Java – GUI programming

Graphical Applications

- The example programs we've explored thus far have been text-based
- They are called *command-line applications*, which interact with the user using simple text prompts
- Let's examine some Java applications that have graphical components
- These components will serve as a foundation to programs that have true graphical user interfaces (GUIs)

GUI Components

- A *GUI component* is an object that represents a screen element such as a button or a text field
- GUI-related classes are defined primarily in the `java.awt` and the `javax.swing` packages
- The *Abstract Windowing Toolkit* (AWT) was the original Java GUI package
- The *Swing* package provides additional and more versatile components
- Both packages are needed to create a Java GUI-based program

GUI Containers

- A *GUI container* is a component that is used to hold and organize other components
- A *frame* is a container that is used to display a GUI-based Java application
- A frame is displayed as a separate window with a title bar – it can be repositioned and resized on the screen as needed
- A *panel* is a container that cannot be displayed on its own but is used to organize other components
- A panel must be added to another container to be displayed

GUI Containers

- A GUI container can be classified as either heavyweight or lightweight
- A *heavyweight container* is one that is managed by the underlying operating system
- A *lightweight container* is managed by the Java program itself
- Occasionally this distinction is important
- A frame is a heavyweight container and a panel is a lightweight container

Labels

- A *label* is a GUI component that displays a line of text
- Labels are usually used to display information or identify other components in the interface
- Let's look at a program that organizes two labels in a panel and displays that panel in a frame
- See Authority.java
- This program is not interactive, but the frame can be repositioned and resized

```
//********************************************************************
//  Authority.java
//
//  Demonstrates the use of frames, panels, and labels.
//********************************************************************
import java.awt.*;
import javax.swing.*;
public class Authority  {
  //-----------------------------------------------------------------
  //  Displays some words of wisdom.
  //-----------------------------------------------------------------
  public static void main (String[] args)  {
    JFrame frame = new JFrame ("Authority");
    frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);

    JPanel primary = new JPanel();
    primary.setBackground (Color.yellow);
    primary.setPreferredSize (new Dimension(250, 75));
    JLabel label1 = new JLabel ("Question authority,");
    JLabel label2 = new JLabel ("but raise your hand first.");

    primary.add (label1);
    primary.add (label2);
    frame.getContentPane().add(primary);
    frame.pack();
    frame.setVisible(true);
  }
}
```

Running Authority.class

Nested Panels

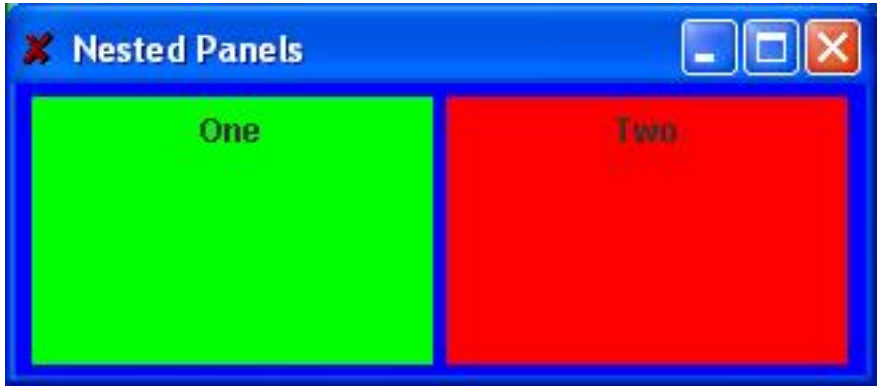
- Containers that contain other components make up the *containment hierarchy* of an interface
- This hierarchy can be as intricate as needed to create the visual effect desired
- The following example nests two panels inside a third panel – note the effect this has as the frame is resized
- See NestedPanels.java

```
//********************************************************************
//  NestedPanels.java
//
//  Demonstrates a basic componenet hierarchy.
//********************************************************************

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

public class NestedPanels
{
  //-----------------------------------------------------------------
  //  Presents two colored panels nested within a third.
  //-----------------------------------------------------------------
  public static void main (String[] args)
  {
    JFrame frame = new JFrame ("Nested Panels");
    frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);

    // Set up first subpanel
    JPanel subPanel1 = new JPanel();
    subPanel1.setPreferredSize (new Dimension(150, 100));
    subPanel1.setBackground (Color.green);
    JLabel label1 = new JLabel ("One");
    subPanel1.add (label1);
```

```
// Set up second subpanel
     JPanel subPanel2 = new JPanel();
     subPanel2.setPreferredSize (new Dimension(150, 100));
     subPanel2.setBackground (Color.red);
     JLabel label2 = new JLabel ("Two");
     subPanel2.add (label2);

     // Set up primary panel
     JPanel primary = new JPanel();
     primary.setBackground (Color.blue);
     primary.add (subPanel1);
     primary.add (subPanel2);

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

NestedPanels.java - Sample Execution

- The following is a sample execution of NestedPanels.class



Images

- Images are often used in a programs with a graphical interface
- Java can manage images in both JPEG and GIF formats
- As we've seen, a `JLabel` object can be used to display a line of text
- It can also be used to display an image
- That is, a label can be composed of text, and image, or both at the same time

Images

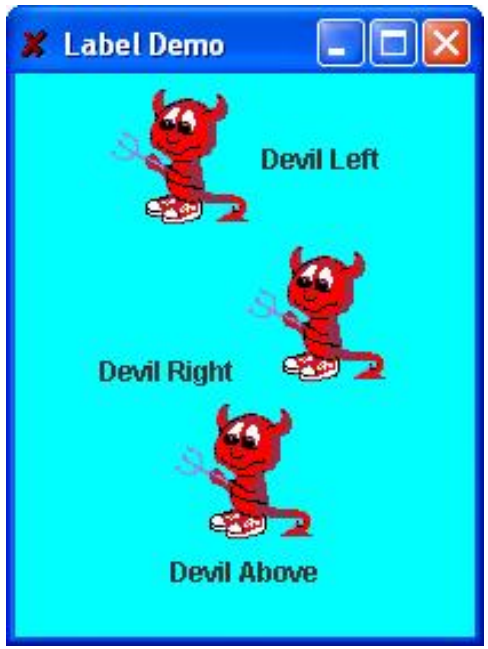
- The `ImageIcon` class is used to represent an image that is stored in a label
- The position of the text relative to the image can be set explicitly
- The alignment of the text and image within the label can be set as well
- See LabelDemo.java

```
//********************************************************************
//  LabelDemo.java
 //  Demonstrates the use of image icons in labels.
//********************************************************************

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

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

      ImageIcon icon = new ImageIcon ("devil.gif");

      JLabel label1, label2, label3;

      label1 = new JLabel ("Devil Left", icon, SwingConstants.CENTER);
```

```
        label2 = new JLabel ("Devil Right", icon, SwingConstants.CENTER);
        label2.setHorizontalTextPosition (SwingConstants.LEFT);
        label2.setVerticalTextPosition (SwingConstants.BOTTOM);

        label3 = new JLabel ("Devil Above", icon, SwingConstants.CENTER);
        label3.setHorizontalTextPosition (SwingConstants.CENTER);
        label3.setVerticalTextPosition (SwingConstants.BOTTOM);

        JPanel panel = new JPanel();
        panel.setBackground (Color.cyan);
        panel.setPreferredSize (new Dimension (200, 250));
        panel.add (label1);
        panel.add (label2);
        panel.add (label3);

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

LabelDemo.java - Sample Execution

- The following is a sample execution of LabelDemo.class



Graphical Objects

- Some objects contain information that determines how the object should be represented visually
- Most GUI components are graphical objects
- We can have some effect on how components get drawn

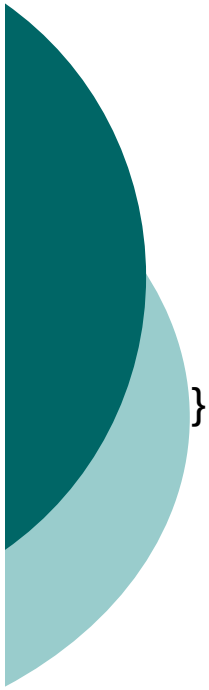
Smiling Face Example

- The `SmilingFace` program draws a face by defining the `paintComponent` method of a panel
- See SmilingFace.java
- See SmilingFacePanel.java
- The `main` method of the `SmilingFace` class instantiates a `SmilingFacePanel` and displays it
- The `SmilingFacePanel` class is derived from the `JPanel` class using inheritance

```
//********************************************************************
//  SmilingFace.java
//
//  Demonstrates the use of a separate panel class.
//********************************************************************

import javax.swing.JFrame;

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

      SmilingFacePanel panel = new SmilingFacePanel();

      frame.getContentPane().add(panel);

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

```
//********************************************************************
//  SmilingFacePanel.java
//
//  Demonstrates the use of a separate panel class.
//********************************************************************

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

public class SmilingFacePanel extends JPanel
{
   private final int BASEX = 120, BASEY = 60; // base point for head

   //-----------------------------------------------------------------
   //  Constructor: Sets up the main characteristics of this panel.
   //-----------------------------------------------------------------
   public SmilingFacePanel ()
   {
      setBackground (Color.blue);
      setPreferredSize (new Dimension(320, 200));
      setFont (new Font("Arial", Font.BOLD, 16));
   }
```

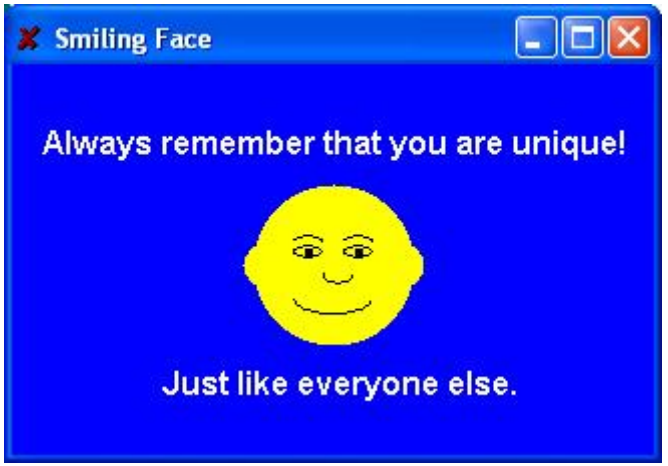
```
//-----------------------------------------------------------
//  Draws a face.
//-----------------------------------------------------------
public void paintComponent (Graphics page)
{
  super.paintComponent (page);

  page.setColor (Color.yellow);
  page.fillOval (BASEX, BASEY, 80, 80);  // head
  page.fillOval (BASEX-5, BASEY+20, 90, 40);  // ears

  page.setColor (Color.black);
  page.drawOval (BASEX+20, BASEY+30, 15, 7);  // eyes
  page.drawOval (BASEX+45, BASEY+30, 15, 7);

  page.fillOval (BASEX+25, BASEY+31, 5, 5);   // pupils
  page.fillOval (BASEX+50, BASEY+31, 5, 5);

  page.drawArc (BASEX+20, BASEY+25, 15, 7, 0, 180);  // eyebrows
  page.drawArc (BASEX+45, BASEY+25, 15, 7, 0, 180);

  page.drawArc (BASEX+35, BASEY+40, 15, 10, 180, 180);  // nose
  page.drawArc (BASEX+20, BASEY+50, 40, 15, 180, 180);  // mouth
```



```
        page.setColor (Color.white);
        page.drawString ("Always remember that you are unique!",
                        BASEX-105, BASEY-15);
        page.drawString ("Just like everyone else.", BASEX-45, BASEY+105);
    }
}
```

SmilingFace.java - Sample Execution

- The following is a sample execution of SmilingFace.class



Smiling Face Example

- Every Swing component has a `paintComponent` method
- The `paintComponent` method accepts a `Graphics` object that represents the graphics context for the panel
- We define the `paintComponent` method to draw the face with appropriate calls to the `Graphics` methods
- Note the difference between drawing on a panel and adding other GUI components to a panel

Splat Example

- The `Splat` example is structured a bit differently
- It draws a set of colored circles on a panel, but each circle is represented as a separate object that maintains its own graphical information
- The `paintComponent` method of the panel "asks" each circle to draw itself
- See Splat.java
- See SplatPanel.java
- See Circle.java

```
//********************************************************************
//  Splat.java
//
//  Demonstrates
//********************************************************************

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

public class Splat
{
   //-----------------------------------------------------------------
   //  Presents a collection of circles.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      JFrame frame = new JFrame ("Splat");
      frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);

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

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

```
//********************************************************************
//  SplatPanel.java
//
//  Demonstrates the use of graphical objects.
//********************************************************************

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

public class SplatPanel extends JPanel
{
   private Circle circle1, circle2, circle3, circle4, circle5;
   //-----------------------------------------------------------------
   //  Constructor: Creates five Circle objects.
   //-----------------------------------------------------------------
   public SplatPanel()
   {
      circle1 = new Circle (30, Color.red, 70, 35);
      circle2 = new Circle (50, Color.green, 30, 20);
      circle3 = new Circle (100, Color.cyan, 60, 85);
      circle4 = new Circle (45, Color.yellow, 170, 30);
      circle5 = new Circle (60, Color.blue, 200, 60);

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

```
  //-----------------------------------------------------------------
  //  Draws this panel by requesting that each circle draw itself.
  //-----------------------------------------------------------------
  public void paintComponent (Graphics page)
  {
    super.paintComponent(page);

    circle1.draw(page);
    circle2.draw(page);
    circle3.draw(page);
    circle4.draw(page);
    circle5.draw(page);
  }
}
```

```
//********************************************************************
//  Circle.java
//
//  Represents a circle with a particular position, size, and color.
//********************************************************************

import java.awt.*;

public class Circle
{
   private int diameter, x, y;
   private Color color;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this circle with the specified values.
   //-----------------------------------------------------------------
   public Circle (int size, Color shade, int upperX, int upperY)
   {
      diameter = size;
      color = shade;
      x = upperX;
      y = upperY;
   }
```

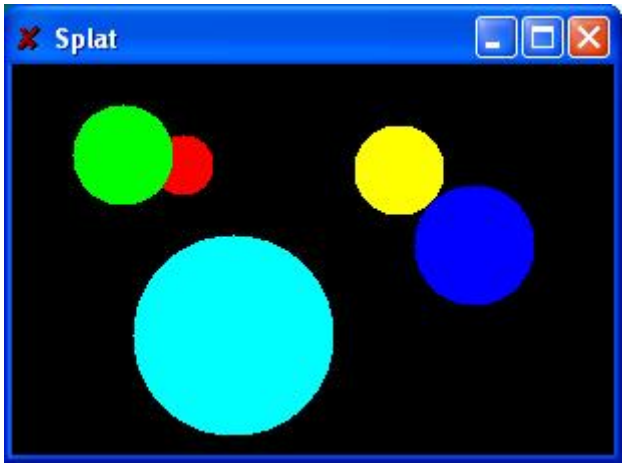
```
//-----------------------------------------------------------------
//  Draws this circle in the specified graphics context.
//-----------------------------------------------------------------
public void draw (Graphics page)
{
  page.setColor (color);
  page.fillOval (x, y, diameter, diameter);
}

//-----------------------------------------------------------------
//  Diameter mutator.
//-----------------------------------------------------------------
public void setDiameter (int size)
{
  diameter = size;
}

//-----------------------------------------------------------------
//  Color mutator.
//-----------------------------------------------------------------
public void setColor (Color shade)
{
  color = shade;
}
```

```
//-----------------------------------------------------------
//  X mutator.
//-----------------------------------------------------------
public void setX (int upperX)
{
  x = upperX;
}


//-----------------------------------------------------------
//  Y mutator.
//-----------------------------------------------------------
public void setY (int upperY)
{
  y = upperY;
}

//-----------------------------------------------------------
//  Diameter accessor.
//-----------------------------------------------------------
public int getDiameter ()
{
  return diameter;
```

```
  //---------------------------------------------------------------
  //  Color accessor.
  //---------------------------------------------------------------
  public Color getColor ()
  {
    return color;
  }

  //---------------------------------------------------------------
  //  X accessor.
  //---------------------------------------------------------------
  public int getX ()
  {
    return x;
  }

  //---------------------------------------------------------------
  //  Y accessor.
  //---------------------------------------------------------------
  public int getY ()
  {
    return y;
  }
}
```

Splat.java - Sample Execution

- The following is a sample execution of Splat.class



Graphical User Interfaces

- A Graphical User Interface (GUI) in Java is created with at least three kinds of objects:
  - components
  - events
  - listeners
- We've previously discussed *components*, which are objects that represent screen elements
  - labels, buttons, text fields, menus, etc.
- Some components are *containers* that hold and organize other components
  - frames, panels, applets, dialog boxes

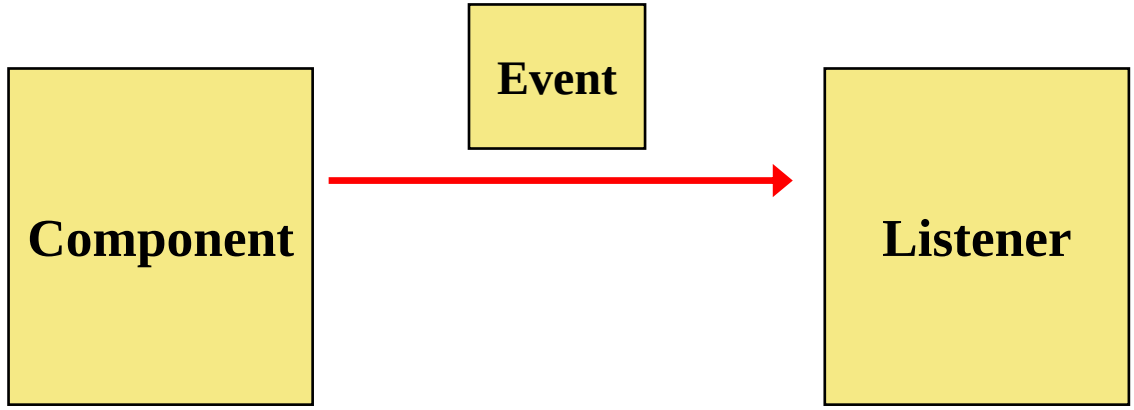
Events

- An *event* is an object that represents some activity to which we may want to respond
- For example, we may want our program to perform some action when the following occurs:
  - the mouse is moved
  - the mouse is dragged
  - a mouse button is clicked
  - a graphical button is clicked
  - a keyboard key is pressed
  - a timer expires
- Events often correspond to user actions, but not always

Events and Listeners

- The Java standard class library contains several classes that represent typical events
- Components, such as a graphical button, generate (or fire) an event when it occurs
- A *listener* object "waits" for an event to occur and responds accordingly
- We can design listener objects to take whatever actions are appropriate when an event occurs

Events and Listeners



A component object may generate an event

A corresponding listener object is designed to respond to the event

When the event occurs, the component calls the appropriate method of the listener, passing an object that describes the event

GUI Development

- Generally we use components and events that are predefined by classes in the Java class library
- Therefore, to create a Java program that uses a GUI we must:
  - instantiate and set up the necessary components
  - implement listener classes for any events we care about
  - establish the relationship between listeners and components that generate the corresponding events
- Let's now explore some new components and see how this all comes together

Buttons

- A *push button* is a component that allows the user to initiate an action by pressing a graphical button using the mouse
- A push button is defined by the `JButton` class
- It generates an *action event*
- The `PushCounter` example displays a push button that increments a counter each time it is pushed
- See PushCounter.java
- See PushCounterPanel.java

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

import javax.swing.JFrame;

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

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

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

```
//********************************************************************
//  PushCounterPanel.java
//
//  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());

    label = new JLabel ("Pushes: " + count);

    add (push);
    add (label);

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

//*****************************************************************
//  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);
    }
  }
}
```

PushCounter.java - Sample Execution

- The following is a sample execution of PushCounter.class

Push Counter Example

- The components of the GUI are the button, a label to display the counter, a panel to organize the components, and the main frame
- The `PushCounterPanel` class is represents the panel used to display the button and label
- The `PushCounterPanel` class is derived from `JPanel` using inheritance
- The constructor of `PushCounterPanel` sets up the elements of the GUI and initializes the counter to zero

Push Counter Example

- The `ButtonListener` class is the listener for the action event generated by the button
- It is implemented as an *inner class*, which means it is defined within the body of another class
- That facilitates the communication between the listener and the GUI components
- Inner classes should only be used in situations where there is an intimate relationship between the two classes and the inner class is not needed in any other context

Push Counter Example

- Listener classes are written by implementing a *listener interface*
- The `ButtonListener` class implements the `ActionListener` interface
- An interface is a list of methods that the implementing class must define
- The only method in the `ActionListener` interface is the `actionPerformed` method
- The Java class library contains interfaces for many types of events

Push Counter Example

- The `PushCounterPanel` constructor:
  - instantiates the `ButtonListener` object
  - establishes the relationship between the button and the listener by the call to `addActionListener`
- When the user presses the button, the button component creates an `ActionEvent` object and calls the `actionPerformed` method of the listener
- The `actionPerformed` method increments the counter and resets the text of the label

Text Fields

- Let's look at another GUI example that uses another type of component
- A *text field* allows the user to enter one line of input
- If the cursor is in the text field, the text field component generates an action event when the enter key is pressed
- See Fahrenheit.java
- See FahrenheitPanel.java

```
//********************************************************************
//  Fahrenheit.java
//
//  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
//
//  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));
    }
  }
}
```

Fahrenheit.java - Sample Execution

- The following is a sample execution of Fahrenheit.class

Fahrenheit Example

- Like the `PushCounter` example, the GUI is set up in a separate panel class
- The `TempListener` inner class defines the listener for the action event generated by the text field
- The `FahrenheitPanel` constructor instantiates the listener and adds it to the text field
- When the user types a temperature and presses enter, the text field generates the action event and calls the `actionPerformed` method of the listener
- The `actionPerformed` method computes the conversion and updates the result label

Determining Event Sources

- Recall that interactive GUIs require establishing a relationship between components and the listeners that respond to component events
- One listener object can be used to listen to two different components
- The source of the event can be determined by using the `getSource` method of the event passed to the listener
- See LeftRight.java
- See LeftRightPanel.java

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

import javax.swing.JFrame;

public class LeftRight
{
  //-----------------------------------------------------------------
  //  Creates 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
//
//  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");
      }
   }
}
```

LeftRight.java - Sample Execution

- The following is a sample execution of LeftRight.class

Dialog Boxes

- A *dialog box* is a window that appears on top of any currently active window
- It may be used to:
  - convey information
  - confirm an action
  - allow the user to enter data
  - pick a color
  - choose a file
- A dialog box usually has a specific, solitary purpose, and the user interaction with it is brief

Dialog Boxes

- The `JOptionPane` class provides methods that simplify the creation of some types of dialog boxes
- See EvenOdd.java

```
//********************************************************************
//  EvenOdd.java
//
//  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");
```

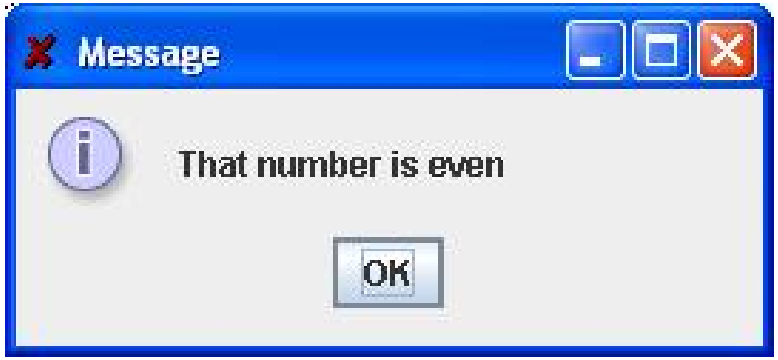
```
            JOptionPane.showMessageDialog (null, result);

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

EvenOdd.java - Sample Execution

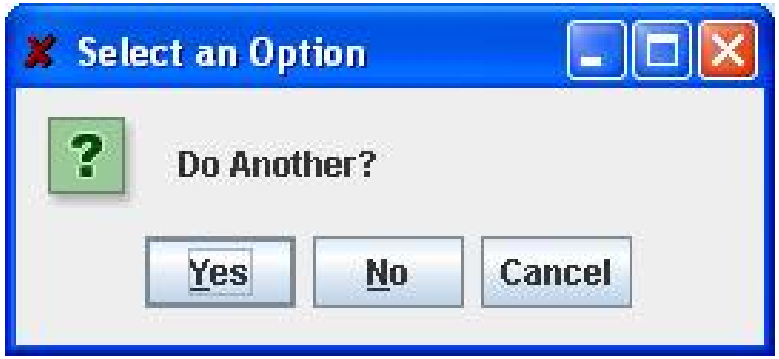
- The following is a sample execution of EvenOdd.class

1.

2.



3.



Check Boxes

- A *check box* is a button that can be toggled on or off
- It is represented by the `JCheckBox` class
- Unlike a push button, which generates an action event, a check box generates an *item event* whenever it changes state (is checked on or off)
- The `ItemListener` interface is used to define item event listeners
- The check box calls the `itemStateChanged` method of the listener when it is toggled

Check Boxes

- Let's examine a program that uses check boxes to determine the style of a label's text string
- It uses the Font class, which represents a character font's:
  - family name (such as Times or Courier)
  - style (bold, italic, or both)
  - font size
- See StyleOptions.java
- See StyleOptionsPanel.java

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

import javax.swing.JFrame;

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

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

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

```
//********************************************************************
//  StyleOptionsPanel.java
//
//  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));
    }
  }
}
```

StyleOptions.java - Sample Execution

- The following is a sample execution of StyleOptions.class

Radio Buttons

- A group of *radio buttons* represents a set of mutually exclusive options – only one can be selected at any given time
- When a radio button from a group is selected, the button that is currently "on" in the group is automatically toggled off
- To define the group of radio buttons that will work together, each radio button is added to a `ButtonGroup` object
- A radio button generates an action event

Radio Buttons

- Let's look at a program that uses radio buttons to determine which line of text to display
- See `QuoteOptions.java`
- See `QuoteOptionsPanel.java`
- Compare and contrast check boxes and radio buttons
  - Check boxes work independently to provide a boolean option
  - Radio buttons work as a group to provide a set of mutually exclusive options

```
//********************************************************************
//  QuoteOptions.java
//
//  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);

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

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

```
//********************************************************************
//  QuoteOptionsPanel.java
//
//  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);
```

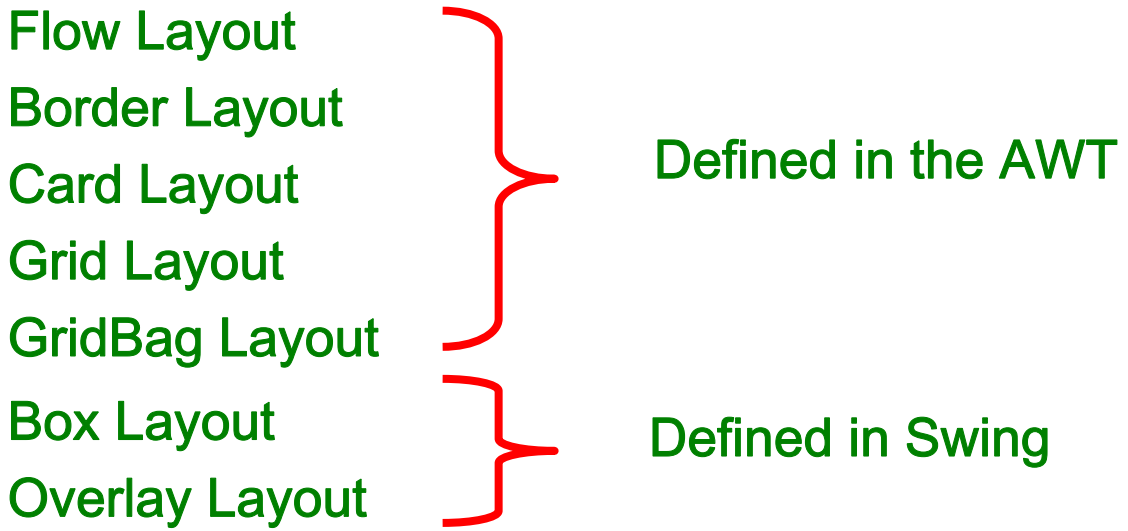
```
    setBackground (Color.green);
    setPreferredSize (new Dimension(300, 100));
  }

//*****************************************************************
  //  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);
    }
  }
}
```

QuoteOptions.java - Sample Execution

- The following is a sample execution of QuoteOptions.class

Layout Managers

- A *layout manager* is an object that determines the way that components are arranged in a container
- There are several predefined layout managers defined in the Java standard class library:



Layout Managers

- Every container has a default layout manager, but we can explicitly set the layout manager as well
- Each layout manager has its own particular rules governing how the components will be arranged
- Some layout managers pay attention to a component's preferred size or alignment, while others do not
- A layout manager attempts to adjust the layout as components are added and as containers are resized

Layout Managers

- We can use the `setLayout` method of a container to change its layout manager

```
JPanel panel = new JPanel();
panel.setLayout(new BorderLayout());
```

- The following example uses a *tabbed pane*, a container which permits one of several panes to be selected
- See `LayoutDemo.java`
- See `IntroPanel.java`

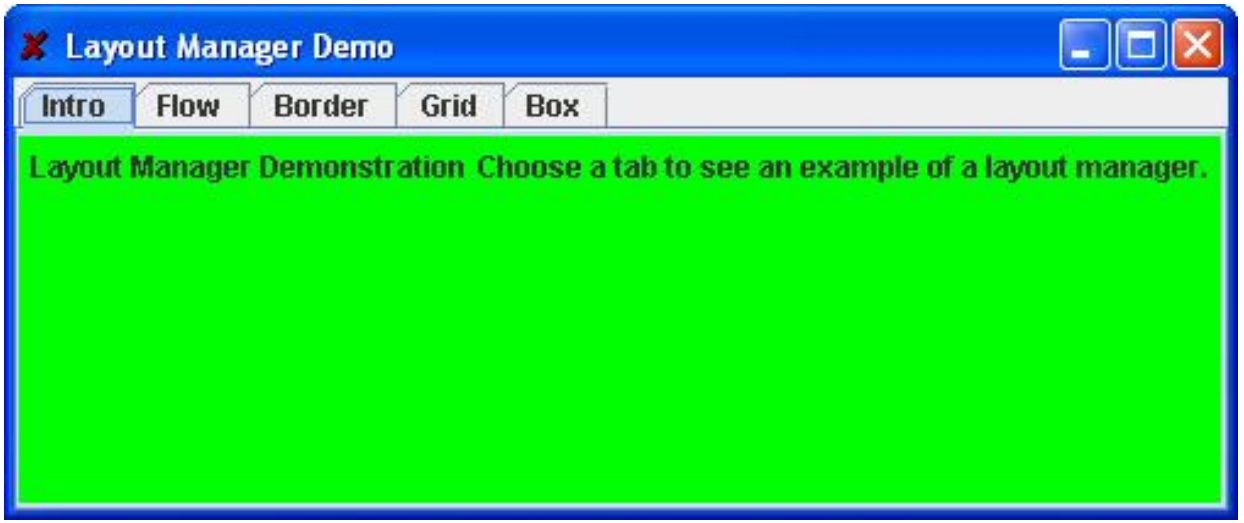
```
//********************************************************************
//  LayoutDemo.java
//
//  Demonstrates the use of flow, border, grid, and box layouts.
//********************************************************************

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
//
//  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);
  }
}
```

LayoutDemo.java - Sample Execution

- The following is a sample execution of LayoutDemo.class




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


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


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

Flow Layout

- *Flow layout* puts as many components as possible on a row, then moves to the next row
- Rows are created as needed to accommodate all of the components
- Components are displayed in the order they are added to the container
- Each row of components is centered horizontally in the window by default, but could also be aligned left or right
- Also, the horizontal and vertical gaps between the components can be explicitly set
- See `FlowPanel.java`

```
//********************************************************************
//  FlowPanel.java
//
//  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");
```

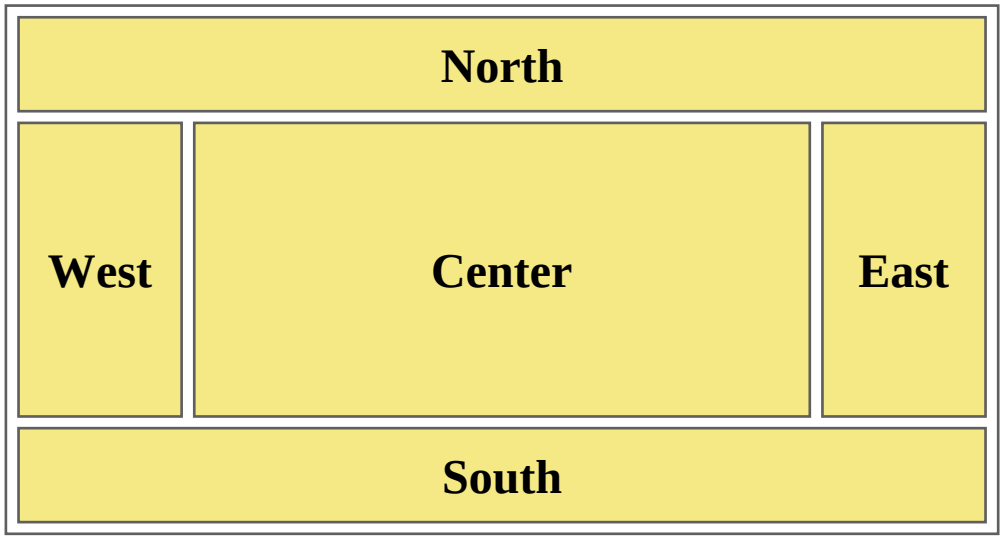
```
        JButton b4 = new JButton ("BUTTON 4");
        JButton b5 = new JButton ("BUTTON 5");

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

FlowPanel.java - Sample Execution

- The following is a sample execution of FlowPanel.class

Border Layout

- A *border layout* defines five areas into which components can be added



Border Layout

- Each area displays one component (which could be a container such as a `JPanel`)
- Each of the four outer areas enlarges as needed to accommodate the component added to it
- If nothing is added to the outer areas, they take up no space and other areas expand to fill the void
- The center area expands to fill space as needed
- See `BorderPanel.java`

```
//********************************************************************
//  BorderPanel.java
//
//  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);
    }
}
```

BorderPanel.java - Sample Execution

- The following is a sample execution of BorderPanel.class

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 of the grid, and all cells have the same size
- As components are added to the container, they fill the grid from left-to-right and top-to-bottom (by default)
- The size of each cell is determined by the overall size of the container
- See `GridPanel.java`

```
//********************************************************************
//  GridPanel.java
//
//  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);
    }
}
```

GridPanel.java - Sample Execution

- The following is a sample execution of GridPanel.class

Box Layout

- A *box layout* organizes components horizontally (in one row) or vertically (in one column)
- Components are placed top-to-bottom or left-to-right in the order in which they are added to the container
- By combining multiple containers using box layout, many different configurations can be created
- Multiple containers with box layouts are often preferred to one container that uses the more complicated gridbag layout manager

Box Layout

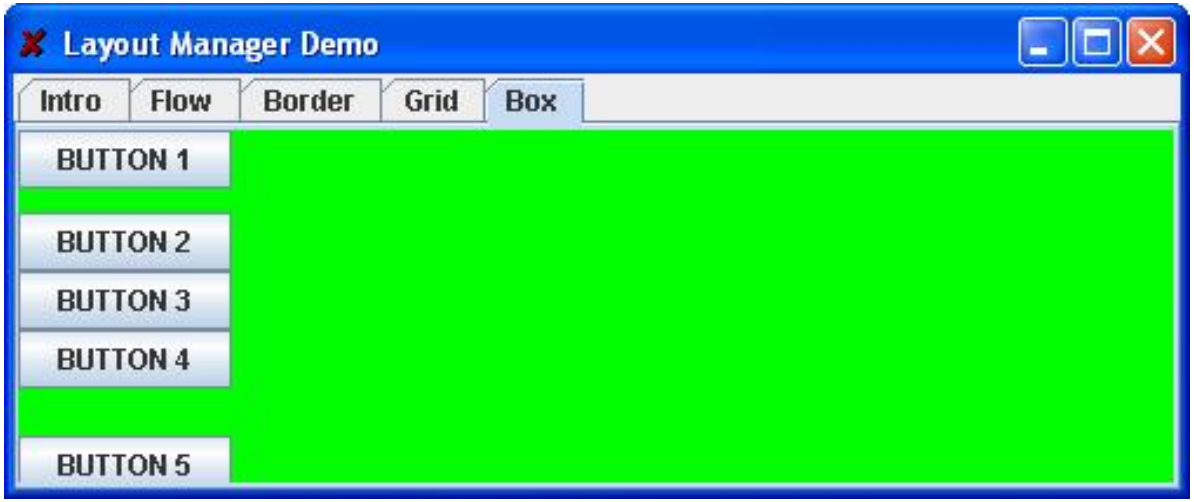
- *Invisible components* can be added to a box layout container to take up space between components
  - *Rigid areas* have a fixed size
  - *Glue* specifies where excess space should go
- A rigid area is created using the `createRigidArea` method of the `Box` class
- Glue is created using the `createHorizontalGlue` or `createVerticalGlue` methods
- See `BoxPanel.java`

```
//********************************************************************
//  BoxPanel.java
//
//  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);
    }
}
```

BoxPanel.java - Sample Execution

The following is a sample execution of BoxPanel.class



Mouse Events

Events related to the mouse are separated into *mouse events* and *mouse motion events*

- Mouse Events:

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 Events

- Mouse Motion Events:

mouse moved	the mouse is moved
mouse dragged	the mouse is moved while the mouse button is pressed down

Listeners for mouse events are created using the `MouseListener` and `MouseMotionListener` interfaces

A `MouseEvent` object is passed to the appropriate method when a mouse event occurs

Mouse Events

- For a given program, we may only care about one or two mouse events
- To satisfy the implementation of a listener interface, empty methods must be provided for unused events
- See `Dots.java`
- See `DotsPanel.java`

```
//********************************************************************
//  Dots.java
//
//  Demonstrates mouse events.
//********************************************************************

import javax.swing.JFrame;

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

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

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

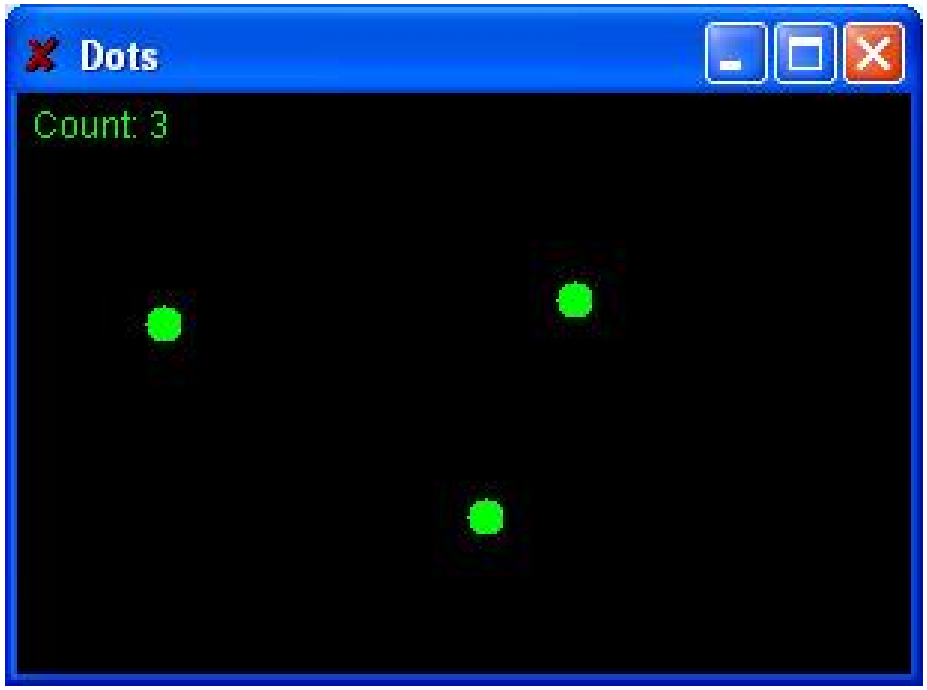
```
//********************************************************************
//  DotsPanel.java
//
//  Represents the primary panel for the Dots program.
//********************************************************************

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

public class DotsPanel extends JPanel
{
  private final int SIZE = 6;  // radius of each dot

  private ArrayList<Point> pointList;

  //-----------------------------------------------------------------
  //  Constructor: Sets up this panel to listen for mouse events.
  //-----------------------------------------------------------------
  public DotsPanel()
  {
    pointList = new ArrayList<Point>();

    addMouseListener (new DotsListener());
```

```
  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);

   for (Point spot : pointList)
      page.fillOval (spot.x-SIZE, spot.y-SIZE, SIZE*2, SIZE*2);

   page.drawString ("Count: " + pointList.size(), 5, 15);
}
```

```
//*******************************************************************
  //  Represents the listener for mouse events.

//*******************************************************************
  private class DotsListener 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)
    {
      pointList.add(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) {}
  }
}
```

Dots.java - Sample Execution

- The following is a sample execution of Dots.class



Mouse Events

- *Rubberbanding* is the visual effect in which a shape is "stretched" as it is drawn using the mouse
- The following example continually redraws a line as the mouse is dragged
- See `RubberLines.java`
- See `RubberLinesPanel.java`

```
//********************************************************************
//  RubberLines.java
//
//  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);
  }
}
```

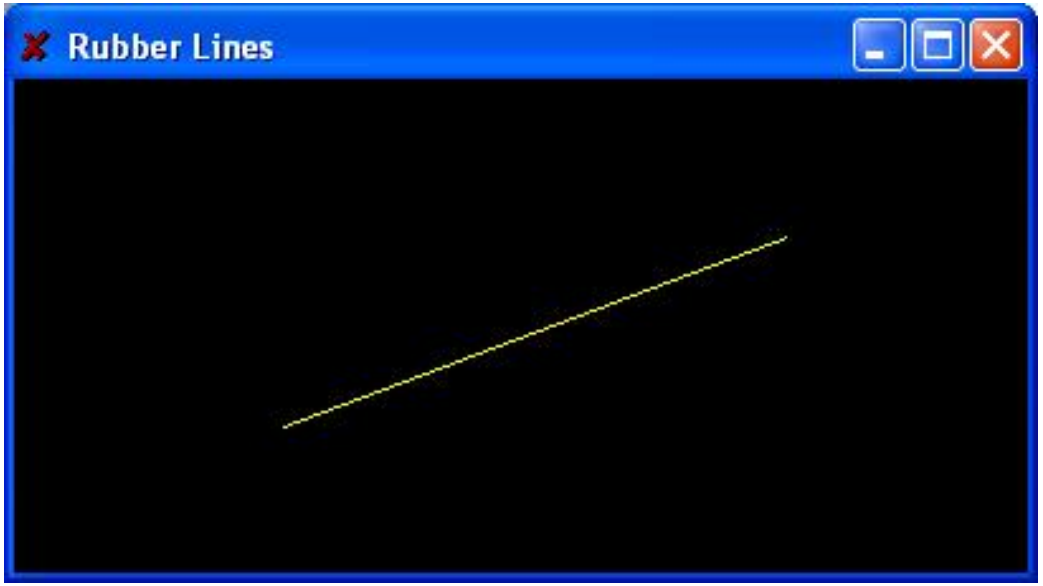
```
//********************************************************************
//  RubberLinesPanel.java
//
//  Represents the primary drawing panel for the RubberLines program.
//********************************************************************

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) {}
  }
}
```

RubberLines.java - Sample Execution

- The following is a sample execution of RubberLines.class



The Component Class Hierarchy

- The Java classes that define GUI components are part of a class hierarchy
- Swing GUI components typically are derived from the `JComponent` class which is derived from the `Container` class which is derived from the `Component` class
- Many Swing components can serve as (limited) containers, because they are derived from the `Container` class
- For example, a `JLabel` object can contain an `ImageIcon`

The Component Class Hierarchy

- An applet is a good example of inheritance
- Recall that when we define an applet, we extend the `Applet` class or the `JApplet` class
- The `Applet` and `JApplet` classes already handle all the details about applet creation and execution, including:
  - interaction with a Web browser
  - accepting applet parameters through HTML
  - enforcing security restrictions

The Component Class Hierarchy

- Our applet classes only have to deal with issues that specifically relate to what our particular applet will do
- When we define `paintComponent` method of an applet, we are actually overriding a method defined originally in the `JComponent` class and inherited by the `JApplet` class

Event Adapter Classes

- Inheritance also gives us a alternate technique for creating listener classes
- We've seen that listener classes can be created by implementing a particular interface, such as `MouseListener`
- We can also create a listener class by extending an *event adapter class*
- Each listener interface that has more than one method has a corresponding adapter class, such as the `MouseAdapter` class

Event Adapter Classes

- Each adapter class implements the corresponding listener and provides empty method definitions
- When you derive a listener class from an adapter class, you only need to override the event methods that pertain to the program
- Empty definitions for unused event methods do not need to be defined because they are provided via inheritance
- See `OffCenter.java`
- See `OffCenterPanel.java`

```
//********************************************************************
//  OffCenter.java
//
//  Demonstrates the use of an event adapter class.
//********************************************************************

import javax.swing.*;

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

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

```
//********************************************************************
//  OffCenterPanel.java
//
//  Represents the primary drawing panel for the OffCenter program.
//********************************************************************

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

public class OffCenterPanel extends JPanel
{
   private final int WIDTH=300, HEIGHT=300;

   private DecimalFormat fmt;
   private Point current;
   private int centerX, centerY;
   private double length;

   //-----------------------------------------------------------------
   //  Constructor: Sets up the panel and necessary data.
   //-----------------------------------------------------------------
```

```
  public OffCenterPanel()
{
  addMouseListener (new OffCenterListener());

  centerX = WIDTH / 2;
  centerY = HEIGHT / 2;

  fmt = new DecimalFormat ("0.##");

  setPreferredSize (new Dimension(WIDTH, HEIGHT));
  setBackground (Color.yellow);
}
//-----------------------------------------------------------------
//  Draws a line from the mouse pointer to the center point of
//  the applet and displays the distance.
//-----------------------------------------------------------------
public void paintComponent (Graphics page)
{
  super.paintComponent (page);

  page.setColor (Color.black);
  page.drawOval (centerX-3, centerY-3, 6, 6);
  if (current != null)
  {
    page.drawLine (current.x, current.y, centerX, centerY);
    page.drawString ("Distance: " + fmt.format(length), 10, 15);
  }
}
```

```
//*********************************************************************
  //  Represents the listener for mouse events. Demonstrates the
  //  ability to extend an adaptor class.

//*********************************************************************
  private class OffCenterListener extends MouseAdapter
  {
    //-----------------------------------------------------------
    //  Computes the distance from the mouse pointer to the center
    //  point of the applet.
    //-----------------------------------------------------------
    public void mouseClicked (MouseEvent event)
    {
      current = event.getPoint();
      length = Math.sqrt(Math.pow((current.x-centerX), 2) +
                         Math.pow((current.y-centerY), 2));
      repaint();
    }
  }
}
```

OffCenter.java - Sample Execution

- The following is a sample execution of OffCenter.class