```
1  // Fig. 23.1: VectorTest.java
2  // Testing the Vector class of the java.util package
3  import java.util.*;
4  import java.awt.*;
5  import java.awt.event.*;
6  import javax.swing.*;
7
8  public class VectorTest extends JFrame {
9
10    public VectorTest()
11    {
12       super( "Vector Example" );
13
```

Fig. 23.1 Demonstrating class **`Vector`** of package **`java.util`** (part 1 of 5).

```
14          final JLabel status = new JLabel();
15          Container c = getContentPane();
16          final Vector v = new Vector( 1 );
17
18          c.setLayout( new FlowLayout() );
19
20          c.add( new JLabel( "Enter a string" ) );
21          final JTextField input = new JTextField( 10 );
22          c.add( input );
23
24          JButton addBtn = new JButton( "Add" );
25          addBtn.addActionListener(
26             new ActionListener() {
27                public void actionPerformed( ActionEvent e )
28                {
29                   v.addElement( input.getText() );
30                   status.setText( "Added to end: " +
31                                   input.getText() );
32                   input.setText( "" );
33                }
34             }
35          );
36          c.add( addBtn );      // add the input value
37
38          JButton removeBtn = new JButton( "Remove" );
39          removeBtn.addActionListener(
40             new ActionListener() {
41                public void actionPerformed( ActionEvent e )
42                {
43                   if ( v.removeElement( input.getText() ) )
44                      status.setText( "Removed: " +
45                                      input.getText() );
46                   else
47                      status.setText( input.getText() +
48                                      " not in vector" );
49                }
50             }
51          );
52          c.add( removeBtn );
53
54          JButton firstBtn = new JButton( "First" );
55          firstBtn.addActionListener(
56             new ActionListener() {
57                public void actionPerformed( ActionEvent e )
58                {
59                   try {
60                      status.setText( "First element: " +
61                                      v.firstElement() );
62                   }
```

Fig. 23.1 Demonstrating class **`Vector`** of package **`java.util`** (part 2 of 5).

```
63                catch ( NoSuchElementException exception ) {
64                   status.setText( exception.toString() );
65                }
66             }
67          }
68       );
69       c.add( firstBtn );
70
71       JButton lastBtn = new JButton( "Last" );
72       lastBtn.addActionListener(
73          new ActionListener() {
74             public void actionPerformed( ActionEvent e )
75             {
76                try {
77                   status.setText( "Last element: " +
78                                   v.lastElement() );
79                }
80                catch ( NoSuchElementException exception ) {
81                   status.setText( exception.toString() );
82                }
83             }
84          }
85       );
86       c.add( lastBtn );
87
88       JButton emptyBtn = new JButton( "Is Empty?" );
89       emptyBtn.addActionListener(
90          new ActionListener() {
91             public void actionPerformed( ActionEvent e )
92             {
93                status.setText( v.isEmpty() ?
94                   "Vector is empty" : "Vector is not empty" );
95             }
96          }
97       );
98       c.add( emptyBtn );
99
100      JButton containsBtn = new JButton( "Contains" );
101      containsBtn.addActionListener(
102         new ActionListener() {
103            public void actionPerformed( ActionEvent e )
104            {
105               String searchKey = input.getText();
106
107               if ( v.contains( searchKey ) )
108                  status.setText( "Vector contains " +
109                                  searchKey );
110               else
111                  status.setText( "Vector does not contain " +
112                                  searchKey );
113            }
114         }
115      );
```

Fig. 23.1 Demonstrating class **`Vector`** of package **`java.util`** (part 3 of 5).

```
116         c.add( containsBtn );
117
118         JButton locationBtn = new JButton( "Location" );
119         locationBtn.addActionListener(
120            new ActionListener() {
121               public void actionPerformed( ActionEvent e )
122               {
123                  status.setText( "Element is at location " +
124                                  v.indexOf( input.getText() ) );
125               }
126            }
127         );
128         c.add( locationBtn );
129
130         JButton trimBtn = new JButton( "Trim" );
131         trimBtn.addActionListener(
132            new ActionListener() {
133               public void actionPerformed( ActionEvent e )
134               {
135                  v.trimToSize();
136                  status.setText( "Vector trimmed to size" );
137               }
138            }
139         );
140         c.add( trimBtn );
141
142         JButton statsBtn = new JButton( "Statistics" );
143         statsBtn.addActionListener(
144            new ActionListener() {
145               public void actionPerformed( ActionEvent e )
146               {
147                  status.setText( "Size = " + v.size() +
148                     "; capacity = " + v.capacity() );
149               }
150            }
151         );
152         c.add( statsBtn );
153
154         JButton displayBtn = new JButton( "Display" );
155         displayBtn.addActionListener(
156            new ActionListener() {
157               public void actionPerformed( ActionEvent e )
158               {
159                  Enumeration enum = v.elements();
160                  StringBuffer buf = new StringBuffer();
161
162                  while ( enum.hasMoreElements() )
163                     buf.append(
164                        enum.nextElement() ).append( "  " );
165
```

Fig. 23.1 Demonstrating class **`Vector`** of package **`java.util`** (part 4 of 5).

```
166                JOptionPane.showMessageDialog( null,
167                   buf.toString(), "Display",
168                   JOptionPane.PLAIN_MESSAGE );
169             }
170          }
171       );
172       c.add( displayBtn );
173       c.add( status );
174
175       setSize( 300, 200 );
176       show();
177    }
178
179    public static void main( String args[] )
180    {
181       VectorTest app = new VectorTest();
182
183       app.addWindowListener(
184          new WindowAdapter() {
185             public void windowClosing( WindowEvent e )
186             {
187                System.exit( 0 );
188             }
189          }
190       );
191    }
192 }
```

Fig. 23.1 Demonstrating class **`Vector`** of package **`java.util`** (part 5 of 5).

```
1  // Fig. 23.2: StackTest.java
2  // Testing the Stack class of the java.util package
3  import java.util.*;
4  import java.awt.*;
5  import java.awt.event.*;
6  import javax.swing.*;
7
8  public class StackTest extends JFrame {
9
10    public StackTest()
11    {
12       super( "Stacks" );
13
14       Container c = getContentPane();
15
16       final JLabel status = new JLabel();
17       final Stack s = new Stack();
18
19       c.setLayout( new FlowLayout() );
20       c.add( new JLabel( "Enter a string" ) );
21       final JTextField input = new JTextField( 10 );
22       c.add( input );
23
24       JButton pushBtn = new JButton( "Push" );
25       pushBtn.addActionListener(
26          new ActionListener() {
27             public void actionPerformed( ActionEvent e )
28             {
29                status.setText( "Pushed: " +
30                   s.push( input.getText() ) );
31             }
32          }
33       );
34       c.add( pushBtn );
35
36       JButton popBtn = new JButton( "Pop" );
37       popBtn.addActionListener(
38          new ActionListener() {
39             public void actionPerformed( ActionEvent e )
40             {
41                try {
42                   status.setText( "Popped: " + s.pop() );
43                }
```

Fig. 23.2 Demonstrating class **`Stack`** of package **`java.util`** (part 1 of 3).

```
44                catch ( EmptyStackException exception ) {
45                   status.setText( exception.toString() );
46                }
47             }
48          }
49       );
50       c.add( popBtn );
51
52       JButton peekBtn = new JButton( "Peek" );
53       peekBtn.addActionListener(
54          new ActionListener() {
55             public void actionPerformed( ActionEvent e )
56             {
57                try {
58                   status.setText( "Top: " + s.peek() );
59                }
60                catch ( EmptyStackException exception ) {
61                   status.setText( exception.toString() );
62                }
63             }
64          }
65       );
66       c.add( peekBtn );
67
68       JButton emptyBtn = new JButton( "Is Empty?" );
69       emptyBtn.addActionListener(
70          new ActionListener() {
71             public void actionPerformed( ActionEvent e )
72             {
73                status.setText( s.empty() ?
74                   "Stack is empty" : "Stack is not empty" );
75             }
76          }
77       );
78       c.add( emptyBtn );
79
80       JButton searchBtn = new JButton( "Search" );
81       searchBtn.addActionListener(
82          new ActionListener() {
83             public void actionPerformed( ActionEvent e )
84             {
85                String searchKey = input.getText();
86                int result = s.search( searchKey );
87
88                if ( result == -1 )
89                   status.setText( searchKey + " not found" );
90                else
91                   status.setText( searchKey +
92                      " found at element " + result );
93             }
94          }
95       );
96       c.add( searchBtn );
```

Fig. 23.2 Demonstrating class **`Stack`** of package **`java.util`** (part 2 of 3).

```
97
98          JButton displayBtn = new JButton( "Display" );
99          displayBtn.addActionListener(
100            new ActionListener() {
101               public void actionPerformed( ActionEvent e )
102               {
103                  Enumeration enum = s.elements();
104                  StringBuffer buf = new StringBuffer();
105
106                  while ( enum.hasMoreElements() )
107                     buf.append(
108                        enum.nextElement() ).append( " " );
109
110                  JOptionPane.showMessageDialog( null,
111                      buf.toString(), "Display",
112                      JOptionPane.PLAIN_MESSAGE );
113               }
114            }
115         );
116         c.add( displayBtn );
117         c.add( status );
118
119         setSize( 675, 100 );
120         show();
121      }
122
123      public static void main( String args[] )
124      {
125         StackTest app = new StackTest();
126
127         app.addWindowListener(
128            new WindowAdapter() {
129               public void windowClosing( WindowEvent e )
130               {
131                  System.exit( 0 );
132               }
133            }
134         );
135      }
136   }
```

Fig. 23.2 Demonstrating class **`Stack`** of package **`java.util`** (part 3 of 3).

```
1  // Fig. 23.3: HashtableTest.java
2  // Demonstrates class Hashtable of the java.util package.
3  import java.util.*;
4  import java.awt.*;
5  import java.awt.event.*;
6  import javax.swing.*;
7
8  public class HashtableTest extends JFrame {
9
10    public HashtableTest()
11    {
12       super( "Hashtable Example" );
13
14       final JLabel status = new JLabel();
15       final Hashtable table = new Hashtable();
16       final JTextArea display = new JTextArea( 4, 20 );
17       display.setEditable( false );
18
19       JPanel northPanel = new JPanel();
20       northPanel.setLayout( new BorderLayout() );
21       JPanel northSubPanel = new JPanel();
```

Fig. 23.3 Demonstrating class **`Hashtable`** (part 1 of 5).

```
22          northSubPanel.add( new JLabel( "First name" ) );
23          final JTextField fName = new JTextField( 8 );
24          northSubPanel.add( fName );
25
26          northSubPanel.add( new JLabel( "Last name (key)" ) );
27          final JTextField lName = new JTextField( 8 );
28          northSubPanel.add( lName );
29          northPanel.add( northSubPanel, BorderLayout.NORTH );
30          northPanel.add( status, BorderLayout.SOUTH );
31
32          JPanel southPanel = new JPanel();
33          southPanel.setLayout( new GridLayout( 2, 5 ) );
34          JButton put = new JButton( "Put" );
35          put.addActionListener(
36             new ActionListener() {
37                public void actionPerformed( ActionEvent e )
38                {
39                   Employee emp = new Employee(
40                      fName.getText(), lName.getText() );
41                   Object val = table.put( lName.getText(), emp );
42
43                   if ( val == null )
44                      status.setText( "Put: " + emp.toString() );
45                   else
46                      status.setText( "Put: " + emp.toString() +
47                         "; Replaced: " + val.toString() );
48                }
49             }
50          );
51          southPanel.add( put );
52
53          JButton get = new JButton( "Get" );
54          get.addActionListener(
55             new ActionListener() {
56                public void actionPerformed( ActionEvent e )
57                {
58                   Object val = table.get( lName.getText() );
59
60                   if ( val != null )
61                      status.setText( "Get: " + val.toString() );
62                   else
63                      status.setText( "Get: " + lName.getText() +
64                                      " not in table" );
65                }
66             }
67          );
68          southPanel.add( get );
69
70          JButton remove = new JButton( "Remove" );
```

Fig. 23.3 Demonstrating class **Hashtable** (part 2 of 5).

```
71          remove.addActionListener(
72             new ActionListener() {
73                public void actionPerformed( ActionEvent e )
74                {
75                   Object val = table.remove( lName.getText() );
76
77                   if ( val != null )
78                      status.setText( "Remove: " +
79                                      val.toString() );
80                   else
81                      status.setText( "Remove: " +
82                         lName.getText() + " not in table" );
83                }
84             }
85          );
86          southPanel.add( remove );
87
88          JButton empty = new JButton( "Empty" );
89          empty.addActionListener(
90             new ActionListener() {
91                public void actionPerformed( ActionEvent e )
92                {
93                   status.setText( "Empty: " + table.isEmpty() );
94                }
95             }
96          );
97          southPanel.add( empty );
98
99          JButton containsKey = new JButton( "Contains key" );
100         containsKey.addActionListener(
101            new ActionListener() {
102               public void actionPerformed( ActionEvent e )
103               {
104                  status.setText( "Contains key: " +
105                     table.containsKey( lName.getText() ) );
106               }
107            }
108         );
109         southPanel.add( containsKey );
110
111         JButton clear = new JButton( "Clear table" );
112         clear.addActionListener(
113            new ActionListener() {
114               public void actionPerformed( ActionEvent e )
115               {
116                  table.clear();
117                  status.setText( "Clear: Table is now empty" );
118               }
119            }
120         );
121         southPanel.add( clear );
122
```

Fig. 23.3 Demonstrating class **`Hashtable`** (part 3 of 5).

```
123       JButton listElems = new JButton( "List objects" );
124       listElems.addActionListener(
125          new ActionListener() {
126             public void actionPerformed( ActionEvent e )
127             {
128                StringBuffer buf = new StringBuffer();
129
130                for ( Enumeration enum = table.elements();
131                      enum.hasMoreElements(); )
132                   buf.append(
133                      enum.nextElement() ).append( '\n' );
134
135                display.setText( buf.toString() );
136             }
137          }
138       );
139       southPanel.add( listElems );
140
141       JButton listKeys = new JButton( "List keys" );
142       listKeys.addActionListener(
143          new ActionListener() {
144             public void actionPerformed( ActionEvent e )
145             {
146                StringBuffer buf = new StringBuffer();
147
148                for ( Enumeration enum = table.keys();
149                      enum.hasMoreElements(); )
150                   buf.append(
151                      enum.nextElement() ).append( '\n' );
152
153                JOptionPane.showMessageDialog( null,
154                   buf.toString(), "Display",
155                   JOptionPane.PLAIN_MESSAGE );
156             }
157          }
158       );
159       southPanel.add( listKeys );
160       Container c = getContentPane();
161       c.add( northPanel, BorderLayout.NORTH );
162       c.add( new JScrollPane( display ), BorderLayout.CENTER );
163       c.add( southPanel, BorderLayout.SOUTH );
164
165       setSize( 540, 300 );
166       show();
167    }
168
169    public static void main( String args[] )
170    {
171       HashtableTest app = new HashtableTest();
172
```

Fig. 23.3 Demonstrating class **`Hashtable`** (part 4 of 5).

```
173         app.addWindowListener(
174            new WindowAdapter() {
175               public void windowClosing( WindowEvent e )
176               {
177                  System.exit( 0 );
178               }
179            }
180         );
181      }
182   }
183
184   class Employee {
185      private String first, last;
186
187      public Employee( String fName, String lName )
188      {
189         first = fName;
190         last = lName;
191      }
192
193      public String toString() { return first + " " + last; }
194   }
```

Fig. 23.3 Demonstrating class **`Hashtable`** (part 5 of 5).

```
1  // Fig. 23.4: PropertiesTest.java
2  // Demonstrates class Properties of the java.util package.
3  import java.io.*;
4  import java.util.*;
5  import java.awt.*;
6  import java.awt.event.*;
7  import javax.swing.*;
8
9  public class PropertiesTest extends JFrame {
10    private JLabel status;
11    private Properties table;
12    private JTextArea display;
13
14    public PropertiesTest()
15    {
16       super( "Properties Test" );
17
18       table = new Properties();
19       Container c = getContentPane();
20       JPanel northPanel = new JPanel();
21       northPanel.setLayout( new BorderLayout() );
22       JPanel northSubPanel = new JPanel();
23       JPanel southPanel = new JPanel();
24
25       northSubPanel.add( new JLabel( "Property value" ) );
26       final JTextField propVal = new JTextField( 10 );
27       northSubPanel.add( propVal );
28       northPanel.add( northSubPanel, BorderLayout.NORTH );
29
30       northSubPanel.add( new JLabel( "Property name (key)" ) );
31       final JTextField propName = new JTextField( 10 );
32       northSubPanel.add( propName );
33
34       display = new JTextArea( 4, 35 );
35
36       JButton put = new JButton( "Put" );
37       put.addActionListener(
38          new ActionListener() {
39             public void actionPerformed( ActionEvent e )
40             {
41                Object val = table.put( propName.getText(),
42                                        propVal.getText() );
43
44                if ( val == null )
45                   showStatus( "Put: " + propName.getText() +
46                               " " + propVal.getText() );
47                else
48                   showStatus( "Put: " + propName.getText() +
49                      " " + propVal.getText() +
50                      "; Replaced: " + val.toString() );
51
```

Fig. 23.4 Demonstrating class **`Properties`** (part 1 of 4).

```
52                listProperties();
53             }
54          }
55       );
56       southPanel.setLayout( new GridLayout( 1, 5 ) );
57       southPanel.add( put );
58
59       JButton clear = new JButton( "Clear" );
60       clear.addActionListener(
61          new ActionListener() {
62             public void actionPerformed( ActionEvent e )
63             {
64                table.clear();
65                showStatus( "Table in memory cleared" );
66                listProperties();
67             }
68          }
69       );
70       southPanel.add( clear );
71
72       JButton getProperty = new JButton( "Get property" );
73       getProperty.addActionListener(
74          new ActionListener() {
75             public void actionPerformed( ActionEvent e )
76             {
77                Object val = table.getProperty(
78                   propName.getText() );
79
80                if ( val != null )
81                   showStatus( "Get property: " +
82                      propName.getText() + " " +
83                      val.toString() );
84                else
85                   showStatus( "Get: " + propName.getText() +
86                      " not in table" );
87
88                listProperties();
89             }
90          }
91       );
92       southPanel.add( getProperty );
93
94       JButton save = new JButton( "Save" );
95       save.addActionListener(
96          new ActionListener() {
97             public void actionPerformed( ActionEvent e )
98             {
99                try {
100                  FileOutputStream output;
101
102                 output = new FileOutputStream( "props.dat" );
103                 table.store( output, "Sample Properties" );
104                 output.close();
```

Fig. 23.4 Demonstrating class **`Properties`** (part 2 of 4).

```
105                listProperties();
106             }
107             catch( IOException ex ) {
108                showStatus( ex.toString() );
109             }
110          }
111       }
112    );
113    southPanel.add( save );
114
115    JButton load = new JButton( "Load" );
116    load.addActionListener(
117       new ActionListener() {
118          public void actionPerformed( ActionEvent e )
119          {
120             try {
121                FileInputStream input;
122
123                input = new FileInputStream( "props.dat" );
124                table.load( input );
125                input.close();
126                listProperties();
127             }
128             catch( IOException ex ) {
129                showStatus( ex.toString() );
130             }
131          }
132       }
133    );
134    southPanel.add( load );
135
136    status = new JLabel();
137    northPanel.add( status, BorderLayout.SOUTH );
138
139    c.add( northPanel, BorderLayout.NORTH );
140    c.add( new JScrollPane( display ), BorderLayout.CENTER );
141    c.add( southPanel, BorderLayout.SOUTH );
142
143    setSize( 550, 225 );
144    show();
145 }
146
147 public void listProperties()
148 {
149    StringBuffer buf = new StringBuffer();
150    String pName, pVal;
151
152    Enumeration enum = table.propertyNames();
153
154    while( enum.hasMoreElements() ) {
155       pName = enum.nextElement().toString();
156       pVal = table.getProperty( pName );
157       buf.append( pName ).append( '\t' );
```

Fig. 23.4 Demonstrating class **Properties** (part 3 of 4).

```
158            buf.append( pVal ).append( '\n' );
159         }
160
161         display.setText( buf.toString() );
162      }
163
164      public void showStatus( String s )
165      {
166         status.setText( s );
167      }
168
169      public static void main( String args[] )
170      {
171         PropertiesTest app = new PropertiesTest();
172         app.addWindowListener(
173            new WindowAdapter() {
174               public void windowClosing( WindowEvent e )
175               {
176                  System.exit( 0 );
177               }
178            }
179         );
180      }
181   }
```

Fig. 23.4 Demonstrating class **`Properties`** (part 4 of 4).

Operator	Name	Description
&	bitwise AND	The bits in the result are set to **1** if the corresponding bits in the two operands are both **1**.
|	bitwise inclusive OR	The bits in the result are set to **1** if at least one of the corresponding bits in the two operands is **1**.
^	bitwise exclusive OR	The bits in the result are set to **1** if exactly one of the corresponding bits in the two operands is **1**.
<<	left shift	Shifts the bits of the first operand left by the number of bits specified by the second operand; fill from the right with **0** bits.
>>	right shift with sign extension	Shifts the bits of the first operand right by the number of bits specified by the second operand. If the first operand is negative, **1**s are shifted in from the left; otherwise, **0**s are shifted in from the left.
>>>	right shift with zero extension	Shifts the bits of the first operand right by the number of bits specified by the second operand; **0**s are shifted in from the left.
~	one's complement	All **0** bits are set to **1** and all **1** bits are set to **0**.

Fig. 23.5 The bitwise operators .

```
1  // Fig. 23.6: PrintBits.java
2  // Printing an unsigned integer in bits
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class PrintBits extends JFrame {
8
9     public PrintBits()
10    {
11       super( "Printing bit representations for numbers" );
12
13       Container c = getContentPane();
14       c.setLayout( new FlowLayout() );
15       c.add( new JLabel( "Enter an integer " ) );
16       final JTextField output = new JTextField( 33 );
17       JTextField input = new JTextField( 10 );
18       input.addActionListener(
19          new ActionListener() {
20             public void actionPerformed( ActionEvent e )
21             {
22                int val = Integer.parseInt(
23                   e.getActionCommand() );
24                output.setText( getBits( val ) );
25             }
26          }
27       );
28       c.add( input );
29
30       c.add( new JLabel( "The integer in bits is" ) );
31       output.setEditable( false );
32       c.add( output );
33
34       setSize( 720, 70 );
```

Fig. 23.6 Displaying the bit representation of an integer (part 1 of 2).

```
35          show();
36       }
37
38       private String getBits( int value )
39       {
40          int displayMask = 1 << 31;
41          StringBuffer buf = new StringBuffer( 35 );
42
43          for ( int c = 1; c <= 32; c++ ) {
44             buf.append(
45                ( value & displayMask ) == 0 ? '0' : '1' );
46             value <<= 1;
47
48             if ( c % 8 == 0 )
49                buf.append( ' ' );
50          }
51
52          return buf.toString();
53       }
54
55       public static void main( String args[] )
56       {
57          PrintBits app = new PrintBits();
58          app.addWindowListener(
59             new WindowAdapter() {
60                public void windowClosing( WindowEvent e )
61                {
62                   System.exit( 0 );
63                }
64             }
65          );
66       }
67    }
```

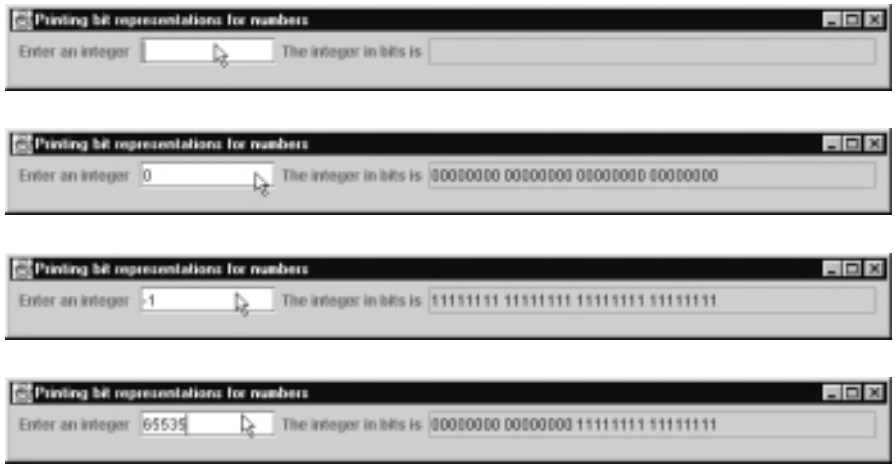

Fig. 23.6 Displaying the bit representation of an integer (part 2 of 2).

Bit 1	Bit 2	Bit 1 & Bit 2
0	0	0
1	0	0
0	1	0
1	1	1

Fig. 23.7 Results of combining two bits with the bitwise AND operator (&).

```
1  // Fig. 23.8: MiscBitOps.java
2  // Using the bitwise AND, bitwise inclusive OR, bitwise
3  // exclusive OR, and bitwise complement operators.
4  import java.awt.*;
5  import java.awt.event.*;
6  import javax.swing.*;
7
8  public class MiscBitOps extends JFrame {
9     private JTextField input1, input2, bits1, bits2;
10    private int val1, val2;
11
12    public MiscBitOps()
13    {
14       super( "Bitwise operators" );
15
16       JPanel inputPanel = new JPanel();
17       inputPanel.setLayout( new GridLayout( 4, 2 ) );
18
19       inputPanel.add( new JLabel( "Enter 2 ints" ) );
20       inputPanel.add( new JLabel( "" ) );
21
22       inputPanel.add( new JLabel( "Value 1" ) );
23       input1 = new JTextField( 8 );
24       inputPanel.add( input1 );
25
26       inputPanel.add( new JLabel( "Value 2" ) );
27       input2 = new JTextField( 8 );
28       inputPanel.add( input2 );
29
30       inputPanel.add( new JLabel( "Result" ) );
31       final JTextField result = new JTextField( 8 );
32       result.setEditable( false );
33       inputPanel.add( result );
34
35       JPanel bitsPanel = new JPanel();
36       bitsPanel.setLayout( new GridLayout( 4, 1 ) );
37       bitsPanel.add( new JLabel( "Bit representations" ) );
38
39       bits1 = new JTextField( 33 );
40       bits1.setEditable( false );
41       bitsPanel.add( bits1 );
42
43       bits2 = new JTextField( 33 );
44       bits2.setEditable( false );
45       bitsPanel.add( bits2 );
46
47       final JTextField bits3 = new JTextField( 33 );
48       bits3.setEditable( false );
49       bitsPanel.add( bits3 );
50
51       JPanel buttonPanel = new JPanel();
52       JButton and = new JButton( "AND" );
```

Fig. 23.8 Demonstrating the bitwise AND, bitwise inclusive OR, bitwise exclusive OR and bitwise complement operators (part 1 of 4).

```
53          and.addActionListener(
54             new ActionListener() {
55                public void actionPerformed( ActionEvent e )
56                {
57                   setFields();
58                   result.setText( Integer.toString( val1 &
59                                                     val2 ) );
60                   bits3.setText( getBits( val1 & val2 ) );
61                }
62             }
63          );
64          buttonPanel.add( and );
65
66          JButton inclusiveOr = new JButton( "Inclusive OR" );
67          inclusiveOr.addActionListener(
68             new ActionListener() {
69                public void actionPerformed( ActionEvent e )
70                {
71                   setFields();
72                   result.setText( Integer.toString( val1 |
73                                                     val2 ) );
74                   bits3.setText( getBits( val1 | val2 ) );
75                }
76             }
77          );
78          buttonPanel.add( inclusiveOr );
79
80          JButton exclusiveOr = new JButton( "Exclusive OR" );
81          exclusiveOr.addActionListener(
82             new ActionListener() {
83                public void actionPerformed( ActionEvent e )
84                {
85                   setFields();
86                   result.setText( Integer.toString( val1 ^
87                                                     val2 ) );
88                   bits3.setText( getBits( val1 ^ val2 ) );
89                }
90             }
91          );
92          buttonPanel.add( exclusiveOr );
93
94          JButton complement = new JButton( "Complement" );
95          complement.addActionListener(
96             new ActionListener() {
97                public void actionPerformed( ActionEvent e )
98                {
99                   input2.setText( "" );
100                  bits2.setText( "" );
101                  int val = Integer.parseInt( input1.getText() );
102                  result.setText( Integer.toString( ~val ) );
103                  bits1.setText( getBits( val ) );
```

Fig. 23.8 Demonstrating the bitwise AND, bitwise inclusive OR, bitwise exclusive OR and bitwise complement operators (part 2 of 4).

```
104                bits3.setText( getBits( ~val ) );
105             }
106          }
107       );
108       buttonPanel.add( complement );
109
110       Container c = getContentPane();
111       c.setLayout( new BorderLayout() );
112       c.add( inputPanel, BorderLayout.WEST );
113       c.add( bitsPanel, BorderLayout.EAST );
114       c.add( buttonPanel, BorderLayout.SOUTH );
115
116       setSize( 600, 150 );
117       show();
118    }
119
120    private void setFields()
121    {
122       val1 = Integer.parseInt( input1.getText() );
123       val2 = Integer.parseInt( input2.getText() );
124
125       bits1.setText( getBits( val1 ) );
126       bits2.setText( getBits( val2 ) );
127    }
128
129    private String getBits( int value )
130    {
131       int displayMask = 1 << 31;
132       StringBuffer buf = new StringBuffer( 35 );
133
134       for ( int c = 1; c <= 32; c++ ) {
135          buf.append(
136             ( value & displayMask ) == 0 ? '0' : '1' );
137          value <<= 1;
138
139          if ( c % 8 == 0 )
140             buf.append( ' ' );
141       }
142
143       return buf.toString();
144    }
```

Fig. 23.8 Demonstrating the bitwise AND, bitwise inclusive OR, bitwise exclusive OR and bitwise complement operators (part 3 of 4).

```
145
146    public static void main( String args[] )
147    {
148       MiscBitOps app = new MiscBitOps();
149       app.addWindowListener(
150          new WindowAdapter() {
151             public void windowClosing( WindowEvent e )
152             {
153                System.exit( 0 );
154             }
155          }
156       );
157    }
158 }
```

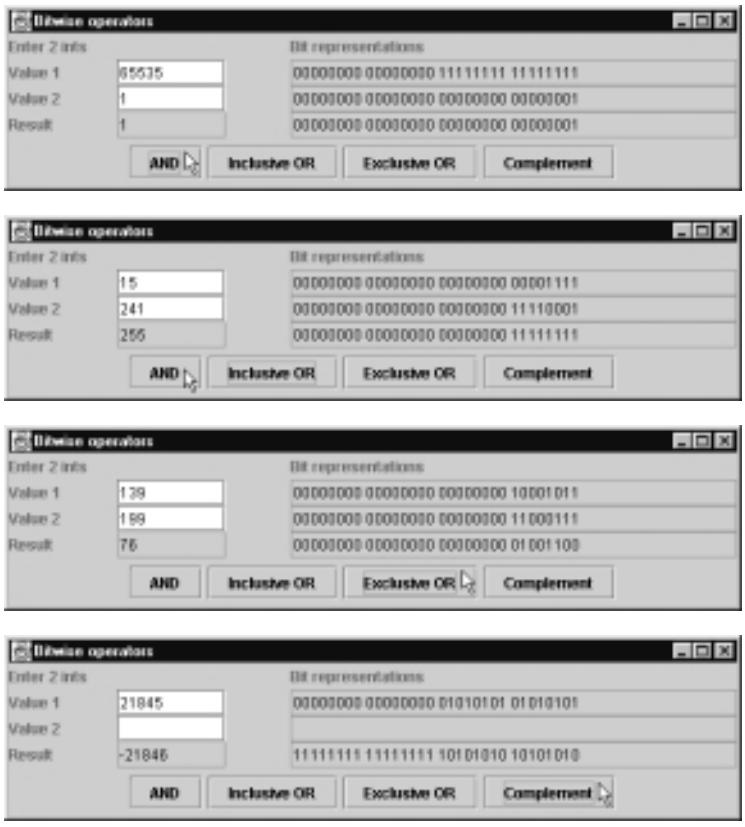

Fig. 23.8 Demonstrating the bitwise AND, bitwise inclusive OR, bitwise exclusive OR and bitwise complement operators (part 4 of 4).

Bit 1	Bit 2	Bit 1 ^ Bit 2
0	0	0
1	0	1
0	1	1
1	1	0

Fig. 23.9 Results of combining two bits with the bitwise exclusive OR operator (^).

Bit 1	Bit 2	Bit 1 | Bit 2
0	0	0
1	0	1
0	1	1
1	1	1

Fig. 23.10 Results of combining two bits with the bitwise inclusive OR operator (|).

```
1  // Fig. 23.11: BitShift.java
2  // Using the bitwise shift operators.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class BitShift extends JFrame {
8
9     public BitShift()
10    {
11       super( "Shifting bits" );
12
13       Container c = getContentPane();
14       c.setLayout( new FlowLayout() );
15       final JTextField bits = new JTextField( 33 );
16       c.add( new JLabel( "Integer to shift " ) );
17
18       final JTextField value = new JTextField( 12 );
```

Fig. 23.11 Demonstrating the bitwise shift operators (part 1 of 4).

```
19          value.addActionListener(
20             new ActionListener() {
21                public void actionPerformed( ActionEvent e )
22                {
23                   int val = Integer.parseInt( value.getText() );
24                   bits.setText( getBits( val ) );
25                }
26             }
27          );
28          c.add( value );
29
30          bits.setEditable( false );
31          c.add( bits );
32
33          JButton left = new JButton( "<<" );
34          left.addActionListener(
35             new ActionListener() {
36                public void actionPerformed( ActionEvent e )
37                {
38                   int val = Integer.parseInt( value.getText() );
39                   val <<= 1;
40                   value.setText( Integer.toString( val ) );
41                   bits.setText( getBits( val ) );
42                }
43             }
44          );
45          c.add( left );
46
47          JButton rightSign = new JButton( ">>" );
48          rightSign.addActionListener(
49             new ActionListener() {
50                public void actionPerformed( ActionEvent e )
51                {
52                   int val = Integer.parseInt( value.getText() );
53                   val >>= 1;
54                   value.setText( Integer.toString( val ) );
55                   bits.setText( getBits( val ) );
56                }
57             }
58          );
59          c.add( rightSign );
60
61          JButton rightZero = new JButton( ">>>" );
62          rightZero.addActionListener(
63             new ActionListener() {
64                public void actionPerformed( ActionEvent e )
65                {
66                   int val = Integer.parseInt( value.getText() );
67                   val >>>= 1;
68                   value.setText( Integer.toString( val ) );
```

Fig. 23.11 Demonstrating the bitwise shift operators (part 2 of 4).

```
69                bits.setText( getBits( val ) );
70             }
71          }
72       );
73       c.add( rightZero );
74
75       setSize( 400, 120 );
76       show();
77    }
78
79    private String getBits( int value )
80    {
81       int displayMask = 1 << 31;
82       StringBuffer buf = new StringBuffer( 35 );
83
84       for ( int c = 1; c <= 32; c++ ) {
85          buf.append(
86             ( value & displayMask ) == 0 ? '0' : '1' );
87          value <<= 1;
88
89          if ( c % 8 == 0 )
90             buf.append( ' ' );
91       }
92
93       return buf.toString();
94    }
95
96    public static void main( String args[] )
97    {
98       BitShift app = new BitShift();
99       app.addWindowListener(
100         new WindowAdapter() {
101            public void windowClosing( WindowEvent e )
102            {
103               System.exit( 0 );
104            }
105         }
106      );
107   }
108}
```

1

2

Fig. 23.11 Demonstrating the bitwise shift operators (part 3 of 4).

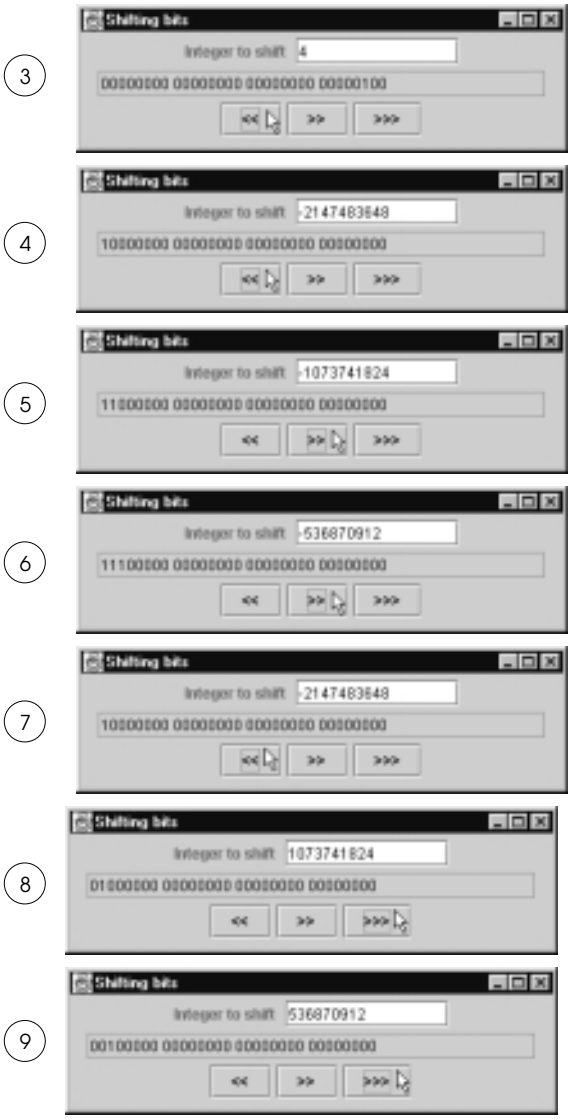

Fig. 23.11 Demonstrating the bitwise shift operators (part 4 of 4).

Bitwise assignment operators	
&=	Bitwise AND assignment operator.
|=	Bitwise inclusive OR assignment operator.
^=	Bitwise exclusive OR assignment operator.
<<=	Left shift assignment operator.
>>=	Right shift with sign extension assignment operator.
>>>=	Right shift with zero extension assignment operator.

Fig. 23.12 The bitwise assignment operators.

```
1  // Fig. 23.13: BitSetTest.java
2  // Using a BitSet to demonstrate the Sieve of Eratosthenes.
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.util.*;
6  import javax.swing.*;
7
```

Fig. 23.13 Demonstrating the Sieve of Eratosthenes using a **`BitSet`** (part 1 of 3).

```
 8
 9  public class BitSetTest extends JFrame {
10
11     public BitSetTest()
12     {
13        super( "BitSets" );
14
15        final BitSet sieve = new BitSet( 1024 );
16        Container c = getContentPane();
17        final JLabel status = new JLabel();
18        c.add( status, BorderLayout.SOUTH );
19        JPanel inputPanel = new JPanel();
20
21        inputPanel.add( new JLabel( "Enter a value from " +
22                                    "1 to 1023" ) );
23        final JTextField input = new JTextField( 10 );
24        input.addActionListener(
25           new ActionListener() {
26              public void actionPerformed( ActionEvent e )
27              {
28                 int val = Integer.parseInt( input.getText() );
29
30                 if ( sieve.get( val ) )
31                    status.setText( val + " is a prime number" );
32                 else
33                    status.setText( val +
34                                    " is not a prime number" );
35              }
36           }
37        );
38        inputPanel.add( input );
39        c.add( inputPanel, BorderLayout.NORTH );
40
41        JTextArea primes = new JTextArea();
42        ScrollPane p = new ScrollPane();
43        p.add( primes );
44
45        c.add( p, BorderLayout.CENTER );
46
47        // set all bits from 1 to 1023
48        int size = sieve.size();
49
50        for ( int i = 1; i < size; i++ )
51           sieve.set( i );
52
53        // perform Sieve of Eratosthenes
54        int finalBit = ( int ) Math.sqrt( sieve.size() );
55
56        for ( int i = 2; i < finalBit; i++ )
57           if ( sieve.get( i ) )
58              for ( int j = 2 * i; j < size; j += i )
59                 sieve.clear( j );
60
```

Fig. 23.13 Demonstrating the Sieve of Eratosthenes using a **`BitSet`** (part 2 of 3).

```
61          int counter = 0;
62
63          for ( int i = 1; i < size; i++ )
64             if ( sieve.get( i ) ) {
65                primes.append( String.valueOf( i ) );
66                primes.append( ++counter % 7 == 0 ? "\n" : "\t" );
67             }
68
69          setSize( 300, 250 );
70          show();
71       }
72
73       public static void main( String args[] )
74       {
75          BitSetTest app = new BitSetTest();
76          app.addWindowListener(
77             new WindowAdapter() {
78                public void windowClosing( WindowEvent e )
79                {
80                   System.exit( 0 );
81                }
82             }
83          );
84       }
85    }
```

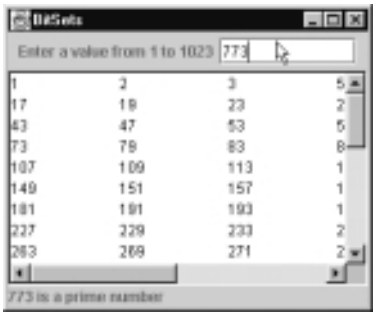

Fig. 23.13 Demonstrating the Sieve of Eratosthenes using a **`BitSet`** (part 3 of 3).