```

```
34            // insert new TreeNode
35            if ( rightNode == null )
36               rightNode = new TreeNode( insertValue );
37            else // continue traversing right subtree
38               rightNode.insert( insertValue );
39         } // end else if
40      } // end method insert
41   } // end class TreeNode
42
43   // class Tree declaration
44   public class Tree
45   {
46      protected TreeNode root;
47
48      // no-argument constructor
49      public Tree()
50      {
51         root = null;
52      } // end no-argument Tree constructor
53
54      // print node value level-by-level
55      public void levelOrderTraversal()
56      {
57         Queue traversal = new Queue(); // create the queue
58         traversal.enqueue( root ); // add the root to the queue
59
60         // level-order traversal of a binary tree
61         try
62         {
63            while ( true )
64            {
65               // get the first node in the queue
66               TreeNode recent = ( TreeNode ) traversal.dequeue();
67
68               // print the node
69               System.out.printf( "%d ", recent.data );
70
71               // add a left node if it exists
72               if ( recent.leftNode != null )
73                  traversal.enqueue( recent.leftNode );
74
75               // add a right node if it exists
76               if ( recent.rightNode != null )
77                  traversal.enqueue( recent.rightNode );
78            } // end while
79         } // end try
80         catch ( EmptyListException e )
81         {
82            // the queue is empty and the method can end
83         } // end catch
84      } // end method levelOrderTraversal
85
86      // insert a new node in the binary search tree
87      public void insertNode( int insertValue )
```

```
88     {
89        if ( root == null )
90           root = new TreeNode( insertValue ); // create the root node here
91        else
92           root.insert( insertValue ); // call the insert method
93     } // end method insertNode
94
95     // begin preorder traversal
96     public void preorderTraversal()
97     {
98        preorderHelper( root );
99     } // end method preorderTraversal
100
101    // recursive method to perform preorder traversal
102    private void preorderHelper( TreeNode node )
103    {
104       if ( node == null )
105          return;
106
107       System.out.printf( "%d ", node.data ); // output node data
108       preorderHelper( node.leftNode );       // traverse left subtree
109       preorderHelper( node.rightNode );      // traverse right subtree
110    } // end method preorderHelper
111
112    // begin inorder traversal
113    public void inorderTraversal()
114    {
115       inorderHelper( root );
116    } // end method inorderTraversal
117
118    // recursive method to perform inorder traversal
119    private void inorderHelper( TreeNode node )
120    {
121       if ( node == null )
122          return;
123
124       inorderHelper( node.leftNode );        // traverse left subtree
125       System.out.printf( "%d ", node.data ); // output node data
126       inorderHelper( node.rightNode );       // traverse right subtree
127    } // end method inorderHelper
128
129    // begin postorder traversal
130    public void postorderTraversal()
131    {
132       postorderHelper( root );
133    } // end method postorderTraversal
134
135    // recursive method to perform postorder traversal
136    private void postorderHelper( TreeNode node )
137    {
138       if ( node == null )
139          return;
140
141       postorderHelper( node.leftNode );      // traverse left subtree
```

```
142         postorderHelper( node.rightNode );     // traverse right subtree
143         System.out.printf( "%d ", node.data ); // output node data
144      } // end method postorderHelper
145   } // end class Tree
```

```
1   // Exercise 17.24 Solution: TreeTest.java
2   // This program tests the Tree class.
3   import java.util.Random;
4   import com.deitel.jhtp7.ch17.Tree;
5
6   public class TreeTest
7   {
8      public static void main( String args[] )
9      {
10        Tree tree = new Tree();
11        int intVal;
12        Random randomNumber = new Random();
13        System.out.println( "Inserting the following values: " );
14
15        // randomly generate numbers and insert in the tree
16        for ( int i = 1; i <= 10; i++ )
17        {
18           intVal = randomNumber.nextInt( 100 );
19           System.out.printf( "%d ", intVal );
20           tree.insertNode( new Integer( intVal ) );
21        } // end for
22
23        // print each of the traversals
24        System.out.println ( "\n\nPreorder traversal" );
25        tree.preorderTraversal();
26
27        System.out.println ( "\n\nInorder traversal" );
28        tree.inorderTraversal();
29
30        System.out.println ( "\n\nPostorder traversal" );
31        tree.postorderTraversal();
32
33        System.out.println( "\n\nLevel-order traversal" );
34        tree.levelOrderTraversal();
35        System.out.println();
36     } // end main
37  } // end class TreeTest
```

```
Inserting the following values:
14 49 63 94 4 88 68 5 55 57

Preorder traversal
14 4 5 49 63 55 57 94 88 68

Inorder traversal
4 5 14 49 55 57 63 68 88 94

Postorder traversal
5 4 57 55 68 88 94 63 49 14

Level-order traversal
14 4 49 5 63 55 94 57 88 68
```

17.25 (*Printing Trees*) Write a recursive method `outputTree` to display a binary tree object on the screen. The method should output the tree row by row, with the top of the tree at the left of the screen and the bottom of the tree toward the right of the screen. Each row is output vertically. For example, the binary tree illustrated in Fig. 17.20 is output as shown in Fig. 17.21.

```
                99
          97
                92
     83
                72
          71
                69
49
                44
          40
                32
     28
                19
          18
                11
```

Fig. 17.21 | Sample output of recursive method `outputTree`.

Note that the rightmost leaf node appears at the top of the output in the rightmost column and the root node appears at the left of the output. Each column of output starts five spaces to the right of the preceding column. Method `outputTree` should receive an argument `totalSpaces` representing the number of spaces preceding the value to be output. (This variable should start at zero so that the root node is output at the left of the screen.) The method uses a modified inorder traversal to output the tree—it starts at the rightmost node in the tree and works back to the left. The algorithm is as follows:

> While the reference to the current node is not null, perform the following:
>
> Recursively call `outputTree` with the right subtree of the current node and `totalSpaces + 5`.
>
> Use a `for` statement to count from 1 to `totalSpaces` and output spaces.
>
> Output the value in the current node.

Set the reference to the current node to refer to the left subtree of the current node. Increment `totalSpaces` by 5.

ANS:

```
// Exercise 17.25 Solution: Tree2.java
// Class Tree which can print itself.
package com.deitel.jhtp7.ch17;

// class TreeNode definition
class TreeNode
{
   // package access members
   TreeNode leftNode; // left node
   int data; // node value
   TreeNode rightNode; // right node

   // constructor initializes data and makes this a leaf node
   public TreeNode( int nodeData )
   {
      data = nodeData;
      leftNode = rightNode = null;  // node has no children
   } // end TreeNode no-argument constructor

   // locate insertion point and insert new node; ignore duplicate values
   public void insert( int insertValue )
   {
      // insert in left subtree
      if ( insertValue < data )
      {
         // insert new TreeNode
         if ( leftNode == null )
            leftNode = new TreeNode( insertValue );
         else // continue traversing left subtree
            leftNode.insert( insertValue );
      } // end if
      else if ( insertValue > data ) // insert in right subtree
      {
         // insert new TreeNode
         if ( rightNode == null )
            rightNode = new TreeNode( insertValue );
         else // continue traversing right subtree
            rightNode.insert( insertValue );
      } // end else if
   } // end method insert
} // end class TreeNode

// class Tree definition
public class Tree2
{
   private TreeNode root;

   // constructor initializes an empty Tree of integers
   public Tree2()
```

```
50    {
51       root = null;
52    } // end Tree no-argument constructor
53
54    // insert a new node in the binary search tree
55    public void insertNode( int insertValue )
56    {
57       if ( root == null )
58          root = new TreeNode( insertValue ); // create the root node here
59       else
60          root.insert( insertValue ); // call the insert method
61    } // end method insertNode
62
63    // begin preorder traversal
64    public void preorderTraversal()
65    {
66       preorderHelper( root );
67    } // end method preorderTraversal
68
69    // recursive method to perform preorder traversal
70    private void preorderHelper( TreeNode node )
71    {
72       if ( node == null )
73          return;
74
75       System.out.printf( "%s ", node.data ); // output node data
76       preorderHelper( node.leftNode );       // traverse left subtree
77       preorderHelper( node.rightNode );      // traverse right subtree
78    } // end method preorderHelper
79
80    // begin inorder traversal
81    public void inorderTraversal()
82    {
83       inorderHelper( root );
84    } // end method inorderTraversal
85
86    // recursive method to perform inorder traversal
87    private void inorderHelper( TreeNode node )
88    {
89       if ( node == null )
90          return;
91
92       inorderHelper( node.leftNode );        // traverse left subtree
93       System.out.printf( "%s ", node.data ); // output node data
94       inorderHelper( node.rightNode );       // traverse right subtree
95    } // end method inorderHelper
96
97    // begin postorder traversal
98    public void postorderTraversal()
99    {
100      postorderHelper( root );
101   } // end method postorderTraversal
102
103   // recursive method to perform postorder traversal
```

```
104    private void postorderHelper( TreeNode node )
105    {
106       if ( node == null )
107          return;
108
109       postorderHelper( node.leftNode );      // traverse left subtree
110       postorderHelper( node.rightNode );     // traverse right subtree
111       System.out.printf( "%s ", node.data ); // output node data
112    } // end method postorderHelper
113
114    // begin printing tree
115    public void outputTree( int totalSpace )
116    {
117       outputTreeHelper( root, totalSpace >= 0 ? totalSpace : 0 );
118    } // end method outputTree
119
120    // recursive method to print tree
121    private void outputTreeHelper( TreeNode currentNode, int spaces )
122    {
123       // recursively print right branch, then left
124       if ( currentNode != null )
125       {
126          outputTreeHelper( currentNode.rightNode, spaces + 5 );
127
128          for ( int k = 1; k <= spaces; k++ )
129             System.out.print( " " );
130
131          System.out.println( currentNode.data );
132          outputTreeHelper( currentNode.leftNode, spaces + 5 );
133       } // end if
134    } // end method outputTreeHelper
135 } // end class Tree2
```

```
1  // Exercise 17.25 Solution: Tree2Test.java
2  // This program tests the Tree2 class.
3  import java.util.Random;
4  import com.deitel.jhtp7.ch17.Tree2;
5
6  public class Tree2Test
7  {
8     public static void main( String args[] )
9     {
10       Tree2 tree = new Tree2();
11       int intVal;
12       Random randomNumber = new Random();
13       System.out.println( "Inserting the following values: " );
14
15       // create Objects to store in tree
16       for ( int i = 1; i <= 10; i++ )
17       {
18          intVal = randomNumber.nextInt( 100 );
19          System.out.printf( "%d ", intVal );
20          tree.insertNode( intVal );
```

```
21         } // end for
22
23         // run three different traversal types
24         System.out.println ( "\n\nPreorder traversal" );
25         tree.preorderTraversal();
26
27         System.out.println ( "\n\nInorder traversal" );
28         tree.inorderTraversal();
29
30         System.out.println ( "\n\nPostorder traversal" );
31         tree.postorderTraversal();
32
33         // print a depiction of the tree
34         System.out.println( "\n\n" );
35         tree.outputTree( 0 );
36      } // end main
37   } // end class Tree2Test
```

```
Inserting the following values:
87 34 69 29 44 58 97 95 14 14

Preorder traversal
87 34 29 14 69 44 58 97 95

Inorder traversal
14 29 34 44 58 69 87 95 97

Postorder traversal
14 29 58 44 69 34 95 97 87


     97
          95
87
          69
                    58
               44
     34
          29
               14
```

Special Section: Building Your Own Compiler

In Exercises 7.34–7.35, we introduced Simpletron Machine Language (SML), and you implemented a Simpletron computer simulator to execute programs written in SML. In this section, we build a compiler that converts programs written in a high-level programming language to SML. This section "ties" together the entire programming process. You will write programs in this new high-level language, compile them on the compiler you build and run them on the simulator you built in Exercise 7.35. You should make every effort to implement your compiler in an object-oriented manner.

17.26 (*The Simple Language*) Before we begin building the compiler, we discuss a simple, yet powerful high-level language similar to early versions of the popular language BASIC. We call the language *Simple*. Every Simple *statement* consists of a *line number* and a Simple *instruction*. Line

numbers must appear in ascending order. Each instruction begins with one of the following Simple *commands*: `rem`, `input`, `let`, `print`, `goto`, `if/goto` or `end` (see Fig. 17.22). All commands except `end` can be used repeatedly. Simple evaluates only integer expressions using the `+`, `-`, `*` and `/` operators. These operators have the same precedence as in Java. Parentheses can be used to change the order of evaluation of an expression.

Command	Example statement	Description
`rem`	`50 rem this is a remark`	Any text following the command `rem` is for documentation purposes only and is ignored by the compiler.
`input`	`30 input x`	Display a question mark to prompt the user to enter an integer. Read that integer from the keyboard and store the integer in `x`.
`let`	`80 let u = 4 * (j - 56)`	Assign `u` the value of `4 * (j - 56)`. Note that an arbitrarily complex expression can appear to the right of the equal sign.
`print`	`10 print w`	Display the value of `w`.
`goto`	`70 goto 45`	Transfer program control to line `45`.
`if/goto`	`35 if i == z goto 80`	Compare `i` and `z` for equality and transfer program control to line `80` if the condition is true; otherwise, continue execution with the next statement.
`end`	`99 end`	Terminate program execution.

Fig. 17.22 | Simple commands.

Our Simple compiler recognizes only lowercase letters. All characters in a Simple file should be lowercase. (Uppercase letters result in a syntax error unless they appear in a `rem` statement, in which case they are ignored.) A *variable name* is a single letter. Simple does not allow descriptive variable names, so variables should be explained in remarks to indicate their use in a program. Simple uses only integer variables. Simple does not have variable declarations—merely mentioning a variable name in a program causes the variable to be declared and initialized to zero. The syntax of Simple does not allow string manipulation (reading a string, writing a string, comparing strings, and so on). If a string is encountered in a Simple program (after a command other than `rem`), the compiler generates a syntax error. The first version of our compiler assumes that Simple programs are entered correctly. Exercise 17.29 asks the reader to modify the compiler to perform syntax error checking.

Simple uses the conditional `if/goto` and unconditional `goto` statements to alter the flow of control during program execution. If the condition in the `if/goto` statement is true, control is transferred to a specific line of the program. The following relational and equality operators are valid in an `if/goto` statement: `<`, `>`, `<=`, `>=`, `==` or `!=`. The precedence of these operators is the same as in Java.

Let us now consider several programs that demonstrate Simple's features. The first program (Fig. 17.23) reads two integers from the keyboard, stores the values in variables `a` and `b` and computes and prints their sum (stored in variable `c`).

```
10 rem   determine and print the sum of two integers
15 rem
20 rem   input the two integers
30 input a
40 input b
45 rem
50 rem   add integers and store result in c
60 let c = a + b
65 rem
70 rem   print the result
80 print c
90 rem   terminate program execution
99 end
```

Fig. 17.23 | Simple program that determines the sum of two integers.

The program of Fig. 17.24 determines and prints the larger of two integers. The integers are input from the keyboard and stored in `s` and `t`. The `if/goto` statement tests the condition `s >= t`. If the condition is true, control is transferred to line `90` and `s` is output; otherwise, `t` is output and control is transferred to the `end` statement in line `99`, where the program terminates.

```
10 rem   determine and print the larger of two integers
20 input s
30 input t
32 rem
35 rem   test if s >= t
40 if s >= t goto 90
45 rem
50 rem   t is greater than s, so print t
60 print t
70 goto 99
75 rem
80 rem   s is greater than or equal to t, so print s
90 print s
99 end
```

Fig. 17.24 | Simple program that finds the larger of two integers.

Simple does not provide a repetition statement (such as Java's `for`, `while` or `do...while`). However, Simple can simulate each of Java's repetition statements by using the `if/goto` and `goto` statements. Figure 17.25 uses a sentinel-controlled loop to calculate the squares of several integers. Each integer is input from the keyboard and stored in variable `j`. If the value entered is the sentinel value `-9999`, control is transferred to line `99`, where the program terminates. Otherwise, `k` is assigned the

square of j, k is output to the screen and control is passed to line 20, where the next integer is input.

```
10 rem   calculate the squares of several integers
20 input j
23 rem
25 rem   test for sentinel value
30 if j == -9999 goto 99
33 rem
35 rem   calculate square of j and assign result to k
40 let k = j * j
50 print k
53 rem
55 rem   loop to get next j
60 goto 20
99 end
```

Fig. 17.25 | Calculate the squares of several integers.

Using the sample programs of Figs. 17.23–17.25 as your guide, write a Simple program to accomplish each of the following:

a) Input three integers, determine their average and print the result.
b) Use a sentinel-controlled loop to input 10 integers and compute and print their sum.
c) Use a counter-controlled loop to input 7 integers, some positive and some negative, and compute and print their average.
d) Input a series of integers and determine and print the largest. The first integer input indicates how many numbers should be processed.
e) Input 10 integers and print the smallest.
f) Calculate and print the sum of the even integers from 2 to 30.
g) Calculate and print the product of the odd integers from 1 to 9.

17.27 (*Building A Compiler; Prerequisites: Complete Exercise 7.34, Exercise 7.35, Exercise 17.12, Exercise 17.13 and Exercise 17.26*) Now that the Simple language has been presented (Exercise 17.26), we discuss how to build a Simple compiler. First, we consider the process by which a Simple program is converted to SML and executed by the Simpletron simulator (see Fig. 17.26). A file containing a Simple program is read by the compiler and converted to SML code. The SML code is output to a file on disk, in which SML instructions appear one per line. The SML file is then loaded into the Simpletron simulator, and the results are sent to a file on disk and to the screen.

Note that the Simpletron program developed in Exercise 7.35 took its input from the keyboard. It must be modified to read from a file so it can run the programs produced by our compiler.

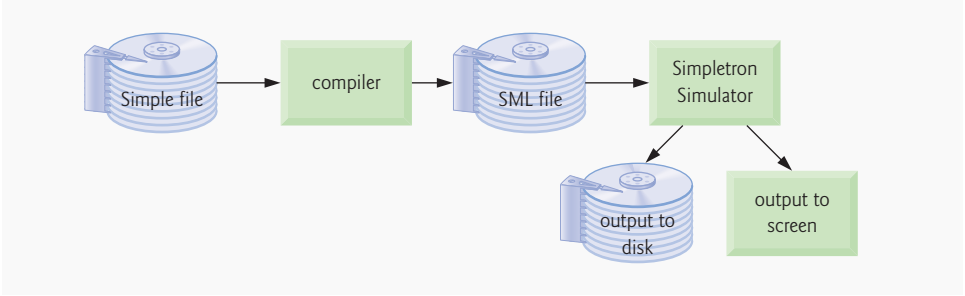

Fig. 17.26 | Writing, compiling and executing a Simple language program.

The Simple compiler performs two *passes* of the Simple program to convert it to SML. The first pass constructs a *symbol table* (object) in which every *line number* (object), *variable name* (object) and *constant* (object) of the Simple program is stored with its type and corresponding location in the final SML code (the symbol table is discussed in detail below). The first pass also produces the corresponding SML instruction object(s) for each of the Simple statements (object, and so on). If the Simple program contains statements that transfer control to a line later in the program, the first pass results in an SML program containing some "unfinished" instructions. The second pass of the compiler locates and completes the unfinished instructions and outputs the SML program to a file.

First Pass

The compiler begins by reading one statement of the Simple program into memory. The line must be separated into its individual *tokens* (i.e., "pieces" of a statement) for processing and compilation. (The `StreamTokenizer` class from the `java.io` package can be used.) Recall that every statement begins with a line number followed by a command. As the compiler breaks a statement into tokens, if the token is a line number, a variable or a constant, it is placed in the symbol table. A line number is placed in the symbol table only if it is the first token in a statement. The `symbolTable` object is an array of `tableEntry` objects representing each symbol in the program. There is no restriction on the number of symbols that can appear in the program. Therefore, the `symbolTable` for a particular program could be large. Make it a 100-element array for now. You can increase or decrease its size once the program is working.

Each `tableEntry` object contains three fields. Field `symbol` is an integer containing the Unicode representation of a variable (remember that variable names are single characters), a line number or a constant. Field `type` is one of the following characters indicating the symbol's type: `'C'` for constant, `'L'` for line number or `'V'` for variable. Field `location` contains the Simpletron memory location (`00` to `99`) to which the symbol refers. Simpletron memory is an array of 100 integers in which SML instructions and data are stored. For a line number, the location is the element in the Simpletron memory array at which the SML instructions for the Simple statement begin. For a variable or constant, the location is the element in the Simpletron memory array in which the variable or constant is stored. Variables and constants are allocated from the end of Simpletron's memory backward. The first variable or constant is stored at location `99`, the next at location `98`, and so on.

The symbol table plays an integral part in converting Simple programs to SML. We learned in Chapter 7 that an SML instruction is a four-digit integer comprised of two parts—the *operation*

code and the *operand*. The operation code is determined by commands in Simple. For example, the simple command `input` corresponds to SML operation code `10` (read), and the Simple command `print` corresponds to SML operation code `11` (write). The operand is a memory location containing the data on which the operation code performs its task (e.g., operation code `10` reads a value from the keyboard and stores it in the memory location specified by the operand). The compiler searches `symbolTable` to determine the Simpletron memory location for each symbol, so the corresponding location can be used to complete the SML instructions.

The compilation of each Simple statement is based on its command. For example, after the line number in a `rem` statement is inserted in the symbol table, the remainder of the statement is ignored by the compiler, because a remark is for documentation purposes only. The `input`, `print`, `goto` and `end` statements correspond to the SML *read, write, branch (*to a specific location) and *halt* instructions. Statements containing these Simple commands are converted directly to SML. (*Note*: A `goto` statement may contain an unresolved reference if the specified line number refers to a statement further into the Simple program file; this is sometimes called a forward reference.)

When a `goto` statement is compiled with an unresolved reference, the SML instruction must be *flagged* to indicate that the second pass of the compiler must complete the instruction. The flags are stored in a 100-element array `flags` of type `int` in which each element is initialized to `-1`. If the memory location to which a line number in the Simple program refers is not yet known (i.e., it is not in the symbol table), the line number is stored in array `flags` in the element with the same index as the incomplete instruction. The operand of the incomplete instruction is set to `00` temporarily. For example, an unconditional branch instruction (making a forward reference) is left as `+4000` until the second pass of the compiler. The second pass will be described shortly.

Compilation of `if/goto` and `let` statements is more complicated than for other statements—they are the only statements that produce more than one SML instruction. For an `if/goto` statement, the compiler produces code to test the condition and to branch to another line if necessary. The result of the branch could be an unresolved reference. Each of the relational and equality operators can be simulated by using SML's *branch zero* and *branch negative* instructions (or possibly a combination of both).

For a `let` statement, the compiler produces code to evaluate an arbitrarily complex arithmetic expression consisting of integer variables and/or constants. Expressions should separate each operand and operator with spaces. Exercise 17.12 and Exercise 17.13 presented the infix-to-postfix conversion algorithm and the postfix evaluation algorithm used by compilers to evaluate expressions. Before proceeding with your compiler, you should complete each of these exercises. When a compiler encounters an expression, it converts the expression from infix notation to postfix notation, then evaluates the postfix expression.

How is it that the compiler produces the machine language to evaluate an expression containing variables? The postfix evaluation algorithm contains a "hook" where the compiler can generate SML instructions rather than actually evaluating the expression. To enable this "hook" in the compiler, the postfix evaluation algorithm must be modified to search the symbol table for each symbol it encounters (and possibly insert it), determine the symbol's corresponding memory location and *push the memory location (instead of the symbol) onto the stack.* When an operator is encountered in the postfix expression, the two memory locations at the top of the stack are popped, and machine language for effecting the operation is produced by using the memory locations as operands. The result of each subexpression is stored in a temporary location in memory and pushed back onto the stack so the evaluation of the postfix expression can continue. When postfix evaluation is complete, the memory location containing the result is the only location left on the stack. This is popped, and SML instructions are generated to assign the result to the variable at the left of the `let` statement.

Second Pass

The second pass of the compiler performs two tasks: Resolve any unresolved references and output the SML code to a file. Resolution of references occurs as follows:

a) Search the `flags` array for an unresolved reference (i.e., an element with a value other than -1).
b) Locate the object in array `symbolTable` containing the symbol stored in the `flags` array (be sure that the type of the symbol is `'L'` for line number).
c) Insert the memory location from field `location` into the instruction with the unresolved reference (remember that an instruction containing an unresolved reference has operand `00`).
d) Repeat steps (a), (b) and (c) until the end of the `flags` array is reached.

After the resolution process is complete, the entire array containing the SML code is output to a disk file with one SML instruction per line. This file can be read by the Simpletron for execution (after the simulator is modified to read its input from a file). Compiling your first Simple program into an SML file and executing that file should give you a real sense of personal accomplishment.

A Complete Example

The following example illustrates complete conversion of a Simple program to SML as it will be performed by the Simple compiler. Consider a Simple program that inputs an integer and sums the values from 1 to that integer. The program and the SML instructions produced by the first pass of the Simple compiler are illustrated in Fig. 17.27. The symbol table constructed by the first pass is shown in Fig. 17.28.

Simple program	SML location and instruction	Description
`5 rem sum 1 to x`	*none*	`rem` ignored
`10 input x`	`00 +1099`	read `x` into location `99`
`15 rem check y == x`	*none*	`rem` ignored
`20 if y == x goto 60`	`01 +2098`	load `y` (`98`) into accumulator
	`02 +3199`	sub `x` (`99`) from accumulator
	`03 +4200`	branch zero to unresolved location
`25 rem increment y`	*none*	`rem` ignored
`30 let y = y + 1`	`04 +2098`	load `y` into accumulator
	`05 +3097`	add `1` (`97`) to accumulator
	`06 +2196`	store in temporary location `96`
	`07 +2096`	load from temporary location `96`
	`08 +2198`	store accumulator in `y`
`35 rem add y to total`	*none*	`rem` ignored

Fig. 17.27 | SML instructions produced after the compiler's first pass. (Part 1 of 2.)

Simple program	SML location and instruction	Description
`40 let t = t + y`	`09 +2095`	load `t` (95) into accumulator
	`10 +3098`	add `y` to accumulator
	`11 +2194`	store in temporary location 94
	`12 +2094`	load from temporary location 94
	`13 +2195`	store accumulator in `t`
`45 rem loop y`	*none*	`rem` ignored
`50 goto 20`	`14 +4001`	branch to location 01
`55 rem output result`	*none*	`rem` ignored
`60 print t`	`15 +1195`	output `t` to screen
`99 end`	`16 +4300`	terminate execution

Fig. 17.27 | SML instructions produced after the compiler's first pass. (Part 2 of 2.)

Symbol	Type	Location
5	L	00
10	L	00
'x'	V	99
15	L	01
20	L	01
'y'	V	98
25	L	04
30	L	04
1	C	97
35	L	09
40	L	09
't'	V	95
45	L	14
50	L	14

Fig. 17.28 | Symbol table for program of Fig. 17.27. (Part 1 of 2.)

Symbol	Type	Location
55	L	15
60	L	15
99	L	16

Fig. 17.28 | Symbol table for program of Fig. 17.27. (Part 2 of 2.)

Most Simple statements convert directly to single SML instructions. The exceptions in this program are remarks, the `if/goto` statement in line `20` and the `let` statements. Remarks do not translate into machine language. However, the line number for a remark is placed in the symbol table in case the line number is referenced in a `goto` statement or an `if/goto` statement. Line `20` of the program specifies that, if the condition `y == x` is true, program control is transferred to line `60`. Since line `60` appears later in the program, the first pass of the compiler has not as yet placed `60` in the symbol table. (Statement line numbers are placed in the symbol table only when they appear as the first token in a statement.) Therefore, it is not possible at this time to determine the operand of the SML *branch zero* instruction at location `03` in the array of SML instructions. The compiler places `60` in location `03` of the `flags` array to indicate that the second pass completes this instruction.

We must keep track of the next instruction location in the SML array because there is not a one-to-one correspondence between Simple statements and SML instructions. For example, the `if/goto` statement of line `20` compiles into three SML instructions. Each time an instruction is produced, we must increment the *instruction counter* to the next location in the SML array. Note that the size of Simpletron's memory could present a problem for Simple programs with many statements, variables and constants. It is conceivable that the compiler will run out of memory. To test for this case, your program should contain a *data counter* to keep track of the location at which the next variable or constant will be stored in the SML array. If the value of the instruction counter is larger than the value of the data counter, the SML array is full. In this case, the compilation process should terminate, and the compiler should print an error message indicating that it ran out of memory during compilation. This serves to emphasize that, although the programmer is freed from the burdens of managing memory by the compiler, the compiler itself must carefully determine the placement of instructions and data in memory and must check for such errors as memory being exhausted during the compilation process.

A Step-by-Step View of the Compilation Process

Let us now walk through the compilation process for the Simple program in Fig. 17.27. The compiler reads the first line of the program

```
5 rem sum 1 to x
```

into memory. The first token in the statement (the line number) is determined using the `StringTokenizer` class. (See Chapter 30 for a discussion of this class.) The token returned by the `StringTokenizer` is converted to an integer by using `static` method `Integer.parseInt()`, so the symbol `5` can be located in the symbol table. If the symbol is not found, it is inserted in the symbol table.

We are at the beginning of the program and this is the first line, and no symbols are in the table yet. Therefore, `5` is inserted into the symbol table as type `L` (line number) and assigned the first location in the SML array (`00`). Although this line is a remark, a space in the symbol table is

still allocated for the line number (in case it is referenced by a `goto` or an `if/goto`). No SML instruction is generated for a `rem` statement, so the instruction counter is not incremented.

```
10 input x
```

is tokenized next. The line number `10` is placed in the symbol table as type `L` and assigned the first location in the SML array (`00` because a remark began the program, so the instruction counter is currently `00`). The command `input` indicates that the next token is a variable (only a variable can appear in an `input` statement). `input` corresponds directly to an SML operation code; therefore, the compiler simply has to determine the location of `x` in the SML array. Symbol `x` is not found in the symbol table, so it is inserted into the symbol table as the Unicode representation of `x`, given type `V` and assigned location `99` in the SML array (data storage begins at `99` and is allocated backward). SML code can now be generated for this statement. Operation code `10` (the SML read operation code) is multiplied by 100, and the location of `x` (as determined in the symbol table) is added to complete the instruction. The instruction is then stored in the SML array at location `00`. The instruction counter is incremented by one, because a single SML instruction was produced.

The statement

```
15 rem   check y == x
```

is tokenized next. The symbol table is searched for line number `15` (which is not found). The line number is inserted as type `L` and assigned the next location in the array, `01`. (Remember that `rem` statements do not produce code, so the instruction counter is not incremented.)

The statement

```
20 if y == x goto 60
```

is tokenized next. Line number `20` is inserted in the symbol table and given type `L` at the next location in the SML array `01`. The command `if` indicates that a condition is to be evaluated. The variable `y` is not found in the symbol table, so it is inserted and given the type `V` and the SML location `98`. Next, SML instructions are generated to evaluate the condition. There is no direct equivalent in SML for the `if/goto`; it must be simulated by performing a calculation using `x` and `y` and branching according to the result. If `y` is equal to `x`, the result of subtracting `x` from `y` is zero, so the *branch zero* instruction can be used with the result of the calculation to simulate the `if/goto` statement. The first step requires that `y` be loaded (from SML location `98`) into the accumulator. This produces the instruction `01 +2098`. Next, `x` is subtracted from the accumulator. This produces the instruction `02 +3199`. The value in the accumulator may be zero, positive or negative. The operator is `==`, so we want to *branch zero.* First, the symbol table is searched for the branch location (`60` in this case), which is not found. So, `60` is placed in the `flags` array at location `03`, and the instruction `03 +4200` is generated. (We cannot add the branch location because we have not yet assigned a location to line `60` in the SML array.) The instruction counter is incremented to `04`.

The compiler proceeds to the statement

```
25 rem   increment y
```

The line number `25` is inserted in the symbol table as type `L` and assigned SML location `04`. The instruction counter is not incremented.

When the statement

```
30 let y = y + 1
```

is tokenized, the line number `30` is inserted in the symbol table as type `L` and assigned SML location `04`. Command `let` indicates that the line is an assignment statement. First, all the symbols on

the line are inserted in the symbol table (if they are not already there). The integer 1 is added to the symbol table as type C and assigned SML location 97. Next, the right side of the assignment is converted from infix to postfix notation. Then the postfix expression (y 1 +) is evaluated. Symbol y is located in the symbol table, and its corresponding memory location is pushed onto the stack. Symbol 1 is also located in the symbol table, and its corresponding memory location is pushed onto the stack. When the operator + is encountered, the postfix evaluator pops the stack into the right operand of the operator and pops the stack again into the left operand of the operator, then produces the SML instructions

```
04 +2098    (load y)
05 +3097    (add 1)
```

The result of the expression is stored in a temporary location in memory (96) with instruction

```
06 +2196    (store temporary)
```

and the temporary location is pushed onto the stack. Now that the expression has been evaluated, the result must be stored in y (i.e., the variable on the left side of =). So, the temporary location is loaded into the accumulator and the accumulator is stored in y with the instructions

```
07 +2096    (load temporary)
08 +2198    (store y)
```

The reader should immediately notice that SML instructions appear to be redundant. We will discuss this issue shortly.

When the statement

```
35 rem   add y to total
```

is tokenized, line number 35 is inserted in the symbol table as type L and assigned location 09.

The statement

```
40 let t = t + y
```

is similar to line 30. The variable t is inserted in the symbol table as type V and assigned SML location 95. The instructions follow the same logic and format as line 30, and the instructions 09 +2095, 10 +3098, 11 +2194, 12 +2094 and 13 +2195 are generated. Note that the result of t + y is assigned to temporary location 94 before being assigned to t (95). Once again, the reader should note that the instructions in memory locations 11 and 12 appear to be redundant. Again, we will discuss this shortly.

The statement

```
45 rem   loop y
```

is a remark, so line 45 is added to the symbol table as type L and assigned SML location 14.

The statement

```
50 goto 20
```

transfers control to line 20. Line number 50 is inserted in the symbol table as type L and assigned SML location 14. The equivalent of goto in SML is the *unconditional branch* (40) instruction that transfers control to a specific SML location. The compiler searches the symbol table for line 20 and finds that it corresponds to SML location 01. The operation code (40) is multiplied by 100, and location 01 is added to it to produce the instruction 14 +4001.

The statement

```
55 rem   output result
```

is a remark, so line 55 is inserted in the symbol table as type L and assigned SML location 15.

The statement

```
60 print t
```

is an output statement. Line number 60 is inserted in the symbol table as type L and assigned SML location 15. The equivalent of print in SML is operation code 11 (*write*). The location of t is determined from the symbol table and added to the result of the operation code multiplied by 100.

The statement

```
99 end
```

is the final line of the program. Line number 99 is stored in the symbol table as type L and assigned SML location 16. The end command produces the SML instruction +4300 (43 is *halt* in SML), which is written as the final instruction in the SML memory array.

This completes the first pass of the compiler. We now consider the second pass. The flags array is searched for values other than -1. Location 03 contains 60, so the compiler knows that instruction 03 is incomplete. The compiler completes the instruction by searching the symbol table for 60, determining its location and adding the location to the incomplete instruction. In this case, the search determines that line 60 corresponds to SML location 15, so the completed instruction 03 +4215 is produced, replacing 03 +4200. The Simple program has now been compiled successfully.

To build the compiler, you will have to perform each of the following tasks:

a) Modify the Simpletron simulator program you wrote in Exercise 7.35 to take its input from a file specified by the user (see Chapter 14). The simulator should output its results to a disk file in the same format as the screen output. Convert the simulator to be an object-oriented program. In particular, make each part of the hardware an object. Arrange the instruction types into a class hierarchy using inheritance. Then execute the program polymorphically simply by telling each instruction to execute itself with an executeInstruction message.

b) Modify the infix-to-postfix evaluation algorithm of Exercise 17.12 to process multidigit integer operands and single-letter variable-name operands. (*Hint*: Class StringTokenizer can be used to locate each constant and variable in an expression, and constants can be converted from strings to integers by using Integer class method parseInt.) [*Note*: The data representation of the postfix expression must be altered to support variable names and integer constants.]

c) Modify the postfix evaluation algorithm to process multidigit integer operands and variable-name operands. Also, the algorithm should now implement the "hook" discussed earlier so that SML instructions are produced rather than directly evaluating the expression. (*Hint*: Class StringTokenizer can be used to locate each constant and variable in an expression, and constants can be converted from strings to integers by using Integer class method parseInt.) [*Note*: The data representation of the postfix expression must be altered to support variable names and integer constants.]

d) Build the compiler. Incorporate parts b) and c) for evaluating expressions in let statements. Your program should contain a method that performs the first pass of the compiler and a method that performs the second pass of the compiler. Both methods can call other methods to accomplish their tasks. Make your compiler as object oriented as possible.

17.28 (*Optimizing the Simple Compiler*) When a program is compiled and converted into SML, a set of instructions is generated. Certain combinations of instructions often repeat themselves, usually in triplets called *productions*. A production normally consists of three instructions, such as *load*, *add* and *store*. For example, Fig. 17.29 illustrates five of the SML instructions that were produced in the compilation of the program in Fig. 17.27. The first three instructions are the production that adds `1` to `y`. Note that instructions `06` and `07` store the accumulator value in temporary location `96`, then load the value back into the accumulator so instruction `08` can store the value in location `98`. Often a production is followed by a load instruction for the same location that was just stored. This code can be *optimized* by eliminating the store instruction and the subsequent load instruction that operate on the same memory location, thus enabling the Simpletron to execute the program faster. Figure 17.30 illustrates the optimized SML for the program of Fig. 17.28. Note that there are four fewer instructions in the optimized code—a memory-space savings of 25%.

```
1  04  +2098  (load)
2  05  +3097  (add)
3  06  +2196  (store)
4  07  +2096  (load)
5  08  +2198  (store)
```

Fig. 17.29 | Unoptimized code from the program of Fig. 19.25.

Simple program	SML location and instruction	Description
`5 rem sum 1 to x`	*none*	`rem` ignored
`10 input x`	`00 +1099`	read `x` into location `99`
`15 rem check y == x`	*none*	`rem` ignored
`20 if y == x goto 60`	`01 +2098`	load `y` (`98`) into accumulator
	`02 +3199`	sub `x` (`99`) from accumulator
	`03 +4211`	branch to location `11` if zero
`25 rem increment y`	*none*	`rem` ignored
`30 let y = y + 1`	`04 +2098`	load `y` into accumulator
	`05 +3097`	add `1` (`97`) to accumulator
	`06 +2198`	store accumulator in `y` (`98`)
`35 rem add y to total`	*none*	`rem` ignored
`40 let t = t + y`	`07 +2096`	load `t` from location (`96`)
	`08 +3098`	add `y` (`98`) accumulator

Fig. 17.30 | Optimized code for the program of Fig. 17.27. (Part 1 of 2.)

Simple program	SML location and instruction	Description
	09 +2196	store accumulator in `t (96)`
`45 rem loop y`	*none*	`rem` ignored
`50 goto 20`	10 +4001	branch to location `01`
`55 rem output result`	*none*	`rem` ignored
`60 print t`	11 +1196	output `t (96)` to screen
`99 end`	12 +4300	terminate execution

Fig. 17.30 | Optimized code for the program of Fig. 17.27. (Part 2 of 2.)

17.29 (*Modifications to the Simple Compiler*) Perform the following modifications to the Simple compiler. Some of these modifications might also require modifications to the Simpletron simulator program written in Exercise 7.35.

a) Allow the remainder operator (`%`) to be used in `let` statements. Simpletron Machine Language must be modified to include a remainder instruction.

b) Allow exponentiation in a `let` statement using `^` as the exponentiation operator. Simpletron Machine Language must be modified to include an exponentiation instruction.

c) Allow the compiler to recognize uppercase and lowercase letters in Simple statements (e.g., `'A'` is equivalent to `'a'`). No modifications to the Simpletron simulator are required.

d) Allow `input` statements to read values for multiple variables such as `input x, y`. No modifications to the Simpletron simulator are required to perform this enhancement to the Simple compiler.

e) Allow the compiler to output multiple values from a single `print` statement, such as `print a, b, c`. No modifications to the Simpletron simulator are required to perform this enhancement.

f) Add syntax-checking capabilities to the compiler so error messages are output when syntax errors are encountered in a Simple program. No modifications to the Simpletron simulator are required.

g) Allow arrays of integers. No modifications to the Simpletron simulator are required to perform this enhancement.

h) Allow subroutines specified by the Simple commands `gosub` and `return`. Command `gosub` passes program control to a subroutine and command `return` passes control back to the statement after the `gosub`. This is similar to a method call in Java. The same subroutine can be called from many `gosub` commands distributed throughout a program. No modifications to the Simpletron simulator are required.

i) Allow repetition statements of the form

```
for x = 2 to 10 step 2
   Simple statements
next
```

This `for` statement loops from `2` to `10` with an increment of `2`. The `next` line marks the end of the body of the `for` line. No modifications to the Simpletron simulator are required.

j) Allow repetition statements of the form

```
for x = 2 to 10
   Simple statements
next
```

This `for` statement loops from `2` to `10` with a default increment of `1`. No modifications to the Simpletron simulator are required.

k) Allow the compiler to process string input and output. This requires the Simpletron simulator to be modified to process and store string values. [*Hint*: Each Simpletron word (i.e., memory location) can be divided into two groups, each holding a two-digit integer. Each two-digit integer represents the Unicode decimal equivalent of a character. Add a machine-language instruction that will print a string beginning at a certain Simpletron memory location. The first half of the Simpletron word at that location is a count of the number of characters in the string (i.e., the length of the string). Each succeeding half-word contains one Unicode character expressed as two decimal digits. The machine-language instruction checks the length and prints the string by translating each two-digit number into its equivalent character.]

l) Allow the compiler to process floating-point values in addition to integers. The Simpletron Simulator must also be modified to process floating-point values.

17.30 (*A Simple Interpreter*) An interpreter is a program that reads a high-level language program statement, determines the operation to be performed by the statement and executes the operation immediately. The high-level language program is not converted into machine language first. Interpreters execute more slowly than compilers do, because each statement encountered in the program being interpreted must first be deciphered at execution time. If statements are contained in a loop, the statements are deciphered each time they are encountered in the loop. Early versions of the BASIC programming language were implemented as interpreters. Most Java programs are run interpretively.

Write an interpreter for the Simple language discussed in Exercise 17.26. The program should use the infix-to-postfix converter developed in Exercise 17.12 and the postfix evaluator developed in Exercise 17.13 to evaluate expressions in a `let` statement. The same restrictions placed on the Simple language in Exercise 17.26 should be adhered to in this program. Test the interpreter with the Simple programs written in Exercise 17.26. Compare the results of running these programs in the interpreter with the results of compiling the Simple programs and running them in the Simpletron simulator built in Exercise 7.35.

17.31 *(Insert/Delete Anywhere in a Linked List)* Our linked list class allowed insertions and deletions at only the front and the back of the linked list. These capabilities were convenient for us when we used inheritance or composition to produce a stack class and a queue class with a minimal amount of code simply by reusing the list class. Linked lists are normally more general than those we provided. Modify the linked list class we developed in this chapter to handle insertions and deletions anywhere in the list.

ANS:

```
1  // Exercise 17.31 Solution: List2.java
2  // ListNode and List class declarations.
3  package com.deitel.jhtp7.ch17;
4
5  // class to represent one node in a list
```

```
6  class ListNode
7  {
8     // package access members; List can access these directly
9     Object data;
10    ListNode nextNode;
11
12    // constructor creates a ListNode that refers to object
13    ListNode( Object object )
14    {
15       this( object, null );
16    } // end ListNode one-argument constructor
17
18    // constructor creates ListNode that refers to
19    // Object and to next ListNode
20    ListNode( Object object, ListNode node )
21    {
22       data = object;
23       nextNode = node;
24    } // end ListNode two-argument constructor
25
26    // return reference to data in node
27    Object getObject()
28    {
29       return data; // return Object in this node
30    } // end method getObject
31
32    // return reference to next node in list
33    ListNode getNext()
34    {
35       return nextNode; // get next node
36    } // end method getNext
37
38    // set reference to next node in list
39    void setNext( ListNode next )
40    {
41       nextNode = next;
42    } // end method setNext
43 } // end class ListNode
44
45 // class List2 declaration
46 public class List2
47 {
48    private ListNode firstNode;
49    private ListNode lastNode;
50    private String name; // string like "list" used in printing
51
52    // constructor creates empty List with "list" as the name
53    public List2()
54    {
55       this( "list" );
56    } // end List no-argument constructor
57
58    // constructor creates an empty List with a name
59    public List2( String listName )
60    {
```

```
61          name = listName;
62          firstNode = lastNode = null;
63       } // end List one-argument constructor
64
65       // insert Object at front of List
66       public void insertAtFront( Object insertItem )
67       {
68          if ( isEmpty() ) // firstNode and lastNode refer to same object
69             firstNode = lastNode = new ListNode( insertItem );
70          else // firstNode refers to new node
71             firstNode = new ListNode( insertItem, firstNode );
72       } // end method insertAtFront
73
74       // insert Object at end of List
75       public void insertAtBack( Object insertItem )
76       {
77          if ( isEmpty() ) // firstNode and lastNode refer to same Object
78             firstNode = lastNode = new ListNode( insertItem );
79          else // lastNode's nextNode refers to new node
80             lastNode = lastNode.nextNode = new ListNode( insertItem );
81       } // end method insertAtBack
82
83       // remove first node from List
84       public Object removeFromFront() throws EmptyListException
85       {
86          if ( isEmpty() ) // throw exception if List is empty
87             throw new EmptyListException( name );
88
89          Object removedItem = firstNode.data; // retrieve data being removed
90
91          // update references firstNode and lastNode
92          if ( firstNode == lastNode )
93             firstNode = lastNode = null;
94          else
95             firstNode = firstNode.nextNode;
96
97          return removedItem; // return removed node data
98       } // end method removeFromFront
99
100      // remove last node from List
101      public Object removeFromBack() throws EmptyListException
102      {
103         if ( isEmpty() ) // throw exception if List is empty
104            throw new EmptyListException( name );
105
106         Object removedItem = lastNode.data; // retrieve data being removed
107
108         // update references firstNode and lastNode
109         if ( firstNode == lastNode )
110            firstNode = lastNode = null;
111         else // locate new last node
112         {
113            ListNode current = firstNode;
114
115            // loop while current node does not refer to lastNode
```

```
116          while ( current.nextNode != lastNode )
117             current = current.nextNode;
118
119          lastNode = current; // current is new lastNode
120          current.nextNode = null;
121       } // end else
122
123       return removedItem; // return removed node data
124    } // end method removeFromBack
125
126    // determine whether list is empty
127    public boolean isEmpty()
128    {
129       return firstNode == null; // return true if List is empty
130    } // end method isEmpty
131
132    // output List contents
133    public void print()
134    {
135       if ( isEmpty() )
136       {
137          System.out.printf( "Empty %s\n", name );
138          return;
139       } // end if
140
141       System.out.printf( "The %s is: ", name );
142       ListNode current = firstNode;
143
144       // while not at end of list, output current node's data
145       while ( current != null )
146       {
147          System.out.printf( "%s ", current.data );
148          current = current.nextNode;
149       } // end while
150
151       System.out.println( "\n" );
152    } // end method print
153
154    // delete node
155    public boolean deleteNode( Integer value )
156    {
157       // empty list
158       if ( isEmpty() )
159          return false;
160       else
161       {
162          // delete first node
163          if ( ( ( Integer ) firstNode.getObject() ) == value )
164          {
165             removeFromFront();
166             return true;
167          } // end if
168          else if ( ( ( Integer ) lastNode.getObject() ) == value )
169          {
170             removeFromBack(); // delete last node
```

```
171             return true;
172          } // end else if
173          else // delete internal node
174          {
175             ListNode current = firstNode.getNext();
176             ListNode previous = firstNode;
177
178             // search for node to be deleted
179             while ( current != lastNode &&
180                ( ( Integer ) current.getObject() ) < value )
181             {
182                previous = current;
183                current = current.getNext();
184             } // end while
185
186             // node to be deleted found in list
187             if ( ( ( Integer ) current.getObject() ) == value )
188             {
189                previous.setNext( current.getNext() );
190                return true;
191             } // end if
192             else // node with appropriate value is not in list
193                return false;
194          } // end else
195       } // end else
196    } // end method deleteNode
197
198    // insert new node
199    public void insertInOrder( Integer value )
200    {
201       // empty list
202       if ( isEmpty() )
203       {
204          ListNode newNode = new ListNode( value );
205          firstNode = lastNode = newNode;
206       } // end if
207       else
208       {
209          // insert at front
210          if ( ( ( Integer ) firstNode.getObject() ) > value )
211             insertAtFront( value );
212          else if ( ( ( Integer ) lastNode.getObject() ) < value )
213             insertAtBack( value ); // insert at back
214          else // insert within list
215          {
216             ListNode current = firstNode.getNext();
217             ListNode previous = firstNode;
218             ListNode newNode = new ListNode( value );
219
220             // search for appropriate insertion location
221             while ( current != lastNode &&
222                ( ( Integer ) current.getObject() ) < value )
223             {
224                previous = current;
225                current = current.getNext();
```

```
            } // end while

            // insert new node
            previous.setNext( newNode );
            newNode.setNext( current );
         } // end else
      } // end else
   } // end method insertInOrder
} // end class List2
```

```
// Exercise 17.31 Solution: ListTest2.java
// Program allows insertion and deletion anywhere in a linked list.
import com.deitel.jhtp7.ch17.List2;
import com.deitel.jhtp7.ch17.EmptyListException;

public class ListTest2
{
   public static void main( String args[] )
   {
      List2 list = new List2();

      // create list
      for ( int a = 20; a >= 2; a -= 2 )
         list.insertInOrder( new Integer( a ) );

      // print list before insertion
      System.out.println( "Before insertion:" );
      list.print();

      int i = 7; // insert new object
      list.insertInOrder( i );
      System.out.printf( "%d inserted.\n", i );

      // print list after insertion
      System.out.println( "After insertion:" );
      list.print();

      try
      {
         // delete node
         if ( list.deleteNode( i ) )
            System.out.printf( "%d deleted.\n", i );
         else
            System.out.printf( "%d not deleted.\n", i );

         // print list after deletion
         System.out.println( "After deletion:" );
         list.print();
         System.out.println();
      } // end try
      catch ( EmptyListException exception )
      {
         System.err.println( "\n" + exception.toString() );
      } // end catch
```

```
45      } // end main
46   } // end class ListTest2
```

```
Before insertion:
The list is: 2 4 6 8 10 12 14 16 18 20

7 inserted.
After insertion:
The list is: 2 4 6 7 8 10 12 14 16 18 20

7 deleted.
After deletion:
The list is: 2 4 6 8 10 12 14 16 18 20
```

17.32 *(Lists and Queues without Tail References)* Our implementation of a linked list (Fig. 17.3) used both a `firstNode` and a `lastNode`. The `lastNode` was useful for the `insertAtBack` and `removeFromBack` methods of the `List` class. The `insertAtBack` method corresponds to the `enqueue` method of the `Queue` class.

Rewrite the `List` class so that it does not use a `lastNode`. Thus, any operations on the tail of a list must begin searching the list from the front. Does this affect our implementation of the `Queue` class (Fig. 17.13)?

ANS:

```
1  // Exercise 17.32 Solution: List.java
2  // An implementation of a list without a lastNode.
3  package com.deitel.jhtp7.ch17;
4
5  // class to represent one node in a list
6  class ListNode
7  {
8     // package access members; List can access these directly
9     Object data;
10    ListNode nextNode;
11
12    // constructor creates a ListNode2 that refers to object
13    ListNode( Object object )
14    {
15       this( object, null );
16    } // end ListNode2 one-argument constructor
17
18    // constructor creates ListNode2 that refers to
19    // Object and to next ListNode2
20    ListNode( Object object, ListNode node )
21    {
22       data = object;
23       nextNode = node;
24    } // end ListNode2 two-argument constructor
25
26    // return reference to data in node
27    Object getObject()
28    {
29       return data; // return Object in this node
```

```
30      } // end method getObject
31
32      // return reference to next node in list
33      ListNode getNext()
34      {
35         return nextNode; // get next node
36      } // end method getNext
37
38      // set reference to next node in list
39      public void setNext( ListNode next )
40      {
41         nextNode = next; // set next node
42      } // end method setNext
43   } // end class ListNode2
44
45   public class List
46   {
47      protected ListNode firstNode;
48      private String name;  // String like "list" used in printing
49
50      // no-argument constructor
51      public List()
52      {
53         this( "list" );
54      } // end no-argument List constructor
55
56      public List( String s )
57      {
58         name = s;
59         firstNode = null;
60      } // end one-argument List constructor
61
62      // insert an object at the front of the list
63      public void insertAtFront( Object insertItem )
64      {
65         if ( isEmpty() )
66            firstNode = new ListNode( insertItem );
67         else
68            firstNode = new ListNode( insertItem, firstNode );
69      } // end method insertAtFront
70
71      // insert an object at the end of the list
72      public void insertAtBack( Object insertItem )
73      {
74         if ( isEmpty() )
75            firstNode = new ListNode( insertItem );
76         else if ( firstNode.nextNode == null )
77            firstNode.nextNode = new ListNode( insertItem );
78         else
79         {
80            ListNode temp = firstNode;
81
82            while ( temp.nextNode != null )
83               temp = temp.nextNode;
84
```

```
85            temp.nextNode = new ListNode( insertItem );
86         } // end else
87      } // end method insertAtBack
88
89      // remove the first node from the list.
90      public Object removeFromFront() throws EmptyListException
91      {
92         Object removeItem = null;
93
94         if ( isEmpty() )
95            throw new EmptyListException( name );
96
97         removeItem = firstNode.data;  // retrieve the data
98         firstNode = firstNode.nextNode;
99
100        return removeItem;
101     } // end method removeFromFront
102
103     // remove the last node from the list.
104     public Object removeFromBack() throws EmptyListException
105     {
106        Object removeItem = null;
107
108        if ( isEmpty() )
109           throw new EmptyListException( name );
110
111        if ( firstNode.nextNode == null ) // list has only one node
112        {
113           removeItem = firstNode.data; // retrieve the data
114           firstNode = null;
115        } // end if
116        else // list has more node
117        {
118           ListNode temp = firstNode;
119
120           while ( temp.nextNode.nextNode != null )
121              temp = temp.nextNode;
122
123           removeItem = temp.nextNode.data; // retrieve the data
124           temp.nextNode = null;
125        } // end else
126
127        return removeItem;
128     } // end method removeFromBack
129
130     // return true if the list is empty
131     public boolean isEmpty()
132     {
133        return firstNode == null;
134     } // end method isEmpty
135
136     // output the list contents
137     public void print()
138     {
139        if ( isEmpty() )
```

```
140        {
141           System.out.println();
142           return;
143        } // end if
144
145        System.out.printf( "The %s is: ", name );
146        ListNode current = firstNode;
147
148        while ( current != null )
149        {
150           System.out.printf( " %s", current.data );
151           current = current.nextNode;
152        } // end while
153
154        System.out.printf( "\n" );
155     } // end method print
156
157     // return first node
158     public ListNode getFirstNode()
159     {
160        return firstNode;
161     } // end method getFirstNode
162  } // end class List
```

```
1  // Exercise 17.32 Solution: QueueInheritance.java
2  // Class QueueInheritance extends class List.
3  package com.deitel.jhtp7.ch17;
4
5  public class QueueInheritance extends List
6  {
7     // no-argument constructor
8     public QueueInheritance()
9     {
10       super( "queue" );
11    } // end no-argument QueueInheritance constructor
12
13    // add object to queue
14    public void enqueue( Object object )
15    {
16       insertAtBack( object );
17    } // end method enqueue
18
19    // remove object from queue
20    public Object dequeue() throws EmptyListException
21    {
22       return removeFromFront();
23    } // end method dequeue
24 } // end class QueueInheritance
```

```
1  // Exercise 17.32 Solution: QueueInheritanceTest.java
2  // Class QueueInheritanceTest.
3  import com.deitel.jhtp7.ch17.QueueInheritance;
```

```
4  import com.deitel.jhtp7.ch17.EmptyListException;
5
6  public class QueueInheritanceTest
7  {
8     public static void main( String args[] )
9     {
10       QueueInheritance queue = new QueueInheritance();
11
12       // use enqueue method
13       queue.enqueue( -1 );
14       queue.print();
15       queue.enqueue( 0 );
16       queue.print();
17       queue.enqueue( 1 );
18       queue.print();
19       queue.enqueue( 5 );
20       queue.print();
21
22       // remove objects from queue
23       try
24       {
25          Object removedObject = null;
26
27          while ( true )
28          {
29             removedObject = queue.dequeue(); // use dequeue method
30             System.out.printf( "%s dequeued\n", removedObject );
31             queue.print();
32          } // end while
33       } // end try
34       catch ( EmptyListException emptyListException )
35       {
36          emptyListException.printStackTrace();
37       } // end catch
38    } // end main
39 } // end class QueueInheritanceTest
```

```
The queue is:  -1
The queue is:  -1 0
The queue is:  -1 0 1
The queue is:  -1 0 1 5
-1 dequeued
The queue is:  0 1 5
0 dequeued
The queue is:  1 5
1 dequeued
The queue is:  5
5 dequeued

com.deitel.jhtp7.ch17.EmptyListException: queue is empty
        at com.deitel.jhtp7.ch17.List.removeFromFront(List.java:95)
        at com.deitel.jhtp7.ch17.QueueInheritance.dequeue(QueueInher-
itance.java:22)
        at QueueInheritanceTest.main(QueueInheritanceTest.java:29)
```

17.33 *(Performance of Binary Tree Sorting and Searching)* One problem with the binary tree sort is that the order in which the data is inserted affects the shape of the tree—for the same collection of data, different orderings can yield binary trees of dramatically different shapes. The performance of the binary tree sorting and searching algorithms is sensitive to the shape of the binary tree. What shape would a binary tree have if its data were inserted in increasing order? in decreasing order? What shape should the tree have to achieve maximal searching performance?

17.34 *(Indexed Lists)* As presented in the text, linked lists must be searched sequentially. For large lists, this can result in poor performance. A common technique for improving list-searching performance is to create and maintain an index to the list. An index is a set of references to key places in the list. For example, an application that searches a large list of names could improve performance by creating an index with 26 entries—one for each letter of the alphabet. A search operation for a last name beginning with 'Y' would then first search the index to determine where the 'Y' entries begin, then "jump into" the list at that point and search linearly until the desired name was found. This would be much faster than searching the linked list from the beginning. Use the `List` class of Fig. 17.3 as the basis of an `IndexedList` class.

Write a program that demonstrates the operation of indexed lists. Be sure to include methods `insertInIndexedList`, `searchIndexedList` and `deleteFromIndexedList`.

```
1  // Exercise 17.34 Solution: List2.java
2  // ListNode2 and List2 class definitions.
3  package com.deitel.jhtp7.ch17;
4
5  // class to represent one node in a list
6  class ListNode2
7  {
8     // package access members; List can access these directly
9     Object data;
10    ListNode2 nextNode;
11
12    // constructor creates a ListNode2 that refers to object
13    ListNode2( Object object )
14    {
15       this( object, null );
16    } // end ListNode2 one-argument constructor
17
18    // constructor creates ListNode2 that refers to
19    // Object and to next ListNode2
20    ListNode2( Object object, ListNode2 node )
21    {
22       data = object;
23       nextNode = node;
24    } // end ListNode2 two-argument constructor
25
26    // return reference to data in node
27    Object getObject()
28    {
29       return data; // return Object in this node
30    } // end method getObject
31
32    // return reference to next node in list
33    ListNode2 getNext()
34    {
```

```
35          return nextNode; // get next node
36       } // end method getNext
37
38       // set reference to next node in list
39       public void setNext( ListNode2 next )
40       {
41          nextNode = next; // set next node
42       } // end method setNext
43    } // end class ListNode2
44
45    // class List2 declaration
46    public class List2
47    {
48       private ListNode2 firstNode;
49       private ListNode2 lastNode;
50       private String name; // string like "List" used in printing
51
52       // constructor creates empty List with "list" as the name
53       public List2()
54       {
55          this( "list" );
56       } // end List2 no-argument constructor
57
58       // constructor creates an empty List with a name
59       public List2( String listName )
60       {
61          name = listName;
62          firstNode = lastNode = null;
63       } // end List2 one-argument constructor
64
65       // insert Object at front of List
66       public void insertAtFront( Object insertItem )
67       {
68          if ( isEmpty() ) // firstNode and lastNode refer to same object
69             firstNode = lastNode = new ListNode2( insertItem );
70          else // firstNode refers to new node
71             firstNode = new ListNode2( insertItem, firstNode );
72       } // end method insertAtFront
73
74       // insert Object at end of List
75       public void insertAtBack( Object insertItem )
76       {
77          if ( isEmpty() ) // firstNode and lastNode refer to same Object
78             firstNode = lastNode = new ListNode2( insertItem );
79          else // lastNode's nextNode refers to new node
80             lastNode = lastNode.nextNode = new ListNode2( insertItem );
81       } // end method insertAtBack
82
83       // remove first node from List
84       public Object removeFromFront() throws EmptyListException
85       {
86          if ( isEmpty() ) // throw exception if List is empty
87             throw new EmptyListException( name );
88
```

```
89          Object removedItem = firstNode.data; // retrieve data being removed
90
91          // update references firstNode and lastNode
92          if ( firstNode == lastNode )
93             firstNode = lastNode = null;
94          else
95             firstNode = firstNode.nextNode;
96
97          return removedItem; // return removed node data
98       } // end method removeFromFront
99
100      // remove last node from List
101      public Object removeFromBack() throws EmptyListException
102      {
103         if ( isEmpty() ) // throw exception if List is empty
104            throw new EmptyListException( name );
105
106         Object removedItem = lastNode.data; // retrieve data being removed
107
108         // update references firstNode and lastNode
109         if ( firstNode == lastNode )
110            firstNode = lastNode = null;
111         else // locate new last node
112         {
113            ListNode2 current = firstNode;
114
115            // loop while current node does not refer to lastNode
116            while ( current.nextNode != lastNode )
117               current = current.nextNode;
118
119            lastNode = current; // current is new lastNode
120            current.nextNode = null;
121         } // end else
122
123         return removedItem; // return removed node data
124      } // end method removeFromBack
125
126      // determine whether list is empty
127      public boolean isEmpty()
128      {
129         return firstNode == null; // return true if List is empty
130      } // end method isEmpty
131
132      // output List contents
133      public void print()
134      {
135         if ( isEmpty() )
136         {
137            System.out.printf( "Empty %s\n", name );
138            return;
139         } // end if
140
141         System.out.printf( "The %s is: ", name );
142         ListNode2 current = firstNode;
143
```

```
144        // while not at end of list, output current node's data
145        while ( current != null )
146        {
147           System.out.printf( "%s ", current.data );
148           current = current.nextNode;
149        } // end while
150
151        System.out.println( "\n" );
152     } // end method print
153
154     // return first list node
155     public ListNode2 getFirstNode()
156     {
157        return firstNode;
158     } // end method getFirstNode
159  } // end class List2
```

```
 1  // Exercise 17.34 Solution: IndexedList.java
 2  // An implementation of an indexed list.
 3  package com.deitel.jhtp7.ch17;
 4
 5  public class IndexedList
 6  {
 7     private List2[] indexer;
 8
 9     // no-argument constructor
10     public IndexedList()
11     {
12        indexer = new List2[ 26 ];
13
14        // initialize indexer
15        for ( int i = 0; i < 26; i++ )
16           indexer[ i ] = new List2();
17     } // end no-argument IndexedList constructor
18
19     // print list element with index
20     public void print()
21     {
22        for ( int i = 0; i < 26; i++ )
23        {
24           System.out.printf( "%c: ", ( char )( i + 'a' ) );
25           indexer[ i ].print();
26        } // end for
27     } // end method print
28
29     // return index of string
30     private int getIndex( String name )
31     {
32        return name.charAt( 0 ) - 'a';
33     } // end method getIndex
34
35     // insert string to indexed list
36     public void insertInIndexedList( String name )
```

```
37      {
38         int index = getIndex( name ); // find the index for the string
39
40         // get the list from the index and the first node
41         List2 start = indexer[ index ];
42         ListNode2 front = start.getFirstNode();
43
44         // if the list is empty
45         if ( front == null )
46            start.insertAtFront( name ); // add to the front of the list
47         else if ( front.getNext() == null ) // list has only one element
48         {
49            // determine whether to add to the front or to the back
50            if ( name.compareTo( ( String )front.getObject() ) > 0 )
51               start.insertAtBack( name );
52            else
53               start.insertAtFront( name );
54         } // end else if
55         else // list has more than one element
56         {
57            // find the place in the list to insert the new element
58            while ( front.getNext().getNext() != null &&
59               name.compareTo( ( String )front.getNext().getObject() ) > 0 )
60               front = front.getNext();
61
62            // if we hit the end
63            if ( front.getNext().getNext() == null &&
64               name.compareTo( ( String )front.getNext().getObject() ) > 0 )
65               start.insertAtBack( name ); // add to the back
66            else
67            {
68               // insert in the middle of the list
69               ListNode2 insert = new ListNode2( name );
70               insert.setNext( front.getNext() );
71               front.setNext( insert );
72            } // end else
73         } // end else
74      } // end insertInIndexedList
75
76      // search string in indexed list
77      public boolean searchIndexedList( String name )
78      {
79         int index = getIndex( name ); // find the index into the list
80         List2 start = indexer[ index ]; // get the list from the index
81         ListNode2 front = start.getFirstNode(); // get the start of the list
82
83         // if the list is empty
84         if ( front == null )
85            return false;
86
87         // find the spot in the list
88         while ( front.getNext() != null &&
89            !name.equals( front.getObject() ) )
90            front = front.getNext();
91
```

```
92          // if we found the item
93          if ( name.equals( front.getObject() ) )
94             return true;
95
96          return false;
97       } // end method searchIndexedList
98
99       // delete string from indexed list
100      public void deleteFromIndexedList( String name )
101      {
102         int index = getIndex( name ); // find the index into the list
103         List2 start = indexer[ index ]; // get the list from the index
104         ListNode2 front = start.getFirstNode(); // get the start of the list
105
106         // if the list is empty
107         if ( front == null )
108            return;
109
110         // if the first item is the node to delete
111         if ( name.equals( front.getObject() ) )
112         {
113            start.removeFromFront();
114            return;
115         } // end if
116
117         // if the list has only one item (which is not the item to delete)
118         if ( front.getNext() == null )
119            return;
120
121         // find the spot in the list to remove
122         while ( front.getNext().getNext() != null &&
123            !name.equals( front.getNext().getObject() ) )
124            front = front.getNext();
125
126         // if we hit the end of the list
127         if ( !name.equals( front.getNext().getObject() ) )
128            return;
129
130         front.setNext( front.getNext().getNext() ); // remove the item
131      } // end method deleteFromIndexedList
132   } // end class IndexedList
```

```
1   // Exercise 17.34 Solution: IndexedTest.java
2   // Program tests an indexed list
3   import com.deitel.jhtp7.ch17.IndexedList;
4
5   public class IndexedTest
6   {
7      public static void main( String args[] )
8      {
9         String[] strings = { "ah", "acme", "apple", "fish", "orange",
10           "pickle", "bread", "steak", "green", "tree", "query", "yellow",
11           "frog", "leaf", "potato", "chicken", "duck", "jelly", "jam",
```

```
12          "butter", "margarine", "hello", "icecream", "bagel", "elephant",
13          "corn", "knife", "liver", "onions", "carrots", "broccoli",
14          "salmon", "blue", "black", "red", "silver", "grey", "white",
15          "brown", "purple", "turkey", "eagle", "hawk", "falcon",
16          "ostrich", "horse", "pig", "zebra", "lion", "tiger" };
17
18       IndexedList list = new IndexedList();
19
20       // create list
21       for ( int a = 0; a < 50; a++ )
22          list.insertInIndexedList( strings[ a ] );
23
24       // print list before insertion
25       System.out.println( "Indexed list:" );
26       list.print();
27       System.out.print( "\n" );
28
29       String insert = "bear"; // insert new object
30       list.insertInIndexedList( insert );
31       System.out.printf( "%s inserted.\n", insert );
32
33       // print list after insertion
34       System.out.println( "After insertion:" );
35       list.print();
36       System.out.print( "\n" );
37
38       // try to find the string "green"
39       if ( list.searchIndexedList( "green" ) )
40          System.out.println( "green found in list.\n" );
41       else
42          System.out.println( "green not found in list.\n" );
43
44       // try to find the string "grass"
45       if ( list.searchIndexedList( "grass" ) )
46          System.out.println( "grass found in list.\n" );
47       else
48          System.out.println( "grass not found in list.\n" );
49
50       // delete three items from the list
51       list.deleteFromIndexedList( "tiger" );
52       list.deleteFromIndexedList( "leopard" );
53       list.deleteFromIndexedList( "lion" );
54
55       System.out.println( "List after deletions:" );
56       list.print();
57    } // end main
58 } // end class IndexedTest
```

```
Indexed list:
a: The list is: acme ah apple

b: The list is: bread bagel black blue broccoli brown butter

c: The list is: chicken carrots corn

d: The list is: duck

e: The list is: eagle elephant

f: The list is: fish falcon frog

g: The list is: green grey

h: The list is: hawk hello horse

i: The list is: icecream

j: The list is: jam jelly

k: The list is: knife

l: The list is: leaf lion liver

m: The list is: margarine

n: Empty list
o: The list is: onions orange ostrich

p: The list is: pickle pig potato purple

q: The list is: query

r: The list is: red

s: The list is: salmon silver steak

t: The list is: tree tiger turkey

u: Empty list
v: Empty list
w: The list is: white

x: Empty list
y: The list is: yellow

z: The list is: zebra


bear inserted.
```

```
After insertion:
a: The list is: acme ah apple

b: The list is: bread bagel bear black blue broccoli brown butter

c: The list is: chicken carrots corn

d: The list is: duck

e: The list is: eagle elephant

f: The list is: fish falcon frog

g: The list is: green grey

h: The list is: hawk hello horse

i: The list is: icecream

j: The list is: jam jelly

k: The list is: knife

l: The list is: leaf lion liver

m: The list is: margarine

n: Empty list
o: The list is: onions orange ostrich

p: The list is: pickle pig potato purple

q: The list is: query

r: The list is: red

s: The list is: salmon silver steak

t: The list is: tree tiger turkey

u: Empty list
v: Empty list
w: The list is: white

x: Empty list
y: The list is: yellow

z: The list is: zebra


green found in list.

grass not found in list.
```

```
List after deletions:
a: The list is: acme ah apple

b: The list is: bread bagel bear black blue broccoli brown butter

c: The list is: chicken carrots corn

d: The list is: duck

e: The list is: eagle elephant

f: The list is: fish falcon frog

g: The list is: green grey

h: The list is: hawk hello horse

i: The list is: icecream

j: The list is: jam jelly

k: The list is: knife

l: The list is: leaf liver

m: The list is: margarine

n: Empty list
o: The list is: onions orange ostrich

p: The list is: pickle pig potato purple

q: The list is: query

r: The list is: red

s: The list is: salmon silver steak

t: The list is: tree turkey

u: Empty list
v: Empty list
w: The list is: white

x: Empty list
y: The list is: yellow

z: The list is: zebra
```

17.35 In Section 17.7, we created a stack class from class `List` with inheritance (Fig. 17.10) and with composition (Fig. 17.12). In Section 17.8 we created a queue class from class `List` with com-

position (Fig. 17.13). Create a queue class by inheriting from class `List`. What are the differences between this class and the one we created with composition?

ANS:

```
// Exercise 17.35 Solution: QueueInheritance.java
// Class QueueInheritance extends class List.
package com.deitel.jhtp7.ch17;

public class QueueInheritance extends List
{
   // no-argument constructor
   public QueueInheritance()
   {
      super( "queue" );
   } // end no-argument QueueInheritance constructor

   // add object to queue
   public void enqueue( Object object )
   {
      insertAtBack( object );
   } // end method enqueue

   // remove object from queue
   public Object dequeue() throws EmptyListException
   {
      return removeFromFront();
   } // end method dequeue
} // end class QueueInheritance
```

```
// Exercise 17.35 Solution: QueueInheritanceTest.java
// Class QueueInheritanceTest.
import com.deitel.jhtp7.ch17.QueueInheritance;
import com.deitel.jhtp7.ch17.EmptyListException;

public class QueueInheritanceTest
{
   public static void main( String args[] )
   {
      QueueInheritance queue = new QueueInheritance();

      // use enqueue method
      queue.enqueue( -1 );
      queue.print();
      queue.enqueue( 1 );
      queue.print();
      queue.enqueue( 0 );
      queue.print();
      queue.enqueue( 5 );
      queue.print();

      // remove objects from queue
      try
```

```
24         {
25            Object removedObject = null;
26
27            while ( true )
28            {
29               removedObject = queue.dequeue(); // use dequeue method
30               System.out.printf(
31                  "%s dequeued\n", removedObject.toString() );
32               queue.print();
33            } // end while
34         } // end try
35         catch ( EmptyListException emptyListException )
36         {
37            emptyListException.printStackTrace();
38         } // end catch
39      } // end main
40   } // end class QueueInheritanceTest
```

```
The queue is: -1

The queue is: -1 1

The queue is: -1 1 0

The queue is: -1 1 0 5

-1 dequeued
The queue is: 1 0 5

1 dequeued
The queue is: 0 5

0 dequeued
The queue is: 5

5 dequeued
Empty queue
com.deitel.jhtp7.ch17.EmptyListException: queue is empty
        at com.deitel.jhtp7.ch17.List.removeFromFront(List.java:81)
        at com.deitel.jhtp7.ch17.QueueInheritance.dequeue(QueueInher-
itance.java:22)
        at QueueInheritanceTest.main(QueueInheritanceTest.java:29)
```