JAVA

SWING

Presented by Bartosz Sakowicz

Overview of Swing

• **The Swing** package is part of the Java^{TM} Foundation Classes (JFC) in the Java platform.

• The JFC encompasses a group of features to help people build GUIs; Swing provides all the components from buttons to split panes and tables.

• The Swing package was first available as an add-on to JDK 1.1. Prior to the introduction of the Swing package, the Abstract Window Toolkit (AWT) components provided all the UI components in the JDK 1.0 and 1.1 platforms.

Overview of Swing(2)

Although the Java 2 Platform still supports the AWT components, it is recommended to use Swing components instead.

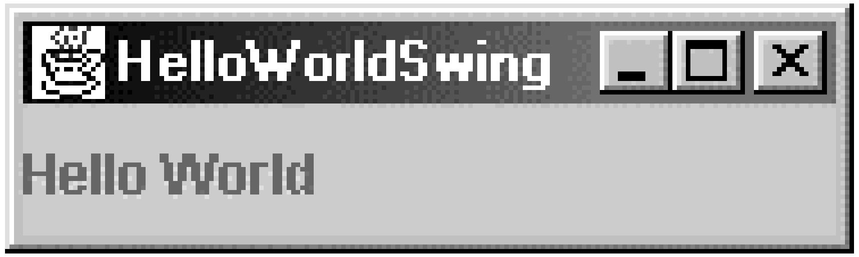
Swing components names start with **J.** The AWT button class, for example, is named Button, whereas the Swing button class is named JButton.

The AWT components are in the java.awt package, whereas the Swing components are in the **javax.swing** package.

HelloWorldSwing

```
import javax.swing.*;

public class HelloWorldSwing {
    public static void main(String[] args) {
        JFrame frame = new JFrame("HelloWorldSwing");
        final JLabel label = new JLabel("Hello World");
        frame.getContentPane().add(label);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          // set close button to close - available from jdk1.3
        frame.pack();   // size the window to fit the preferred size and
                        // layouts of its subcomponents.
        frame.setVisible(true);          // show window
    }
}
```



Basic Swing Application

The basic code in every Swing program:

1. Import necessary packages:

```
import javax.swing.*; // for swing components
import java.awt.*;
import java.awt.event.*;
// Swing components use the AWT
// infrastructure, including the AWT event model.
```

2. Set up a top-level container.

JFrame - implements a single main window.

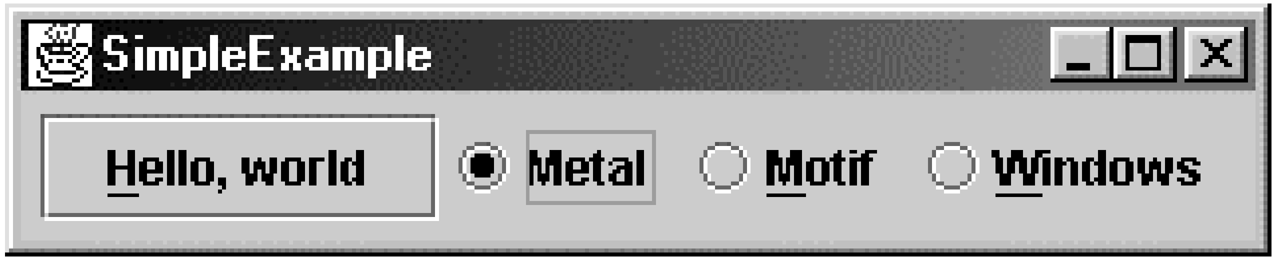
JDialog - implements a secondary window (a window that's dependent on another window).

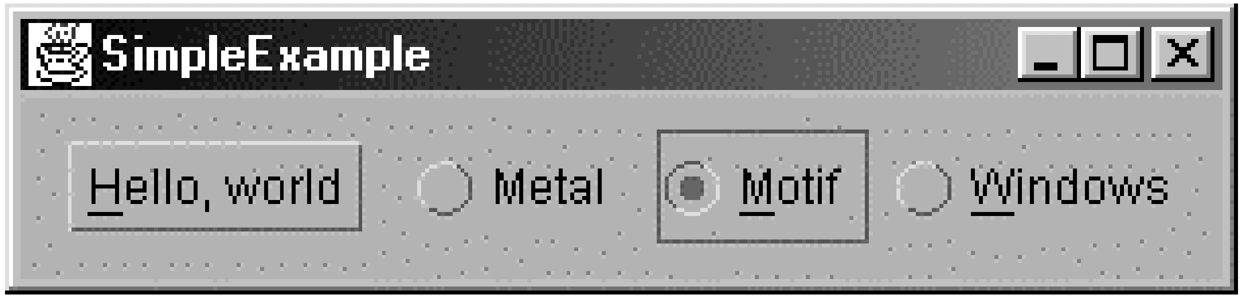
JApplet - implements an applet's display area within a browser window.

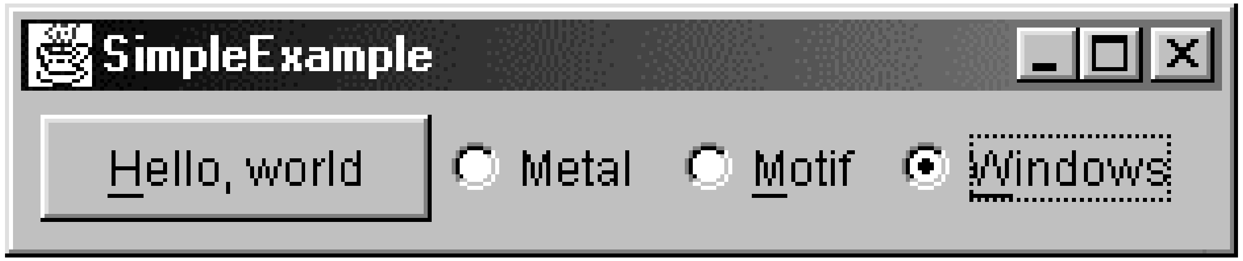
Look and feel

There are few views of a GUI that uses Swing components:

UIManager.setLookAndFeel(...)







Setting Look and feel

Arguments you can use for setLookAndFeel:

UIManager.getCrossPlatformLookAndFeelClassName()

Returns the string for the one look-and-feel guaranteed to work -- the Java Look & Feel.

UIManager.getSystemLookAndFeelClassName()

Specifies the look and feel for the current platform. On Microsoft Windows platforms, this specifies the Windows Look & Feel. On Mac OS platforms, this specifies the Mac OS Look & Feel. On Sun platforms, it specifies the CDE/Motif Look & Feel.

"javax.swing.plaf.metal.MetalLookAndFeel"

Specifies the Java Look & Feel. (The codename for this look and feel was *Metal*.) This string is the value returned by the getCrossPlatformLookAndFeelClassName method.

Setting Look and feel(2)

"com.sun.java.swing.plaf.windows.WindowsLookAndFeel"

Specifies the Windows Look & Feel. Currently, you can use this look and feel only on Microsoft Windows systems.

"com.sun.java.swing.plaf.motif.MotifLookAndFeel"

Specifies the CDE/Motif Look & Feel. This look and feel can be used on any platform.

"javax.swing.plaf.mac.MacLookAndFeel"

Specifies the Mac OS Look & Feel, which can be used only on Mac OS platforms.

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.

• All the object has to do is implement the appropriate interface and be registered as an **event listener** on the appropriate event source.

• **Event-handling code executes in an single thread, the event-dispatching thread.** This ensures that each event handler finishes execution before the next one executes.

Event handlers

Every event handler requires three pieces of code:

1. 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. **Example:**

public class MyClass implements ActionListener {

2. Another line of code registers an instance of the event handler class as a listener on one or more components. **Example:**

someComponent.addActionListener(instanceOfMyClass);

3. In the event handler class, a few lines of code implement the methods in the listener interface. For example:

public void actionPerformed(ActionEvent e) {

...//code that reacts to the action... }

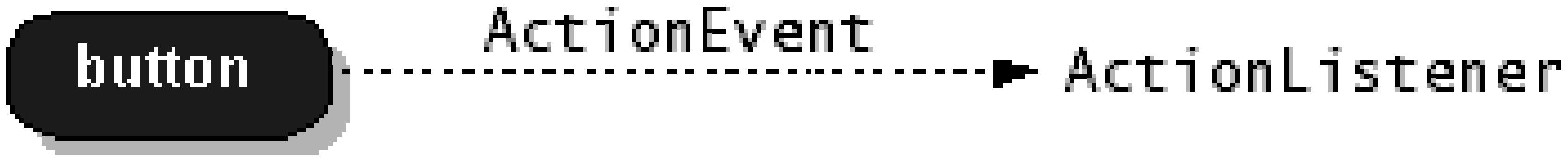
Event handlers(2)

Event handlers can be instances of any class.

Often an event handler that has only a few lines of code is implemented using an *anonymous inner class.* **Anonymous inner classes can be confusing.**

Example:

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        numClicks++;
        label.setText(labelPrefix + numClicks);
    }
});
```



Presented by Bartosz Sakowicz

DMCS TUL

Writing event handlers

The code that implements the event handling for the button:

```
public class Beeper ... implements ActionListener {
        ... //where initialization occurs:
        button.addActionListener(this);
        ...
        public void actionPerformed(ActionEvent e) {
                ...//Make a beep sound...
        }
}
```

Kinds of events

Examples of Events and Their Associated Event Listeners

Act that Results in the Event	**Listener Type**
User clicks a button, presses Return 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***

Using adapters

Most listener interfaces contain more than one method. For example, the MouseListener interface contains five methods: **mousePressed, mouseReleased, mouseEntered, mouseExited, and mouseClicked.**

Even if you care only about mouse clicks, if your class directly implements MouseListener, then you must implement all five MouseListener methods. Methods for those events you don't care about can have empty bodies.

To help you avoid cluttering your code with empty method bodies, the API generally includes an *adapter* class for each listener interface with more than one method. For example, the MouseAdapter class implements the MouseListener interface. An adapter class implements empty versions of all its interface's methods.

Using adapters(2)

Example:

```
public class MyClass extends MouseAdapter {
	...
	someObject.addMouseListener(this);
	...
	public void mouseClicked(MouseEvent e) {
		...//Event handler implementation goes here...
	}
}
```

The action event API

The ActionListener interface contains a single method, and thus has no corresponding adapter class:

void actionPerformed(ActionEvent)

Called just after the user informs the listened-to component that an action should occur.

The actionPerformed method has a single parameter: an ActionEvent object. The ActionEvent class defines two useful methods:

String getActionCommand()

Returns the string associated with this action. Most objects that can fire action events support a method called setActionCommand that lets you set this string. If you don't set the action command explicitly, then it's generally the text displayed in the component.

The action event API(2)

int getModifiers()

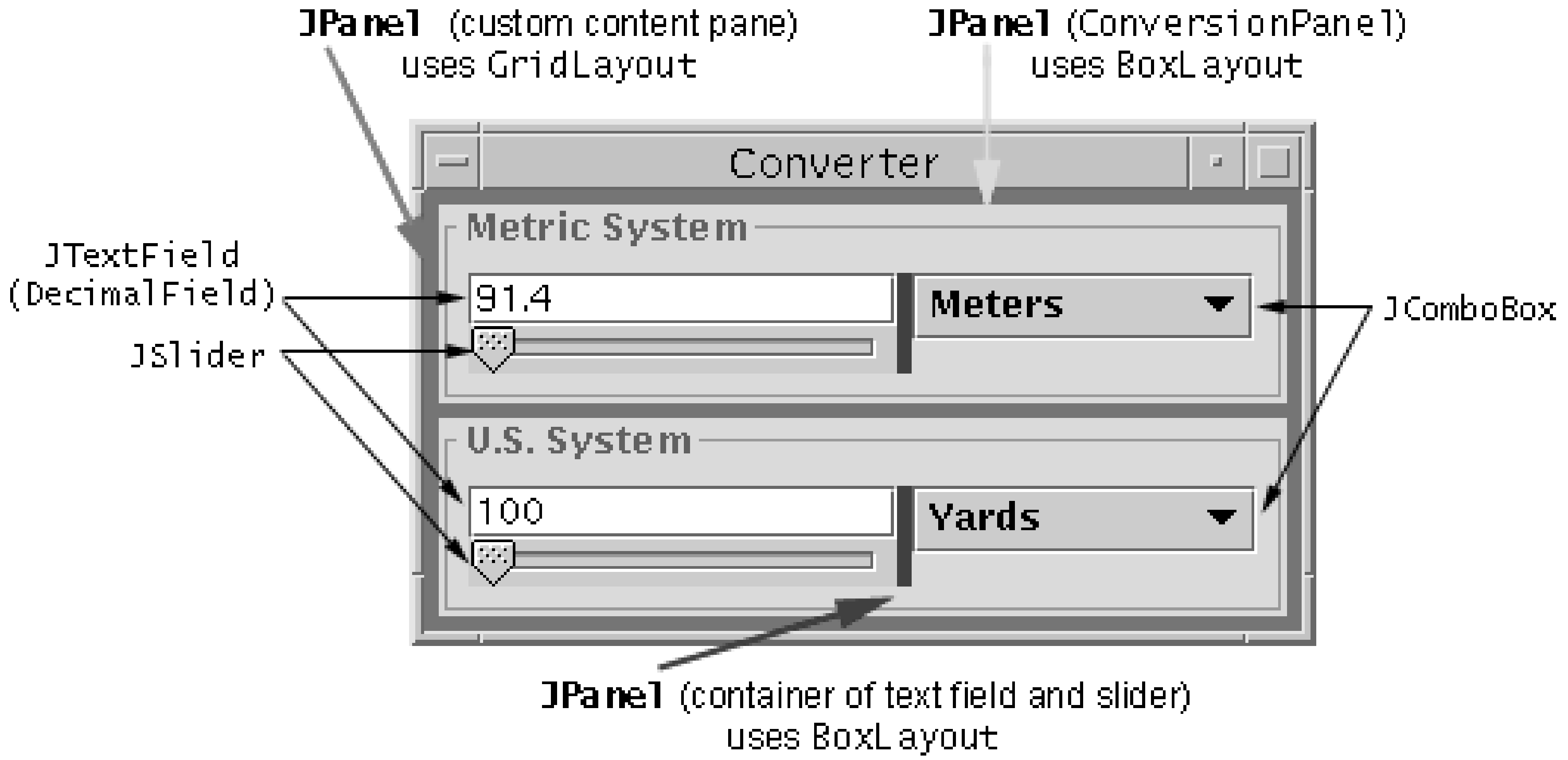
Returns an integer representing the modifier keys the user was pressing when the action event occurred. You can use the ActionEvent-defined constants SHIFT_MASK, CTRL_MASK, META_MASK, and ALT_MASK to determine which keys were pressed.

For example, if the user Shift-selects a menu item, then the following expression is nonzero:

actionEvent.getModifiers() & ActionEvent.SHIFT_MASK

The rest of events can be handled in analogous way. Of course different ones have different API. There is more than 30 different types of listeners.

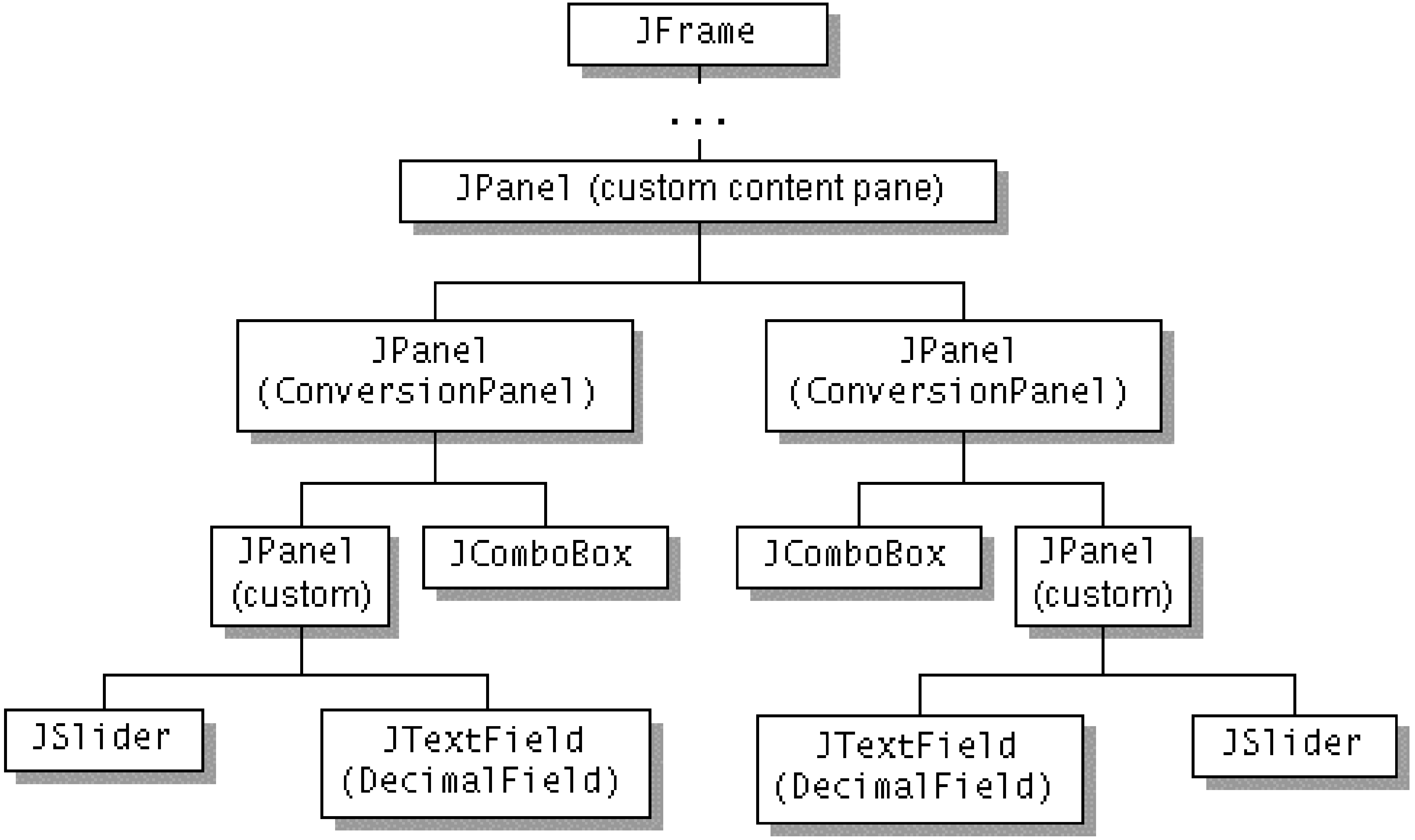
Swing components example



Presented by Bartosz Sakowicz

DMCS TUL

Containment hierarchy example



Swing components

<u>Top-Level Containers</u>
The components at the top of any Swing containment hierarchy.

<u>General-Purpose Containers</u>
Intermediate containers that can be used under many different circumstances.

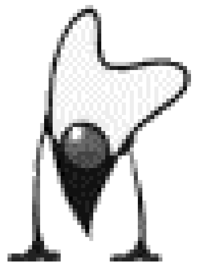
<u>Special-Purpose Containers</u>
Intermediate containers that play specific roles in the UI.

<u>Basic Controls</u>
Atomic components that exist primarily to get input from the user; they generally also show simple state.

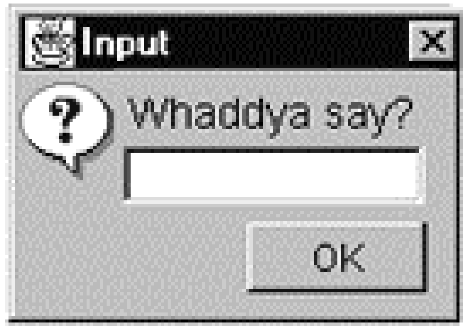
<u>Uneditable Information Displays</u>
Atomic components that exist solely to give the user information.

<u>Editable Displays of Formatted Information</u>
Atomic components that display highly formatted information that (if you choose) can be edited by the user.

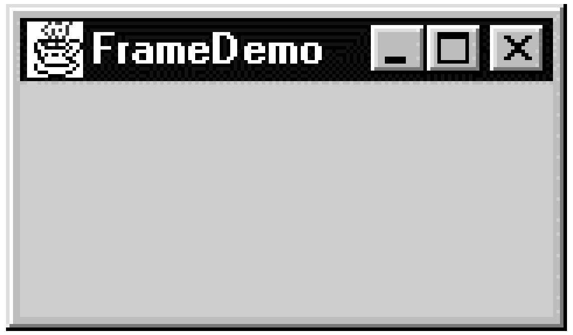
Top-level containers



Applet

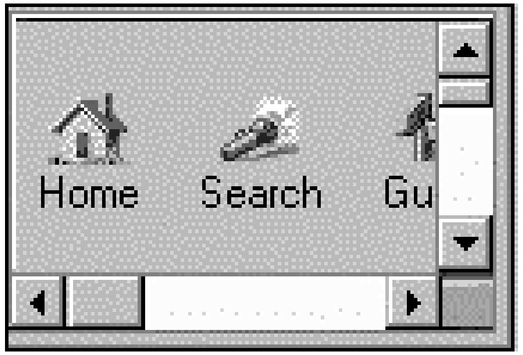


Dialog

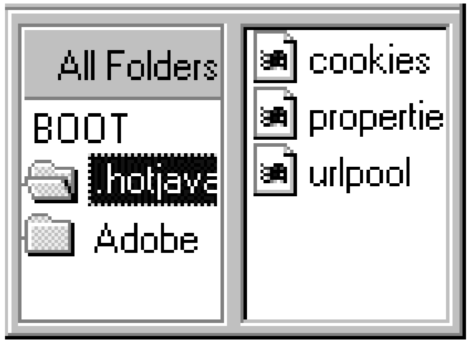


Frame

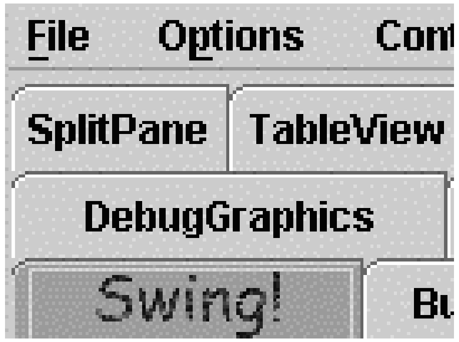
General-purpose containers



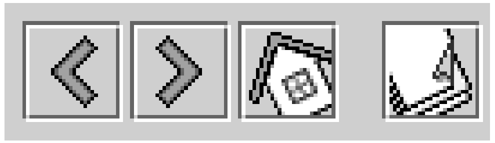
Scroll pane



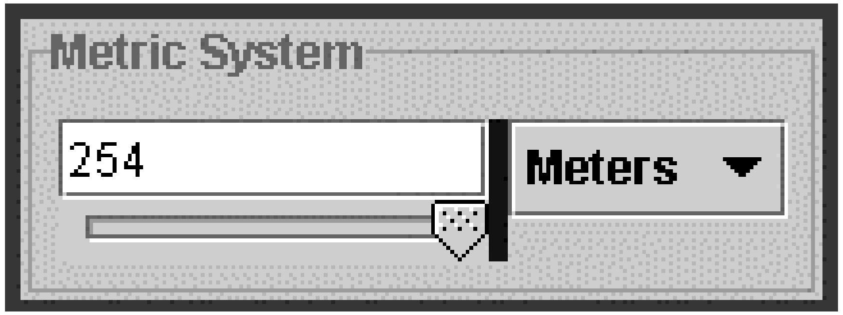
Split pane



Tabbed pane



Tool bar

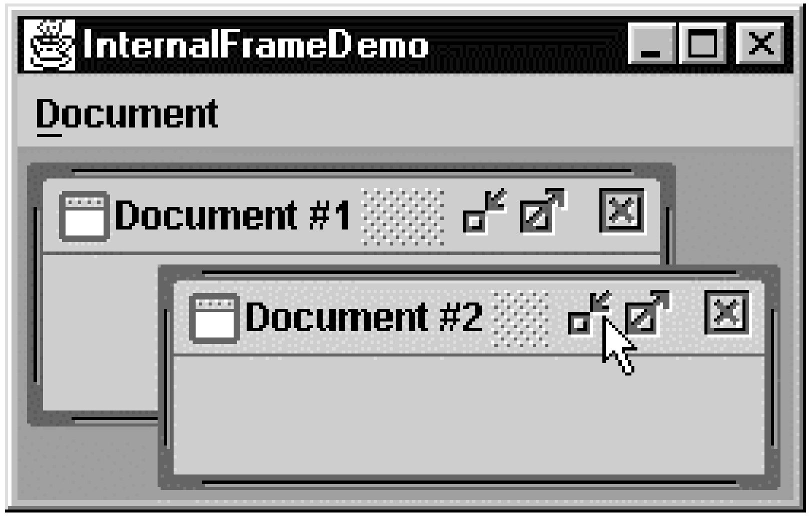


Panel

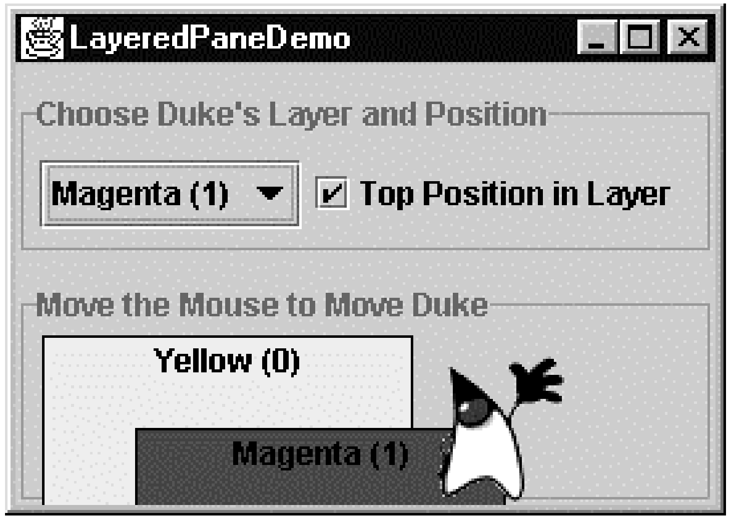
Presented by Bartosz Sakowicz

DMCS TUL

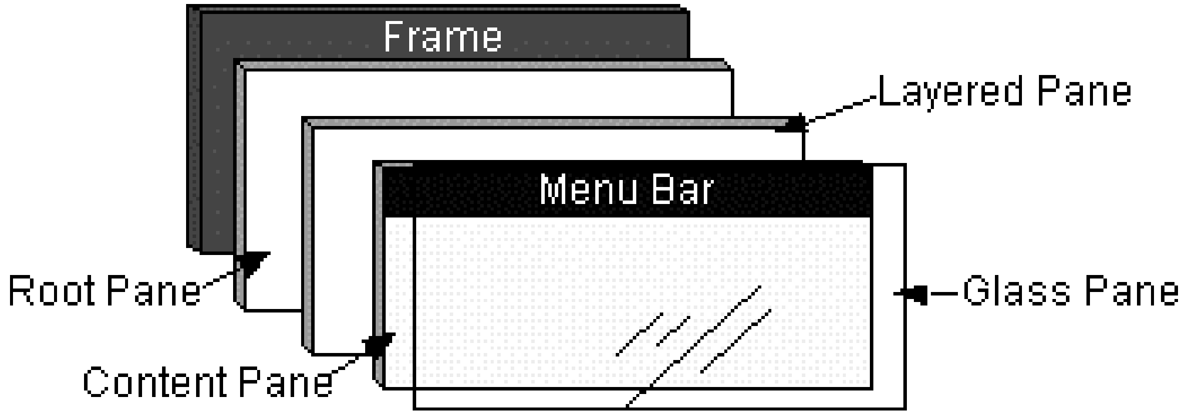
Special-purpose containers



Internal frame



Layered pane

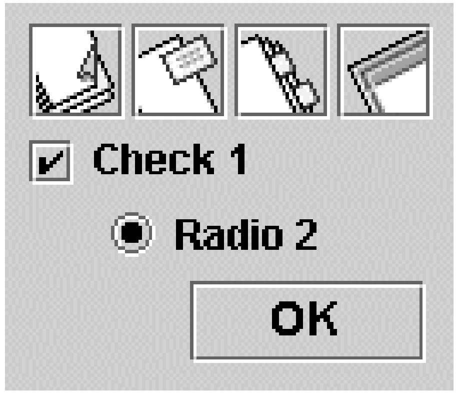


Root pane

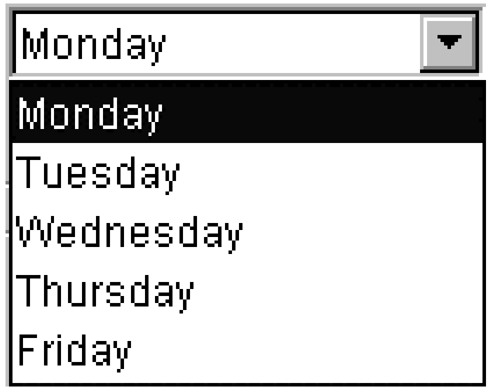
Presented by Bartosz Sakowicz

DMCS TUL

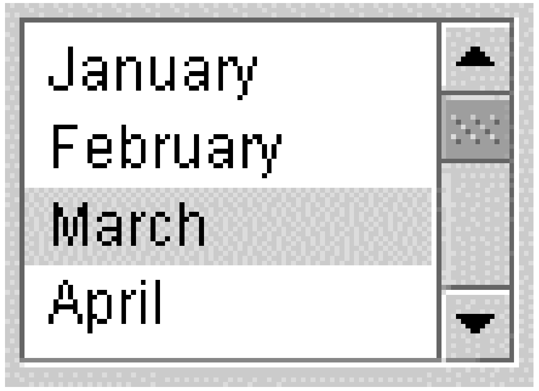
Basic controls



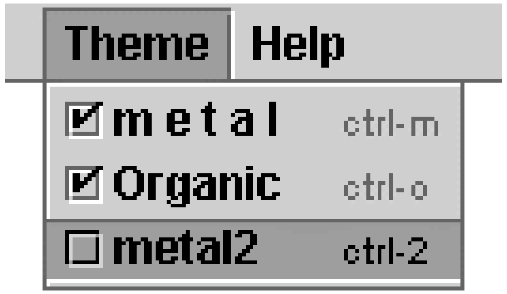
Buttons



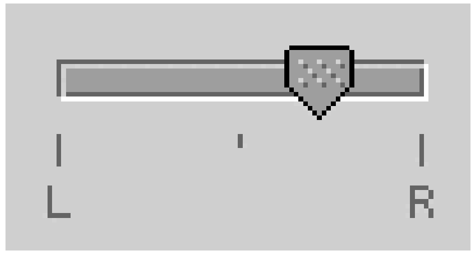
Combo box



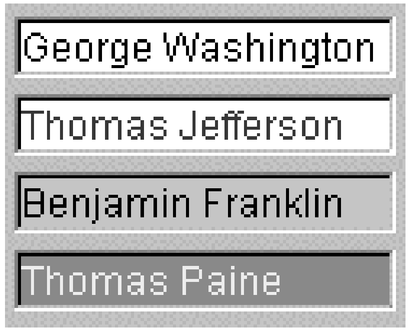
List



Menu



Slider

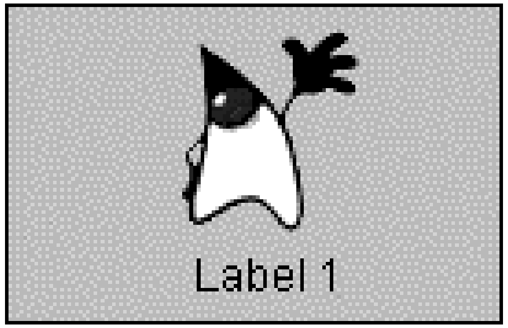


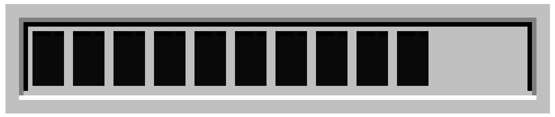
Text fields

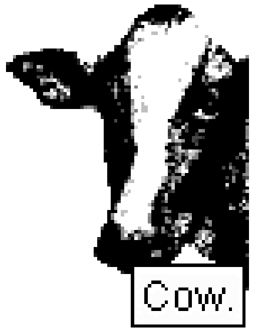
Presented by Bartosz Sakowicz

DMCS TUL

Uneditable information displays

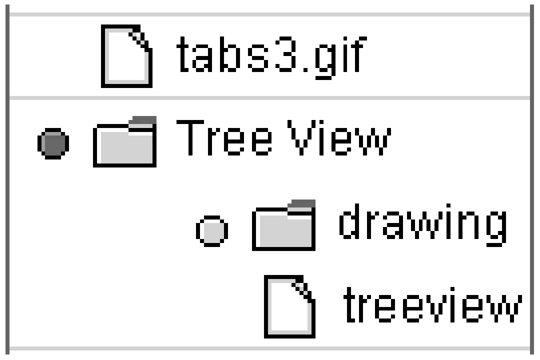




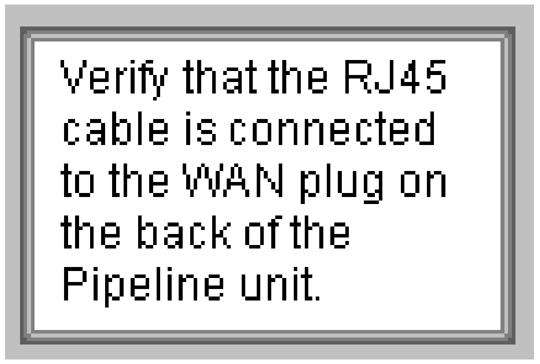


Label **Progress bar** **Tooltip**

Editable information displays

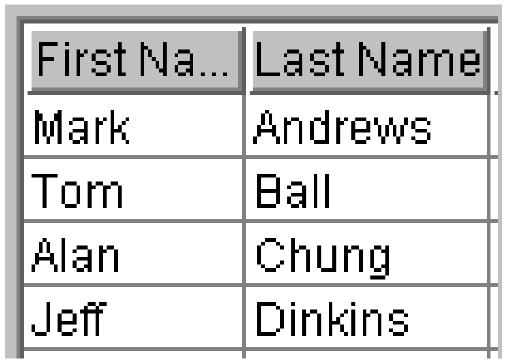


Tree



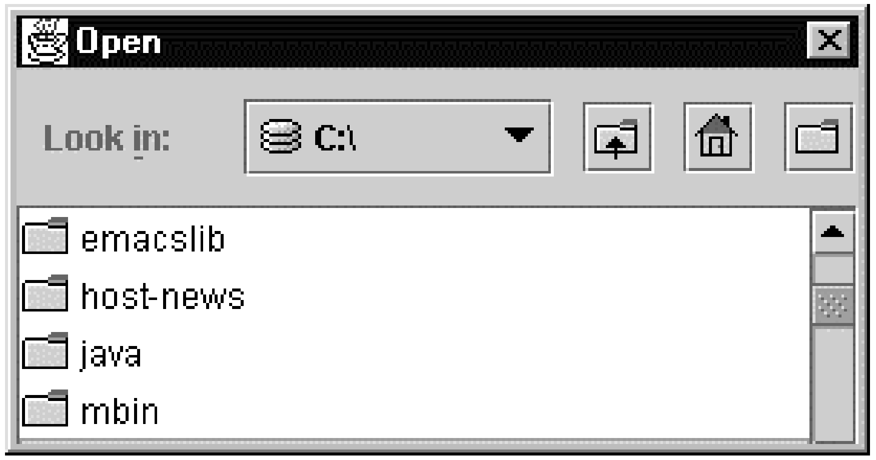
Verify that the RJ45 cable is connected to the WAN plug on the back of the Pipeline unit.

Text

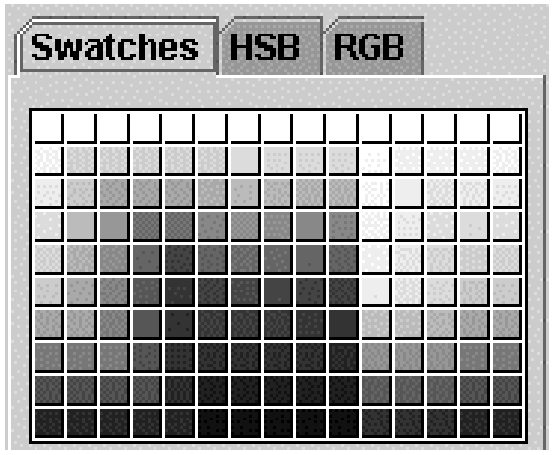


First Na...	Last Name
Mark	Andrews
Tom	Ball
Alan	Chung
Jeff	Dinkins

Table



File chooser

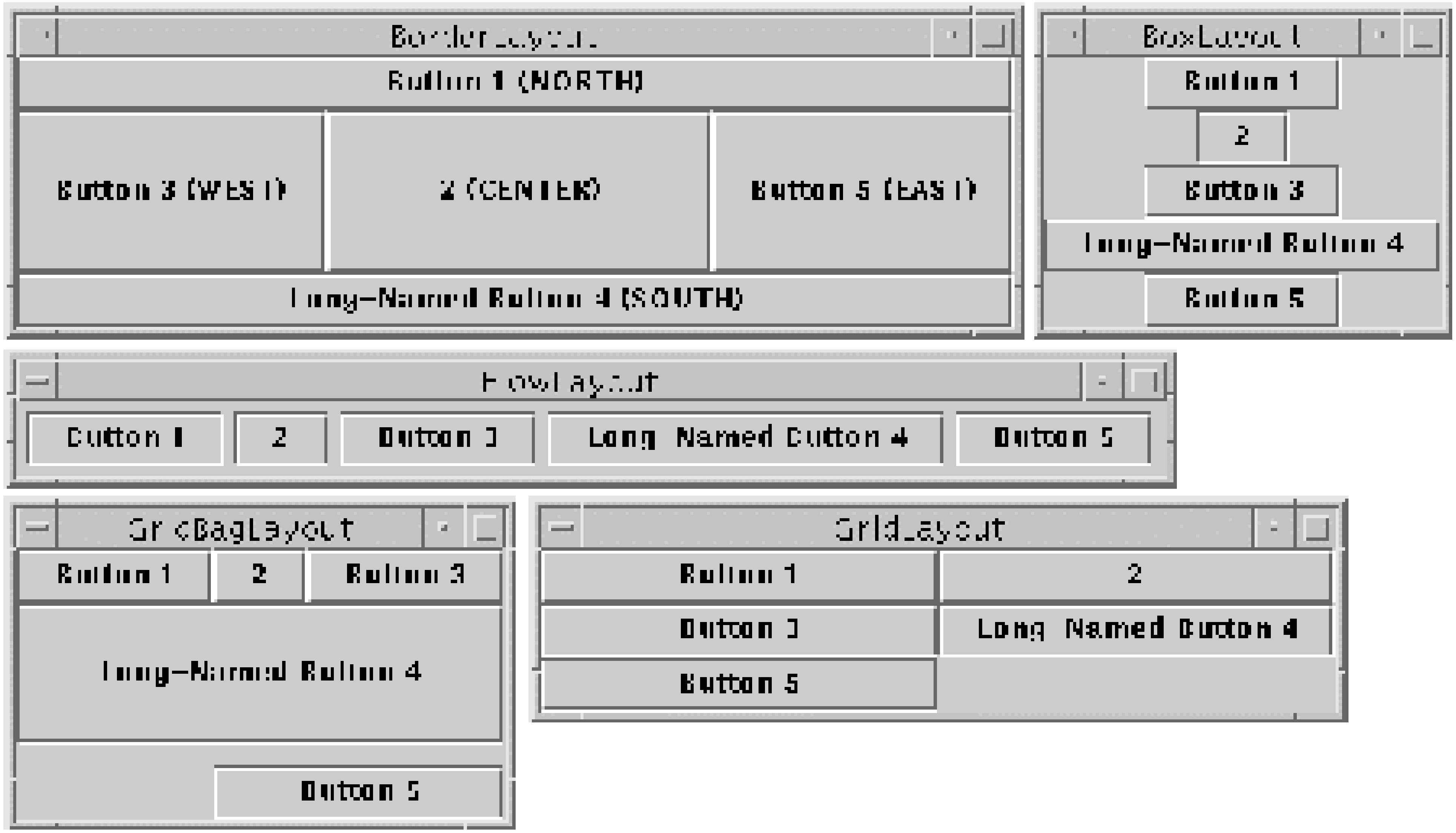


Color chooser

Presented by Bartosz Sakowicz

DMCS TUL

Layout management



Different layouts with identical code.

Presented by Bartosz Sakowicz

DMCS TUL

Layout management(2)

Layout management is the process of determining the size and position of components.

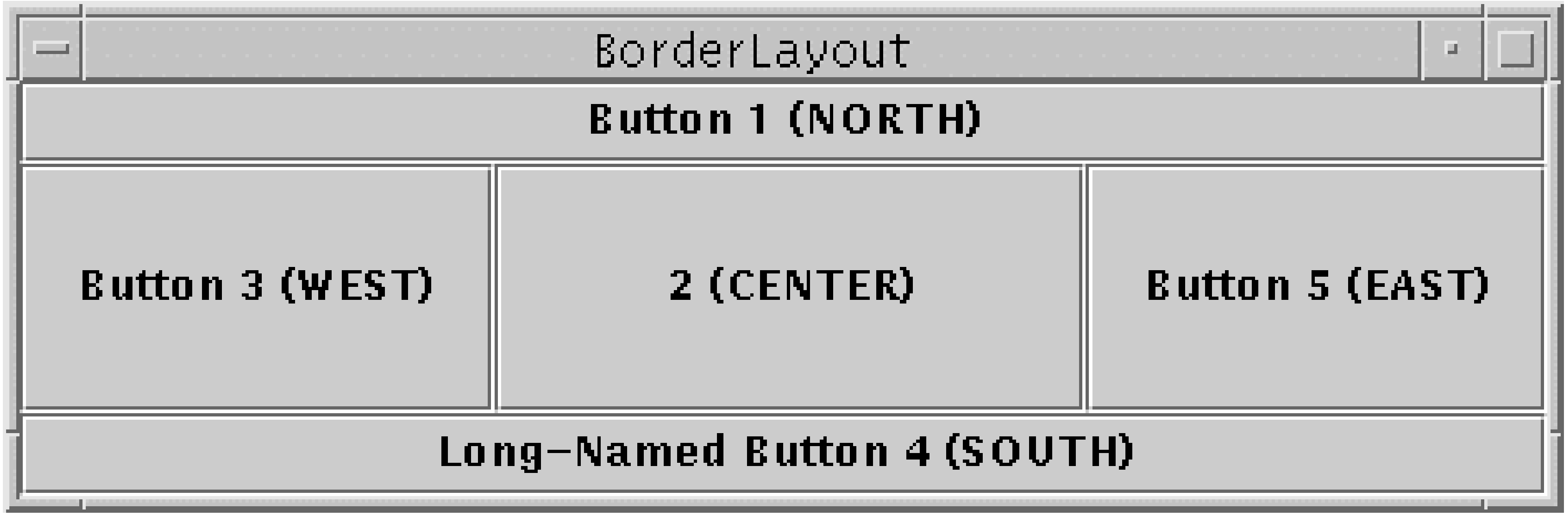
By default, each container has a **layout manager** -- an object that performs layout management for the components within the container.

Components can provide size and alignment hints to layout managers, but layout managers have the final say on the size and position of those components.

The Java platform supplies five (there are more - eg. CardLayout) commonly used layout managers: **BorderLayout, BoxLayout, FlowLayout, GridBagLayout,** and **GridLayout.**

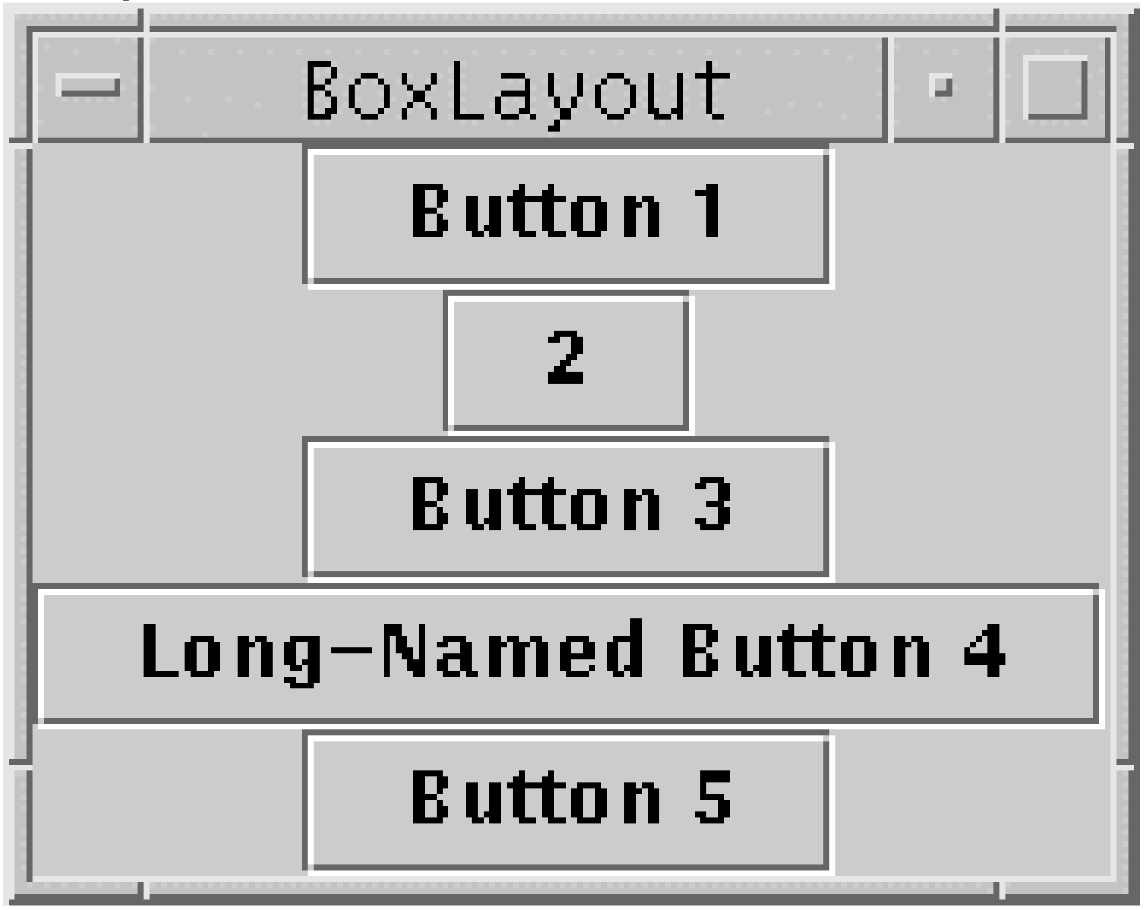
BorderLayout

BorderLayout is the default layout manager for every content pane. A BorderLayout has five areas available to hold components: north, south, east, west, and center. All extra space is placed in the center area.



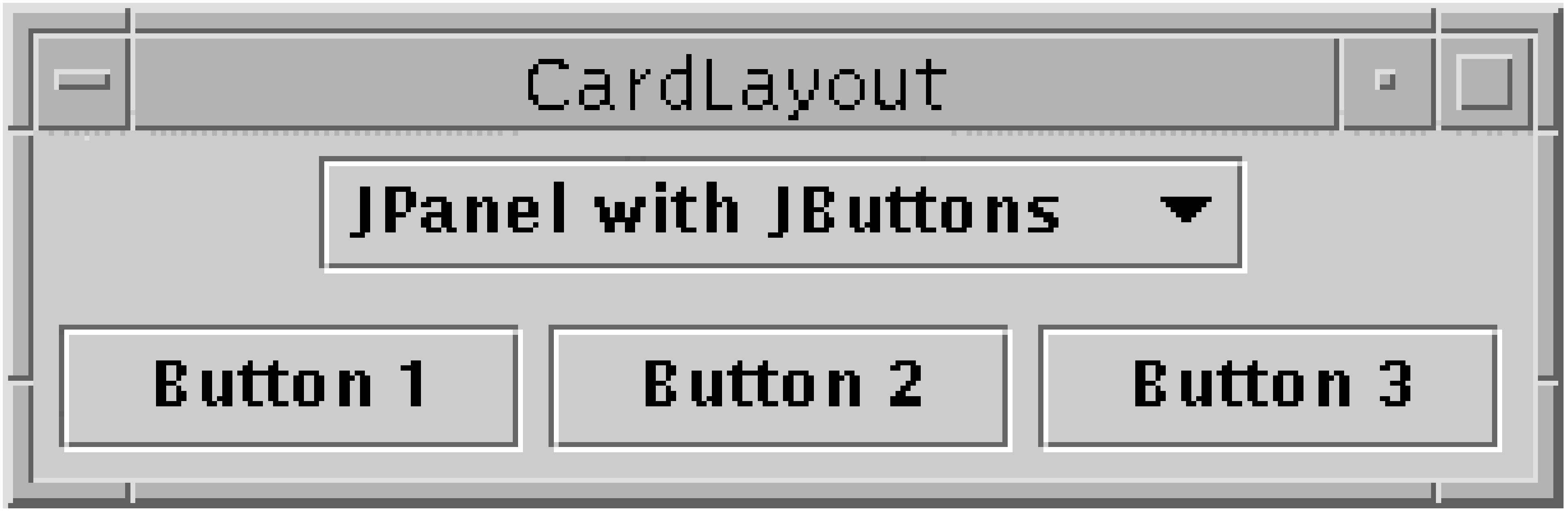
BoxLayout

The BoxLayout class puts components in a single row or column. It respects the components' requested maximum sizes, and also lets you align components.



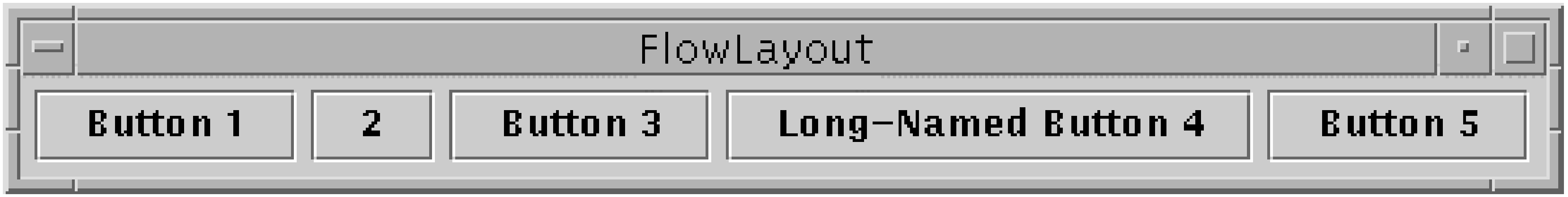
CardLayout

The CardLayout class 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.



FlowLayout

FlowLayout is the default layout manager for every JPanel. It simply lays out components from left to right, starting new rows if necessary.

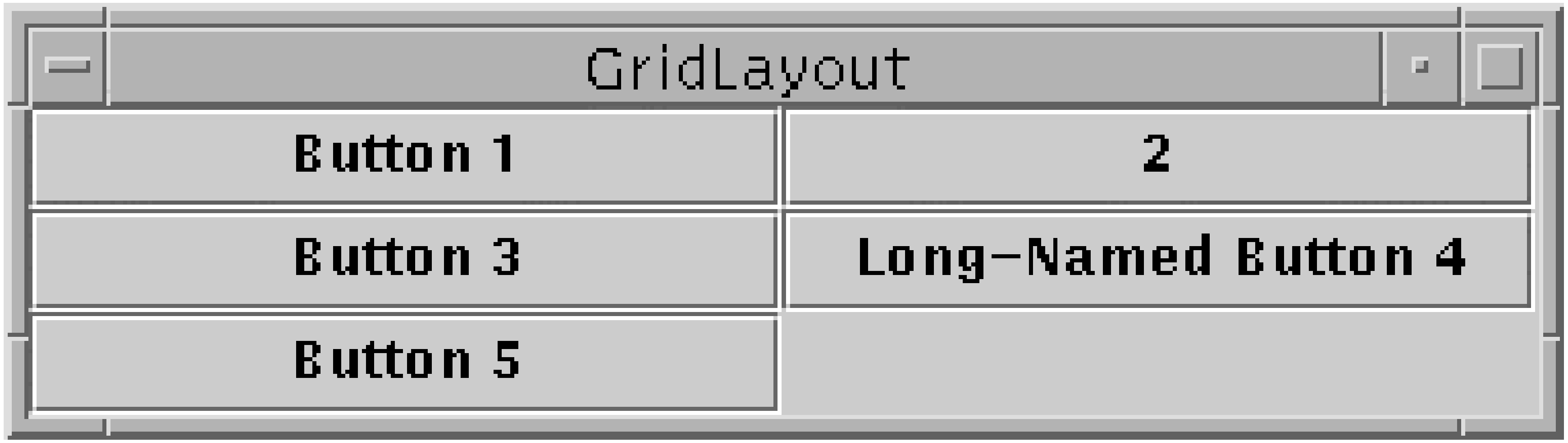


Presented by Bartosz Sakowicz

DMCS TUL

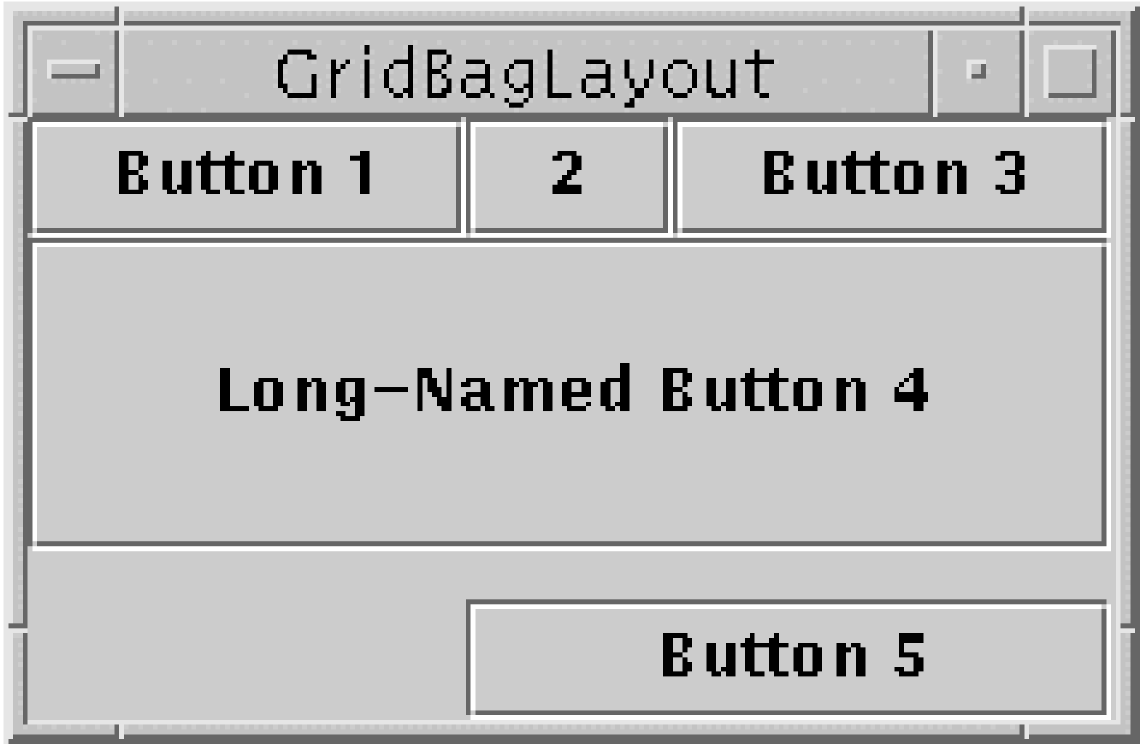
GridLayout

GridLayout simply makes a bunch of components equal in size and displays them in the requested number of rows and columns.



GridBagLayout

GridBagLayout is the most sophisticated, flexible layout manager the Java platform provides. 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 aren't necessarily all the same height; similarly, grid columns can have different widths.



Setting the Layout Manager

JPanel pane = new JPanel();

pane.setLayout(new BorderLayout());

It is possible perform layout without managers. By setting a container's layout property to null, you make the container use no layout manager. With this strategy, called **absolute positioning**, you must specify the size and position of every component within that container.

Providing Hints about a Component

You do this by specifying the minimum, preferred, and maximum sizes of the component. You can invoke the component's methods for setting size hints -- **setMinimumSize, setPreferredSize, and setMaximumSize.**

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.

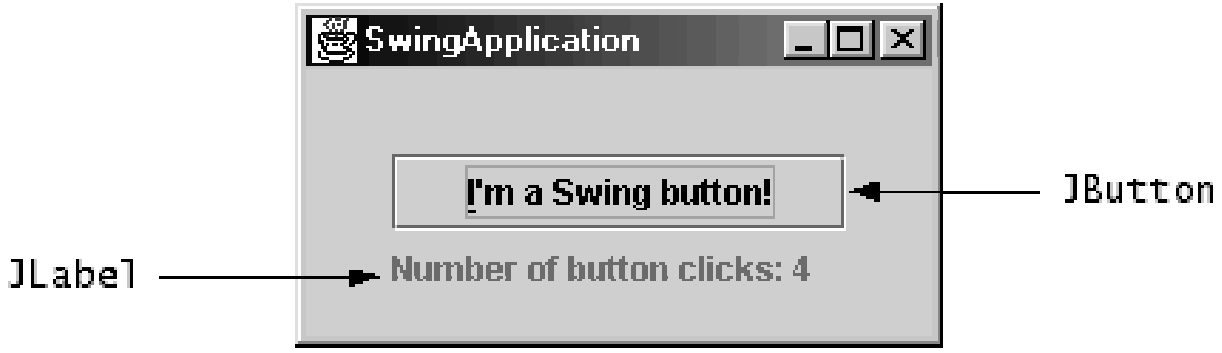
Invisible components - You can create lightweight components that perform no painting, but that can take up space in the GUI.

Empty borders - No matter what the layout manager, you can affect the apparent amount of space between components by adding empty borders to components.

Adding borders around components

To add borders to yours components you use **setBorder** method:

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



Absolute positioning

```
public class NoneWindow extends JFrame {
private boolean laidOut = false;
private JButton b1, b2, b3;
public NoneWindow() {
        Container contentPane = getContentPane();
        contentPane.setLayout(null);
        b1 = new JButton("one"); contentPane.add(b1);
        b2 = new JButton("two"); contentPane.add(b2);
        b3 = new JButton("three"); contentPane.add(b3);
```

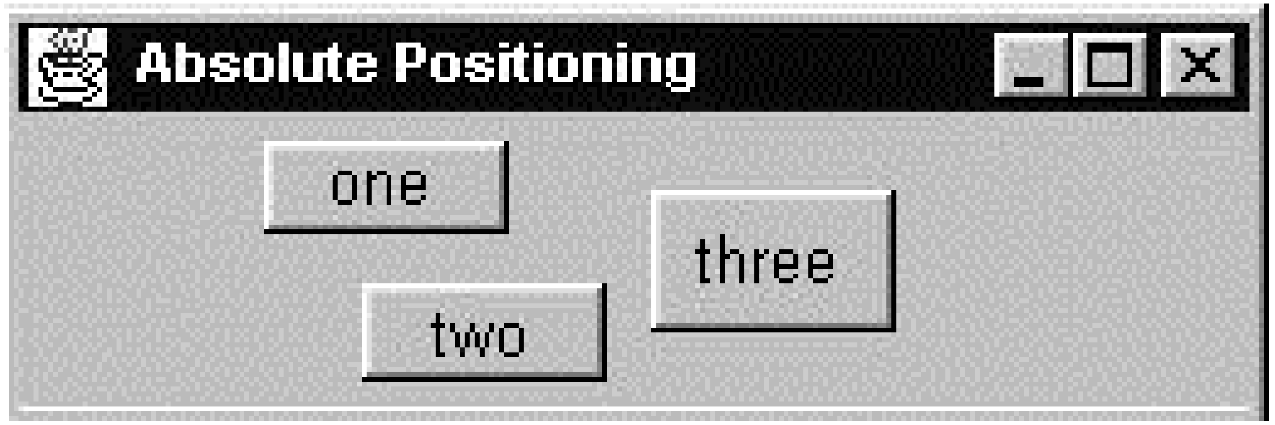
Absolute positioning(2)

```
        Insets insets = contentPane.getInsets();

        b1.setBounds(25 + insets.left, 5 + insets.top, 75, 20);
        b2.setBounds(55 + insets.left, 35 + insets.top, 75, 20);
        b3.setBounds(150 + insets.left, 15 + insets.top, 75, 30);

    }

    }
```



Using intermediate Swing components

Swing provides several general-purpose intermediate containers:

Panel

The most flexible, frequently used intermediate container. Implemented with the JPanel class, panels add almost no functionality beyond what all JComponent objects have. They are often used to group components, whether because the components are related or just because grouping them makes layout easier. A panel can use any layout manager, and you can give it a border. The content panes of top-level containers are often implemented as JPanel instances.

Scroll Pane

Provides scroll bars around a large or growable component.

Using intermediate Swing components(2)

Split Pane

Displays two components in a fixed amount of space, letting the user adjust the amount of space devoted to each component.

Tabbed Pane

Contains multiple components but shows only one at a time. The user can easily switch between components.

Tool Bar

Holds a group of components (usually buttons) in a row or column, optionally allowing the user to drag the tool bar into different locations.

Using intermediate Swing components(3)

The rest of the Swing intermediate containers are more specialized:

<u>Internal Frame</u>

Looks like a frame and has much the same API, but must appear within another window.

<u>Layered Pane</u>

Provides a third dimension, depth, for positioning components. You specify the position and size of each component. One type of layered pane, a desktop pane, is designed primarily to contain and manage internal frames.

Using atomic components

The following atomic components exist primarily to get input from the user:

Button, Check Box, Radio Button

Provides easy-to-use, easy-to-customize button implementations.

Combo Box

Provides both uneditable and editable combo boxes -- buttons that bring up menus of choices.

List

Displays a group of items that the user can choose.

Menu

Includes menu bar, menu, and menu item implementations, including specialized menu items such as check box menu items.

Using atomic components(2)

Slider

Lets the user choose one of a continuous range of values.

Spinner

Using tiny up/down arrows, lets the user choose from an ordered sequence of items.

Text Field

Lets the user enter a single line of text data.

Using atomic components(3)

Atomic components which exist only to give information:

Label

Presents some text, an icon, or both.

Progress Bar

Displays progress toward a goal.

Tool Tip

Brings up a small window that describes another component.

Using atomic components(4)

Atomic components which provide formatted information and a way of editing it:

Color Chooser

A UI for choosing colors; can be used inside or outside a dialog.

File Chooser

A UI for choosing files and directories.

Table

An extremely flexible component that displays data in a grid format.

Using atomic components(5)

Text Support

A framework including everything from simple text components, such as text fields, to a full-featured, extensible kit for building text editors.

Tree

A component that displays hierarchical data.

Every component has his own API. For details please refer to API specification and Java Tutorial: *http://java.sun.com.*