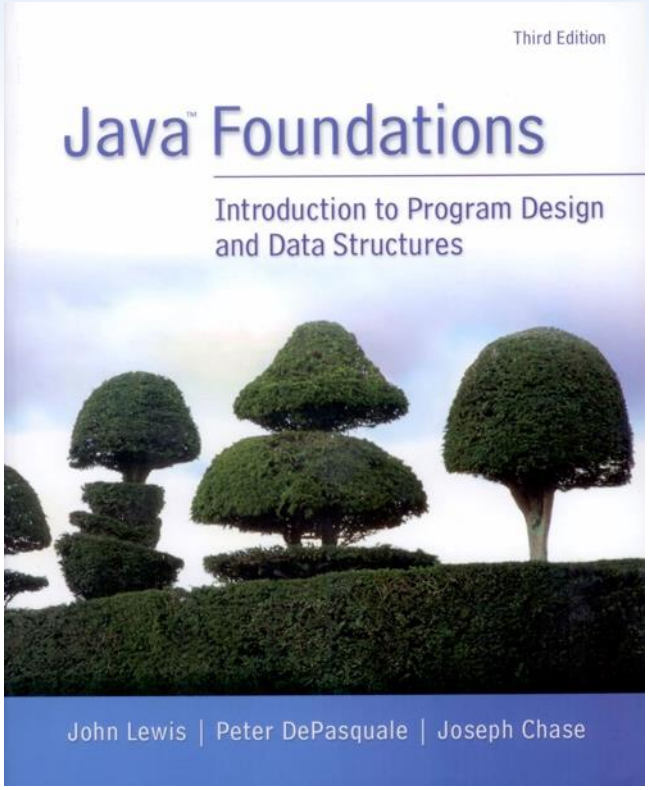


Chapter 6

Graphical User Interfaces

Chapter Scope

- GUI components, events, and listeners
- Containers
- Buttons, text fields, sliders, combo boxes
- Layout managers
- Mouse and keyboard events
- Dialog boxes
- Borders, tool tips, and mnemonics

GUI Elements

- Examples from previous chapters are known as *command-line applications*, which interact with the user through simple prompts and feedback
- Command-line applications lack the rich user experience
- With graphical user interfaces (GUI), the user is not limited to responding to prompts in a particular order and receiving feedback in one place

GUI Elements

- Three kinds of objects are needed to create a GUI in Java:
  - components
  - events
  - listeners
- *Component* – an object that defines a screen element used to display information or allow the user to interact with the program
  - A *container* is a special type of component that is used to hold and organize other components

GUI Elements

- *Event* – an object that represents some occurrence in which we may be interested
  - Often correspond to user actions (mouse button press, keyboard key press)
  - Most GUI components generate events to indicate a user action related to that component
  - Program that is oriented around GUI, responding to user events is called *event-driven*

GUI Elements

- *Listener* – an object that "waits" for an event to occur and responds in way when it does
  - In designing a GUI-based program we need to establish the relationships between the listener, the event it listens for, and the component that generates the event
- It's common to use existing components and events from the Java class library.
- We will, however, write our own listener classes to perform whatever actions we desire when events occur

GUI Elements

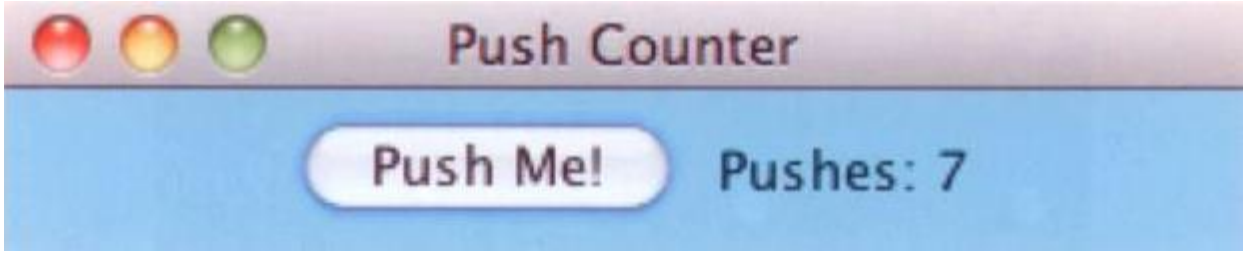
- To create a Java program that uses a GUI, we must:
  - instantiate and set up the necessary components,
  - implement listener classes that define what happens when particular events occur, and
  - establish the relationship between the listeners and the components that generate the events of interest
- Java components and other GUI-related classes are defined primarily in two packages
  - java.awt
  - javax.swing

GUI Elements

- The *Abstract Window Toolkit* (AWT) was the original Java GUI package
  - contains many important classes we will use
- The *Swing* package was added later
  - provides components that are more versatile than those of AWT
- Let's look at a simple example that contains all of the basic GUI elements
  - the example presents the user with a single push button
  - each time the button is pushed, a counter is updated and displayed

PushCounter Example

- Let's look at a simple example that contains all of the basic GUI elements
  - the example presents the user with a single push button
  - each time the button is pushed, a counter is updated and displayed



```
//********************************************************************
//  PushCounter.java       Java Foundations
//
//  Demonstrates a graphical user interface and an event listener.
//********************************************************************

import javax.swing.JFrame;

public class PushCounter
{
   //-----------------------------------------------------------------
   //  Creates and displays the main program frame.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("Push Counter");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      PushCounterPanel panel = new PushCounterPanel();
      frame.getContentPane().add(panel);

      frame.pack();
      frame.setVisible(true);
   }
}
```

```
//********************************************************************
//  PushCounterPanel.java       Java Foundations
//
//  Demonstrates a graphical user interface and an event listener.
//********************************************************************

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

public class PushCounterPanel extends JPanel
{
   private int count;
   private JButton push;
   private JLabel label;

   //-----------------------------------------------------------------
   //  Constructor: Sets up the GUI.
   //-----------------------------------------------------------------
   public PushCounterPanel()
   {
      count = 0;

      push = new JButton("Push Me!");
      push.addActionListener(new ButtonListener());
```

```
      add(push);
      add(label);

      setBackground(Color.cyan);
      setPreferredSize(new Dimension(300, 40));
   }

   //*****************************************************************
   //  Represents a listener for button push (action) events.
   //*****************************************************************
   private class ButtonListener implements ActionListener
   {
      //--------------------------------------------------------------
      //  Updates the counter and label when the button is pushed.
      //--------------------------------------------------------------
      public void actionPerformed(ActionEvent event)
      {
         count++;
         label.setText("Pushes: " + count);
      }
   }
}
```

Frames and Panels

- We classify containers as either
  - *heavyweight* – managed by the underlying operating system
    - examples: a *frame*, an *applet* (used to display and execute a Java application through a Web browser)
  - *lightweight* – managed by the Java program
    - example: *a panel*
    - more complex than heavyweight containers
- We'll often create a GUI-based application by creating a frame the contains a panel

PushCounter Example

- In the `main` method, the frame for the program is constructed, set up, and displayed
- The call to the `setDefaultCloseOperation` method determines what will happen when the close button in the corner of the frame is clicked
- The content pane of the frame is obtained using the `getContentPane` method
- The content pane's `add` method is used to add the panel

PushCounter Example

- The panel contains our button and text label
- A panel's `add` method allows a component to be added to the panel
- A container is governed by a *layout manager*
- The default layout manager for a panel simply displays components in the order they are added, with as many components on one line as possible
- The `pack` method of the frame sets the frame size accordingly

PushCounter Example

- The components:
  - *label* - displays a line of text
    - can be used to also display an image
    - labels are non-interactive
  - *push button* – allows the user to initiate an action with a press of the mouse
    - generates an *action event*
    - different components generate different types of events
- The `ButtonListener` class represents the action listener for the button

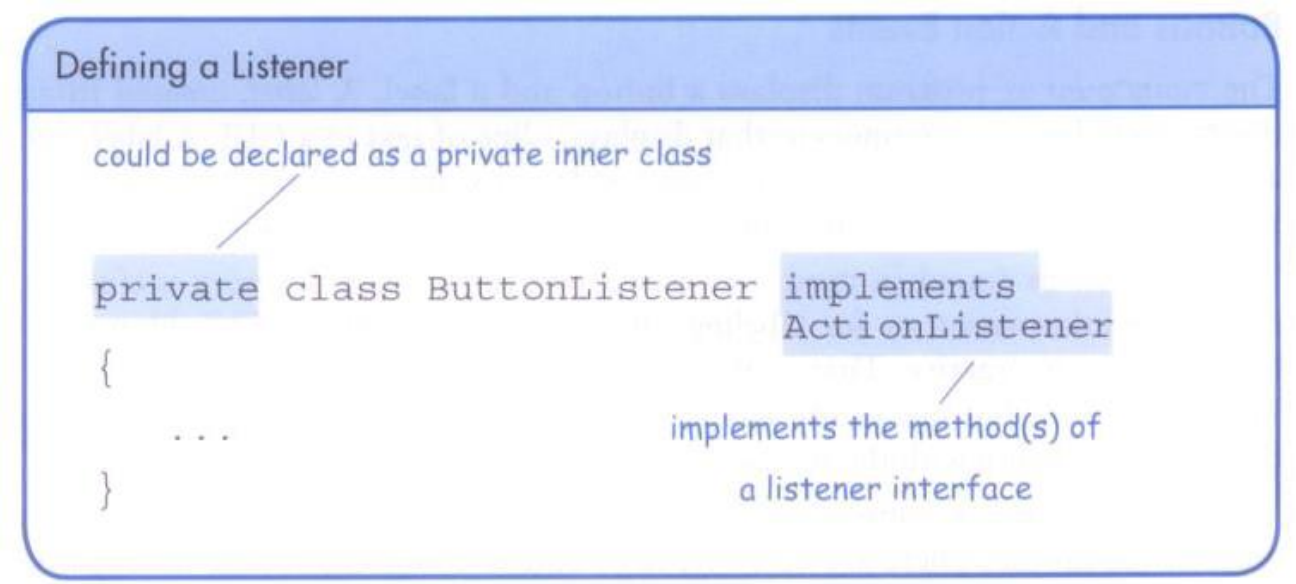
PushCounter Example

- `ButtonListener` class was created as an *inner class* – a class defined inside of another class
- Inner classes have access to the members of the class that contains it
- Listener classes are written by implementing an interface (list of methods the implementing class must implement)
- The `ButtonListener` implements the `ActionListener` interface

PushCounter Example

- `ActionListener` interface
  - only method listed is the `actionPerformed` method
  - the button component generates the action event resulting in a call to the `actionPerformed` method, passing an `ActionEvent` object
- In the example, when the event occurs, the counter is incremented, and the label is updated (via the label's `setText` method)

Defining a Listener



Determining Event Sources

- In the next example, we use one listener to listen to two different components
- The example contains one label and two buttons
  - when the Left button is pushed, the label displays “Left”
  - when the Right button is pushed, the label displays “Right”

Determining Event Sources

- The `LeftRightPanel` class creates one listener and applies it to both buttons
- When either button is pressed, the `actionPerfomed` method of the listener is invoked.
- The `getSource` method returns a reference to the component that generated the event
- We could have created two listener classes. Then the `actionPerformed` method would not have to determine where the event is originating

```
//********************************************************************
//  LeftRight.java       Java Foundations
//
//  Demonstrates the use of one listener for multiple buttons.
//********************************************************************

import javax.swing.JFrame;

public class LeftRight
{
   //-----------------------------------------------------------------
   //  Creates and displays the main program frame.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("Left Right");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      frame.getContentPane().add(new LeftRightPanel());

      frame.pack();
      frame.setVisible(true);
   }
}
```

```
//********************************************************************
//  LeftRightPanel.java       Java Foundations
//
//  Demonstrates the use of one listener for multiple buttons.
//********************************************************************

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

public class LeftRightPanel extends JPanel
{
   private JButton left, right;
   private JLabel label;
   private JPanel buttonPanel;

   //-----------------------------------------------------------------
   //  Constructor: Sets up the GUI.
   //-----------------------------------------------------------------
   public LeftRightPanel()
   {
      left = new JButton("Left");
      right = new JButton("Right");

      ButtonListener listener = new ButtonListener();
      left.addActionListener(listener);
      right.addActionListener(listener);

      label = new JLabel("Push a button");
```

```
      buttonPanel = new JPanel();
      buttonPanel.setPreferredSize(new Dimension(200, 40));
      buttonPanel.setBackground(Color.blue);
      buttonPanel.add(left);
      buttonPanel.add(right);

      setPreferredSize(new Dimension(200, 80));
      setBackground(Color.cyan);
      add(label);
      add(buttonPanel);
   }

   //*****************************************************************
   //  Represents a listener for both buttons.
   //*****************************************************************
   private class ButtonListener implements ActionListener
   {
      //--------------------------------------------------------------
      //  Determines which button was pressed and sets the label
      //  text accordingly.
      //--------------------------------------------------------------
      public void actionPerformed(ActionEvent event)
      {
         if (event.getSource() == left)
            label.setText("Left");
         else
            label.setText("Right");
      }
   }
}
```

More Components

- In addition to push buttons, there are variety of other interactive components
  - *text fields* – allows the user to enter typed input from the keyboard
  - *check boxes* – a button that can be toggled on or off using the mouse (indicates a boolean value is set or unset)
  - *radio buttons* – used with other radio buttons to provide a set of mutually exclusive options
  - *sliders* – allows the user to specify a numeric value within a bounded range
  - *combo boxes* – allows the user to select one of several options from a "drop down" menu
  - *timers* – helps us manage an activity over time, has no visual representation

Text Fields

- A text field generates an action event when the Enter or Return key is pressed (and the cursor is in the field)
- Note that the push button and the text field generate the same kind of event – an action event
- An alternative implementation could involve adding a push button to the panel which causes the conversion to occur when the user pushes the button

Fahrenheit Example

- The Fahrenheit example uses three labels and text field
- When a temperature is entered in the text field (and the return button pressed), the corresponding Celsius temperature is displayed

```
//********************************************************************
//  Fahrenheit.java       Java Foundations
//
//  Demonstrates the use of text fields.
//********************************************************************

import javax.swing.JFrame;

public class Fahrenheit
{
   //-----------------------------------------------------------------
   //  Creates and displays the temperature converter GUI.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("Fahrenheit");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      FahrenheitPanel panel = new FahrenheitPanel();
      frame.getContentPane().add(panel);

      frame.pack();
      frame.setVisible(true);
   }
}
```

```
//********************************************************************
//  FahrenheitPanel.java       Java Foundations
//
//  Demonstrates the use of text fields.
//********************************************************************

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

public class FahrenheitPanel extends JPanel
{
   private JLabel inputLabel, outputLabel, resultLabel;
   private JTextField fahrenheit;

   //-----------------------------------------------------------------
   //  Constructor: Sets up the main GUI components.
   //-----------------------------------------------------------------
   public FahrenheitPanel()
   {
      inputLabel = new JLabel("Enter Fahrenheit temperature:");
      outputLabel = new JLabel("Temperature in Celsius: ");
      resultLabel = new JLabel("---");

      fahrenheit = new JTextField(5);
      fahrenheit.addActionListener(new TempListener());
```

```
      add(inputLabel);
      add(fahrenheit);
      add(outputLabel);
      add(resultLabel);

      setPreferredSize(new Dimension(300, 75));
      setBackground(Color.yellow);
   }

   //*****************************************************************
   //  Represents an action listener for the temperature input field.
   //*****************************************************************
   private class TempListener implements ActionListener
   {
      //--------------------------------------------------------------
      //  Performs the conversion when the enter key is pressed in
      //  the text field.
      //--------------------------------------------------------------
      public void actionPerformed(ActionEvent event)
      {
         int fahrenheitTemp, celsiusTemp;

         String text = fahrenheit.getText();

         fahrenheitTemp = Integer.parseInt(text);
         celsiusTemp = (fahrenheitTemp-32) * 5/9;

         resultLabel.setText(Integer.toString(celsiusTemp));
      }
   }
}
```

Check Boxes

- A check box generates an *item event* when it changes state from selected (checked) to deselected (unchecked) and vice versa
- The `JCheckBox` class is used to define check boxes
- In the example, we use the same listener to handle both check boxes (bold and italic)
- The example also uses the `Font` class to display and change the text label
- The font style is represented as an integer using constants defined in the class

StyleOptions Example

- The phrase is displayed in regular font style, bold, italic, or both, depending on which check boxes are selected

```
//********************************************************************
//  StyleOptions.java       Java Foundations
//
//  Demonstrates the use of check boxes.
//********************************************************************

import javax.swing.JFrame;

public class StyleOptions
{
   //-----------------------------------------------------------------
   //  Creates and displays the style options frame.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("Style Options");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      frame.getContentPane().add(new StyleOptionsPanel());

      frame.pack();
      frame.setVisible(true);
   }
}
```

```
//********************************************************************
//  StyleOptionsPanel.java       Java Foundations
//
//  Demonstrates the use of check boxes.
//********************************************************************

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

public class StyleOptionsPanel extends JPanel
{
   private JLabel saying;
   private JCheckBox bold, italic;

   //-----------------------------------------------------------------
   //  Sets up a panel with a label and some check boxes that
   //  control the style of the label's font.
   //-----------------------------------------------------------------
   public StyleOptionsPanel()
   {
      saying = new JLabel("Say it with style!");
      saying.setFont(new Font("Helvetica", Font.PLAIN, 36));

      bold = new JCheckBox("Bold");
      bold.setBackground(Color.cyan);
      italic = new JCheckBox("Italic");
      italic.setBackground(Color.cyan);

      StyleListener listener = new StyleListener();
      bold.addItemListener(listener);
      italic.addItemListener(listener);
```

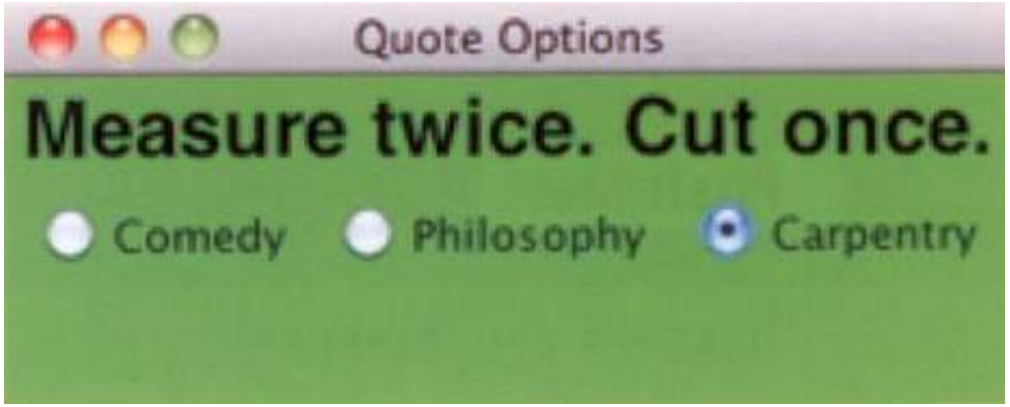
```
      add(saying);
      add(bold);
      add(italic);

      setBackground(Color.cyan);
      setPreferredSize(new Dimension(300, 100));
   }

   //*****************************************************************
   //  Represents the listener for both check boxes.
   //*****************************************************************
   private class StyleListener implements ItemListener
   {
      //--------------------------------------------------------------
      //  Updates the style of the label font style.
      //--------------------------------------------------------------
      public void itemStateChanged(ItemEvent event)
      {
         int style = Font.PLAIN;

         if (bold.isSelected())
            style = Font.BOLD;

         if (italic.isSelected())
            style += Font.ITALIC;

         saying.setFont(new Font("Helvetica", style, 36));
      }
   }
}
```

Radio Buttons

- A *radio button* is used with other radio buttons to provide a set of mutually exclusive options
- Radio buttons have meaning only when used with one or more other radio buttons
- At any point in time, only one button of the group is selected (on)
- Radio buttons produce an action event when selected
- Radio buttons are defined by the `JRadioButton` class
- The `ButtonGroup` class is used to define a set of related radio buttons

QuoteOptions Example

- A different quote is displayed depending on which radio button is selected
- Since only one quote is displayed at any time, mutually-exclusive radio buttons are used



```
//********************************************************************
//  QuoteOptions.java       Java Foundations
//
//  Demonstrates the use of radio buttons.
//********************************************************************

import javax.swing.JFrame;

public class QuoteOptions
{
   //-----------------------------------------------------------------
   //  Creates and presents the program frame.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("Quote Options");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      frame.getContentPane().add(new QuoteOptionsPanel());

      frame.pack();
      frame.setVisible(true);
   }
}
```

```
//********************************************************************
//  QuoteOptionsPanel.java       Java Foundations
//
//  Demonstrates the use of radio buttons.
//********************************************************************

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

public class QuoteOptionsPanel extends JPanel
{
   private JLabel quote;
   private JRadioButton comedy, philosophy, carpentry;
   private String comedyQuote, philosophyQuote, carpentryQuote;

   //-----------------------------------------------------------------
   //  Sets up a panel with a label and a set of radio buttons
   //  that control its text.
   //-----------------------------------------------------------------
   public QuoteOptionsPanel()
   {
      comedyQuote = "Take my wife, please.";
      philosophyQuote = "I think, therefore I am.";
      carpentryQuote = "Measure twice. Cut once.";

      quote = new JLabel(comedyQuote);
      quote.setFont(new Font("Helvetica", Font.BOLD, 24));
```

```
        comedy = new JRadioButton("Comedy", true);
        comedy.setBackground(Color.green);
        philosophy = new JRadioButton("Philosophy");
        philosophy.setBackground(Color.green);
        carpentry = new JRadioButton("Carpentry");
        carpentry.setBackground(Color.green);

        ButtonGroup group = new ButtonGroup();
        group.add(comedy);
        group.add(philosophy);
        group.add(carpentry);

        QuoteListener listener = new QuoteListener();
        comedy.addActionListener(listener);
        philosophy.addActionListener(listener);
        carpentry.addActionListener(listener);

        add(quote);
        add(comedy);
        add(philosophy);
        add(carpentry);

        setBackground(Color.green);
        setPreferredSize(new Dimension(300, 100));
    }
```

```
   //*****************************************************************
   //  Represents the listener for all radio buttons
   //*****************************************************************
   private class QuoteListener implements ActionListener
   {
      //--------------------------------------------------------------
      //  Sets the text of the label depending on which radio
      //  button was pressed.
      //--------------------------------------------------------------
      public void actionPerformed(ActionEvent event)
      {
         Object source = event.getSource();

         if (source == comedy)
            quote.setText(comedyQuote);
         else
            if (source == philosophy)
               quote.setText(philosophyQuote);
            else
               quote.setText(carpentryQuote);
      }
   }
}
```

Sliders

- *Sliders* allow the user to specify a numeric value within a bounded range
- A slider can be presented either vertically or horizontally
- Optional features include
  - tick marks on the slider
  - labels indicating the range of values
- A slider produces a *change event*, indicating that the position of the slider and the value it represents has changed
- A slider is defined by the `JSlider` class

SlideColors Example

- The color corresponding to the values of the three sliders is shown in the small panel on the right

```
//********************************************************************
//  SlideColor.java       Java Foundations
//
//  Demonstrates the use slider components.
//********************************************************************

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

public class SlideColor
{
   //-----------------------------------------------------------------
   //  Presents a frame with a control panel and a panel that
   //  changes color as the sliders are adjusted.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("Slide Colors");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      frame.getContentPane().add(new SlideColorPanel());

      frame.pack();
      frame.setVisible(true);
   }
}
```

```
//********************************************************************
//  SlideColorPanel.java       Java Foundations
//
//  Represents the slider control panel for the SlideColor program.
//********************************************************************

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

public class SlideColorPanel extends JPanel
{
   private JPanel controls, colorPanel;
   private JSlider rSlider, gSlider, bSlider;
   private JLabel rLabel, gLabel, bLabel;

   //-----------------------------------------------------------------
   //  Sets up the sliders and their labels, aligning them along
   //  their left edge using a box layout.
   //-----------------------------------------------------------------
   public SlideColorPanel()
   {
      rSlider = new JSlider(JSlider.HORIZONTAL, 0, 255, 0);
      rSlider.setMajorTickSpacing(50);
      rSlider.setMinorTickSpacing(10);
      rSlider.setPaintTicks(true);
      rSlider.setPaintLabels(true);
      rSlider.setAlignmentX(Component.LEFT_ALIGNMENT);
```

```
gSlider = new JSlider(JSlider.HORIZONTAL, 0, 255, 0);
gSlider.setMajorTickSpacing(50);
gSlider.setMinorTickSpacing(10);
gSlider.setPaintTicks(true);
gSlider.setPaintLabels(true);
gSlider.setAlignmentX(Component.LEFT_ALIGNMENT);

bSlider = new JSlider(JSlider.HORIZONTAL, 0, 255, 0);
bSlider.setMajorTickSpacing(50);
bSlider.setMinorTickSpacing(10);
bSlider.setPaintTicks(true);
bSlider.setPaintLabels(true);
bSlider.setAlignmentX(Component.LEFT_ALIGNMENT);

SliderListener listener = new SliderListener();
rSlider.addChangeListener(listener);
gSlider.addChangeListener(listener);
bSlider.addChangeListener(listener);

rLabel = new JLabel("Red: 0");
rLabel.setAlignmentX(Component.LEFT_ALIGNMENT);
gLabel = new JLabel("Green: 0");
gLabel.setAlignmentX(Component.LEFT_ALIGNMENT);
bLabel = new JLabel("Blue: 0");
bLabel.setAlignmentX(Component.LEFT_ALIGNMENT);

controls = new JPanel();
BoxLayout layout = new BoxLayout(controls, BoxLayout.Y_AXIS);
controls.setLayout(layout);
```

```
      controls.add(rLabel);
      controls.add(rSlider);
      controls.add(Box.createRigidArea(new Dimension (0, 20)));
      controls.add(gLabel);
      controls.add(gSlider);
      controls.add(Box.createRigidArea(new Dimension (0, 20)));
      controls.add(bLabel);
      controls.add(bSlider);

      colorPanel = new JPanel();
      colorPanel.setPreferredSize(new Dimension(100, 100));
      colorPanel.setBackground(new Color(0, 0, 0));

      add(controls);
      add(colorPanel);
   }

   //*****************************************************************
   //  Represents the listener for all three sliders.
   //*****************************************************************
   private class SliderListener implements ChangeListener
   {
      private int red, green, blue;
```

```
      //-----------------------------------------------------------------
      //  Gets the value of each slider, then updates the labels and
      //  the color panel.
      //-----------------------------------------------------------------
      public void stateChanged(ChangeEvent event)
      {
         red = rSlider.getValue();
         green = gSlider.getValue();
         blue = bSlider.getValue();

         rLabel.setText("Red: " + red);
         gLabel.setText("Green: " + green);
         bLabel.setText("Blue: " + blue);

         colorPanel.setBackground(new Color(red, green, blue));
      }
   }
}
```

Combo Boxes

- A *combo box* allows a user to select one of several options from a "drop down" menu
- When the user presses a combo box using a mouse, a list of options is displayed from which the user can choose
- A combo box is defined by the `JComboBox` class
- Combo boxes generate an action event whenever the user makes a selection from it

JavaJukeBox Example

- The user can select a song using the combo box, then play and stop the song using buttons

```
//********************************************************************
//  JukeBox.java       Java Foundations
//
//  Demonstrates the use of a combo box.
//********************************************************************

import javax.swing.*;

public class JukeBox
{
   //-----------------------------------------------------------------
   //  Creates and displays the controls for a juke box.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("Java Juke Box");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      frame.getContentPane().add(new JukeBoxControls());

      frame.pack();
      frame.setVisible(true);
   }
}
```

```
//********************************************************************
//  JukeBoxControls.java       Java Foundations
//
//  Represents the control panel for the juke box.
//********************************************************************

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.applet.AudioClip;
import java.net.URL;

public class JukeBoxControls extends JPanel
{
   private JComboBox musicCombo;
   private JButton stopButton, playButton;
   private AudioClip[] music;
   private AudioClip current;

   //-----------------------------------------------------------------
   //  Sets up the GUI for the juke box.
   //-----------------------------------------------------------------
   public JukeBoxControls()
   {
      URL url1, url2, url3, url4, url5, url6;
      url1 = url2 = url3 = url4 = url5 = url6 = null;
```

```
// Obtain and store the audio clips to play
try
{
   url1 = new URL("file", "localhost", "westernBeat.wav");
   url2 = new URL("file", "localhost", "classical.wav");
   url3 = new URL("file", "localhost", "jeopardy.au");
   url4 = new URL("file", "localhost", "newAgeRythm.wav");
   url5 = new URL("file", "localhost", "eightiesJam.wav");
   url6 = new URL("file", "localhost", "hitchcock.wav");
}
catch (Exception exception) {}

music = new AudioClip[7];
music[0] = null;  // Corresponds to "Make a Selection..."
music[1] = JApplet.newAudioClip(url1);
music[2] = JApplet.newAudioClip(url2);
music[3] = JApplet.newAudioClip(url3);
music[4] = JApplet.newAudioClip(url4);
music[5] = JApplet.newAudioClip(url5);
music[6] = JApplet.newAudioClip(url6);

// Create the list of strings for the combo box options
String[] musicNames = {"Make A Selection...", "Western Beat",
         "Classical Melody", "Jeopardy Theme", "New Age Rythm",
         "Eighties Jam", "Alfred Hitchcock's Theme"};

musicCombo = new JComboBox(musicNames);
musicCombo.setBackground(Color.cyan);
```

```
    //  Set up the buttons
    playButton = new JButton("Play", new ImageIcon("play.gif"));
    playButton.setBackground(Color.cyan);
    stopButton = new JButton("Stop", new ImageIcon("stop.gif"));
    stopButton.setBackground(Color.cyan);

    //  Set up this panel
    setPreferredSize(new Dimension (250, 100));
    setBackground(Color.cyan);
    add(musicCombo);
    add(playButton);
    add(stopButton);

    musicCombo.addActionListener(new ComboListener());
    stopButton.addActionListener(new ButtonListener());
    playButton.addActionListener(new ButtonListener());

    current = null;
}
```

```
//*****************************************************************
//  Represents the action listener for the combo box.
//*****************************************************************
private class ComboListener implements ActionListener
{
   //--------------------------------------------------------------
   //  Stops playing the current selection (if any) and resets
   //  the current selection to the one chosen.
   //--------------------------------------------------------------
   public void actionPerformed(ActionEvent event)
   {
      if (current != null)
         current.stop();

      current = music[musicCombo.getSelectedIndex()];
   }
 }
```

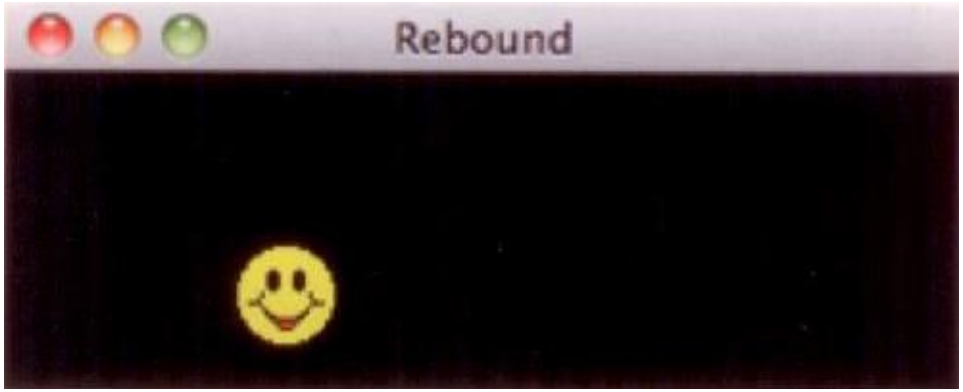
```
   //*****************************************************************
   //  Represents the action listener for both control buttons.
   //*****************************************************************
   private class ButtonListener implements ActionListener
   {
      //--------------------------------------------------------------
      //  Stops the current selection (if any) in either case. If
      //  the play button was pressed, start playing it again.
      //--------------------------------------------------------------
      public void actionPerformed(ActionEvent event)
      {
         if (current != null)
            current.stop();

         if (event.getSource() == playButton)
            if (current != null)
               current.play();
      }
   }
}
```

Timers

- *Timers* do not have a visual representation, but are a AWT component
- Timers are defined by the `Timer` class and are provided to help manage an activity over time
- A timer object generates an action event at regular intervals

Rebound Example

- When executing, the image bounces around the panel
- Whenever the timer expires, the image's position is updated, creating the illusion of movement



```
//********************************************************************
//  Rebound.java       Java Foundations
//
//  Demonstrates an animation and the use of the Timer class.
//********************************************************************

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

public class Rebound
{
   //-----------------------------------------------------------------
   //  Displays the main frame of the program.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("Rebound");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      frame.getContentPane().add(new ReboundPanel());

      frame.pack();
      frame.setVisible(true);
   }
}
```

```
//********************************************************************
//  ReboundPanel.java       Java Foundations
//
//  Represents the primary panel for the Rebound program.
//********************************************************************

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

public class ReboundPanel extends JPanel
{
   private final int WIDTH = 300, HEIGHT = 100;
   private final int DELAY = 20, IMAGE_SIZE = 35;

   private ImageIcon image;
   private Timer timer;
   private int x, y, moveX, moveY;

   //-----------------------------------------------------------------
   //  Sets up the panel, including the timer for the animation.
   //-----------------------------------------------------------------
   public ReboundPanel()
   {
      timer = new Timer(DELAY, new ReboundListener());

      image = new ImageIcon("happyFace.gif");
```

```
      x = 0;
      y = 40;
      moveX = moveY = 3;

      setPreferredSize(new Dimension(WIDTH, HEIGHT));
      setBackground(Color.black);
      timer.start();
   }

   //-----------------------------------------------------------------
   //  Draws the image in the current location.
   //-----------------------------------------------------------------
   public void paintComponent(Graphics page)
   {
      super.paintComponent(page);
      image.paintIcon(this, page, x, y);
   }
```

```
   //*****************************************************************
   //  Represents the action listener for the timer.
   //*****************************************************************
   private class ReboundListener implements ActionListener
   {
      //--------------------------------------------------------------
      //  Updates the position of the image and possibly the direction
      //  of movement whenever the timer fires an action event.
      //--------------------------------------------------------------
      public void actionPerformed(ActionEvent event)
      {
         x += moveX;
         y += moveY;

         if (x <= 0 || x >= WIDTH-IMAGE_SIZE)
            moveX = moveX * -1;

         if (y <= 0 || y >= HEIGHT-IMAGE_SIZE)
            moveY = moveY * -1;

         repaint();
      }
   }
}
```

Layout Managers

- Every container is managed by an object known as a *layout manager* that determines how the components in the container are arranged visually
- The layout manager is consulted when needed, such as when the container is resized or when a component is added
- Every container has a default layout manager, but we can replace it if desired
- A layout manager determines the size and position of each component

Layout Managers

- Some of the layout managers defined in the Java API:

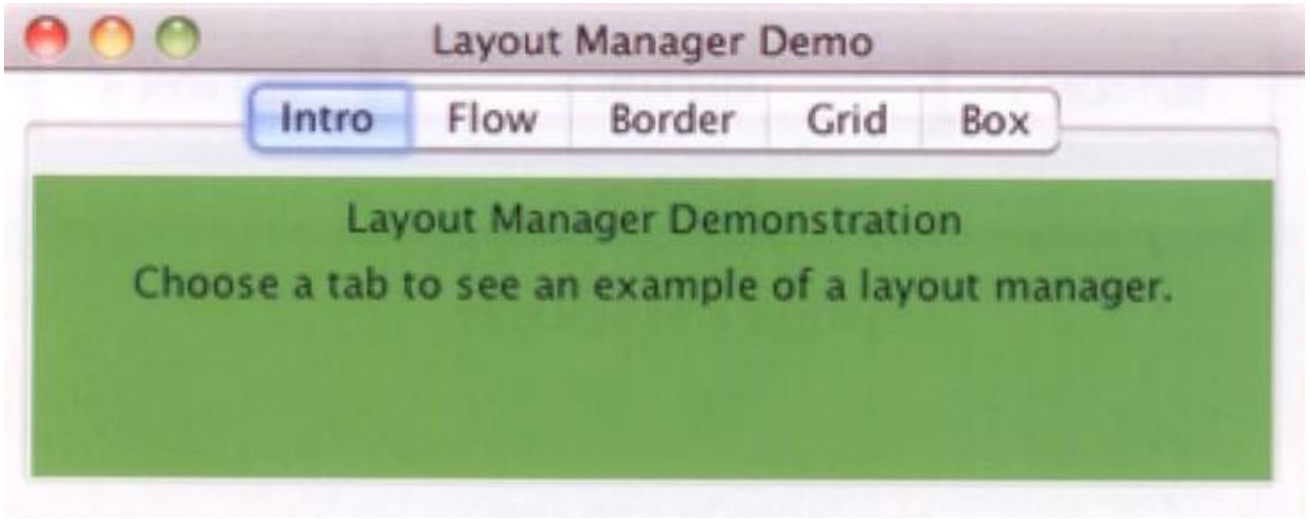
Layout Manager	Description
Border Layout	Organizes components into five areas (North, South, East, West, and Center).
Box Layout	Organizes components into a single row or column.
Card Layout	Organizes components into one area such that only one is visible at any time.
Flow Layout	Organizes components from left to right, starting new rows as necessary.
Grid Layout	Organizes components into a grid of rows and columns.
GridBag Layout	Organizes components into a grid of cells, allowing components to span more than one cell.

Layout Managers

- Every layout manager has its own rules and properties governing the layout of the components it contains
- For some layout managers, the order in which you add the components affects their positioning
- We use the `setLayout` method of a container to change its layout manager

LayoutDemo Example

- One program, using a tabbed pane, is used to show the results of various layout managers

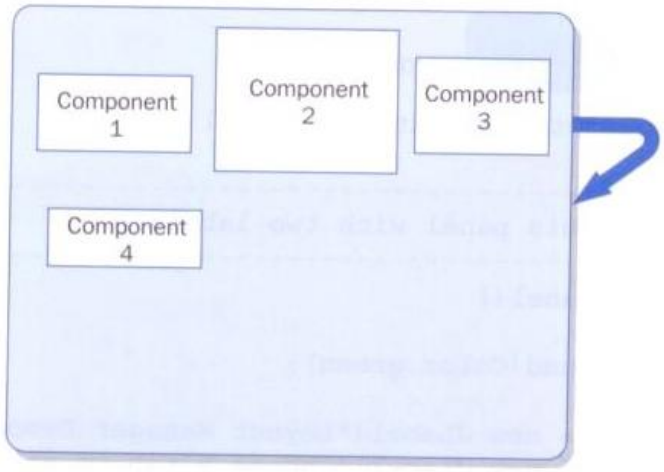


```
//********************************************************************
//  LayoutDemo.java       Java Foundations
//
//  Demonstrates the use of flow, border, grid, and box layouts.
//********************************************************************

import javax.swing.*;

public class LayoutDemo
{
   //-----------------------------------------------------------------
   //  Sets up a frame containing a tabbed pane. The panel on each
   //  tab demonstrates a different layout manager.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("Layout Manager Demo");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      JTabbedPane tp = new JTabbedPane();
      tp.addTab("Intro", new IntroPanel());
      tp.addTab("Flow", new FlowPanel());
      tp.addTab("Border", new BorderPanel());
      tp.addTab("Grid", new GridPanel());
      tp.addTab("Box", new BoxPanel());

      frame.getContentPane().add(tp);

      frame.pack();
      frame.setVisible(true);
   }
}
```

```
//********************************************************************
//  IntroPanel.java       Java Foundations
//
//  Represents the introduction panel for the LayoutDemo program.
//********************************************************************

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

public class IntroPanel extends JPanel
{
   //-----------------------------------------------------------------
   //  Sets up this panel with two labels.
   //-----------------------------------------------------------------
   public IntroPanel()
   {
      setBackground(Color.green);

      JLabel l1 = new JLabel("Layout Manager Demonstration");
      JLabel l2 = new JLabel("Choose a tab to see an example of " +
                             "a layout manager.");
      add(l1);
      add(l2);
   }
}
```

Flow Layout

- The `JPanel` class uses flow layout by default
- Puts as many components as possible on a row, at their preferred size
- When a component can not fit on a row, it is put on the next row



Flow Layout

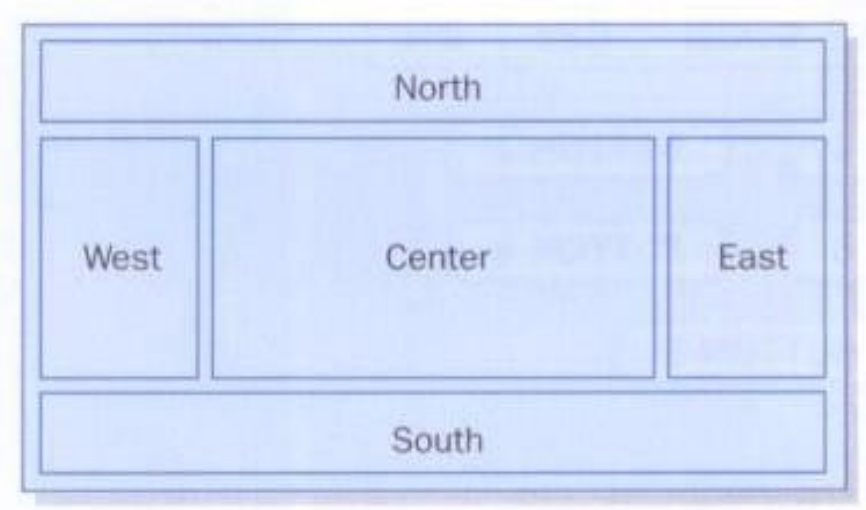
- When the window is resized, the layout manager automatically repositions the buttons

```
//********************************************************************
//  FlowPanel.java       Java Foundations
//
//  Represents the panel in the LayoutDemo program that demonstrates
//  the flow layout manager.
//********************************************************************

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

public class FlowPanel extends JPanel
{
   //-----------------------------------------------------------------
   //  Sets up this panel with some buttons to show how flow layout
   //  affects their position.
   //-----------------------------------------------------------------
   public FlowPanel()
   {
      setLayout(new FlowLayout());

      setBackground(Color.green);

      JButton b1 = new JButton("BUTTON 1");
      JButton b2 = new JButton("BUTTON 2");
      JButton b3 = new JButton("BUTTON 3");
      JButton b4 = new JButton("BUTTON 4");
      JButton b5 = new JButton("BUTTON 5");
```

```
        add(b1);
        add(b2);
        add(b3);
        add(b4);
        add(b5);
    }
}
```

Border Layout

- A *border layout* has five areas to which components can be added: North, South, East, West, and Center



Border Layout

- The four outer areas are as large as needed in order to accommodate the component they contain
- If no components are added to a region, the region takes up no room in the overall layout
- The Center area expands to fill any available space
- The size of the gaps between the areas can be adjusted

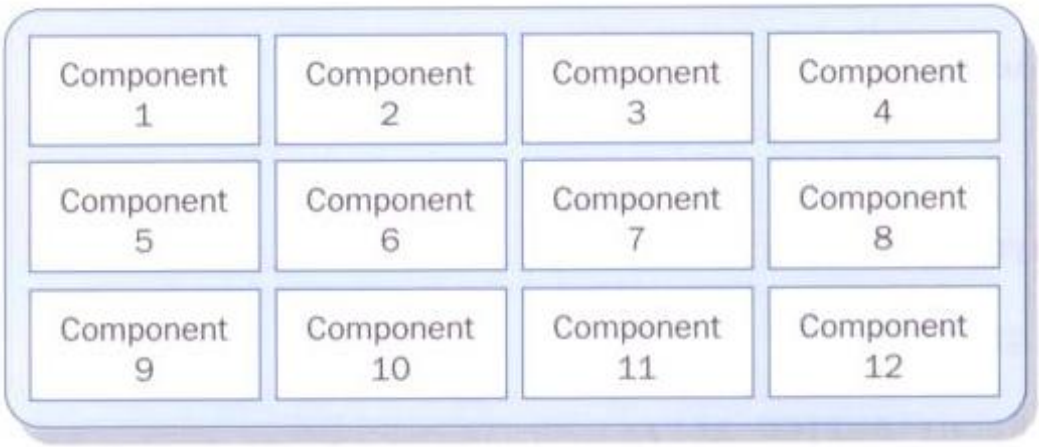
Border Layout

```
//********************************************************************
//  BorderPanel.java       Java Foundations
//
//  Represents the panel in the LayoutDemo program that demonstrates
//  the border layout manager.
//********************************************************************

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

public class BorderPanel extends JPanel
{
   //-----------------------------------------------------------------
   //  Sets up this panel with a button in each area of a border
   //  layout to show how it affects their position, shape, and size.
   //-----------------------------------------------------------------
   public BorderPanel()
   {
      setLayout(new BorderLayout());

      setBackground(Color.green);

      JButton b1 = new JButton("BUTTON 1");
      JButton b2 = new JButton("BUTTON 2");
      JButton b3 = new JButton("BUTTON 3");
      JButton b4 = new JButton("BUTTON 4");
      JButton b5 = new JButton("BUTTON 5");
```

```
        add(b1, BorderLayout.CENTER);
        add(b2, BorderLayout.NORTH);
        add(b3, BorderLayout.SOUTH);
        add(b4, BorderLayout.EAST);
        add(b5, BorderLayout.WEST);
    }
}
```

Grid Layout

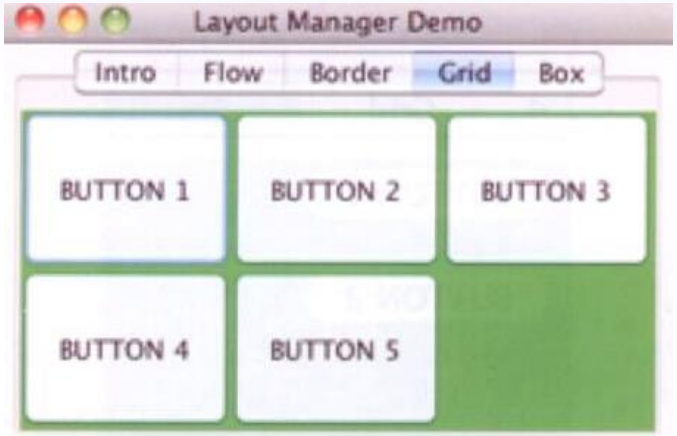
- A *grid layout* presents a container's components in a rectangular grid of rows and columns
- One component is placed in each cell, and all cells are the same size

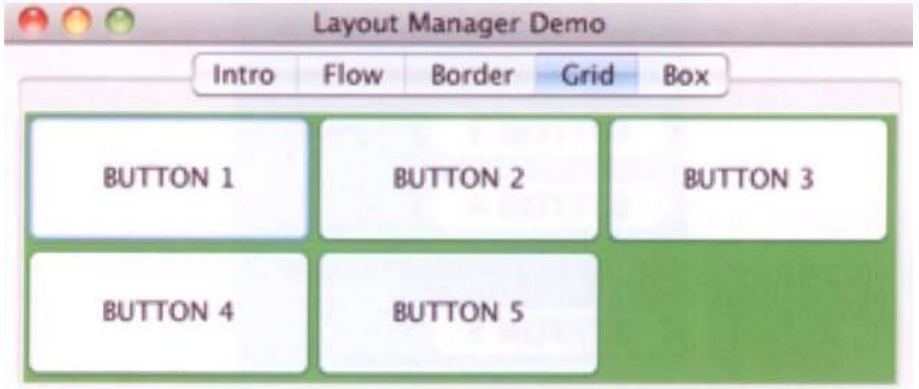


Grid Layout

- The number of rows and columns in a grid layout is established by using parameters to the constructor when the layout manager is created
- As components are added to the grid layout, they fill the grid from left to right, top to bottom
- There is no way to explicitly assign a component to a particular location in the grid other than the order in which they are added to the container

Grid Layout


Layout Manager Demo
Intro
Flow
Border
Grid
Box
BUTTON 1
BUTTON 2
BUTTON 3
BUTTON 4
BUTTON 5

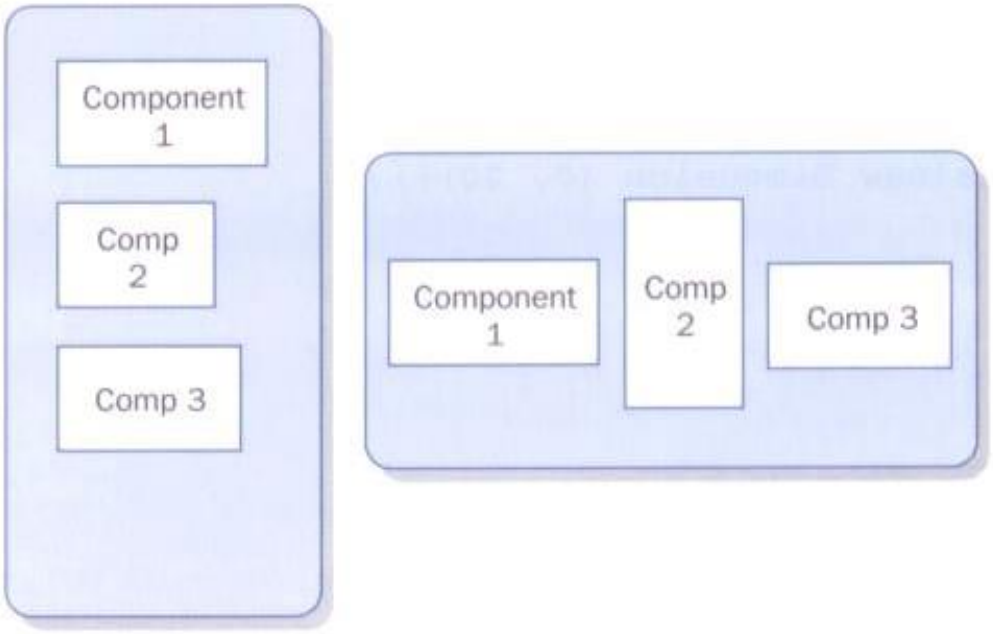

Layout Manager Demo
Intro
Flow
Border
Grid
Box
BUTTON 1
BUTTON 2
BUTTON 3
BUTTON 4
BUTTON 5

```
//********************************************************************
//  GridPanel.java       Java Foundations
//
//  Represents the panel in the LayoutDemo program that demonstrates
//  the grid layout manager.
//********************************************************************

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

public class GridPanel extends JPanel
{
   //-----------------------------------------------------------------
   //  Sets up this panel with some buttons to show how grid
   //  layout affects their position, shape, and size.
   //-----------------------------------------------------------------
   public GridPanel()
   {
      setLayout(new GridLayout(2, 3));

      setBackground(Color.green);

      JButton b1 = new JButton("BUTTON 1");
      JButton b2 = new JButton("BUTTON 2");
      JButton b3 = new JButton("BUTTON 3");
      JButton b4 = new JButton("BUTTON 4");
      JButton b5 = new JButton("BUTTON 5");
```

```
        add(b1);
        add(b2);
        add(b3);
        add(b4);
        add(b5);
    }
}
```

Box Layout

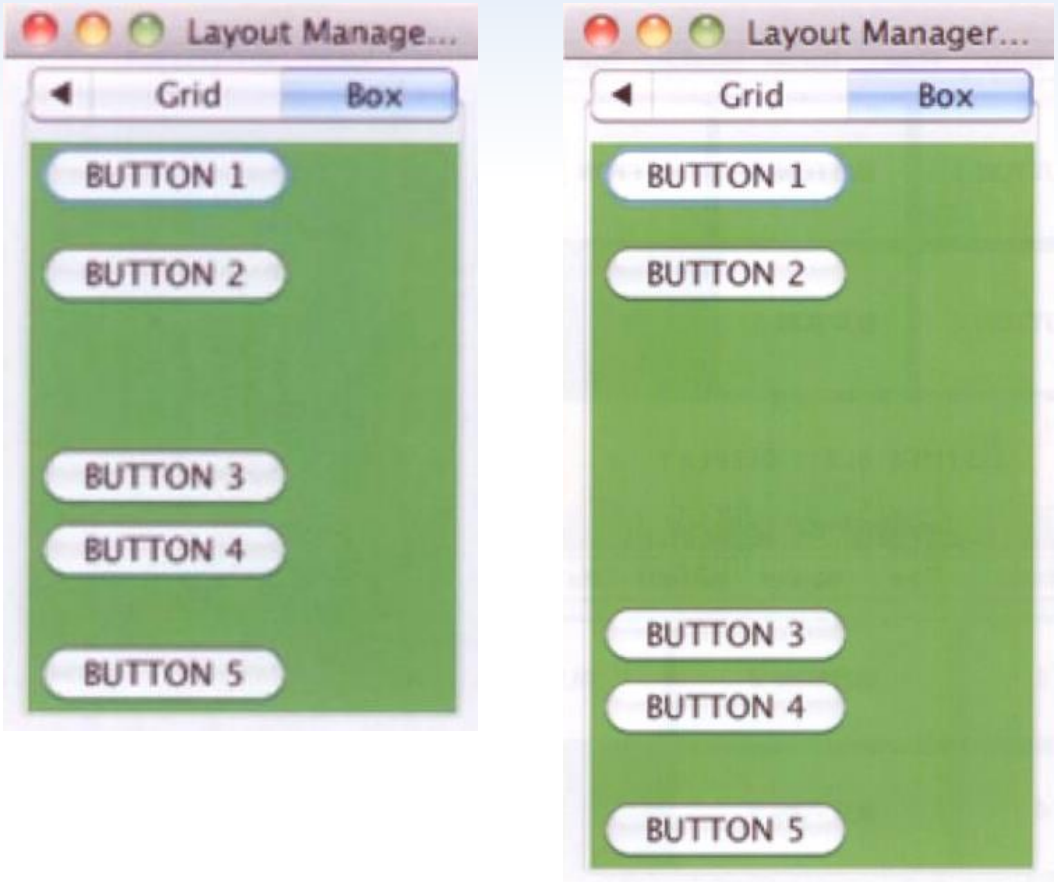
- A *box layout* organizes components either vertically or horizontally, in one row or one column



Box Layout

- When combined with other layout managers, box layout can produce complex GUI designs
- Components are organized in the order in which they are added to the container
- Two types of invisible components can be added to control spacing between other components
  - *rigid areas* – with a fixed size
  - *glue* – specifies where excess space should go as needed

Box Layout


Layout Manage...
Grid
Box
BUTTON 1
BUTTON 2
BUTTON 3
BUTTON 4
BUTTON 5
Layout Manager...
Grid
Box
BUTTON 1
BUTTON 2
BUTTON 3
BUTTON 4
BUTTON 5

```
//********************************************************************
//  BoxPanel.java       Java Foundations
//
//  Represents the panel in the LayoutDemo program that demonstrates
//  the box layout manager.
//********************************************************************

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

public class BoxPanel extends JPanel
{
   //-----------------------------------------------------------------
   //  Sets up this panel with some buttons to show how a vertical
   //  box layout (and invisible components) affects their position.
   //-----------------------------------------------------------------
   public BoxPanel()
   {
      setLayout(new BoxLayout (this, BoxLayout.Y_AXIS));

      setBackground(Color.green);

      JButton b1 = new JButton("BUTTON 1");
      JButton b2 = new JButton("BUTTON 2");
      JButton b3 = new JButton("BUTTON 3");
      JButton b4 = new JButton("BUTTON 4");
      JButton b5 = new JButton("BUTTON 5");
```

```
        add(b1);
        add(Box.createRigidArea(new Dimension (0, 10)));
        add(b2);
        add(Box.createVerticalGlue());
        add(b3);
        add(b4);
        add(Box.createRigidArea(new Dimension (0, 20)));
        add(b5);
    }
}
```

Containment Hierarchies

- The way components are grouped into containers, and the way those containers are nested within each other, establishes the *containment hierarchy* for a GUI
- For any Java GUI program, there is generally one primary (top-level) container, such as a frame or applet
- The top-level container often contains one or more containers, such as panels
- These panels may contain other panels to organize the other components as desired

Mouse and Key Events

- In addition to component events, events are also fired when a user interacts with the computer's mouse and keyboard
- Mouse events
  - *mouse events* – occur when the user interacts with another component via the mouse. To use, implement the `MouseListener` interface class
  - *mouse motion events* – occur while the mouse is in motion. To use, implement the `MouseMotionListener` interface class

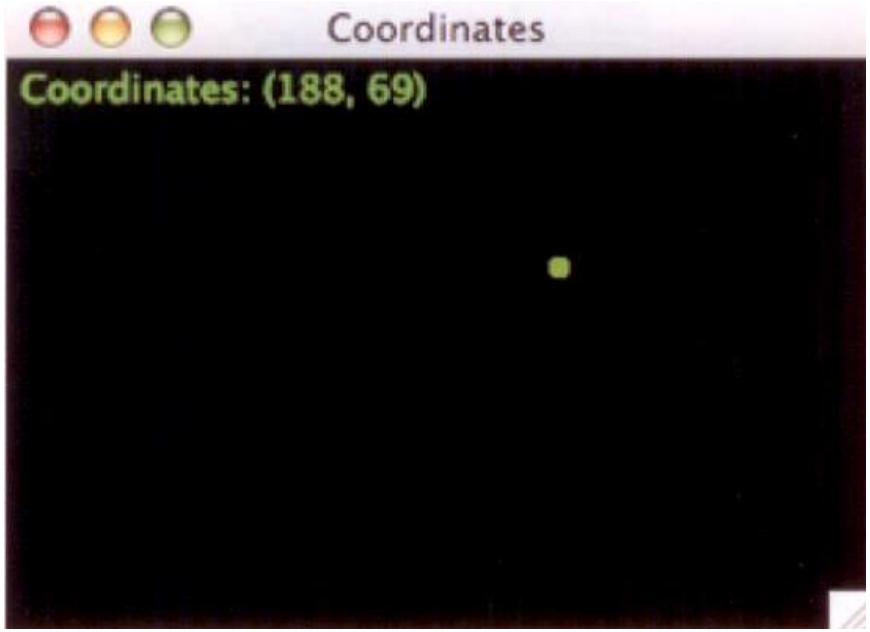
Mouse and Mouse Motion Events

Mouse Event	Description
mouse pressed	The mouse button is pressed down.
mouse released	The mouse button is released.
mouse clicked	The mouse button is pressed down and released without moving the mouse in between.
mouse entered	The mouse pointer is moved onto (over) a component.
mouse exited	The mouse pointer is moved off of a component.

Mouse Motion Event	Description
mouse moved	The mouse is moved.
mouse dragged	The mouse is moved while the mouse button is pressed down.

Coordinates Example

- Clicking the mouse causes a dot to appear in that location and the coordinates to be displayed



```
//********************************************************************
//  Coordinates.java       Java Foundations
//
//  Demonstrates mouse events.
//********************************************************************

import javax.swing.JFrame;

public class Coordinates
{
   //-----------------------------------------------------------------
   //  Creates and displays the application frame.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("Coordinates");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      frame.getContentPane().add(new CoordinatesPanel());

      frame.pack();
      frame.setVisible(true);
   }
}
```

```
//********************************************************************
//  CoordinatesPanel.java       Java Foundations
//
//  Represents the primary panel for the Coordinates program.
//********************************************************************

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

public class CoordinatesPanel extends JPanel
{
   private final int SIZE = 6;  // diameter of dot

   private int x = 50, y = 50;  // coordinates of mouse press

   //-----------------------------------------------------------------
   //  Constructor: Sets up this panel to listen for mouse events.
   //-----------------------------------------------------------------
   public CoordinatesPanel()
   {
      addMouseListener(new CoordinatesListener());

      setBackground(Color.black);
      setPreferredSize(new Dimension(300, 200));
   }
```

```
//-----------------------------------------------------------------
//  Draws all of the dots stored in the list.
//-----------------------------------------------------------------
public void paintComponent(Graphics page)
{
   super.paintComponent(page);

   page.setColor(Color.green);

   page.fillOval(x, y, SIZE, SIZE);

   page.drawString("Coordinates: (" + x + ", " + y + ")", 5, 15);
}

//*****************************************************************
//  Represents the listener for mouse events.
//*****************************************************************
private class CoordinatesListener implements MouseListener
{
   //--------------------------------------------------------------
   //  Adds the current point to the list of points and redraws
   //  the panel whenever the mouse button is pressed.
   //--------------------------------------------------------------
   public void mousePressed(MouseEvent event)
   {
      x = event.getX();
      y = event.getY();
      repaint();
   }
```

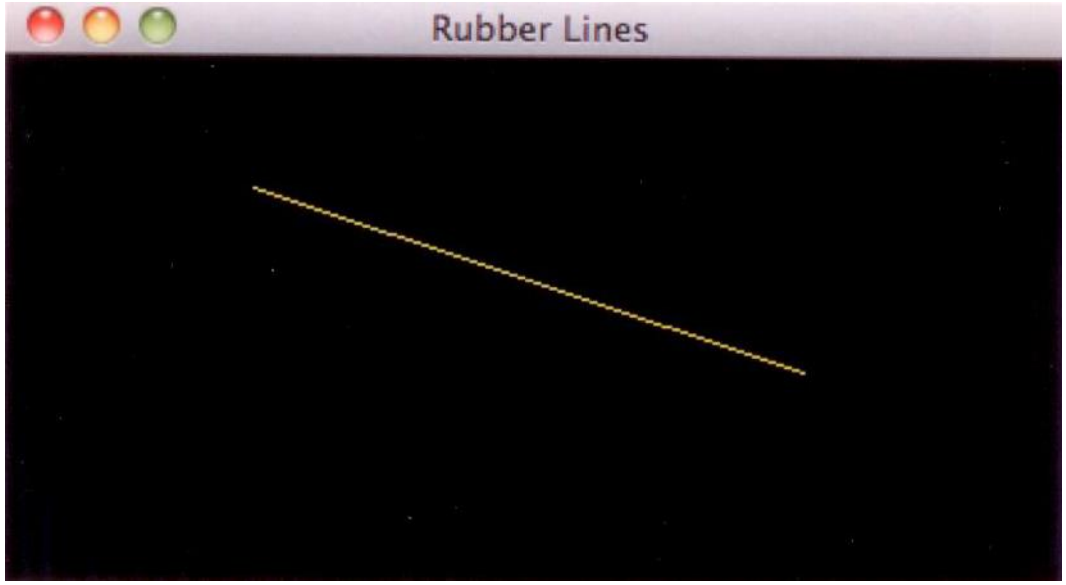
```
      //-----------------------------------------------------------------
      //  Provide empty definitions for unused event methods.
      //-----------------------------------------------------------------
      public void mouseClicked(MouseEvent event) {}
      public void mouseReleased(MouseEvent event) {}
      public void mouseEntered(MouseEvent event) {}
      public void mouseExited(MouseEvent event) {}
   }
}
```

Coordinates Example

- The event object passed to the listener is used to get the coordinates of the event (its been ignored in previous examples)
- Unused methods of the `MouseListener` interface are given empty methods

RubberLines Example

- As the mouse is dragged, the line is redrawn
- This creates a *rubberbanding* effect, as if the line is being pulled into shape



RubberLines Example

- This example uses both mouse and mouse motion events
- The initial click is captured using the mouse pressed event
- Then the line is updated continually using the mouse dragged event

```
//********************************************************************
//  RubberLines.java       Java Foundations
//
//  Demonstrates mouse events and rubberbanding.
//********************************************************************

import javax.swing.JFrame;

public class RubberLines
{
   //-----------------------------------------------------------------
   //  Creates and displays the application frame.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("Rubber Lines");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      frame.getContentPane().add(new RubberLinesPanel());

      frame.pack();
      frame.setVisible(true);
   }
}
```

```
//********************************************************************
//  RubberLinesPanel.java       Java Foundations
//
//  Represents the primary drawing panel for the RubberLines program.
//********************************************************************

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

public class RubberLinesPanel extends JPanel
{
   private Point point1 = null, point2 = null;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this panel to listen for mouse events.
   //-----------------------------------------------------------------
   public RubberLinesPanel()
   {
      LineListener listener = new LineListener();
      addMouseListener(listener);
      addMouseMotionListener(listener);

      setBackground(Color.black);
      setPreferredSize(new Dimension(400, 200));
   }
```

```
//-----------------------------------------------------------------
//  Draws the current line from the intial mouse-pressed point to
//  the current position of the mouse.
//-----------------------------------------------------------------
public void paintComponent(Graphics page)
{
   super.paintComponent(page);

   page.setColor (Color.yellow);
   if (point1 != null && point2 != null)
      page.drawLine(point1.x, point1.y, point2.x, point2.y);
}

//*****************************************************************
//  Represents the listener for all mouse events.
//*****************************************************************
private class LineListener implements MouseListener,
                                      MouseMotionListener
{
   //--------------------------------------------------------------
   //  Captures the initial position at which the mouse button is
   //  pressed.
   //--------------------------------------------------------------
   public void mousePressed(MouseEvent event)
   {
      point1 = event.getPoint();
   }
```

```
      //-----------------------------------------------------------------
      //  Gets the current position of the mouse as it is dragged and
      //  redraws the line to create the rubberband effect.
      //-----------------------------------------------------------------
      public void mouseDragged(MouseEvent event)
      {
         point2 = event.getPoint();
         repaint();
      }

      //-----------------------------------------------------------------
      //  Provide empty definitions for unused event methods.
      //-----------------------------------------------------------------
      public void mouseClicked(MouseEvent event) {}
      public void mouseReleased(MouseEvent event) {}
      public void mouseEntered(MouseEvent event) {}
      public void mouseExited(MouseEvent event) {}
      public void mouseMoved(MouseEvent event) {}
   }
}
```

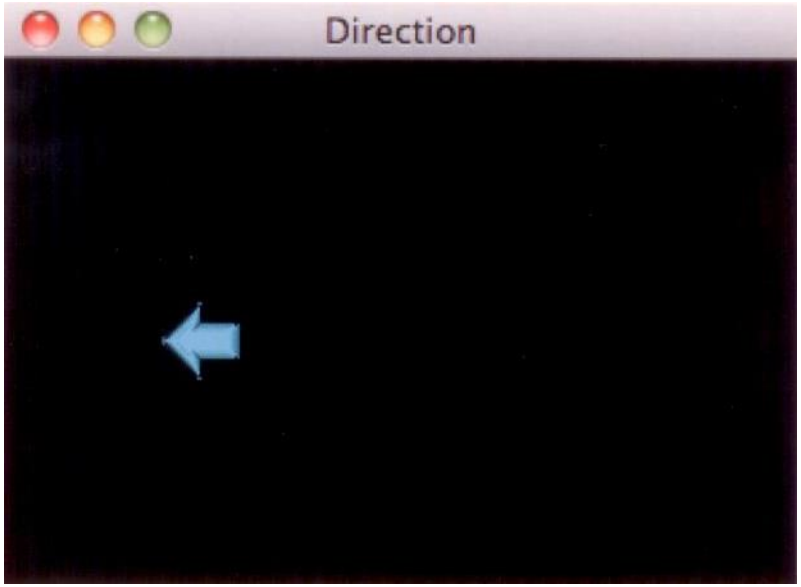
Key Events

- A *key event* is generated when the user presses a keyboard key
- This allows a program to respond immediately to the user while they are typing
- The `KeyListener` interface defines three methods used to respond to keyboard activity

```
void keyPressed (KeyEvent event)
    Called when a key is pressed.

void keyReleased (KeyEvent event)
    Called when a key is released.

void keyTyped (KeyEvent event)
    Called when a pressed key or key combination produces
    a key character.
```

Direction Example

- As the user presses the arrow keys on the keyboard, an arrow image is displayed and moved in the appropriate direction



```
//********************************************************************
//  Direction.java       Java Foundations
//
//  Demonstrates key events.
//********************************************************************

import javax.swing.JFrame;

public class Direction
{
   //-----------------------------------------------------------------
   //  Creates and displays the application frame.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("Direction");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      frame.getContentPane().add(new DirectionPanel());

      frame.pack();
      frame.setVisible(true);
   }
}
```

```
//********************************************************************
//  DirectionPanel.java       Java Foundations
//
//  Represents the primary display panel for the Direction program.
//********************************************************************

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

public class DirectionPanel extends JPanel
{
   private final int WIDTH = 300, HEIGHT = 200;
   private final int JUMP = 10;  // increment for image movement

   private final int IMAGE_SIZE = 31;

   private ImageIcon up, down, right, left, currentImage;
   private int x, y;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this panel and loads the images.
   //-----------------------------------------------------------------
   public DirectionPanel()
   {
      addKeyListener (new DirectionListener());

      x = WIDTH / 2;
      y = HEIGHT / 2;
```

```
      up = new ImageIcon("arrowUp.gif");
      down = new ImageIcon("arrowDown.gif");
      left = new ImageIcon("arrowLeft.gif");
      right = new ImageIcon("arrowRight.gif");

      currentImage = right;

      setBackground(Color.black);
      setPreferredSize(new Dimension(WIDTH, HEIGHT));
      setFocusable(true);
   }

   //-----------------------------------------------------------------
   //  Draws the image in the current location.
   //-----------------------------------------------------------------
   public void paintComponent(Graphics page)
   {
      super.paintComponent(page);
      currentImage.paintIcon(this, page, x, y);
   }
```

```
//*****************************************************************
//  Represents the listener for keyboard activity.
//*****************************************************************
private class DirectionListener implements KeyListener
{
   //--------------------------------------------------------------
   //  Responds to the user pressing arrow keys by adjusting the
   //  image and image location accordingly.
   //--------------------------------------------------------------
   public void keyPressed(KeyEvent event)
   {
      switch (event.getKeyCode())
      {
         case KeyEvent.VK_UP:
            currentImage = up;
            y -= JUMP;
            break;
         case KeyEvent.VK_DOWN:
            currentImage = down;
            y += JUMP;
            break;
         case KeyEvent.VK_LEFT:
            currentImage = left;
            x -= JUMP;
            break;
```

```
            case KeyEvent.VK_RIGHT:
               currentImage = right;
               x += JUMP;
               break;
         }

         repaint();
      }

      //-----------------------------------------------------------------
      //  Provide empty definitions for unused event methods.
      //-----------------------------------------------------------------
      public void keyTyped(KeyEvent event) {}
      public void keyReleased(KeyEvent event) {}
   }
}
```

Extending Adapter Classes

- In our previous examples, we've created the listener classes by implementing a particular listener interface
- An alternative technique for creating a listener class is to use inheritance and extend an *adapter class*
- Each listener interface that contains more than one method has a corresponding adapter class containing empty definitions for all methods in the interface
- We can override any event methods we need in our new child class

Extending Adapter Classes

- For example:
  - The `MouseAdapter` class implements the `MouseListener` interface class and provides empty method definitions for the five mouse event methods
  - By subclassing `MouseAdapter`, we can avoid implementing the interface directly
- This approach can save coding time and keep source code easier to read

Dialog Boxes

- A *dialog box* is a graphical window that pops up on top of any currently active window so that the user can interact with it
- A dialog box can serve a variety of purposes
  - conveying information
  - confirming an action
  - permitting the user to enter information
- The `JOptionPane` class simplifies the creation and use of basic dialog boxes

Dialog Boxes

- `JOptionPane` dialog boxes fall into three categories
  - *message dialog boxes* – used to display an output string
  - *input dialog boxes* – presents a prompt and a single input txt file into which the user can enter one string of data
  - *confirm dialog box* – presents the user with a simple yes-or-no question
- These three types of dialog boxes are created using static methods in the `JOptionPane` class
- Many of the `JOptionPane` methods allow the program to tailor the contents of the dialog box

JOptionPane Methods

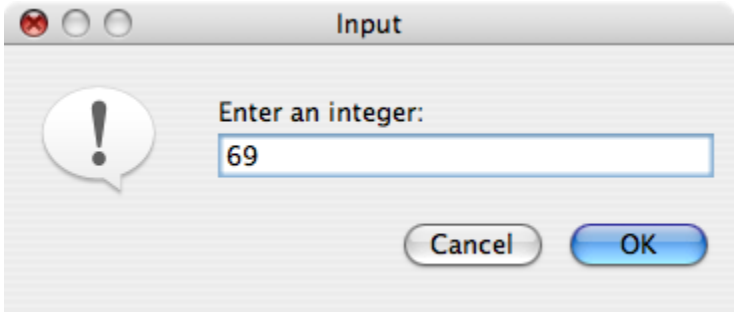
`static String showInputDialog (Object msg)`
Displays a dialog box containing the specified message and an input text field. The contents of the text field are returned.

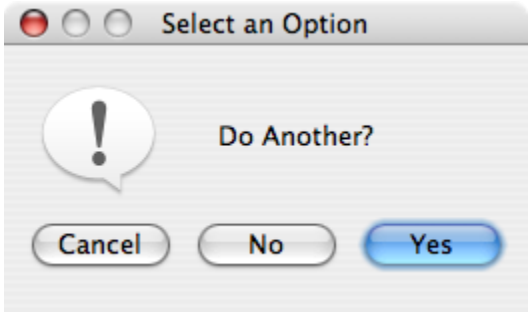
`static int showConfirmDialog (Component parent, Object msg)`
Displays a dialog box containing the specified message and Yes/No button options. If the parent component is null, the box is centered on the screen.

`static void showMessageDialog (Component parent, Object msg)`
Displays a dialog box containing the specified message. If the parent component is null, the box is centered on the screen.

EvenOdd Example

- Determines if an integer is even or odd
- Instead of a single frame, it uses three dialog boxes



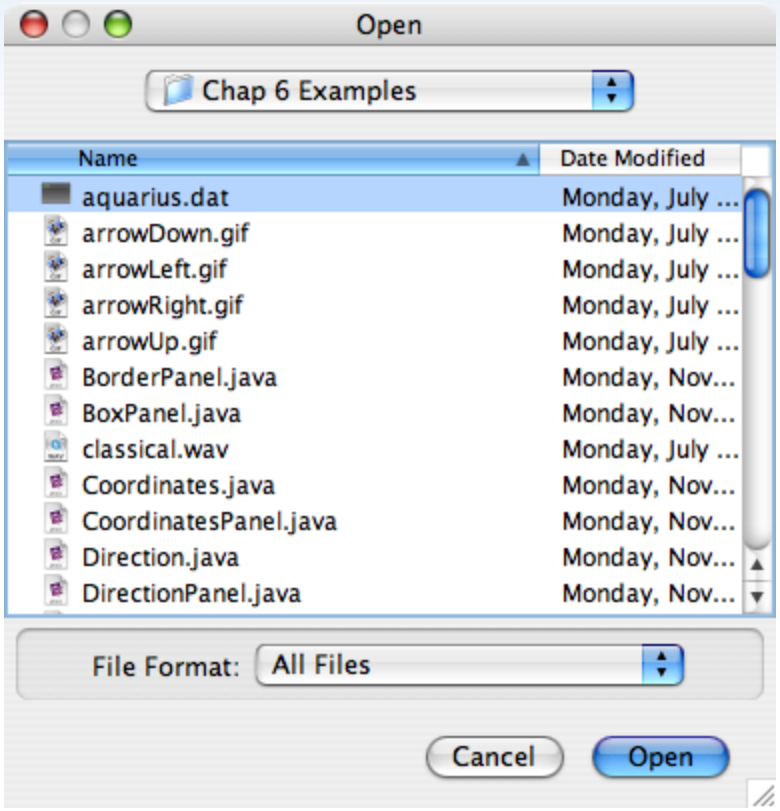


```
//********************************************************************
//  EvenOdd.java       Java Foundations
//
//  Demonstrates the use of the JOptionPane class.
//********************************************************************

import javax.swing.JOptionPane;

public class EvenOdd
{
   //-----------------------------------------------------------------
   //  Determines if the value input by the user is even or odd.
   //  Uses multiple dialog boxes for user interaction.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      String numStr, result;
      int num, again;

      do
      {
         numStr = JOptionPane.showInputDialog("Enter an integer: ");

         num = Integer.parseInt(numStr);

         result = "That number is " + ((num%2 == 0) ? "even" : "odd");
```

```
         JOptionPane.showMessageDialog(null, result);

         again = JOptionPane.showConfirmDialog(null, "Do Another?");
      }
      while (again == JOptionPane.YES_OPTION);
   }
}
```

File Choosers

- A *file chooser* is a specialized dialog box used to select a file from a disk or other storage medium
- The dialog automatically presents a standardized file selection window
- Filters can be applied to the file chooser programmatically
- The `JFileChooser` class creates this type of dialog box

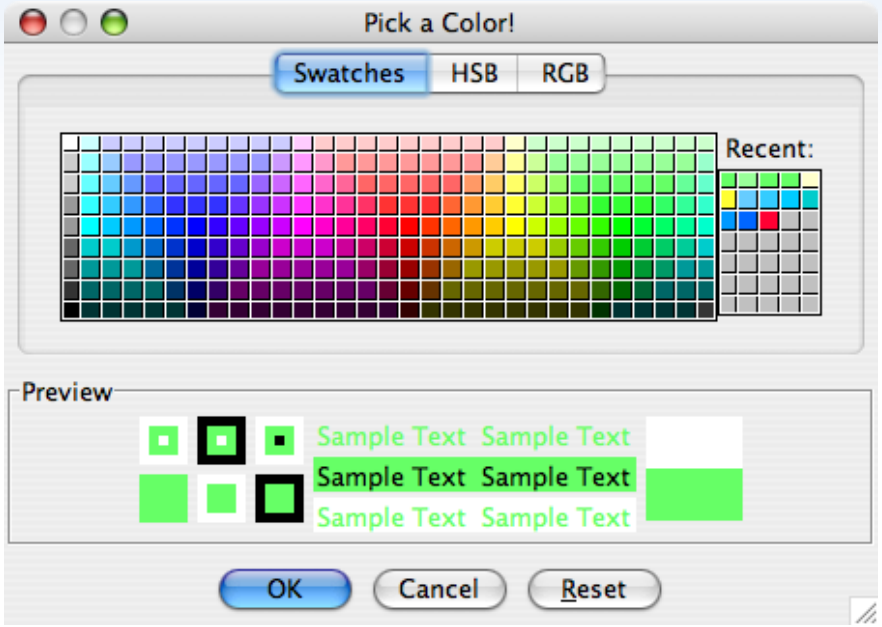


```
//********************************************************************
//  DisplayFile.java       Java Foundations
//
//  Demonstrates the use of a file chooser and a text area.
//********************************************************************

import java.util.Scanner;
import java.io.*;
import javax.swing.*;

public class DisplayFile
{
   //-----------------------------------------------------------------
   //  Opens a file chooser dialog, reads the selected file and
   //  loads it into a text area.
   //-----------------------------------------------------------------
   public static void main(String[] args) throws IOException
   {
      JFrame frame = new JFrame("Display File");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      JTextArea ta = new JTextArea(20, 30);
      JFileChooser chooser = new JFileChooser();

      int status = chooser.showOpenDialog(null);
```

```
      if (status != JFileChooser.APPROVE_OPTION)
         ta.setText("No File Chosen");
      else
      {
         File file = chooser.getSelectedFile();
         Scanner scan = new Scanner(file);

         String info = "";
         while (scan.hasNext())
            info += scan.nextLine() + "\n";

         ta.setText(info);
      }

      frame.getContentPane().add(ta);
      frame.pack();
      frame.setVisible(true);
   }
}
```

Color Choosers

- A *color chooser* dialog box can be displayed, permitting the user to select color from a list
- The `JColorChooser` represents a color chooser dialog box
- The user can also specify a color using RGB values



Borders

- Java provides the ability to put a *border* around any Swing component
- A border is not a component but defines how the edge of a component should be drawn
- Border provide visual cues as to how GUI components are organized
- The `BorderFactory` class is useful for creating borders for components

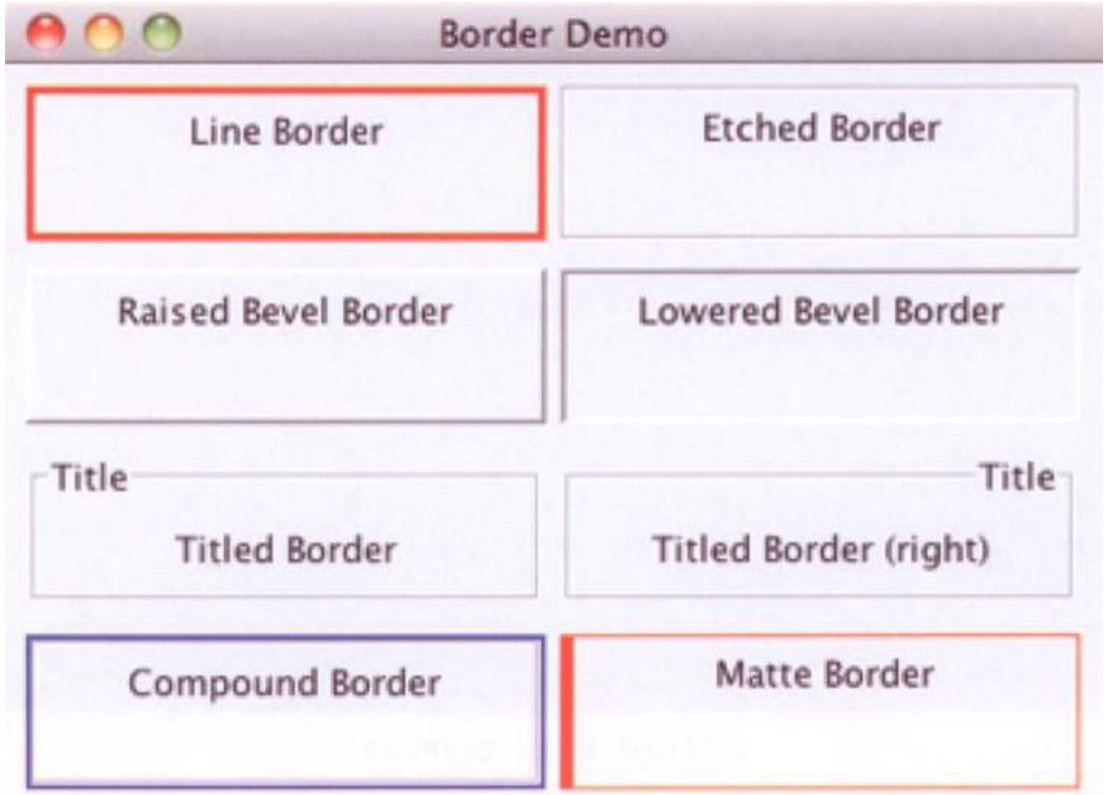
Borders

- Some borders that can be defined using the `BorderFactory` class:

Border	Description
Empty Border	Puts buffering space around the edge of a component, but otherwise has no visual effect.
Line Border	A simple line surrounding the component.
Etched Border	Creates the effect of an etched groove around a component.
Bevel Border	Creates the effect of a component raised above the surface or sunken below it.
Titled Border	Includes a text title on or around the border.
Matte Border	Allows the size of each edge to be specified. Uses either a soild color or an image.
Compound Border	A combination of two borders.

BorderDemo Example

- Displays several small panels with various borders



```
//********************************************************************
//  BorderDemo.java       Java Foundations
//
//  Demonstrates the use of various types of borders.
//********************************************************************

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

public class BorderDemo
{
   //-----------------------------------------------------------------
   //  Creates several bordered panels and displays them.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      JFrame frame = new JFrame("Border Demo");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      JPanel panel = new JPanel();
      panel.setLayout(new GridLayout(0, 2, 5, 10));
      panel.setBorder(BorderFactory.createEmptyBorder(8, 8, 8, 8));

      JPanel p1 = new JPanel();
      p1.setBorder(BorderFactory.createLineBorder(Color.red, 3));
      p1.add(new JLabel("Line Border"));
      panel.add(p1);
```

```
JPanel p2 = new JPanel();
p2.setBorder(BorderFactory.createEtchedBorder());
p2.add(new JLabel("Etched Border"));
panel.add(p2);

JPanel p3 = new JPanel();
p3.setBorder(BorderFactory.createRaisedBevelBorder());
p3.add(new JLabel("Raised Bevel Border"));
panel.add(p3);

JPanel p4 = new JPanel();
p4.setBorder(BorderFactory.createLoweredBevelBorder());
p4.add(new JLabel("Lowered Bevel Border"));
panel.add(p4);

JPanel p5 = new JPanel();
p5.setBorder(BorderFactory.createTitledBorder("Title"));
p5.add(new JLabel("Titled Border"));
panel.add(p5);

JPanel p6 = new JPanel();
TitledBorder tb = BorderFactory.createTitledBorder("Title");
tb.setTitleJustification(TitledBorder.RIGHT);
p6.setBorder(tb);
p6.add(new JLabel("Titled Border (right)"));
panel.add (p6);
```

```
      JPanel p7 = new JPanel();
      Border b1 = BorderFactory.createLineBorder(Color.blue, 2);
      Border b2 = BorderFactory.createEtchedBorder();
      p7.setBorder (BorderFactory.createCompoundBorder(b1, b2));
      p7.add (new JLabel("Compound Border"));
      panel.add(p7);

      JPanel p8 = new JPanel();
      Border mb = BorderFactory.createMatteBorder(1, 5, 1, 1,
                                                  Color.red);
      p8.setBorder(mb);
      p8.add(new JLabel("Matte Border"));
      panel.add(p8);

      frame.getContentPane().add (panel);
      frame.pack();
      frame.setVisible(true);
   }
}
```

Tool Tips

- A *tool tip* is a short line of text that appears over a component when the mouse cursor is rested momentarily on top of the component
- Tool tips usually inform the user about the component
- Tool tips can be assigned by using the `setToolTipText` method of a component

```
JButton button = new Button("Compute");
button.setToolTipText("Calculates the area under the curve");
```

Mnemonics

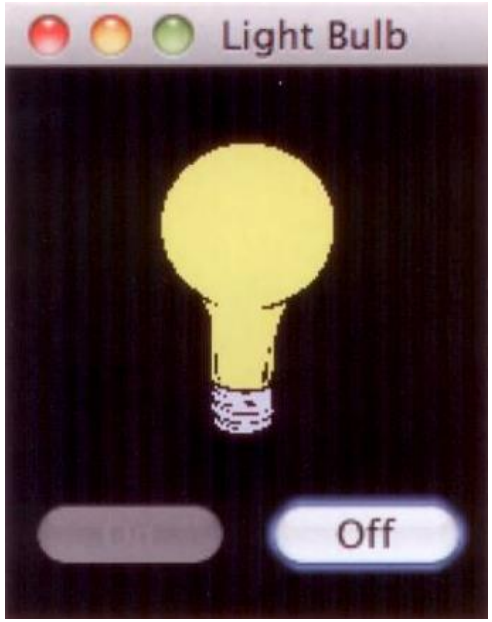
- A *mnemonic* is a character that allows the user to push a button or make a menu choice using the keyboard in addition to the mouse
- For example, when a mnemonic has been defined for a button, the user can hold down the Alt key and press the mnemonic character to activate (depress) the button
- We set the mnemonic for a component using the `setMnemonic` method of the component
- A character in the label may be underlined to indicate that it can be used as a shortcut

Disabling Components

- A component can be disabled to indicate it should not (cannot) be used
- A disabled component is usually "greyed out"
- This helps guide the user

LightBulb Example

- Tool tips, mnemonics, and disabled components are used in this example
- It displays an "on" or "off" bulb depending on which button is pressed



```
//********************************************************************
//  LightBulb.java       Java Foundations
//
//  Demonstrates mnemonics and tool tips.
//********************************************************************

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

public class LightBulb
{
   //-----------------------------------------------------------------
   //  Sets up a frame that displays a light bulb image that can be
   //  turned on and off.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("Light Bulb");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      LightBulbPanel bulb = new LightBulbPanel();
      LightBulbControls controls = new LightBulbControls (bulb);

      JPanel panel = new JPanel();
      panel.setBackground(Color.black);
      panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
```

```
      panel.add(Box.createRigidArea (new Dimension (0, 20)));
      panel.add(bulb);
      panel.add(Box.createRigidArea (new Dimension (0, 10)));
      panel.add(controls);
      panel.add(Box.createRigidArea (new Dimension (0, 10)));

      frame.getContentPane().add(panel);

      frame.pack();
      frame.setVisible(true);
   }
}
```

```
//********************************************************************
//  LightBulbPanel.java       Java Foundations
//
//  Represents the image for the LightBulb program.
//********************************************************************

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

public class LightBulbPanel extends JPanel
{
   private boolean on;
   private ImageIcon lightOn, lightOff;
   private JLabel imageLabel;

   //-----------------------------------------------------------------
   //  Constructor: Sets up the images and the initial state.
   //-----------------------------------------------------------------
   public LightBulbPanel()
   {
      lightOn = new ImageIcon("lightBulbOn.gif");
      lightOff = new ImageIcon("lightBulbOff.gif");

      setBackground(Color.black);

      on = true;
      imageLabel = new JLabel(lightOff);
      add(imageLabel);
   }
```

```
   //-----------------------------------------------------------------
   //  Paints the panel using the appropriate image.
   //-----------------------------------------------------------------
   public void paintComponent(Graphics page)
   {
      super.paintComponent(page);

      if (on)
         imageLabel.setIcon(lightOn);
      else
         imageLabel.setIcon(lightOff);
   }

   //-----------------------------------------------------------------
   //  Sets the status of the light bulb.
   //-----------------------------------------------------------------
   public void setOn(boolean lightBulbOn)
   {
      on = lightBulbOn;
   }
}
```

```
//********************************************************************
//  LightBulbControls.java       Java Foundations
//
//  Represents the control panel for the LightBulb program.
//********************************************************************

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

public class LightBulbControls extends JPanel
{
   private LightBulbPanel bulb;
   private JButton onButton, offButton;

   //-----------------------------------------------------------------
   //  Sets up the lightbulb control panel.
   //-----------------------------------------------------------------
   public LightBulbControls(LightBulbPanel bulbPanel)
   {
      bulb = bulbPanel;

      onButton = new JButton("On");
      onButton.setEnabled(false);
      onButton.setMnemonic('n');
      onButton.setToolTipText("Turn it on!");
      onButton.addActionListener(new OnListener());
```

```
      offButton = new JButton("Off");
      offButton.setEnabled(true);
      offButton.setMnemonic('f');
      offButton.setToolTipText("Turn it off!");
      offButton.addActionListener(new OffListener());

      setBackground(Color.black);
      add(onButton);
      add(offButton);
   }

   //*****************************************************************
   //  Represents the listener for the On button.
   //*****************************************************************
   private class OnListener implements ActionListener
   {
      //--------------------------------------------------------------
      //  Turns the bulb on and repaints the bulb panel.
      //--------------------------------------------------------------
      public void actionPerformed(ActionEvent event)
      {
         bulb.setOn(true);
         onButton.setEnabled(false);
         offButton.setEnabled(true);
         bulb.repaint();
      }
   }
```

```
   //*****************************************************************
   //  Represents the listener for the Off button.
   //*****************************************************************
   private class OffListener implements ActionListener
   {
      //--------------------------------------------------------------
      //  Turns the bulb off and repaints the bulb panel.
      //--------------------------------------------------------------
      public void actionPerformed(ActionEvent event)
      {
         bulb.setOn(false);
         onButton.setEnabled(true);
         offButton.setEnabled(false);
         bulb.repaint();
      }
   }
}
```

GUI Design

- Keep in mind our goal is to solve a problem
- Fundamental ideas of good GUI design include
  - knowing the user
  - preventing user errors
  - optimizing user abilities
  - being consistent
- We should design interfaces so that the user can make as few mistakes as possible