Object Oriented Programming

10636212

Dr. Ashraf Armoush

Chapter 07

GUI Components
(Part 1)

GUI Components

- A **GUI component** is an object with which the user interacts via the mouse, the keyboard or another form of input.
- GUI components that enable application developers to create robust GUIs.
- Some basic GUI components:
  - JLabel
  - JTextField
  - Jbutton
  - JList
  - JCheckBox
  - JComboBox
  - JTextArea
  - JPanel

Swing Toolkit

- The Swing toolkit includes a rich set of components for building GUIs and adding interactivity to Java applications.

- Most Swing components are **pure Java components**
- **They are written, manipulated and displayed completely in Java.**
- They are part of the **Java Foundation Classes (JFC)**
- **Which Swing Packages Should You Use?**
  - The Swing API has 18 public packages
  - Fortunately, most programs use only a small subset of the API.

    `javax.swing`

    `javax.swing.event` (not always required)

Swing vs. AWT

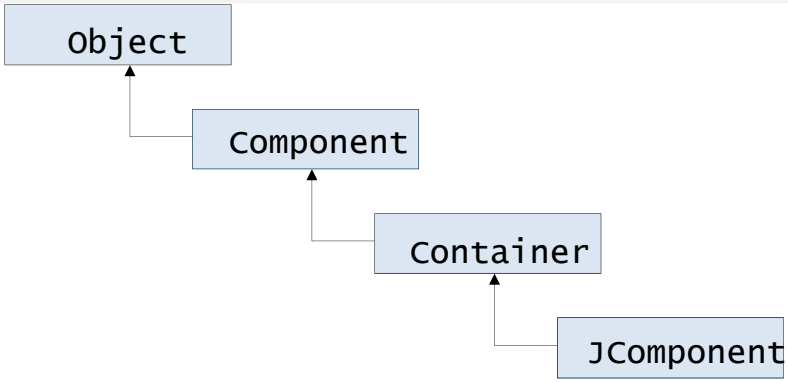
- There are two sets of GUI components in Java.
  - **Abstract Window Toolkit (`java.awt`)**
  - **Swing Toolkit (`javax.swing`)**

- AWT
  - When a Java application with an **AWT** GUI executes on different Java platforms, the application's GUI components display differently on each platform.
  - ***Heavyweight components*****,** because they rely on the local platform's (e.g. windowing system) to determine their functionality and their look-and-feel.

Swing vs. AWT

- Swing
  - ***Lightweight components:***
  - Most Swing components are not tied to actual GUI components supported by the underlying platform.
  - Pure Java design.
  - Swing components are generally slower than AWT.
  - Swing components support a pluggable look and feel that allow you to specify a uniform look-and-feel for your application across all platforms or to use each platform's custom look-and-feel.

JComponent Class



- **JComponent** is superclass of all lightweight Swing components and declares their common attributes and behaviors.
- Supports some common lightweight features:
  - A pluggable look-and-feel.
  - Shortcut keys (called mnemonics)
  - Common event-handling capabilities.
  - Brief descriptions (called tool tips)
  - Support for user-interface localization (interface to display in different languages)

Displaying Text and Images

- In a large GUI
  - Difficult to identify the purpose of every component.
  - Provide text stating each component's purpose.
- Such text is known as a **label**
  - Provide text on GUI
  - Defined with class `Jlabel` (a subclass of `JComponent`)
  - Can display:
    - Single line of read-only text
    - Image
    - Text and image
- Tool Tips:
  - Use tool tips to add descriptive text to your GUI components
  - This text helps user determine the GUI component's purpose.
  - Method `setToolTipText` (inherited by **JLabel** from **JComponent**) specifies the **tool tip** that is displayed when the user positions the mouse cursor over a **JComponent** (such as a **JLabel**).

FlowLayout

- `FlowLayout`
  - GUI components are placed on a container from left to right in the order in which the program attaches them to the container.
  - When there is no more room to fit components left to right, components continue to display left to right on the next line.
  - If the container is resized, a `FlowLayout` reflows the components to accommodate the new width of the container, possibly with fewer or more rows of GUI components.

- Method `setLayout` is inherited from class Container.
  - argument must be an object of a class that implements the LayoutManager interface (e.g., FlowLayout).

JLabel Demo

```
import java.awt.FlowLayout; // specifies how components are arranged
import javax.swing.JFrame; // provides basic window features
import javax.swing.JLabel; // displays text and images
import javax.swing.SwingConstants; // common constants used with Swing
import javax.swing.Icon; // interface used to manipulate images
import javax.swing.ImageIcon; // loads images

public class LabelFrame extends JFrame
{
   private JLabel label1; // JLabel with just text
   private JLabel label2; // JLabel constructed with text and icon
   private JLabel label3; // JLabel with added text and icon

   // LabelFrame constructor adds JLabels to JFrame
   public LabelFrame()
   {
      super( "Testing JLabel" );
      setLayout( new FlowLayout() ); // set frame layout
```

Custom GUIs are often built in classes that extend `JFrame`

JLabel Demo (cont.)

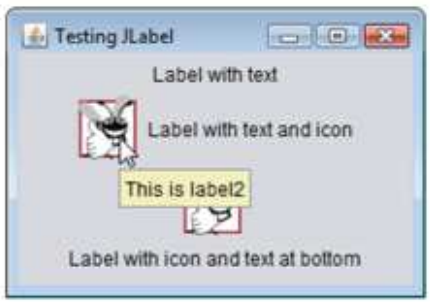
```
      // JLabel constructor with a string argument
      label1 = new JLabel( "Label with text" );
      label1.setToolTipText( "This is label1" );
      add( label1 ); // add label1 to JFrame

      // JLabel constructor with string, Icon and alignment arguments
      Icon bug = new ImageIcon( getClass().getResource(  "bug1.png" ) );
      label2 = new JLabel( "Label with text and icon", bug,
         SwingConstants.LEFT );
      label2.setToolTipText( "This is label2" );
      add( label2 ); // add label2 to JFrame

      label3 = new JLabel(); // JLabel constructor no arguments
      label3.setText( "Label with icon and text at bottom" );
      label3.setIcon( bug ); // add icon to JLabel
      label3.setHorizontalTextPosition( SwingConstants.CENTER );
      label3.setVerticalTextPosition( SwingConstants.BOTTOM );
      label3.setToolTipText( "This is label3" );
      add( label3 ); // add label3 to JFrame
   } // end LabelFrame constructor
} // end class LabelFrame
```

Create a `JLabel` with the specified text then set its tooltip

Create an empty `JLabel` then use set methods to change its characteristics.

JLabel Demo (cont.)

```
import javax.swing.JFrame;

public class LabelTest
{
   public static void main( String[] args )
   {
      LabelFrame labelFrame = new LabelFrame(); // create LabelFrame
      labelFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      labelFrame.setSize( 260, 180 ); // set frame size
      labelFrame.setVisible( true ); // display frame
   } // end main
} // end class LabelTest
```



Icon

- Icons enhance the look-and-feel of an application and are also commonly used to indicate functionality.
- An icon is normally specified with an `Icon` argument to a constructor or to the component's `setIcon` method.
- An Icon is an object of any class that implements interface Icon (package javax.swing).
- `ImageIcon` (package javax.swing) supports several image formats, including:
  - Graphics Interchange Format (GIF)
  - Portable Network Graphics (PNG)
  - Joint Photographic Experts Group (JPEG).

JLabel (cont.)

- Class `JLabel` provides methods to change a label's appearance after it has been instantiated.
  - Method `setText` sets the text displayed on the label.
  - Method `getText` retrieves the current text displayed on a label.
  - Method `setIcon` specifies the Icon to display on a label.
  - Method `getIcon` retrieves the current Icon displayed on a label.
  - Methods `setHorizontalTextPosition` and `setVerticalTextPosition` specify the text position in the label.

Swing Constants

- Interface `SwingConstants` (package javax.swing) declares a set of common integer constants (such as `SwingConstants.LEFT`) that are used with many Swing components.
- By default, the text appears to the right of the image when a label contains both text and an image.

Constant	Description
Horizontal-position constants	
`SwingConstants.LEFT`	Place text on the left.
`SwingConstants.CENTER`	Place text in the center.
`SwingConstants.RIGHT`	Place text on the right.
Vertical-position constants	
`SwingConstants.TOP`	Place text at the top.
`SwingConstants.CENTER`	Place text in the center.
`SwingConstants.BOTTOM`	Place text at the bottom.

- The horizontal and vertical alignments of a JLabel can be set with methods respectively:
  - `setHorizontalAlignment`
  - `setVerticalAlignment`

Text Fields and Event Handling

- GUIs are event driven.
- When the user interacts with a GUI component, the interaction—known as an event—drives the program to perform a task.
- The code that performs a task in response to an event is called an event handler
- The overall process of responding to events is known as event handling.

JTextField and JPasswordField

- JTextField: a single line area in which user can enter text.
  - Main Methods:
    - setText
    - getText
    - setEditable

- JPasswordField
  - Class JPasswordField extends JTextField and adds methods that are specific to processing passwords.
  - JPasswordField shows that characters are being typed as the user enters them, but hides the actual characters with an echo character.
  - getPassword method returns the password as an array of characters

Nested Class

- All the classes discussed so far were so-called top-level classes—that is, they were not declared inside another class.
- Java allows you to declare classes inside other classes—these are called nested classes.
  - Can be static or non-static.
  - Non-static nested classes are called inner classes and are frequently used to implement event handlers.

Nested Class (cont.)

- Before an object of an inner class can be created, there must first be an object of the top-level class that contains the inner class.
- This is required because an inner-class object implicitly has a reference to an object of its top-level class.
- There is also a special relationship between these objects—the inner-class object is allowed to directly access all the variables and methods of the outer class.
- A nested class that is static does not require an object of its top-level class and does not implicitly have a reference to an object of the top-level class.
- Inner classes can be declared public, protected or private.
- Since event handlers tend to be specific to the application in which they are defined, they are often implemented as private inner classes.

Event Handling

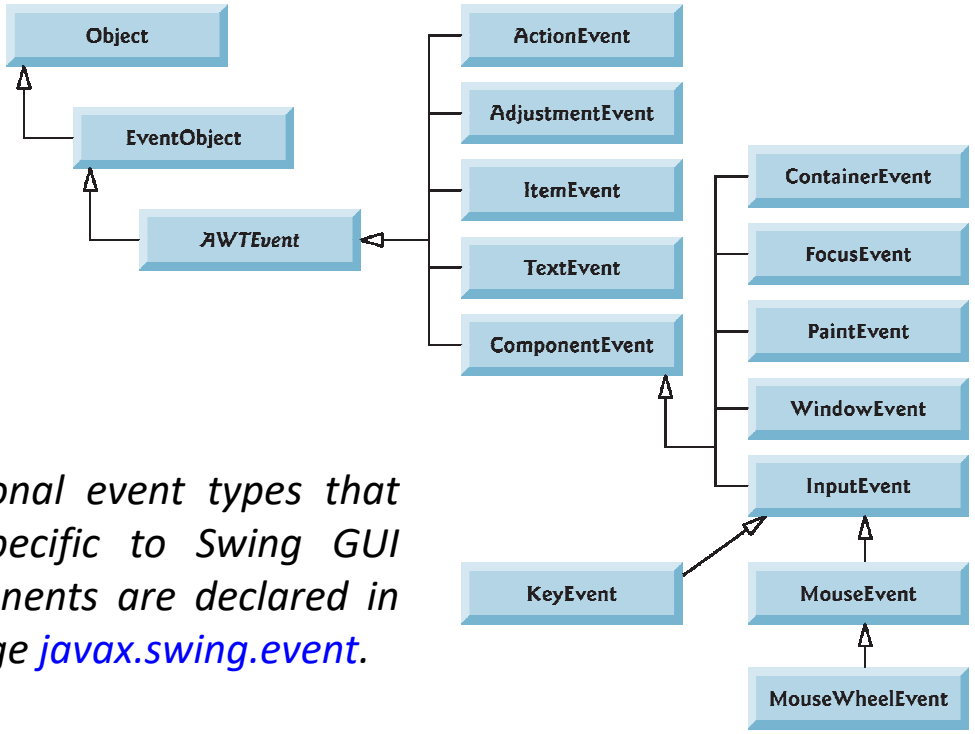
- Before an application can respond to an event for a particular GUI component, you must perform several coding steps:
    1. Create a class that represents the event handler.
    2. Implement an appropriate interface, known as an event-listener interface, in the class from *Step 1.*
    3. Indicate that an object of the class from Steps 1 and 2 should be notified when the event occurs. This is known as registering the event handler.

Event Handling (cont.)

- GUI components can generate many events in response to user interactions.
- Each event is represented by a class and can be processed only by the appropriate type of event handler.

- e.g. (Text Field)
  - When the user presses *Enter* in a `JTextField` or `JPasswordField`, an `ActionEvent` (package java.awt.event) occurs.
  - Processed by an object that implements the interface `ActionListener` (package java.awt.event).
  - To handle this event, a class must implement interface `ActionListener` and declare method `actionPerformed()`
    - This method specifies the tasks to perform when an ActionEvent occurs.

Event Handling (cont.)

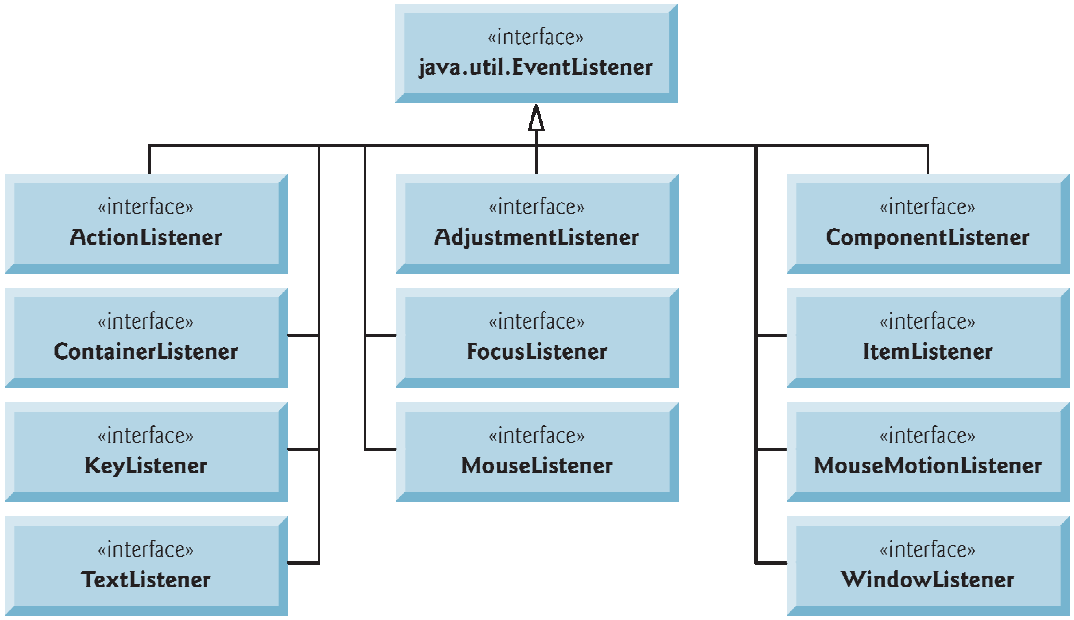


Additional event types that are specific to Swing GUI components are declared in package javax.swing.event.

Fig. 14.11 | Some event classes of package `java.awt.event`.

Event Handling (cont.)

- Some common event-listener interfaces of package (java.awt.event)



Text Fields Demo

```
import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JPasswordField;
import javax.swing.JOptionPane;

public class TextFieldFrame extends JFrame
{
   private JTextField textField1; // text field with set size
   private JTextField textField2; // text field constructed with text
   private JTextField textField3; // text field with text and size
   private JPasswordField passwordField; // password field with text

   // TextFieldFrame constructor adds JTextFields to JFrame
   public TextFieldFrame()
   {
      super( "Testing JTextField and JPasswordField" );
      setLayout( new FlowLayout() ); // set frame layout
```

Text Fields Demo (cont.)

```
// construct textfield with 10 columns
textField1 = new JTextField( 10 );
add( textField1 ); // add textField1 to JFrame

// construct textfield with default text
textField2 = new JTextField( "Enter text here" );
add( textField2 ); // add textField2 to JFrame

// construct textfield with default text and 21 columns
textField3 = new JTextField( "Uneditable text field", 21 );
textField3.setEditable( false ); // disable editing
add( textField3 ); // add textField3 to JFrame

// construct passwordfield with default text
passwordField = new JPasswordField( "Hidden text" );
add( passwordField ); // add passwordField to JFram

// register event handlers
TextFieldHandler handler = new TextFieldHandler();
textField1.addActionListener( handler );
textField2.addActionListener( handler );
textField3.addActionListener( handler );
passwordField.addActionListener( handler );
} // end TextFieldFrame constructor
```

Width of the `JTextField` is based on the component's current font unless a layout manager overrides that size.

Width of the `JTextField` is based on the default text unless a layout manager overrides that size.

Width based on second argument unless a layout manager overrides that size.

Text in this component will be hidden by asterisks (*) by default.

`TextFieldHandler` inner class implements `ActionListener` interface, so it can respond to `JTextField` events. Lines 43–46 register the object `handler` to respond to each component's events.

Text Fields Demo (cont.)

```
// private inner class for event handling
private class TextFieldHandler implements ActionListener
{
   // process text field events
   public void actionPerformed( ActionEvent event )
   {
      String string = ""; // declare string to display

      // user pressed Enter in JTextField textField1
      if ( event.getSource() == textField1 )
         string = String.format( "textField1: %s",
            event.getActionCommand() );

      // user pressed Enter in JTextField textField2
      else if ( event.getSource() == textField2 )
         string = String.format( "textField2: %s",
            event.getActionCommand() );

      // user pressed Enter in JTextField textField3
      else if ( event.getSource() == textField3 )
         string = String.format( "textField3: %s",
            event.getActionCommand() );
```

A `TextFieldHandler` is an `ActionListener`.

Called when the user presses *Enter* in a `JTextField` or `JPasswordField`.

`getSource` specifies which component the user interacted with

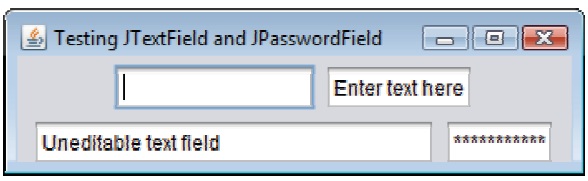
Obtains the text the user typed in the textfield.

Text Fields Demo (cont.)

```
            // user pressed Enter in JTextField passwordField
            else if ( event.getSource() == passwordField )
               string = String.format( "passwordField: %s",
                  event.getActionCommand() );

            // display JTextField content
            JOptionPane.showMessageDialog( null, string );
         } // end method actionPerformed
      } // end private inner class TextFieldHandler
   } // end class TextFieldFrame
```


Text Fields Demo (cont.)

```
// Testing TextFieldFrame.
import javax.swing.JFrame;

public class TextFieldTest
{
   public static void main( String[] args )
   {
      TextFieldFrame textFieldFrame = new TextFieldFrame();
      textFieldFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      textFieldFrame.setSize( 350, 100 ); // set frame size
      textFieldFrame.setVisible( true ); // display frame
   } // end main
} // end class TextFieldTest
```

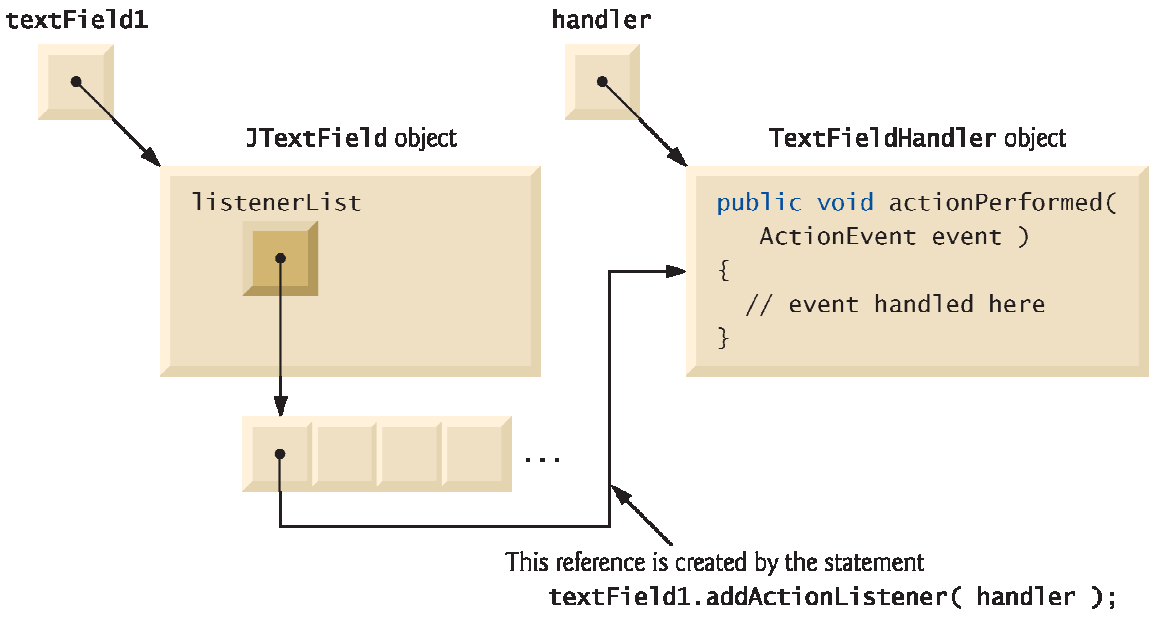


How Event Handling Works

- Every `JComponent` has a variable `listenerList` that refers to an `EventListenerList` (package javax.swing.event).
- Maintains references to registered listeners in the `listenerList`.
- When a listener is registered, a new entry is placed in the component's listenerList.
- Every entry also includes the listener's type.

How Event Handling Works (cont.)

- Event registration for a `JTextField textField1`



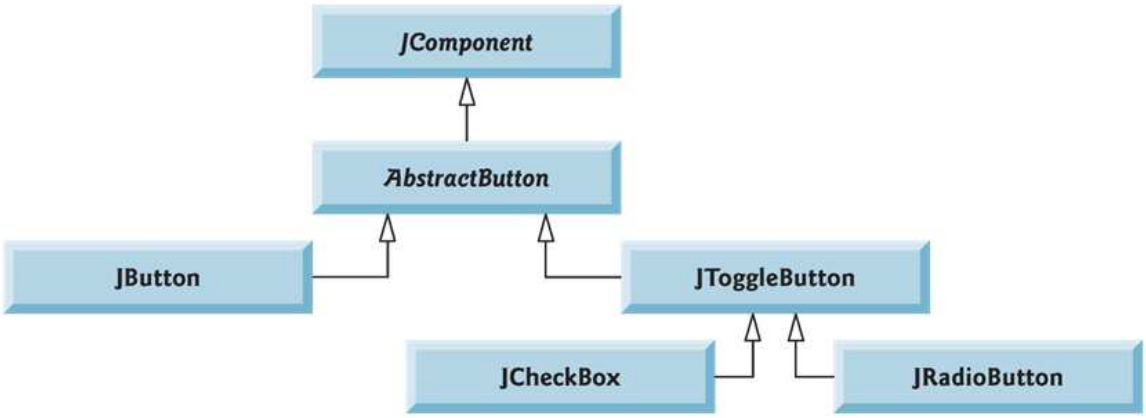
How Event Handling Works (cont.)

- Each event type has one or more corresponding event-listener interfaces.
  - `ActionEvent` is handled by `ActionListener`
  - `MouseEvent` is handled by `MouseListener` and `MouseMotionListener`
  - `KeyEvent` is handled by `KeyListener`
- When an event occurs, the GUI component receives (from the JVM) a unique event ID specifying the event type.
  - The component uses the event ID to decide the listener type to which the event should be dispatched and to decide which method to call on each listener object.

JButton

- A button is a component the user clicks to trigger a specific action.
- Several types of buttons
  - command buttons
  - checkboxes
  - toggle buttons
  - radio buttons
- Button types are subclasses of `AbstractButton` (package `javax.swing`), which declares the common features of Swing buttons.

JButton (cont.)



- Command buttons are created with class JButton.
- JButton, like JTextField , generates an ActionEvent that can be processed by any ActionListener object.
- The text on the face of a JButton is called a button label.

JButton (cont.)

- A JButton can display an Icon.
- A JButton can also have a rollover Icon
  - displayed when the user positions the mouse over the JButton.
  - The icon on the JButton changes as the mouse moves in and out of the JButton's area on the screen.
- The method setRolloverIcon specifies the image displayed on the JButton when the user positions the mouse over it.

Example

```
1  // ButtonFrame.java
2  // Creating JButtons.
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.JButton;
8  import javax.swing.Icon;
9  import javax.swing.ImageIcon;
10 import javax.swing.JOptionPane;
11
12 public class ButtonFrame extends JFrame
13 {
14    private JButton plainJButton; // button with just text
15    private JButton fancyJButton; // button with icons
16
17    // ButtonFrame adds JButtons to JFrame
18    public ButtonFrame()
19    {
20       super( "Testing Buttons" );
21       setLayout( new FlowLayout() ); // set frame layout
22
23       plainJButton = new JButton( "Plain Button" ); // button with text
24       add( plainJButton ); // add plainJButton to JFrame
```


Example (cont.)

```
25
26       Icon bug1 = new ImageIcon( getClass().getResource( "bug1.gif" ) );
27       Icon bug2 = new ImageIcon( getClass().getResource( "bug2.gif" ) );
28       fancyJButton = new JButton( "Fancy Button", bug1 ); // set image
29       fancyJButton.setRolloverIcon( bug2 ); // set rollover image
30       add( fancyJButton ); // add fancyJButton to JFrame
31
32       // create new ButtonHandler for button event handling
33       ButtonHandler handler = new ButtonHandler();
34       fancyJButton.addActionListener( handler );
35       plainJButton.addActionListener( handler );
36    } // end ButtonFrame constructor
37
38    // inner class for button event handling
39    private class ButtonHandler implements ActionListener
40    {
41       // handle button event
42       public void actionPerformed( ActionEvent event )
43       {
44          JOptionPane.showMessageDialog( ButtonFrame.this, String.format(
45             "You pressed: %s", event.getActionCommand() ) );
46       } // end method actionPerformed
47    } // end private inner class ButtonHandler
48 } // end class ButtonFrame
```

Load two images from the same location as class `ButtonFrame`, then use the first as the default icon on the `JButton` and the second as the rollover icon.

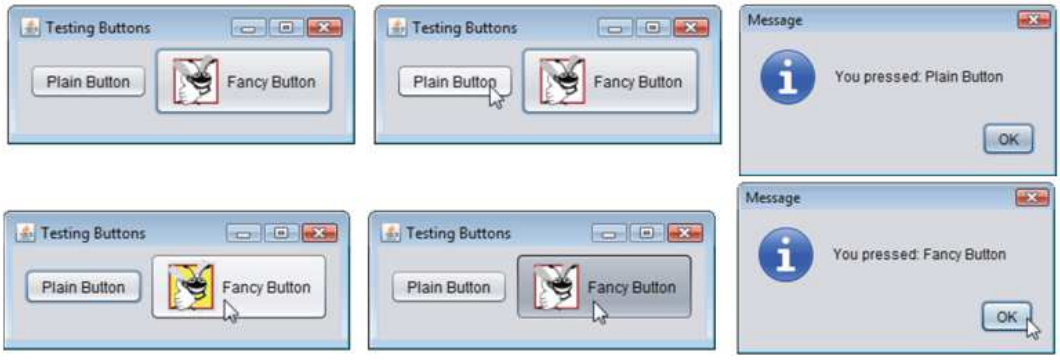
Create object of inner class `ButtonHandler` and register it to handle the `ActionEvents` for both `JButtons`.

Objects of this class can respond to `ActionEvents`.

`ButtonFrame.this` is special notation that enables the inner class to access the `this` reference from the top-level class `ButtonFrame`.

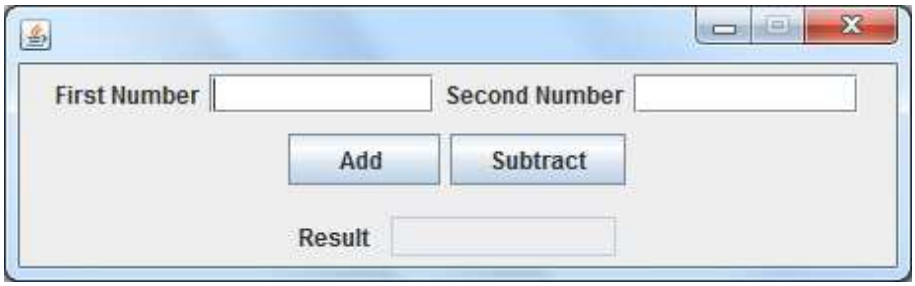
Example (cont.)

```
1  // ButtonTest.java
2  // Testing ButtonFrame.
3  import javax.swing.JFrame;
4
5  public class ButtonTest
6  {
7     public static void main( String[] args )
8     {
9        ButtonFrame buttonFrame = new ButtonFrame(); // create ButtonFrame
10       buttonFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       buttonFrame.setSize( 275, 110 ); // set frame size
12       buttonFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class ButtonTest
```



JButton (cont.)

- A command button generates an `ActionEvent` when the user clicks it.
- Command buttons are created with class JButton.
- The text on the face of a JButton is called a button label.



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

public class AddSub extends JFrame implements ActionListener {
          private JLabel firstNumberLabel,secondNumberLabel,resultLabel;
          private JButton addButton,subButton;
          private JTextField textFiedl1,textFiedl2,textFiedl3;
          private JPanel panel1,panel2,panel3;
          public AddSub(){
           firstNumberLabel = new JLabel("First Number");
           secondNumberLabel = new JLabel("Second Number");
           resultLabel = new JLabel("Result  ");

           addButton = new JButton("   Add    ");
           addButton.addActionListener(this);
           subButton = new JButton(" Subtract ");
           subButton.addActionListener(this);

           textFiedl1 = new JTextField(10);
           textFiedl2 = new JTextField(10);
           textFiedl3 = new JTextField(10);
           textFiedl3.setEditable(false);

           panel1 = new JPanel();
           panel1.add(firstNumberLabel); panel1.add(textFiedl1);
           panel1.add(secondNumberLabel);panel1.add(textFiedl2);
           panel2 = new JPanel(); panel2.add(addButton); panel2.add(subButton);
           panel3 = new JPanel(); panel3.add(resultLabel); panel3.add(textFiedl3);
```



```
          add(panel1,BorderLayout.NORTH);
          add(panel2,BorderLayout.CENTER);
          add(panel3,BorderLayout.SOUTH);

          setSize(550,130);
          setResizable(false);
}

@Override
public void actionPerformed(ActionEvent e) {
     int firstNumber,secondNumber,res;
     firstNumber = Integer.parseInt(textFiedl1.getText());
     secondNumber = Integer.parseInt(textFiedl2.getText());
     if(e.getSource()==addButton)
     {
          res = firstNumber + secondNumber;
          textFiedl3.setText(Integer.toString(res));
     }
     else
     {
          res = firstNumber - secondNumber;
          textFiedl3.setText(""+res);
     }

}
```

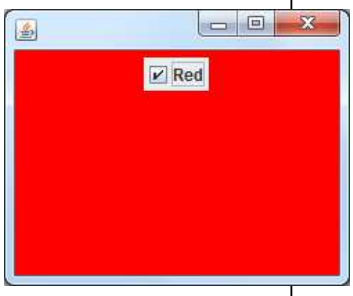
```
        public static void main(String []args)
        {
                AddSub fr = new AddSub();
                fr.setDefaultCloseOperation(EXIT_ON_CLOSE);
                fr.setVisible(true);

        } //end method main
} // end class AddSub
```

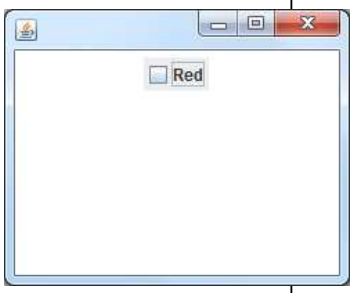
- Note:
  - The `AddSub` class extends the `JFrame` class and implements the `ActionListener` interface.
    - It implements `actionPerformed` method.
    - In this case you have a direct access to all members.
  - The default layout for `JFrame` is `BoarderLayout.`
  - The default layout for `JPanel` is `FlowLayout`.

JCheckBox

- It is a subclass of `JToggleButton`
- When the user clicks a `JCheckBox`, an ItemEvent occurs.
  - Handled by an `ItemListener` object
    - must implement method `itemStateChanged`.
- An `ItemListener` is registered with method `addItemListener`.
- `JCheckBox` method isSelected returns true if a `JCheckBox` is selected.
- `String` passed to the `JCheckBox` constructor is the checkbox label that appears to the right of the `JCheckBox` by default.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JCheckBoxTest extends JFrame implements ItemListener{
        JCheckBox   redCheckBox;
        public JCheckBoxTest(){
                redCheckBox = new JCheckBox("Red");
                redCheckBox.addItemListener(this);
                add(redCheckBox);
                setLayout(new FlowLayout());
                setSize(250,200);
        }
        public void itemStateChanged(ItemEvent e)
        {       Container cont = getContentPane();
                if(redCheckBox.isSelected())
                        cont.setBackground(Color.red);
                else
                        cont.setBackground(Color.white);
        }
        public static void main(String []args)
        {
                JCheckBoxTest fr = new JCheckBoxTest();
                fr.setDefaultCloseOperation(EXIT_ON_CLOSE);
                fr.setVisible(true);
        }
}
```





Example


```
1  // CheckBoxFrame.java
2  // Creating JCheckBox buttons.
3  import java.awt.FlowLayout;
4  import java.awt.Font;
5  import java.awt.event.ItemListener;
6  import java.awt.event.ItemEvent;
7  import javax.swing.JFrame;
8  import javax.swing.JTextField;
9  import javax.swing.JCheckBox;
10
11 public class CheckBoxFrame extends JFrame
12 {
13    private JTextField textField; // displays text in changing fonts
14    private JCheckBox boldJCheckBox; // to select/deselect bold
15    private JCheckBox italicJCheckBox; // to select/deselect italic
16
17    // CheckBoxFrame constructor adds JCheckBoxes to JFrame
18    public CheckBoxFrame()
19    {
20       super( "JCheckBox Test" );
21       setLayout( new FlowLayout() ); // set frame layout
22
23       // set up JTextField and set its font
24       textField = new JTextField( "Watch the font style change", 20 );
```

```
25          textField.setFont( new Font( "Serif", Font.PLAIN, 14 ) );
26          add( textField ); // add textField to JFrame
27
28          boldJCheckBox = new JCheckBox( "Bold" ); // create bold checkbox
29          italicJCheckBox = new JCheckBox( "Italic" ); // create italic
30          add( boldJCheckBox ); // add bold checkbox to JFrame
31          add( italicJCheckBox ); // add italic checkbox to JFrame
32
33          // register listeners for JCheckBoxes
34          CheckBoxHandler handler = new CheckBoxHandler();
35          boldJCheckBox.addItemListener( handler );
36          italicJCheckBox.addItemListener( handler );
37       } // end CheckBoxFrame constructor
38
39       // private inner class for ItemListener event handling
40       private class CheckBoxHandler implements ItemListener
41       {
42          // respond to checkbox events
43          public void itemStateChanged( ItemEvent event )
44          {
45             Font font = null; // stores the new Font
46
47             // determine which CheckBoxes are checked and create Font
48             if ( boldJCheckBox.isSelected() && italicJCheckBox.isSelected() )
49                font = new Font( "Serif", Font.BOLD + Font.ITALIC, 14 );
50             else if ( boldJCheckBox.isSelected() )
51                font = new Font( "Serif", Font.BOLD, 14 );
52             else if ( italicJCheckBox.isSelected() )
53                font = new Font( "Serif", Font.ITALIC, 14 );
54             else
55                font = new Font( "Serif", Font.PLAIN, 14 );
56
57             textField.setFont( font ); // set textField's font
58          } // end method itemStateChanged
59       } // end private inner class CheckBoxHandler
60    } // end class CheckBoxFrame
```

setFont can be used to change the font for any component.

Create and register the event handler for both **JCheckBoxes**.

An object of this class can respond to **ItemEvents**.

JCheckBox method **isSelected** returns **true** if the **JCheckBox** on which it's called is checked.

```
1  // CheckBoxTest.java
2  // Testing CheckBoxFrame.
3  import javax.swing.JFrame;
4
5  public class CheckBoxTest
6  {
7     public static void main( String[] args )
8     {
9        CheckBoxFrame checkBoxFrame = new CheckBoxFrame();
10       checkBoxFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       checkBoxFrame.setSize( 275, 100 ); // set frame size
12       checkBoxFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class CheckBoxTest
```

JRadioButton

- Similar to checkboxes in that they have two states:
  - selected
  - not selected (also called deselected).
- Radio buttons normally appear as a group in which only one button can be selected at a time.
  - Selecting a different radio button forces all others to be deselected.
- Used to represent mutually exclusive options.
- The logical relationship between radio buttons is maintained by a `ButtonGroup` object (from `javax.swing`)
  - ButtonGroup is not a GUI component.

JRadioButton

- Constructors:

```
JRadioButton R1 = new JRadioButton("R1", true); //selected
JRadioButton R2 = new JRadioButton("R2", false);//not selected
JRadioButton R3 = new JRadioButton("R3");
ButtonGroup group = new ButtonGroup();
group.add(R1); group.add(R2); …   // define a logical relationship
```

- Main Method:
  - `Boolean isSelected()`
    - Returns the state of the button. True if it is selected, false if it is not.
- `JRadioButton` generates `ItemEvent` when it is clicked.

- If more than one selected `JRadioButton` object is added to the group, the selected one that was added first will be selected when the GUI is displayed.

Example

```
1  // RadioButtonFrame.java
2  // Creating radio buttons using ButtonGroup and JRadioButton.
3  import java.awt.FlowLayout;
4  import java.awt.Font;
5  import java.awt.event.ItemListener;
6  import java.awt.event.ItemEvent;
7  import javax.swing.JFrame;
8  import javax.swing.JTextField;
9  import javax.swing.JRadioButton;
10 import javax.swing.ButtonGroup;
11
12 public class RadioButtonFrame extends JFrame
13 {
14    private JTextField textField; // used to display font changes
15    private Font plainFont; // font for plain text
16    private Font boldFont; // font for bold text
17    private Font italicFont; // font for italic text
18    private Font boldItalicFont; // font for bold and italic text
19    private JRadioButton plainJRadioButton; // selects plain text
20    private JRadioButton boldJRadioButton; // selects bold text
21    private JRadioButton italicJRadioButton; // selects italic text
22    private JRadioButton boldItalicJRadioButton; // bold and italic
23    private ButtonGroup radioGroup; // buttongroup to hold radio buttons
24
```


Example (cont.)

```
25    // RadioButtonFrame constructor adds JRadioButtons to JFrame
26    public RadioButtonFrame()
27    {
28       super( "RadioButton Test" );
29       setLayout( new FlowLayout() ); // set frame layout
30
31       textField = new JTextField( "Watch the font style change", 25 );
32       add( textField ); // add textField to JFrame
33
34       // create radio buttons
35       plainJRadioButton = new JRadioButton( "Plain", true );
36       boldJRadioButton = new JRadioButton( "Bold", false );
37       italicJRadioButton = new JRadioButton( "Italic", false );
38       boldItalicJRadioButton = new JRadioButton( "Bold/Italic", false );
39       add( plainJRadioButton ); // add plain button to JFrame
40       add( boldJRadioButton ); // add bold button to JFrame
41       add( italicJRadioButton ); // add italic button to JFrame
42       add( boldItalicJRadioButton ); // add bold and italic button
43
44       // create logical relationship between JRadioButtons
45       radioGroup = new ButtonGroup(); // create ButtonGroup
46       radioGroup.add( plainJRadioButton ); // add plain to group
47       radioGroup.add( boldJRadioButton ); // add bold to group
```

Example (cont.)

```
48          radioGroup.add( italicJRadioButton ); // add italic to group
49          radioGroup.add( boldItalicJRadioButton ); // add bold and italic
50
51          // create font objects
52          plainFont = new Font( "Serif", Font.PLAIN, 14 );
53          boldFont = new Font( "Serif", Font.BOLD, 14 );
54          italicFont = new Font( "Serif", Font.ITALIC, 14 );
55          boldItalicFont = new Font( "Serif", Font.BOLD + Font.ITALIC, 14 );
56          textField.setFont( plainFont ); // set initial font to plain
57
58          // register events for JRadioButtons
59          plainJRadioButton.addItemListener(
60             new RadioButtonHandler( plainFont ) );
61          boldJRadioButton.addItemListener(
62             new RadioButtonHandler( boldFont ) );
63          italicJRadioButton.addItemListener(
64             new RadioButtonHandler( italicFont ) );
65          boldItalicJRadioButton.addItemListener(
66             new RadioButtonHandler( boldItalicFont ) );
67       } // end RadioButtonFrame constructor
68
```

Notice that we are creating a separate event-handling object for each `JRadioButton`. This enables us to specify the exact `Font` will be used when a particular one is selected.

Example (cont.)

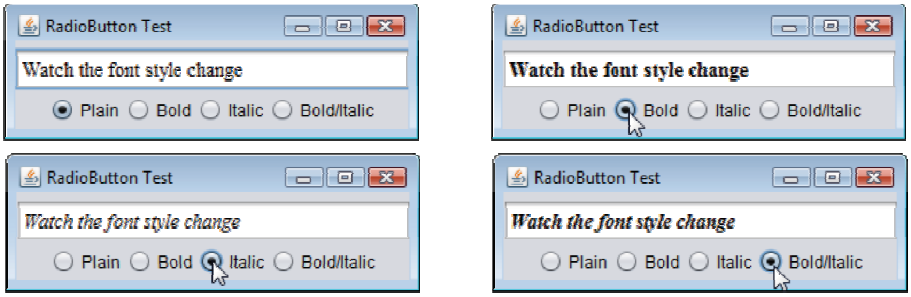
```
69       // private inner class to handle radio button events
70       private class RadioButtonHandler implements ItemListener
71       {
72          private Font font; // font associated with this listener
73
74          public RadioButtonHandler( Font f )
75          {
76             font = f; // set the font of this listener
77          } // end constructor RadioButtonHandler
78
79          // handle radio button events
80          public void itemStateChanged( ItemEvent event )
81          {
82             textField.setFont( font ); // set font of textField
83          } // end method itemStateChanged
84       } // end private inner class RadioButtonHandler
85    } // end class RadioButtonFrame
```

Objects of this class can respond to `ItemEvents`.

Stores the `Font` that is specific to a particular radio button.

Example (cont.)

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



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, while 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 (cont.)

- **There are three ways for you to arrange components in a GUI:**

1. 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***.

2. Layout managers

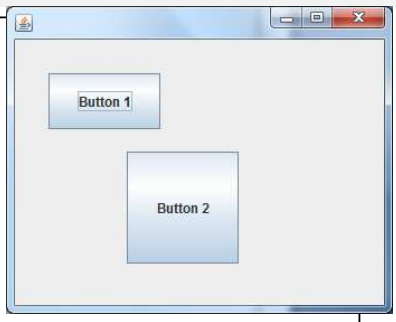
- Simpler and faster than absolute positioning.
- Lose some control over the size and the precise positioning of GUI components.

3. 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.

Absolute Positioning (Do not use it !!!)

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class AbsolutePositioning extends JFrame{
        JButton b1, b2;
        public AbsolutePositioning(){
                setLayout(null);
                b1 = new JButton("Button 1");
                b1.setSize(100,50);
                b1.setLocation(30,30);
                b2 = new JButton("Button 2");
                b2.setSize(100,100);
                b2.setLocation(100,100);
                add(b1); add(b2);
                setSize(250,250);
        }
        public static void main(String []args){
                AbsolutePositioning fr = new AbsolutePositioning();
                fr.setDefaultCloseOperation(EXIT_ON_CLOSE);
                fr.setVisible(true);
        }
}
```



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` method `setAlignment` changes the alignment for the FlowLayout.
  - `FlowLayout.LEFT`
  - `FlowLayout.CENTER`
  - `FlowLayout.RIGHT`

FlowLayout (cont.)

```
import java.awt.FlowLayout;
import java.awt.Container;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JButton;
public class FlowLayoutFrame extends JFrame implements ActionListener
{  private JButton leftJButton, centerJButton, rightJButton;
   private FlowLayout layout; // layout object
   private Container container; // container to set layout
   public FlowLayoutFrame()
   {
      super( "FlowLayout Demo" );
      layout = new FlowLayout(); // create FlowLayout
      container = getContentPane(); // get container to layout
      setLayout( layout ); // set frame layout
      leftJButton = new JButton( "Left" ); // create Left button
      add( leftJButton );          leftJButton.addActionListener(this);
      centerJButton = new JButton( "Center" ); // create Center button
      add( centerJButton ); centerJButton.addActionListener(this);
      rightJButton = new JButton( "Right" ); // create Right button
      add( rightJButton );  rightJButton.addActionListener(this);
   }
```

FlowLayout (cont.)

```
   public void actionPerformed( ActionEvent event )
   {
            if(event.getSource()==leftJButton)
                     layout.setAlignment( FlowLayout.LEFT );
            else if(event.getSource()==rightJButton)
                     layout.setAlignment( FlowLayout.RIGHT );
            else
                     layout.setAlignment( FlowLayout.CENTER );
            layout.layoutContainer( container );
   } // end method actionPerformed
   public static void main( String[] args )
   {
      FlowLayoutFrame flowLayoutFrame = new FlowLayoutFrame();
      flowLayoutFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      flowLayoutFrame.setSize( 400, 100 ); // set frame size
      flowLayoutFrame.setVisible( true ); // display frame
   } // end main
} // end class FlowLayoutFrame
```

BorderLayout

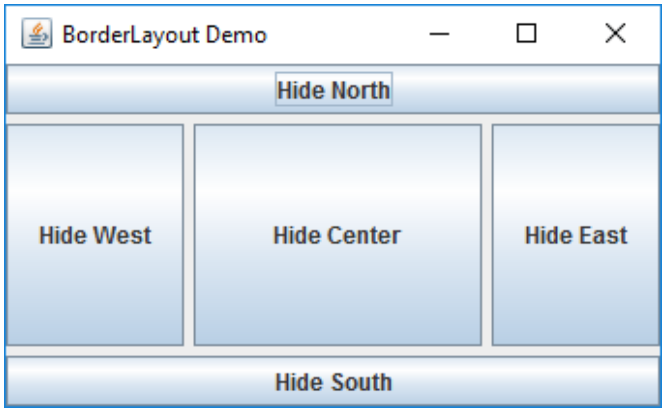
- `BorderLayout`
  - the default layout manager for a `JFrame`
  - arranges components into five regions: NORTH, SOUTH, EAST, WEST and CENTER.
- `BorderLayout` implements interface `LayoutManager2` (a subinterface of `LayoutManager` that adds several methods for enhanced layout processing).
- `BorderLayout` 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` constructor arguments specify the number of pixels between components. (horizontal gap space) and (vertical gap space), respectively.
  - The default is one pixel of gap space horizontally and vertically.

```
import java.awt.BorderLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JButton;

public class BorderLayoutFrame extends JFrame implements ActionListener
{
   private final JButton[] buttons; // array of buttons to hide portions
   private static final String[] names = {"Hide North", "Hide South",
      "Hide East", "Hide West", "Hide Center"};
   private final BorderLayout layout;

   // set up GUI and event handling
   public BorderLayoutFrame()
   {
      super("BorderLayout Demo");

      layout = new BorderLayout(5, 5); // 5 pixel gaps
      setLayout(layout);
      buttons = new JButton[names.length];

      // create JButtons and register listeners for them
      for (int count = 0; count < names.length; count++)
      {
         buttons[count] = new JButton(names[count]);
         buttons[count].addActionListener(this);
      }
```



```
      add(buttons[0], BorderLayout.NORTH);
      add(buttons[1], BorderLayout.SOUTH);
      add(buttons[2], BorderLayout.EAST);
      add(buttons[3], BorderLayout.WEST);
      add(buttons[4], BorderLayout.CENTER);
   } // end BorderLayoutFrame constructor

   // handle button events
   @Override
   public void actionPerformed(ActionEvent event)
   {
      // check event source and lay out content pane correspondingly
      for (JButton button : buttons)
      {
         if (event.getSource() == button)
            button.setVisible(false); // hide the button that was clicked
         else
            button.setVisible(true); // show other buttons
      }

      layout.layoutContainer(getContentPane()); // lay out content pane
   }
} // end class BorderLayoutFrame
```

```
import javax.swing.JFrame;
public class BorderLayoutDemo
{
   public static void main(String[] args)
   {
      BorderLayoutFrame borderLayoutFrame = new BorderLayoutFrame();
      borderLayoutFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      borderLayoutFrame.setSize(300, 200);
      borderLayoutFrame.setVisible(true);
   }
} // end class BorderLayoutDemo
```



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.

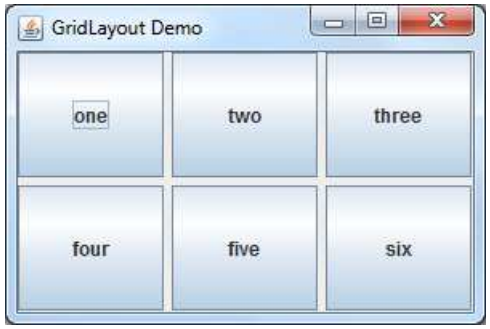
GridLayout (cont.)

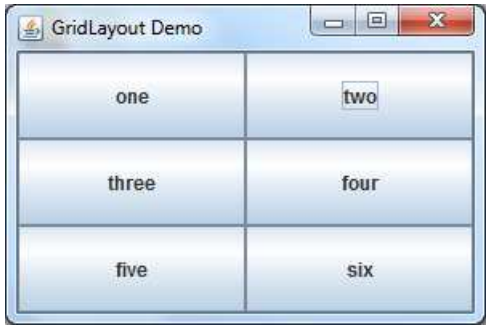
```
import java.awt.GridLayout;
import java.awt.Container;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JButton;
public class GridLayoutFrame extends JFrame implements ActionListener
{
   private JButton[] buttons; // array of buttons
   private static final String[] names =
      { "one", "two", "three", "four", "five", "six" };
   private boolean toggle = true; // toggle between two layouts
   private Container container; // frame container
   private GridLayout gridLayout1; // first gridlayout
   private GridLayout gridLayout2; // second gridlayout
   public GridLayoutFrame()
   {
      super( "GridLayout Demo" );
      gridLayout1 = new GridLayout( 2, 3, 5, 5 ); // 2 by 3; gaps of 5
      gridLayout2 = new GridLayout( 3, 2 ); // 3 by 2; no gaps
      container = getContentPane(); // get content pane
      setLayout( gridLayout1 ); // set JFrame layout
      buttons = new JButton[ names.length ]; // create array of JButtons
```


GridLayout (cont.)

```
      for ( int count = 0; count < names.length; count++ ){
         buttons[ count ] = new JButton( names[ count ] );
         buttons[ count ].addActionListener( this ); // register listener
         add( buttons[ count ] ); // add button to JFrame
      } // end for
   } // end GridLayoutFrame constructor
   // handle button events by toggling between layouts
   public void actionPerformed( ActionEvent event )
   { if ( toggle )
         container.setLayout( gridLayout2 ); // set layout to second
      else
         container.setLayout( gridLayout1 ); // set layout to first
      toggle = !toggle; // set toggle to opposite value
      container.validate(); // re-lay out container
   } // end method actionPerformed
   public static void main( String[] args )
   {
      GridLayoutFrame gridLayoutFrame = new GridLayoutFrame();
      gridLayoutFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      gridLayoutFrame.setSize( 300, 200 ); // set frame size
      gridLayoutFrame.setVisible( true ); // display frame
   } // end main
} // end class GridLayoutFrame
```

GridLayout (cont.)





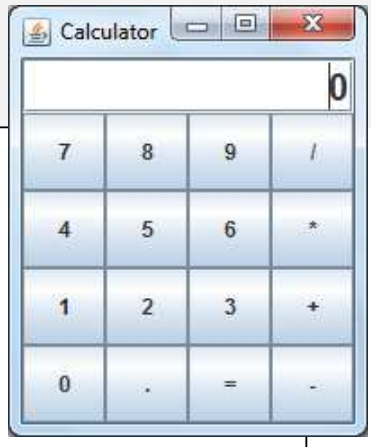
Using Panels to Manage More Complex Layouts

- 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.

Example

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Panel_LayoutFrame extends JFrame
{
   private JButton[] buttons; // array of buttons
   private static final String[] names =
      {"7","8","9","/","4","5","6","*","1","2","3","+","0",".","=","-"};
   private JTextField result;
   private JPanel keys;
   public Panel_LayoutFrame()
   {
      super( "Complex Layout with Panel Demo" );
      keys = new JPanel();
      keys.setLayout( new GridLayout(4,4)); // set JPanel layout
      buttons = new JButton[ names.length ]; // create array of JButtons
      for ( int count = 0; count < names.length; count++ )
      {
         buttons[ count ] = new JButton( names[ count ] );
         keys.add( buttons[ count ] ); // add button to JPanel
      }
```


Example (cont.)




```
      result = new JTextField("0");
      result.setHorizontalAlignment(JTextField.RIGHT);
      Font font1 = new Font("SansSerif", Font.BOLD, 20);
      result.setFont(font1);
      add(result, BorderLayout.NORTH);
      add(keys, BorderLayout.CENTER);
   } // end Panel_LayoutFrame constructor

   public static void main( String[] args )
   {
      Panel_LayoutFrame Panel_LayoutFrame = new Panel_LayoutFrame();
      Panel_LayoutFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      Panel_LayoutFrame.setSize( 200, 240 ); // set frame size
      Panel_LayoutFrame.setVisible( true ); // display frame
   } // end main
} // end class Panel_LayoutFrame
```

JComboBox

- 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` generates `ItemEvent`
- **Constructors:**
  - Default Constructor : create an empty Combo Box
  - Constructor with an array of objects (e.g strings)

```
String S[] = {"A" , "B" , "C", "D" };
JComboBox comboBox1 = new JComboBox (S);
// it takes array of strings
```


JComboBox (cont.)

- **Methods:**
  - `int getSelectedIndex()`:
    - returns the index of the selected item.
  - `void addItem(Object ob)`:
    - Adds an item to the item list.
  - `void setMaximumRowCount(int count)`:
    - Sets the maximum number of rows the JComboBox displays.
  - `void showPopup()`:
    - Causes the combo box to display its popup window
  - `object getSelectedItem()`:
    - Returns the current selected item.
  - `void removeItemAt( int index)`:
    - Removes the item at an Index
  - `void removeAllItems()`:
    - Removes all items from the item list.

JComboBox (cont.)

- The first item added to a `JComboBox` appears as the currently selected item when the `JComboBox` is displayed.
- Other items are selected by clicking the `JComboBox`, then selecting an item from the list that appears.
- `JComboBox` method `setMaximumRowCount` sets the maximum number of elements that are displayed when the user clicks the `JComboBox`.
- If there are additional items, the `JComboBox` provides a scrollbar that allows the user to scroll through all the elements in the list.

JComboBox (cont.)

- For each item selected from a `JComboBox`, another item is first deselected

  ➔ so two `ItemEvent`s occur when an item is selected.

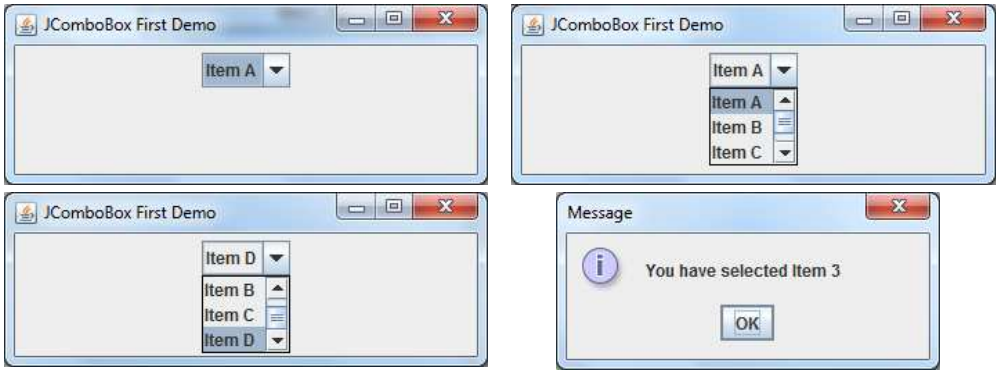
- `ItemEvent` method `getStateChange` returns the type of state change.
  - `ItemEvent.SELECTED` indicates that an item was selected.

Anonymous Inner Class

- An anonymous inner class is an inner class that is declared without a name and typically appears inside a method declaration.
- As with other inner classes, an anonymous inner class can access its top-level class's members.
- An anonymous inner class has limited access to the local variables of the method in which it's declared.
- Since an anonymous inner class has no name, **one object** of the anonymous inner class must be created at the point where the class is declared.
  - You define it, and create an object of that type as a parameter all in one line.


```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class ComboBox_FirstDemo extends JFrame {
   private JComboBox comboBox1;
   private String names[] = { "Item A", "Item B",  "Item C", "Item D" };
   public ComboBox_FirstDemo()
   {
      super( "JComboBox First Demo" );
      setLayout( new FlowLayout() );
      // set up JComboBox and register its event handler
      comboBox1 = new JComboBox( names );
      comboBox1.setMaximumRowCount( 3 );
      comboBox1.addItemListener(
         // anonymous inner class to handle JComboBox events
         new ItemListener() {
            // handle JComboBox event
            public void itemStateChanged( ItemEvent event )
            {
               // determine whether check box selected
                 if ( event.getStateChange() == ItemEvent.SELECTED )
                JOptionPane.showMessageDialog(null,"You have selected Item "
                                          + comboBox1.getSelectedIndex());
            }
         }  // end anonymous inner class
      ); // end call to addItemListener
```

```
      add( comboBox1 );
      setSize( 350, 100 );
      setVisible( true );
   }
   public static void main( String args[] )
   {
      ComboBox_FirstDemo application = new ComboBox_FirstDemo();
      application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE );
   }
}  // end class ComboBox_FirstDemo
```



Two files will be generated:

- ComboBox_FirstDemo.class
- ComboBox_FirstDemo$1.class

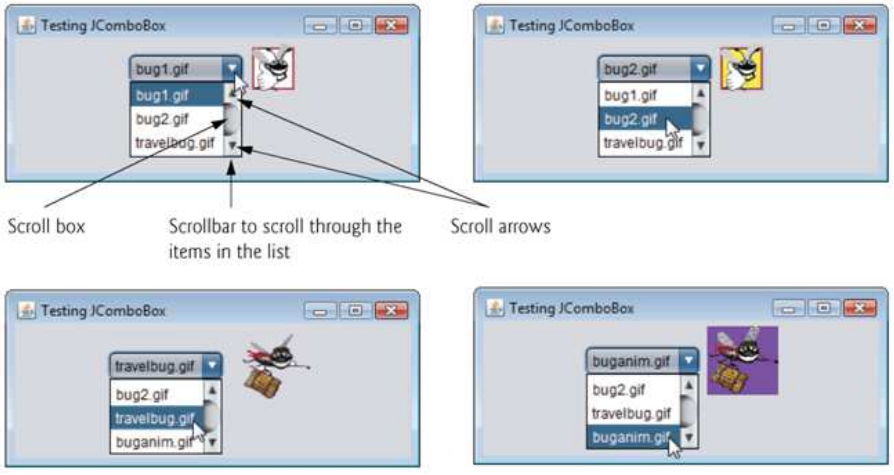
Example 2

```
import java.awt.FlowLayout;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JComboBox;
import javax.swing.Icon;
import javax.swing.ImageIcon;

public class ComboBoxFrame extends JFrame
{
   private JComboBox imagesJComboBox; // combobox to hold names of icons
   private JLabel label; // label to display selected icon
   private static final String[] names =
      { "bug1.gif", "bug2.gif",  "travelbug.gif", "buganim.gif" };
   private Icon[] icons = {
      new ImageIcon( getClass().getResource( names[ 0 ] ) ),
      new ImageIcon( getClass().getResource( names[ 1 ] ) ),
      new ImageIcon( getClass().getResource( names[ 2 ] ) ),
      new ImageIcon( getClass().getResource( names[ 3 ] ) ) };
```

```
    // ComboBoxFrame constructor adds JComboBox to JFrame
   public ComboBoxFrame()
   {
      super( "Testing JComboBox" );
      setLayout( new FlowLayout() ); // set frame layout
      imagesJComboBox = new JComboBox( names ); // set up JComboBox
      imagesJComboBox.setMaximumRowCount( 3 ); // display three rows
      imagesJComboBox.addItemListener(
         new ItemListener() // anonymous inner class
         {
            // handle JComboBox event
            public void itemStateChanged( ItemEvent event )
            {
               // determine whether item selected
               if ( event.getStateChange() == ItemEvent.SELECTED )
                  label.setIcon( icons[
                     imagesJComboBox.getSelectedIndex() ] );
            } // end method itemStateChanged
         } // end anonymous inner class
      ); // end call to addItemListener
      add( imagesJComboBox ); // add combobox to JFrame
      label = new JLabel( icons[ 0 ] ); // display first icon
      add( label ); // add label to JFrame
   } // end ComboBoxFrame constructor
} // end class ComboBoxFrame
```



```
import javax.swing.JFrame;

public class ComboBoxTest
{
   public static void main( String[] args )
   {
      ComboBoxFrame comboBoxFrame = new ComboBoxFrame();
      comboBoxFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      comboBoxFrame.setSize( 350, 150 ); // set frame size
      comboBoxFrame.setVisible( true ); // display frame
   } // end main
} // end class ComboBoxTest
```



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`
- `JList` generates `ListSelectionEvent` in single-selection lists.
  - You have to implement `ListSelectionListener`
    - one abstarct method `valueChanged(ListSelectionEvent event)`
  - `addListSelectionListener()` to registers a listener
- 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.

JList – Methods:

- `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).
- `getSelectedIndex()`
  - Returns the selected item's index(in single selection mode)
- `setListData(Object[] listData)`
  - Constructs a read-only ListModel from an array of objects,

JList (cont.)

- Each JFrame actually consists of three layers
  - **The Background**:
  - **The content pane**:
    - The content pane appears in front of the background and is where the GUI components in the JFrame are displayed.
  - **The glass pane**:
    - 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).

Example

```
1  // 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
```

Example (cont.)

```
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
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
```

Populate the `JList` with the `Strings` in array `colorNames`.

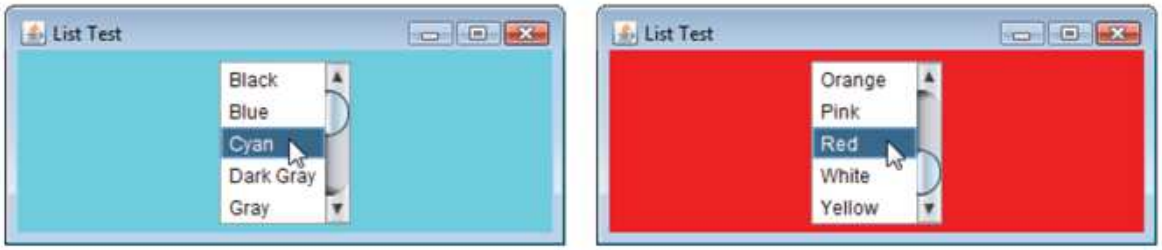
Allow only single selections.

Provide scrollbars for the `JList` if necessary.

Choose the appropriate `Color` to change the window's background color.

Example (cont.)

```
1  // 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
```



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.

Multiple-Selection Lists (cont.)

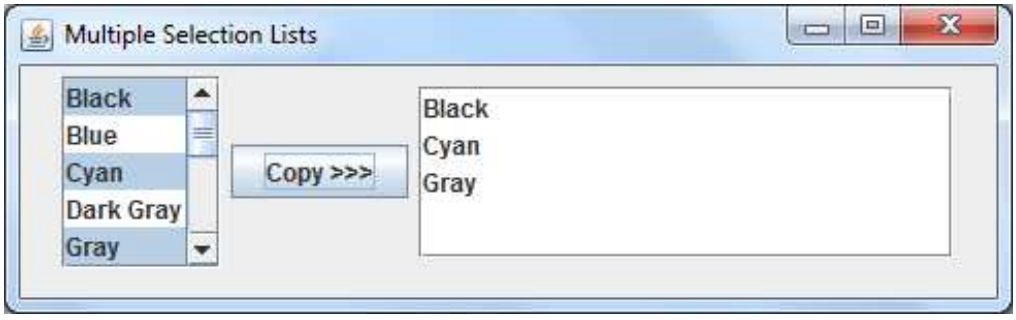
- If a `JList` does not contain items it will not display in a `FlowLayout`.
  - use `JList` methods `setFixedCellWidth` and `setFixedCellHeight` to set the item width and height
- There are no events to indicate that a user has made multiple selections in a multiple-selection list.
  - An event generated 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`.

```
import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JButton;
import javax.swing.JScrollPane;
import javax.swing.ListSelectionModel;
public class MultipleSelectionFrame extends JFrame
{
   private JList colorJList; // list to hold color names
   private JList copyJList; // list to copy color names into
   private JButton copyJButton; // button to copy selected names
   private static final String[] colorNames = { "Black", "Blue", "Cyan",
      "Dark Gray", "Gray", "Green", "Light Gray", "Magenta", "Orange",
      "Pink", "Red", "White", "Yellow" };
   // MultipleSelectionFrame constructor
   public MultipleSelectionFrame()
   {
      super( "Multiple Selection Lists" );
      setLayout( new FlowLayout() ); // set frame layout
      colorJList = new JList( colorNames ); // holds names of all colors
      colorJList.setVisibleRowCount( 5 ); // show five rows
      colorJList.setSelectionMode(
         ListSelectionModel.MULTIPLE_INTERVAL_SELECTION );
      add( new JScrollPane( colorJList ) ); // add list with scrollpane
```



```
      copyJButton = new JButton( "Copy >>>" ); // create copy button
      copyJButton.addActionListener(
         new ActionListener() // anonymous inner class
         {
            // handle button event
            public void actionPerformed( ActionEvent event )
            {
               // place selected values in copyJList
               copyJList.setListData( colorJList.getSelectedValues() );
            } // end method actionPerformed
         } // end anonymous inner class
      ); // end call to addActionListener
      add( copyJButton ); // add copy button to JFrame
      copyJList = new JList(); // create list to hold copied color names
      copyJList.setVisibleRowCount( 5 ); // show 5 rows
      copyJList.setFixedCellWidth( 100 ); // set width
      copyJList.setFixedCellHeight( 15 ); // set height
      copyJList.setSelectionMode(
         ListSelectionModel.SINGLE_INTERVAL_SELECTION );
      add( new JScrollPane( copyJList ) ); // add list with scrollpane
   } // end MultipleSelectionFrame constructor
} // end class MultipleSelectionFrame
```

```
import javax.swing.JFrame;
public class MultipleSelectionTest
{
   public static void main( String[] args )
   {
      MultipleSelectionFrame multipleSelectionFrame =
         new MultipleSelectionFrame();
      multipleSelectionFrame.setDefaultCloseOperation(
         JFrame.EXIT_ON_CLOSE );
      multipleSelectionFrame.setSize( 350, 150 ); // set frame size
      multipleSelectionFrame.setVisible( true ); // display frame
   } // end main
} // end class MultipleSelectionTest
```



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 Interafce Methods

- `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.

MouseMotionListener Interafce Methods

- `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 when the mouse cursor is on a component.
  - All move events are sent to the component over which the mouse is currently positioned.

Mouse Event Handling (cont.)

- Each mouse event-handling method receives a `MouseEvent` object.
  - MouseEvent methods `getX()` and `getY()` return the *x*- and *y*-coordinates of the mouse at the time the event occurred.
  - Coordinates are measured from the upper-left corner of the GUI component on which the event occurred.
  - Methods `addMouseListener` and `addMouseMotionListener` register `MouseListener`s and `MouseMotionListener`s, respectively.

- The methods and constants of class `InputEvent` (MouseEvent's superclass) enable you to determine which mouse button the user clicked.

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.

Example

```
import java.awt.Color;
import java.awt.BorderLayout;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.event.MouseEvent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
public class MouseTrackerFrame extends JFrame
{
   private JPanel mousePanel; // panel in which mouse events will occur
   private JLabel statusBar; // label that displays event information
   // MouseTrackerFrame constructor sets up GUI and
   // registers mouse event handlers
   public MouseTrackerFrame()
   {
      super( "Demonstrating Mouse Events" );
      mousePanel = new JPanel(); // create panel
      mousePanel.setBackground( Color.WHITE ); // set background color
      add( mousePanel, BorderLayout.CENTER ); // add panel to JFrame
      statusBar = new JLabel( "Mouse outside JPanel" );
      add( statusBar, BorderLayout.SOUTH ); // add label to JFrame
```


Example (cont.)

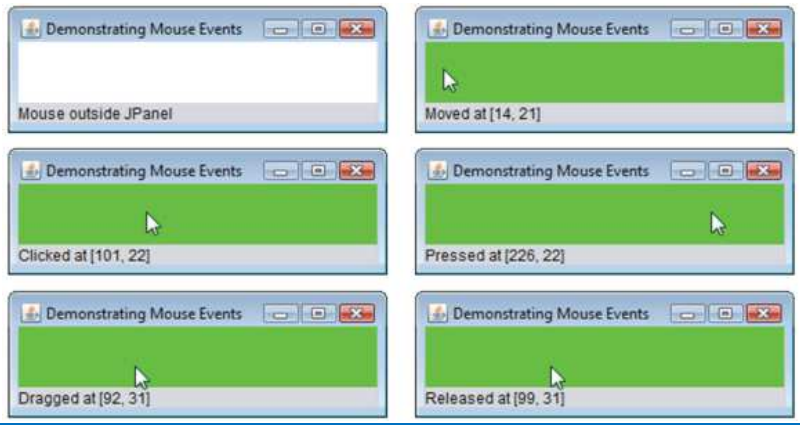
```
      // create and register listener for mouse and mouse motion events
      MouseHandler handler = new MouseHandler();
      mousePanel.addMouseListener( handler );
      mousePanel.addMouseMotionListener( handler );
   } // end MouseTrackerFrame constructor
   private class MouseHandler implements MouseListener, MouseMotionListener
   {  // MouseListener event handlers
      // handle event when mouse released immediately after press
      public void mouseClicked( MouseEvent event )
      {
         statusBar.setText( String.format( "Clicked at [%d, %d]",
            event.getX(), event.getY() ) );
      } // end method mouseClicked
      public void mousePressed( MouseEvent event )
      {
         statusBar.setText( String.format( "Pressed at [%d, %d]",
            event.getX(), event.getY() ) );
      } // end method mousePressed
      public void mouseReleased( MouseEvent event )
      {
         statusBar.setText( String.format( "Released at [%d, %d]",
            event.getX(), event.getY() ) );
      } // end method mouseReleased
```

Example (cont.)

```
      public void mouseEntered( MouseEvent event )
      {
         statusBar.setText( String.format( "Mouse entered at [%d, %d]",
            event.getX(), event.getY() ) );
         mousePanel.setBackground( Color.GREEN );
      } // end method mouseEntered
      public void mouseExited( MouseEvent event )
      {
         statusBar.setText( "Mouse outside JPanel" );
         mousePanel.setBackground( Color.WHITE );
      } // end method mouseExited
      public void mouseDragged( MouseEvent event )
      {
         statusBar.setText( String.format( "Dragged at [%d, %d]",
            event.getX(), event.getY() ) );
      } // end method mouseDragged
      public void mouseMoved( MouseEvent event )
      {
         statusBar.setText( String.format( "Moved at [%d, %d]",
            event.getX(), event.getY() ) );
      } // end method mouseMoved
   } // end inner class MouseHandler
} // end class MouseTrackerFrame
```


Example (cont.)

```
import javax.swing.JFrame;
public class MouseTracker
{
   public static void main( String[] args )
   {
      MouseTrackerFrame mouseTrackerFrame = new MouseTrackerFrame();
      mouseTrackerFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      mouseTrackerFrame.setSize( 300, 100 ); // set frame size
      mouseTrackerFrame.setVisible( true ); // display frame
   } // end main
} // end class MouseTracker
```



Adapter Classes

- Many event-listener interfaces contain multiple methods.
  - e.g. `WindowListener` contains 7 abstract methods. (**Problem**)
- 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 implementations and override only the method(s) you need for event handling.

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

Example

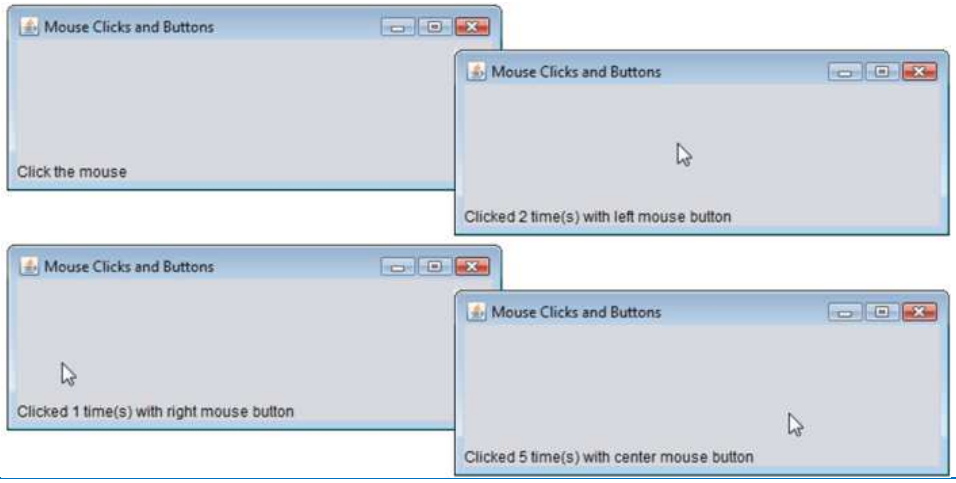
```
import java.awt.BorderLayout;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import javax.swing.JFrame;
import javax.swing.JLabel;
public class MouseDetailsFrame extends JFrame
{
   private String details; // String that is displayed in the statusBar
   private JLabel statusBar; // JLabel that appears at bottom of window
   // constructor sets title bar String and register mouse listener
   public MouseDetailsFrame()
   {
      super( "Mouse Clicks and Buttons" );
      statusBar = new JLabel( "Click the mouse" );
      add( statusBar, BorderLayout.SOUTH );
      addMouseListener( new MouseClickHandler() ); // add handler
   } // end MouseDetailsFrame constructor
   // inner class to handle mouse events
   private class MouseClickHandler extends MouseAdapter
   {
      // handle mouse-click event and determine which button was pressed
      public void mouseClicked( MouseEvent event )
      {
```

Example (cont.)

```
         int xPos = event.getX(); // get x-position of mouse
         int yPos = event.getY(); // get y-position of mouse
         details = String.format( "Clicked %d time(s)",
            event.getClickCount() );

         if (event.getButton() == MouseEvent.BUTTON1) // left mouse button
            details += " with left mouse button";
         else if (event.getButton() == MouseEvent.BUTTON2) // middle mouse button
            details += " with center mouse button";
         else if (event.getButton() == MouseEvent.BUTTON3) // right mouse button
            details += " with right mouse button";
          statusBar.setText( details ); // display message in statusBar
      } // end method mouseClicked
   } // end private inner class MouseClickHandler
} // end class MouseDetailsFrame
```


Example (cont.)

```
import javax.swing.JFrame;
public class MouseDetails{
   public static void main( String[] args ){
      MouseDetailsFrame mouseDetailsFrame = new MouseDetailsFrame();
      mouseDetailsFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      mouseDetailsFrame.setSize( 400, 150 ); // set frame size
      mouseDetailsFrame.setVisible( true ); // display frame
   } // end main
} // end class MouseDetails
```



Example2

```
import java.awt.Color;
import java.awt.Point;
import java.awt.Rectangle;
import java.awt.Graphics;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionAdapter;
import java.awt.event.MouseAdapter;
import javax.swing.JPanel;
import java.util.Vector;
public class PaintPanel extends JPanel
{
   private Point point1, point2;
   private Vector shapeVector;
   private boolean dragStatus = false;
   private Rectangle currRec, newRec;
   // set up GUI and register mouse event handlers
   public PaintPanel()
   {
            point1 = point2 = new Point(0,0);
            shapeVector = new Vector();
            // handle frame mouse event
```


Example 2 (cont.)

```
addMouseListener(
      new MouseAdapter() // anonymous inner class
      {
          public void mousePressed( MouseEvent event )
          {
              point1 = event.getPoint(); // find the first point
              if (event.getButton() == MouseEvent.BUTTON3)//right mouse button
                   shapeVector.clear(); // clear the shapes vector
          } // end method mousePressed
           public void mouseReleased( MouseEvent event )
           {
                point2 = event.getPoint(); // find the second point
                newRec = new Rectangle(Math.min(point1.x,point2.x),
                                       Math.min(point1.y,point2.y),
                                       Math.abs(point2.x-point1.x),
                                       Math.abs(point2.y-point1.y));
                shapeVector.add(newRec);
                dragStatus = false;
                repaint(); // repaint JFrame
              } // end method mouseReleased
      } // end anonymous inner class
  ); // end call to addMouseMotionListener
```

Example2

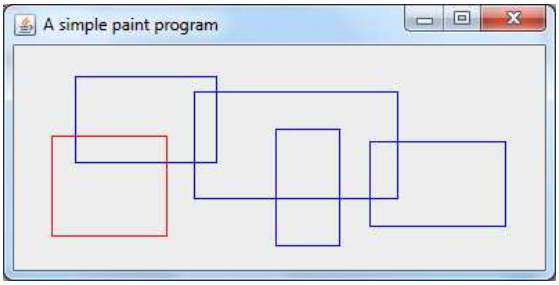
```
      // handle frame mouse motion event
      addMouseMotionListener(
         new MouseMotionAdapter() // anonymous inner class
         {
        // store drag coordinates and repaint
            public void mouseDragged( MouseEvent event )
            {
                  point2 = event.getPoint(); // find the second point
                  currRec = new Rectangle(Math.min(point1.x,point2.x),
                                          Math.min(point1.y,point2.y),
                                          Math.abs(point2.x-point1.x),
                                          Math.abs(point2.y-point1.y));
                  dragStatus = true;
                  repaint(); // repaint JFrame
            } // end method mouseDragged
         } // end anonymous inner class
      ); // end call to addMouseMotionListener

   } // end PaintPanel constructor
```


Example2 (cont.)

```
   public void paintComponent( Graphics g )
   {
      super.paintComponent( g ); // clears drawing area
      Rectangle rect;
      g.setColor(Color.blue);
      for(int i=0; i<shapeVector.size(); i++)
      {
           rect = (Rectangle) shapeVector.get(i);
           g.drawRect(rect.x, rect.y, rect.width, rect.height);
      }
      g.setColor(Color.red);
      if(dragStatus)
           g.drawRect(currRec.x, currRec.y, currRec.width, currRec.height);
   } // end method paintComponent
} // end class PaintPanel
```

Example2 (cont.)

```
import java.awt.BorderLayout;
import javax.swing.JFrame;
public class Painter
{
   public static void main( String[] args )
   {
      JFrame application = new JFrame( "A simple paint program" );
      PaintPanel paintPanel = new PaintPanel(); // create paint panel
      application.add( paintPanel, BorderLayout.CENTER ); // in center
      application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      application.setSize( 400, 200 ); // set frame size
      application.setVisible( true ); // display frame
   } // end main
} // end class Painter
```



Key Event Handling

- `KeyEvent` is generated when keys on the keyboard are pressed and released.
  - Class `KeyEvent` is a subclass of `InputEvent`.
- `KeyListener` interfaceis used for handling key events.
- A `KeyListener` must define 3 methods:
  - 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.
  - ❖ each receives a `KeyEvent` object as its argument
- Registers key event handlers with method `addKeyListener`

Key Event Handling (cont.)

- `KeyEvent` Methods:
  - `int 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.
  - `char getKeyChar()`: gets the Unicode value of the character typed.
  - `boolean isActionKey()`: determines whether the key in the event was an action key.
  - `int 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.

Example

```
import java.awt.Color;
import java.awt.event.KeyListener;
import java.awt.event.KeyEvent;
import javax.swing.JFrame;
import javax.swing.JTextArea;
public class KeyDemoFrame extends JFrame implements KeyListener
{
   private String line1 = ""; // first line of textarea
   private String line2 = ""; // second line of textarea
   private String line3 = ""; // third line of textarea
   private JTextArea textArea; // textarea to display output
   // KeyDemoFrame constructor
   public KeyDemoFrame()
   {
      super( "Demonstrating Keystroke Events" );
      textArea = new JTextArea( 10, 15 ); // set up JTextArea
      textArea.setText( "Press any key on the keyboard..." );
      textArea.setEnabled( false ); // disable textarea
      textArea.setDisabledTextColor( Color.BLACK ); // set text color
      add( textArea ); // add textarea to JFrame
      addKeyListener( this ); // allow frame to process key events
   } // end KeyDemoFrame constructor
```

Example (cont.)

```
   // handle press of any key
   public void keyPressed( KeyEvent event )
   {
      line1 = String.format( "Key pressed: %s",
        KeyEvent.getKeyText( event.getKeyCode() ) ); // show pressed key
      setLines2and3( event ); // set output lines two and three
   } // end method keyPressed
   // handle release of any key
   public void keyReleased( KeyEvent event )
   {
      line1 = String.format( "Key released: %s",
         KeyEvent.getKeyText( event.getKeyCode() ) ); // show released key
      setLines2and3( event ); // set output lines two and three
   } // end method keyReleased
   // handle press of an action key
   public void keyTyped( KeyEvent event )
   {
      line1 = String.format( "Key typed: %s", event.getKeyChar() );
      setLines2and3( event ); // set output lines two and three
   } // end method keyTyped
```


Example (cont.)

```
   // set second and third lines of output
   private void setLines2and3( KeyEvent event )
   {
      line2 = String.format( "This key is %san action key",
         ( event.isActionKey() ? "" : "not " ) );
      String temp = KeyEvent.getKeyModifiersText( event.getModifiers() );
      line3 = String.format( "Modifier keys pressed: %s",
         ( temp.equals( "" ) ? "none" : temp ) ); // output modifiers
      textArea.setText( String.format( "%s\n%s\n%s\n",
         line1, line2, line3 ) ); // output three lines of text
   } // end method setLines2and3
} // end class KeyDemoFrame
```

```
import javax.swing.JFrame;
public class KeyDemo
{  public static void main( String[] args )
   {
      KeyDemoFrame keyDemoFrame = new KeyDemoFrame();
      keyDemoFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      keyDemoFrame.setSize( 350, 100 ); // set frame size
      keyDemoFrame.setVisible( true ); // display frame
   } // end main
} // end class KeyDemo
```

Example (cont.)

JTextArea

- A `JTextArea` provides an area for manipulating multiple lines of text.
  - It is a subclass of `JTextComponent`, which declares common methods for `JTextField`s, `JTextArea`s and several other text-based GUI components.
- Constructor:
  - `JTextArea(int rows, int columns)`
    - Constructs a new empty TextArea with the specified number of rows and columns.
  - `JTextArea(String text)`
    - Constructs a new TextArea with the specified text displayed.
  - `JTextArea(String text, int rows, int columns)`
    - Constructs a new TextArea with the specified text and number of rows and columns.
- Main Methods:
  - `String getSelectedText()` :(inherited from JTextComponent) returns the selected text from a JTextArea.
  - `void setText(String t)`: changes the text in a JTextArea. When text reaches the right edge of a JTextArea the text can wrap to the next line.
  - `void setEditable(boolean b)`

JTextArea (cont.)

- Box: (from javax.swing)
  - It is a subclass of Container
  - uses a BoxLayout to arrange the GUI components horizontally or vertically.

- Box static methods:
  - createHorizontalBox() :
    - creates a Box that arranges components left to right in the order that they are attached.
  - createVerticalBox():
    - creates a Box that arranges components top to bottom in the order that they are attached.

JTextArea (cont.)

- JScrollPane
  - Provides a scrollable view of a lightweight component.
  - By default, it provides scrollbars only if they are required.
  - You can set the horizontal and vertical scrollbar policies of a JScrollPane when it's constructed.
  - Constructor:
    - `JScrollPane(Component view)`
    - `JScrollPane(Component view, int vsbPolicy, int hsbPolicy)`
  - You can also use JScrollPane methods
    - `void setHorizontalScrollBarPolicy(int policy)`
    - `void setVerticalScrollBarPolicy(int policy)`

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.

Example

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.Box;
import javax.swing.JFrame;
import javax.swing.JTextArea;
import javax.swing.JButton;
import javax.swing.JScrollPane;
public class TextAreaFrame extends JFrame
{
   private JTextArea textArea1; // displays demo string
   private JTextArea textArea2; // highlighted text is copied here
   private JButton copyJButton; // initiates copying of text
   // no-argument constructor
   public TextAreaFrame()
   {
      super( "TextArea Demo" );
      Box box = Box.createHorizontalBox(); // create box
      String demo = "This is a demo string to\n" +
         "illustrate copying text\nfrom one textarea to \n" +
         "another textarea using an\nexternal event\n";
      textArea1 = new JTextArea( demo, 10, 15 ); // create textarea1
      box.add( new JScrollPane( textArea1 ) ); // add scrollpane
```

Example (cont.)

```
      copyJButton = new JButton( "Copy >>>" ); // create copy button
      box.add( copyJButton ); // add copy button to box
      copyJButton.addActionListener(
         new ActionListener() // anonymous inner class
         {
            // set text in textArea2 to selected text from textArea1
            public void actionPerformed( ActionEvent event )
            {
               textArea2.setText( textArea1.getSelectedText() );
            } // end method actionPerformed
         } // end anonymous inner class
      ); // end call to addActionListener
      textArea2 = new JTextArea( 10, 15 ); // create second textarea
      textArea2.setEditable( false ); // disable editing
      box.add( new JScrollPane( textArea2 ) ); // add scrollpane
      add( box ); // add box to frame
   } // end TextAreaFrame constructor
} // end class TextAreaFrame
```


Example (cont.)

```
import javax.swing.JFrame;
public class TextAreaDemo
{
   public static void main( String[] args )
   {
      TextAreaFrame textAreaFrame = new TextAreaFrame();
      textAreaFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      textAreaFrame.setSize( 425, 200 ); // set frame size
      textAreaFrame.setVisible( true ); // display frame
   } // end main
} // end class TextAreaDemo
```