

# 20. Containers, Layout Managers, and Borders



# How a Component is Displayed?

User interface components like JButton cannot be displayed without being placed in a container.

A container is a component that is capable of containing other components. You do not display a user interface component; you place it in a container, and the container displays the components it contains.



# What Does a Container Do?

The base class for all containers is `java.awt.Container`, which is a subclass of `java.awt.Component`. The `Container` class has the following essential functions:

- ➡ It adds and removes components using various add and remove methods.
- ➡ It maintains a layout property for specifying a layout manager that is used to lay out components in the container. Every container has a default layout manager.
- ➡ It provides registration methods for the `java.awt.event.ContainerEvent`.



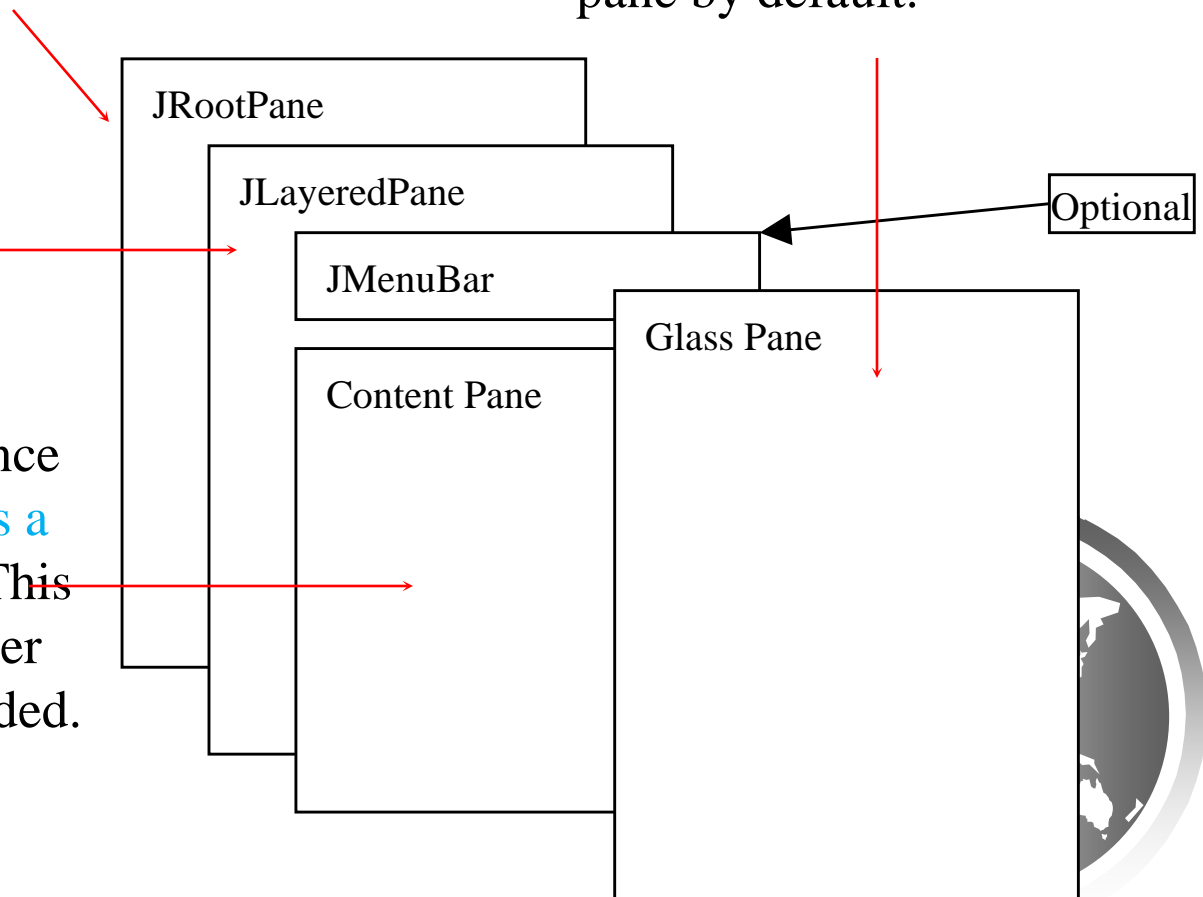
# Structures of the Swing Containers

A container used behind the scenes by Swing's **top-level containers**, such as JFrame, JApplet, and JDialog

A container that **manages the optional menu bar and the content pane**

The content pane is an instance of Container. **By default, it is a JPanel with BorderLayout**. This is the container where the user interface components are added.

The glass pane floats on top of everything. It is a hidden pane by default.



# JFrame

JFrame, a Swing version of Frame, is a top-level container for Java graphics applications. Like Frame, JFrame is displayed as a standalone window with a title bar and a border. The following properties are often useful in JFrame.

- contentPane
- iconImage
- jMenuBar
- layout
- title
- resizable



# JApplet

JApplet is a Swing version of Applet. Since it is a subclass of Applet, it has all the functions required by the Web browser. Here are the four essential methods defined in Applet:

- getContentPane
- JMenuBar
- layout



# JPanel

Panels act as sub-containers for grouping user interface components. javax.swing.JPanel is different from JFrame and JApplet.

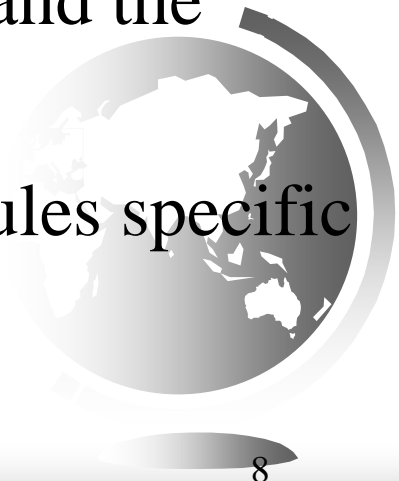
First, JPanel is **not a top-level container**; it must be placed inside another container, and it can be placed inside another JPanel.

Second, since JPanel is a subclass of JComponent, it is a **lightweight** component, but JFrame and JApplet are heavyweight components.



# About Layout Managers

- ➡ Each container has a layout manager, which is responsible for arranging the components in a container.
- ➡ The container's `setLayout` method can be used to set a layout manager.
- ➡ Certain types of containers have default layout managers.
- ➡ The layout manager places the components according to the layout manager's rules, property settings and the constraints associated with each component.
- ➡ Each layout manager has a particular set of rules specific to that layout manager.





# The Size of Components in a Container

The size of a component in a container is determined by many factors, such as:

- ➡ The type of layout manager used by the container.
- ➡ The layout constraints associated with each component
- ➡ The size of the container.
- ➡ Certain properties common to all components (such as preferredSize, minimumSize, maximumSize, alignmentX, and alignmentY).

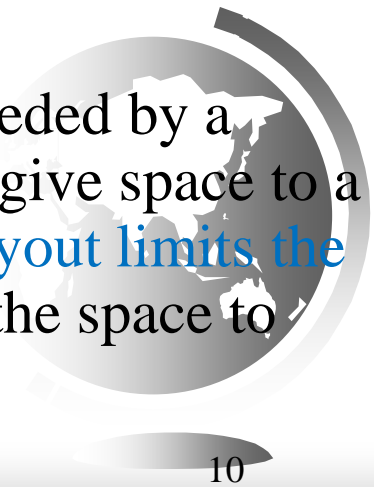


# preferredSize, minimumSize, and maximumSize

The **preferredSize** property indicates the ideal size at which the component looks best. Depending on the rules of the particular layout manager, this property may or may not be considered. For example, the preferred size of a component is used in a container with a `FlowLayout` manager, but ignored if it is placed in a container with a `GridLayout` manager.

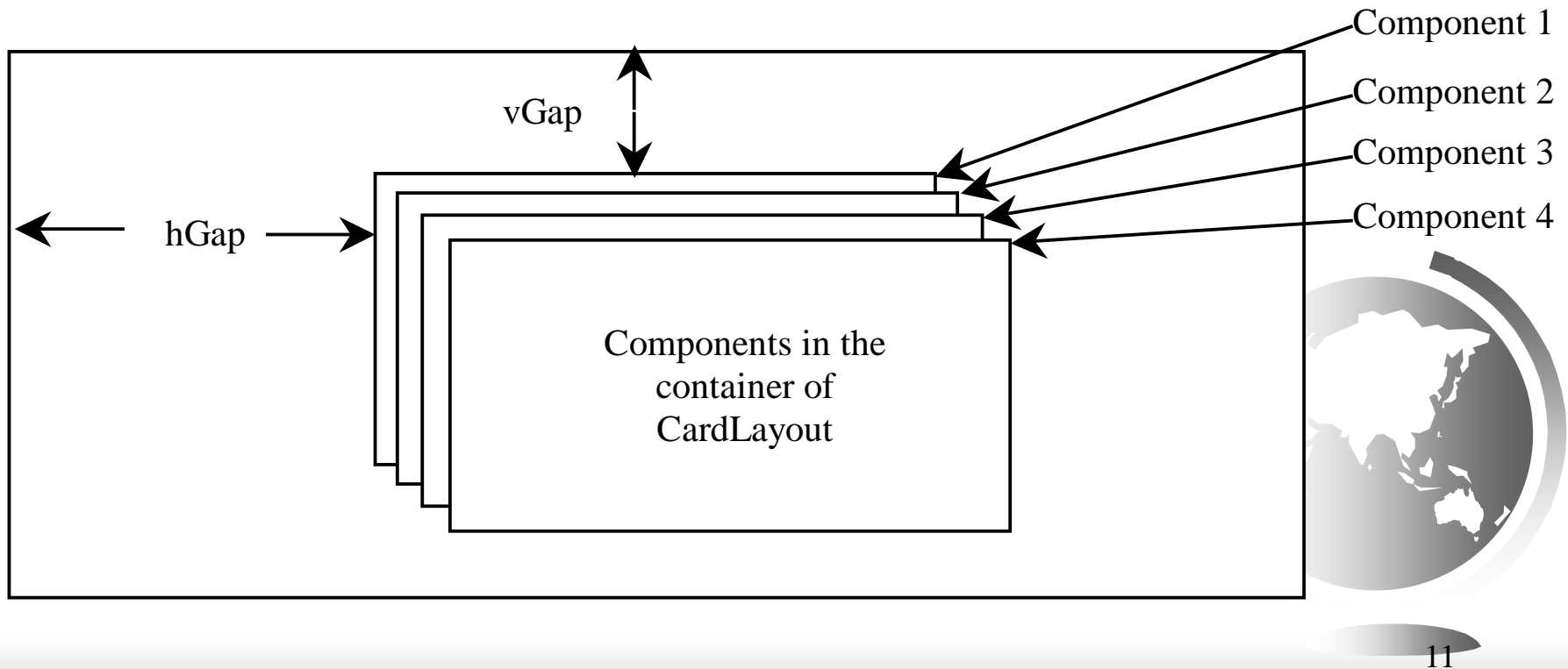
The **minimumSize** property specifies the minimum size at which the component is useful. For most GUI components, `minimumSize` is the same as `preferredSize`. Layout managers generally respect `minimumSize` more than `preferredSize`.

The **maximumSize** property specifies the maximum size needed by a component, so that the layout manager won't wastefully give space to a component that does not need it. For instance, `BorderLayout` limits the center component's size to its maximum size, and gives the space to edge components.

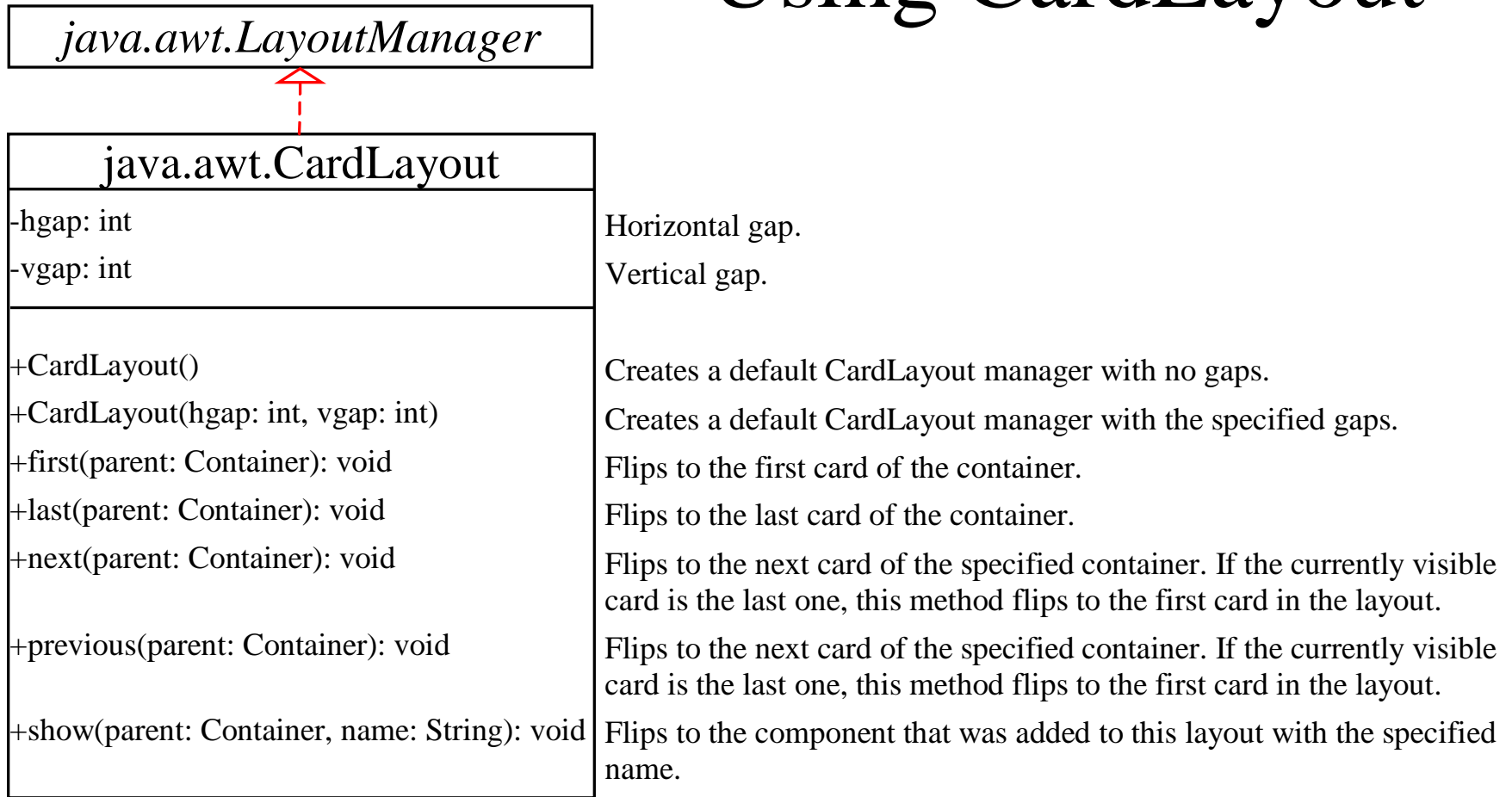


# CardLayout

CardLayout places components in the container as cards. Only one card is visible at a time, and the container acts as a stack of cards. The ordering of cards is determined by the container's own internal ordering of its component objects. CardLayout defines a set of methods that allow an application to flip through the cards sequentially or to show a specified card directly.



# Using CardLayout



To add a component into a container, use the **add(Component c, String name)** method defined in the LayoutManager interface. **The String parameter, name, gives an explicit identity to the component in the container.**

# Example: Using CardLayout

Objective: Create two panels in a frame. The first panel holds named components. The second panel uses buttons and a choice box to control which component is shown.

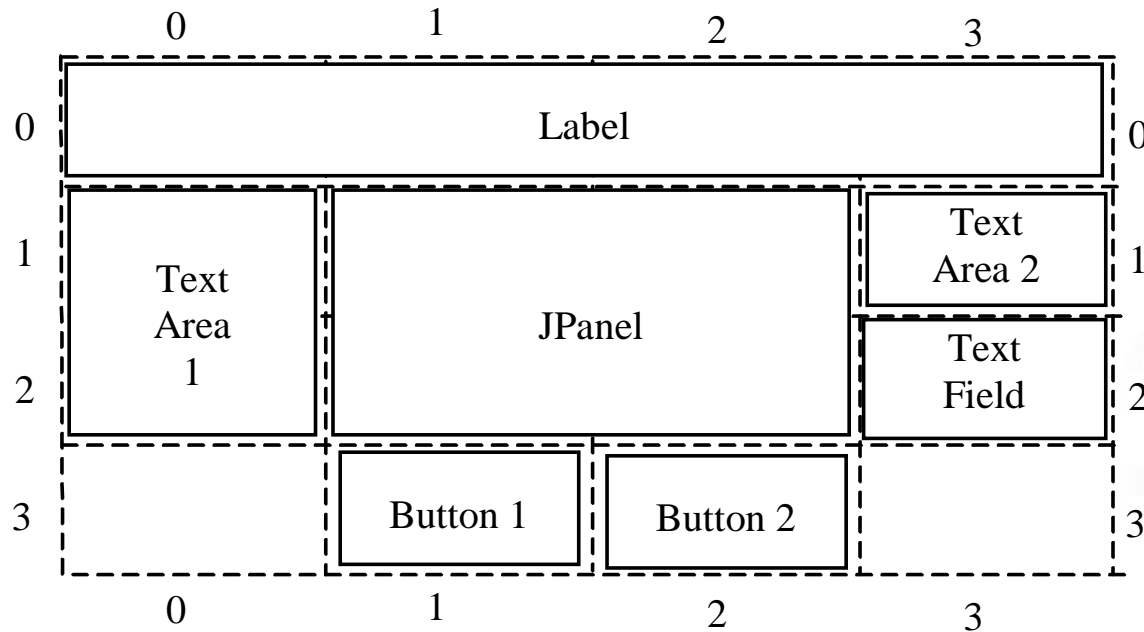


ShowCardLayout

Run

# GridBagLayout

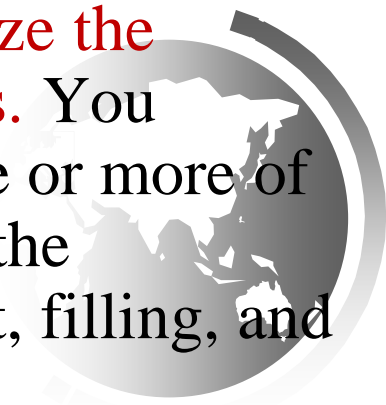
The GridBagLayout manager is the most flexible and the most complex. It is similar to the GridLayout manager in the sense that both layout managers **arrange components in a grid**. The components can vary in size, however, and can be added in any order in GridBagLayout.



# GridBagConstraints

Each GridBagLayout uses a dynamic rectangular grid of cells, with each component occupying one or more cells called its display area. Each component managed by a GridBagLayout is associated with a GridBagConstraints instance that specifies how the component is laid out within its display area. How a GridBagLayout places a set of components depends on the GridBagConstraints and minimum size of each component, as well as the preferred size of the component's container.

To use GridBagLayout effectively, you must customize the GridBagConstraints of one or more of its components. You customize a GridBagConstraints object by setting one or more of its public instance variables. These variables specify the component location, size, growth factor, anchor, inset, filling, and padding.



# GridBagConstraints Parameters

**Location** parameters: `gridx` and `gridy`

The variables `gridx` and `gridy` specify the cell at the upper left of the component's display area, where the upper-leftmost cell has the address `gridx=0, gridy=0`. Note that `gridx` specifies the column in which the component will be placed, and `gridy` specifies the row in which it will be placed. In Figure 33.5, Button 1 has a `gridx` value of 1 and a `gridy` value of 3, and Label has a `gridx` value of 0 and a `gridy` value of 0.





# GridBagConstraints Parameters, cont.

**Size** parameters: gridwidth and gridheight

The variables gridwidth and gridheight specify the number of cells in a row (for gridheight) or column (for gridwidth) in the component's display area. The default value is 1. In Figure 33.5, the JPanel in the center occupies two columns and two rows, so its gridwidth is 2, and its gridheight is 2. Text Area 2 occupies one row and one column; therefore its gridwidth is 1, and its gridheight is 1.



# GridBagConstraints Parameters, cont.

**Growth** parameters: weightx and weighty

The variables weightx and weighty specify the extra horizontal and vertical space to allocate for the component when the resulting layout is smaller horizontally than the area it needs to fill.

The GridBagLayout manager calculates the weight of a column to be the maximum weightx (weighty) of all the components in a column (row).

The extra space is distributed to each column (row) in **proportion** to its weight.



# GridBagConstraints Parameters, cont.

**Anchor** parameter:

The variable anchor specifies where in the area the component is placed **when it does not fill the entire area**. Valid values are:

GridBagConstraints.CENTER (the default)

GridBagConstraints.NORTH

GridBagConstraints.NORTHEAST

GridBagConstraints.EAST

GridBagConstraints.SOUTHEAST

GridBagConstraints.SOUTH

GridBagConstraints.SOUTHWEST

GridBagConstraints.WEST

GridBagConstraints.NORTHWEST



# GridBagConstraints Parameters, cont.

**Fill** parameter:

Used when the component's display area **is larger than the component's requested size** to determine whether and how to resize the component.

Valid values (defined as GridBagConstraints constants) include NONE (the default),

HORIZONTAL (make the component wide enough to fill its display area horizontally, but do not change its height),

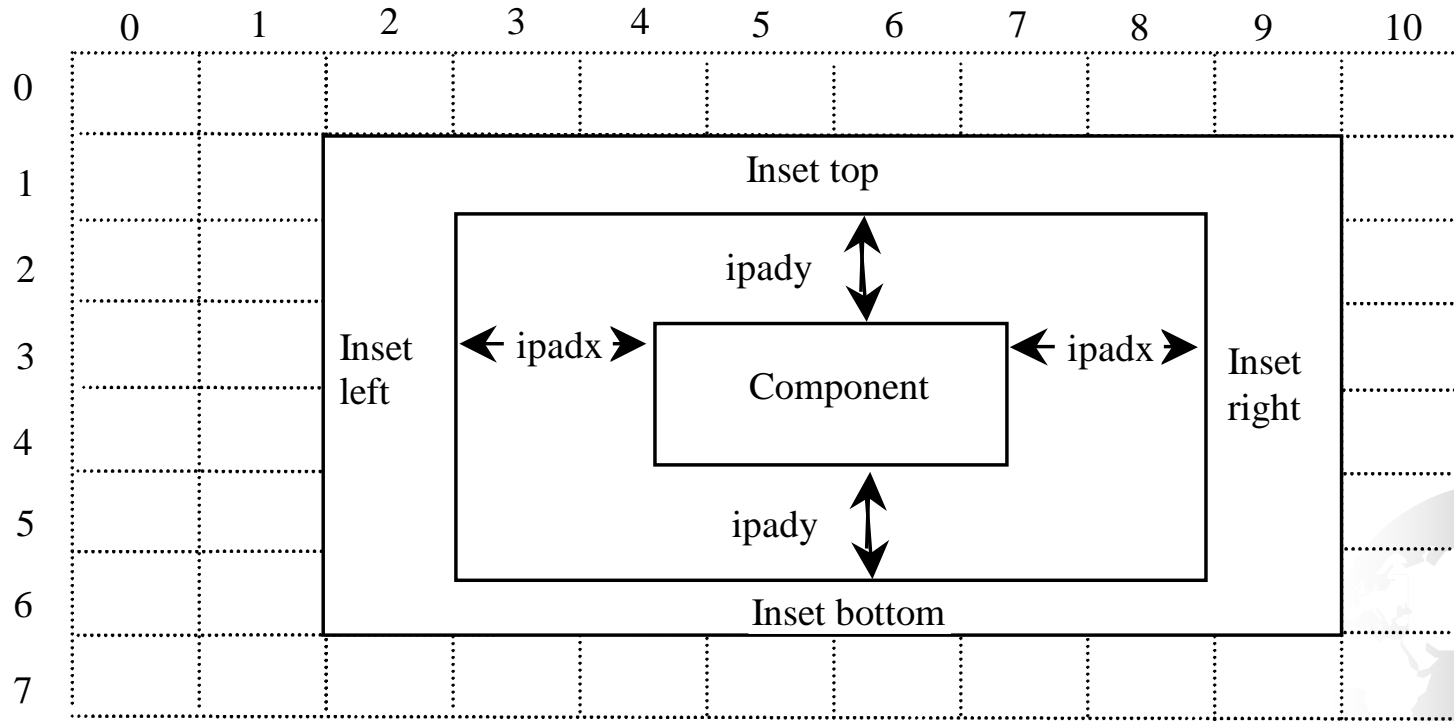
VERTICAL (make the component tall enough to fill its display area vertically, but do not change its width),

and BOTH (make the component fill its display area entirely).



# GridBagConstraints Parameters, cont.

Inset and padding parameters:



☞ GridBagConstraints(int gridx,int gridy,  
int gridwidth,int gridheight,  
double weightx,double weighty,  
int anchor,int fill, Insets insets,  
int ipadx,int ipady)建立一个新的GridBagConstraints对象，并指定其参数的值

。

参数说明：

**gridx,gridy** —— 设置组件的位置，gridx设置为GridBagConstraints.RELATIVE代表此组件位于之前所加入组件的右边。gridy设置为GridBagConstraints.RELATIVE代表此组件位于以前所加入组件的下面。

**gridwidth,gridheight** —— 用来设置组件所占的单位长度与高度，默认值皆为1。  
你可以使用GridBagConstraints.REMAINDER常量，代表此组件为此行或此列的最后一个组件，而且会占据所有剩余的空间。

**weightx,weighty** —— 用来设置窗口变大时，各组件跟着变大的比例。  
当数字越大，表示组件能得到更多的空间，默认值皆为0。

**anchor** —— 当组件空间大于组件本身时，要将组件置于何处。  
有CENTER(默认值)、NORTH、NORTHEAST、EAST、SOUTHEAST、WEST、NORTHWEST选择。

**fill** ----- 当组件空间大于组件本身时，要将组件如何填充。

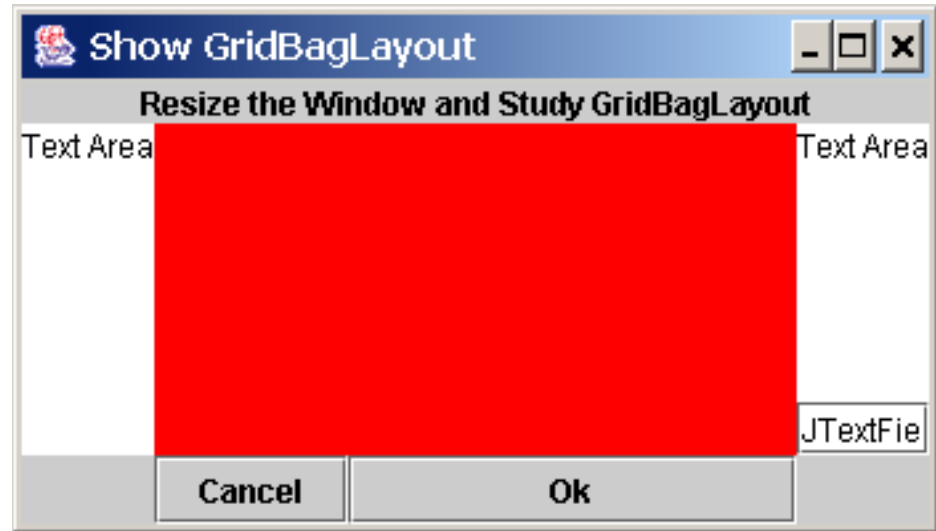
**insets** —— 设置组件之间彼此的间距。  
它有四个参数，分别是上，左，下，右，默认为(0,0,0,0)。

**ipadx,ipady** —— 设置组件间距，默认值为0。



# Example: Using GridBagLayout Manager

Objective: Write a program that uses the GridBagLayout manager to create a layout.



ShowGridBagLayout

Run

# Using No Layout Manager

You can place components in a container without using any layout manager. In this case, you **need to** set layout for the container using

```
container.setLayout(null);
```

The components must be placed using the component's instance method `setBounds ( )`.

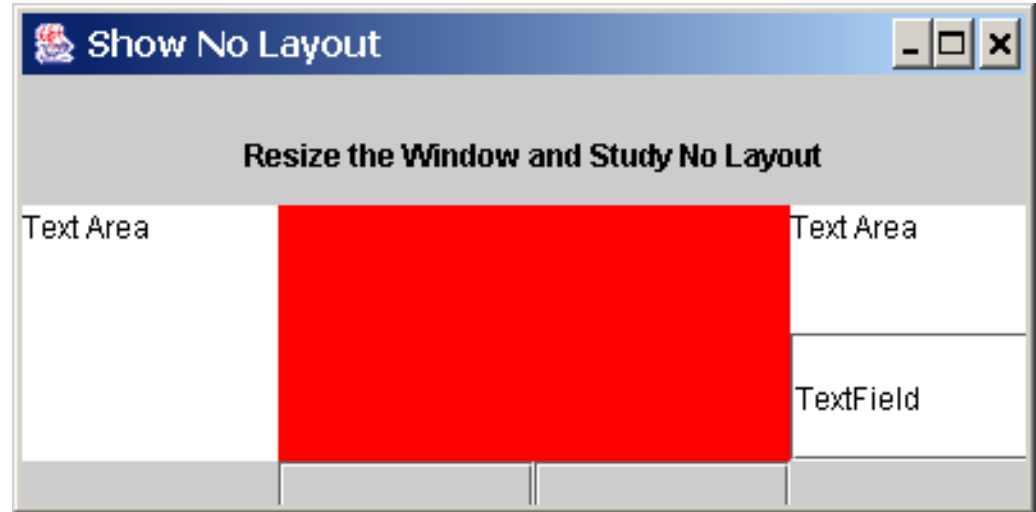
Tip: Do not use the no-layout-manager option to develop platform-independent applications.





# Example: Using No Layout Manager

This example shows a program that places the same components in the same layout as in the preceding example, but without using a layout manager.



ShowNoLayout

Run

# BoxLayout

Flow layout arranges components **in rows**.

`javax.swing.BoxLayout` is a Swing layout manager that arranges components **in a row or a column**. To create a `BoxLayout`, use the following constructor:

```
public BoxLayout(Container target, int axis)
```

This constructor is **different from other layout constructors**. The constructor creates a layout manager that is dedicated to the given target container. The axis parameter is `BoxLayout.X_AXIS` or `BoxLayout.Y_AXIS`, which specifies whether the components are laid out horizontally or vertically.



# Creating a BorderLayout

For example the following code creates a horizontal BorderLayout for panel p1:

```
JPanel p1 = new JPanel();  
BoxLayout boxLayout = new BorderLayout(p1,  
    BorderLayout.X_AXIS);  
p1.setLayout(boxLayout);
```

You still need to invoke the `setLayout` method on p1 to set the layout manager.



# The Box Class

You can use BorderLayout in any container, but **it is simpler to use the Box class, which is a container of BorderLayout**. To create a Box container, use one of the following two static methods:

```
Box box1 = Box.createHorizontalBox();
```

```
Box box2 = Box.createVerticalBox();
```

The former creates a box that contains components horizontally, and the latter creates a box that contains components vertically.

You can add components to a box in the same way that you add them to the containers of FlowLayout or GridLayout using the add method, as follows:

```
box1.add(new JButton("A Button"));
```



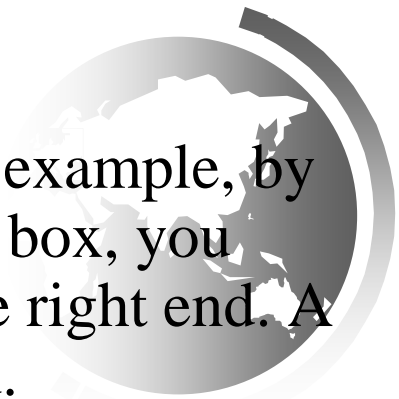
# Fillers in BoxLayout

*A strut* simply adds some space between components. The static method `createHorizontalStrut(int)` in the `Box` class is used to create a horizontal strut, and the static method `createVerticalStrut(int)` to create a vertical strut.

*A rigid area* is a two-dimensional space that can be created using the static method `createRigidArea(dimension)` in the `Box` class. For example, the following code adds a rigid area 10 pixels wide and 20 pixels high into a box.

```
box2.add(Box.createRigidArea(new Dimension(10, 20));
```

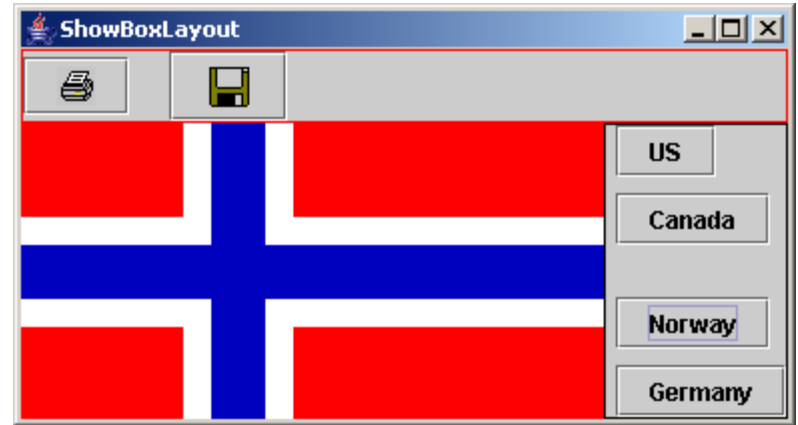
*A glue* separates components as much as possible. For example, by adding a glue between two components in a horizontal box, you place one component at the left end and the other at the right end. A glue can be created using the `Box.createGlue()` method.



# Example: Using BoxLayout Manager

Problem: Write a program that creates a horizontal box and a vertical box. The horizontal box holds two buttons with print and save icons. The horizontal box holds four buttons for selecting flags.

When a button in the vertical box is clicked, a corresponding flag icon is displayed in the label centered in the applet.



ShowBoxLayout

Run

# OverlayLayout

OverlayLayout is a Swing layout manager that arranges components on top of each other. To create an OverlayLayout, use the following constructor:

```
public OverlayLayout(Container target)
```

The constructor creates a layout manager that is **dedicated** to the given target container. For example, the following code creates an OverlayLayout for panel p1:

```
JPanel p1 = new JPanel();  
OverlayLayout overlayLayout = new OverlayLayout(p1);  
p1.setLayout(overlayLayout);
```

You **still need to invoke the setLayout method** on p1 to set the layout manager.



# Order of Components in OverlayLayout Containers

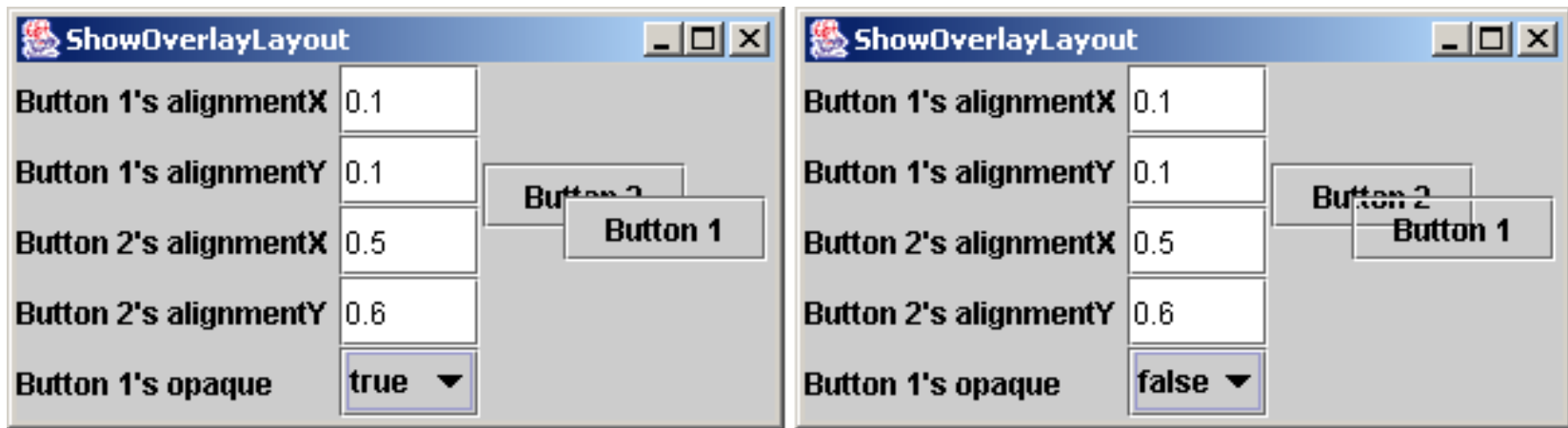
A component is on top of the other if it is added to the container before the other one. Suppose components p1, p2 and p3 are added to a container of the OverlayLayout in this order, then p1 is on top of p2 and p2 is on top of p3.





# Example: Using OverlayLayout Manager

Problem: Write a program that overlays two buttons in a panel of OverlayLayout.



ShowOverlayLayout

Run

# SpringLayout

SpringLayout is a new Swing layout manager introduced in JDK 1.4. **The idea of SpringLayout is to put a flexible spring around a component.** The spring may compress or expand to place the components in desired locations.

To create a SpringLayout, use its no-arg constructor:

```
public SpringLayout()
```



# The Spring Class

A spring is an instance of the **Spring class**, which can be created using one of the following two static methods:

- `public static Spring constant(int pref)`  
Returns a spring whose minimum, preferred, and maximum values each have the value `pref`.
- `public static Spring constant(int min, int pref, int max)`  
Returns a spring with the specified minimum, preferred, and maximum values.



# Manipulating Springs

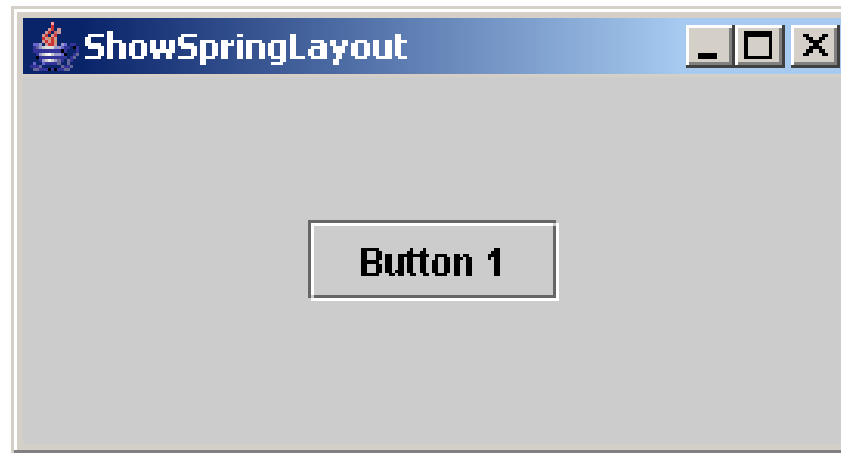
Each spring has a preferred value, minimum value, maximum value, and actual value. The getPreferredValue(), getMinimumValue(), getMaximumValue(), and getValue() methods retrieve these values. The setValue(int value) method can be used to set an actual value.

The Spring class defines the static sum(Spring s1, Spring s2) to **produce a combined new spring**, the static minus(Spring s) to produce a new spring running on the opposite direction, and the static max(Spring s1, Spring s2) to produce a new spring with larger values from s1 and s2.



# Example: Using SpringLayout Manager

Problem: Write a program that places a button in the center of the container.



ShowSpringLayout

Run

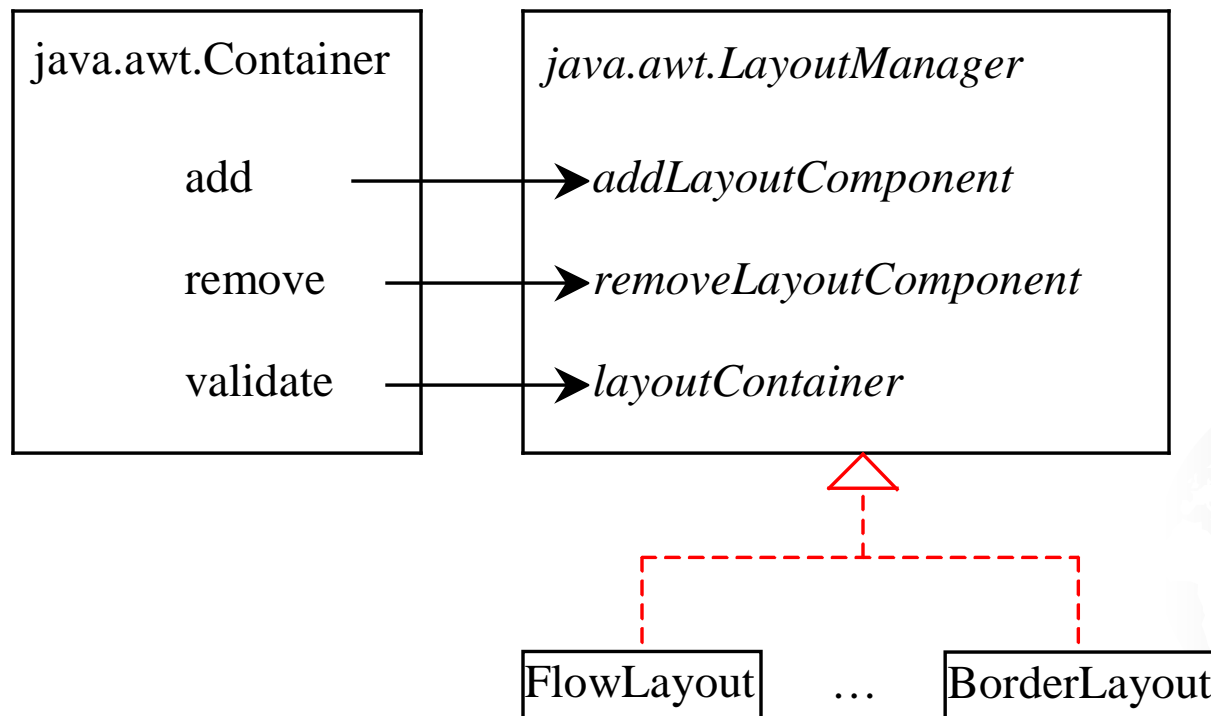
# Creating Custom Layout Managers

In addition to the layout managers provided in Java, you can create your own layout managers. To do so, you need to understand how a layout manager lays out components. **A container's `setLayout` method specifies a layout manager for the container. The layout manager is responsible for laying out the components and displaying them in a desired location with an appropriate size.** Every layout manager must directly or indirectly implement the **`LayoutManager`** interface.

For instance, `FlowLayout` directly implements `LayoutManager`, and `BorderLayout` implements `LayoutManager2`, which implements `LayoutManager`. The `LayoutManager` interface provides the following methods for laying out components in a container:

# How Does a Container Interact with a Layout Manager

The add, remove, and validate methods in Container invoke the methods defined in the LayoutManager interface.



```

public void layoutContainer(Container parent) {
    int numberOfComponents = parent.getComponentCount();

    Insets insets = parent.getInsets();
    int w = parent.getSize().width - insets.left - insets.right;
    int h = parent.getSize().height - insets.bottom - insets.top;

    if (majorDiagonal) {
        int x = 10, y = 10;

        for (int j = 0; j < numberOfComponents; j++) {
            Component c = parent.getComponent(j);
            Dimension d = c.getPreferredSize();

            if (c.isVisible())
                if (lastFill && (j == numberOfComponents - 1))
                    c.setBounds(x, y, w - x, h - y);
                else
                    c.setBounds(x, y, d.width, d.height);
            x += d.height + gap;
            y += d.height + gap;
        }
    }
    else { // It is subdiagonal
        int x = w - 10, y = 10;

        for (int j = 0; j < numberOfComponents; j++) {
            Component c = parent.getComponent(j);
            Dimension d = c.getPreferredSize();

            if (c.isVisible())
                if (lastFill & (j == numberOfComponents - 1))
                    c.setBounds(0, y, x, h - y);
                else
                    c.setBounds(x - d.width, y, d.width, d.height);

            x -= (d.height + gap);
            y += d.height + gap;
        }
    }
}

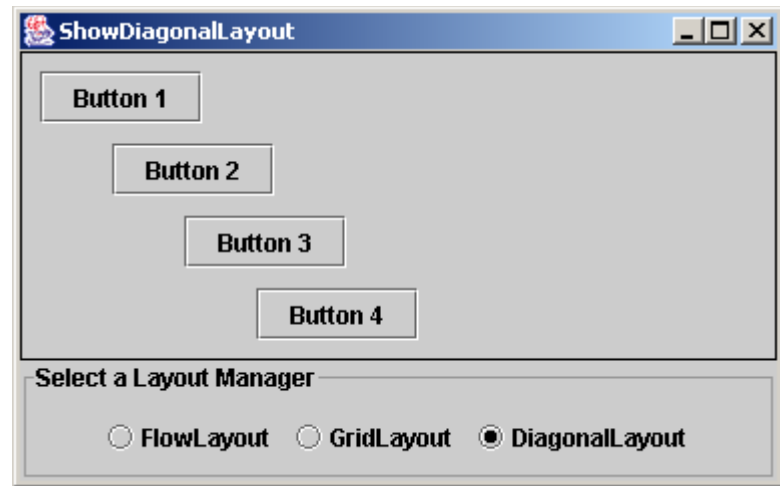
```





# Example: Creating A Custom Layout Manager

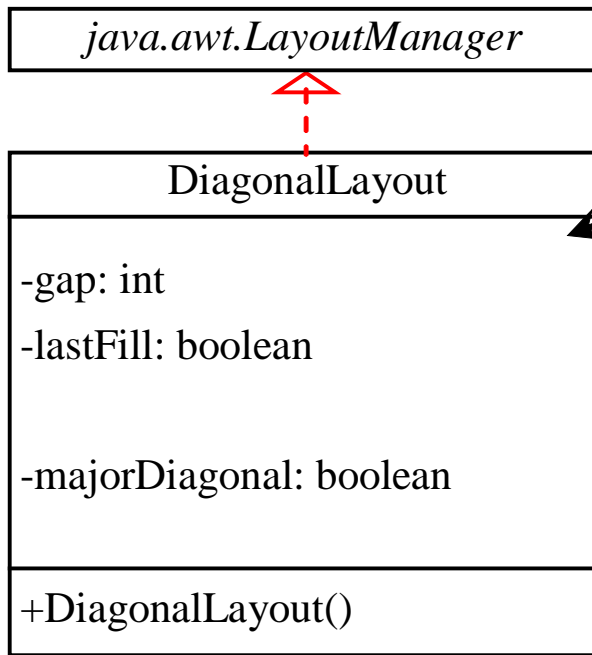
Problem: This example creates a layout manager named DiagonalLayout that places the components in a diagonal. To test DiagonalLayout, the example creates an applet with radio buttons named “FlowLayout,” “GridLayout,” and “DiagonalLayout,” as shown in the figure. You can dynamically select one of these three layouts in the panel.



ShowDiagonalLayout

Run

# Custom DiagonalLayout Manager



JavaBeans properties with get and set methods omitted in the UML diagram.

The gap between the components.

A Boolean value indicating whether the last component in the container is stretched to fill the rest of the space.

A Boolean value indicating whether the components are placed along the major diagonal or the subdiagonal.

Creates a DiagonalLayout.

DiagonalLayout



# JScrollPane

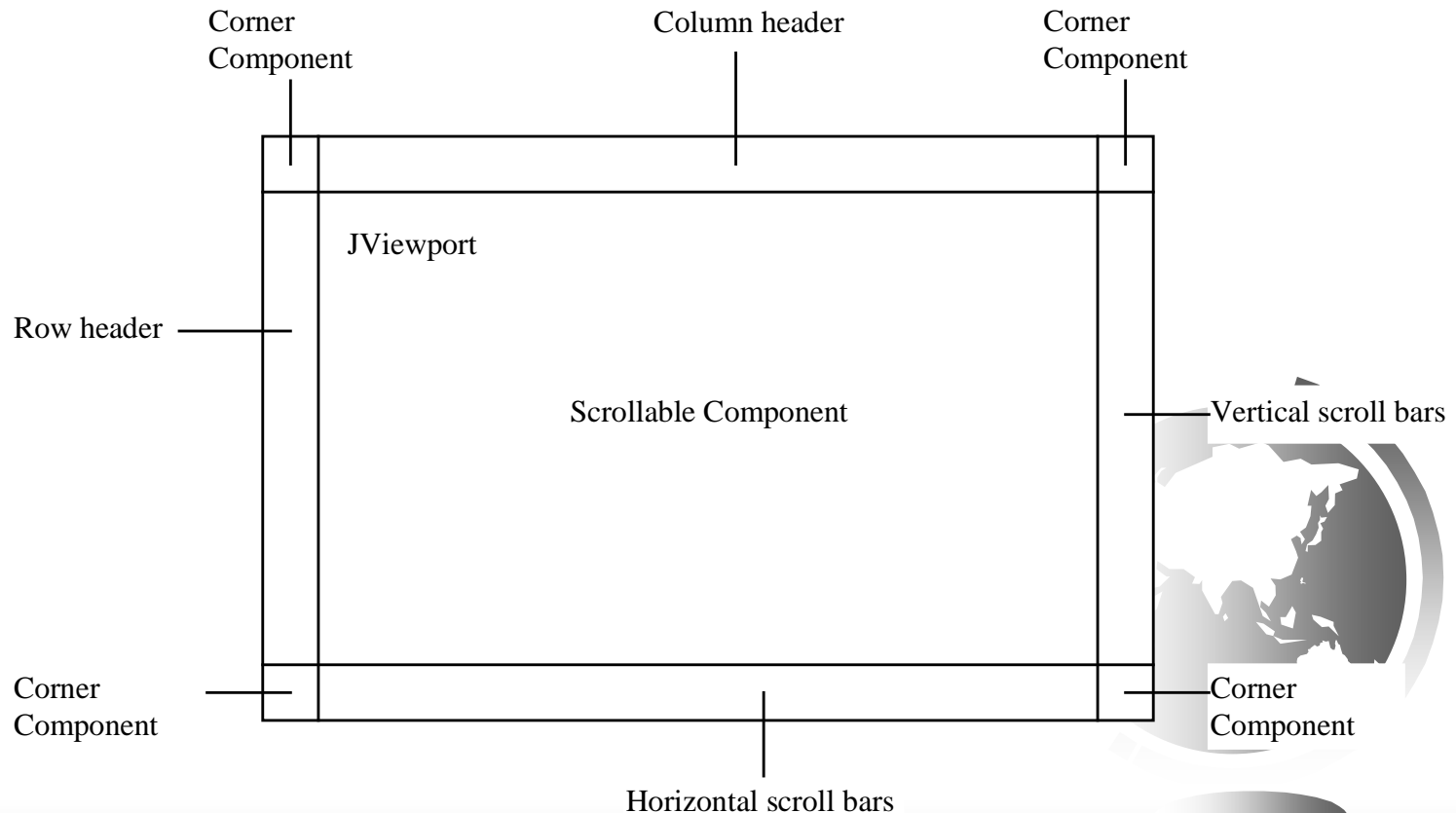
Often you need to use a scrollbar to scroll the contents of an object that does not fit completely into the viewing area. JScrollBar and JSlider can be used for this purpose, but you have to *manually* write the code to implement scrolling with it. JScrollPane is a component that supports *automatic* scrolling without coding.

*A scroll pane* is a component that supports automatically scrolling without coding.



# Scroll Pane Structures

A JScrollPane can be viewed as a specialized container with a view port for displaying the contained component. In addition to horizontal and vertical scrollbars, a JScrollPane can have a column header, a row header, and corners.



# Using JScrollPane

<i>JComponent</i>	
↑	
<i>JScrollPane</i>	
#columnHeader: JViewport	The column header. (default: null)
#rowHeader: JViewport	The row header. (default: null)
#horizontalScrollBarPolicy: int	The display policy for the horizontal scrollbar. (default: <u>JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED</u> )
#verticalScrollBarPolicy: int	The display policy for the horizontal scrollbar. (default: <u>JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED</u> )
#viewport: Jviewport	The scroll pane's viewport.
#horizontalScrollBar: JScrollBar	The scroll pane's horizontal scrollbar.
#verticalScrollBar: JscrollBar	The scroll pane's vertical scrollbar.
-viewportBorder: Border	The border around the viewport.
+JScrollPane()	Creates an empty JScrollPane where both horizontal and vertical scrollbars appear when needed.
+JScrollPane(view: Component)	Creates a JScrollPane that displays the contents of the specified <del>comp</del> component, where both horizontal and vertical scrollbars appear whenever the component's contents are larger than the view.
+JScrollPane(view: Component, vsbPolicy: int, hsbPolicy: int)	Creates a JScrollPane that displays the contents of the specified component with the specified horizontal and vertical scrollbars policies.
+JScrollPane(vsbPolicy: int, hsbPolicy: int)	Creates an empty JScrollPane with the specified horizontal and vertical scrollbars policies.
+setCorner(key: String, corner: Component): void	Adds a component in one of the scroll panes corners.
+setViewportView(view: Component): void	Adds a view component to the viewport.
All the properties have their supporting accessor and mutator methods.	



```

// Create images in labels
private JLabel lblIndianaMap = new JLabel(
    new ImageIcon(getClass().getResource("image/indianaMap.gif")));
private JLabel lblOhioMap = new JLabel(
    new ImageIcon(getClass().getResource("/image/ohioMap.gif")));

// Create a scroll pane to scroll map in the labels
private JScrollPane jspMap = new JScrollPane(lblIndianaMap);

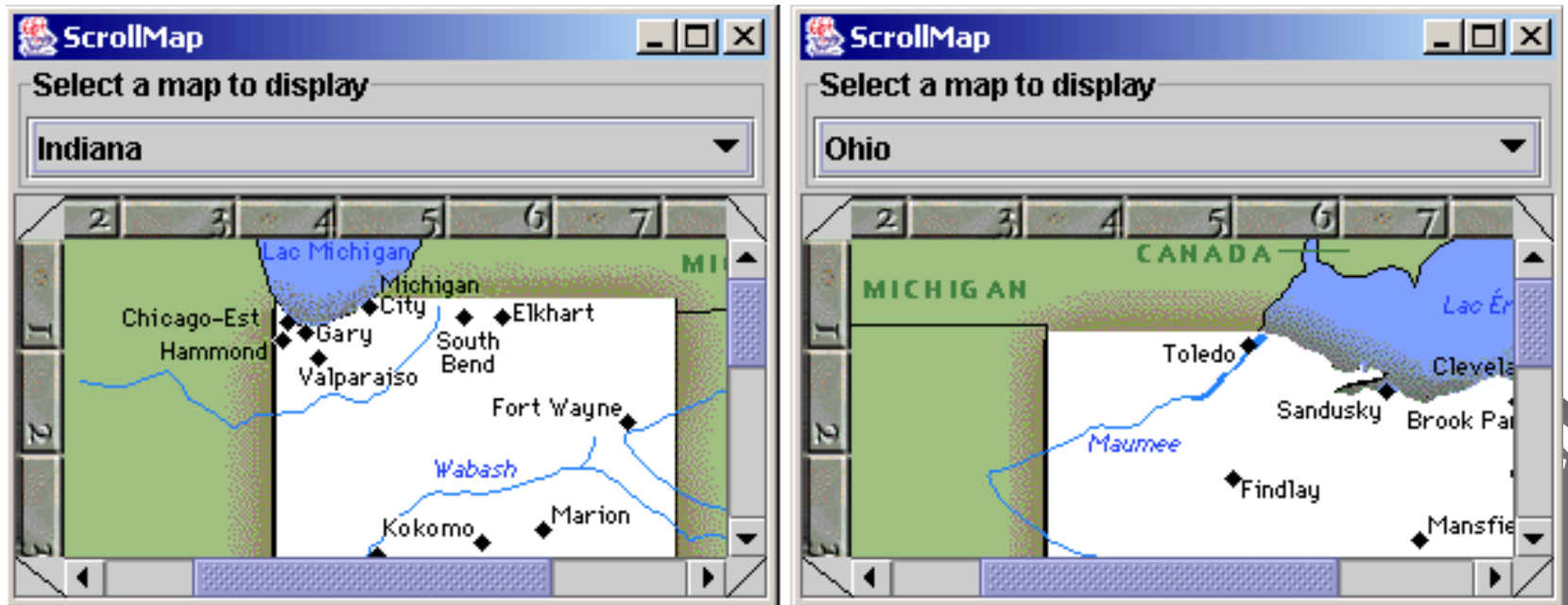
// Set row header, column header and corner header
jspMap.setColumnHeaderView(new JLabel(new ImageIcon(getClass().
    getResource("/image/horizontalRuler.gif"))));
jspMap.setRowHeaderView(new JLabel(new ImageIcon(getClass().
    getResource("/image/verticalRuler.gif"))));
jspMap.setCorner(JScrollPane.UPPER_LEFT_CORNER,
    new CornerPanel(JScrollPane.UPPER_LEFT_CORNER));
jspMap.setCorner(ScrollPaneConstants.UPPER_RIGHT_CORNER,
    new CornerPanel(JScrollPane.UPPER_RIGHT_CORNER));
jspMap.setCorner(JScrollPane.LOWER_RIGHT_CORNER,
    new CornerPanel(JScrollPane.LOWER_RIGHT_CORNER));
jspMap.setCorner(JScrollPane.LOWER_LEFT_CORNER,
    new CornerPanel(JScrollPane.LOWER_LEFT_CORNER));

```



# Example: Using Scroll Panes

Problem: This example uses a scroll pane to browse a large map. The program lets you choose a map from a combo box and display it in the scroll pane,

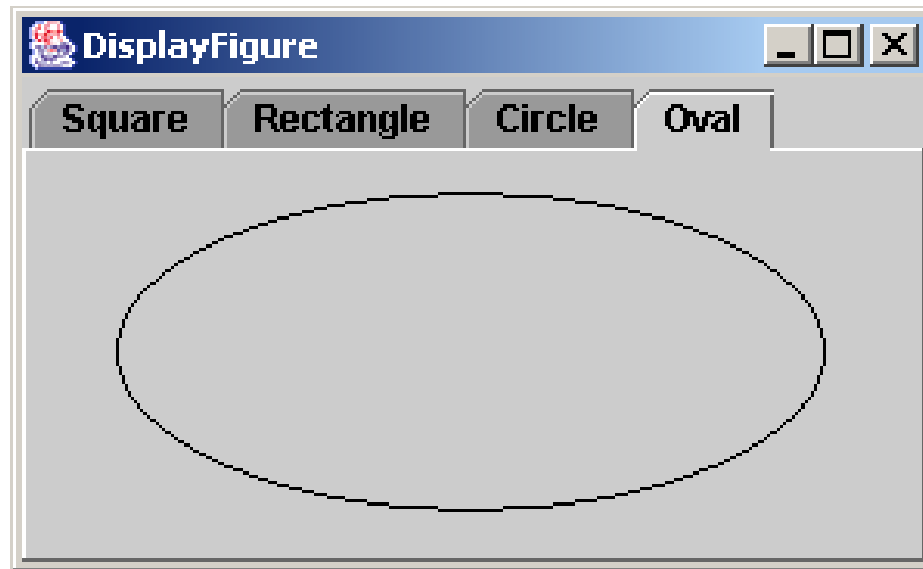


ScrollMap

Run

# JTabbedPane


- ➡ A *tabbed pane* provides a set of mutually exclusive tabs for accessing multiple components.





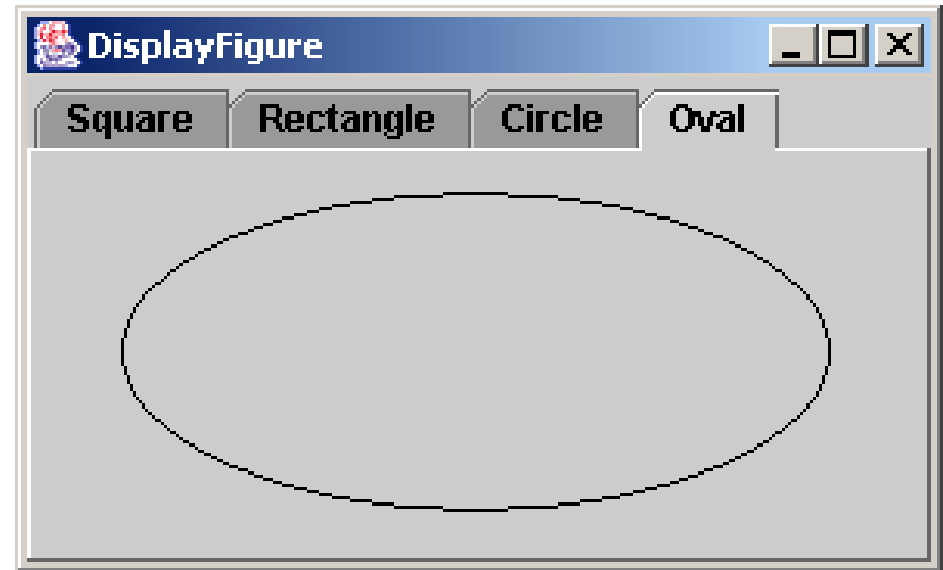
# Using JTabbedPane

Usually you place the panels inside a JTabbedPane and associate a tab with each panel. JTabbedPane is easy to use, since the selection of the panel is handled automatically by clicking the corresponding tab. You can switch between a group of panels by clicking on a tab with a given title and/or icon.

<i>JComponent</i>	
	
<i>JTabbedPane</i>	
#tabPlacement: int	The tab placement for this tabbed pane. Possible values are: JTabbedPane.TOP, JTabbedPane.BOTTOM, JTabbedPane.LEFT, and JTabbedPane.RIGHT. (default: JTabbedPane.TOP).
+JTabbedPane()	Constructs a JTabbedPane with default tab placement.
+JTabbedPane(tabPlacement: int)	Constructs a JTabbedPane with the specified tab placement.
+getIconAt(index: int): Icon	Returns the icon at the specified tab index.
+setIconAt(index: int, icon: Icon): void	Sets the icon at the specified tab index.
+getTabCount(): int	Returns the number of tabs in this tabbed pane.
+getTabPlacement(): int	Returns the placement of the tabs for this tabbed pane.
+setTabPlacement(tabPlacement: int) : void	Sets the placement of the tabs for this tabbed pane.
+getTitleAt(int index) : String	Returns the tab title at the specified tab index.
+setTitleAt(index: int, title: String): void	Sets the tab title at the specified tab index.
+getToolTipTextAt(index: int): String	Returns the tool tip text at the specified tab index.
+setToolTipTextAt(index: int, tooltipText: String): void	Sets the tool tip text at the specified tab index.
+getSelectedComponent(): Component	Returns the currently selected component for this tabbed pane.
+setSelectedComponent(c: Component): void	Sets the currently selected component for this tabbed pane.
+getSelectedIndex(): int	Returns the currently selected index for this tabbed pane.
+setSelectedIndex(index: int): void	Sets the currently selected index for this tabbed pane.
+indexOfComponent(component: Component): void	Returns the index of the tab for the specified component.
+indexOfTab(icon: Icon): int	Returns the index of the tab for the specified icon.
+indexOfTab(title: String): int	Returns the index of the tab for the specified title.

# Example: Using JTabbedPane

Problem: This example uses a tabbed pane with four tabs to display four types of figures: Square, Rectangle, Circle, and Oval. You can select a figure to display by clicking the corresponding tab.

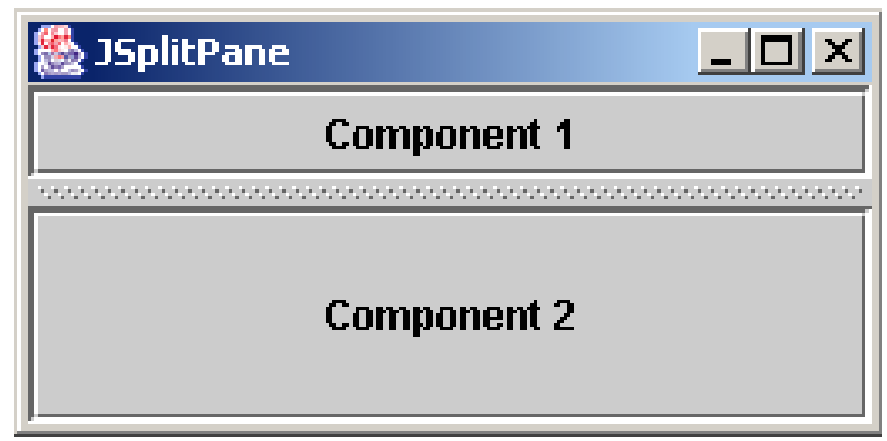
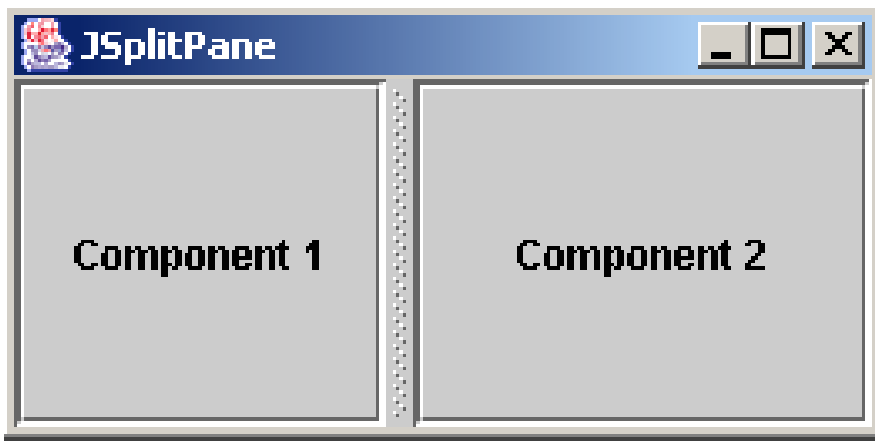


DisplayFigure

Run

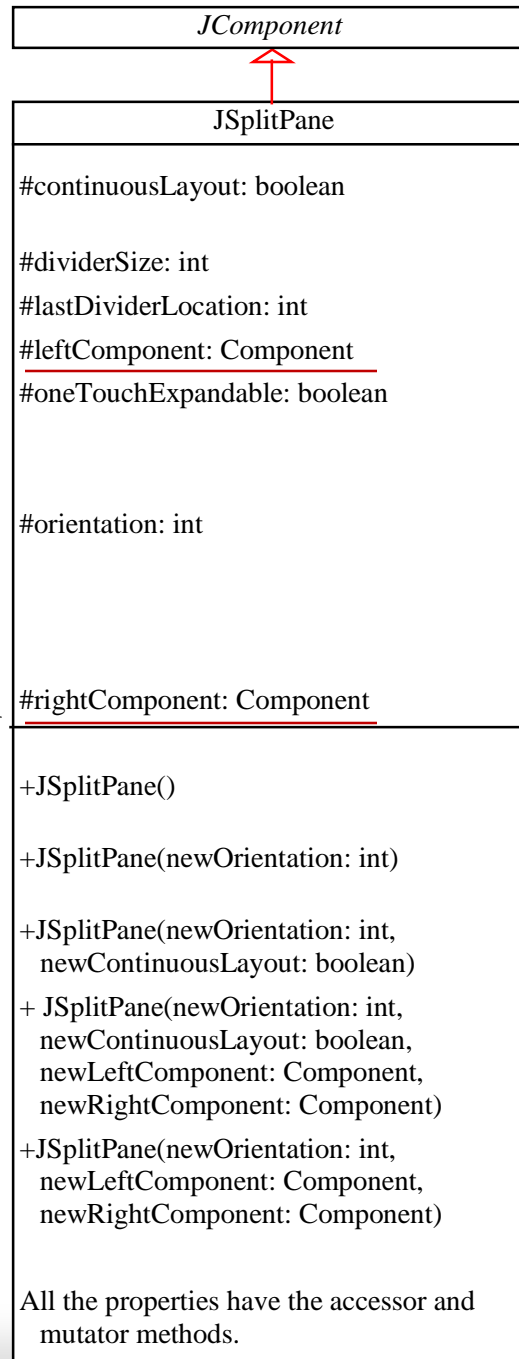
# JSplitPane

JSplitPane is a convenient Swing container that contains two components with a separate bar known as a *divider*.



# Using JSplitPane

The bar can divide the container horizontally or vertically, and can be dragged to change the amount of space occupied by each component.



A Boolean value indicating whether or not the views are continuously redisplayed while resizing.

Size of the divider.

Previous location of the split pane.

The left or top component.

A Boolean property with the default value false. If the property is true, the divider has an expanding and contracting look, so that it can expand and contract with one touch.

Specifies whether the container is divided horizontally or vertically. The possible values are JSplitPane.HORIZONTAL\_SPLIT and JSplitPane.VERTICAL\_SPLIT. The default value is JSplitPane.HORIZONTAL\_SPLIT, which divides the container into a left part and a right part.

The right or bottom component.

Creates a JSplitPane configured to arrange the child components side-by-side horizontally with no continuous.

Creates a JSplitPane configured with the specified orientation and no continuous layout.

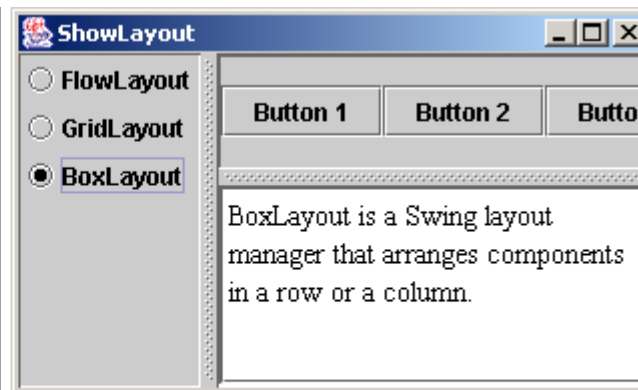
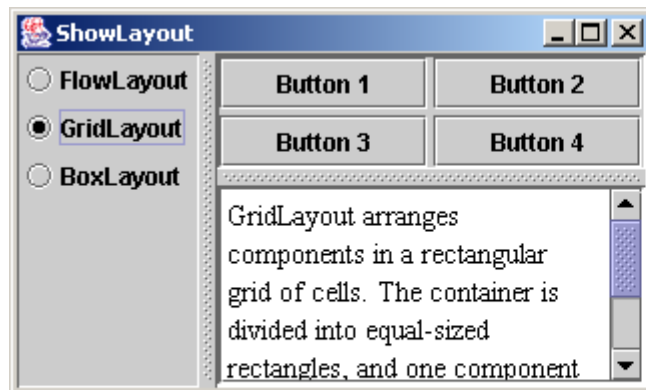
Creates a JSplitPane with the specified orientation and continuous layout.

Creates a JSplitPane with the specified orientation and continuous layout, and the left (top) and right (bottom) components.

Creates a JSplitPane with the specified orientation, and the left (top) and right (bottom) components. No continuous layout.

# Example: Using JSplitPane

Problem: Write a program that uses radio buttons to let the user select a FlowLayout, GridLayout, or BoxLayout manager dynamically for a panel. The panel contains four buttons. The description of the currently selected layout manager is displayed in a text area. The radio buttons, buttons, and text area are placed in two split panes.

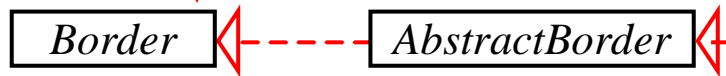


ShowLayout

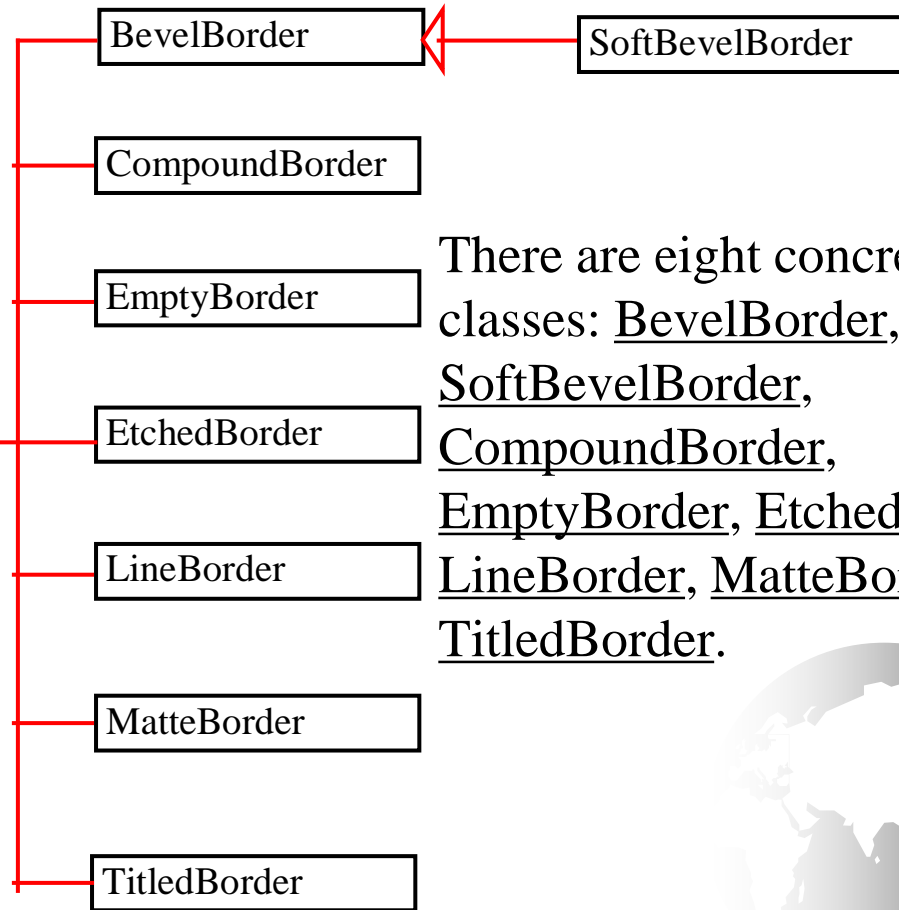
Run

# Swing Borders

A Swing border is defined in the Border interface. Every instance of JComponent can set a border through the border property defined in JComponent. If a border is present, it replaces the inset.



The AbstractBorder class implements an empty border with no size. This provides a convenient base class from which other border classes can be easily derived.



There are eight concrete border classes: BevelBorder, SoftBevelBorder, CompoundBorder, EmptyBorder, EtchedBorder, LineBorder, MatteBorder, and TitledBorder.



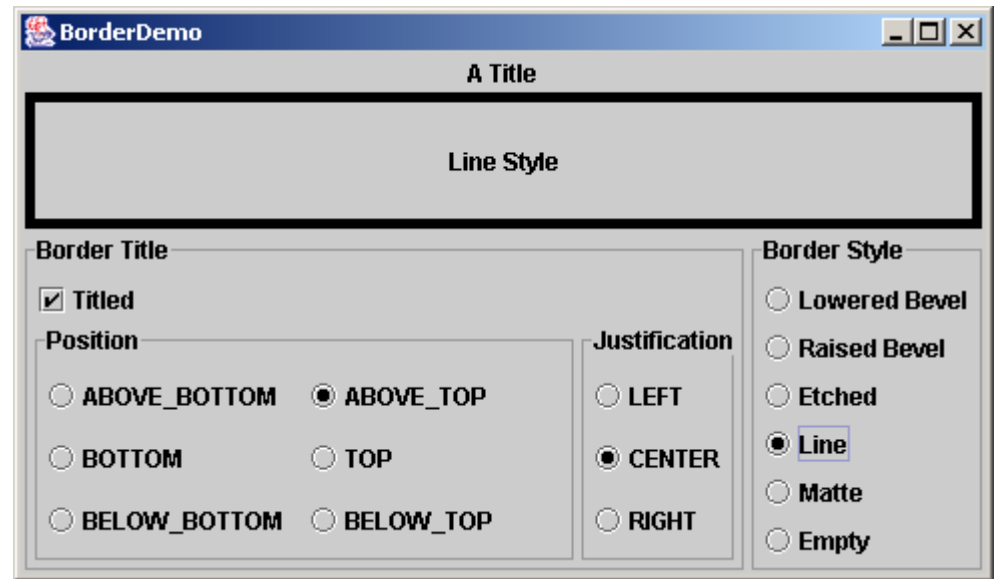
# Static Method for Creating Borders

- ☞ `createTitledBorder(String title)`
- ☞ `createLoweredBevelBorder()`
- ☞ `createRaisedBevelBorder()`
- ☞ `createLineBorder(Color color)`
- ☞ `createLineBorder(Color color, int thickness)`
- ☞ `createEtchedBorder()`
- ☞ `createEtchedBorder(Color highlight, Color shadow, boolean selected)`
- ☞ `createEmptyBorder()`
- ☞ `createMatteBorder(int top, int left, int bottom, int right, Icon tileIcon)`
- ☞ `createCompoundBorder(Border outsideBorder, Border insideBorder)`



# Example: Using Borders

Problem: This example gives a program that creates and displays various types of borders. You can select a border with a title or without a title. For a border without a title, you can choose a border style from Lowered Bevel, Raised Bevel, Etched, Line, Matte, or Empty. For a border with a title, you can specify the title position and justification. You can also embed another border into a titled border.



BorderDemo

Run

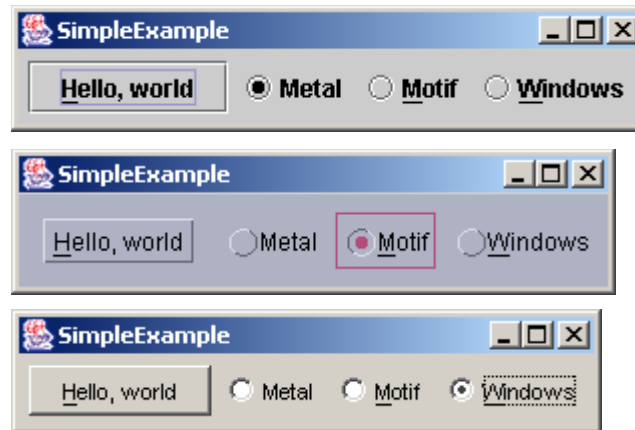


# Pluggable Look-and-Feel

The pluggable look-and-feel feature lets you design a single set of GUI components that **automatically has the look-and-feel of any OS platform**. The implementation of this feature is independent of the underlying native GUI, yet it can imitate the native behavior of the native GUI.

Currently, Java supports the following three look-and-feel styles:

- Metal
- Motif
- Windows



# Setting Look-And-Feel

The javax.swing.UIManager class manages the look-and-feel of the user interface. You can use one of the following three methods to set the look-and-feel for Metal, Motif, or Windows:

```
UIManager.setLookAndFeel
```

```
(UIManager.getCrossPlatformLookAndFeelClassName());
```

```
UIManager.setLookAndFeel
```

```
(new com.sun.java.swing.plaf.motif.MotifLookAndFeel());
```

```
UIManager.setLookAndFeel
```

```
(new com.sun.java.swing.plaf.windows.WindowsLookAndFeel());
```



# Setting Look-And-Feel in Static Initialization Block

To ensure that the setting takes effect, the setLookAndFeel method **should be executed before any** of the components are instantiated. Thus, you can put the code in a static block, as shown below:

```
static {  
    try {  
        // Set a look-and-feel, e.g.,  
        // UIManager.setLookAndFeel  
        //   (UIManager.getCrossPlatformLookAndFeelClassName());  
    }  
    catch (UnsupportedLookAndFeelException ex) {}  
}
```

Static initialization blocks are executed when the class is loaded.

