Lecture 10b

GUI Components: Part 2

OBJECTIVES

In this lecture you will learn:

- To create and manipulate sliders, menus, pop-up menus and windows.
- To change the look-and-feel of a GUI, using Swing's pluggable look-and-feel.
- To create a multiple-document interface with `JDesktopPane` and `JInternalFrame`.
- To use additional layout managers.

Outline

22.1 Introduction

- **Pluggable look-and-feel (PLAF)**
  - **Swing can customize the look-and-feel of the GUI**
  - **Motif**
    - **A popular UNIX look-and-feel**
- **Multiple-document interface (MDI)**
  - **A main window (the parent window) containing other windows (child windows)**
  - **Manages several open documents parallel**

22.2 `JSlider`

- `JSlider`
  - Enables the user to select from a range of integer values
  - Inherits from `JComponent`
  - Contains:
    - Tick marks
      - Can display major tick marks, minor tick marks and labels for tick marks
      - Are not displayed by default
    - Thumb
      - Allows the user to select a value
  - Snap-to ticks
    - Cause the thumb to snap to the closest tick mark

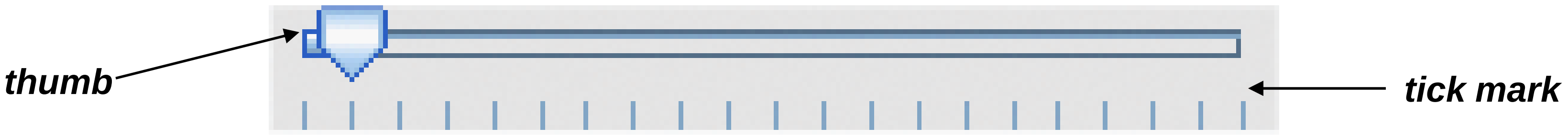


Fig. 22.1 | **`JSlider` component with horizontal orientation.**

22.2 `JSlider` (Cont.)

- If a `JSlider` has the focus (is the currently selected GUI component in the user interface)
  - Left/right arrow keys cause the thumb of the `JSlider` to decrease/increase by 1
  - Down/up arrow keys cause the thumb of the `JSlider` to decrease/increase by 1
  - *PgDn* (page down)/*PgUp* (page up) *keys* cause the thumb of the `JSlider` to decrease/increase by block increments of one-tenth of the range of values
  - *Home*/*End keys* move the thumb of the `JSlider` to the minimum/maximum value of the `JSlider`

22.2 `JSlider` (Cont.)

- Can have either horizontal or vertical orientation
  - Minimum value is at the left or bottom end of the `JSlider`
  - Maximum value is at the right or top end of the `JSlider`
  - `JSlider` method `setInverted` reverses the minimum and maximum value positions
- Generate `ChangeEvent`s in response to user interactions
  - An object of a class that implements interface `ChangeListener` and declares method `stateChanged` can respond to `ChangeEvent`s

Look-and-Feel Observation 22.1

If a new GUI component has a minimum width and height (i.e., smaller dimensions would render the component ineffective on the display), override method `getMinimumSize` to return the minimum width and height as an instance of class `Dimension`.

Software Engineering Observation 22.1

For many GUI components, method `getMinimumSize` is implemented to return the result of a call to the component's `getPreferredSize` method.

```
// Fig. 22.2: OvalPanel.java
// A customized JPanel class.
import java.awt.Graphics;
import java.awt.Dimension;
import javax.swing.JPanel;

public class OvalPanel extends JPanel
{
   private int diameter = 10; // default diameter of 10

   // draw an oval of the specified diameter
   public void paintComponent( Graphics g )
   {
      super.paintComponent( g );

      g.fillOval( 10, 10, diameter, diameter ); // draw circle
   } // end method paintComponent

   // validate and set diameter, then repaint
   public void setDiameter( int newDiameter )
   {
      // if diameter invalid, default to 10
      diameter = ( newDiameter >= 0 ? newDiameter : 10 );
      repaint(); // repaint panel
   } // end method setDiameter

```

Used as the width and height of the bounding box in which the circle is displayed

Draws a filled circle

Change the circle's **`diameter`** and **`repaint`**

```
   // used by layout manager to determine preferred size
   public Dimension getPreferredSize()
   {
      return new Dimension( 200, 200 );
   } // end method getPreferredSize

   // used by layout manager to determine minimum size
   public Dimension getMinimumSize()
   {
      return getPreferredSize();
   } // end method getMinimumSize
} // end class OvalPanel
```

Return the preferred width and height of an **`OvalPanel`**

Return an **`OvalPanel`**’s minimum width and height

```java
// Fig. 22.3: SliderFrame.java
// Using JSliders to size an oval.
import java.awt.BorderLayout;
import java.awt.Color;
import javax.swing.JFrame;
import javax.swing.JSlider;
import javax.swing.SwingConstants;
import javax.swing.event.ChangeListener;
import javax.swing.event.ChangeEvent;

public class SliderFrame extends JFrame
{
   private JSlider diameterJSlider; // slider to select diameter
   private OvalPanel myPanel; // panel to draw circle

   // no-argument constructor
   public SliderFrame()
   {
      super( "Slider Demo" );

      myPanel = new OvalPanel(); // create panel to draw circle
      myPanel.setBackground( Color.YELLOW ); // set background to yellow

      // set up JSlider to control diameter value
      diameterJSlider =
         new JSlider( SwingConstants.HORIZONTAL, 0, 200, 10 );
      diameterJSlider.setMajorTickSpacing( 10 ); // create tick
      diameterJSlider.setPaintTicks( true ); // paint ticks on slider

```

Create **`OvalPanel`** object **`myPanel`**

Create **`JSlider`** object **`diameterSlider`** as a horizontal JSlider with a range of **`0-200`** and a initial value of **`10`**

Indicate that each major-tick mark represents **`10`** values and that the tick marks should be displayed

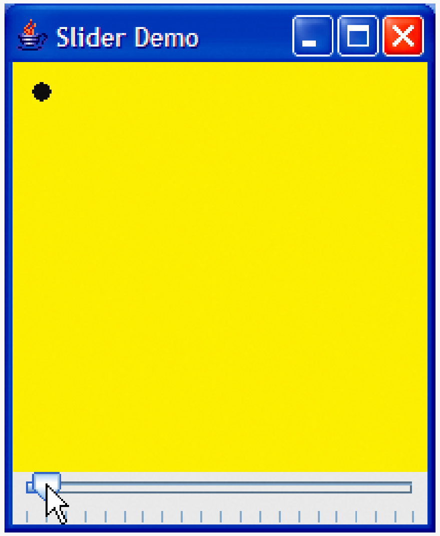
```
      // register JSlider event listener
      diameterJSlider.addChangeListener(

         new ChangeListener() // anonymous inner class
         {
            // handle change in slider value
            public void stateChanged( ChangeEvent e )
            {
               myPanel.setDiameter( diameterJSlider.getValue() );
            } // end method stateChanged
         } // end anonymous inner class
      ); // end call to addChangeListener

      add( diameterJSlider, BorderLayout.SOUTH ); //
      add( myPanel, BorderLayout.CENTER ); // add par
   } // end SliderFrame constructor
} // end class SliderFrame
```

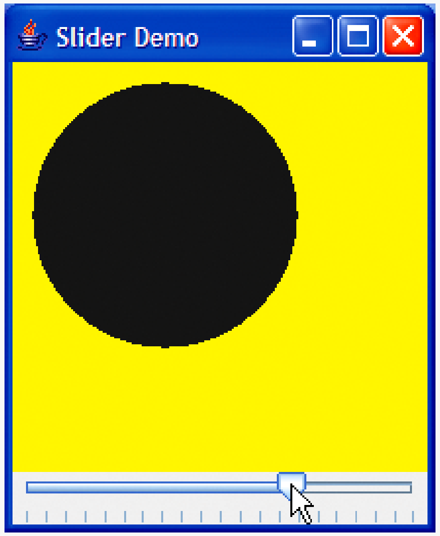
Register a **ChangeListener** to handle **diameterSlider**’s events

Method **stateChanged** is called in response to a user interaction

Call **myPanel**’s **setDiameter** method and pass the current thumb position value returned by **JSlider** method **getValue**

```java
// Fig. 22.4: SliderDemo.java
// Testing SliderFrame.
import javax.swing.JFrame;

public class SliderDemo
{
   public static void main( String args[] )
   {
      SliderFrame sliderFrame = new SliderFrame();
      sliderFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      sliderFrame.setSize( 220, 270 ); // set frame size
      sliderFrame.setVisible( true ); // display frame
   } // end main
} // end class SliderDemo
```





22.3 Windows: Additional Notes

- **`JFrame`**
  - **Is a window with a title bar and a border**
  - **A subclass of `java.awt.Frame`**
    - **Which is a subclass of `java.awt.Window`**
  - **One of the few Swing GUI components that is not a lightweight GUI component**
  - **Java application windows look like every other window displayed on that platform**

Good Programming Practice 22.1

A windows is an expensive system resource. Return it to the system when it is no longer needed.

22.3 Windows: Additional Notes (Cont.)

- `JFrame` method `setDefaultCloseOperation` determines what happens when the user closes the window
  - `DISPOSE_ON_CLOSE`
    - Dispose of the `Window` to return resources to the system
  - `DO_NOTHING_ON_CLOSE`
    - Indicates that the program will determine what to do when the user indicates that the window should close
  - `HIDE_ON_CLOSE`
    - The default
- `JFrame` method `setVisible`
  - Display the window on the screen
- `JFrame` method `setLocation`
  - Specify the window’s position when it appears on the screen

Common Programming Error 22.1

Forgetting to call method `setVisible` on a window is a runtime logic error—the window is not displayed.

Common Programming Error 22.2

Forgetting to call the `setSize` method on a window is a runtime logic error—only the title bar appears.

22.3 Windows: Additional Notes (Cont.)

- **User manipulation of the window generates window events**
  - **Method `addWindowListener` registers event listeners for window events**
  - **Interface `WindowListener` provides seven window-event-handling methods**
    - **`windowActivated` – called when the user makes a window the main window**
    - **`windowClosed` – called after the window is closed**
    - **`windowClosing` – called when the user initiates closing of the window**

22.3 Windows: Additional Notes (Cont.)

- `windowDeactivated` – called when the user makes another window the main window
- `windowDeiconified` – called when the user restores a window from being minimized
- `windowIconified` – called when the user minimizes a window
- `windowOpened` – called when a program first displays a window on the screen

22.4 Using Menus with Frames

- Menus
  - Allow the user to perform actions without unnecessarily cluttering a GUI with extra components
  - Can be attached only to objects of the classes that provide member `setMenuBar`, such as `JFrame` and `JApplet`
  - Class `MenuBar`
    - Contains the methods necessary to manage a menu bar
  - Class `JMenu`
    - Contains the methods necessary for managing menus
  - Class `JMenuItem`
    - Contains the methods necessary to manage menu items
      - Can be used to initiate an action or can be a submenu

Look-and-Feel Observation 22.2

Menus simplify GUIs because components can be hidden within them. These components will only be visible when the user looks for them by selecting the menu.

22.4 Using Menus with Frames (Cont.)

- Class `JCheckBoxMenuItem`
  - Contains the methods necessary to manage menu items that can be toggled on or off
- Class `JRadioButtonMenuItem`
  - Contains the methods necessary to manage menu items that can be toggled on or off like `JCheckBoxMenuItem`s
  - When multiple `JRadioButtonMenuItem`s are maintained as part of a `ButtonGroup`, only one item in the group can be selected at a given time
- Mnemonics
  - Special characters that can provide quick access to a menu or menu item from the keyboard

```java
// Fig. 22.5: MenuFrame.java
// Demonstrating menus.
import java.awt.Color;
import java.awt.Font;
import java.awt.BorderLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;
import javax.swing.JFrame;
import javax.swing.JRadioButtonMenuItem;
import javax.swing.JCheckBoxMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JLabel;
import javax.swing.SwingConstants;
import javax.swing.ButtonGroup;
import javax.swing.JMenu;
import javax.swing.JMenuItem;
import javax.swing.JMenuBar;

```

```
public class MenuFrame extends JFrame
{
   private final Color colorValues[] =
      { Color.BLACK, Color.BLUE, Color.RED, Color.GREEN };
   private JRadioButtonMenuItem colorItems[]; // color menu items
   private JRadioButtonMenuItem fonts[]; // font menu items
   private JCheckBoxMenuItem styleItems[]; // font style menu items
   private JLabel displayJLabel; // displays sample text
   private ButtonGroup fontButtonGroup; // manages font menu items
   private ButtonGroup colorButtonGroup; // manages color menu items
   private int style; // used to create style for font

   // no-argument constructor set up GUI
   public MenuFrame()
   {
      super( "Using JMenus" );

      JMenu fileMenu = new JMenu( "File" ); // create file menu
      fileMenu.setMnemonic( 'F' ); // set mnemonic to F

      // create About... menu item
      JMenuItem aboutItem = new JMenuItem( "About..." );
      aboutItem.setMnemonic( 'A' ); // set mnemonic to A
      fileMenu.add( aboutItem ); // add about item to file menu
      aboutItem.addActionListener(

```

Create a **JMenu**

Call **JMenu** method **setMnemonic**

Add the "**About…**" **JMenuItem** to **fileMenu**

```
            new ActionListener() // anonymous inner class
            {
               // display message dialog when user selects About...
               public void actionPerformed( ActionEvent event )
               {
                  JOptionPane.showMessageDialog( MenuFrame.this,
                     "This is an example\nof using menus",
                     "About", JOptionPane.PLAIN_MESSAGE );
               } // end method actionPerformed
            } // end anonymous inner class
         ); // end call to addActionListener

         JMenuItem exitItem = new JMenuItem( "Exit" ); // create exit item
         exitItem.setMnemonic( 'x' ); // set mnemonic to x
         fileMenu.add( exitItem ); // add exit item to file menu
         exitItem.addActionListener(

            new ActionListener() // anonymous inner class
            {
               // terminate application when user clicks exitItem
               public void actionPerformed( ActionEvent event )
               {
                  System.exit( 0 ); // exit application
               } // end method actionPerformed
            } // end anonymous inner class
         ); // end call to addActionListener

```

Create an **ActionListener** to process **aboutItem**'s action event

Display a message dialog box

Create and add menu item **exitItem**

Register an **ActionListener** that terminates the application

```
      JMenuBar bar = new JMenuBar(); // create menu bar
      setJMenuBar( bar ); // add menu bar to application
      bar.add( fileMenu ); // add file menu to menu bar

      JMenu formatMenu = new JMenu( "Format" ); // create format menu
      formatMenu.setMnemonic( 'r' ); // set mnemonic to r

      // array listing string colors
      String colors[] = { "Black", "Blue", "Red", "Green" };

      JMenu colorMenu = new JMenu( "Color" ); // create color menu
      colorMenu.setMnemonic( 'C' ); // set mnemonic to C

      // create radiobutton menu items for colors
      colorItems = new JRadioButtonMenuItem[ colors.length ];
      colorButtonGroup = new ButtonGroup(); // manages colors
      ItemHandler itemHandler = new ItemHandler(); // handler for

      // create color radio button menu items
      for ( int count = 0; count < colors.length; count++ )
      {
         colorItems[ count ] =
            new JRadioButtonMenuItem( colors[ count ] ); // create item
         colorMenu.add( colorItems[ count ] ); // add item to color menu
         colorButtonGroup.add( colorItems[ count ] ); // add to group
         colorItems[ count ].addActionListener( itemHandler );
      } // end for

```

Add **fileMenu** to a **JMenuBar** and attach the **JMenuBar** to the application window

Create menu **formatMenu**

Create submenu **colorMenu**

Create **JRadioButtonMenuItem** array **colorItems**

Create a **ButtonGroup** to ensure that only one of the menu items is selected at a time

Add **JRadioButtonMenuItem**s to **colorMenu** and register **ActionListener**s

```
      colorItems[ 0 ].setSelected( true ); // select first Color item

      formatMenu.add( colorMenu ); // add color menu to format menu
      formatMenu.addSeparator(); // add separator in menu

      // array listing font names
      String fontNames[] = { "Serif", "Monospaced", "SansSerif" };
      JMenu fontMenu = new JMenu( "Font" ); // create font menu
      fontMenu.setMnemonic( 'n' ); // set mnemonic to n

      // create radiobutton menu items for font names
      fonts = new JRadioButtonMenuItem[ fontNames.length ];
      fontButtonGroup = new ButtonGroup(); // manages font names

      // create Font radio button menu items
      for ( int count = 0; count < fonts.length; count++ )
      {
         fonts[ count ] = new JRadioButtonMenuItem( fontNames[ count
         fontMenu.add( fonts[ count ] ); // add font to font menu
         fontButtonGroup.add( fonts[ count ] ); // add to button group
         fonts[ count ].addActionListener( itemHandler ); // add handler
      } // end for

      fonts[ 0 ].setSelected( true ); // select first Font menu item
      fontMenu.addSeparator(); // add separator bar to font menu

```

Invoke **AbstractButton** method **setSelected**

Add **colorMenu** to **formatMenu** and add a horizontal separator line

Create **JRadioButtonMenuItem** array **fonts**

Create a **ButtonGroup** to ensure that only one of the menu items is selected at a time

Add **JRadioButtonMenuItem**s to **colorMenu** and register **ActionListener**s

Set default selection and add horizontal separator

```
      String styleNames[] = { "Bold", "Italic" }; // names of styles
      styleItems = new JCheckBoxMenuItem[ styleNames.length ];
      StyleHandler styleHandler = new StyleHandler(); // style handler

      // create style checkbox menu items
      for ( int count = 0; count < styleNames.length; count++ )
      {
         styleItems[ count ] =
            new JCheckBoxMenuItem( styleNames[ count ] ); // for style
         fontMenu.add( styleItems[ count ] ); // add to font menu
         styleItems[ count ].addItemListener( styleHandler ); // handler
      } // end for

      formatMenu.add( fontMenu ); // add Font menu to Format menu
      bar.add( formatMenu ); // add Format menu to menu bar

      // set up label to display text
      displayJLabel = new JLabel( "Sample Text", SwingConstants.CENTER );
      displayJLabel.setForeground( colorValues[ 0 ] );
      displayJLabel.setFont( new Font( "Serif", Font.PLAIN, 72 ) );

      getContentPane().setBackground( Color.CYAN ); // set background
      add( displayJLabel, BorderLayout.CENTER ); // add displayJLabel
   } // end MenuFrame constructor

```

Create **JCheckBoxMenuItems**

Add **fontMenu** to **formatMenu** and **formatMenu** to the **JMenuBar**

```
   // inner class to handle action events from menu items
   private class ItemHandler implements ActionListener
   {
      // process color and font selections
      public void actionPerformed( ActionEvent event )
      {
         // process color selection
         for ( int count = 0; count < colorItems.length; count++ )
         {
            if ( colorItems[ count ].isSelected() )
            {
               displayJLabel.setForeground( colorValues[ count ] );
               break;
            } // end if
         } // end for

         // process font selection
         for ( int count = 0; count < fonts.length; count++ )
         {
            if ( event.getSource() == fonts[ count ] )
            {
               displayJLabel.setFont(
                  new Font( fonts[ count ].getText(), style, 72 ) );
            } // end if
         } // end for

```

Determine the selected **`JRadioButtonMenuItem`**

`getSource` method returns a reference to the **`JRadioButtonMenuItem`** that generated the event

```
         repaint(); // redraw application
      } // end method actionPerformed
   } // end class ItemHandler

   // inner class to handle item events from check box menu items
   private class StyleHandler implements ItemListener
   {
      // process font style selections
      public void itemStateChanged( ItemEvent e )
      {
         style = 0; // initialize style

         // check for bold selection
         if ( styleItems[ 0 ].isSelected() )
            style += Font.BOLD; // add bold to style

         // check for italic selection
         if ( styleItems[ 1 ].isSelected() )
            style += Font.ITALIC; // add italic to style

         displayJLabel.setFont(
            new Font( displayJLabel.getFont().getName(), style, 72 ) );
         repaint(); // redraw application
      } // end method itemStateChanged
   } // end class StyleHandler
} // end class MenuFrame
```

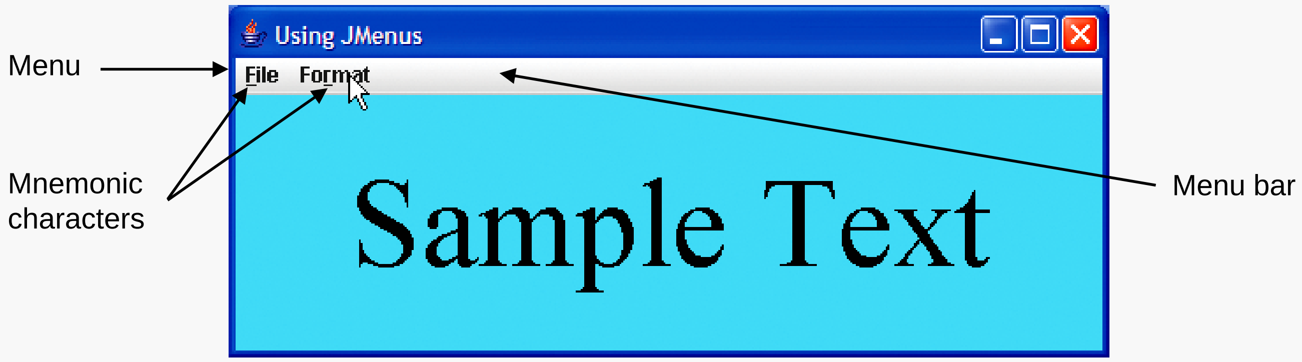
Called if the user selects a **`JCheckBoxMenuItem`** in the **`fontMenu`**

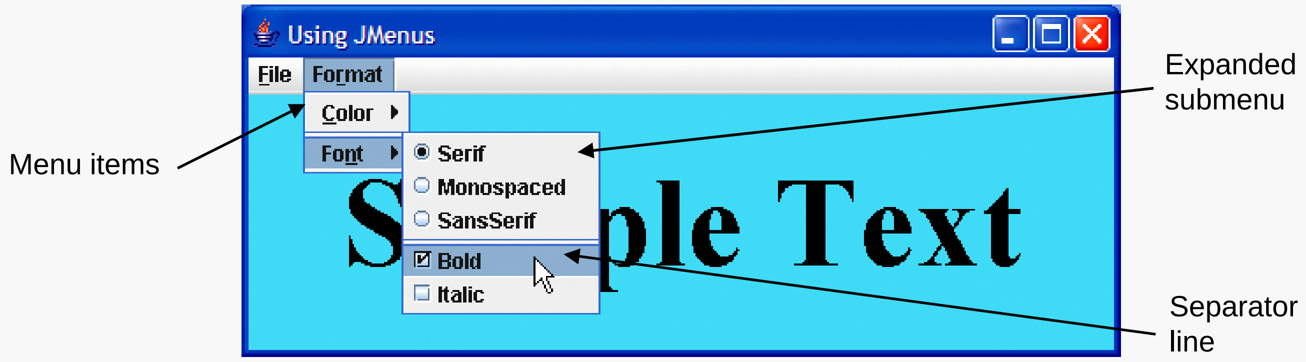
Determine whether either or both of the **`JCheckBoxMenuItem`**s are selected

Look-and-Feel Observation 22.3

Mnemonics provide quick access to menu commands and button commands through the keyboard.

```
// Fig. 22.6: MenuTest.java
// Testing MenuFrame.
import javax.swing.JFrame;

public class MenuTest
{
   public static void main( String args[] )
   {
      MenuFrame menuFrame = new MenuFrame(); // create MenuFrame
      menuFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      menuFrame.setSize( 500, 200 ); // set frame size
      menuFrame.setVisible( true ); // display frame
   } // end main
} // end class MenuTest
```




Using JMenus
File
Format
Color
Font
Serif
Monospaced
SansSerif
Bold
Italic
Sample Text
Menu items
Expanded submenu
Separator line

Look-and-Feel Observation 22.4

Different mnemonics should be used for each button or menu item. Normally, the first letter in the label on the menu item or button is used as the mnemonic. If several buttons or menu items start with the same letter, choose the next most prominent letter in the name (e.g., `x` is commonly chosen for a button or menu item called `Exit`).

22.4 Using Menus with Frames (Cont.)

- **`showMessageDialog` method**
  - **Specifying the parent window helps determine where the dialog box will be displayed**
    - **If specified as `null`, the dialog box appears in the center of the screen**
    - **Otherwise, it appears centered over the specified parent window**
  - **Modal dialog box**
    - **Does not allow any other window in the application to be accessed until the dialog box is dismissed**
    - **Dialog boxes are typically modal**

Common Programming Error 22.3

Forgetting to set the menu bar with `JFrame` method `setJMenuBar` results in the menu bar not being displayed on the `JFrame`.

Look-and-Feel Observation 22.5

Menus appear left to right in the order that they are added to a `JMenuBar`.

Look-and-Feel Observation 22.6

A submenu is created by adding a menu as a menu item in another menu. When the mouse is positioned over a submenu (or the submenu's mnemonic is pressed), the submenu expands to show its menu items.

Look-and-Feel Observation 22.7

Separators can be added to a menu to group menu items logically.

Look-and-Feel Observation 22.8

Any lightweight GUI component (i.e., a component that is a subclass of `JComponent`) can be added to a `JMenu` or to a `JMenuBar`.

22.5 JPopupMenu

- Context-sensitive pop-up menus
  - Provide options that are specific to the component for which the pop-up trigger event was generated
    - On most systems, the pop-up trigger event occurs when the user presses and releases the right mouse button
  - Created with class `JPopupMenu`

Look-and-Feel Observation 22.9

The pop-up trigger event is platform specific. On most platforms that use a mouse with multiple buttons, the pop-up trigger event occurs when the user clicks the right mouse button on a component that supports a pop-up menu.

```java
// Fig. 22.7: PopupFrame.java
// Demonstrating JPopupMenus.
import java.awt.Color;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JRadioButtonMenuItem;
import javax.swing.JPopupMenu;
import javax.swing.ButtonGroup;

public class PopupFrame extends JFrame
{
   private JRadioButtonMenuItem items[]; // holds items for colors
   private final Color colorValues[] =
      { Color.BLUE, Color.YELLOW, Color.RED }; // colors to be used
   private JPopupMenu popupMenu; // allows user to select color

   // no-argument constructor sets up GUI
   public PopupFrame()
   {
      super( "Using JPopupMenus" );

      ItemHandler handler = new ItemHandler(); // handler for menu items
      String colors[] = { "Blue", "Yellow", "Red" }; // array of colors

```

An instance of class **`ItemHandler`** will process the item events from the menu items

```java
      ButtonGroup colorGroup = new ButtonGroup(); // manages color items
      popupMenu = new JPopupMenu(); // create pop-up menu
      items = new JRadioButtonMenuItem[ 3 ]; // items for selecting color

      // construct menu item, add to popup menu, enable event handling
      for ( int count = 0; count < items.length; count++ )
      {
         items[ count ] = new JRadioButtonMenuItem( colors[ count ] );
         popupMenu.add( items[ count ] ); // add item to pop-up menu
         colorGroup.add( items[ count ] ); // add item to button group
         items[ count ].addActionListener( handler ); // add handler
      } // end for

      setBackground( Color.WHITE ); // set background to

      // declare a MouseListener for the window to display pop-up menu
      addMouseListener(

         new MouseAdapter() // anonymous inner class
         {
            // handle mouse press event
            public void mousePressed( MouseEvent event )
            {
               checkForTriggerEvent( event ); // check for trigger
            } // end method mousePressed

```

Create a **`JPopupMenu`** object

Create and add **`JRadioButtonMenuItem`** and register **`ActionListener`**s

Register a **`MouseListener`** to handle the mouse events of the application window

```
            // handle mouse release event
            public void mouseReleased( MouseEvent event )
            {
               checkForTriggerEvent( event ); // check for trigger
            } // end method mouseReleased

            // determine whether event should trigger popup menu
            private void checkForTriggerEvent( MouseEvent event )
            {
               if ( event.isPopupTrigger() )
                  popupMenu.show(
                     event.getComponent(), event.getX(), event.getY() );
            } // end method checkForTriggerEvent
         } // end anonymous inner class
      ); // end call to addMouseListener
   } // end PopupFrame constructor

```

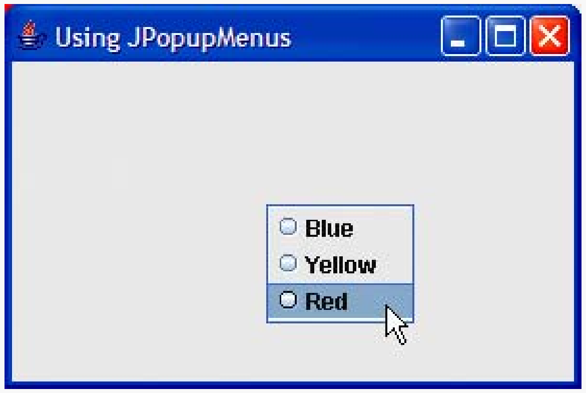
If the pop-up trigger event occurred, **`JPopupMenu`** method **`show`** displays the **`JPopupMenu`**

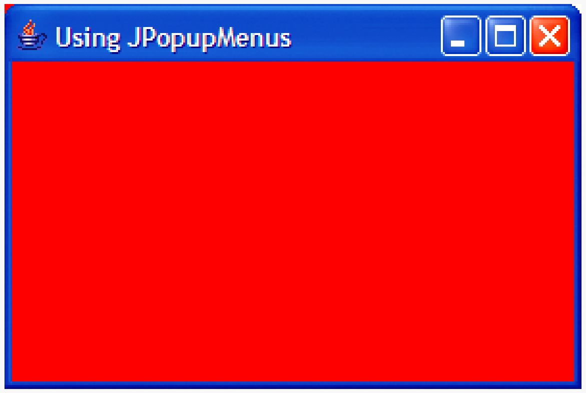
Origin component and coordinates arguments determine where the **`JPopupMenu`** will appear

```
   // private inner class to handle menu item events
   private class ItemHandler implements ActionListener
   {
      // process menu item selections
      public void actionPerformed( ActionEvent event )
      {
         // determine which menu item was selected
         for ( int i = 0; i < items.length; i++ )
         {
            if ( event.getSource() == items[ i ] )
            {
               getContentPane().setBackground( colorValues[ i ] );
               return;
            } // end if
         } // end for
      } // end method actionPerformed
   } // end private inner class ItemHandler
} // end class PopupFrame
```

Determine which **`JRadioButtonMenuItem`** the user selected and set the background color

```java
// Fig. 22.8: PopupTest.java
// Testing PopupFrame.
import javax.swing.JFrame;

public class PopupTest
{
   public static void main( String args[] )
   {
      PopupFrame popupFrame = new PopupFrame(); // create PopupFrame
      popupFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      popupFrame.setSize( 300, 200 ); // set frame size
      popupFrame.setVisible( true ); // display frame
   } // end main
} // end class PopupTest
```





Look-and-Feel Observation 22.10

Displaying a `JPopupMenu` for the pop-up trigger event of multiple GUI components requires registering mouse-event handlers for each of those GUI components.

22.6 Pluggable Look-and-Feel

- **Java applications' appearances**
  - **A program that uses Java's Abstract Window Toolkit GUI components takes on the look-and-feel of the platform**
    - **Allows users of the application on each platform to use GUI components with which they are already familiar**
    - **Also introduces interesting portability issues**
  - **Swing's lightweight GUI components provide uniform functionality**
    - **Define a uniform cross-platform look-and-feel (known as the metal look-and-feel)**
    - **Also can customize the look-and-feel to appear as a Microsoft Windows-style, Motif-style (UNIX) or Macintosh look-and-feel**

Portability Tip 22.1

GUI components look different on different platforms and may require different amounts of space to display. This could change their layout and alignments.

Portability Tip 22.2

GUI components on different platforms have different default functionality (e.g., some platforms allow a button with the focus to be “pressed” with the space bar, and some do not).

```
// Fig. 22.9: LookAndFeelFrame.java
// Changing the look and feel.
import java.awt.GridLayout;
import java.awt.BorderLayout;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;
import javax.swing.JFrame;
import javax.swing.UIManager;
import javax.swing.JRadioButton;
import javax.swing.ButtonGroup;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JComboBox;
import javax.swing.JPanel;
import javax.swing.SwingConstants;
import javax.swing.SwingUtilities;

public class LookAndFeelFrame extends JFrame
{
   // string names of look and feels
   private final String strings[] = { "Metal", "Motif", "Windows" };
   private UIManager.LookAndFeelInfo looks[]; // look and feels
   private JRadioButton radio[]; // radiobuttons to select look and feel
   private ButtonGroup group; // group for radiobuttons
   private JButton button; // displays look of button
   private JLabel label; // displays look of label
   private JComboBox comboBox; // displays look of combo box

   // set up GUI
   public LookAndFeelFrame()
   {
      super( "Look and Feel Demo" );

      JPanel northPanel = new JPanel(); // create north panel
      northPanel.setLayout( new GridLayout( 3, 1, 0, 5 ) );

      label = new JLabel( "This is a Metal look-and-feel",
         SwingConstants.CENTER ); // create label
      northPanel.add( label ); // add label to panel

      button = new JButton( "JButton" ); // create button
      northPanel.add( button ); // add button to panel

      comboBox = new JComboBox( strings ); // create combobox
      northPanel.add( comboBox ); // add combobox to panel

      // create array for radio buttons
      radio = new JRadioButton[ strings.length ];

      JPanel southPanel = new JPanel(); // create south panel
      southPanel.setLayout( new GridLayout( 1, radio.length ) );

      group = new ButtonGroup(); // button group for look and feels
      ItemHandler handler = new ItemHandler(); // look and feel handler

      for ( int count = 0; count < radio.length; count++ )
      {
         radio[ count ] = new JRadioButton( strings[ count ] );
         radio[ count ].addItemListener( handler ); // add handler
         group.add( radio[ count ] ); // add radiobutton to group
         southPanel.add( radio[ count ] ); // add radiobutton to panel
      } // end for

      add( northPanel, BorderLayout.NORTH )
      add( southPanel, BorderLayout.SOUTH )

      // get installed look-and-feel information
      looks = UIManager.getInstalledLookAndFeels();
      radio[ 0 ].setSelected( true ); // set default selection
   } // end LookAndFeelFrame constructor

   // use UIManager to change look-and-feel of GUI
   private void changeTheLookAndFeel( int value )
   {
      try // change look and feel
      {
         // set look and feel for this application
         UIManager.setLookAndFeel( looks[ value ].getClassName() );

         // update components in this application
         SwingUtilities.updateComponentTreeUI( this );
      } // end try
```

Get the array of **`UIManager.LookAndFeelInfo`** objects that describe each look-and-feel available on your system

Invoke **`static`** method **`setLookAndFeel`** to change the look-and-feel

Invoke **`static`** method **`updateComponentTreeUI`** to change the look-and-feel of every GUI component attached to the application

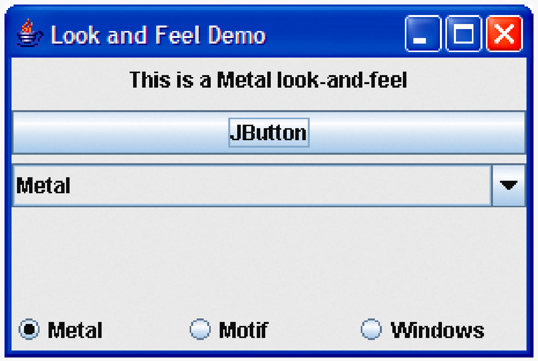
```
      catch ( Exception exception )
      {
         exception.printStackTrace();
      } // end catch
   } // end method changeTheLookAndFeel

   // private inner class to handle radio button events
   private class ItemHandler implements ItemListener
   {
      // process user's look-and-feel selection
      public void itemStateChanged( ItemEvent event )
      {
         for ( int count = 0; count < radio.length; count++ )
         {
            if ( radio[ count ].isSelected() )
            {
               label.setText( String.format( "This is a %s look-and-feel",
                  strings[ count ] ) );
               comboBox.setSelectedIndex( count ); // set combobox index
               changeTheLookAndFeel( count ); // change look and feel
            } // end if
         } // end for
      } // end method itemStateChanged
   } // end private inner class ItemHandler
} // end class LookAndFeelFrame
```

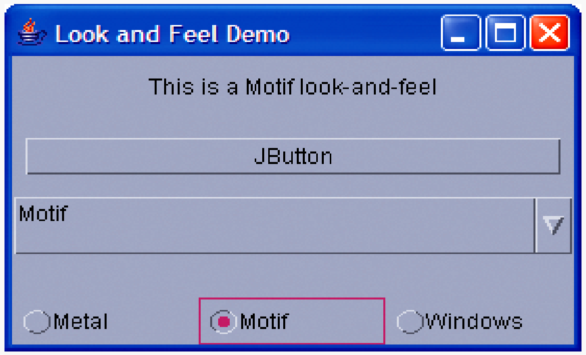
Call utility method **`changeTheLookAndFeel`**

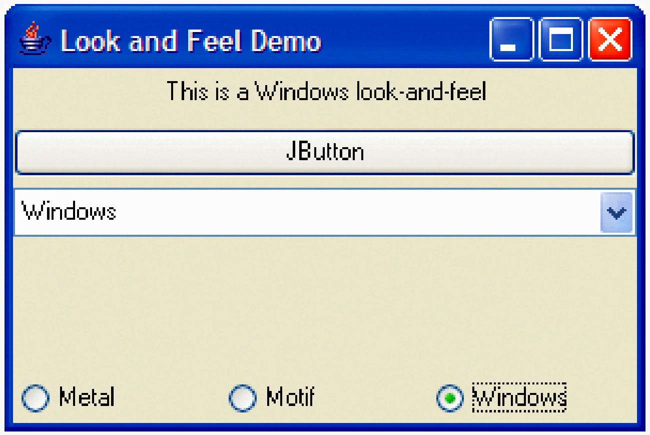
Performance Tip 22.1

Each look-and-feel is represented by a Java class. `UIManager` method `getInstalledLookAnd-Feels` does not load each class. Rather, it provides the names of the available look-and-feel classes so that a choice can be made (presumably once at program start-up). This reduces the overhead of having to load all the look-and-feel classes even if the program will not use some of them.

```java
// Fig. 22.10: LookAndFeelDemo.java
// Changing the look and feel.
import javax.swing.JFrame;

public class LookAndFeelDemo
{
   public static void main( String args[] )
   {
      LookAndFeelFrame lookAndFeelFrame = new LookAndFeelFrame();
      lookAndFeelFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      lookAndFeelFrame.setSize( 300, 200 ); // set frame size
      lookAndFeelFrame.setVisible( true ); // display frame
   } // end main
} // end class LookAndFeelDemo
```






Look and Feel Demo
This is a Windows look-and-feel
JButton
Windows
Metal
Motif
Windows

22.7 `JDesktopPane` and `JInternalFrame`

- **Multiple-document interface**
  - **A main window (called the parent window) contains other windows (called child windows)**
  - **Manages several open documents that are being processed in parallel**
  - **Implemented by Swing's `JDesktopPane` and `JInternalFrame`**

```
// Fig. 22.11: DesktopFrame.java
// Demonstrating JDesktopPane.
import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.util.Random;
import javax.swing.JFrame;
import javax.swing.JDesktopPane;
import javax.swing.JMenuBar;
import javax.swing.JMenu;
import javax.swing.JMenuItem;
import javax.swing.JInternalFrame;
import javax.swing.JPanel;
import javax.swing.ImageIcon;

public class DesktopFrame extends JFrame
{
   private JDesktopPane theDesktop;

   // set up GUI
   public DesktopFrame()
   {
      super( "Using a JDesktopPane" );

      JMenuBar bar = new JMenuBar(); // create menu bar
      JMenu addMenu = new JMenu( "Add" ); // create Add menu
      JMenuItem newFrame = new JMenuItem( "Internal Frame" );

```

Create a **JMenuBar**, a **JMenu** and a **JMenuItem**

```
      addMenu.add( newFrame ); // add new frame item to Add menu
      bar.add( addMenu ); // add Add menu to menu bar
      setJMenuBar( bar ); // set menu bar for this appl

      theDesktop = new JDesktopPane(); // create deskto
      add( theDesktop ); // add desktop pane to frame

      // set up listener for newFrame menu item
      newFrame.addActionListener(

         new ActionListener() // anonymous inner class
         {
            // display new internal window
            public void actionPerformed( ActionEvent event )
            {
               // create internal frame
               JInternalFrame frame = new JInternalFrame(
                  "Internal Frame", true, true, true, true );

               MyJPanel panel = new MyJPanel(); // create new panel
               frame.add( panel, BorderLayout.CENTER ); // add panel
               frame.pack(); // set internal frame to size of contents

```

Add the **`JMenuItem`** to the **`JMenu`** and the **`JMenu`** to the **`JMenuBar`** and set the **`JMenuBar`** for the application window

The **`JDesktopPane`** will be used to manage the **`JInternalFrame`** child windows

Create a **`JInternalFrame`** object

Constructor arguments specify title bar string and whether or not the user can resize, close, maximize and minimize the internal frame

Set the size of the child window

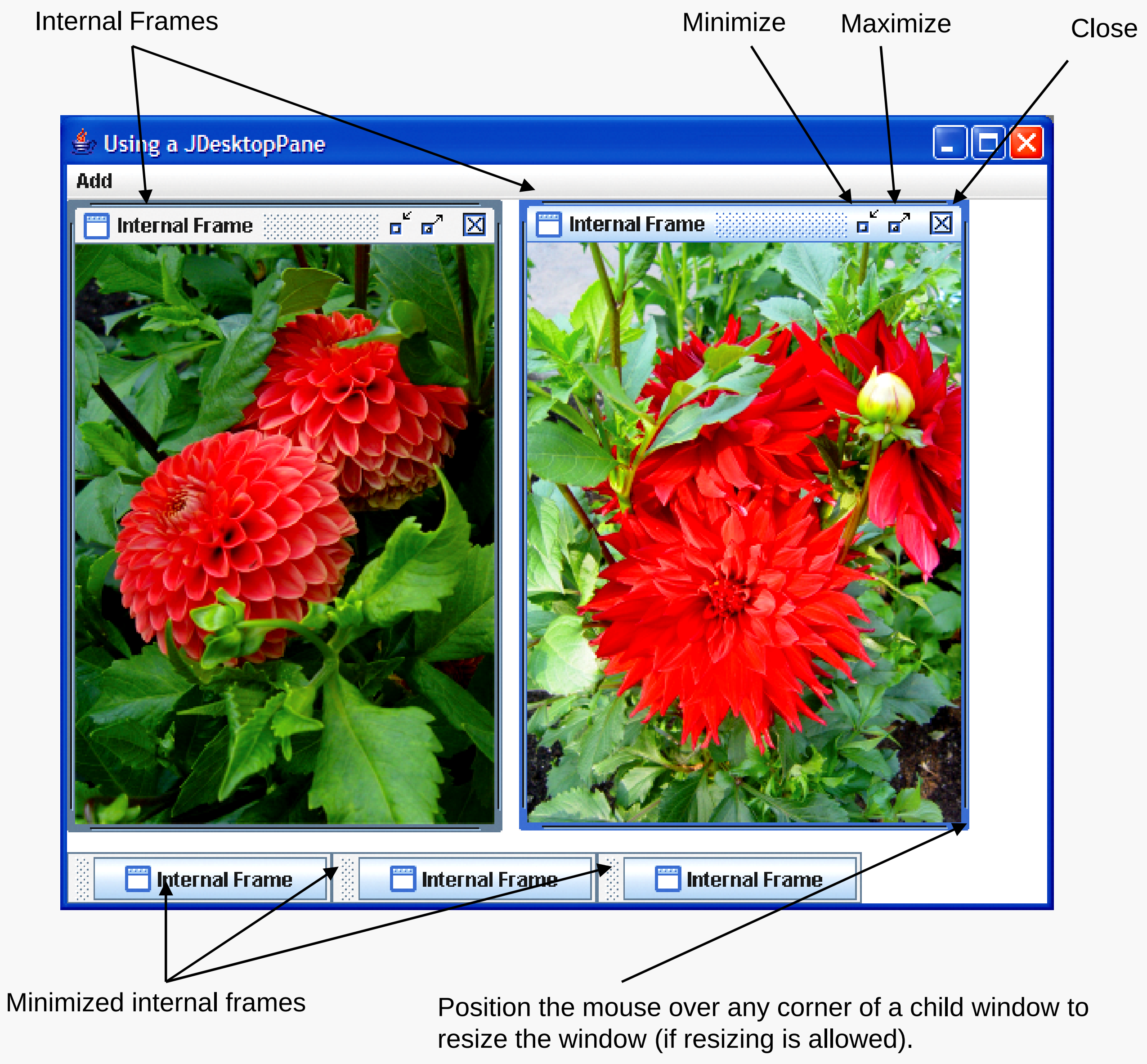
```
               theDesktop.add( frame ); // attach internal frame
               frame.setVisible( true ); // show internal frame
            } // end method actionPerformed
         } // end anonymous inner class
      ); // end call to addActionListener
   } // end DesktopFrame constructor
} // end class DesktopFrame

// class to display an ImageIcon on a panel
class MyJPanel extends JPanel
{
   private static Random generator = new Random();
   private ImageIcon picture; // image to be displayed
   private String[] images = { "yellowflowers.png", "purpleflowers.png",
      "redflowers.png", "redflowers2.png", "lavenderflowers.png" };

   // load image
   public MyJPanel()
   {
      int randomNumber = generator.nextInt( 5 );
      picture = new ImageIcon( images[ randomNumber ] ); // set icon
   } // end MyJPanel constructor

```

Add the **`JInternalFrame`** to **`theDesktop`** and display the **`JInternalFrame`**

```java
   // display imageIcon on panel
   public void paintComponent( Graphics g )
   {
      super.paintComponent( g );
      picture.paintIcon( this, g, 0, 0 ); // display icon
   } // end method paintComponent

   // return image dimensions
   public Dimension getPreferredSize()
   {
      return new Dimension( picture.getIconWidth(),
         picture.getIconHeight() );
   } // end method getPreferredSize
} // end class MyJPanel
```

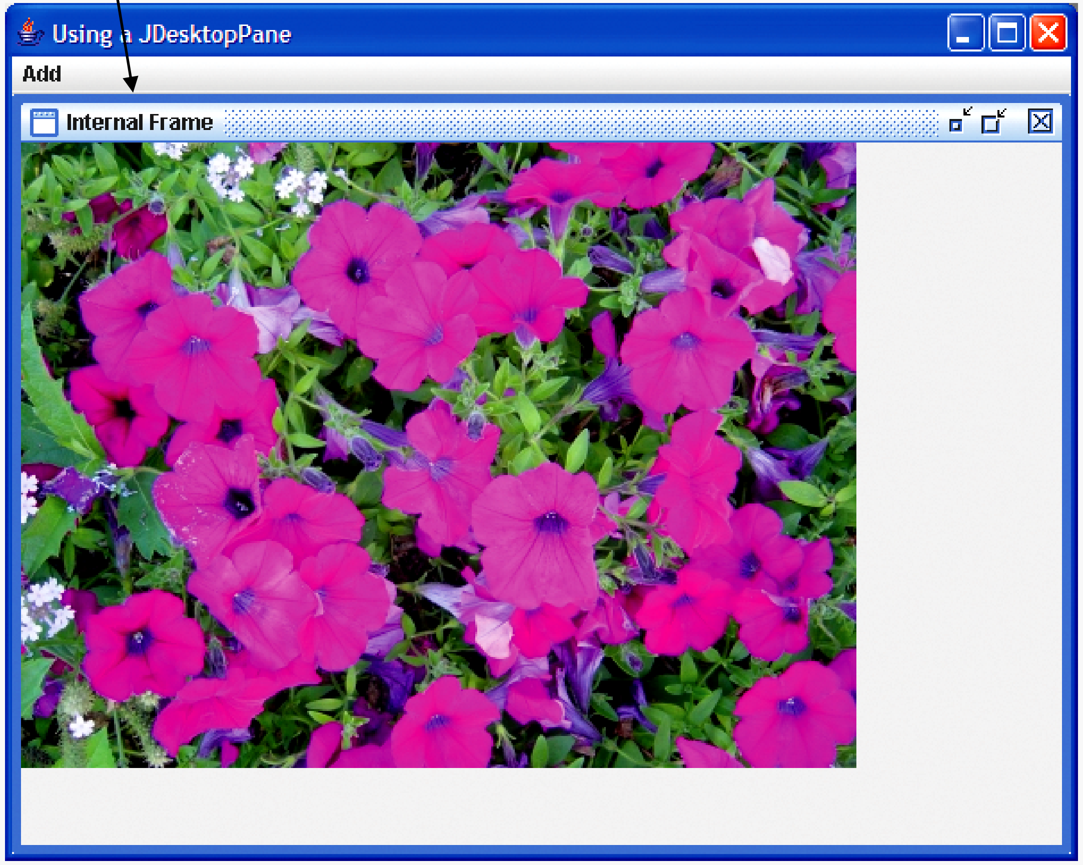
Specify the panel's preferred size for use by the **pack** method

```
// Fig. 22.12: DesktopTest.java
// Demonstrating JDesktopPane.
import javax.swing.JFrame;

public class DesktopTest
{
   public static void main( String args[] )
   {
      DesktopFrame desktopFrame = new DesktopFrame();
      desktopFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      desktopFrame.setSize( 600, 480 ); // set frame size
      desktopFrame.setVisible( true ); // display frame
   } // end main
} // end class DesktopTest
```



Position the mouse over any corner of a child window to resize the window (if resizing is allowed).

Maximized internal frame



22.8 `JTabbedPane`

- **`JTabbedPane`**
  - **Arranges GUI components into layers in which only one layer is visible at a time**
    - **When the user clicks a tab, the appropriate layer is displayed**
      - **The tabs can be positioned at top (default), left, right or bottom**
      - **Any component can be placed on a tab**
      - **If the tabs do not fit on one line, they will wrap to form additional lines of tabs**

```java
// Fig. 22.13: JTabbedPaneFrame.java
// Demonstrating JTabbedPane.
import java.awt.BorderLayout;
import java.awt.Color;
import javax.swing.JFrame;
import javax.swing.JTabbedPane;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JButton;
import javax.swing.SwingConstants;

public class JTabbedPaneFrame extends JFrame
{
   // set up GUI
   public JTabbedPaneFrame()
   {
      super( "JTabbedPane Demo " );

      JTabbedPane tabbedPane = new JTabbedPane(); // create JTabbedPane

      // set up panel1 and add it to JTabbedPane
      JLabel label1 = new JLabel( "panel one", SwingConstants.CENTER );
      JPanel panel1 = new JPanel(); // create first panel
      panel1.add( label1 ); // add label to panel
      tabbedPane.addTab( "Tab One", null, panel1, "First Panel" );

```

Create an empty **JTabbedPane** with default settings

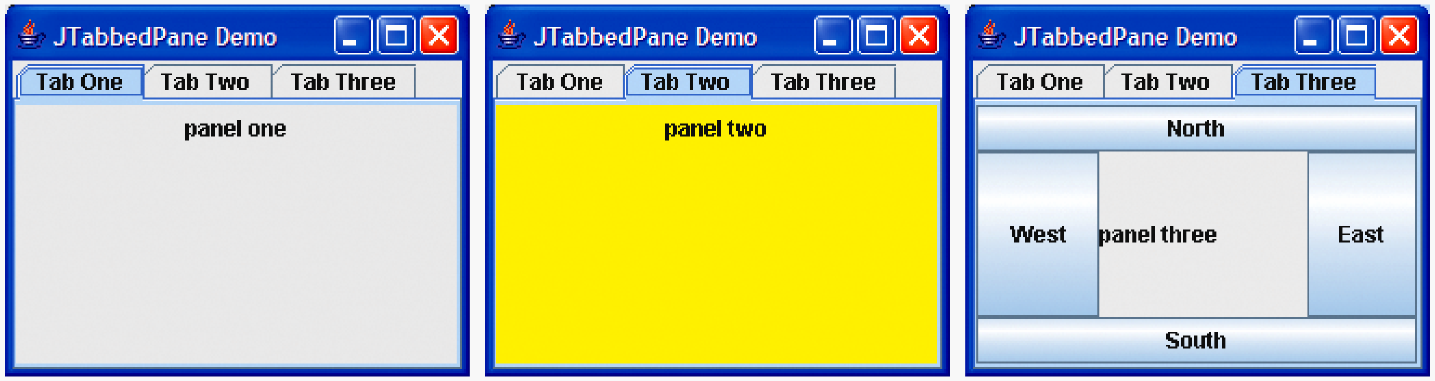
Call **JTabbedPane** method **addTab** with arguments that specify the tab's string title, an **Icon** reference to display on the tab, the **COMPONENT** to display when the user clicks on the tab and the tab's tooltip string

```
      // set up panel2 and add it to JTabbedPane
      JLabel label2 = new JLabel( "panel two", SwingConstants.CENTER );
      JPanel panel2 = new JPanel(); // create second panel
      panel2.setBackground( Color.YELLOW ); // set background to yellow
      panel2.add( label2 ); // add label to panel
      tabbedPane.addTab( "Tab Two", null, panel2, "Second Panel" );

      // set up panel3 and add it to JTabbedPane
      JLabel label3 = new JLabel( "panel three" );
      JPanel panel3 = new JPanel(); // create third panel
      panel3.setLayout( new BorderLayout() ); // use borderlayout
      panel3.add( new JButton( "North" ), BorderLayout.NORTH );
      panel3.add( new JButton( "West" ), BorderLayout.WEST );
      panel3.add( new JButton( "East" ), BorderLayout.EAST );
      panel3.add( new JButton( "South" ), BorderLayout.SOUTH );
      panel3.add( label3, BorderLayout.CENTER );
      tabbedPane.addTab( "Tab Three", null, panel3, "Third Panel" );

      add( tabbedPane ); // add JTabbedPane to frame
   } // end JTabbedPaneFrame constructor
} // end class JTabbedPaneFrame
```

Add **`panel2`** to **`tabbedPane`**

Add **`panel3`** to **`tabbedPane`**

```
// Fig. 22.14: JTabbedPaneDemo.java
// Demonstrating JTabbedPane.
import javax.swing.JFrame;

public class JTabbedPaneDemo
{
   public static void main( String args[] )
   {
      JTabbedPaneFrame tabbedPaneFrame = new JTabbedPaneFrame();
      tabbedPaneFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      tabbedPaneFrame.setSize( 250, 200 ); // set frame size
      tabbedPaneFrame.setVisible( true ); // display frame
   } // end main
} // end class JTabbedPaneDemo
```



22.9 Layout Managers: `BoxLayout` and `GridBagLayout`

- `BoxLayout` Layout Manager
  - Arranges GUI components horizontally along a container’s *x*-axis or vertically along its *y*-axis

Layout Manager	Description
`BoxLayout`	A layout manager that allows GUI components to be arranged left-to-right or top-to-bottom in a container. Class `Box` declares a container with `BoxLayout` as its default layout manager and provides `static` methods to create a `Box` with a horizontal or vertical `BoxLayout`.
`GridBagLayout`	A layout manager similar to `GridLayout`, but unlike it in that components can vary in size and can be added in any order.

Fig. 22.15 | Additional layout managers.

```java
// Fig. 22.16: BoxLayoutFrame.java
// Demonstrating BoxLayout.
import java.awt.Dimension;
import javax.swing.JFrame;
import javax.swing.Box;
import javax.swing.JButton;
import javax.swing.BoxLayout;
import javax.swing.JPanel;
import javax.swing.JTabbedPane;

public class BoxLayoutFrame extends JFrame
{
   // set up GUI
   public BoxLayoutFrame()
   {
      super( "Demonstrating BoxLayout" );

      // create Box containers with BoxLayout
      Box horizontal1 = Box.createHorizontalBox();
      Box vertical1 = Box.createVerticalBox();
      Box horizontal2 = Box.createHorizontalBox();
      Box vertical2 = Box.createVerticalBox();

      final int SIZE = 3; // number of buttons on each Box

      // add buttons to Box horizontal1
      for ( int count = 0; count < SIZE; count++ )
         horizontal1.add( new JButton( "Button " + count ) );

```

Create **`Box`** containers with **`static Box`** methods **`createHorizontalBox`** and **`createVerticalBox`**

Add three **`JButton`**s to **`horizontal1`**

```
      // create strut and add buttons to Box vertical1
      for ( int count = 0; count < SIZE; count++ )
      {
         vertical1.add( Box.createVerticalStrut( 25 ) );
         vertical1.add( new JButton( "Button " + count ) );
      } // end for

      // create horizontal glue and add buttons to Box horizontal2
      for ( int count = 0; count < SIZE; count++ )
      {
         horizontal2.add( Box.createHorizontalGlue() );
         horizontal2.add( new JButton( "Button " + count ) );
      } // end for

      // create rigid area and add buttons to Box vertical2
      for ( int count = 0; count < SIZE; count++ )
      {
         vertical2.add( Box.createRigidArea( new Dimension( 12, 8 ) ) );
         vertical2.add( new JButton( "Button " + count ) );
      } // end for

      // create vertical glue and add buttons to panel
      JPanel panel = new JPanel();
      panel.setLayout( new BoxLayout( panel, BoxLayout.Y_AXIS ) );

```

Add three vertical struts and three **`JButton`**s to **`vertical1`**

Add horizontal glue and three **`JButton`**s to **`horizontal2`**

Add three rigid areas and three **`JButton`**s to **`vertical2`**

Use **`Container`** method **`setLayout`** to set **`panel`**’s layout to a vertical **`BoxLayout`**

```
      for ( int count = 0; count < SIZE; count++ )
      {
         panel.add( Box.createGlue() );
         panel.add( new JButton( "Button " + count ) );
      } // end for

      // create a JTabbedPane
      JTabbedPane tabs = new JTabbedPane(
         JTabbedPane.TOP, JTabbedPane.SCROLL_TAB_LAYOUT );

      // place each container on tabbed pane
      tabs.addTab( "Horizontal Box", horizontal1 );
      tabs.addTab( "Vertical Box with Struts", vertical1 );
      tabs.addTab( "Horizontal Box with Glue", horizontal2 );
      tabs.addTab( "Vertical Box with Rigid Areas", vertical2 );
      tabs.addTab( "Vertical Box with Glue", panel );

      add( tabs ); // place tabbed pane on frame
   } // end BoxLayoutFrame constructor
} // end class BoxLayoutFrame
```

Add glue and three **`JButton`**s to **`panel`**

Create a **`JTabbedPane`** where the tabs should scroll if there are too many tabs to fit on one line

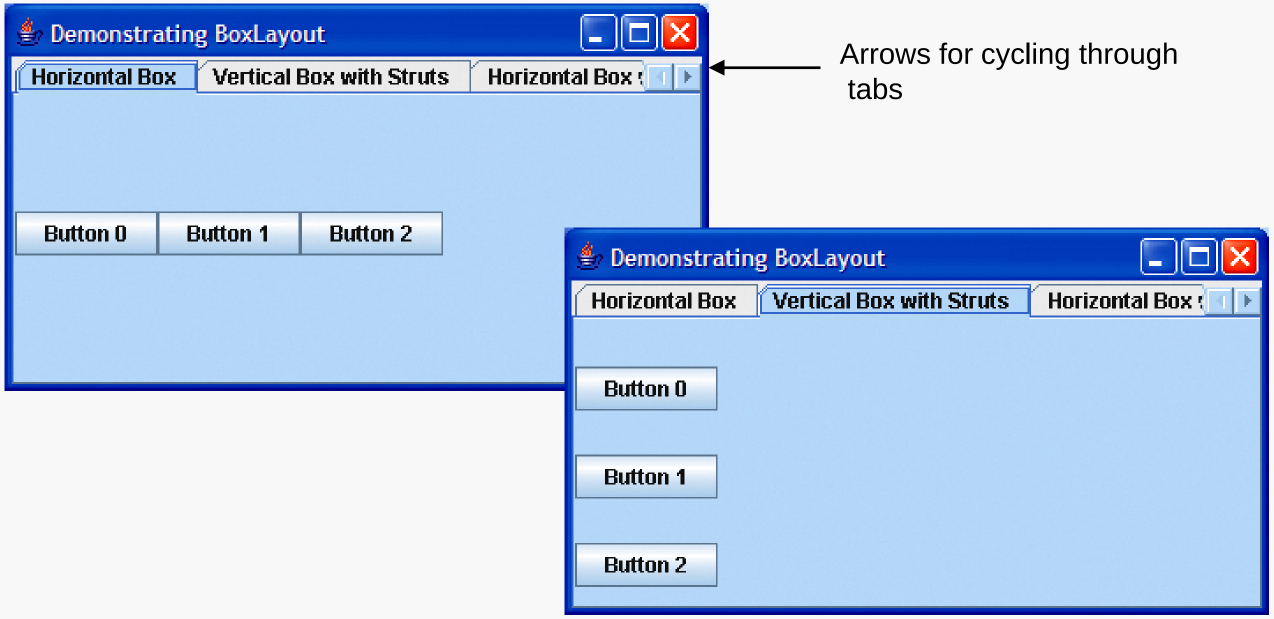
22.9 Layout Managers: `BoxLayout` and `GridBagLayout` (Cont.)

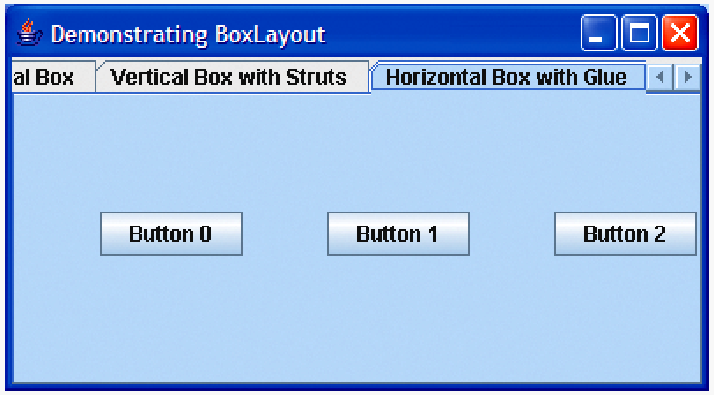
- **Vertical struts**
  - **Invisible GUI component that has a fixed pixel height**
    - **Used to guarantee a fixed amount of space between GUI components**
    - **`static` `Box` method `createVerticalStrut`**
      - **`int` argument determines the height of the strut in pixels**
    - **`Box` also declares method `createHorizontalStrut`**

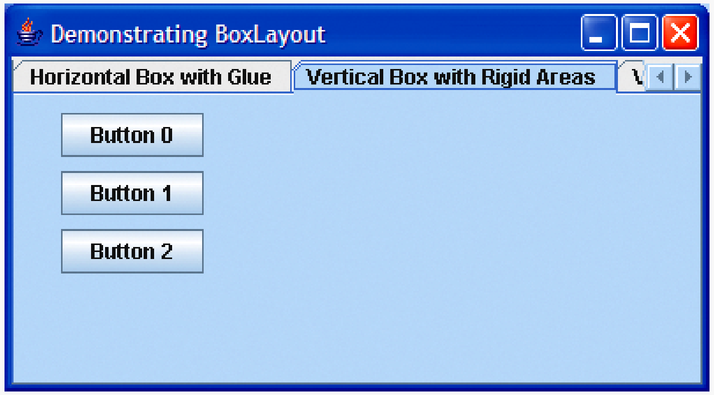
22.9 Layout Managers: `BoxLayout` and `GridBagLayout` (Cont.)

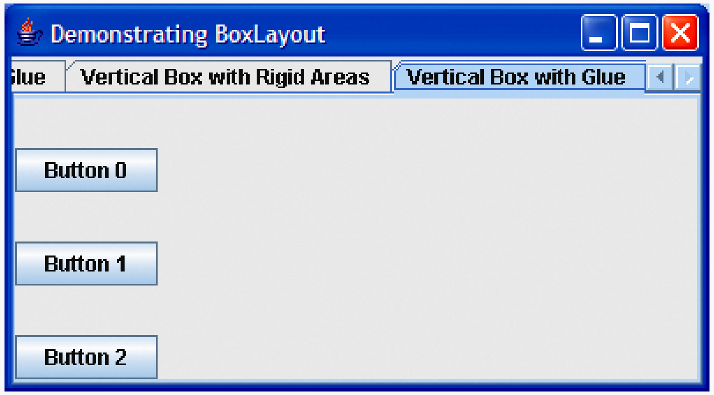
- **Horizontal glue**
  - **An invisible GUI component that occupies additional space between fixed-size GUI components**
    - **When the container is resized, components separated by glue remain the same size, but the glue stretches or contracts to occupy the space between them**
  - **`static` `Box` methods `createHorizontalGlue` and `createVerticalGlue`**

```java
// Fig. 22.17: BoxLayoutDemo.java
// Demonstrating BoxLayout.
import javax.swing.JFrame;

public class BoxLayoutDemo
{
   public static void main( String args[] )
   {
      BoxLayoutFrame boxLayoutFrame = new BoxLayoutFrame();
      boxLayoutFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      boxLayoutFrame.setSize( 400, 220 ); // set frame size
      boxLayoutFrame.setVisible( true ); // display frame
   } // end main
} // end class BoxLayoutDemo
```




Demonstrating BoxLayout
al Box
Vertical Box with Struts
Horizontal Box with Glue
Button 0
Button 1
Button 2


Demonstrating BoxLayout
Horizontal Box with Glue
Vertical Box with Rigid Areas
Button 0
Button 1
Button 2


Demonstrating BoxLayout
lue
Vertical Box with Rigid Areas
Vertical Box with Glue
Button 0
Button 1
Button 2

22.9 Layout Managers: `BoxLayout` and `GridBagLayout` (Cont.)

- **Rigid areas**
  - **An invisible GUI component that always has a fixed pixel width and height**
    - **`Dimension` object argument to `static Box` method `createRigidArea` specifies the area's width and height**

22.9 Layout Managers: `BoxLayout` and `GridBagLayout` (Cont.)

- `GridBagLayout` Layout Manager
  - Similar to `GridLayout` in that it arranges components in a grid, but more flexible
    - The components can vary in size and can be added in any order
  - Determining the appearance of the GUI
    - Draw the GUI on paper
    - Draw a grid over it, dividing the components into rows and columns
      - The initial row and column numbers should be 0
      - Used by the `GridBagLayout` layout manager to properly place the components in the grid

22.9 Layout Managers: `BoxLayout` and `GridBagLayout` (Cont.)

- **`GridBagConstraints` object**
  - **Describes how a component is placed in a `GridBagLayout`**
  - **`anchor` specifies the relative position of the component in an area that it does not fill**
    - **Constants: `NORTH`, `NORTHEAST`, `EAST`, `SOUTHEAST`, `SOUTH`, `SOUTHWEST`, `WEST`, `NORTHWEST` and `CENTER` (the default)**
  - **`fill` defines how the component grows if the area in which it can be displayed is larger than the component**
    - **Constants: `NONE` (the default), `VERTICAL`, `HORIZONTAL` and `BOTH`**

22.9 Layout Managers: `BoxLayout` and `GridBagLayout` (Cont.)

- `gridx` and `gridy` specify where the upper-left corner of the component is placed in the grid
- `gridwidth` and `gridheight` specify the number of columns and rows a component occupies
- `weightx` and `weighty` specify how to distribute extra horizontal and vertical space to grid slots in a `GridBagLayout` when the container is resized
  - A zero value indicates that the grid slot does not grow in that dimension on its own
    - However, if the component spans a column/row containing a component with nonzero weight value, it will grow in the same proportion as the other components in that column/row
  - Use positive nonzero weight values to prevent “huddling”

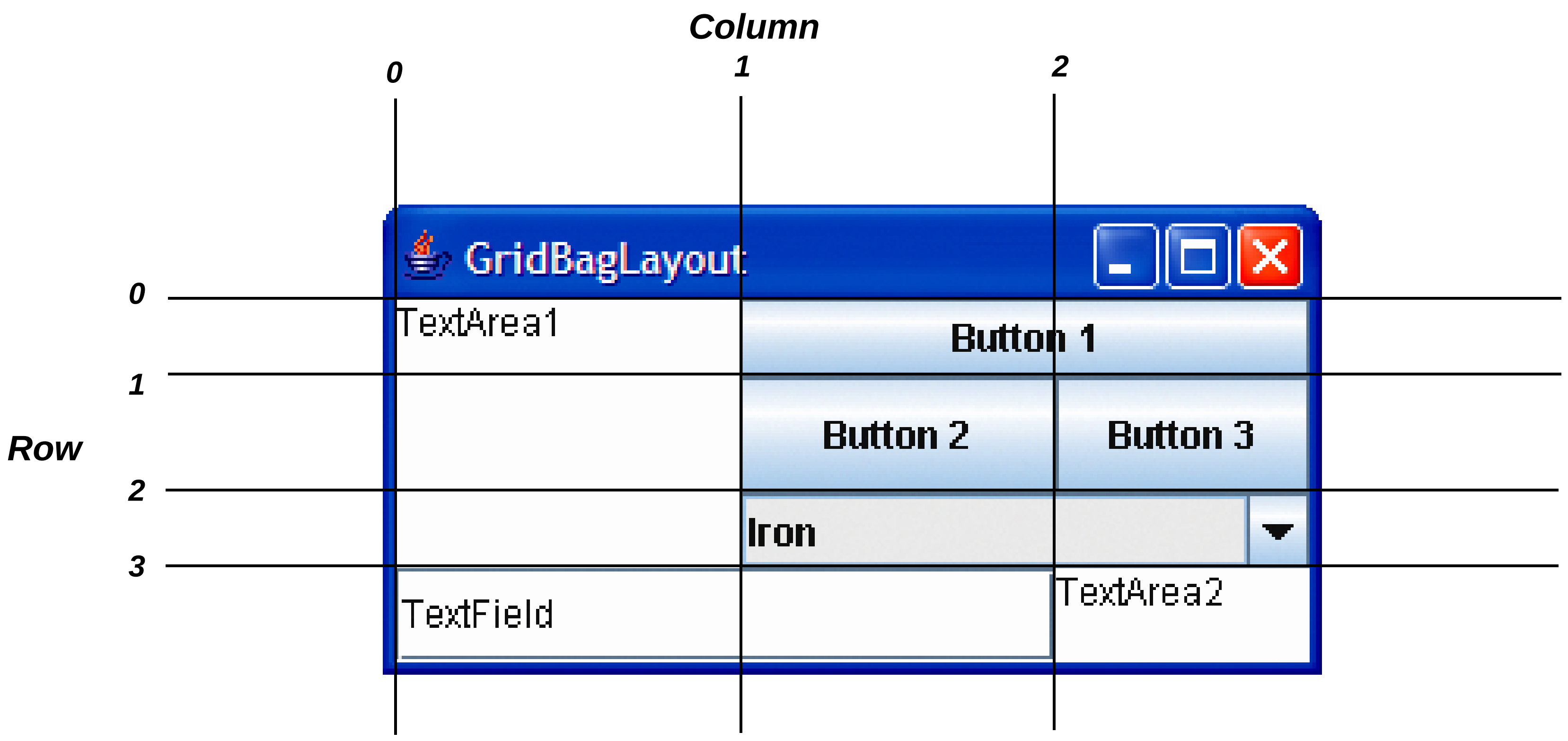
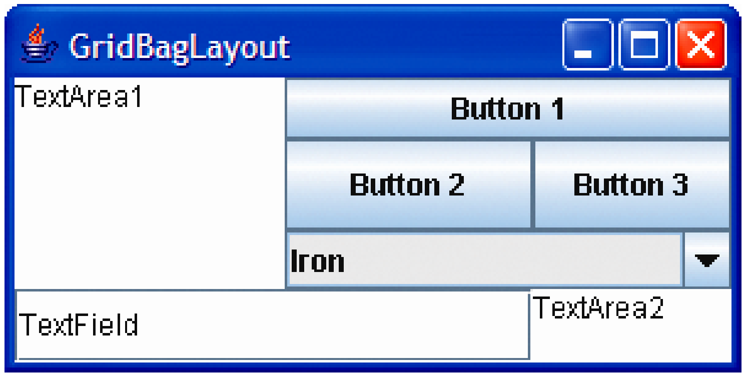


Fig. 22.18 | Designing a GUI that will use `GridBagLayout`.

GridBagCons-traints field	Description
anchor	Specifies the relative position (NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST, NORTHWEST, CENTER) of the component in an area that it does not fill.
fill	Resizes the component in specified direction (NONE, HORIZONTAL, VERTICAL, BOTH) when the display area is larger than the component.
gridx	The column in which the component will be placed.
gridy	The row in which the component will be placed.
gridwidth	The number of columns the component occupies.
gridheight	The number of rows the component occupies.
weightx	The amount of extra space to allocate horizontally. The grid slot can become wider when extra space is available.
weighty	The amount of extra space to allocate vertically. The grid slot can become taller when extra space is available.

Fig. 22.19 | GridBagConstraints fields.



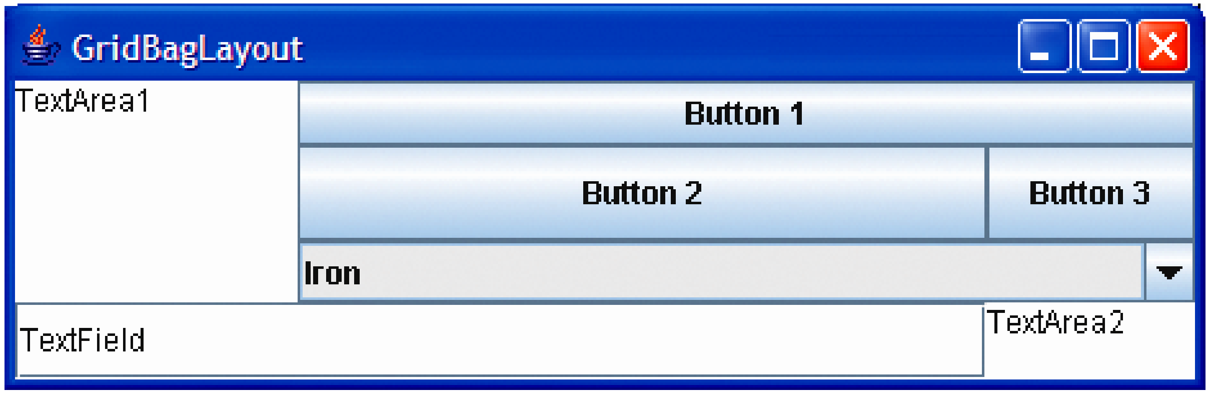


Fig. 22.20 | `GridBagLayout` **with the weights set to zero.**

```java
// Fig. 22.21: GridBagFrame.java
// Demonstrating GridBagLayout.
import java.awt.GridBagLayout;
import java.awt.GridBagConstraints;
import java.awt.Component;
import javax.swing.JFrame;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.JButton;
import javax.swing.JComboBox;

public class GridBagFrame extends JFrame
{
   private GridBagLayout layout; // layout of this frame
   private GridBagConstraints constraints; // constraints of this layout

   // set up GUI
   public GridBagFrame()
   {
      super( "GridBagLayout" );
      layout = new GridBagLayout();
      setLayout( layout ); // set frame layout
      constraints = new GridBagConstraints(); // instantiate constraints

      // create GUI components
      JTextArea textArea1 = new JTextArea( "TextArea1", 5, 10 );
      JTextArea textArea2 = new JTextArea( "TextArea2", 2, 2 );

```

Create a **GridBagLayout** object

Create a **GridBagConstraints** object

```
        String names[] = { "Iron", "Steel", "Brass" };
        JComboBox comboBox = new JComboBox( names );

        JTextField textField = new JTextField( "TextField" );
        JButton button1 = new JButton( "Button 1" );
        JButton button2 = new JButton( "Button 2" );
        JButton button3 = new JButton( "Button 3" );

        // weightx and weighty for textArea1 are both 0: the default
        // anchor for all components is CENTER: the default
        constraints.fill = GridBagConstraints.BOTH;
        addComponent( textArea1, 0, 0, 1, 3 );

        // weightx and weighty for button1 are both 0: t
        constraints.fill = GridBagConstraints.HORIZONTAL
        addComponent( button1, 0, 1, 2, 1 );

        // weightx and weighty for comboBox are both 0: the default
        // fill is HORIZONTAL
        addComponent( comboBox, 2, 1, 2, 1 );

        // button2
        constraints.weightx = 1000;  // can grow wider
        constraints.weighty = 1;     // can grow taller
        constraints.fill = GridBagConstraints.BOTH;
        addComponent( button2, 1, 1, 1, 1 );

```

Cause the **`JTextArea`** to always fill its entire allocated area

Call utility method **`addComponent`** with the **`JTextArea`** object, row, column and numbers of columns and rows to span as arguments

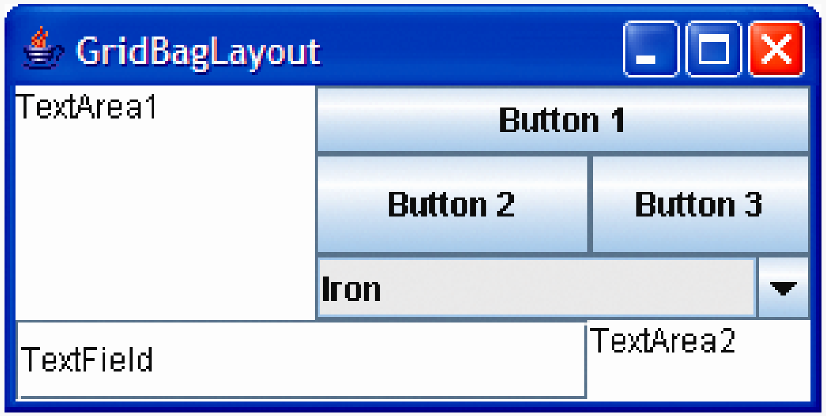
When the window is resized, **`button2`** will grow

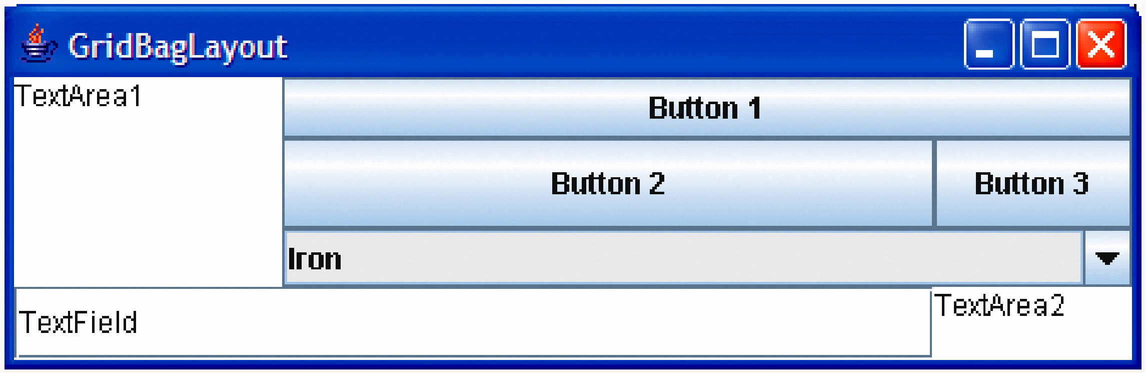
```
      // fill is BOTH for button3
      constraints.weightx = 0;
      constraints.weighty = 0;
      addComponent( button3, 1, 2, 1, 1 );

      // weightx and weighty for textField are both 0, fill is BOTH
      addComponent( textField, 3, 0, 2, 1 );

      // weightx and weighty for textArea2 are both 0, fill is BOTH
      addComponent( textArea2, 3, 2, 1, 1 );
   } // end GridBagFrame constructor

   // method to set constraints on
   private void addComponent( Component component,
      int row, int column, int width, int height )
   {
      constraints.gridx = column; // set gridx
      constraints.gridy = row; // set gridy
      constraints.gridwidth = width; // set gridwidth
      constraints.gridheight = height; // set gridheight
      layout.setConstraints( component, constraints ); // set constraints
      add( component ); // add component
   } // end method addComponent
} // end class GridBagFrame
```

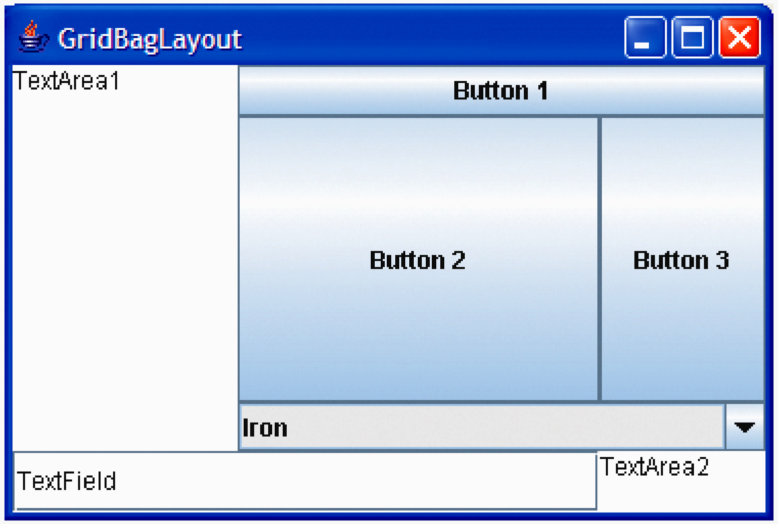
button3 will still grow because of the weight values of **button2**

Set constraints and add component

```java
// Fig. 22.22: GridBagDemo.java
// Demonstrating GridBagLayout.
import javax.swing.JFrame;

public class GridBagDemo
{
   public static void main( String args[] )
   {
      GridBagFrame gridBagFrame = new GridBagFrame();
      gridBagFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      gridBagFrame.setSize( 300, 150 ); // set frame size
      gridBagFrame.setVisible( true ); // display frame
   } // end main
} // end class GridBagDemo
```




GridBagLayout
TextArea1
Button 1
Button 2
Button 3
Iron
TextField
TextArea2


GridBagLayout
TextArea1
Button 1
Button 2
Button 3
Iron
TextField
TextArea2

22.9 Layout Managers: `BoxLayout` and `GridBagLayout` (Cont.)

- `GridBagConstraints` constants
  - `RELATIVE`
    - Specifies that the next-to-last component in a particular row should be placed to the right of the previous component in the row
  - `REMAINDER`
    - Specifies that a component is the last component in a row
  - Components that are not the second-to-last or last component on a row must specify values for `gridwidth` and `gridheight`

```java
// Fig. 22.23: GridBagFrame2.java
// Demonstrating GridBagLayout constants.
import java.awt.GridBagLayout;
import java.awt.GridBagConstraints;
import java.awt.Component;
import javax.swing.JFrame;
import javax.swing.JComboBox;
import javax.swing.JTextField;
import javax.swing.JList;
import javax.swing.JButton;

public class GridBagFrame2 extends JFrame
{
   private GridBagLayout layout; // layout of this frame
   private GridBagConstraints constraints; // constraints of this layout

   // set up GUI
   public GridBagFrame2()
   {
      super( "GridBagLayout" );
      layout = new GridBagLayout();
      setLayout( layout ); // set frame layout
      constraints = new GridBagConstraints(); // instantiate constraints

      // create GUI components
      String metals[] = { "Copper", "Aluminum", "Silver" };
      JComboBox comboBox = new JComboBox( metals );

      JTextField textField = new JTextField( "TextField" );

```

Create a **`GridBagLayout`** object

```
      String fonts[] = { "Serif", "Monospaced" };T
      JList list = new JList( fonts );

      String names[] = { "zero", "one", "two", "three", "four" };
      JButton buttons[] = new JButton[ names.length ];

      for ( int count = 0; count < buttons.length; count++ )
         buttons[ count ] = new JButton( names[ count ] );

      // define GUI component constraints for textField
      constraints.weightx = 1;
      constraints.weighty = 1;
      constraints.fill = GridBagConstraints.BOTH;
      constraints.gridwidth = GridBagConstraints.REMAINDER;
      addComponent( textField );

      // buttons[0] -- weightx and weighty are 1: fill is BOTH
      constraints.gridwidth = 1;
      addComponent( buttons[ 0 ] );

      // buttons[1] -- weightx and weighty are 1: fill is BOTH
      constraints.gridwidth = GridBagConstraints.RELATIVE;
      addComponent( buttons[ 1 ] );

      // buttons[2] -- weightx and weighty are 1: fill is BOTH
      constraints.gridwidth = GridBagConstraints.REMAINDER;
      addComponent( buttons[ 2 ] );

```

Specify that the **`JTextField`** is the last component on the line

Specify that the **`JButton`** is to be placed relative to the previous component

This **`JButton`** is the last component on the line

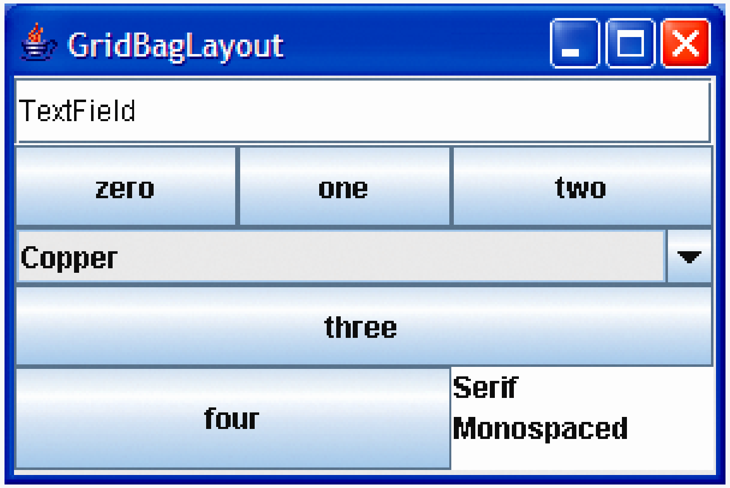
```
      // comboBox -- weightx is 1: fill is BOTH
      constraints.weighty = 0;
      constraints.gridwidth = GridBagConstraints.REMAINDER;
      addComponent( comboBox );

      // buttons[3] -- weightx is 1: fill is BOTH
      constraints.weighty = 1;
      constraints.gridwidth = GridBagConstraints.REMAINDER;
      addComponent( buttons[ 3 ] );

      // buttons[4] -- weightx and weighty are 1: fill is BOTH
      constraints.gridwidth = GridBagConstraints.RELATIVE;
      addComponent( buttons[ 4 ] );

      // list -- weightx and weighty are 1: fill is BOTH
      constraints.gridwidth = GridBagConstraints.REMAINDER;
      addComponent( list );
   } // end GridBagFrame2 constructor

   // add a component to the container
   private void addComponent( Component component )
   {
      layout.setConstraints( component, constraints );
      add( component ); // add component
   } // end method addComponent
} // end class GridBagFrame2
```

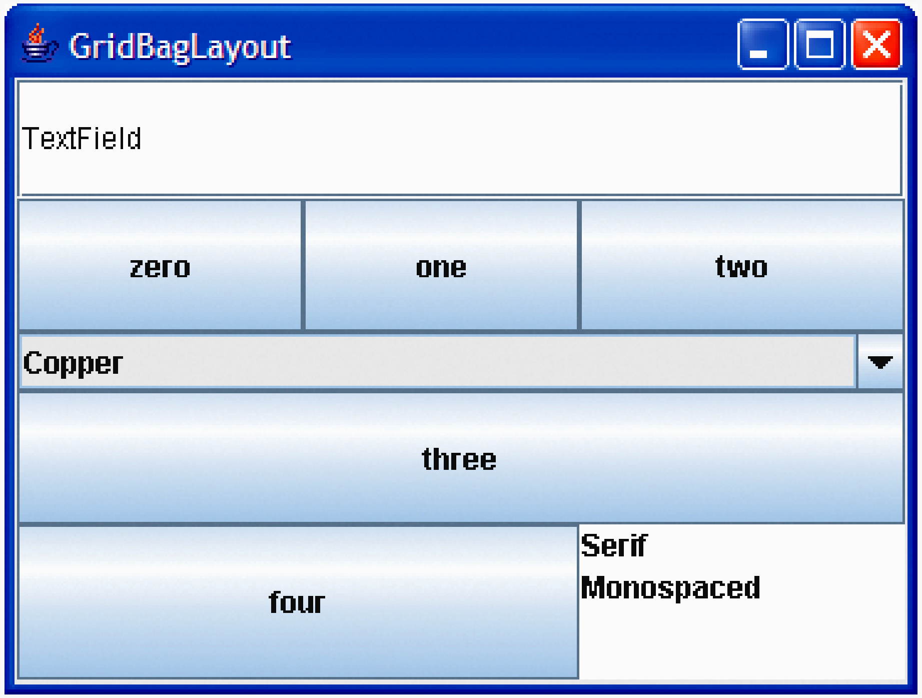
The **`JComboBox`** is the only component on the line

This **`JButton`** is the only component on the line

This **`JButton`** is the next-to-last component on the line

```
// Fig. 22.24: GridBagDemo2.java
// Demonstrating GridBagLayout constants.
import javax.swing.JFrame;

public class GridBagDemo2
{
   public static void main( String args[] )
   {
      GridBagFrame2 gridBagFrame = new GridBagFrame2();
      gridBagFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      gridBagFrame.setSize( 300, 200 ); // set frame size
      gridBagFrame.setVisible( true ); // display frame
   } // end main
} // end class GridBagDemo2
```




GridBagLayout
TextField
zero
one
two
Copper
three
four
Serif
Monospaced