GUI Components: Part II

JList

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

```
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.)


```
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
```

Populate the `JList` with the `String`s in array `colorNames`.

Allow only single selections.

Provide scrollbars for the `JList` if necessary.

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

```
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() ] ); // Choose the appropriate Color to change the window's background color.
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.)


```
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`.

JList (cont.)

- setVisibleRowCount specifies the number of items visible in the list.
- setSelectionMode specifies the list's selection mode.
- Class ListSelectionModel (of package javax.swing) declares selection-mode constants
  - SINGLE_SELECTION (only one item to be selected at a time)
  - SINGLE_INTERVAL_SELECTION (allows selection of several contiguous items)
  - MULTIPLE_INTERVAL_SELECTION (does not restrict the items that can be selected).

JList (cont.)

- Unlike a JComboBox, a JList *does not* provide a scrollbar if there are more items in the list than the number of visible rows.
  - A JScrollPane object is used to provide the scrolling capability.
- addListSelectionListener registers a ListSelectionListener (package javax.swing.event) as the listener for a JList's selection events.

JList (cont.)

- Each JFrame actually consists of three layers—the background, the content pane and the glass pane.
- The content pane appears in front of the background and is where the GUI components in the JFrame are displayed.
- The glass pane is displays tool tips and other items that should appear in front of the GUI components on the screen.
- The content pane completely hides the background of the JFrame.
- To change the background color behind the GUI components, you must change the content pane's background color.
- Method getContentPane returns a reference to the JFrame's content pane (an object of class Container).
- List method getSelectedIndex returns the selected item's index.

Multiple-Selection Lists

- A multiple-selection list enables the user to select many items from a JList.
- A SINGLE_INTERVAL_SELECTION list allows selecting a contiguous range of items.
  - To do so, click the first item, then press and hold the *Shift* key while clicking the last item in the range.
- A MULTIPLE_INTERVAL_SELECTION list (the default) allows continuous range selection as described for a SINGLE_INTERVAL_SELECTION list and allows miscellaneous items to be selected by pressing and holding the *Ctrl* key while clicking each item to select.
  - To deselect an item, press and hold the *Ctrl* key while clicking the item a second time.

```
1  // Fig. 14.25: MultipleSelectionFrame.java
2  // Copying items from one List to another.
3  import java.awt.FlowLayout;
4  import java.awt.event.ActionListener;
5  import java.awt.event.ActionEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JList;
8  import javax.swing.JButton;
9  import javax.swing.JScrollPane;
10 import javax.swing.ListSelectionModel;
11
12 public class MultipleSelectionFrame extends JFrame
13 {
14    private JList colorJList; // list to hold color names
15    private JList copyJList; // list to copy color names into
16    private JButton copyJButton; // button to copy selected names
17    private static final String[] colorNames = { "Black", "Blue", "Cyan",
18       "Dark Gray", "Gray", "Green", "Light Gray", "Magenta", "Orange",
19       "Pink", "Red", "White", "Yellow" };
20
21    // MultipleSelectionFrame constructor
22    public MultipleSelectionFrame()
23    {
```

Fig. 14.25 | `JList` that allows multiple selections. (Part 1 of 3.)


```
24       super( "Multiple Selection Lists" );
25       setLayout( new FlowLayout() ); // set frame layout
26
27       colorJList = new JList( colorNames ); // holds names of all colors
28       colorJList.setVisibleRowCount( 5 ); // show five rows
29       colorJList.setSelectionMode(
30          ListSelectionModel.MULTIPLE_INTERVAL_SELECTION );
31       add( new JScrollPane( colorJList ) ); // add list with scrollpane
32
33       copyJButton = new JButton( "Copy >>>" ); // create copy button
34       copyJButton.addActionListener(
35
36          new ActionListener() // anonymous inner class
37          {
38             // handle button event
39             public void actionPerformed( ActionEvent event )
40             {
41                // place selected values in copyJList
42                copyJList.setListData( colorJList.getSelectedValues() );
43             } // end method actionPerformed
44          } // end anonymous inner class
45       ); // end call to addActionListener
46
47       add( copyJButton ); // add copy button to JFrame
```

Allow multiple selections or intervals in the `JList`.

Fig. 14.25 | `JList` that allows multiple selections. (Part 2 of 3.)

```
48
49          copyJList = new JList(); // create list to hold copied color names
50          copyJList.setVisibleRowCount( 5 ); // show 5 rows
51          copyJList.setFixedCellWidth( 100 ); // set width
52          copyJList.setFixedCellHeight( 15 ); // set height
53          copyJList.setSelectionMode(
54             ListSelectionModel.SINGLE_INTERVAL_SELECTION );
55          add( new JScrollPane( copyJList ) ); // add list with scrollpane
56       } // end MultipleSelectionFrame constructor
57    } // end class MultipleSelectionFrame
```

Allow only single intervals (ranges) to be selected.

Fig. 14.25 | `JList` that allows multiple selections. (Part 3 of 3.)


```
1  // Fig. 14.26: MultipleSelectionTest.java
2  // Testing MultipleSelectionFrame.
3  import javax.swing.JFrame;
4
5  public class MultipleSelectionTest
6  {
7     public static void main( String[] args )
8     {
9        MultipleSelectionFrame multipleSelectionFrame =
10          new MultipleSelectionFrame();
11       multipleSelectionFrame.setDefaultCloseOperation(
12          JFrame.EXIT_ON_CLOSE );
13       multipleSelectionFrame.setSize( 350, 150 ); // set frame size
14       multipleSelectionFrame.setVisible( true ); // display frame
15    } // end main
16 } // end class MultipleSelectionTest
```

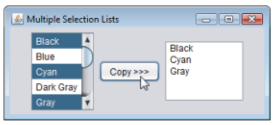

Fig. 14.26 | Test class for `MultipleSelectionFrame`.

Multiple-Selection Lists (cont.)

- If a JList does not contain items it will not diplay in a FlowLayout.
  - use JList methods setFixedCellWidth and setFixedCellHeight to set the item width and height
- There are no events to indicate that a user has made multiple selections in a multiple-selection list.
  - An event generated by another GUI component (known as an external event) specifies when the multiple selections in a JList should be processed.
- Method setListData sets the items displayed in a JList.
- Method getSelectedValues returns an array of Objects representing the selected items in a JList.

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.

`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.)

`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 (cont.)

- Each mouse event-handling method receives a MouseEvent object that contains information about the mouse event that occurred, including the *x*- and y-coordinates of the location where the event occurred.
- Coordinates are measured from the upper-left corner of the GUI component on which the event occurred.
- The x-coordinates start at 0 and increase from left to right. The *y*-coordinates start at 0 and increase from top to bottom.
- The methods and constants of class InputEvent (Mouse-Event's superclass) enable you to determine which mouse button the user clicked.

Software Engineering Observation 14.7

Method calls to `mouseDragged` *are sent to the* `MouseMotionListener` *for the* `Component` *on which a mouse drag operation started. Similarly, the* `mouseReleased` *method call at the end of a drag operation is sent to the* `MouseListener` *for the* `Component` *on which the drag operation started.*

Mouse Event Handling (cont.)

- Interface MouseWheelListener enables applications to respond to the rotation of a mouse wheel.
- Method mouseWheelMoved receives a MouseWheelEvent as its argument.
- Class MouseWheelEvent (a subclass of MouseEvent) contains methods that enable the event handler to obtain information about the amount of wheel rotation.


```
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
17    // MouseTrackerFrame constructor sets up GUI and
18    // registers mouse event handlers
19    public MouseTrackerFrame()
20    {
21       super( "Demonstrating Mouse Events" );
22
```

Fig. 14.28 | Mouse event handling. (Part 1 of 4.)

```
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
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
```

Object that handles both mouse events and mouse motion events.

An object of this class *is a* `MouseListener` and *is a* `MouseMotionListener`

Get the mouse coordinates at the time the click event occurred.

Fig. 14.28 | Mouse event handling. (Part 2 of 4.)


```
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
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
```

Get the mouse coordinates at the time the pressed event occurred.

Get the mouse coordinates at the time the released event occurred.

Get the mouse coordinates at the time the entered event occurred then change the background to green.

Fig. 14.28 | Mouse event handling. (Part 3 of 4.)

```
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
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
```

Change the background to white when the mouse exits the area.

Get the mouse coordinates at the time the dragged event occurred.

Get the mouse coordinates at the time the moved event occurred.

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


```
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.)

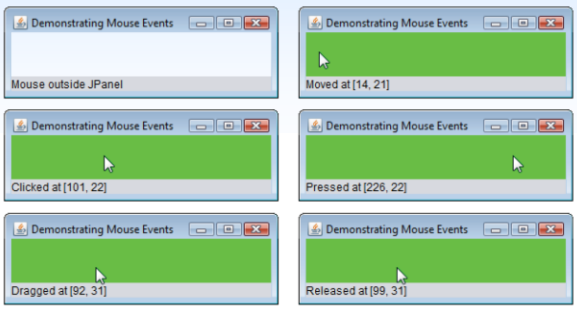

Fig. 14.29 | Test class for `MouseTrackerFrame`. (Part 2 of 2.)

Mouse Event Handling (cont.)

- BorderLayout arranges components into five regions: NORTH, SOUTH, EAST, WEST and CENTER.
- BorderLayout sizes the component in the CENTER to use all available space that is not occupied
- Methods addMouseListener and addMouseMotionListener register MouseListeners and MouseMotionListeners, respectively.
- MouseEvent methods getX and getY return the *x*- and *y*-coordinates of the mouse at the time the event occurred.

Adapter Classes

- Many event-listener interfaces contain multiple methods.
- An adapter class implements an interface and provides a default implementation (with an empty method body) of each method in the interface.
- You extend an adapter class to inherit the default implementation of every method and override only the method(s) you need for event handling.

Software Engineering Observation 14.8

When a class implements an interface, the class has an is-a *relationship with that interface. All direct and indirect subclasses of that class inherit this interface. Thus, an object of a class that extends an event-adapter class* is an *object of the corresponding event-listener type (e.g., an object of a subclass of* `MouseAdapter` is a `MouseListener`*).*

Event-adapter class in `java.awt.event`	Implements interface
`ComponentAdapter`	`ComponentListener`
`ContainerAdapter`	`ContainerListener`
`FocusAdapter`	`FocusListener`
`KeyAdapter`	`KeyListener`
`MouseAdapter`	`MouseListener`
`MouseMotionAdapter`	`MouseMotionListener`
`WindowAdapter`	`WindowListener`

Fig. 14.30 | Event-adapter classes and the interfaces they implement in package `java.awt.event`.


```
1  // Fig. 14.31: MouseDetailsFrame.java
2  // Demonstrating mouse clicks and distinguishing between mouse buttons.
3  import java.awt.BorderLayout;
4  import java.awt.event.MouseAdapter;
5  import java.awt.event.MouseEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JLabel;
8
9  public class MouseDetailsFrame extends JFrame
10 {
11    private String details; // String that is displayed in the statusBar
12    private JLabel statusBar; // JLabel that appears at bottom of window
13
14    // constructor sets title bar String and register mouse listener
15    public MouseDetailsFrame()
16    {
17       super( "Mouse clicks and buttons" );
18
19       statusBar = new JLabel( "Click the mouse" );
20       add( statusBar, BorderLayout.SOUTH );
21       addMouseListener( new MouseClickHandler() ); // add handler
22    } // end MouseDetailsFrame constructor
23
```

Fig. 14.31 | Left, center and right mouse-button clicks. (Part 1 of 2.)

```
24     // inner class to handle mouse events
25     private class MouseClickHandler extends MouseAdapter
26     {
27        // handle mouse-click event and determine which button was pressed
28        public void mouseClicked( MouseEvent event )
29        {
30           int xPos = event.getX(); // get x-position of mouse
31           int yPos = event.getY(); // get y-position of mouse
32
33           details = String.format( "Clicked %d time(s)",
34              event.getClickCount() );
35
36           if ( event.isMetaDown() ) // right mouse button
37              details += " with right mouse button";
38           else if ( event.isAltDown() ) // middle mouse button
39              details += " with center mouse button";
40           else // left mouse button
41              details += " with left mouse button";
42
43           statusBar.setText( details ); // display message in statusBar
44        } // end method mouseClicked
45     } // end private inner class MouseClickHandler
46  } // end class MouseDetailsFrame
```

Adapter enables us to override the one method we use in this example.

Returns the number of mouse clicks. If you wait long enouogh between clicks, the count resets to 0.

Help determine which button the user pressed on the mouse.

Fig. 14.31 | Left, center and right mouse-button clicks. (Part 2 of 2.)


```
1  // Fig. 14.32: MouseDetails.java
2  // Testing MouseDetailsFrame.
3  import javax.swing.JFrame;
4
5  public class MouseDetails
6  {
7     public static void main( String[] args )
8     {
9        MouseDetailsFrame mouseDetailsFrame = new MouseDetailsFrame();
10       mouseDetailsFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       mouseDetailsFrame.setSize( 400, 150 ); // set frame size
12       mouseDetailsFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class MouseDetails
```

Fig. 14.32 | Test class for `MouseDetailsFrame`. (Part 1 of 2.)

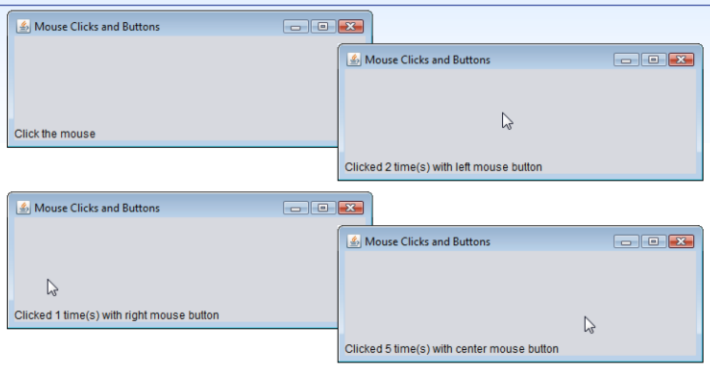

Fig. 14.32 | Test class for `MouseDetailsFrame`. (Part 2 of 2.)

Adapter Classes (cont.)

- A mouse can have one, two or three buttons.
- Class MouseEvent inherits several methods from InputEvent that can distinguish among mouse buttons or mimic a multibutton mouse with a combined keystroke and mouse-button click.
- Java assumes that every mouse contains a left mouse button.

Adapter Classes (cont.)

- In the case of a one- or two-button mouse, a Java application assumes that the center mouse button is clicked if the user holds down the *Alt* key and clicks the left mouse button on a two-button mouse or the only mouse button on a one-button mouse.
- In the case of a one-button mouse, a Java application assumes that the right mouse button is clicked if the user holds down the *Meta* key (sometimes called the Command key or the "Apple" key on a Mac) and clicks the mouse button.

InputEvent method	Description
`isMetaDown()`	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 hold down the *Meta* key on the keyboard and click the mouse button.
`isAltDown()`	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 and click the only or left mouse button, respectively.

Fig. 14.33 | `InputEvent` methods that help distinguish among left-, center- and right-mouse-button clicks.

Adapter Classes (cont.)

- The number of consecutive mouse clicks is returned by MouseEvent method getClickCount.
- Methods isMetaDown and isAltDown determine which mouse button the user clicked.

JPanel Subclass for Drawing with the Mouse

- Use a JPanel as a dedicated drawing area in which the user can draw by dragging the mouse.
- Lightweight Swing components that extend class JComponent (such as JPanel) contain method paintComponent
  - called when a lightweight Swing component is displayed
- Override this method to specify how to draw.
  - Call the superclass version of paintComponent as the first statement in the body of the overridden method to ensure that the component displays correctly.

JPanel Subclass for Drawing with the Mouse (cont.)

- JComponent support transparency.
  - To display a component correctly, the program must determine whether the component is transparent.
  - The code that determines this is in superclass JComponent's paintComponent implementation.
  - When a component is transparent, paintComponent will not clear its background
  - When a component is opaque, paintComponent clears the component's background
  - The transparency of a Swing lightweight component can be set with method setOpaque (a false argument indicates that the component is transparent).

Look-and-Feel Observation 14.12

Most Swing GUI components can be transparent or opaque. If a Swing GUI component is opaque, its background will be cleared when its `paintComponent` *method is called. Only opaque components can display a customized background color.* `JPanel` *objects are opaque by default.*

Error-Prevention Tip 14.1

In a `JComponent` *subclass's* `paintComponent` *method, the first statement should always call to the superclass's* `paintComponent` *method to ensure that an object of the subclass displays correctly.*

Common Programming Error 14.5

If an overridden `paintComponent` *method does not call the superclass's version, the subclass component may not display properly. If an overridden* `paintComponent` *method calls the superclass's version after other drawing is performed, the drawing will be erased.*

```
1  // Fig. 14.34: PaintPanel.java
2  // Using class MouseMotionAdapter.
3  import java.awt.Point;
4  import java.awt.Graphics;
5  import java.awt.event.MouseEvent;
6  import java.awt.event.MouseMotionAdapter;
7  import javax.swing.JPanel;
8
9  public class PaintPanel extends JPanel
10 {
11    private int pointCount = 0; // count number of points
12
13    // array of 10000 java.awt.Point references
14    private Point[] points = new Point[ 10000 ];
15
```

Fig. 14.34 | Adapter class used to implement event handlers. (Part 1 of 3.)


```
16    // set up GUI and register mouse event handler
17    public PaintPanel()
18    {
19       // handle frame mouse motion event
20       addMouseMotionListener(
21
22          new MouseMotionAdapter() // anonymous inner class
23          {
24             // store drag coordinates and repaint
25             public void mouseDragged( MouseEvent event )
26             {
27                if ( pointCount < points.length )
28                {
29                   points[ pointCount ] = event.getPoint(); // find point
30                   pointCount++; // increment number of points in array
31                   repaint(); // repaint JFrame
32                } // end if
33             } // end method mouseDragged
34          } // end anonymous inner class
35       ); // end call to addMouseMotionListener
36    } // end PaintPanel constructor
37
```

Store points as user drags the mouse.

Request that this `PaintPanel` be repainted. Causes a call to `paintComponent`.

Fig. 14.34 | Adapter class used to implement event handlers. (Part 2 of 3.)

```
38     // draw ovals in a 4-by-4 bounding box at specified locations on window
39     public void paintComponent( Graphics g )
40     {
41        super.paintComponent( g ); // clears drawing area
42
43        // draw all points in array
44        for ( int i = 0; i < pointCount; i++ )
45           g.fillOval( points[ i ].x, points[ i ].y, 4, 4 );
46     } // end method paintComponent
47  } // end class PaintPanel
```

Draws a filled circle at the specified coordinates.

Fig. 14.34 | Adapter class used to implement event handlers. (Part 3 of 3.)

JPanel Subclass for Drawing with the Mouse (cont.)

- Class Point (package java.awt) represents an *x*-*y* coordinate.
  - We use objects of this class to store the coordinates of each mouse drag event.
- Class Graphics is used to draw.
- MouseEvent method getPoint obtains the Point where the event occurred.
- Method repaint (inherited from Component) indicates that a Component should be refreshed on the screen as soon as possible.

Look-and-Feel Observation 14.13

Calling `repaint` *for a Swing GUI component indicates that the component should be refreshed on the screen as soon as possible. The background of the GUI component is cleared only if the component is opaque.* `JComponent` *method* `setOpaque` *can be passed a* `boolean` *argument indicating whether the component is opaque (*`true`*) or transparent (*`false`*).*

JPanel Subclass for Drawing with the Mouse (cont.)

- Graphics method fillOval draws a solid oval.
  - Four parameters represent a rectangular area (called the bounding box) in which the oval is displayed.
  - The first two are the upper-left x-coordinate and the upper-left *y*-coordinate of the rectangular area.
  - The last two represent the rectangular area's width and height.
- Method fillOval draws the oval so it touches the middle of each side of the rectangular area.

Look-and-Feel Observation 14.14

Drawing on any GUI component is performed with coordinates that are measured from the upper-left corner (0, 0) of that GUI component, not the upper-left corner of the screen.


```
1  // Fig. 14.35: Painter.java
2  // Testing PaintPanel.
3  import java.awt.BorderLayout;
4  import javax.swing.JFrame;
5  import javax.swing.JLabel;
6
7  public class Painter
8  {
9     public static void main( String[] args )
10    {
11       // create JFrame
12       JFrame application = new JFrame( "A simple paint program" );
13
14       PaintPanel paintPanel = new PaintPanel(); // create paint panel
15       application.add( paintPanel, BorderLayout.CENTER ); // in center
16
17       // create a label and place it in SOUTH of BorderLayout
18       application.add( new JLabel( "Drag the mouse to draw" ),
19          BorderLayout.SOUTH );
20
21       application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
22       application.setSize( 400, 200 ); // set frame size
23       application.setVisible( true ); // display frame
24    } // end main
25 } // end class Painter
```

Creates the dedicated drawing area.

Attaches the dedicated drawing area to the center of the window.

Fig. 14.35 | Test class for `PaintFrame`. (Part 1 of 2.)

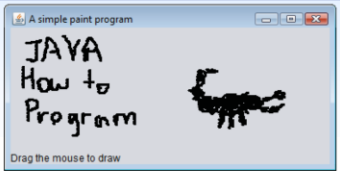

Fig. 14.35 | Test class for `PaintFrame`. (Part 2 of 2.)

Key Event Handling

- KeyListener interface for handling key events.
- Key events are generated when keys on the keyboard are pressed and released.
- A KeyListener must define methods keyPressed, keyReleased and keyTyped
  - each receives a KeyEvent as its argument
- Class KeyEvent is a subclass of InputEvent.
- Method keyPressed is called in response to pressing any key.
- Method keyTyped is called in response to pressing any key that is not an action key.
- Method keyReleased is called when the key is released after any keyPressed or keyTyped event.

```
1  // Fig. 14.36: KeyDemoFrame.java
2  // Demonstrating keystroke events.
3  import java.awt.Color;
4  import java.awt.event.KeyListener;
5  import java.awt.event.KeyEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JTextArea;
8
9  public class KeyDemoFrame extends JFrame implements KeyListener
10 {
11    private String line1 = ""; // first line of textarea
12    private String line2 = ""; // second line of textarea
13    private String line3 = ""; // third line of textarea
14    private JTextArea textArea; // textarea to display output
15
16    // KeyDemoFrame constructor
17    public KeyDemoFrame()
18    {
19       super( "Demonstrating Keystroke Events" );
20
21       textArea = new JTextArea( 10, 15 ); // set up JTextArea
22       textArea.setText( "Press any key on the keyboard..." );
23       textArea.setEnabled( false ); // disable textarea
```

This class can handle its own KeyEvents.

Fig. 14.36 | Key event handling. (Part 1 of 3.)


```
24       textArea.setDisabledTextColor( Color.BLACK ); // set text color
25       add( textArea ); // add textarea to JFrame
26
27       addKeyListener( this ); // allow frame to process key events
28    } // end KeyDemoFrame constructor
29
30    // handle press of any key
31    public void keyPressed( KeyEvent event )
32    {
33       line1 = String.format( "Key pressed: %s",
34          KeyEvent.getKeyText( event.getKeyCode() ) ); // show pressed key
35       setLines2and3( event ); // set output lines two and three
36    } // end method keyPressed
37
38    // handle release of any key
39    public void keyReleased( KeyEvent event )
40    {
41       line1 = String.format( "Key released: %s",
42          KeyEvent.getKeyText( event.getKeyCode() ) ); // show released key
43       setLines2and3( event ); // set output lines two and three
44    } // end method keyReleased
45
```

Registers the object of this class as the event handler.

Gets text of pressed key.

Gets text of pressed key.

Fig. 14.36 | Key event handling. (Part 2 of 3.)

```
46     // handle press of an action key
47     public void keyTyped( KeyEvent event )
48     {
49        line1 = String.format( "Key typed: %s", event.getKeyChar() );
50        setLines2and3( event ); // set output lines two and three
51     } // end method keyTyped
52
53     // set second and third lines of output
54     private void setLines2and3( KeyEvent event )
55     {
56        line2 = String.format( "This key is %san action key",
57           ( event.isActionKey() ? "" : "not " ) );
58
59        String temp = KeyEvent.getKeyModifiersText( event.getModifiers() );
60
61        line3 = String.format( "Modifier keys pressed: %s",
62           ( temp.equals( "" ) ? "none" : temp ) ); // output modifiers
63
64        textArea.setText( String.format( "%s\n%s\n%s\n",
65           line1, line2, line3 ) ); // output three lines of text
66     } // end method setLines2and3
67  } // end class KeyDemoFrame
```

Gets text of pressed modifier keys.

Fig. 14.36 | Key event handling. (Part 3 of 3.)


```
1  // Fig. 14.37: KeyDemo.java
2  // Testing KeyDemoFrame.
3  import javax.swing.JFrame;
4
5  public class KeyDemo
6  {
7     public static void main( String[] args )
8     {
9        KeyDemoFrame keyDemoFrame = new KeyDemoFrame();
10       keyDemoFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       keyDemoFrame.setSize( 350, 100 ); // set frame size
12       keyDemoFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class KeyDemo
```

Fig. 14.37 | Test class for `KeyDemoFrame`. (Part 1 of 2.)

Fig. 14.37 | Test class for `KeyDemoFrame`. (Part 2 of 2.)

Key Event Handling (cont.)

- Registers key event handlers with method addKeyListener from class Component.
- KeyEvent method getKeyCode gets the virtual key code of the pressed key.
- KeyEvent contains virtual key-code constants that represents every key on the keyboard.
- Value returned by getKeyCode can be passed to static KeyEvent method getKeyText to get a string containing the name of the key that was pressed.
- KeyEvent method getKeyChar (which returns a char) gets the Unicode value of the character typed.
- KeyEvent method isActionKey determines whether the key in the event was an action key.

Key Event Handling (cont.)

- Method getModifiers determines whether any modifier keys (such as *Shift*, *Alt* and *Ctrl*) were pressed when the key event occurred.
  - Result can be passed to static KeyEvent method getKeyModifiersText to get a string containing the names of the pressed modifier keys.
- InputEvent methods isAltDown, isControlDown, isMetaDown and isShiftDown each return a boolean indicating whether the particular key was pressed during the key event.

Introduction to 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.

Introduction to Layout Managers (cont.)

- 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 (cont.)

  - 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.

Look-and-Feel Observation 14.16

It's possible to set a `Container`*'s layout to* `null`*, which indicates that no layout manager should be used. In a* `Container` *without a layout manager, you must position and size the components in the given container and take care that, on resize events, all components are repositioned as necessary. A component's resize events can be processed by a* `ComponentListener`*.*

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.

Look-and-Feel Observation 14.17

Each individual container can have only one layout manager, but multiple containers in the same application can each use different layout managers.

```
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
18    // set up GUI and register button listeners
19    public FlowLayoutFrame()
20    {
21       super( "FlowLayout Demo" );
22
```

Fig. 14.39 | `FlowLayout` allows components to flow over multiple lines. (Part 1 of 4.)


```
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
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
```

Left aligns the components.

Lays out the container's components again based on the layout changes.

Fig. 14.39 | `FlowLayout` allows components to flow over multiple lines. (Part 2 of 4.)

```
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
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
63         // set up rightJButton and register listener
64         rightJButton = new JButton( "Right" ); // create Right button
65         add( rightJButton ); // add Right button to frame
```

Center aligns the components.

Lays out the container's components again based on the layout changes.

Fig. 14.39 | `FlowLayout` allows components to flow over multiple lines. (Part 3 of 4.)


```
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
```

Right aligns the components.

Lays out the container's components again based on the layout changes.

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

```
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.)

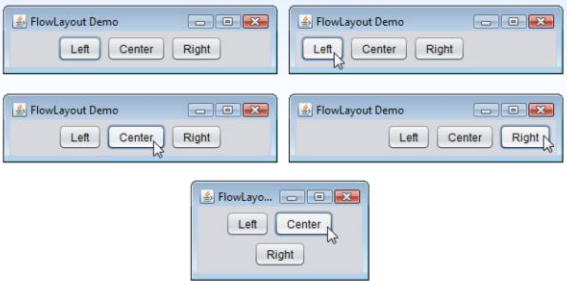

Fig. 14.40 | Test class for `FlowLayoutFrame`. (Part 2 of 2.)

FlowLayout (cont.)

- FlowLayout method setAlignment changes the alignment for the FlowLayout.
  - FlowLayout.LEFT
  - FlowLayout.CENTER
  - FlowLayout.RIGHT
- LayoutManager interface method layoutContainer (which is inherited by all layout managers) specifies that a container should be rearranged based on the adjusted layout.

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 LayoutManager (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.

```
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
```

Custom `BorderLayout` with horizontal and vertical gap space.

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


```
24
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
```

This `BorderLayoutFrame` handles each `JButton`'s `ActionEvent`.

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

```
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
```

Hides the button.

Shows the button.

Re-lays out the container.

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


```
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.)

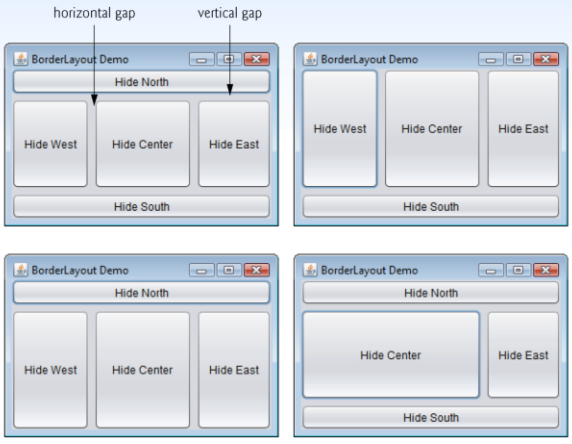

Fig. 14.42 | Test class for `BorderLayoutFrame`. (Part 2 of 3.)

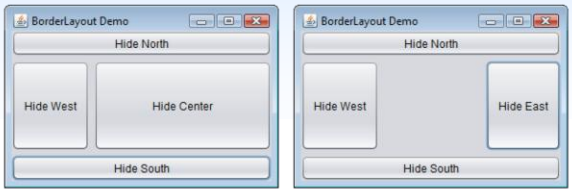

Fig. 14.42 | Test class for `BorderLayoutFrame`. (Part 3 of 3.)

BorderLayout (cont.)

- BorderLayout constructor arguments specify the number of pixels between components that are arranged horizontally (horizontal gap space) and between components that are arranged vertically (vertical gap space), respectively.
  - The default is one pixel of gap space horizontally and vertically.

Look-and-Feel Observation 14.18

If no region is specified when adding a `Component` *to a* `BorderLayout`*, the layout manager assumes that the* `Component` *should be added to region* `BorderLayout.CENTER`.

Common Programming Error 14.6

When more than one component is added to a region in a `BorderLayout`*, only the last component added to that region will be displayed. There is no error that indicates this problem.*

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.

```
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.)


```
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
```

Custom `GridLayout`s: One with 2 rows, 3 columns and 5 pixels of gap space between components and the other with 3 rows, two columns and the default gap space.

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

```
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
```

Changes the layout

Re-lays out the container.

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


```
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.


```
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.)

```
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
```

Places the `JPanel` with 5 buttons in the South region of the window's `BorderLayout`.

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


```
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`.

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.


```
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
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";
```

Container that arranges components horizontally.

Fig. 14.47 | Copying selected text from one `JTextArea` to another. (Part 1 of 2.)

```
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
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
```

Copies selected text into `textArea2`.

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


```
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
```

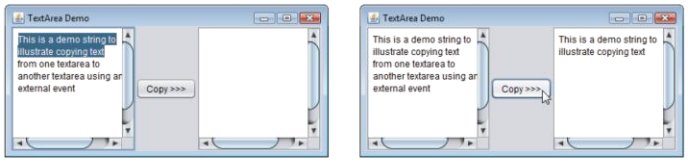

Fig. 14.48 | Test class for `TextAreaFrame`.

JTextArea

- A JTextArea provides an area for manipulating multiple lines of text.
- JTextArea is a subclass of JTextComponent.

JTextArea (cont.)

- Box is a subclass of Container that uses a BoxLayout to arrange the GUI components horizontally or vertically.
- Box static method createHorizontalBox creates a Box that arranges components left to right in the order that they are attached.
- JTextArea method getSelectedText (inherited from JTextComponent) returns the selected text from a JTextArea.
- JTextArea method setText changes the text in a JTextArea.
- When text reaches the right edge of a JTextArea the text can wrap to the next line.
  - Referred to as line wrapping.
  - By default, JTextArea does not wrap lines.

JTextArea (cont.)

- You can set the horizontal and vertical scrollbar policies of a JScrollPane when it's constructed.
- You can also use JScrollPane methods setHorizontalScrollBarPolicy and setVerticalScrollBarPolicy to change the scrollbar policies.

JTextArea (cont.)

- Class JScrollPane declares the constants
  - JScrollPane.VERTICAL_SCROLLBAR_ALWAYS
    JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS
  - to indicate that a scrollbar should always appear, constants
    - JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED
      JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED
  - to indicate that a scrollbar should appear only if necessary (the defaults) and constants
    - JScrollPane.VERTICAL_SCROLLBAR_NEVER
      JScrollPane.HORIZONTAL_SCROLLBAR_NEVER
  - to indicate that a scrollbar should never appear.
- If policy is set to HORIZONTAL_SCROLLBAR_NEVER, a JTextArea attached to the JScrollPane will automatically wrap lines.