Introduction to Java™

Module 12: GUIs

About the JFC and Swing

Features of the Java Foundation Classes

Feature	Description
Swing GUI Components	Includes everything from buttons to split panes to tables.
Pluggable Look-and-Feel Support	Gives any program that uses Swing components a choice of look and feel. For example, the same program can use either the Java or the Windows look and feel. Many more look-and-feel packages are available from various sources. As of v1.4.2, the Java platform supports the GTK+ look and feel, which makes hundreds of existing look and feels available to Swing programs.
Accessibility API	Enables assistive technologies, such as screen readers and Braille displays, to get information from the user interface.
Java 2D API	Enables developers to easily incorporate high-quality 2D graphics, text, and images in applications and applets. Java 2D includes extensive APIs for generating and sending high-quality output to printing devices.
Drag-and-Drop Support	Provides the ability to drag and drop between Java applications and native applications.
Internationalization	Allows developers to build applications that can interact with users worldwide in their own languages and cultural conventions. With the input method framework developers can build applications that accept text in languages that use thousands of different characters, such as Japanese, Chinese, etc.

Which Releases Contain the Swing API?

- The Swing API is powerful, flexible and immense.
  - In release 1.4 of Java, the Swing API has 17 public packages:

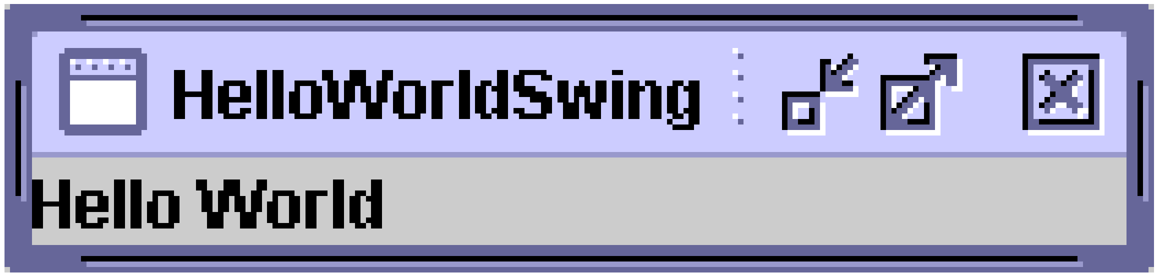
    javax.accessibility javax.swing.plaf javax.swing.text.html
    javax.swing javax.swing.plaf.basic javax.swing.text.parser
    javax.swing.border javax.swing.plaf.metal javax.swing.text.rtf
    javax.swing.text javax.swing.plaf.multi javax.swing.tree
    javax.swing.eventjavax.swing.table
    javax.swing.undojavax.swing.filechooser
    javax.swing.colorchooser
- Most programs use only a small subset of the API.
  - Usually you will use two Swing packages:
    - ✓ javax.swing
    - ✓ javax.swing.event (not always required)

Hello World in Swing

```java
import javax.swing.*;

public class HelloWorldSwing {
    /**
     * Create the GUI and show it.  For thread safety,
     * this method should be invoked from the event-dispatching thread.
     */
    private static void createAndShowGUI() {
        //Make sure we have nice window decorations.
        JFrame.setDefaultLookAndFeelDecorated(true);

        //Create and set up the window.
        JFrame frame = new JFrame("HelloWorldSwing");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //Add the ubiquitous "Hello World" label.
        JLabel label = new JLabel("Hello World");
        frame.getContentPane().add(label);

        //Display the window.
        frame.pack();
        frame.setVisible(true);
    }
```

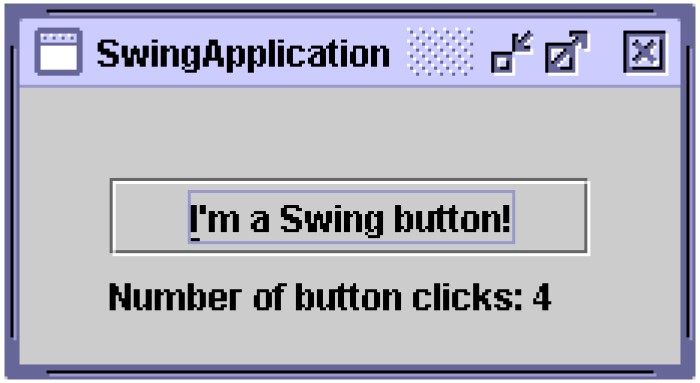
Hello World in Swing

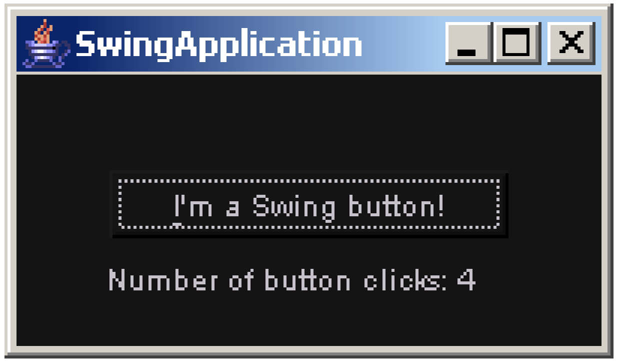
```
    public static void main(String[] args) {
        //Schedule a job for the event-dispatching thread:
        //creating and showing this application's GUI.
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                createAndShowGUI();
            }
        });
    }
}
```

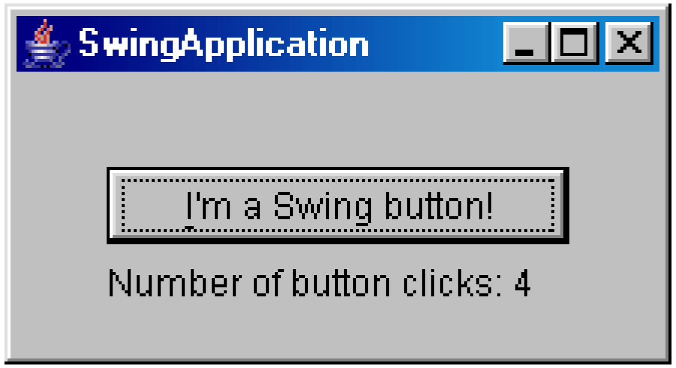


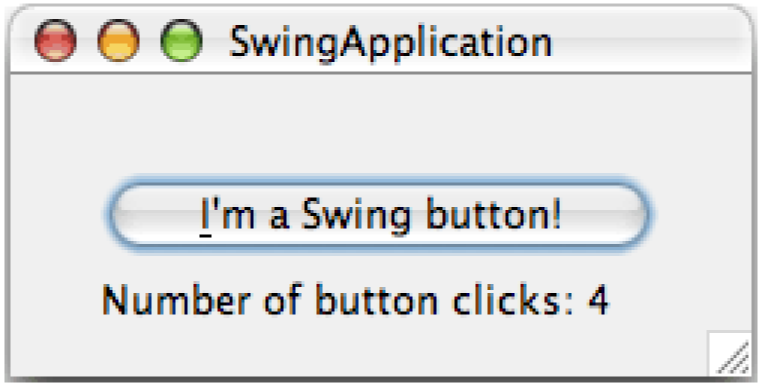
Hello World

Look and Feel









Setting Up Buttons and Labels

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

public class SwingApplication implements ActionListener {
    private static String labelPrefix = "Number of button clicks:";
    private int numClicks = 0;
    final JLabel label = new JLabel(labelPrefix + "0    ");

    //Specify the look and feel to use.  Valid values:
    //null (use the default), "Metal", "System", "Motif", "GTK+"
    final static String LOOKANDFEEL = null;
```

Setting Up Buttons and Labels

```
public Component createComponents() {
    JButton button = new JButton("I'm a Swing button!");
    button.setMnemonic(KeyEvent.VK_I);
    button.addActionListener(this);
    label.setLabelFor(button);

    /*
     * An easy way to put space between a top-level container and its
     * contents is to put the contents in a JPanel that has an "empty" border.
     */
    JPanel pane = new JPanel(new GridLayout(0, 1));
    pane.add(button);
    pane.add(label);
    pane.setBorder(BorderFactory.createEmptyBorder(
                                    30, //top
                                    30, //left
                                    10, //bottom
                                    30) //right
                                    );
    return pane;
}
```

Setting Up Buttons and Labels

```
public void actionPerformed(ActionEvent e) {
    numClicks++;
    label.setText(labelPrefix + numClicks);
}
private static void initLookAndFeel() {
    String lookAndFeel = null;
    if (LOOKANDFEEL != null) {
        if (LOOKANDFEEL.equals("Metal")) {
            lookAndFeel = UIManager.getCrossPlatformLookAndFeelClassName();
        } else if (LOOKANDFEEL.equals("System")) {
            lookAndFeel = UIManager.getSystemLookAndFeelClassName();
        } else if (LOOKANDFEEL.equals("Motif")) {
            lookAndFeel = "com.sun.java.swing.plaf.motif.MotifLookAndFeel";
        } else if (LOOKANDFEEL.equals("GTK+")) { //new in 1.4.2
            lookAndFeel = "com.sun.java.swing.plaf.gtk.GTKLookAndFeel";
        } else {
            System.err.println("Unexpected value of LOOKANDFEEL specified: "
                               + LOOKANDFEEL);
            lookAndFeel = UIManager.getCrossPlatformLookAndFeelClassName();
        }
```

Setting Up Buttons and Labels

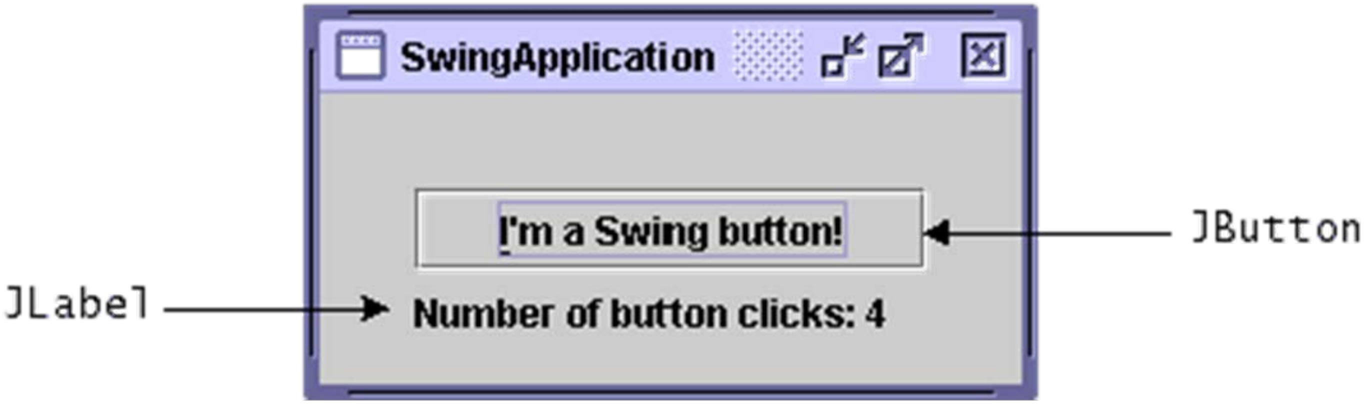
```
        try {
            UIManager.setLookAndFeel(lookAndFeel);
        } catch (ClassNotFoundException e) {
            System.err.println("Couldn't find class for specified look and
                               feel:" + lookAndFeel);
            System.err.println("Did you include the L&F library in the class
                               path?");
            System.err.println("Using the default look and feel.");
        } catch (UnsupportedLookAndFeelException e) {
            System.err.println("Can't use the specified look and feel ("
                               + lookAndFeel + ") on this platform.");
            System.err.println("Using the default look and feel.");
        } catch (Exception e) {
            System.err.println("Couldn't get specified look and feel ("
                               + lookAndFeel + "), for some reason.");
            System.err.println("Using the default look and feel.");
            e.printStackTrace();
        }
    }
}
```

Setting Up Buttons and Labels

```
/**
 * Create the GUI and show it.  For thread safety, this method
 * should be invoked from the event-dispatching thread.
 */
private static void createAndShowGUI() {
    //Set the look and feel.
    initLookAndFeel();

    //Make sure we have nice window decorations.
    JFrame.setDefaultLookAndFeelDecorated(true);

    //Create and set up the window.
    JFrame frame = new JFrame("SwingApplication");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    SwingApplication app = new SwingApplication();
    Component contents = app.createComponents();
    frame.getContentPane().add(contents, BorderLayout.CENTER);

    //Display the window.
    frame.pack();
    frame.setVisible(true);
}
```

Setting Up Buttons and Labels

```
    public static void main(String[] args) {
        //Schedule a job for the event-dispatching thread:
        //creating and showing this application's GUI.
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                createAndShowGUI();
            }
        });
    }
}
```



Setting Up Buttons and Labels

- Here's the code that initializes the button:

```
JButton button = new JButton("I'm a Swing button!");
button.setMnemonic('i');
button.addActionListener(/*...create an action listener...*/);
```

  - The first line creates the button. The second sets the letter "i" as the mnemonic that the user can use to simulate a click of the button. The third line registers an event handler for the button click
- Here's the code that initializes and manipulates the label:

```
...// where instance variables are declared:
private static String labelPrefix = "Number of button clicks: ";
private int numClicks = 0;
...// in GUI initialization code:
final JLabel label = new JLabel(labelPrefix + "0 ");
... label.setLabelFor(button);
...// in the event handler for button clicks:
label.setText(labelPrefix + numClicks);
```

  - The line that invokes the setLabelFor method exists solely as a hint to assistive technologies, such as screen readers, that the label describes the button.

Adding Components to Containers

- The previous example groups its label and button in a ***container (a JPanel)*** before adding them to the frame.
  - Here's the code that initializes the container:

    ```
    JPanel panel = new JPanel(new GridLayout(0,1));
    panel.add(button);
    panel.add(label);
    panel.setBorder(BorderFactory.createEmptyBorder(...));
    ```

    - ✓ The first line creates the container and assigns it a ***layout manager***
      - Determines the size and position of each component added to the container.
        new GridLayout(0,1) creates a layout manager that forces the container's contents to be displayed in a single column, with every component having the same size.
    - ✓ The next two lines add the button and the label to the container.
    - ✓ The last line adds a border to it.

Adding Borders Around Components

- To add a border to the panel:

```
pane.setBorder(BorderFactory.createEmptyBorder(
              30, //top
              30, //left
              10, //bottom
              30) //right
);
```

- The code creates and sets a border that provides some empty space around the container's contents
  - 30 extra pixels on the top, left, and right and 10 extra pixels on the bottom.
  - Borders are a feature that JPanel inherits from the *JComponent* class.
  - A Border object isn't a *JComponent*;
    - ✓ it's used by one or more JComponents to paint the component's edges.

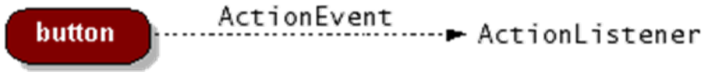
Handling Events

- Every time the user types a character or pushes a mouse button, an event occurs.
  - Any object can be notified of the event.
  - Just implement the appropriate interface and be registered as an *event listener* on the appropriate *event source*.
  - Our example (SwingApplication class) implements an event handler for button clicks (action events).
- Here's the relevant code:

```
public class SwingApplication implements ActionListener {
    ...
    JButton button = new JButton("I'm a Swing button!");
    button.addActionListener(this);
    ....
    public void actionPerformed(ActionEvent e) {
          numClicks++;
          label.setText(labelPrefix + numClicks);
    }
}
```

Handling Events

- Every event handler requires three pieces of code:
  - In the declaration for the event handler class, one line of code specifies that the class either implements a listener interface or extends a class that implements a listener interface.
    - ✓ For example: public class MyClass implements ActionListener {
  - Another line of code registers an instance of the event handler class as a listener on one or more components.
    - ✓ For example: someComponent.addActionListener(instanceOfMyClass);
  - The event handler class has code that implements the methods in the listener interface.
    - ✓ For example: *public void actionPerformed(ActionEvent e)*{ ***//code that reacts to the action...*** }
- To detect when the user clicks an onscreen button (or does the keyboard equivalent), we must have an object that implements the ActionListener interface.
  - Register this object as an action listener on the button, using *addActionListener* method.
  - When the user clicks the button, it **fires an action event.**
  - This results in the invocation of the action listener's actionPerformed method
    - ✓ The single argument to the method is an *ActionEvent object* that gives information about the event and its source.

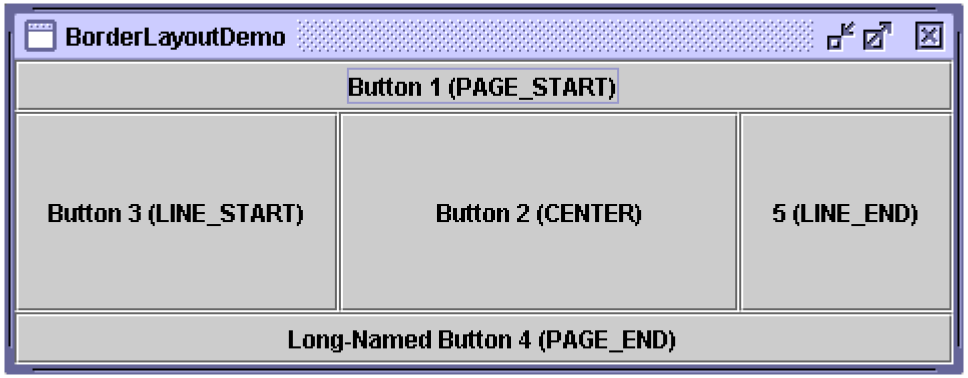


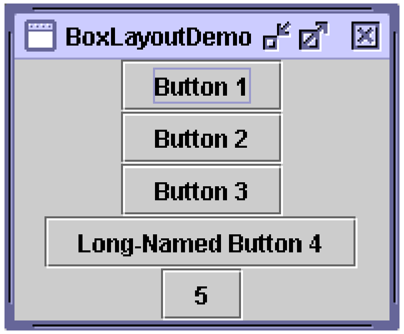
Handling Events

Some Events and Their Associated Event Listeners

Act that Results in the Event	**Listener Type**
User clicks a button, presses Enter while typing in a text field, or chooses a menu item	ActionListener
User closes a frame (main window)	WindowListener
User presses a mouse button while the cursor is over a component	MouseListener
User moves the mouse over a component	MouseMotionListener
Component becomes visible	ComponentListener
Component gets the keyboard focus	FocusListener
Table or list selection changes	ListSelectionListener
Any property in a component changes such as the text on a label	PropertyChangeListener

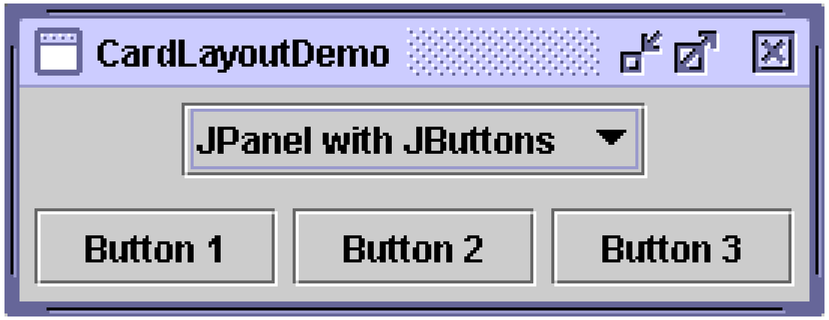
A Visual Guide to Layout Managers

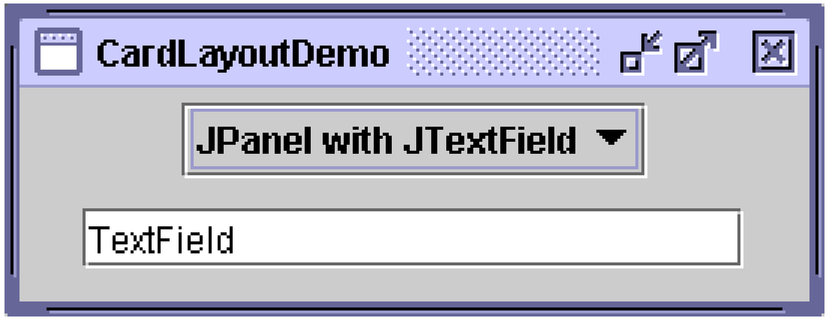




- **BorderLayout**
  - Every content pane (the content pane is the main container in all frames, applets, and dialogs.) is initialized to use a BorderLayout.
  - A BorderLayout places components in up to five areas: top, bottom, left, right, and center. All extra space is placed in the center area.
- **BoxLayout**
  - The BoxLayout class puts components in a single row or column.
  - It respects the components' requested maximum sizes and also lets you align components.

A Visual Guide to Layout Managers

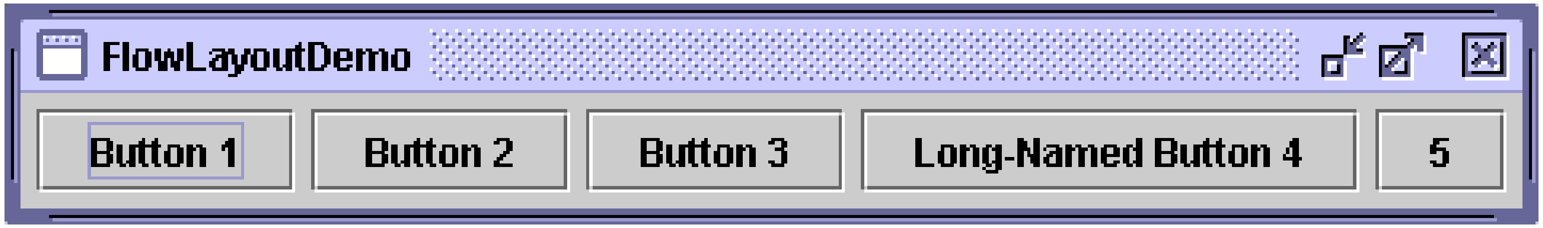




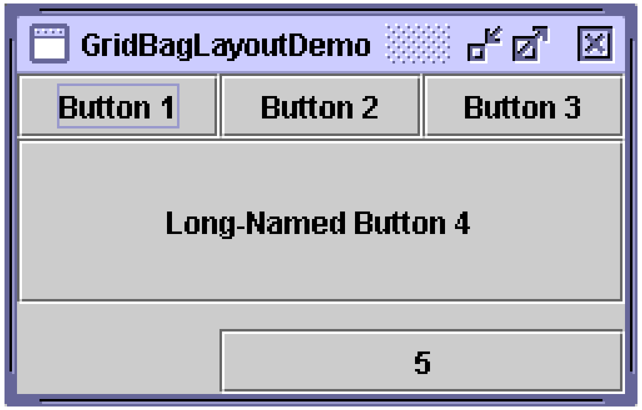
- **CardLayout**
  - Lets you implement an area that contains different components at different times.
  - A CardLayout is often controlled by a combo box, with the state of the combo box determining which panel (group of components) the CardLayout displays.
  - An alternative to using CardLayout is using a tabbed pane , which provides similar functionality but with a pre-defined GUI.

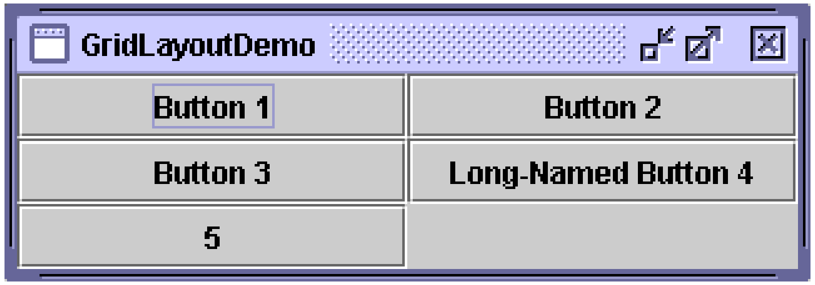
A Visual Guide to Layout Managers

- **FlowLayout**
  - The default layout manager for every JPanel.
  - It simply lays out components in a single row, starting a new row if its container isn't sufficiently wide.
  - Both panels in CardLayoutDemo, shown previously, use FlowLayout.



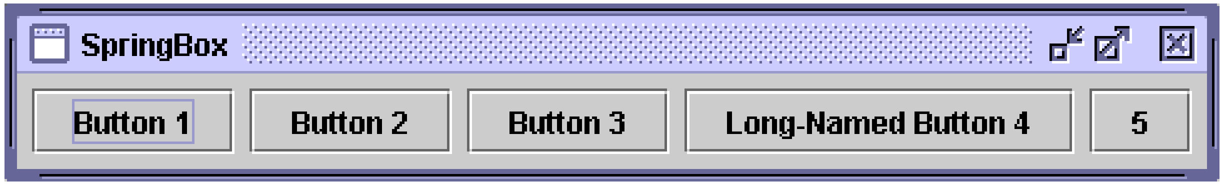
A Visual Guide to Layout Managers

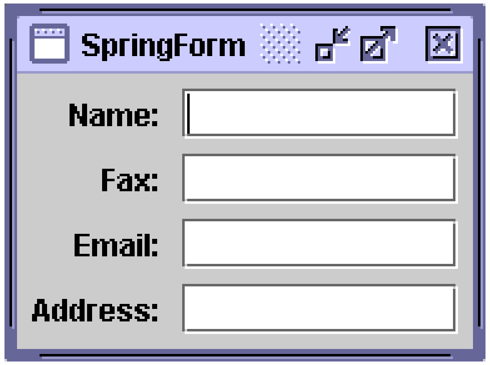




- **GridBagLayout**
  - A sophisticated, flexible layout manager.
  - It aligns components by placing them within a grid of cells, allowing some components to span more than one cell.
  - The rows in the grid can have different heights.
  - Grid columns can have different widths.
- **GridLayout**
  - Simply makes a bunch of components equal in size and displays them in the requested number of rows and columns.

A Visual Guide to Layout Managers





- **SpringLayout**
  - Flexible layout manager designed for use by GUI builders.
  - Lets you specify precise relationships between the edges of components under its control.
    - ✓ E.g. you might define that the left edge of one component is a certain distance (which can be dynamically calculated) from the right edge of a second component.

Using a Layout Manager

- Setting the Layout Manager
  - As a rule, the only containers whose layout managers you need to worry about are JPanels and content panes .
    - ✓ Each JPanel object is **initialized to use a FlowLayout**, unless you specify differently when creating the JPanel.
    - ✓ Content panes **use BorderLayout by default**.
    - ✓ You're free to change the layout manager to a different one.
- You can set a panel's layout manager using the JPanel constructor.
  - `JPanel panel = new JPanel(new BorderLayout());`
- After a container has been created, you can set its layout manager using the **setLayout** method.

  -
    ```
    Container contentPane = frame.getContentPane();
    contentPane.setLayout(new FlowLayout());
    ```
- By setting a container's layout property to null, you make the container use no layout manager.
  - This strategy is called *absolute positioning*
  - You must specify the size and position of every component within that container.
  - Doesn't adjust well when the top-level container is resized.
  - It also doesn't adjust well to differences between users and systems, such as different font sizes and locales .

Using a Layout Manager

- Adding Components to a Container
  - The arguments you specify to the add method depend on the layout manager that the panel or content pane is using.
  - E.g. BorderLayout requires that you specify the area to which the component should be added:
    `pane.add(aComponent, BorderLayout.PAGE_START);`
  - Some layout managers, such as GridBagLayout and SpringLayout, require elaborate setup procedures.
  - Many layout managers, however, simply place components based on the order they were added to their container.
- Swing containers other than JPanel and content panes generally provide API that you should use instead of the add method.
  - E.g., instead of adding a component directly to a scroll pane (or, actually, to its viewport), you either specify the component in the JScrollPane constructor or use setViewportView.
  - Because of specialized API like this, you don't need to know which layout manager (if any) many Swing containers use.
    - ✓ (For the curious: scroll panes happen to use a layout manager named ScrollPaneLayout.)

Using a Layout Manager

- Providing Size and Alignment Hints
  - Sometimes you need to customize the size hints that a component provides to its container's layout manager, so that the component will be laid out well.
  - You can do this by specifying one or more of the minimum, preferred, and maximum sizes of the component.
  - You can invoke the component's methods for setting size hints
    - ✓ **setMinimumSize, setPreferredSize, and setMaximumSize**.
  - Or you can create a subclass of the component that overrides the appropriate getter methods
    - ✓ **getMinimumSize, getPreferredSize, and getMaximumSize**.
  - Here is an example of making a component's maximum size unlimited: `component.setMaximumSize(new Dimension(Integer.MAX_VALUE, Integer.MAX_VALUE));`
- Many layout managers don't pay attention to a component's requested maximum size (except BoxLayout and SpringLayout).
- You can also provide alignment hints.
  - E.g., you can specify that the top edges of two components should be aligned.
  - You set alignment hints either by invoking the component's **setAlignmentX** and **setAlignmentY** methods, or by overriding the component's **getAlignmentX** and **getAlignmentY** methods.
  - Most layout managers ignore alignment hints (except BoxLayout).

Using a Layout Manager

- Putting Space Between Components
  - Three factors influence the amount of space between visible components in a container:
  - The layout manager
    - ✓ Some layout managers automatically put space between components; others don't.
    - ✓ Some let you specify the amount of space between components.
  - Invisible components
    - ✓ You can create **lightweight components that perform no painting**, but that can take up space in the GUI.
    - ✓ Often, you use invisible components in containers controlled by BoxLayout.
  - Empty borders
    - ✓ No matter what the layout manager, you can affect the apparent amount of space between components by adding empty borders to components.
    - ✓ The best candidates for empty borders are components that typically have no default border, ***such as panels and labels***.
    - ✓ Some other components **might not work well with borders in some look-and-feel** implementations, because of the way their painting code is implemented.

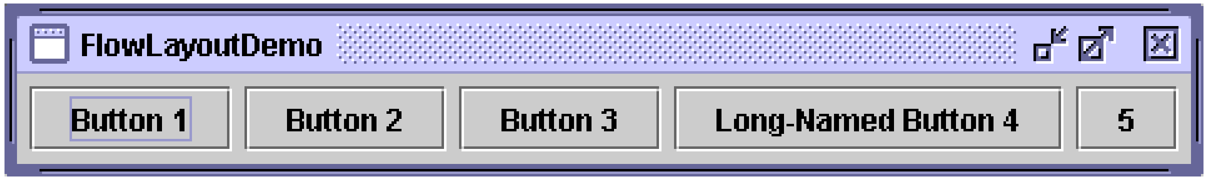
Using a Layout Manager

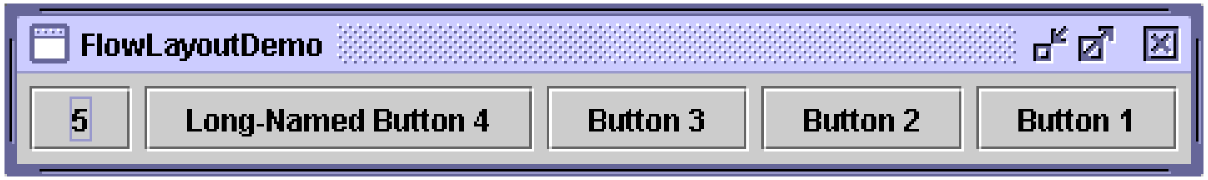
- Setting the Container's Orientation
  - This presentation is written in English, with text that runs from left to right, and then top to bottom.
  - However, many other languages have different orientations.
  - The *componentOrientation* property provides a way of indicating that a particular component should use something different from the default left-to-right, top-to-bottom orientation.
  - In a component such as a radio button, the orientation might be used as a hint that the look and feel should switch the locations of the icon and text in the button.
  - In a container, the orientation is used as a hint to the layout manager.
- To set a container's orientation, you can use either the Component-defined method **setComponentOrientation** or, to set the orientation on the container's children as well, applyComponentOrientation .
  - The argument to either method can be a ***constant such as*** *ComponentOrientation.RIGHT_TO_LEFT*,
  - or it can be a call to the ***ComponentOrientation method getOrientation(Locale)*** .
- Default orientation (left-to-right)
  - Right-to-left orientation is supported by FlowLayout, BorderLayout, BoxLayout, GridBagLayout, and GridLayout.

Using a Layout Manager

- E.g., the following code causes all JComponents to be initialized with an Arabic-language locale, and then sets the orientation of the content pane and all components inside it accordingly:

```
    JComponent.setDefaultLocale(new Locale("ar"));
    JFrame frame = new JFrame();
    ...
    Container contentPane = frame.getContentPane();
    contentPane.applyComponentOrientation(
          ComponentOrientation.getOrientation(
                 contentPane.getLocale()));
```





Tips on Choosing a Layout Manager

- Layout managers have different strengths and weaknesses. Keep in mind that flexible layout managers such as **GridBagLayout and SpringLayout can fulfill many layout needs.**
- **Scenario:** You need to display a component in as much space as it can get.
  - If it's the only component in its container, use GridLayout or BorderLayout.
  - Otherwise, BorderLayout or GridBagLayout might be a good match.
    - ✓ If you use BorderLayout, you'll need to put the space-hungry component in the center.
    - ✓ With GridBagLayout, you'll need to set the constraints for the component so that *fill=GridBagConstraints.BOTH*.
  - Another possibility is to use BoxLayout, making the space-hungry component specify very large preferred and maximum sizes.
- **Scenario:** You need to display a few components in a compact row at their natural size.
  - **Consider using a JPanel to group the components** and using either the JPanel's default FlowLayout manager or the BoxLayout manager. SpringLayout is also good for this.

Tips on Choosing a Layout Manager

- **Scenario:** You need to display a few components of the same size in rows and columns.
  - GridLayout is perfect for this.
- **Scenario:** You need to display a few components in a row or column, possibly with varying amounts of space between them, custom alignment, or custom component sizes.
  - BoxLayout is perfect for this.
- **Scenario:** You need to display aligned columns, as in a form-like interface where a column of labels is used to describe text fields in an adjacent column.
  - SpringLayout is a natural choice for this.
- **Scenario:** You have a complex layout with many components.
  - Consider using a very flexible layout manager such as GridBagLayout or SpringLayout
  - Alternatively, consider grouping the components into one or more JPanels to simplify layout.
    - ✓ Each JPanel might use a different layout manager.

Text Fields

- A text field is a basic text control that lets the user enter a small amount of text.
  - When the user indicates that text entry is complete (usually by pressing Enter), the text field fires an action event.
  - If you need to obtain more than one line of input from the user, you should use a text area instead.
- The Swing API provides several classes for components that are either varieties of text fields or that include text fields.
  - JTextField
    - ✓ basic text fields. Example:

      ```
      textField = new JTextField(20);
      textField.addActionListener(this);
      ```
  - JFormattedTextField
    - ✓ A JTextField subclass that allows you to specify the legal set of characters the user can enter.
  - JPasswordField
    - ✓ A JTextField subclass that doesn't show the characters the user types.
  - JComboBox
    - ✓ Can be editable, and provides a menu of strings to choose from.
  - JSpinner
    - ✓ Combines a formatted text field with a couple of small buttons that let the user choose the previous or next available value.