Java Programming

Graphical User Interface (GUI-2)

Introduction

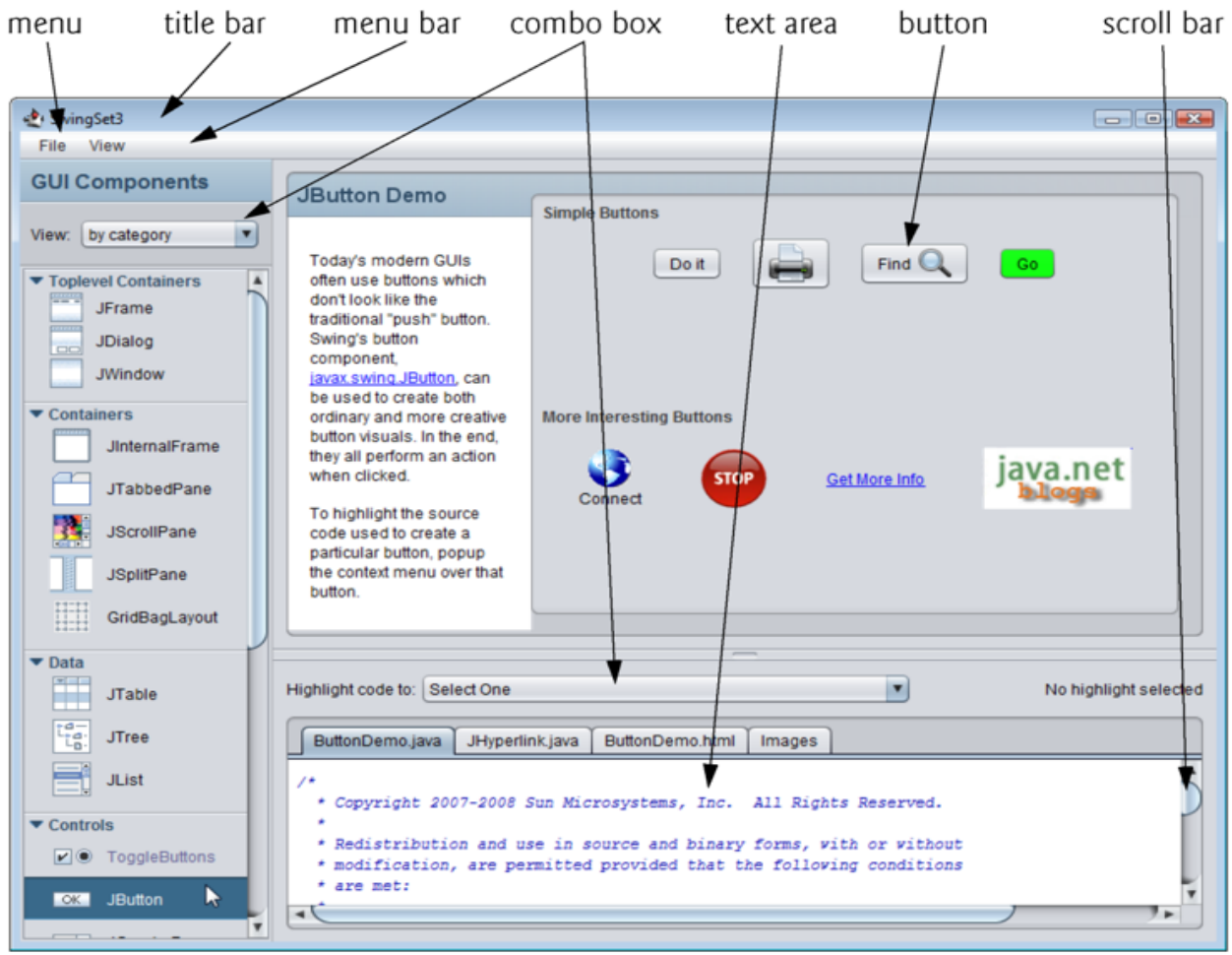

Fig. 14.1 | **SwingSet3** application demonstrates many of Java's Swing GUI components.

Introduction

- JFrame
- JOptionPane
- JLabel
- JTextField
- JButton
- JCkeckBox
- JRadioButton
- JComboBox
- JList
- JPanel
- JTextArea
- Jslider
- JPopupMenu
- JToolBar
- JTree
- JTable
- JDeskPane
- JInternalFrame
- JTabbedPane

Layout Managers

- Layout managers arrange GUI components in a container for presentation purposes
- Can use for basic layout capabilities
- Enable you to concentrate on the basic look-and-feel—the layout manager handles the layout details.
- Layout managers implement interface `LayoutManager` (in package `java.awt`).
- `Container`'s `setLayout` method takes an object that implements the `LayoutManager` interface as an argument.

Layout Managers

- There are three ways for you to arrange components in a GUI:
  - Absolute positioning
    - Greatest level of control.
    - Set Container's layout to null.
    - Specify the absolute position of each GUI component with respect to the upper-left corner of the Container by using Component methods setSize and setLocation or setBounds.
    - Must specify each GUI component's size.

Introduction to Layout Managers

- Layout managers
  - Simpler and faster than absolute positioning.
  - Lose some control over the size and the precise positioning of GUI components.

- Visual programming in an IDE
  - Use tools that make it easy to create GUIs.
  - Allows you to drag and drop GUI components from a tool box onto a design area.
  - You can then position, size and align GUI components as you like.

Layout Managers

Layout manager	Description
`FlowLayout`	Default for `javax.swing.JPanel`. Places components sequentially (left to right) in the order they were added. It's also possible to specify the order of the components by using the `Container` method `add`, which takes a `Component` and an integer index position as arguments.
`BorderLayout`	Default for `JFrames` (and other windows). Arranges the components into five areas: `NORTH`, `SOUTH`, `EAST`, `WEST` and `CENTER`.
`GridLayout`	Arranges the components into rows and columns.

Fig. 14.38 | Layout managers.

FlowLayout

- `FlowLayout` is the simplest layout manager.
- GUI components placed from left to right in the order in which they are added to the container.
- When the edge of the container is reached, components continue to display on the next line.
- `FlowLayout` allows GUI components to be left aligned, centered (the default) and right aligned.

FlowLayout

```
1  // Fig. 14.39: FlowLayoutFrame.java
2  // Demonstrating FlowLayout alignments.
3  import java.awt.FlowLayout;
4  import java.awt.Container;
5  import java.awt.event.ActionListener;
6  import java.awt.event.ActionEvent;
7  import javax.swing.JFrame;
8  import javax.swing.JButton;
9
10 public class FlowLayoutFrame extends JFrame
11 {
12    private JButton leftJButton; // button to set alignment left
13    private JButton centerJButton; // button to set alignment center
14    private JButton rightJButton; // button to set alignment right
15    private FlowLayout layout; // layout object
16    private Container container; // container to set layout
17
```

Fig. 14.39 | `FlowLayout` allows components to flow over multiple lines. (Part 1 of 5.)

FlowLayout

```
18      // set up GUI and register button listeners
19      public FlowLayoutFrame()
20      {
21         super( "FlowLayout Demo" );
22
23         layout = new FlowLayout(); // create FlowLayout
24         container = getContentPane(); // get container to layout
25         setLayout( layout ); // set frame layout
26
27         // set up leftJButton and register listener
28         leftJButton = new JButton( "Left" ); // create Left button
29         add( leftJButton ); // add Left button to frame
30         leftJButton.addActionListener(
31
```

Fig. 14.39 | FlowLayout allows components to flow over multiple lines. (Part 2 of 5.)

FlowLayout

```
32            new ActionListener() // anonymous inner class
33            {
34               // process leftJButton event
35               public void actionPerformed( ActionEvent event )
36               {
37                  layout.setAlignment( FlowLayout.LEFT );
38
39                  // realign attached components
40                  layout.layoutContainer( container );
41               } // end method actionPerformed
42            } // end anonymous inner class
43         ); // end call to addActionListener
44
45         // set up centerJButton and register listener
46         centerJButton = new JButton( "Center" ); // create Center button
47         add( centerJButton ); // add Center button to frame
48         centerJButton.addActionListener(
49
```

Fig. 14.39 | `FlowLayout` allows components to flow over multiple lines. (Part 3 of 5.)

FlowLayout

```
50            new ActionListener() // anonymous inner class
51            {
52               // process centerJButton event
53               public void actionPerformed( ActionEvent event )
54               {
55                  layout.setAlignment( FlowLayout.CENTER );
56
57                  // realign attached components
58                  layout.layoutContainer( container );
59               } // end method actionPerformed
60            } // end anonymous inner class
61         ); // end call to addActionListener
62
```

Fig. 14.39 | `FlowLayout` allows components to flow over multiple lines. (Part 4 of 5.)

FlowLayout

```
63          // set up rightJButton and register listener
64          rightJButton = new JButton( "Right" ); // create Right button
65          add( rightJButton ); // add Right button to frame
66          rightJButton.addActionListener(
67
68             new ActionListener() // anonymous inner class
69             {
70                // process rightJButton event
71                public void actionPerformed( ActionEvent event )
72                {
73                   layout.setAlignment( FlowLayout.RIGHT );
74
75                   // realign attached components
76                   layout.layoutContainer( container );
77                } // end method actionPerformed
78             } // end anonymous inner class
79          ); // end call to addActionListener
80       } // end FlowLayoutFrame constructor
81    } // end class FlowLayoutFrame
```

Fig. 14.39 | FlowLayout allows components to flow over multiple lines. (Part 5 of 5.)

FlowLayout

```
1  // Fig. 14.40: FlowLayoutDemo.java
2  // Testing FlowLayoutFrame.
3  import javax.swing.JFrame;
4
5  public class FlowLayoutDemo
6  {
7     public static void main( String[] args )
8     {
9        FlowLayoutFrame flowLayoutFrame = new FlowLayoutFrame();
10       flowLayoutFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       flowLayoutFrame.setSize( 300, 75 ); // set frame size
12       flowLayoutFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class FlowLayoutDemo
```

Fig. 14.40 | Test class for `FlowLayoutFrame`. (Part 1 of 2.)

FlowLayout

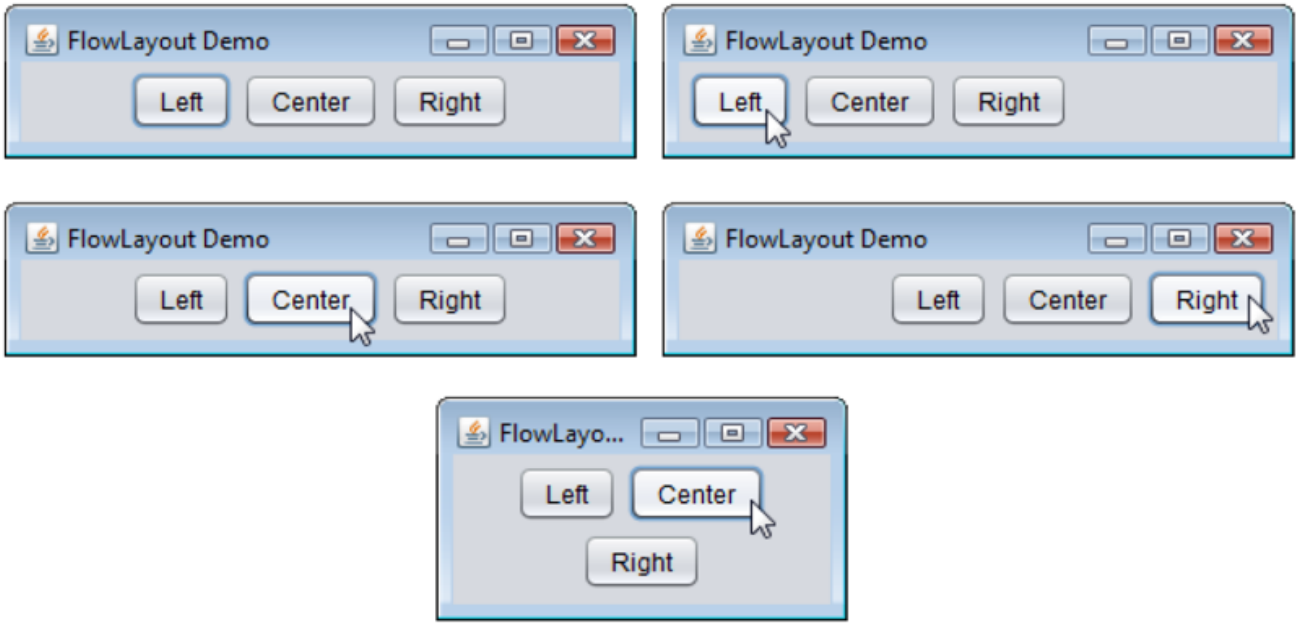

Fig. 14.40 | Test class for `FlowLayoutFrame`. (Part 2 of 2.)

BorderLayout

- `BorderLayout`
  - the default layout manager for a `Jframe`
  - arranges components into five regions: `NORTH`, `SOUTH`, `EAST`, `WEST` and `CENTER`.
  - `NORTH` corresponds to the top of the container.

- `BorderLayout` implements interface `LayoutManager2` (a subinterface of `LayoutManager` that adds several methods for enhanced layout processing).

- `BorderLayout` limits a `Container` to at most five components—one in each region.
  - The component placed in each region can be a container to which other components are attached.

BorderLayout

```
1  // Fig. 14.41: BorderLayoutFrame.java
2  // Demonstrating BorderLayout.
3  import java.awt.BorderLayout;
4  import java.awt.event.ActionListener;
5  import java.awt.event.ActionEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JButton;
8
9  public class BorderLayoutFrame extends JFrame implements ActionListener
10 {
11    private JButton[] buttons; // array of buttons to hide portions
12    private static final String[] names = { "Hide North", "Hide South",
13       "Hide East", "Hide West", "Hide Center" };
14    private BorderLayout layout; // borderlayout object
15
16    // set up GUI and event handling
17    public BorderLayoutFrame()
18    {
19       super( "BorderLayout Demo" );
20
21       layout = new BorderLayout( 5, 5 ); // 5 pixel gaps
22       setLayout( layout ); // set frame layout
23       buttons = new JButton[ names.length ]; // set size of array
24
```

Fig. 14.41 | `BorderLayout` containing five buttons. (Part 1 of 3.)

BorderLayout

```
25          // create JButtons and register listeners for them
26          for ( int count = 0; count < names.length; count++ )
27          {
28             buttons[ count ] = new JButton( names[ count ] );
29             buttons[ count ].addActionListener( this );
30          } // end for
31
32          add( buttons[ 0 ], BorderLayout.NORTH ); // add button to north
33          add( buttons[ 1 ], BorderLayout.SOUTH ); // add button to south
34          add( buttons[ 2 ], BorderLayout.EAST ); // add button to east
35          add( buttons[ 3 ], BorderLayout.WEST ); // add button to west
36          add( buttons[ 4 ], BorderLayout.CENTER ); // add button to center
37       } // end BorderLayoutFrame constructor
38
```

Fig. 14.41 | BorderLayout containing five buttons. (Part 2 of 3.)

BorderLayout

```
39      // handle button events
40      public void actionPerformed( ActionEvent event )
41      {
42         // check event source and lay out content pane correspondingly
43         for ( JButton button : buttons )
44         {
45            if ( event.getSource() == button )
46               button.setVisible( false ); // hide button clicked
47            else
48               button.setVisible( true ); // show other buttons
49         } // end for
50
51         layout.layoutContainer( getContentPane() ); // lay out content pane
52      } // end method actionPerformed
53   } // end class BorderLayoutFrame
```

Fig. 14.41 | `BorderLayout` containing five buttons. (Part 3 of 3.)

BorderLayout

```
1  // Fig. 14.42: BorderLayoutDemo.java
2  // Testing BorderLayoutFrame.
3  import javax.swing.JFrame;
4
5  public class BorderLayoutDemo
6  {
7     public static void main( String[] args )
8     {
9        BorderLayoutFrame borderLayoutFrame = new BorderLayoutFrame();
10       borderLayoutFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       borderLayoutFrame.setSize( 300, 200 ); // set frame size
12       borderLayoutFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class BorderLayoutDemo
```

Fig. 14.42 | Test class for `BorderLayoutFrame`. (Part 1 of 3.)

BorderLayout

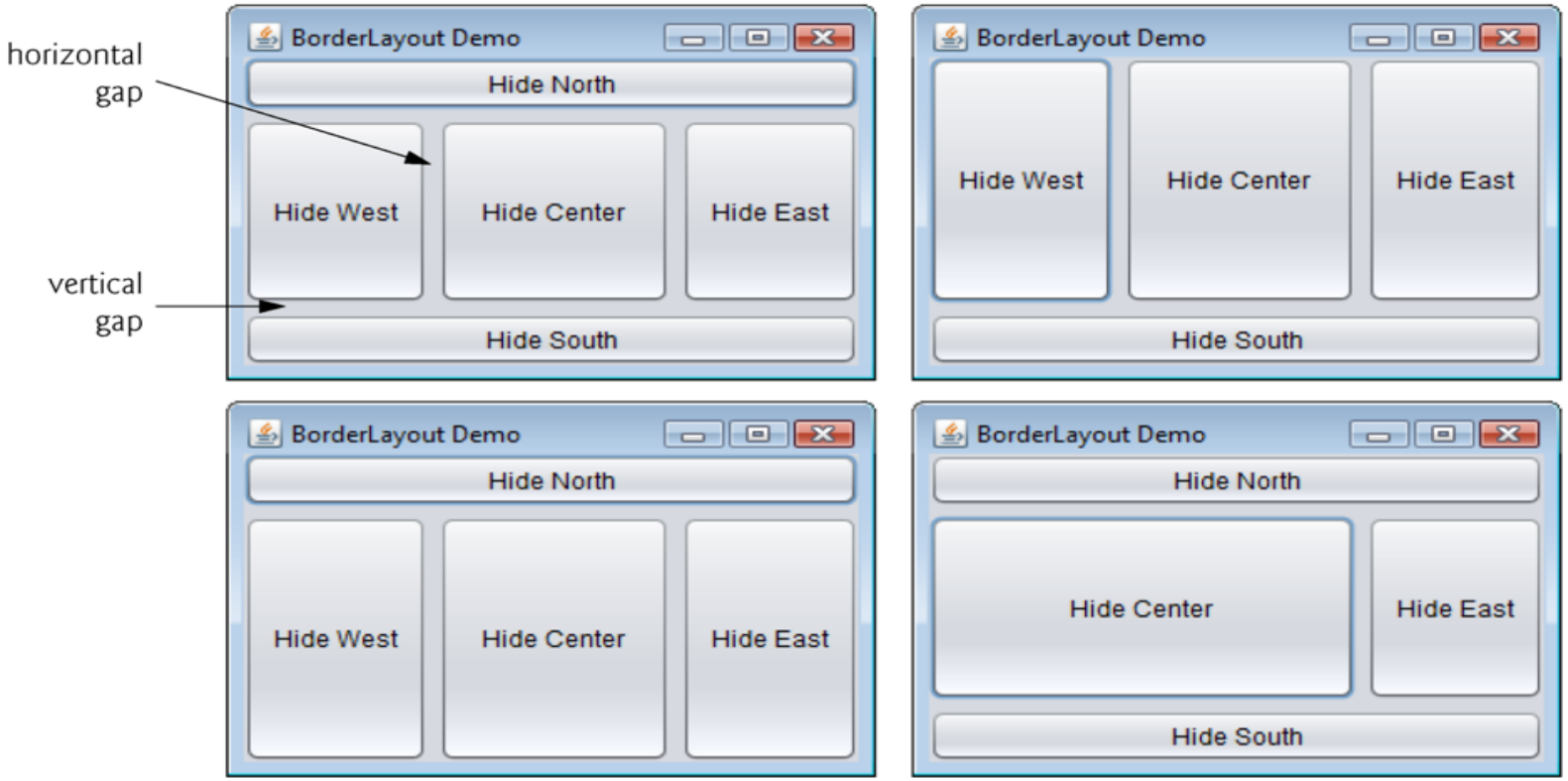

Fig. 14.42 | Test class for `BorderLayoutFrame`. (Part 2 of 3.)

BorderLayout

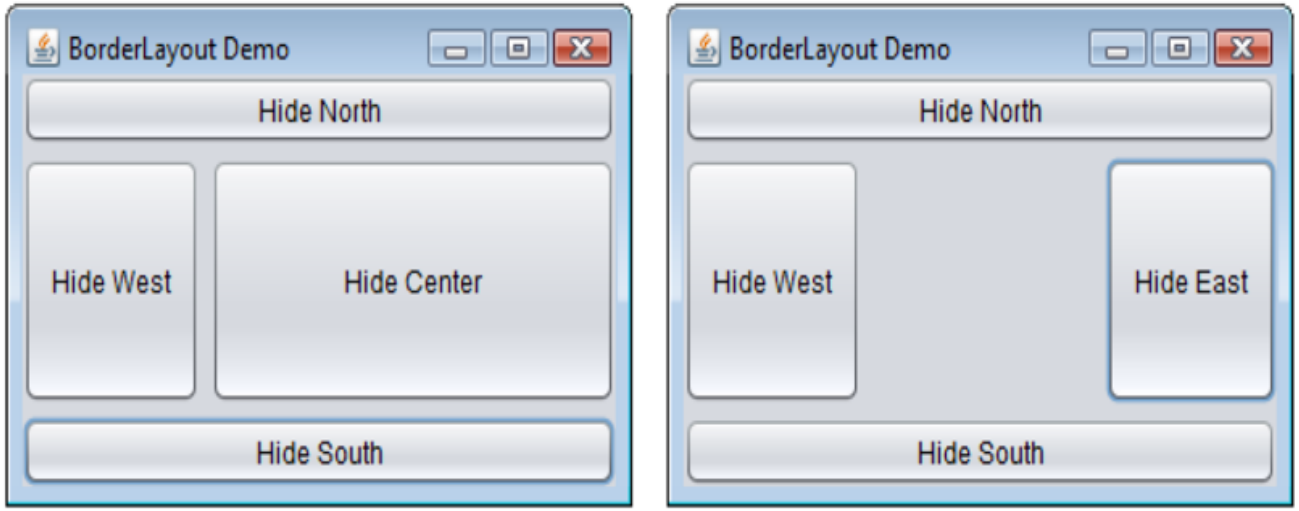

Fig. 14.42 | Test class for `BorderLayoutFrame`. (Part 3 of 3.)

GridLayout

- `GridLayout` divides the container into a grid of rows and columns.
  - Implements interface `LayoutManager`.
  - Every `Component` has the same width and height.
  - Components are added starting at the top-left cell of the grid and proceeding left to right until the row is full. Then the process continues left to right on the next row of the grid, and so on.
- `Container` method `validate` recomputes the container's layout based on the current layout manager and the current set of displayed GUI components.

GridLayout

```
1  // Fig. 14.43: GridLayoutFrame.java
2  // Demonstrating GridLayout.
3  import java.awt.GridLayout;
4  import java.awt.Container;
5  import java.awt.event.ActionListener;
6  import java.awt.event.ActionEvent;
7  import javax.swing.JFrame;
8  import javax.swing.JButton;
9
10 public class GridLayoutFrame extends JFrame implements ActionListener
11 {
12    private JButton[] buttons; // array of buttons
13    private static final String[] names =
14       { "one", "two", "three", "four", "five", "six" };
15    private boolean toggle = true; // toggle between two layouts
16    private Container container; // frame container
17    private GridLayout gridLayout1; // first gridlayout
18    private GridLayout gridLayout2; // second gridlayout
19
```

Fig. 14.43 | `GridLayout` containing six buttons. (Part 1 of 3.)

GridLayout

```
20      // no-argument constructor
21      public GridLayoutFrame()
22      {
23         super( "GridLayout Demo" );
24         gridLayout1 = new GridLayout( 2, 3, 5, 5 ); // 2 by 3; gaps of 5
25         gridLayout2 = new GridLayout( 3, 2 ); // 3 by 2; no gaps
26         container = getContentPane(); // get content pane
27         setLayout( gridLayout1 ); // set JFrame layout
28         buttons = new JButton[ names.length ]; // create array of JButtons
29
30         for ( int count = 0; count < names.length; count++ )
31         {
32            buttons[ count ] = new JButton( names[ count ] );
33            buttons[ count ].addActionListener( this ); // register listener
34            add( buttons[ count ] ); // add button to JFrame
35         } // end for
36      } // end GridLayoutFrame constructor
37
```

Fig. 14.43 | `GridLayout` containing six buttons. (Part 2 of 3.)

GridLayout

```
38      // handle button events by toggling between layouts
39      public void actionPerformed( ActionEvent event )
40      {
41         if ( toggle )
42            container.setLayout( gridLayout2 ); // set layout to second
43         else
44            container.setLayout( gridLayout1 ); // set layout to first
45
46         toggle = !toggle; // set toggle to opposite value
47         container.validate(); // re-lay out container
48      } // end method actionPerformed
49   } // end class GridLayoutFrame
```

Fig. 14.43 | `GridLayout` containing six buttons. (Part 3 of 3.)

GridLayout

```
1  // Fig. 14.44: GridLayoutDemo.java
2  // Testing GridLayoutFrame.
3  import javax.swing.JFrame;
4
5  public class GridLayoutDemo
6  {
7     public static void main( String[] args )
8     {
9        GridLayoutFrame gridLayoutFrame = new GridLayoutFrame();
10       gridLayoutFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       gridLayoutFrame.setSize( 300, 200 ); // set frame size
12       gridLayoutFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class GridLayoutDemo
```

Fig. 14.44 | Test class for `GridLayoutFrame`.

Using Panels to Manage More Complex Layouts

- Complex GUIs require that each component be placed in an exact location.
  - Often consist of multiple panels, with each panel's components arranged in a specific layout.
- Class `JPanel` extends `JComponent` and `JComponent` extends class `Container`, so every `JPanel` is a `Container`.
- Every `JPanel` may have components, including other panels, attached to it with `Container` method `add`.
- `JPanel` can be used to create a more complex layout in which several components are in a specific area of another container.

Using Panels to Manage More Complex Layouts

```
1  // Fig. 14.45: PanelFrame.java
2  // Using a JPanel to help lay out components.
3  import java.awt.GridLayout;
4  import java.awt.BorderLayout;
5  import javax.swing.JFrame;
6  import javax.swing.JPanel;
7  import javax.swing.JButton;
8
9  public class PanelFrame extends JFrame
10 {
11    private JPanel buttonJPanel; // panel to hold buttons
12    private JButton[] buttons; // array of buttons
13
14    // no-argument constructor
15    public PanelFrame()
16    {
17       super( "Panel Demo" );
18       buttons = new JButton[ 5 ]; // create buttons array
19       buttonJPanel = new JPanel(); // set up panel
20       buttonJPanel.setLayout( new GridLayout( 1, buttons.length ) );
21
```

Fig. 14.45 | `JPanel` with five `JButtons` in a `GridLayout` attached to the `SOUTH` region of a `BorderLayout`. (Part 1 of 2.)

Using Panels to Manage More Complex Layouts

```
22          // create and add buttons
23          for ( int count = 0; count < buttons.length; count++ )
24          {
25             buttons[ count ] = new JButton( "Button " + ( count + 1 ) );
26             buttonJPanel.add( buttons[ count ] ); // add button to panel
27          } // end for
28
29          add( buttonJPanel, BorderLayout.SOUTH ); // add panel to JFrame
30       } // end PanelFrame constructor
31    } // end class PanelFrame
```

Fig. 14.45 | `JPanel` with five `JButtons` in a `GridLayout` attached to the `SOUTH` region of a `BorderLayout`. (Part 2 of 2.)

Using Panels to Manage More Complex Layouts

```
1  // Fig. 14.46: PanelDemo.java
2  // Testing PanelFrame.
3  import javax.swing.JFrame;
4
5  public class PanelDemo extends JFrame
6  {
7     public static void main( String[] args )
8     {
9        PanelFrame panelFrame = new PanelFrame();
10       panelFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       panelFrame.setSize( 450, 200 ); // set frame size
12       panelFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class PanelDemo
```

Fig. 14.46 | Test class for `PanelFrame`.

JComboBox; Using an Anonymous Inner Class for Event Handling

- A combo box (or drop-down list) enables the user to select one item from a list.
- Combo boxes are implemented with class `JComboBox`, which extends class `JComponent`.
- `JComboBox`es generate `ItemEvent`s.

JComboBox; Using an Anonymous Inner Class for Event Handling

```
1  // Fig. 14.21: ComboBoxFrame.java
2  // JComboBox that displays a list of image names.
3  import java.awt.FlowLayout;
4  import java.awt.event.ItemListener;
5  import java.awt.event.ItemEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JLabel;
8  import javax.swing.JComboBox;
9  import javax.swing.Icon;
10 import javax.swing.ImageIcon;
11
12 public class ComboBoxFrame extends JFrame
13 {
14    private JComboBox imagesJComboBox; // combobox to hold names of icons
15    private JLabel label; // label to display selected icon
16
17    private static final String[] names =
18       { "bug1.gif", "bug2.gif",  "travelbug.gif", "buganim.gif" };
19    private Icon[] icons = {
20       new ImageIcon( getClass().getResource( names[ 0 ] ) ),
21       new ImageIcon( getClass().getResource( names[ 1 ] ) ),
22       new ImageIcon( getClass().getResource( names[ 2 ] ) ),
23       new ImageIcon( getClass().getResource( names[ 3 ] ) ) };
24
```

Fig. 14.21 | JComboBox that displays a list of image names. (Part 1 of 3.)

JComboBox; Using an Anonymous Inner Class for Event Handling

```
25      // ComboBoxFrame constructor adds JComboBox to JFrame
26      public ComboBoxFrame()
27      {
28         super( "Testing JComboBox" );
29         setLayout( new FlowLayout() ); // set frame layout
30
31         imagesJComboBox = new JComboBox( names ); // set up JComboBox
32         imagesJComboBox.setMaximumRowCount( 3 ); // display three rows
33
34         imagesJComboBox.addItemListener(
35            new ItemListener() // anonymous inner class
36            {
37               // handle JComboBox event
38               public void itemStateChanged( ItemEvent event )
39               {
40                  // determine whether item selected
41                  if ( event.getStateChange() == ItemEvent.SELECTED )
42                     label.setIcon( icons[
43                        imagesJComboBox.getSelectedIndex() ] );
44               } // end method itemStateChanged
45            } // end anonymous inner class
46         ); // end call to addItemListener
47
```

Fig. 14.21 | JComboBox that displays a list of image names. (Part 2 of 3.)

JComboBox; Using an Anonymous Inner Class for Event Handling

```
48          add( imagesJComboBox ); // add combobox to JFrame
49          label = new JLabel( icons[ 0 ] ); // display first icon
50          add( label ); // add label to JFrame
51       } // end ComboBoxFrame constructor
52    } // end class ComboBoxFrame
```

Fig. 14.21 | JComboBox that displays a list of image names. (Part 3 of 3.)

JComboBox; Using an Anonymous Inner Class for Event Handling

```
1  // Fig. 14.22: ComboBoxTest.java
2  // Testing ComboBoxFrame.
3  import javax.swing.JFrame;
4
5  public class ComboBoxTest
6  {
7     public static void main( String[] args )
8     {
9        ComboBoxFrame comboBoxFrame = new ComboBoxFrame();
10       comboBoxFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       comboBoxFrame.setSize( 350, 150 ); // set frame size
12       comboBoxFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class ComboBoxTest
```

Fig. 14.22 | Testing ComboBoxFrame. (Part 1 of 2.)

JComboBox; Using an Anonymous Inner Class for Event Handling

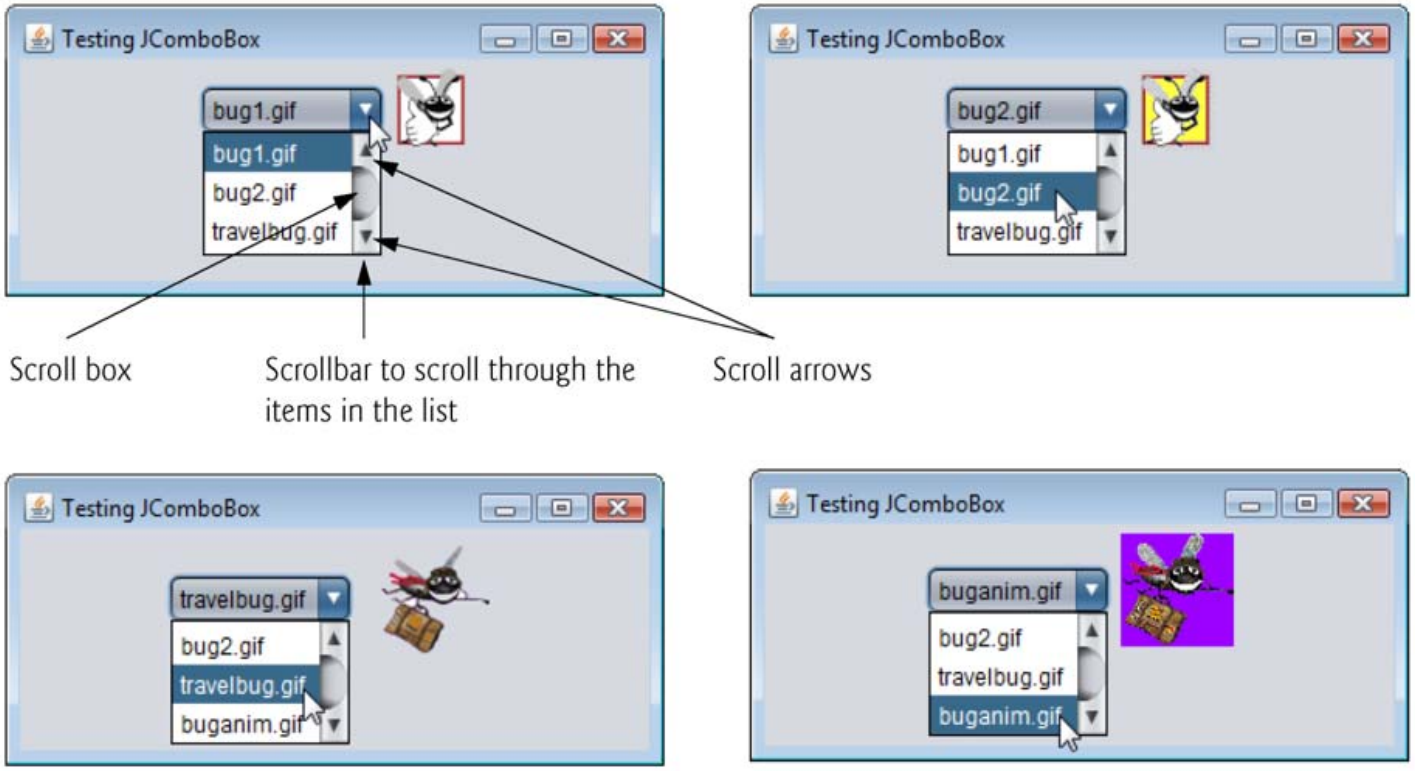

Fig. 14.22 | Testing `ComboBoxFrame`. (Part 2 of 2.)

JList

- A list displays a series of items from which the user may select one or more items.
- Lists are created with class `JList`, which directly extends class `JComponent`.
- Supports single-selection lists (only one item to be selected at a time) and multiple-selection lists (any number of items to be selected).
- `JList`s generate `ListSelectionEvent`s in single-selection lists.

JList

```
1  // Fig. 14.23: ListFrame.java
2  // JList that displays a list of colors.
3  import java.awt.FlowLayout;
4  import java.awt.Color;
5  import javax.swing.JFrame;
6  import javax.swing.JList;
7  import javax.swing.JScrollPane;
8  import javax.swing.event.ListSelectionListener;
9  import javax.swing.event.ListSelectionEvent;
10 import javax.swing.ListSelectionModel;
11
12 public class ListFrame extends JFrame
13 {
14    private JList colorJList; // list to display colors
15    private static final String[] colorNames = { "Black", "Blue", "Cyan",
16       "Dark Gray", "Gray", "Green", "Light Gray", "Magenta",
17       "Orange", "Pink", "Red", "White", "Yellow" };
18    private static final Color[] colors = { Color.BLACK, Color.BLUE,
19       Color.CYAN, Color.DARK_GRAY, Color.GRAY, Color.GREEN,
20       Color.LIGHT_GRAY, Color.MAGENTA, Color.ORANGE, Color.PINK,
21       Color.RED, Color.WHITE, Color.YELLOW };
22
```

Fig. 14.23 | `JList` that displays a list of colors. (Part 1 of 3.)

JList

```
23      // ListFrame constructor add JScrollPane containing JList to JFrame
24      public ListFrame()
25      {
26         super( "List Test" );
27         setLayout( new FlowLayout() ); // set frame layout
28
29         colorJList = new JList( colorNames ); // create with colorNames
30         colorJList.setVisibleRowCount( 5 ); // display five rows at once
31
32         // do not allow multiple selections
33         colorJList.setSelectionMode( ListSelectionModel.SINGLE_SELECTION );
34
35         // add a JScrollPane containing JList to frame
36         add( new JScrollPane( colorJList ) );
37
```

Fig. 14.23 | `JList` that displays a list of colors. (Part 2 of 3.)

JList

```
38          colorJList.addListSelectionListener(
39             new ListSelectionListener() // anonymous inner class
40             {
41                // handle list selection events
42                public void valueChanged( ListSelectionEvent event )
43                {
44                   getContentPane().setBackground(
45                      colors[ colorJList.getSelectedIndex() ] );
46                } // end method valueChanged
47             } // end anonymous inner class
48          ); // end call to addListSelectionListener
49       } // end ListFrame constructor
50    } // end class ListFrame
```

Fig. 14.23 | `JList` that displays a list of colors. (Part 3 of 3.)

JList

```
1  // Fig. 14.24: ListTest.java
2  // Selecting colors from a JList.
3  import javax.swing.JFrame;
4
5  public class ListTest
6  {
7     public static void main( String[] args )
8     {
9        ListFrame listFrame = new ListFrame(); // create ListFrame
10       listFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       listFrame.setSize( 350, 150 ); // set frame size
12       listFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class ListTest
```

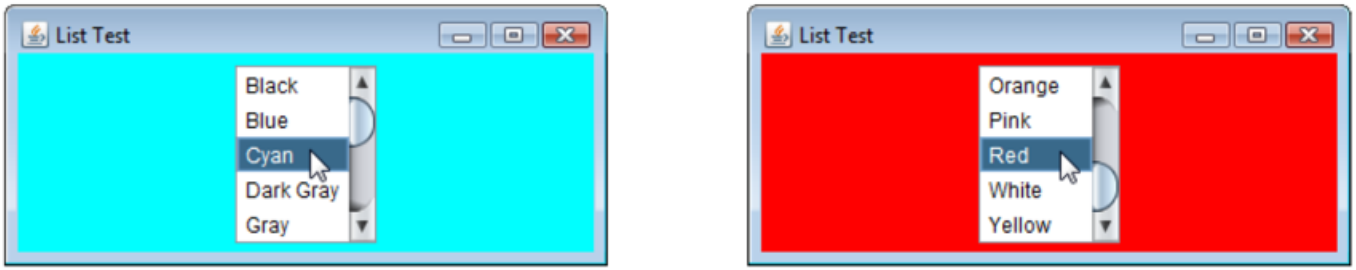

Fig. 14.24 | Test class for `ListFrame`.

Mouse Event Handling

- `MouseListener` and `MouseMotionListener` event-listener interfaces for handling mouse events.
  - Any GUI component

- Package `javax.swing.event` contains interface `MouseInputListener`, which extends interfaces `MouseListener` and `MouseMotionListener` to create a single interface containing all the methods.

- `MouseListener` and `MouseMotionListener` methods are called when the mouse interacts with a `Component` if appropriate event-listener objects are registered for that `Component`.

Mouse Event Handling

MouseListener and MouseMotionListener interface methods

Methods of interface `MouseListener`

`public void mousePressed( MouseEvent event )`

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

`public void mouseClicked( MouseEvent event )`

Called when a mouse button is *pressed and released* while the mouse cursor remains stationary on a component. This event is always preceded by a call to `mousePressed`.

`public void mouseReleased( MouseEvent event )`

Called when a mouse button is *released after being pressed*. This event is always preceded by a call to `mousePressed` and one or more calls to `mouseDragged`.

`public void mouseEntered( MouseEvent event )`

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

`public void mouseExited( MouseEvent event )`

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

Fig. 14.27 | `MouseListener` and `MouseMotionListener` interface methods. (Part 1 of 2.)

Mouse Event Handling

MouseListener and MouseMotionListener interface methods

Methods of interface `MouseMotionListener`

`public void mouseDragged( MouseEvent event )`

Called when the mouse button is *pressed* while the mouse cursor is on a component and the mouse is *moved* while the mouse button *remains pressed*. This event is always preceded by a call to `mousePressed`. All drag events are sent to the component on which the user began to drag the mouse.

`public void mouseMoved( MouseEvent event )`

Called when the mouse is *moved* (with no mouse buttons pressed) when the mouse cursor is on a component. All move events are sent to the component over which the mouse is currently positioned.

Fig. 14.27 | `MouseListener` and `MouseMotionListener` interface methods. (Part 2 of 2.)

Mouse Event Handling

```
1  // Fig. 14.28: MouseTrackerFrame.java
2  // Demonstrating mouse events.
3  import java.awt.Color;
4  import java.awt.BorderLayout;
5  import java.awt.event.MouseListener;
6  import java.awt.event.MouseMotionListener;
7  import java.awt.event.MouseEvent;
8  import javax.swing.JFrame;
9  import javax.swing.JLabel;
10 import javax.swing.JPanel;
11
12 public class MouseTrackerFrame extends JFrame
13 {
14    private JPanel mousePanel; // panel in which mouse events will occur
15    private JLabel statusBar; // label that displays event information
16
```

Fig. 14.28 | Mouse event handling. (Part 1 of 5.)

Mouse Event Handling

```
17      // MouseTrackerFrame constructor sets up GUI and
18      // registers mouse event handlers
19      public MouseTrackerFrame()
20      {
21         super( "Demonstrating Mouse Events" );
22
23         mousePanel = new JPanel(); // create panel
24         mousePanel.setBackground( Color.WHITE ); // set background color
25         add( mousePanel, BorderLayout.CENTER ); // add panel to JFrame
26
27         statusBar = new JLabel( "Mouse outside JPanel" );
28         add( statusBar, BorderLayout.SOUTH ); // add label to JFrame
29
30         // create and register listener for mouse and mouse motion events
31         MouseHandler handler = new MouseHandler();
32         mousePanel.addMouseListener( handler );
33         mousePanel.addMouseMotionListener( handler );
34      } // end MouseTrackerFrame constructor
35
```

Fig. 14.28 | Mouse event handling. (Part 2 of 5.)

Mouse Event Handling

```
36      private class MouseHandler implements MouseListener,
37         MouseMotionListener
38      {
39         // MouseListener event handlers
40         // handle event when mouse released immediately after press
41         public void mouseClicked( MouseEvent event )
42         {
43            statusBar.setText( String.format( "Clicked at [%d, %d]",
44               event.getX(), event.getY() ) );
45         } // end method mouseClicked
46
47         // handle event when mouse pressed
48         public void mousePressed( MouseEvent event )
49         {
50            statusBar.setText( String.format( "Pressed at [%d, %d]",
51               event.getX(), event.getY() ) );
52         } // end method mousePressed
53
```

Fig. 14.28 | Mouse event handling. (Part 3 of 5.)

Mouse Event Handling

```
54          // handle event when mouse released
55          public void mouseReleased( MouseEvent event )
56          {
57             statusBar.setText( String.format( "Released at [%d, %d]",
58                event.getX(), event.getY() ) );
59          } // end method mouseReleased
60
61          // handle event when mouse enters area
62          public void mouseEntered( MouseEvent event )
63          {
64             statusBar.setText( String.format( "Mouse entered at [%d, %d]",
65                event.getX(), event.getY() ) );
66             mousePanel.setBackground( Color.GREEN );
67          } // end method mouseEntered
68
69          // handle event when mouse exits area
70          public void mouseExited( MouseEvent event )
71          {
72             statusBar.setText( "Mouse outside JPanel" );
73             mousePanel.setBackground( Color.WHITE );
74          } // end method mouseExited
75
```

Fig. 14.28 | Mouse event handling. (Part 4 of 5.)

Mouse Event Handling

```
76          // MouseMotionListener event handlers
77          // handle event when user drags mouse with button pressed
78          public void mouseDragged( MouseEvent event )
79          {
80             statusBar.setText( String.format( "Dragged at [%d, %d]",
81                event.getX(), event.getY() ) );
82          } // end method mouseDragged
83
84          // handle event when user moves mouse
85          public void mouseMoved( MouseEvent event )
86          {
87             statusBar.setText( String.format( "Moved at [%d, %d]",
88                event.getX(), event.getY() ) );
89          } // end method mouseMoved
90       } // end inner class MouseHandler
91    } // end class MouseTrackerFrame
```

Fig. 14.28 | Mouse event handling. (Part 5 of 5.)

Mouse Event Handling

```
1  // Fig. 14.29: MouseTrackerFrame.java
2  // Testing MouseTrackerFrame.
3  import javax.swing.JFrame;
4
5  public class MouseTracker
6  {
7     public static void main( String[] args )
8     {
9        MouseTrackerFrame mouseTrackerFrame = new MouseTrackerFrame();
10       mouseTrackerFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       mouseTrackerFrame.setSize( 300, 100 ); // set frame size
12       mouseTrackerFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class MouseTracker
```

Fig. 14.29 | Test class for `MouseTrackerFrame`. (Part 1 of 2.)

Mouse Event Handling

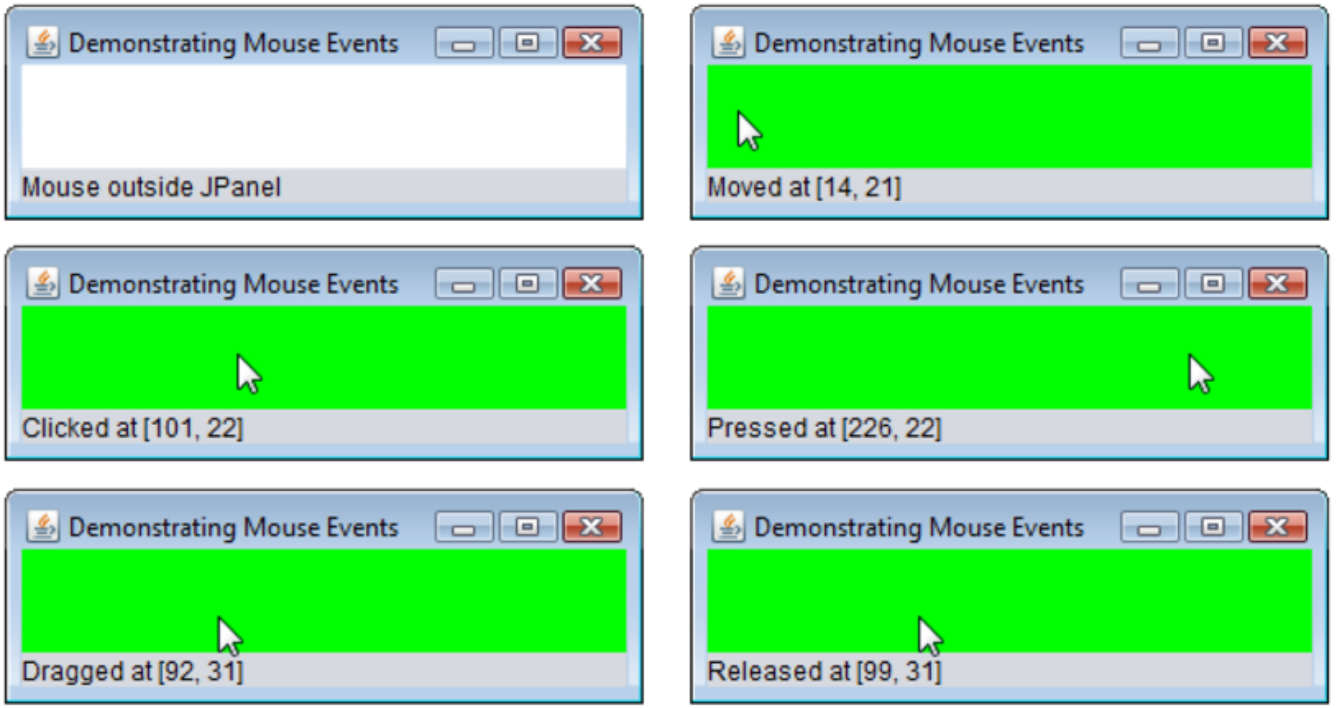

Fig. 14.29 | Test class for `MouseTrackerFrame`. (Part 2 of 2.)

JTextArea

- A `JTextArea` provides an area for manipulating multiple lines of text.

- `JTextArea` is a subclass of `JTextComponent`, which declares common methods for `JTextFields`, `JTextAreas` and several other text-based GUI components.

JTextArea

```
1  // Fig. 14.47: TextAreaFrame.java
2  // Copying selected text from one textarea to another.
3  import java.awt.event.ActionListener;
4  import java.awt.event.ActionEvent;
5  import javax.swing.Box;
6  import javax.swing.JFrame;
7  import javax.swing.JTextArea;
8  import javax.swing.JButton;
9  import javax.swing.JScrollPane;
10
11 public class TextAreaFrame extends JFrame
12 {
13    private JTextArea textArea1; // displays demo string
14    private JTextArea textArea2; // highlighted text is copied here
15    private JButton copyJButton; // initiates copying of text
16
```

Fig. 14.47 | Copying selected text from one `JTextArea` to another. (Part 1 of 3.)

JTextArea

```
17      // no-argument constructor
18      public TextAreaFrame()
19      {
20         super( "TextArea Demo" );
21         Box box = Box.createHorizontalBox(); // create box
22         String demo = "This is a demo string to\n" +
23            "illustrate copying text\nfrom one textarea to \n" +
24            "another textarea using an\nexternal event\n";
25
26         textArea1 = new JTextArea( demo, 10, 15 ); // create textArea1
27         box.add( new JScrollPane( textArea1 ) ); // add scrollpane
28
29         copyJButton = new JButton( "Copy >>>" ); // create copy button
30         box.add( copyJButton ); // add copy button to box
31         copyJButton.addActionListener(
32
```

Fig. 14.47 | Copying selected text from one `JTextArea` to another. (Part 2 of 3.)

JTextArea

```
33            new ActionListener() // anonymous inner class
34            {
35               // set text in textArea2 to selected text from textArea1
36               public void actionPerformed( ActionEvent event )
37               {
38                  textArea2.setText( textArea1.getSelectedText() );
39               } // end method actionPerformed
40            } // end anonymous inner class
41         ); // end call to addActionListener
42
43         textArea2 = new JTextArea( 10, 15 ); // create second textarea
44         textArea2.setEditable( false ); // disable editing
45         box.add( new JScrollPane( textArea2 ) ); // add scrollpane
46
47         add( box ); // add box to frame
48      } // end TextAreaFrame constructor
49   } // end class TextAreaFrame
```

Fig. 14.47 | Copying selected text from one `JTextArea` to another. (Part 3 of 3.)

JTextArea

```
1  // Fig. 14.48: TextAreaDemo.java
2  // Copying selected text from one textarea to another.
3  import javax.swing.JFrame;
4
5  public class TextAreaDemo
6  {
7     public static void main( String[] args )
8     {
9        TextAreaFrame textAreaFrame = new TextAreaFrame();
10       textAreaFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       textAreaFrame.setSize( 425, 200 ); // set frame size
12       textAreaFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class TextAreaDemo
```

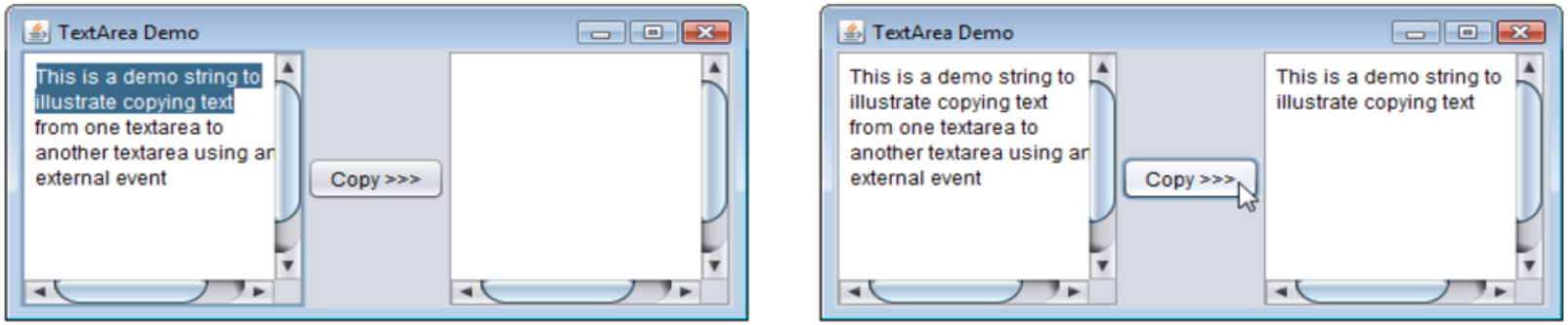

Fig. 14.48 | Test class for `TextAreaFrame`.