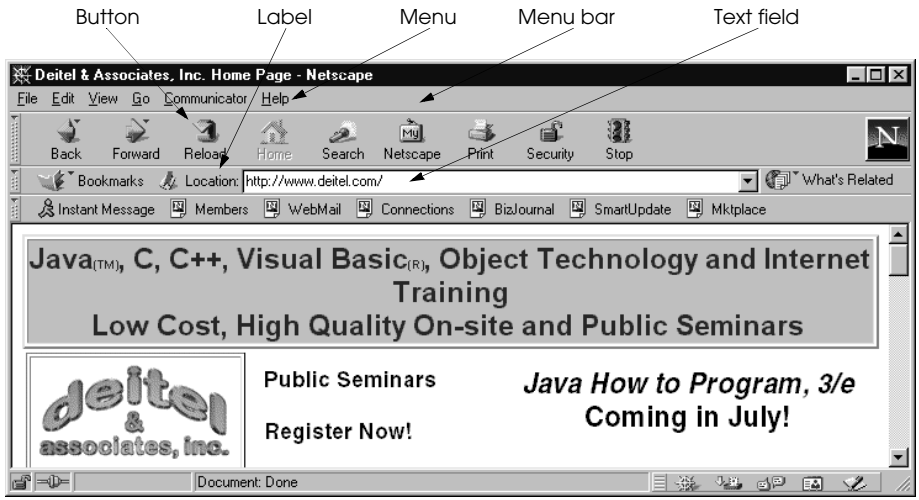

Fig. 12.1 A sample Netscape Communicator window with GUI components.

Component	Description
`JLabel`	An area where uneditable text or icons can be displayed.
`JTextField`	An area in which the user inputs data from the keyboard. The area can also display information.
`JButton`	An area that triggers an event when clicked.
`JCheckBox`	A GUI component that is either selected or not selected.
`JComboBox`	A drop-down list of items from which the user can make a selection by clicking an item in the list or by typing into the box, if permitted.
`JList`	An area where a list of items is displayed from which the user can make a selection by clicking once on any element in the list. Double-clicking an element in the list generates an action event. Multiple elements can be selected.
`JPanel`	A container in which components can be placed.

Fig. 12.2 Some basic GUI components.

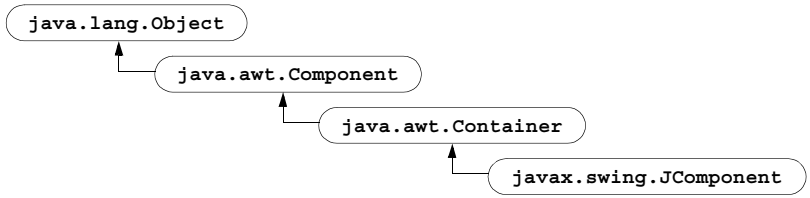

Fig. 12.3 Common superclasses of many of the Swing components.

```
1  // Fig. 12.4: LabelTest.java
2  // Demonstrating the JLabel class.
3  import javax.swing.*;
4  import java.awt.*;
5  import java.awt.event.*;
6
7  public class LabelTest extends JFrame {
8     private JLabel label1, label2, label3;
9
10    public LabelTest()
11    {
12       super( "Testing JLabel" );
13
14       Container c = getContentPane();
15       c.setLayout( new FlowLayout() );
16
17       // JLabel constructor with a string argument
18       label1 = new JLabel( "Label with text" );
19       label1.setToolTipText( "This is label1" );
20       c.add( label1 );
21
22       // JLabel constructor with string, Icon and
23       // alignment arguments
24       Icon bug = new ImageIcon( "bug1.gif" );
25       label2 = new JLabel( "Label with text and icon",
26                            bug, SwingConstants.LEFT );
27       label2.setToolTipText( "This is label2" );
28       c.add( label2 );
29
30       // JLabel constructor no arguments
31       label3 = new JLabel();
32       label3.setText( "Label with icon and text at bottom" );
33       label3.setIcon( bug );
34       label3.setHorizontalTextPosition(
35          SwingConstants.CENTER );
36       label3.setVerticalTextPosition(
37          SwingConstants.BOTTOM );
38       label3.setToolTipText( "This is label3" );
39       c.add( label3 );
40
41       setSize( 275, 170 );
42       show();
43    }
44
45    public static void main( String args[] )
46    {
47       LabelTest app = new LabelTest();
48
49       app.addWindowListener(
50          new WindowAdapter() {
51             public void windowClosing( WindowEvent e )
52             {
53                System.exit( 0 );
54             }
55          }
```

```
56          );
57       }
58    }
```

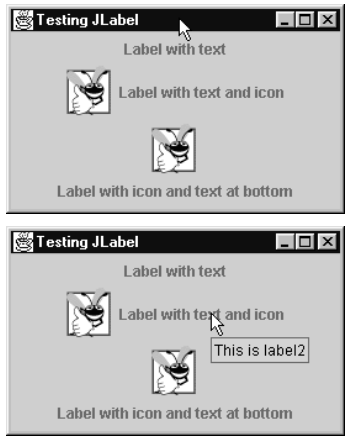

Fig. 12.4 Demonstrating class **`JLabel`**.

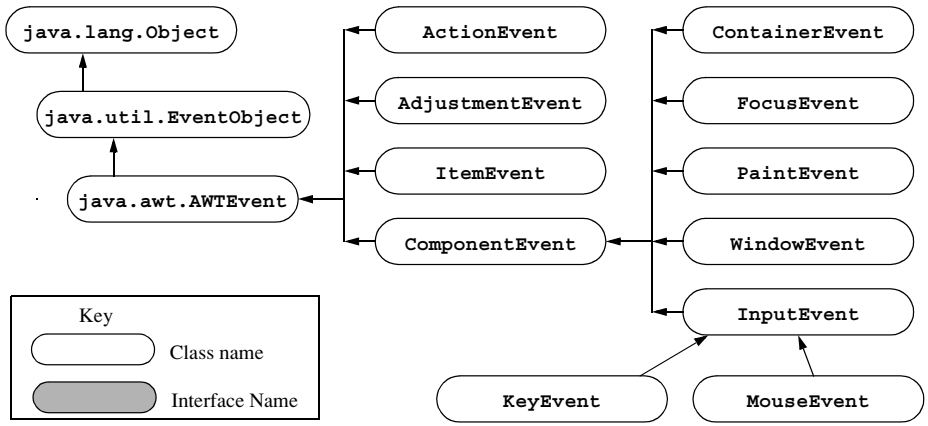

Fig. 12.5 Some event classes of package **`java.awt.event`**.

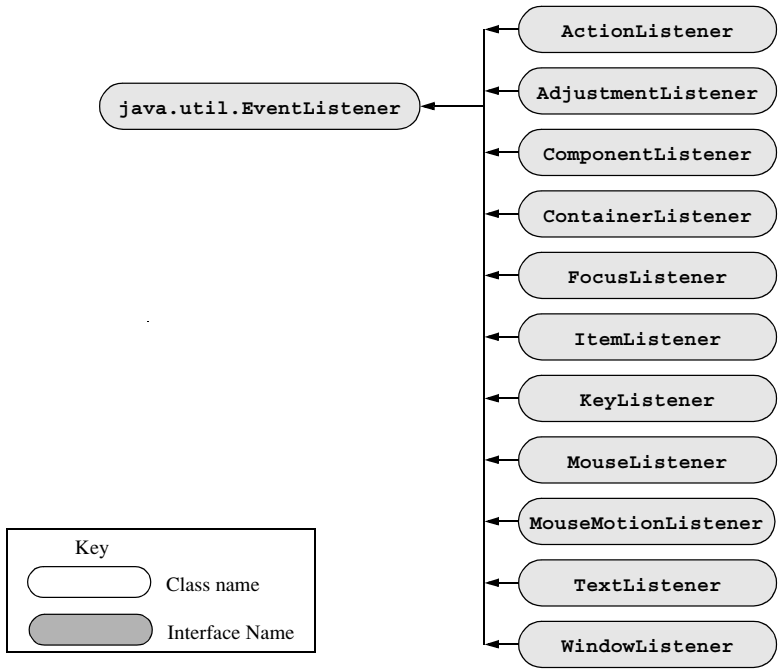

Fig. 12.6 Event-listener interfaces of package `java.awt.event`.

```
1  // Fig. 12.7: TextFieldTest.java
2  // Demonstrating the JTextField class.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class TextFieldTest extends JFrame {
8     private JTextField text1, text2, text3;
9     private JPasswordField password;
10
11    public TextFieldTest()
12    {
13       super( "Testing JTextField and JPasswordField" );
14
15       Container c = getContentPane();
16       c.setLayout( new FlowLayout() );
17
18       // construct textfield with default sizing
19       text1 = new JTextField( 10 );
20       c.add( text1 );
21
22       // construct textfield with default text
23       text2 = new JTextField( "Enter text here" );
24       c.add( text2 );
25
26       // construct textfield with default text and
27       // 20 visible elements and no event handler
28       text3 = new JTextField( "Uneditable text field", 20 );
29       text3.setEditable( false );
30       c.add( text3 );
31
32       // construct textfield with default text
33       password = new JPasswordField( "Hidden text" );
34       c.add( password );
35
36       TextFieldHandler handler = new TextFieldHandler();
37       text1.addActionListener( handler );
38       text2.addActionListener( handler );
39       text3.addActionListener( handler );
40       password.addActionListener( handler );
41
42       setSize( 325, 100 );
43       show();
44    }
45
46    public static void main( String args[] )
47    {
48       TextFieldTest app = new TextFieldTest();
49
50       app.addWindowListener(
51          new WindowAdapter() {
52             public void windowClosing( WindowEvent e )
53             {
54                System.exit( 0 );
55             }
```

```
56            }
57         );
58      }
59
60      // inner class for event handling
61      private class TextFieldHandler implements ActionListener {
62         public void actionPerformed( ActionEvent e )
63         {
64            String s = "";
65
66            if ( e.getSource() == text1 )
67               s = "text1: " + e.getActionCommand();
68            else if ( e.getSource() == text2 )
69               s = "text2: " + e.getActionCommand();
70            else if ( e.getSource() == text3 )
71               s = "text3: " + e.getActionCommand();
72            else if ( e.getSource() == password ) {
73               JPasswordField pwd =
74                  (JPasswordField) e.getSource();
75               s = "password: " +
76                   new String( pwd.getPassword() );
77            }
78
79            JOptionPane.showMessageDialog( null, s );
80         }
81      }
82   }
```

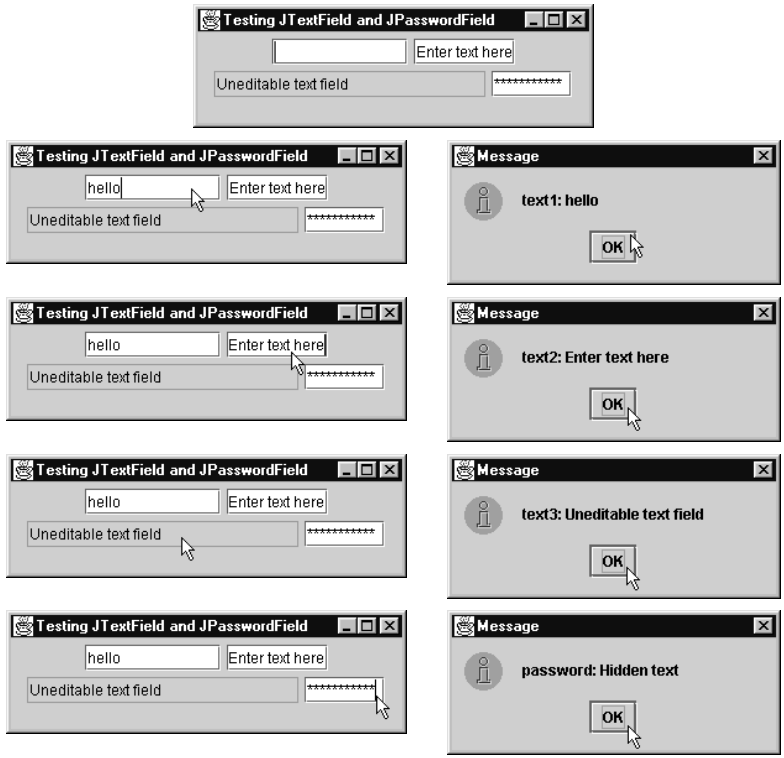

Fig. 12.7 Demonstrating **`JTextField`**s and **`JPasswordField`**s.

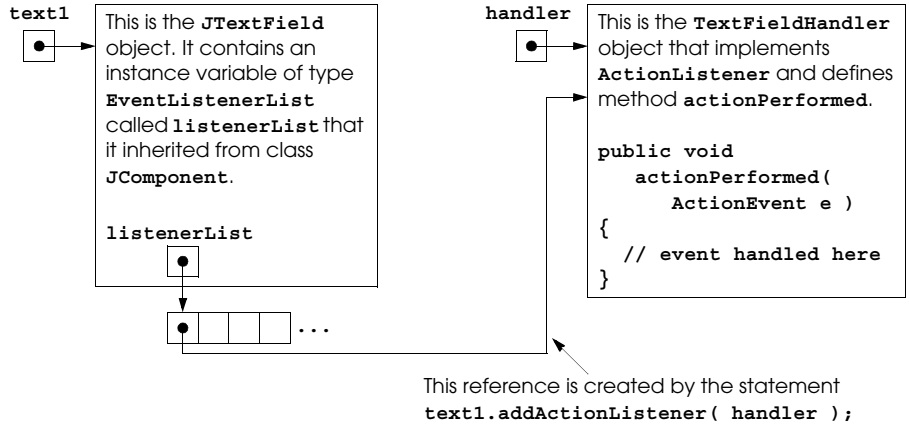

Fig. 12.8 Event registration for **`JTextField text1`**.

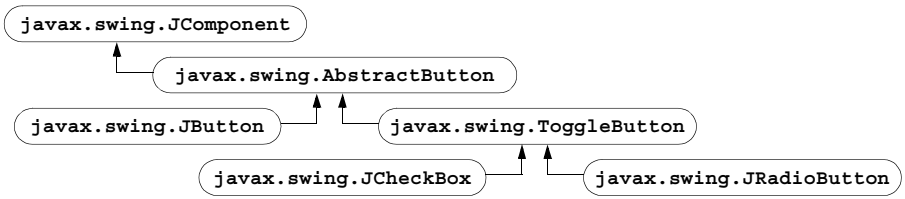

Fig. 12.9 The button hierarchy.

```
1  // Fig. 12.10: ButtonTest.java
2  // Creating JButtons.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class ButtonTest extends JFrame {
8     private JButton plainButton, fancyButton;
9
10    public ButtonTest()
11    {
12       super( "Testing Buttons" );
13
14       Container c = getContentPane();
15       c.setLayout( new FlowLayout() );
16
17       // create buttons
18       plainButton = new JButton( "Plain Button" );
19       c.add( plainButton );
20
21       Icon bug1 = new ImageIcon( "bug1.gif" );
22       Icon bug2 = new ImageIcon( "bug2.gif" );
23       fancyButton = new JButton( "Fancy Button", bug1 );
24       fancyButton.setRolloverIcon( bug2 );
25       c.add( fancyButton );
26
27       // create an instance of inner class ButtonHandler
28       // to use for button event handling
29       ButtonHandler handler = new ButtonHandler();
30       fancyButton.addActionListener( handler );
31       plainButton.addActionListener( handler );
32
33       setSize( 275, 100 );
34       show();
35    }
36
37    public static void main( String args[] )
38    {
39       ButtonTest app = new ButtonTest();
40
41       app.addWindowListener(
42          new WindowAdapter() {
43             public void windowClosing( WindowEvent e )
44             {
45                System.exit( 0 );
46             }
47          }
48       );
49    }
50
51    // inner class for button event handling
52    private class ButtonHandler implements ActionListener {
53       public void actionPerformed( ActionEvent e )
54       {
55          JOptionPane.showMessageDialog( null,
```

```
56                "You pressed: " + e.getActionCommand() );
57             }
58          }
59       }
```

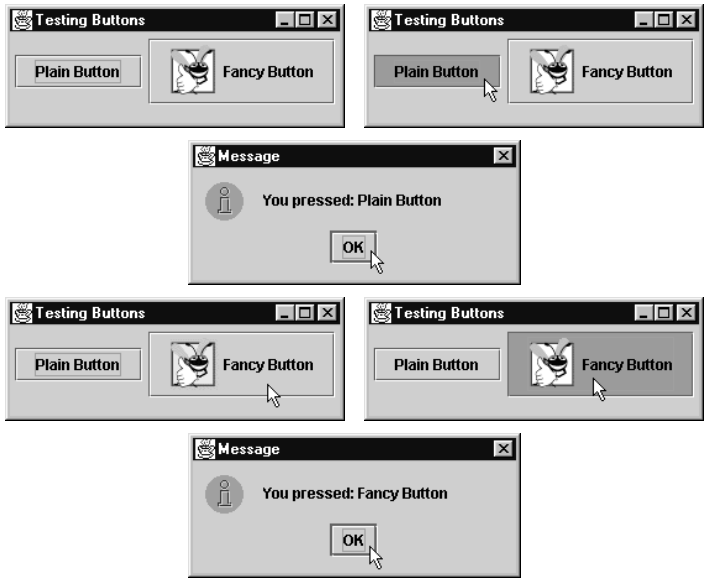

Fig. 12.10 Demonstrating command buttons and action events.

```
1  // Fig. 12.11: CheckBoxTest.java
2  // Creating Checkbox buttons.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class CheckBoxTest extends JFrame {
8     private JTextField t;
9     private JCheckBox bold, italic;
10
11    public CheckBoxTest()
12    {
13       super( "JCheckBox Test" );
14
15       Container c = getContentPane();
16       c.setLayout(new FlowLayout());
17
18       t = new JTextField( "Watch the font style change", 20 );
19       t.setFont( new Font( "TimesRoman", Font.PLAIN, 14 ) );
20       c.add( t );
21
22       // create checkbox objects
23       bold = new JCheckBox( "Bold" );
24       c.add( bold );
25
26       italic = new JCheckBox( "Italic" );
27       c.add( italic );
28
29       CheckBoxHandler handler = new CheckBoxHandler();
30       bold.addItemListener( handler );
31       italic.addItemListener( handler );
32
33       addWindowListener(
34          new WindowAdapter() {
35             public void windowClosing( WindowEvent e )
36             {
37                System.exit( 0 );
38             }
39          }
40       );
41
42       setSize( 275, 100 );
43       show();
44    }
45
46    public static void main( String args[] )
47    {
48       new CheckBoxTest();
49    }
50
51    private class CheckBoxHandler implements ItemListener {
52       private int valBold = Font.PLAIN;
53       private int valItalic = Font.PLAIN;
```

Fig. 12.11 Program that creates two **JCheckBox** buttons (part 1 of 2).

```
54
55          public void itemStateChanged( ItemEvent e )
56          {
57             if ( e.getSource() == bold )
58                if ( e.getStateChange() == ItemEvent.SELECTED )
59                   valBold = Font.BOLD;
60                else
61                   valBold = Font.PLAIN;
62
63             if ( e.getSource() == italic )
64                if ( e.getStateChange() == ItemEvent.SELECTED )
65                   valItalic = Font.ITALIC;
66                else
67                   valItalic = Font.PLAIN;
68
69             t.setFont(
70               new Font( "TimesRoman", valBold + valItalic, 14 ) );
71             t.repaint();
72          }
73       }
74    }
```

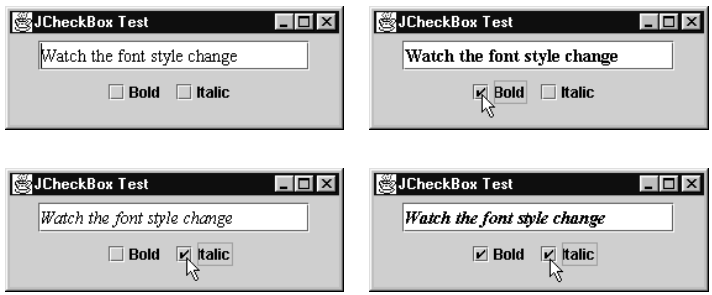

Fig. 12.11 Program that creates two **`JCheckBox`** buttons (part 2 of 2).

```
1  // Fig. 12.12: RadioButtonTest.java
2  // Creating radio buttons using ButtonGroup and JRadioButton.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class RadioButtonTest extends JFrame {
8     private JTextField t;
9     private Font plainFont, boldFont,
10                 italicFont, boldItalicFont;
11    private JRadioButton plain, bold, italic, boldItalic;
12    private ButtonGroup radioGroup;
13
14    public RadioButtonTest()
15    {
16       super( "RadioButton Test" );
17
18       Container c = getContentPane();
19       c.setLayout( new FlowLayout() );
20
21       t = new JTextField( "Watch the font style change", 25 );
22       c.add( t );
23
24       // Create radio buttons
25       plain = new JRadioButton( "Plain", true );
26       c.add( plain );
27       bold = new JRadioButton( "Bold", false);
28       c.add( bold );
29       italic = new JRadioButton( "Italic", false );
30       c.add( italic );
31       boldItalic = new JRadioButton( "Bold/Italic", false );
32       c.add( boldItalic );
33
34       // register events
35       RadioButtonHandler handler = new RadioButtonHandler();
36       plain.addItemListener( handler );
37       bold.addItemListener( handler );
38       italic.addItemListener( handler );
39       boldItalic.addItemListener( handler );
40
41       // create logical relationship between JRadioButtons
42       radioGroup = new ButtonGroup();
43       radioGroup.add( plain );
44       radioGroup.add( bold );
45       radioGroup.add( italic );
46       radioGroup.add( boldItalic );
47
48       plainFont = new Font( "TimesRoman", Font.PLAIN, 14 );
49       boldFont = new Font( "TimesRoman", Font.BOLD, 14 );
50       italicFont = new Font( "TimesRoman", Font.ITALIC, 14 );
51       boldItalicFont =
52          new Font( "TimesRoman", Font.BOLD + Font.ITALIC, 14 );
53       t.setFont( plainFont );
```

Fig. 12.12 Creating and manipulating radio buttons (part 1 of 2).

```
54
55          setSize( 300, 100 );
56          show();
57       }
58
59       public static void main( String args[] )
60       {
61          RadioButtonTest app = new RadioButtonTest();
62
63          app.addWindowListener(
64             new WindowAdapter() {
65                public void windowClosing( WindowEvent e )
66                {
67                   System.exit( 0 );
68                }
69             }
70          );
71       }
72
73       private class RadioButtonHandler implements ItemListener {
74          public void itemStateChanged( ItemEvent e )
75          {
76             if ( e.getSource() == plain )
77                t.setFont( plainFont );
78             else if ( e.getSource() == bold )
79                t.setFont( boldFont );
80             else if ( e.getSource() == italic )
81                t.setFont( italicFont );
82             else if ( e.getSource() == boldItalic )
83                t.setFont( boldItalicFont );
84
85             t.repaint();
86          }
87       }
88    }
```

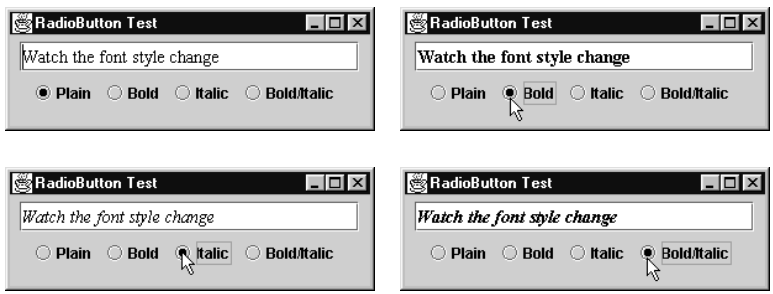

Fig. 12.12 Creating and manipulating radio buttons (part 2 of 2).

```
1  // Fig. 12.13: ComboBoxTest.java
2  // Using a JComboBox to select an image to display.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class ComboBoxTest extends JFrame {
8     private JComboBox images;
9     private JLabel label;
10    private String names[] =
11       { "bug1.gif", "bug2.gif",
12         "travelbug.gif", "buganim.gif" };
13    private Icon icons[] =
14       { new ImageIcon( names[ 0 ] ),
15         new ImageIcon( names[ 1 ] ),
16         new ImageIcon( names[ 2 ] ),
17         new ImageIcon( names[ 3 ] ) };
18
19    public ComboBoxTest()
20    {
21       super( "Testing JComboBox" );
22
23       Container c = getContentPane();
24       c.setLayout( new FlowLayout() );
25
26       images = new JComboBox( names );
27       images.setMaximumRowCount( 3 );
28
29       images.addItemListener(
30          new ItemListener() {
31             public void itemStateChanged( ItemEvent e )
32             {
33                label.setIcon(
34                   icons[ images.getSelectedIndex() ] );
35             }
36          }
37       );
38
39       c.add( images );
40
41       label = new JLabel( icons[ 0 ] );
42       c.add( label );
43
44       setSize( 350, 100 );
45       show();
46    }
47
48    public static void main( String args[] )
49    {
50       ComboBoxTest app = new ComboBoxTest();
51
52       app.addWindowListener(
53          new WindowAdapter() {
54             public void windowClosing( WindowEvent e )
55             {
```

```
56                     System.exit( 0 );
57                  }
58               }
59            );
60         }
61      }
```

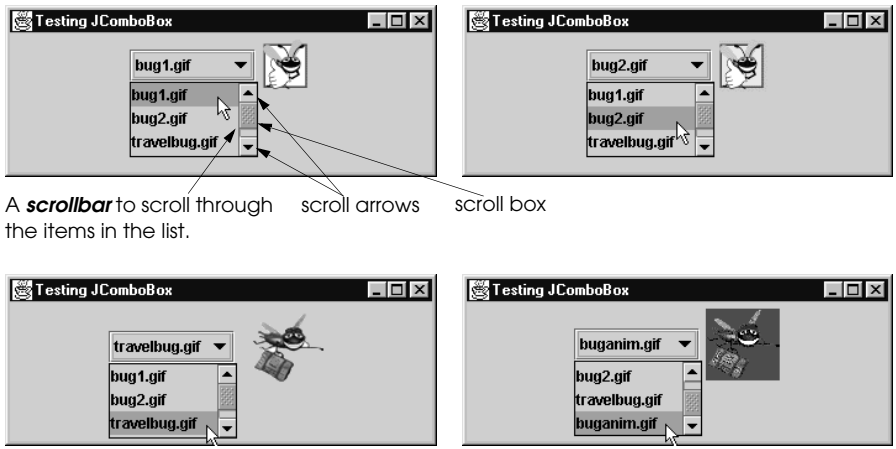

Fig. 12.13 Program that uses a **JComboBox** to select an icon.

```
1  // Fig. 12.14: ListTest.java
2  // Selecting colors from a JList.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6  import javax.swing.event.*;
7
8  public class ListTest extends JFrame {
9     private JList colorList;
10    private Container c;
11
12    private String colorNames[] =
13       { "Black", "Blue", "Cyan", "Dark Gray", "Gray", "Green",
14         "Light Gray", "Magenta", "Orange", "Pink", "Red",
15         "White", "Yellow" };
16
17    private Color colors[] =
18       { Color.black, Color.blue, Color.cyan, Color.darkGray,
19         Color.gray, Color.green, Color.lightGray,
20         Color.magenta, Color.orange, Color.pink, Color.red,
21         Color.white, Color.yellow };
22
23    public ListTest()
24    {
25       super( "List Test" );
26
27       c = getContentPane();
28       c.setLayout( new FlowLayout() );
29
30       // create a list with the items in the colorNames array
31       colorList = new JList( colorNames );
32       colorList.setVisibleRowCount( 5 );
33
34       // do not allow multiple selections
35       colorList.setSelectionMode(
36          ListSelectionModel.SINGLE_SELECTION );
37
38       // add a JScrollPane containing the JList
39       // to the content pane
40       c.add( new JScrollPane( colorList ) );
41
42       // set up event handler
43       colorList.addListSelectionListener(
44          new ListSelectionListener() {
45             public void valueChanged( ListSelectionEvent e )
46             {
47                c.setBackground(
48                   colors[ colorList.getSelectedIndex() ] );
49             }
50          }
51       );
52
53       setSize( 350, 150 );
54       show();
55    }
```

```
56
57       public static void main( String args[] )
58       {
59          ListTest app = new ListTest();
60
61          app.addWindowListener(
62             new WindowAdapter() {
63                public void windowClosing( WindowEvent e )
64                {
65                   System.exit( 0 );
66                }
67             }
68          );
69       }
70    }
```

Fig. 12.14 Selecting colors from a `JList`.

```
// Fig. 12.15: MultipleSelection.java
// Copying items from one List to another.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MultipleSelection extends JFrame {
   private JList colorList, copyList;
   private JButton copy;
   private String colorNames[] =
      { "Black", "Blue", "Cyan", "Dark Gray", "Gray",
        "Green", "Light Gray", "Magenta", "Orange", "Pink",
        "Red", "White", "Yellow" };

   public MultipleSelection()
   {
      super( "Multiple Selection Lists" );

      Container c = getContentPane();
      c.setLayout( new FlowLayout() );

      colorList = new JList( colorNames );
      colorList.setVisibleRowCount( 5 );
      colorList.setFixedCellHeight( 15 );
      colorList.setSelectionMode(
         ListSelectionModel.MULTIPLE_INTERVAL_SELECTION );
      c.add( new JScrollPane( colorList ) );

      // create copy button
      copy = new JButton( "Copy >>>" );
      copy.addActionListener(
         new ActionListener() {
            public void actionPerformed( ActionEvent e )
            {
               // place selected values in copyList
               copyList.setListData(
                  colorList.getSelectedValues() );
            }
         }
      );
      c.add( copy );

      copyList = new JList( );
      copyList.setVisibleRowCount( 5 );
      copyList.setFixedCellWidth( 100 );
      copyList.setFixedCellHeight( 15 );
      copyList.setSelectionMode(
         ListSelectionModel.SINGLE_INTERVAL_SELECTION );
      c.add( new JScrollPane( copyList ) );

      setSize( 300, 120 );
      show();
   }

```

```
55      public static void main( String args[] )
56      {
57         MultipleSelection app = new MultipleSelection();
58
59         app.addWindowListener(
60            new WindowAdapter() {
61               public void windowClosing( WindowEvent e )
62               {
63                  System.exit( 0 );
64               }
65            }
66         );
67      }
68   }
```

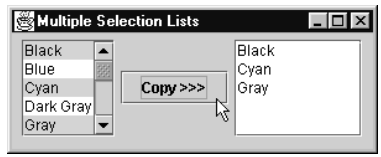

Fig. 12.15 Using a multiple-selection `JList`.

MouseListener and MouseMotionListener interface methods

`public void mousePressed( MouseEvent e )  // MouseListener`

Called when a mouse button is pressed with the mouse cursor on a component.

`public void mouseClicked( MouseEvent e )  // MouseListener`

Called when a mouse button is pressed and released on a component without moving the mouse cursor.

`public void mouseReleased( MouseEvent e ) // MouseListener`

Called when a mouse button is released after being pressed. This event is always preceded by a **`mousePressed`** event.

`public void mouseEntered( MouseEvent e )  // MouseListener`

Called when the mouse cursor enters the bounds of a component.

`public void mouseExited( MouseEvent e )   // MouseListener`

Called when the mouse cursor leaves the bounds of a component.

`public void mouseDragged( MouseEvent e )  // MouseMotionListener`

Called when the mouse button is pressed and the mouse is moved. This event is always preceded by a call to **`mousePressed`**.

`public void mouseMoved( MouseEvent e )    // MouseMotionListener`

Called when the mouse is moved with the mouse cursor on a component.

Fig. 12.16 **`MouseListener`** and **`MouseMotionListener`** interface methods.

```
// Fig. 12.17: MouseTracker.java
// Demonstrating mouse events.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MouseTracker extends JFrame
             implements MouseListener, MouseMotionListener {
   private JLabel statusBar;

   public MouseTracker()
   {
      super( "Demonstrating Mouse Events" );

      statusBar = new JLabel();
      getContentPane().add( statusBar, BorderLayout.SOUTH );

      // application listens to its own mouse events
      addMouseListener( this );
      addMouseMotionListener( this );

      setSize( 275, 100 );
      show();
   }

   // MouseListener event handlers
   public void mouseClicked( MouseEvent e )
   {
      statusBar.setText( "Clicked at [" + e.getX() +
                         ", " + e.getY() + "]" );
   }

   public void mousePressed( MouseEvent e )
   {
      statusBar.setText( "Pressed at [" + e.getX() +
                         ", " + e.getY() + "]" );
   }

   public void mouseReleased( MouseEvent e )
   {
      statusBar.setText( "Released at [" + e.getX() +
                         ", " + e.getY() + "]" );
   }

   public void mouseEntered( MouseEvent e )
   {
      statusBar.setText( "Mouse in window" );
   }

   public void mouseExited( MouseEvent e )
   {
      statusBar.setText( "Mouse outside window" );
   }

```

```
56      // MouseMotionListener event handlers
57      public void mouseDragged( MouseEvent e )
58      {
59         statusBar.setText( "Dragged at [" + e.getX() +
60                            ", " + e.getY() + "]" );
61      }
62
63      public void mouseMoved( MouseEvent e )
64      {
65         statusBar.setText( "Moved at [" + e.getX() +
66                            ", " + e.getY() + "]" );
67      }
68
69      public static void main( String args[] )
70      {
71         MouseTracker app = new MouseTracker();
72
73         app.addWindowListener(
74            new WindowAdapter() {
75               public void windowClosing( WindowEvent e )
76               {
77                  System.exit( 0 );
78               }
79            }
80         );
81      }
82   }
```

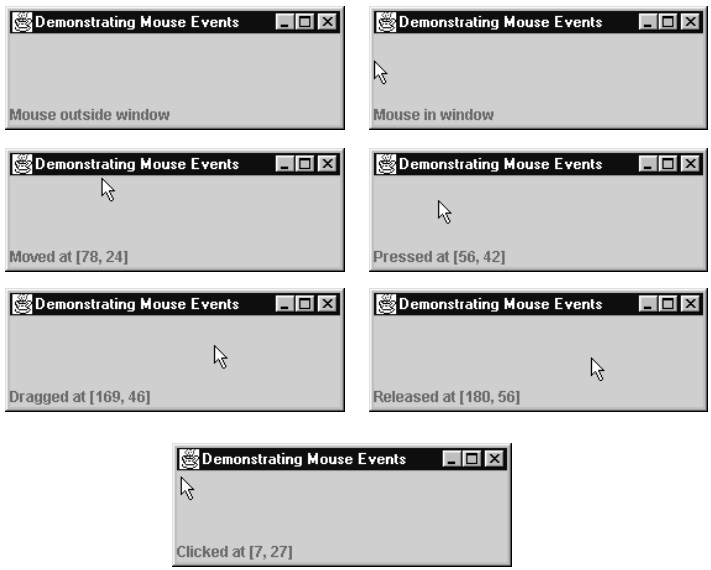

Fig. 12.17 Demonstrating mouse event handling.

Event adapter class	Implements interface
`ComponentAdapter`	`ComponentListener`
`ContainerAdapter`	`ContainerListener`
`FocusAdapter`	`FocusListener`
`KeyAdapter`	`KeyListener`
`MouseAdapter`	`MouseListener`
`MouseMotionAdapter`	`MouseMotionListener`
`WindowAdapter`	`WindowListener`

Fig. 12.18 Event adapter classes and the interfaces they implement.

```
1  // Fig. 12.19: Painter.java
2  // Using class MouseMotionAdapter.
3  import javax.swing.*;
4  import java.awt.event.*;
5  import java.awt.*;
6
7  public class Painter extends JFrame {
8     private int xValue = -10, yValue = -10;
9
10    public Painter()
11    {
12       super( "A simple paint program" );
13
14       getContentPane().add(
15          new Label( "Drag the mouse to draw" ),
16          BorderLayout.SOUTH );
17
18       addMouseMotionListener(
19          new MouseMotionAdapter() {
20             public void mouseDragged( MouseEvent e )
21             {
22                xValue = e.getX();
23                yValue = e.getY();
24                repaint();
25             }
26          }
27       );
28
29       setSize( 300, 150 );
30       show();
31    }
32
33    public void paint( Graphics g )
34    {
35       g.fillOval( xValue, yValue, 4, 4 );
36    }
37
38    public static void main( String args[] )
39    {
40       Painter app = new Painter();
41
42       app.addWindowListener(
43          new WindowAdapter() {
44             public void windowClosing( WindowEvent e )
45             {
46                System.exit( 0 );
47             }
48          }
49       );
50    }
51 }
```

Fig. 12.19 Program that demonstrates adapter classes.

```
1  // Fig. 12.20: MouseDetails.java
2  // Demonstrating mouse clicks and
3  // distinguishing between mouse buttons.
4  import javax.swing.*;
5  import java.awt.*;
6  import java.awt.event.*;
7
8  public class MouseDetails extends JFrame {
9     private String s = "";
10    private int xPos, yPos;
11
12    public MouseDetails()
13    {
14       super( "Mouse clicks and buttons" );
15
16       addMouseListener( new MouseClickHandler() );
17
18       setSize( 350, 150 );
19       show();
20    }
21
22    public void paint( Graphics g )
23    {
24       g.drawString( "Clicked @ [" + xPos + ", " + yPos + "]",
25                     xPos, yPos );
26    }
27
28    public static void main( String args[] )
29    {
30       MouseDetails app = new MouseDetails();
31
32       app.addWindowListener(
33          new WindowAdapter() {
34             public void windowClosing( WindowEvent e )
35             {
36                System.exit( 0 );
37             }
38          }
39       );
40    }
41
42    // inner class to handle mouse events
43    private class MouseClickHandler extends MouseAdapter {
44       public void mouseClicked( MouseEvent e )
45       {
46          xPos = e.getX();
47          yPos = e.getY();
48
49          String s =
50             "Clicked " + e.getClickCount() + " time(s)";
51
52          if ( e.isMetaDown() )      // Right mouse button
53             s += " with right mouse button";
54          else if ( e.isAltDown() )  // Middle mouse button
55             s += " with center mouse button";
```

```
56             else                          // Left mouse button
57                s += " with left mouse button";
58
59             setTitle( s );  // set the title bar of the window
60             repaint();
61          }
62       }
63    }
```

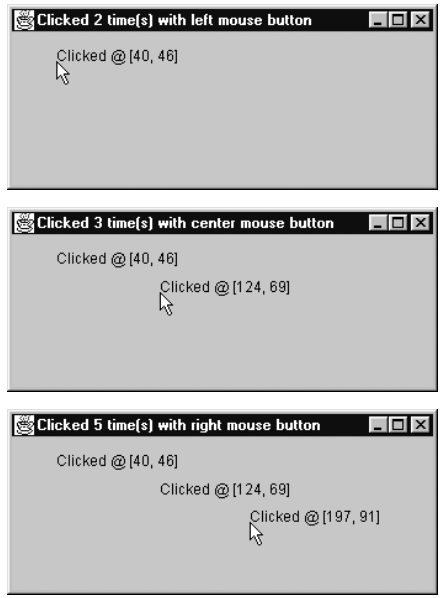

Fig. 12.20 Distinguishing among left, center and right mouse button clicks,

InputEvent method	Description
`isMetaDown()`	This method returns **`true`** when the user clicks the right mouse button on a mouse with two or three buttons. To simulate a right-mouse-button click on a one-button mouse, the user can press the *Meta* key on the keyboard and click the mouse button.
`isAltDown()`	This method returns **`true`** when the user clicks the middle mouse button on a mouse with three buttons. To simulate a middle-mouse-button click on a one- or two-button mouse, the user can press the *Alt* key on the keyboard and click the mouse button.

Fig. 12.21 **`InputEvent`** methods that help distinguish among left-, center- and right- mouse- button clicks.

```
1  // Fig. 12.22: KeyDemo.java
2  // Demonstrating keystroke events.
3  import javax.swing.*;
4  import java.awt.*;
5  import java.awt.event.*;
6
7  public class KeyDemo extends JFrame implements KeyListener {
8     private String line1 = "", line2 = "";
9     private String line3 = "";
10    private JTextArea textArea;
11
12    public KeyDemo()
13    {
14       super( "Demonstrating Keystroke Events" );
15
16       textArea = new JTextArea( 10, 15 );
17       textArea.setText( "Press any key on the keyboard..." );
18       textArea.setEnabled( false );
19
20       // allow frame to process Key events
21       addKeyListener( this );
22
23       getContentPane().add( textArea );
24
25       setSize( 350, 100 );
26       show();
27    }
28
29    public void keyPressed( KeyEvent e )
30    {
31       line1 = "Key pressed: " +
32                e.getKeyText( e.getKeyCode() );
33       setLines2and3( e );
34    }
35
36    public void keyReleased( KeyEvent e )
37    {
38       line1 = "Key released: " +
39                e.getKeyText( e.getKeyCode() );
40       setLines2and3( e );
41    }
42
43    public void keyTyped( KeyEvent e )
44    {
45       line1 = "Key typed: " + e.getKeyChar();
46       setLines2and3( e );
47    }
48
49    private void setLines2and3( KeyEvent e )
50    {
51       line2 = "This key is " +
52               ( e.isActionKey() ? "" : "not " ) +
53               "an action key";
54
```

```
55          String temp =
56             e.getKeyModifiersText( e.getModifiers() );
57
58          line3 = "Modifier keys pressed: " +
59                  ( temp.equals( "" ) ? "none" : temp );
60
61          textArea.setText(
62             line1 + "\n" + line2 + "\n" + line3 + "\n" );
63       }
64
65       public static void main( String args[] )
66       {
67          KeyDemo app = new KeyDemo();
68
69          app.addWindowListener(
70             new WindowAdapter() {
71                public void windowClosing( WindowEvent e )
72                {
73                   System.exit( 0 );
74                }
75             }
76          );
77       }
78    }
```

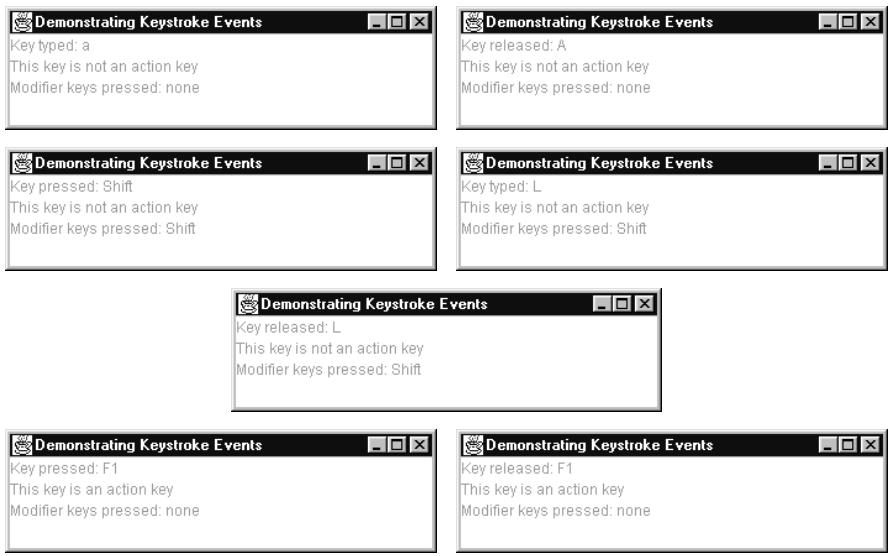

Fig. 12.22 Demonstrating key event handling.

Layout manager	Description
`FlowLayout`	Default for `java.applet.Applet`, `java.awt.Panel` and `javax.swing.JPanel`. Places components sequentially (left to right) in the order they were added. It is also possible to specify the order of the components using the `Container` method `add` that takes a `Component` and an integer index position as arguments.
`BorderLayout`	Default for the content panes of `JFrame`s (and other windows) and `JApplet`s. Arranges the components into five areas: North, South, East, West and Center.
`GridLayout`	Arranges the components into rows and columns.

Fig. 12.23 Layout managers.

```
1  // Fig. 12.24: FlowLayoutDemo.java
2  // Demonstrating FlowLayout alignments.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class FlowLayoutDemo extends JFrame {
8     private JButton left, center, right;
9     private Container c;
10    private FlowLayout layout;
11
12    public FlowLayoutDemo()
13    {
14       super( "FlowLayout Demo" );
15
16       layout = new FlowLayout();
17
18       c = getContentPane();
19       c.setLayout( layout );
20
21       left = new JButton( "Left" );
22       left.addActionListener(
23          new ActionListener() {
24             public void actionPerformed( ActionEvent e )
25             {
26                layout.setAlignment( FlowLayout.LEFT );
27
28                // re-align attached components
29                layout.layoutContainer( c );
30             }
31          }
32       );
33       c.add( left );
34
35       center = new JButton( "Center" );
36       center.addActionListener(
37          new ActionListener() {
38             public void actionPerformed( ActionEvent e )
39             {
40                layout.setAlignment( FlowLayout.CENTER );
41
42                // re-align attached components
43                layout.layoutContainer( c );
44             }
45          }
46       );
47       c.add( center );
48
49       right = new JButton( "Right" );
50       right.addActionListener(
51          new ActionListener() {
52             public void actionPerformed( ActionEvent e )
53             {
54                layout.setAlignment( FlowLayout.RIGHT );
55
```

```
56                   // re-align attached components
57                   layout.layoutContainer( c );
58                }
59             }
60          );
61          c.add( right );
62
63          setSize( 300, 75 );
64          show();
65       }
66
67       public static void main( String args[] )
68       {
69          FlowLayoutDemo app = new FlowLayoutDemo();
70
71          app.addWindowListener(
72             new WindowAdapter() {
73                public void windowClosing( WindowEvent e )
74                {
75                   System.exit( 0 );
76                }
77             }
78          );
79       }
80    }
```

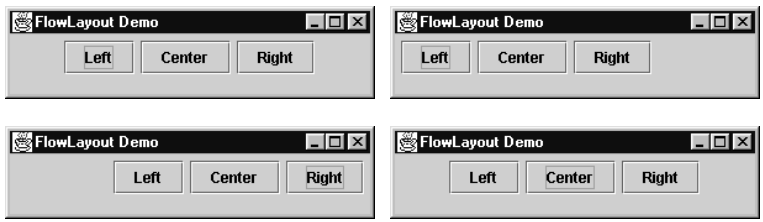

Fig. 12.24 Program that demonstrates components in **`FlowLayout`**.

```
1  // Fig. 12.25: BorderLayoutDemo.java
2  // Demonstrating BorderLayout.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class BorderLayoutDemo extends JFrame
8                                 implements ActionListener {
9     private JButton b[];
10    private String names[] =
11       { "Hide North", "Hide South", "Hide East",
12         "Hide West", "Hide Center" };
13    private BorderLayout layout;
14
15    public BorderLayoutDemo()
16    {
17       super( "BorderLayout Demo" );
18
19       layout = new BorderLayout( 5, 5 );
20
21       Container c = getContentPane();
22       c.setLayout( layout );
23
24       // instantiate button objects
25       b = new JButton[ names.length ];
26
27       for ( int i = 0; i < names.length; i++ ) {
28          b[ i ] = new JButton( names[ i ] );
29          b[ i ].addActionListener( this );
30       }
31
32       // order not important
33       c.add( b[ 0 ], BorderLayout.NORTH );  // North position
34       c.add( b[ 1 ], BorderLayout.SOUTH );  // South position
35       c.add( b[ 2 ], BorderLayout.EAST );   // East position
36       c.add( b[ 3 ], BorderLayout.WEST );   // West position
37       c.add( b[ 4 ], BorderLayout.CENTER ); // Center position
38
39       setSize( 300, 200 );
40       show();
41    }
42
43    public void actionPerformed( ActionEvent e )
44    {
45       for ( int i = 0; i < b.length; i++ )
46          if ( e.getSource() == b[ i ] )
47             b[ i ].setVisible( false );
48          else
49             b[ i ].setVisible( true );
50
51       // re-layout the content pane
52       layout.layoutContainer( getContentPane() );
53    }
54
```

```
55       public static void main( String args[] )
56       {
57          BorderLayoutDemo app = new BorderLayoutDemo();
58
59          app.addWindowListener(
60             new WindowAdapter() {
61                public void windowClosing( WindowEvent e )
62                {
63                   System.exit( 0 );
64                }
65             }
66          );
67       }
68    }
```

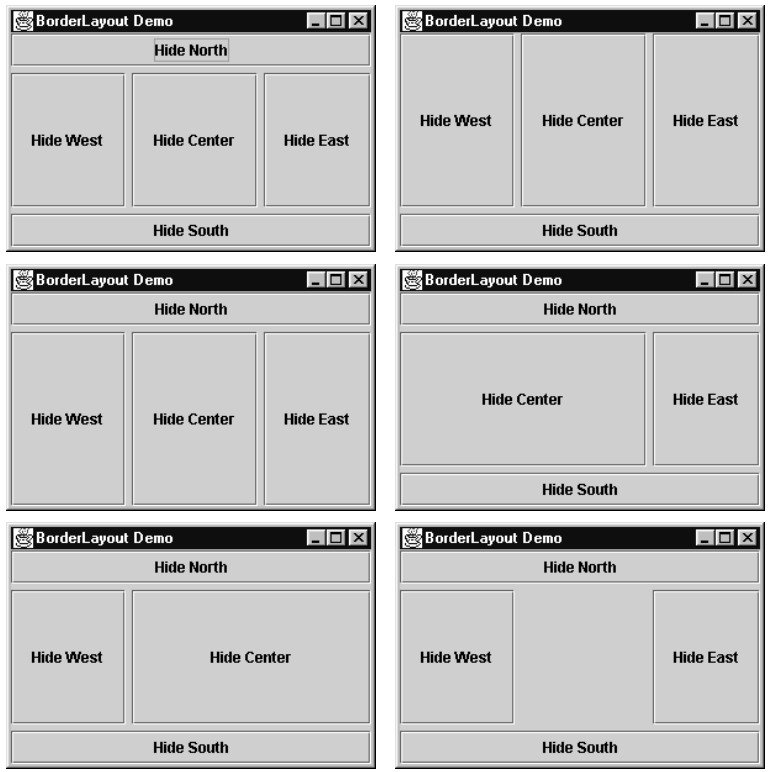

Fig. 12.25 Demonstrating components in **`BorderLayout`**.

```
1  // Fig. 12.26: GridLayoutDemo.java
2  // Demonstrating GridLayout.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class GridLayoutDemo extends JFrame
8                              implements ActionListener {
9     private JButton b[];
10    private String names[] =
11       { "one", "two", "three", "four", "five", "six" };
12    private boolean toggle = true;
13    private Container c;
14    private GridLayout grid1, grid2;
15
16    public GridLayoutDemo()
17    {
18       super( "GridLayout Demo" );
19
20       grid1 = new GridLayout( 2, 3, 5, 5 );
21       grid2 = new GridLayout( 3, 2 );
22
23       c = getContentPane();
24       c.setLayout( grid1 );
25
26       // create and add buttons
27       b = new JButton[ names.length ];
28
29       for (int i = 0; i < names.length; i++ ) {
30          b[ i ] = new JButton( names[ i ] );
31          b[ i ].addActionListener( this );
32          c.add( b[ i ] );
33       }
34
35       setSize( 300, 150 );
36       show();
37    }
38
39    public void actionPerformed( ActionEvent e )
40    {
41       if ( toggle )
42          c.setLayout( grid2 );
43       else
44          c.setLayout( grid1 );
45
46       toggle = !toggle;
47       c.validate();
48    }
49
50    public static void main( String args[] )
51    {
52       GridLayoutDemo app = new GridLayoutDemo();
53
54       app.addWindowListener(
55          new WindowAdapter() {
```

```
56                public void windowClosing( WindowEvent e )
57                {
58                   System.exit( 0 );
59                }
60             }
61          );
62       }
63    }
```

Fig. 12.26 Program that demonstrates components in **`GridLayout`**.

```
1  // Fig. 12.27: PanelDemo.java
2  // Using a JPanel to help lay out components.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class PanelDemo extends JFrame {
8     private JPanel buttonPanel;
9     private JButton buttons[];
10
11    public PanelDemo()
12    {
13       super( "Panel Demo" );
14
15       Container c = getContentPane();
16       buttonPanel = new JPanel();
17       buttons = new JButton[ 5 ];
18
19       buttonPanel.setLayout(
20          new GridLayout( 1, buttons.length ) );
21
22       for ( int i = 0; i < buttons.length; i++ ) {
23          buttons[ i ] = new JButton( "Button " + (i + 1) );
24          buttonPanel.add( buttons[ i ] );
25       }
26
27       c.add( buttonPanel, BorderLayout.SOUTH );
28
29       setSize( 425, 150 );
30       show();
31    }
32
33    public static void main( String args[] )
34    {
35       PanelDemo app = new PanelDemo();
36
37       app.addWindowListener(
38          new WindowAdapter() {
39             public void windowClosing( WindowEvent e )
40             {
41                System.exit( 0 );
42             }
43          }
44       );
45    }
46 }
```

Fig. 12.27 A **JPanel** with five **JButton**s in a **GridLayout** attached to the **SOUTH** region of a **BorderLayout** (part 1 of 2).

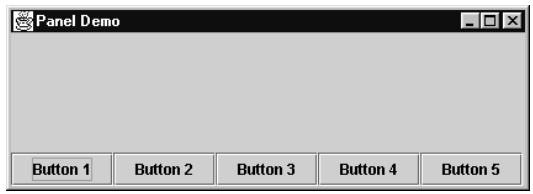

Fig. 12.27 A **`JPanel`** with five **`JButton`**s in a **`GridLayout`** attached to the **`SOUTH`** region of a **`BorderLayout`** (part 2 of 2).