PHẦN 2

THIẾT KẾ GIAO DIỆN SỬ DỤNG SWING

What is a SWING

- Swing is the next-generation GUI toolkit that Sun Microsystems created to enable enterprise development in Java. By *enterprise development*, we mean that programmers can use Swing to create large-scale Java applications with a wide array of powerful components. In addition, you can easily extend or modify these components to control their appearance and behavior .
- Swing is actually part of a larger family of Java products known as the Java Foundation Classes (`JFC`), which incorporate many of the features of Netscape's Internet Foundation Classes (`IFC`) as well as design aspects from IBM's Taligent division and Lighthouse Design.
- 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. Although the Java 2 Platform still supports the `AWT` components, we strongly encourage you to use Swing components instead.

What Are the Java Foundation Classes?

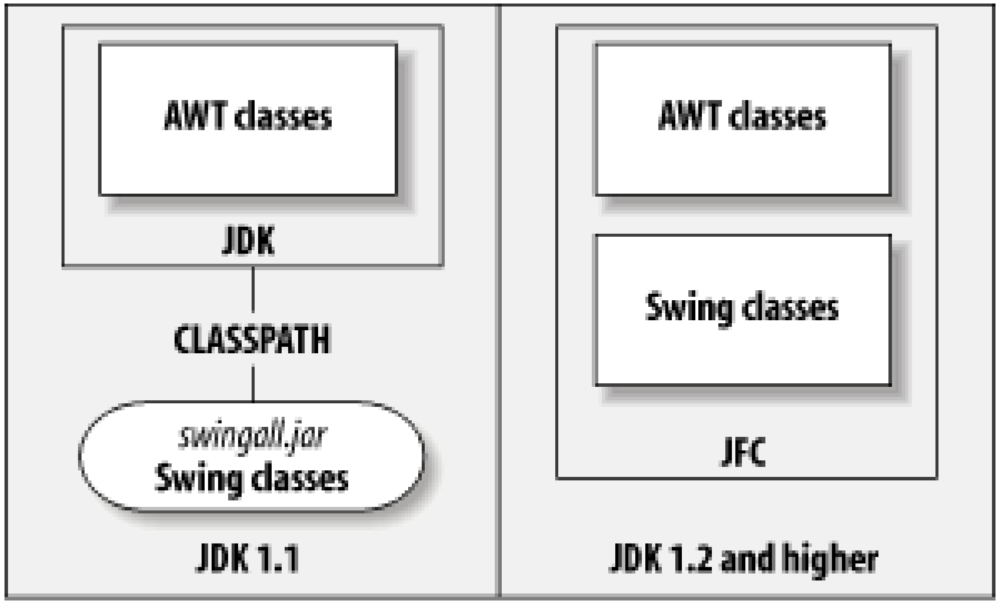
- The Swing API is only one of five libraries that make up the JFC. The JFC also consists of the Abstract Window Toolkit (AWT), the Accessibility API, the 2D API, and enhanced support for Drag and Drop capabilities
- *AWT*
  The Abstract Window Toolkit is the basic GUI toolkit shipped with all versions of the Java Development Kit. While Swing does not reuse any of the older AWT components, it does build on the lightweight component facilities introduced in AWT 1.1.
- *Accessibility*
  The accessibility package provides assistance to users who have trouble with traditional user interfaces. Accessibility tools can be used in conjunction with devices such as audible text readers or braille keyboards to allow direct access to the Swing components. All Swing components support accessibility.

What Are the Java Foundation Classes?

- *2D API*
  The 2D API contains classes for implementing various painting styles, complex shapes, fonts, and colors. This Java package is loosely based on APIs that were licensed from IBM's Taligent division. The 2D API classes are not part of Swing.
- *Drag and Drop*
  Drag and Drop (DnD) is one of the more common metaphors used in graphical interfaces today. The user is allowed to click and "hold" a GUI object, moving it to another window or frame in the desktop with predictable results. The DnD API allows users to implement droppable elements that transfer information between Java applications and native applications. Although DnD is not part of Swing, it is crucial to a commercial-quality application.

Is Swing a Replacement for AWT?

- No. Swing is actually built on top of the core AWT libraries. Because Swing does not contain any platform-specific (native) code, you can deploy the Swing distribution on any platform that implements the Java 1.1.5 or above virtual machine
- Swing depends extensively on the event-handling mechanism of AWT 1.1, although it does not define a comparatively large amount of events for itself
- SWING is a rethinking the AWT

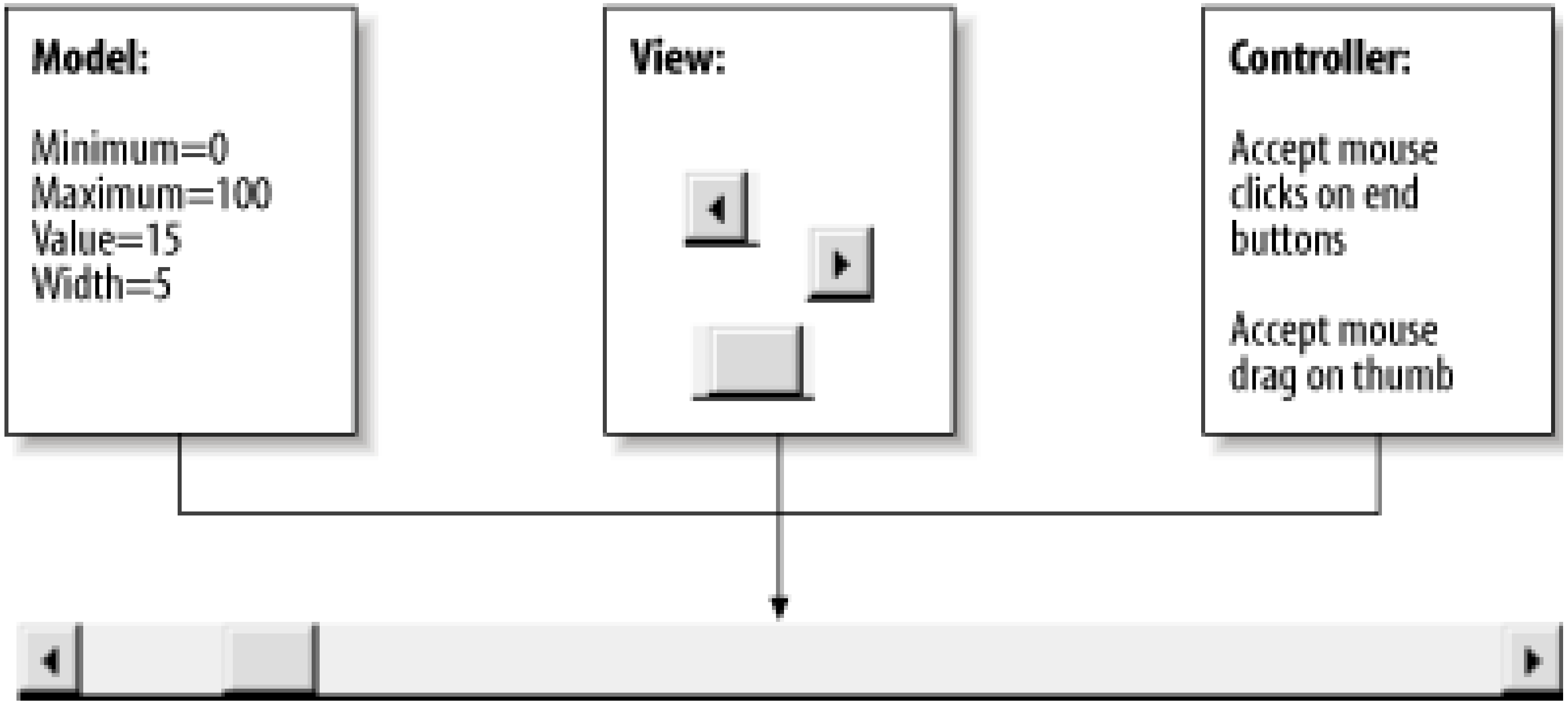


The Model-View-Controller Architecture

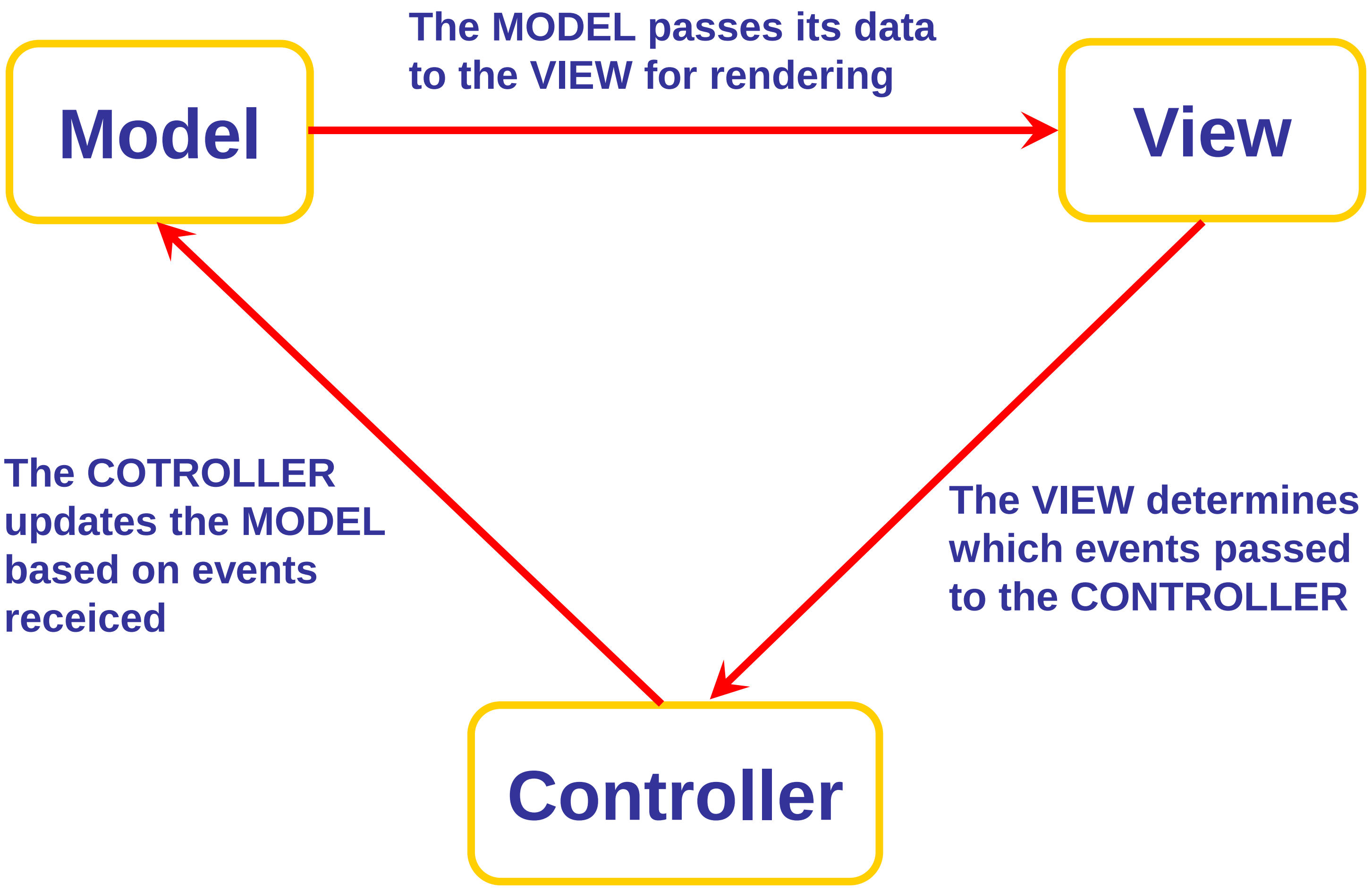
- Swing uses the *model-view-controller architecture* (MVC) as the fundamental design behind each of its components. Essentially, MVC breaks GUI components into three elements:

1. *Model*
   The model encompasses the state data for each component. For example, the model of a scrollbar component might contain information about the current position of its adjustable "thumb," its minimum and maximum values, and the thumb's width (relative to the range of values).
2. *View*
   The view refers to how you see the component on the screen. For a good example of how views can differ, look at an application window on two different GUI platforms.
3. *Controller*
   The controller is the portion of the user interface that dictates how the component interacts with events. Events come in many forms — e.g., a mouse click, gaining or losing focus, a keyboard event that triggers a specific menu command, or even a directive to repaint part of the screen. The controller decides how each component reacts to the event—if it reacts at all.

The three elements of a MVC architecture

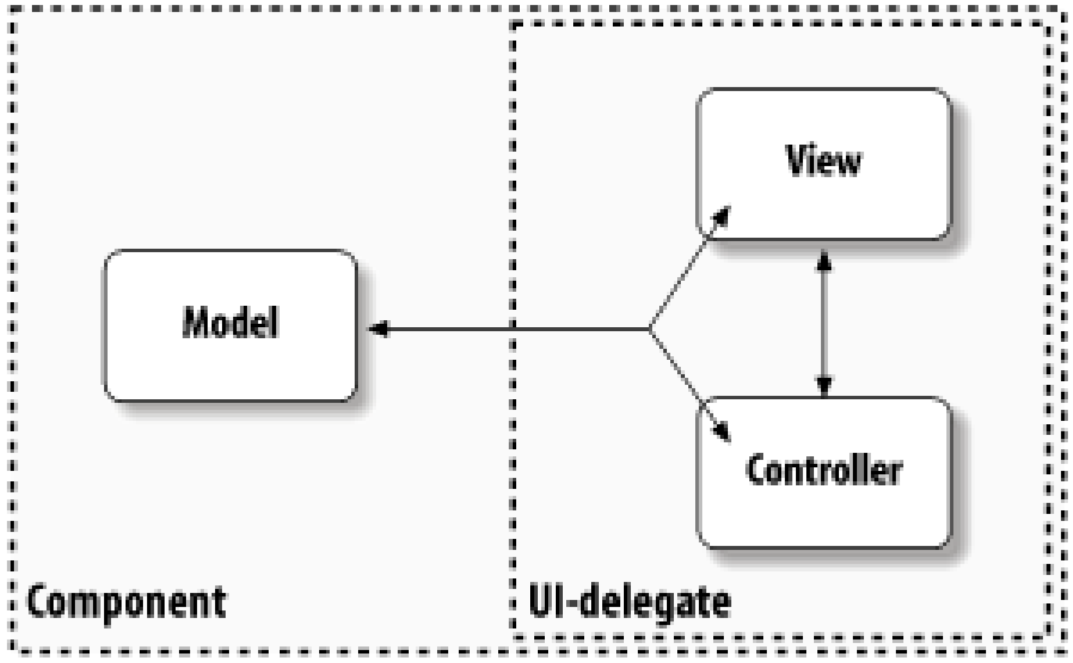

Model:
Minimum=0
Maximum=100
Value=15
Width=5
View:
Controller:
Accept mouse clicks on end buttons
Accept mouse drag on thumb

Communication through the MVC architecture


The MODEL passes its data to the VIEW for rendering
Model
View
The COTROLLER updates the MODEL based on events receiced
The VIEW determines which events passed to the CONTROLLER
Controller

MVC in Swing

- Swing actually uses a simplified variant of the MVC design called the *model-delegate*. This design combines the view and the controller object into a single element, the *UI delegate*, which draws the component to the screen and handles GUI events.
- Each Swing component contains a model and a UI delegate. The model is responsible for maintaining information about the component's state. The UI delegate is responsible for maintaining information about how to draw the component on the screen. In addition, the UI delegate (in conjunction with AWT) reacts to various events that propagate through the component



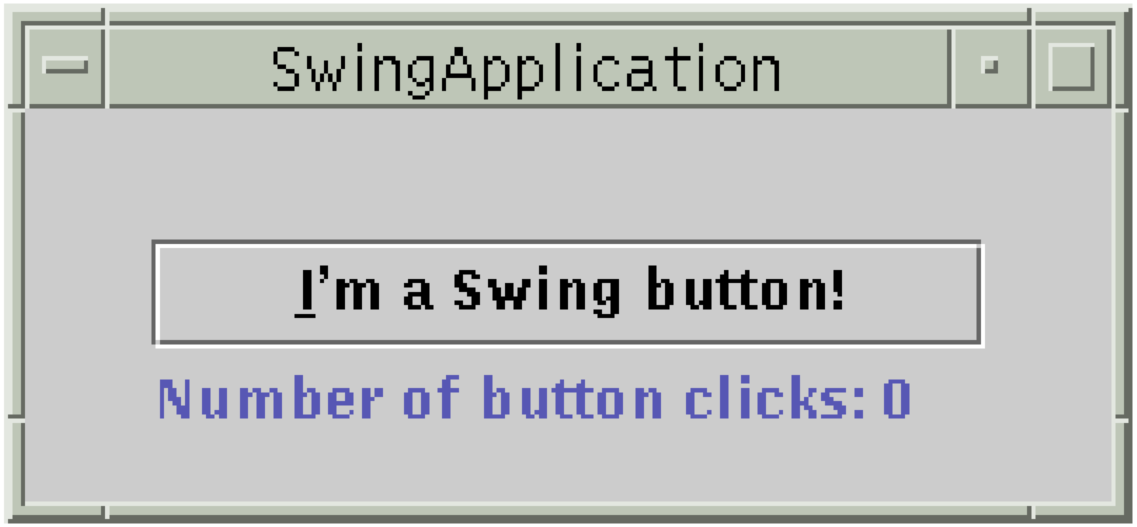
SWING OVERVIEW

- Swing Components and the Containment Hierarchy
  Swing provides many standard GUI components such as buttons, lists, menus, and text areas, which you combine to create your program's GUI. It also includes containers such as windows and tool bars.
- Layout Management
  Containers use *layout managers* to determine the size and position of the components they contain.
- Event Handling
  Event handling is how programs respond to external events, such as the user pressing a mouse button. Swing programs perform all their painting and event handling in the event-dispatching thread.

SWING OVERVIEW

- Painting
  Painting means drawing the component on-screen. Although it's easy to customize a component's painting, most programs don't do anything more complicated than customizing a component's border.
- More Swing Features and Concepts
  Swing offers many features, many of which rely on support provided by the JComponent class. Some of the interesting features this lesson hasn't discussed yet include support for icons, actions, Pluggable Look & Feel technology, assistive technologies, and separate models.

Swing Components



SwingApplication creates four commonly used Swing components:

1. a *frame*, or main window (**JFrame**)
2. a *panel*, sometimes called a *pane* (**JPanel**)
3. a button (**JButton**)
4. a label (**JLabel**)

Swing Components

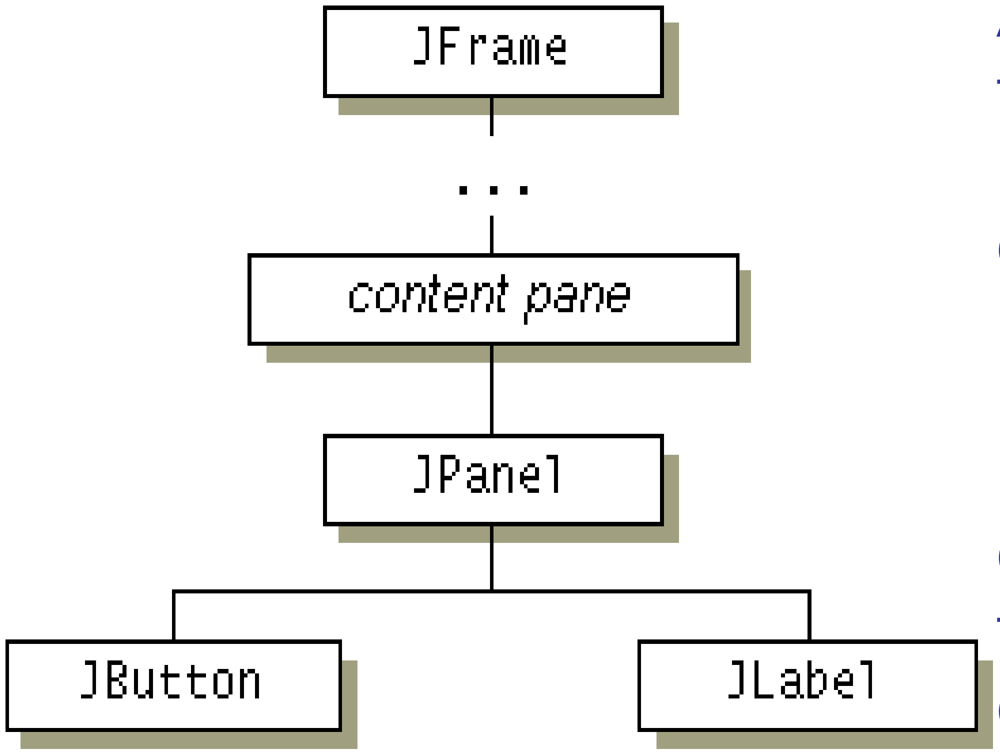
- The **frame** is a *top-level container*. It exists mainly to provide a place for other Swing components to paint themselves. The other commonly used top-level containers are dialogs (JDialog) and applets (JApplet)
- The **panel** is an intermediate container. Its only purpose is to simplify the positioning of the button and label. Other intermediate Swing containers, such as scroll panes (JScrollPane) and tabbed panes (JTabbedPane), typically play a more visible, interactive role in a program's GUI

Swing Components

- The button and label are *atomic components* -- components that exist not to hold random Swing components, but as self-sufficient entities that present bits of information to the user. Often, atomic components also get input from the user. The Swing API provides many atomic components, including Buttons (**JCheckBox**, **JRadioButton**), combo boxes (**JComboBox**), text fields (**JTextField**), and tables (**JTable**).

The Containment Hierarchy

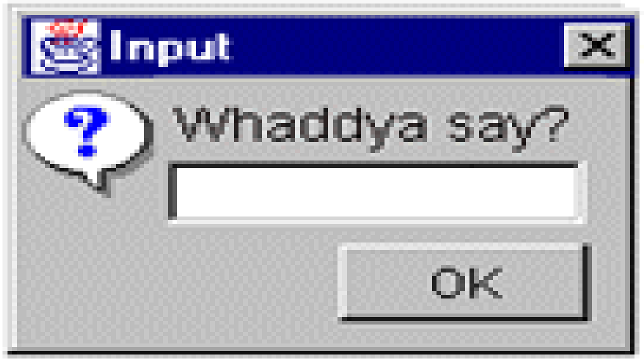
A diagram of the *containment hierarchy* shows each container created or used by the program, along with the components it contains.

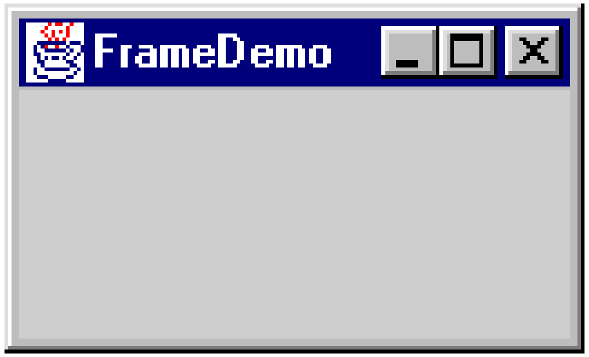


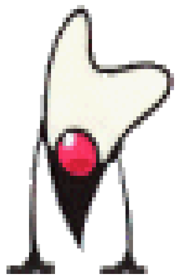
SwingApplication

As the figure shows, even the simplest Swing program has multiple levels in its containment hierarchy. The root of the containment hierarchy is always a top-level container. The top-level container provides a place for its descendent Swing components to paint themselves.

SWING COMPONENTS





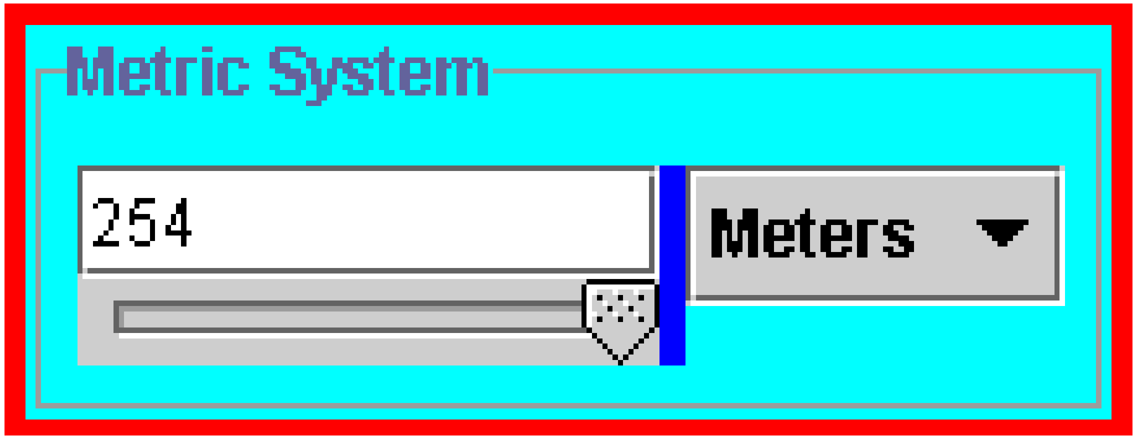


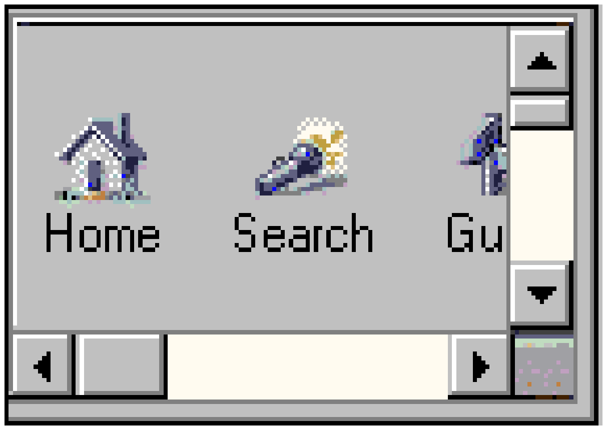
Dialog **Frame** **Applet**

Top-Level Containers

The components at the top of any Swing containment hierarchy.

SWING COMPONENTS



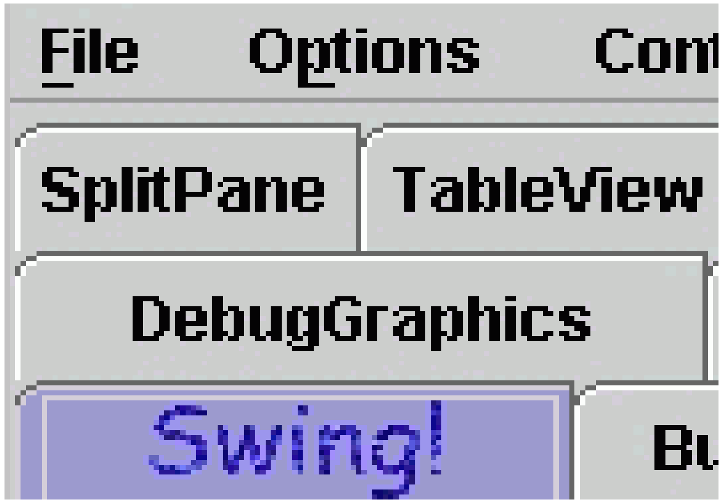


Panel **Scroll pane**

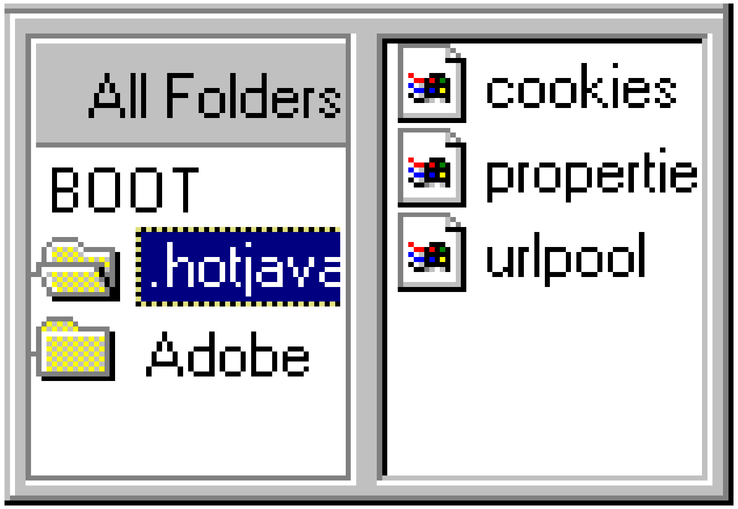
General-Purpose Containers

Intermediate containers that can be used under many different circumstances.

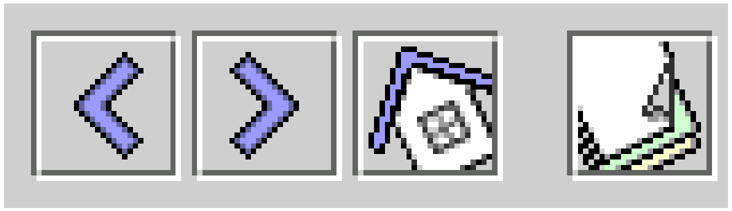
SWING COMPONENTS



Tabbed pane



Split pane

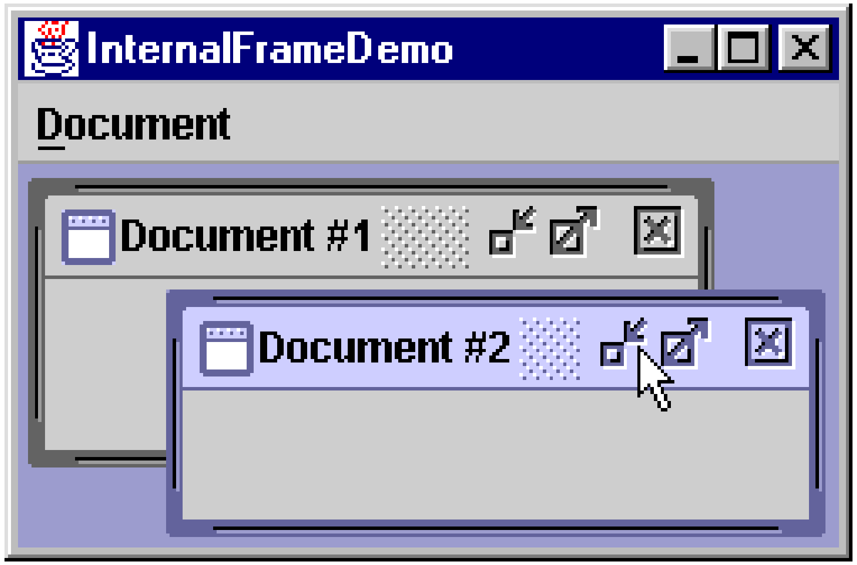


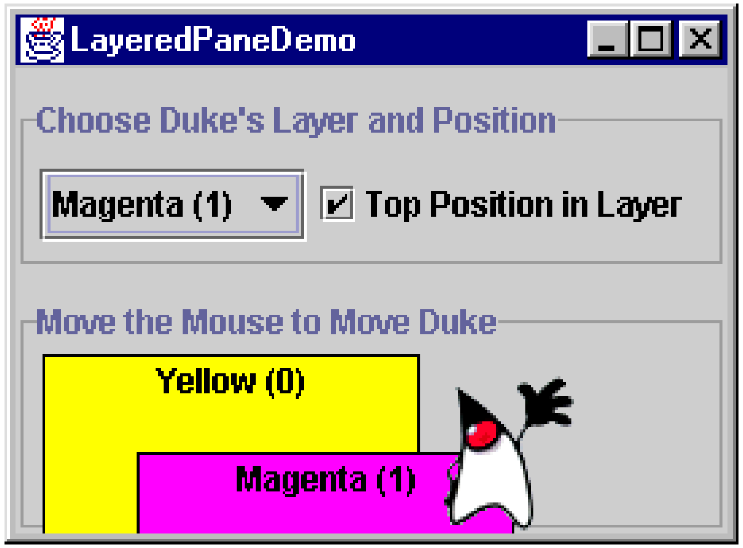
Tool bar

General-Purpose Containers

Intermediate containers that can be used under many different circumstances.

SWING COMPONENTS



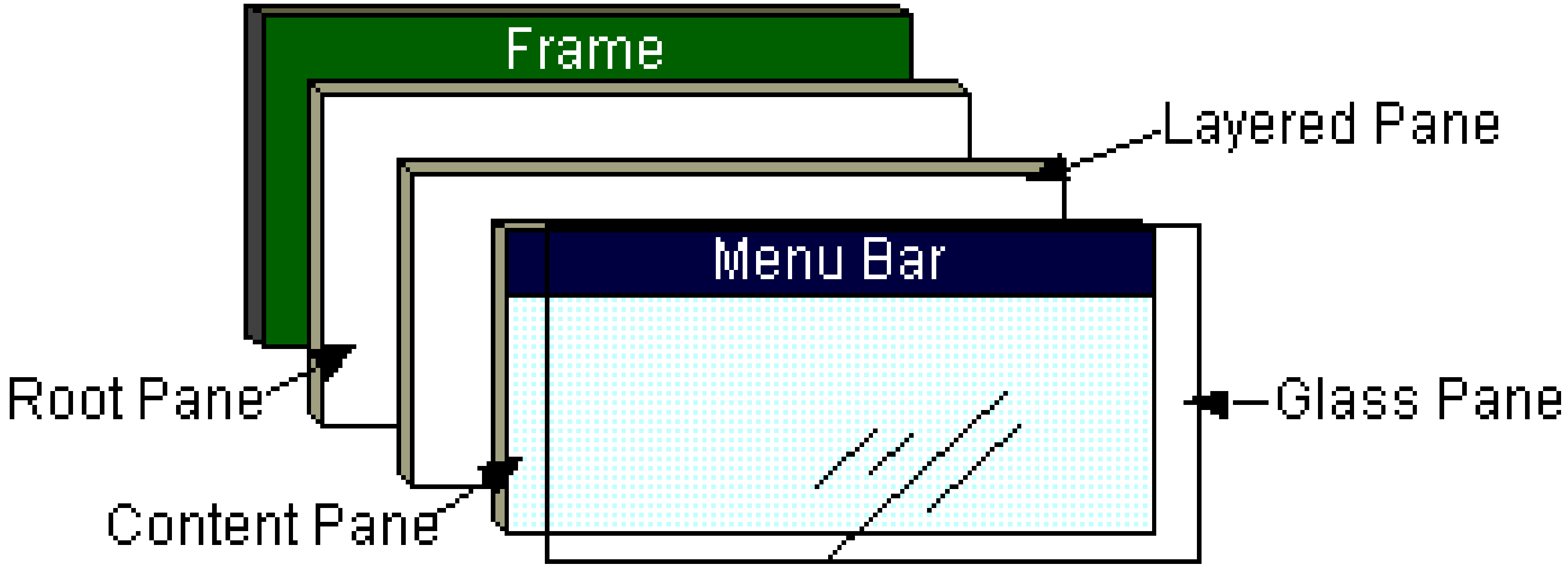


Internal Frame **Layered pane**

Special-Purpose Containers

Intermediate containers that play specific roles in the UI.

SWING COMPONENTS

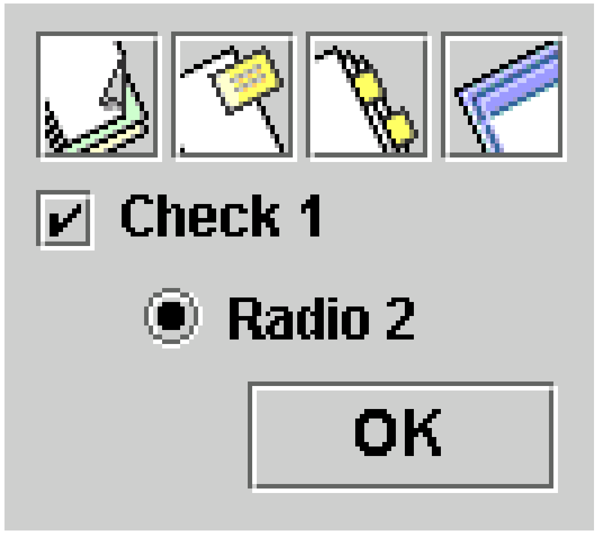


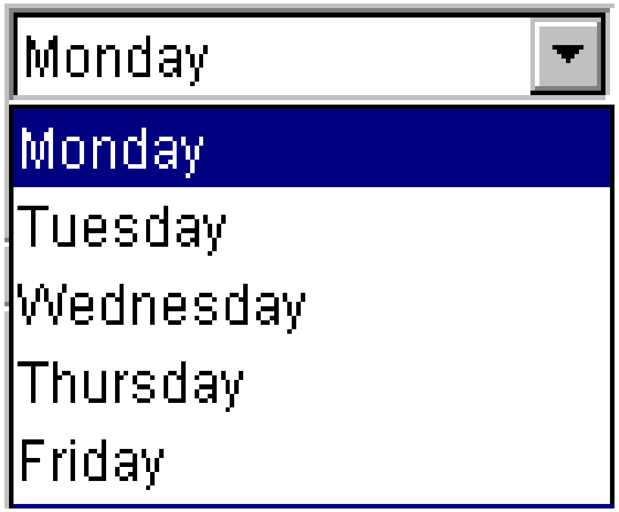
Root Pane & Glass pane

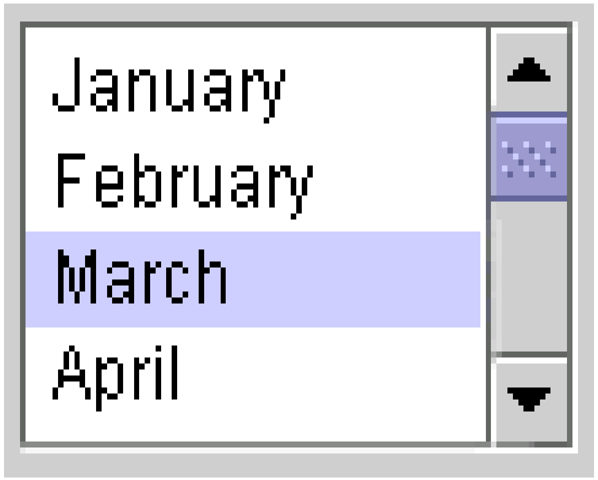
Special-Purpose Containers

Intermediate containers that play specific roles in the UI.

SWING COMPONENTS





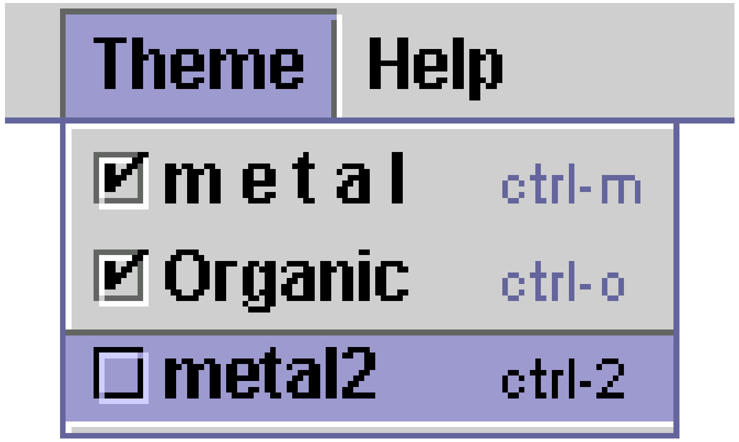


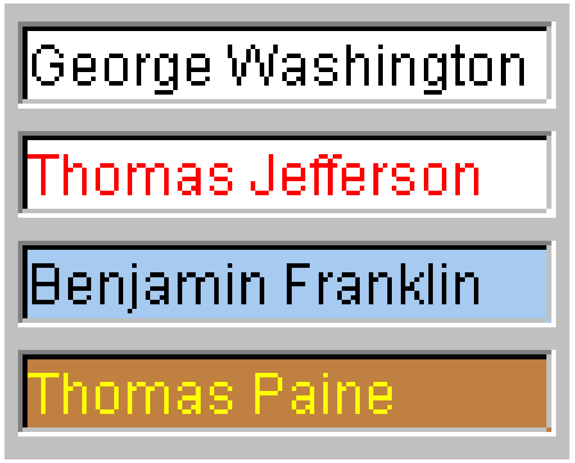
Buttons **Combo Box** **List**

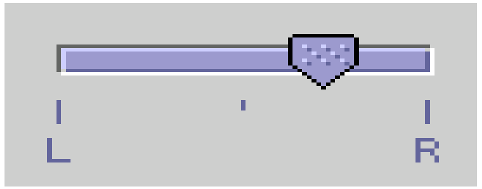
Basic Controls

Atomic components that exist primarily to get input from the user; they generally also show simple state. .

SWING COMPONENTS





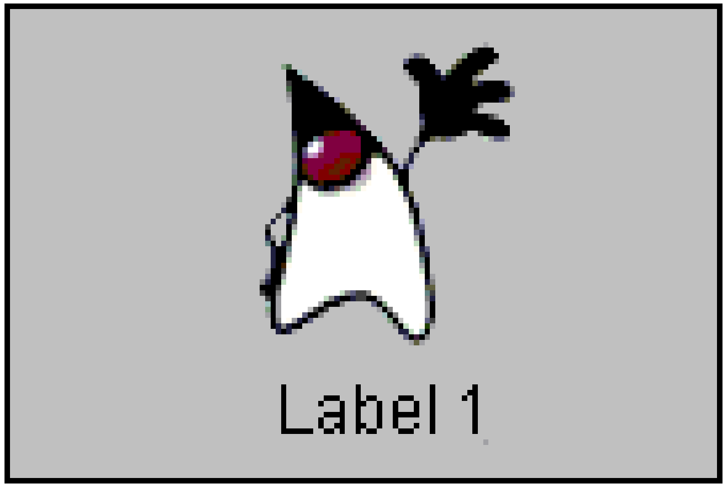


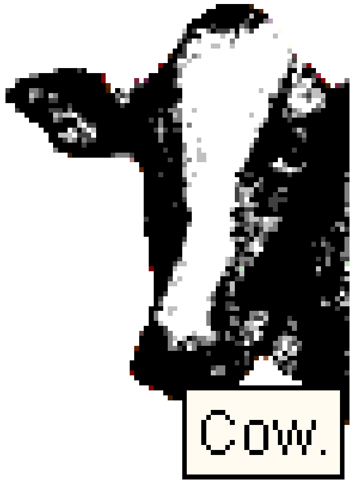
Menu **Text fields** **Slider**

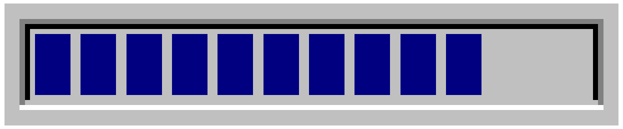
Basic Controls

Atomic components that exist primarily to get input from the user; they generally also show simple state. .

SWING COMPONENTS





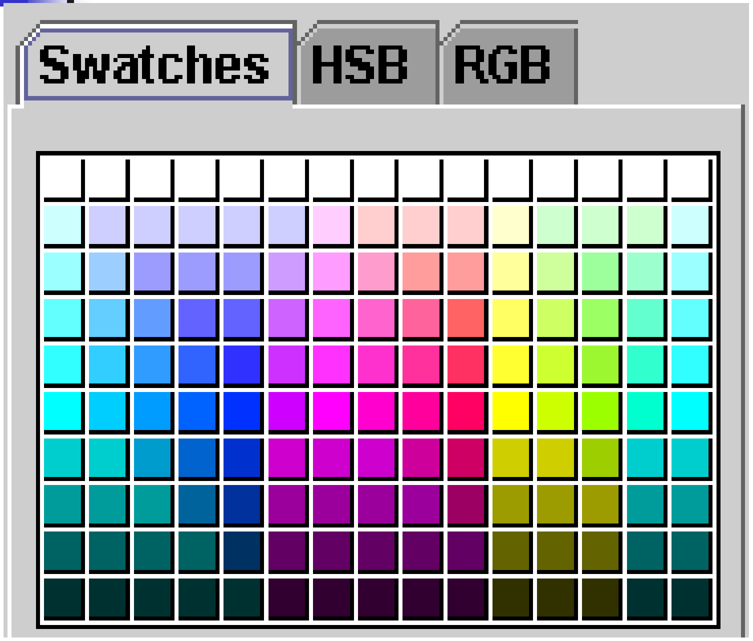


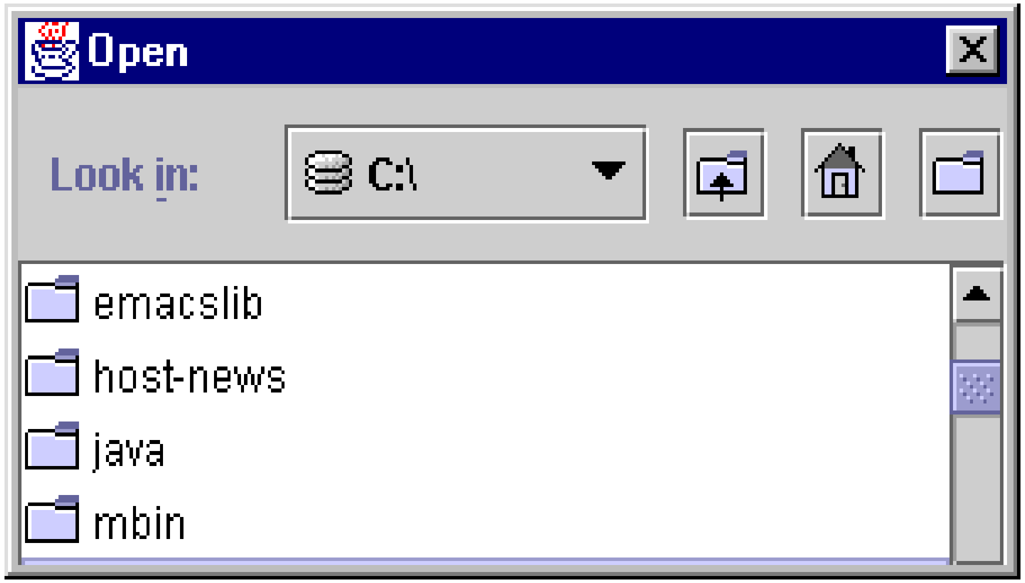
Label **Tool tip** **Progress Bar**

Uneditable Information Displays

Atomic components that exist solely to give the user information.

SWING COMPONENTS



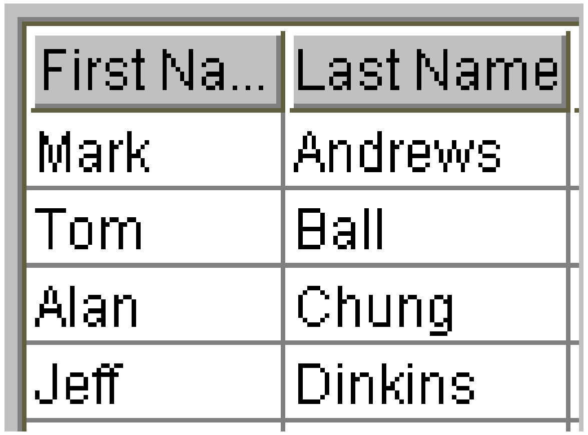


Color Chooser **File Chooser**

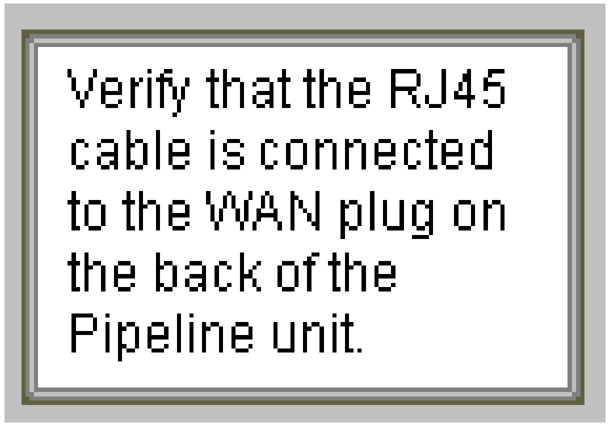
Editable Displays of Formatted Information

Atomic components that display highly formatted information that (if you choose) can be edited by the user.

SWING COMPONENTS



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



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

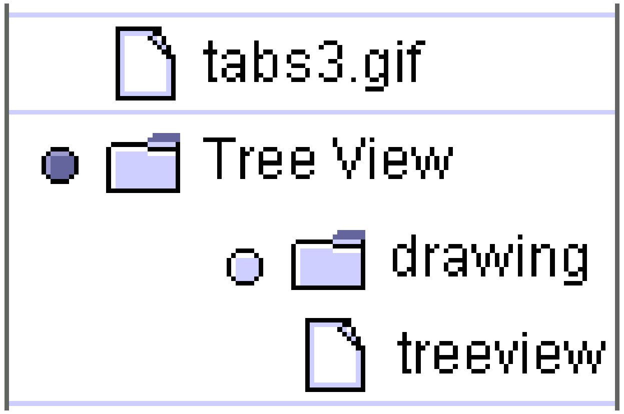


Table **Text** **Tree**

Editable Displays of Formatted Information

Atomic components that display highly formatted information that (if you choose) can be edited by the user.

Layout Management

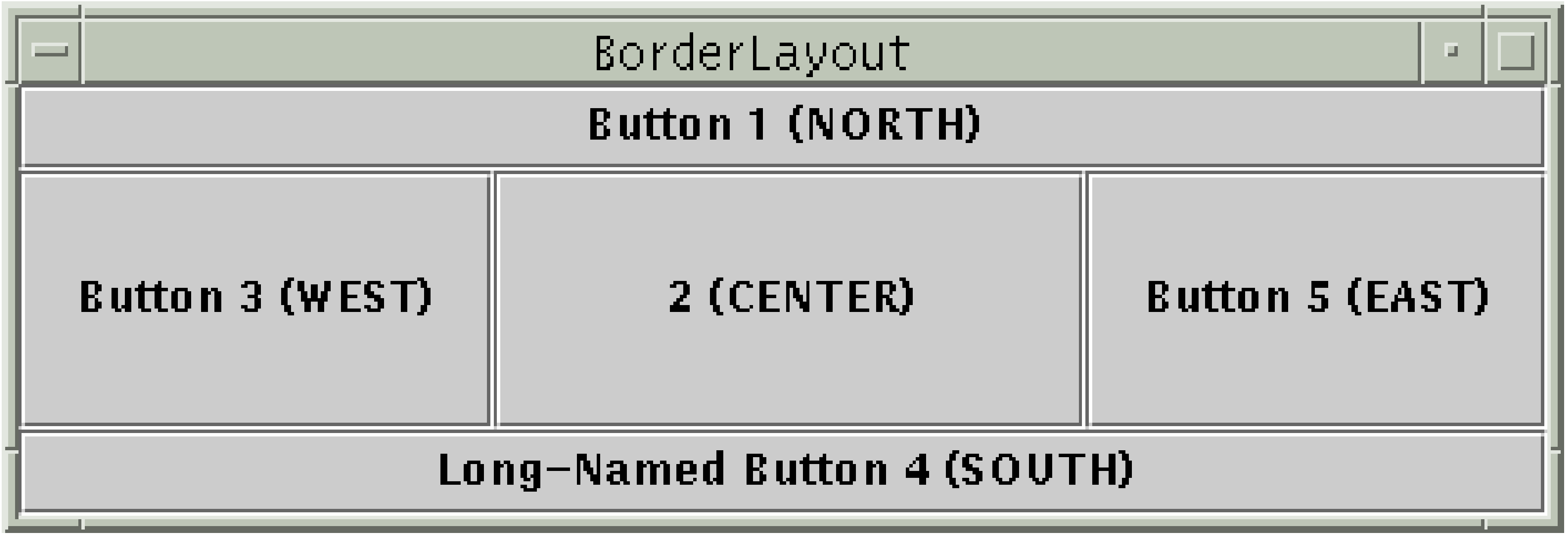
- *Layout management* is the process of determining the size and position of components. By default, each container has a *layout manager* -- an object that performs layout management for the components within the container. Components can provide size and alignment hints to layout managers, but layout managers have the final say on the size and position of those components.
- The Java platform supplies five commonly used layout managers: BorderLayout, BoxLayout, FlowLayout, GridBagLayout, and GridLayout. These layout managers are designed for displaying multiple components at once, and are shown in the preceding figure. A sixth provided class, CardLayout, is a special-purpose layout manager used in combination with other layout managers.

Layout Management

- Whenever you use the add method to put a component in a container, you must take the container's layout manager into account.
- Some layout managers, such as BorderLayout, require you to specify the component's relative position in the container, using an additional argument with the add method. Occasionally, a layout manager such as GridBagLayout requires elaborate setup procedures.

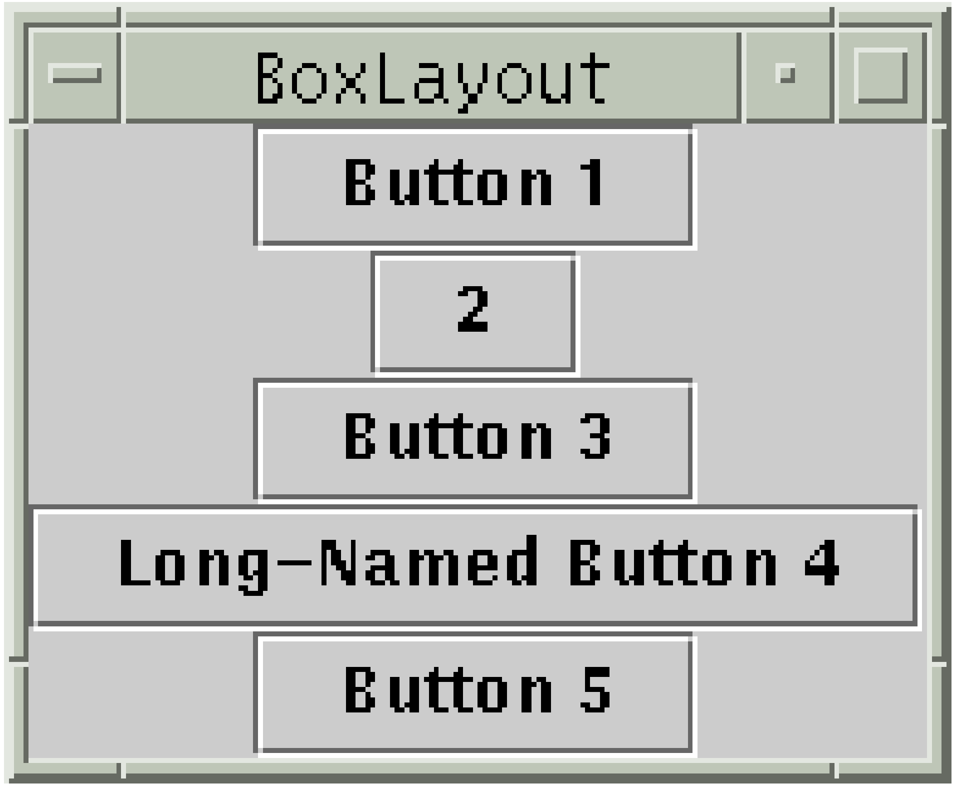
Layout Management – BorderLayout

- A BorderLayout has five areas available to hold components: north, south, east, west, and center. All extra space is placed in the center area.



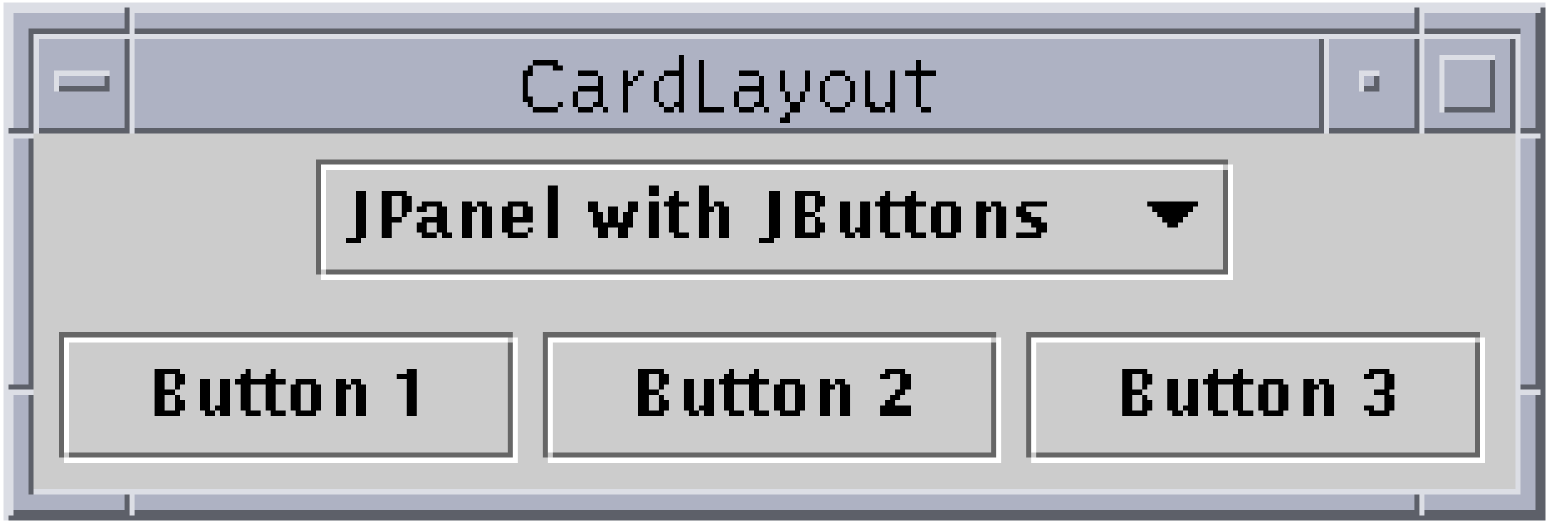
Layout Management – BoxLayout

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



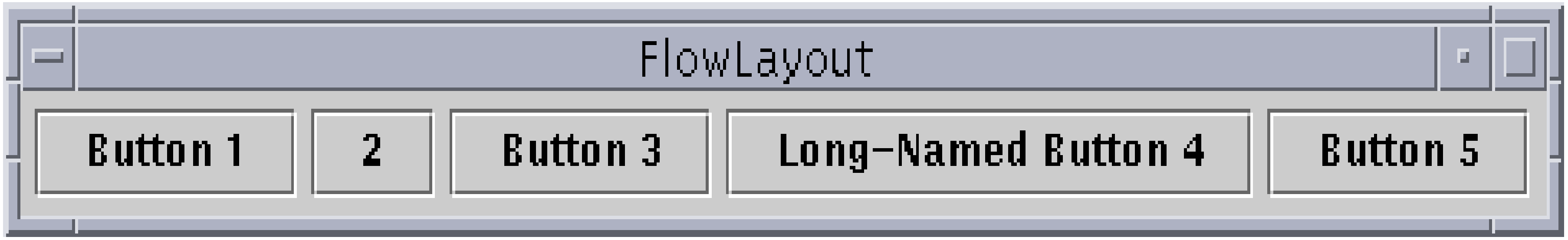
Layout Management – CardLayout

- 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. Tabbed panes are intermediate Swing containers that provide similar functionality.



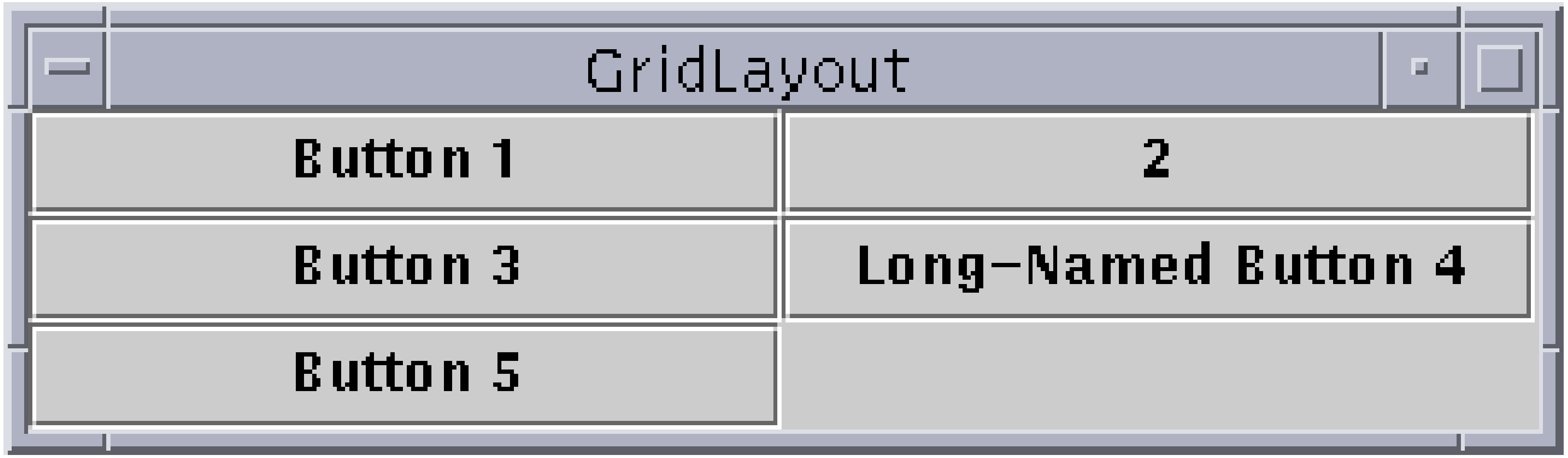
Layout Management – FlowLayout

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



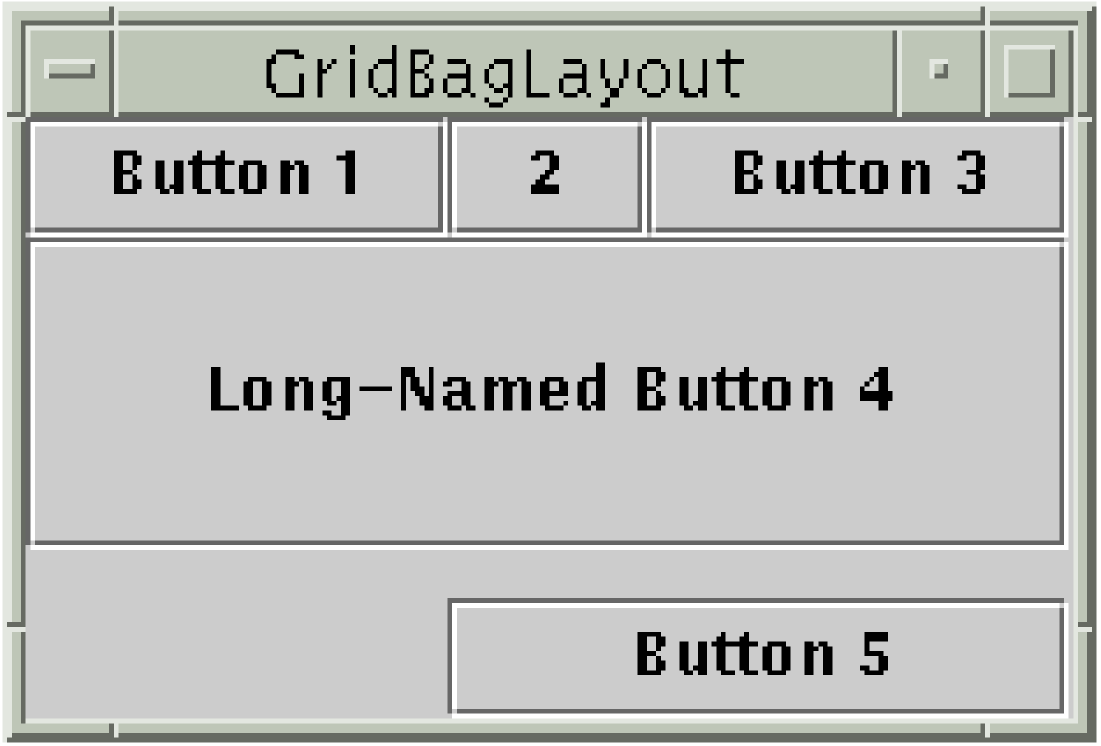
Layout Management – GridLayout

- GridLayout simply makes a bunch of components equal in size and displays them in the requested number of rows and columns. Here's an applet that uses a GridLayout to control the display of five buttons:



Layout Management – 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.



Event Handling

- Every time the user types a character or pushes a mouse button, an event occurs. Any object can be notified of the event. All it has to do is implement the appropriate interface and be registered as an *event listener* on the appropriate *event source*.

Act that results in the event	**Listener type**
1. User clicks a button, presses Return while typing in a text field, or chooses a menu item	**ActionListener**
2. User closes a frame (main window)	**WindowListener**
3. User presses a mouse button while the cursor is over a component	**MouseListener**
4. User moves the mouse over a component	**MouseMotionListener**

Event Handling - Event Handler

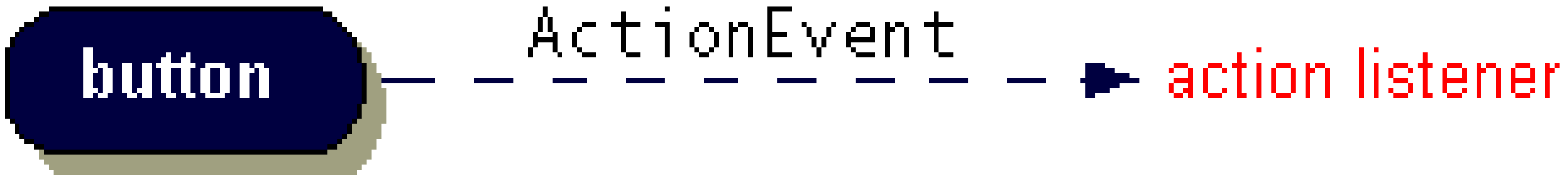
Every event handler requires three bits of code:

1. In the declaration for the event handler class, code that 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** {
2. Code that registers an instance of the event handler class as a listener upon one or more components. For example:
   someComponent.**addActionListener**(instanceOfMyClass);
3. Code that implements the methods in the listener interface. For example:
   public void **actionPerformed**(ActionEvent e) {
   ...//code that reacts to the action...
   }

Event Handling - Event Handler

To example:

- To detect when the user clicks an on-screen button (or does the keyboard equivalent), a program must have an object that implements the ActionListener interface. The program must register this object as an action listener on the button (the event source), using the addActionListener method.
- When the user clicks the on-screen button, the button fires an action event. This results in the invocation of the action listener's actionPerformed method (the only method in the ActionListener interface). The single argument to the method is an ActionEvent object that gives information about the event and its source.



Caption: When the user clicks a button, the button's action listeners are notified.

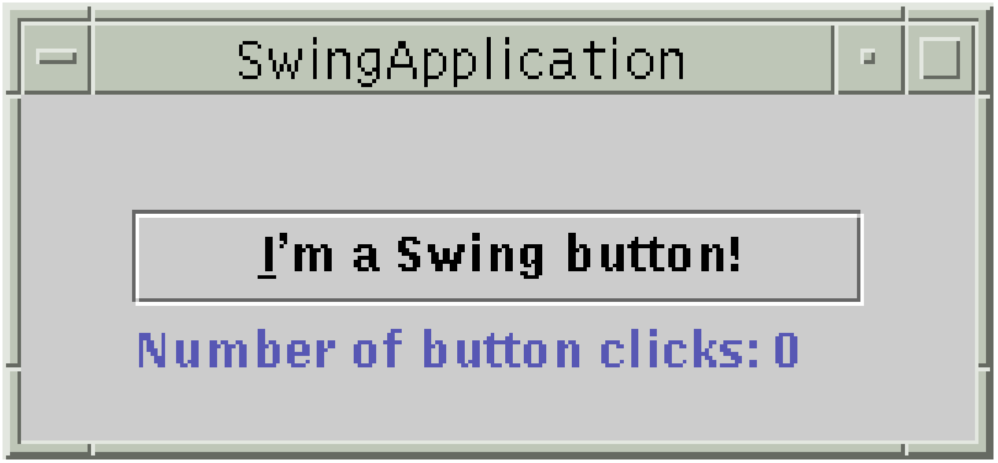
Event Handling - Event Handler

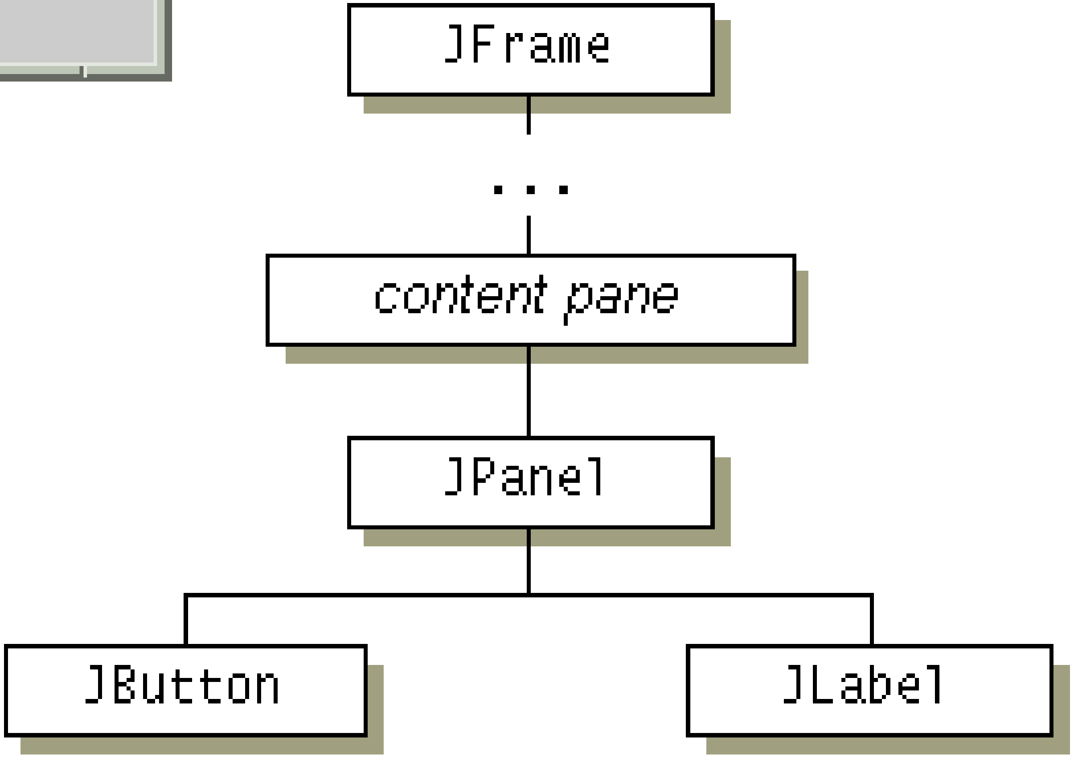
- Event handlers can be instances of any class.
- Often, an event handler that has only a few lines of code is implemented using an *anonymous inner class* -- an unnamed class defined inside of another class.
- Anonymous inner classes can be somewhat confusing at first, but once you're used to them they make code clearer by keeping the implementation of an event handler close to where the event handler is registered.

Painting

- When a Swing GUI needs to paint itself -- whether for the first time, in response to becoming unhidden, or because it needs to reflect a change in the program's state -- it starts with the highest component that needs to be repainted and works its way down the containment hierarchy.
- Swing components generally repaint themselves whenever necessary. When you invoke the ***setText*** method on a component, for example, the component should automatically repaint itself and, if appropriate, resize itself. If it doesn't, it's a bug. The workaround is to invoke the ***repaint*** method on the component to request that the component be scheduled for painting. If the component's size or position needs to change but doesn't do so automatically, you should invoke ***revalidate*** upon the component before invoking ***repaint***.

Painting : An Example of Painting



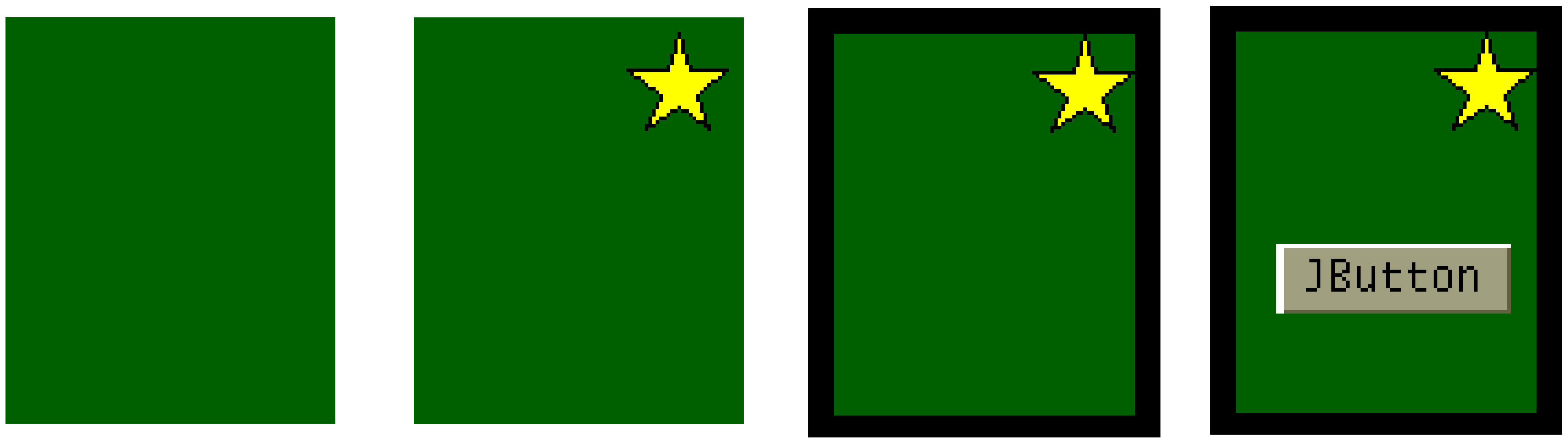


Painting : An Example of Painting

1. The top-level container, JFrame, paints itself.
2. The content pane first paints its background, which is a solid gray rectangle. It then tells the JPanel to paint itself.
3. The JPanel first paints its background, a solid gray rectangle. Next, it paints its border. The border is an EmptyBorder, which has no effect except for increasing the JPanel's size by reserving some space at the edge of the panel. Finally, the panel asks its children to paint themselves.
4. To paint itself, the JButton paints its background rectangle, if necessary, and then paints the text that it contains. If the button has the keyboard focus, meaning that any typing goes directly to the button for processing, then the button does some look-and-feel-specific painting to make clear that it has the focus.
5. To paint itself, the JLabel paints its text.

Painting

1. Background (if opaque)
2. custom painting (if any)
3. border (if any)
4. children (if any)



The following figure illustrates the order in which each component that inherits from *JComponent* paints itself.

More Swing Features and Concepts

- Features that JComponent provides
- Icons
- Actions
- Pluggable Look & Feel support
- Support for assistive technologies

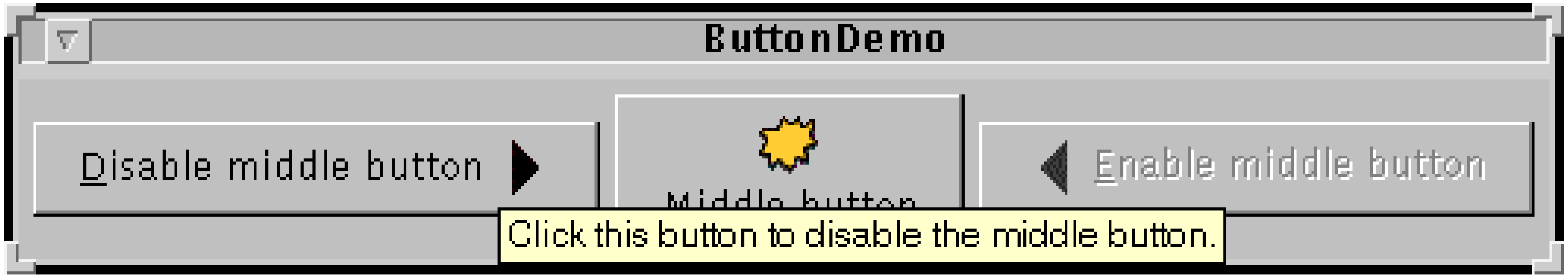
Features that JComponent Provides

- Except for the top-level containers, all components that begin with J inherit from the JComponent class. They get many features from JComponent, such as the ability to have borders, tool tips, and a configurable look and feel. They also inherit many convenient methods.

JComponent : Tool tips

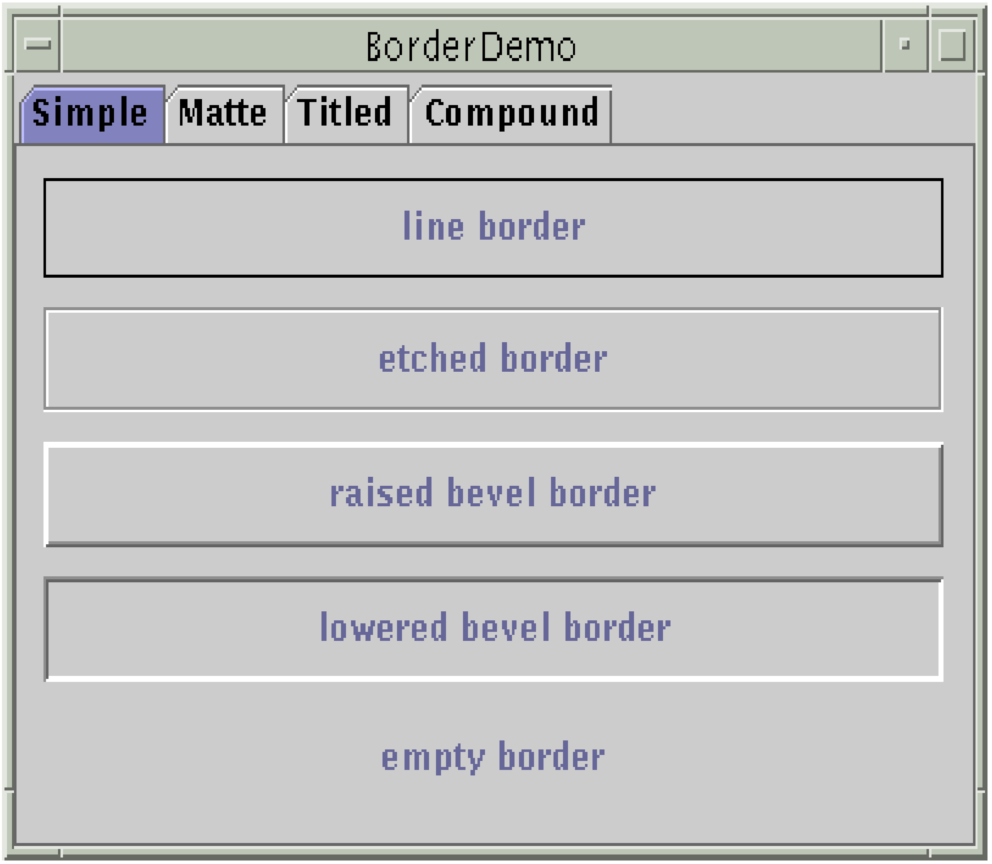
- By specifying a string with the **setToolTipText** method, you can provide help to users of a component. When the cursor pauses over the component, the specified string is displayed in a small window that appears near the component.
- You just use the **setToolTipText** method to set up a tool tip for the component. For example:

```
b1.setToolTipText("Click this button to disable the middle button.");
b2.setToolTipText("This middle button does nothing.");
b3.setToolTipText("Click this button to enable the middle button.");
```



JComponent : Borders

- The setBorder method allows you to specify the border that a component displays around its edges.
- To put a border around a JComponent, you use its setBorder method. You can use the BorderFactory class to create most of the borders that Swing provides. Here is an example of code that creates a bordered container



```
JPanel pane = new JPanel();
pane.setBorder(BorderFactory.createLineBorder(Color.black));
```

JComponent : Keyboard-generated actions

- Using the registerKeyboardAction method, you can enable the user to use the keyboard, instead of the mouse, to operate the GUI.
- The combination of character and modifier keys that the user must press to start an action is represented by a *KeyStroke* object. The resulting action event must be handled by an *action listener*. Each keyboard action works under exactly one of three conditions: only when the actual component has the focus, only when the component or one of its containers has the focus, or any time that anything in the component's window has the focus.

JComponent : pluggable look and feel

- Behind the scenes, each JComponent object has a corresponding ComponentUI object that performs all the drawing, event handling, size determination, and so on for that JComponent. Exactly which ComponentUI object is used depends on the current look and feel, which you can set using the UIManager.setLookAndFeel method.

1. Setting the Look and Feel
2. How the UI Manager Chooses the Look and Feel
3. Changing the Look and Feel After Startup

Setting the Look and Feel

- To programmatically specify a look and feel, use the UIManager.setLookAndFeel method. For example, the bold code in the following snippet makes the program use the Java Look & Feel: public static void main(String[] args) {

```
   try { UIManager.setLookAndFeel(
         UIManager.getCrossPlatformLookAndFeelClassName()); }
   catch (Exception e) { }
   new SwingApplication(); //Create and show the GUI.
 }
```

- The argument to setLookAndFeel is the fully qualified name of the appropriate subclass of LookAndFeel. To specify the Java Look & Feel, we used the getCrossPlatformLookAndFeelClassName method. If you want to specify the native look and feel for whatever platform the user runs the program on, use getSystemLookAndFeelClassName
- Windows Look & Feel:

```
   UIManager.setLookAndFeel(
"com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
```

How the UI Manager Chooses the Look and Feel

1. If the program sets the look and feel before any components are created, the UI manager tries to create an instance of the specified look-and-feel class. If successful, all components use that look and feel.
2. If the program hasn't successfully specified a look and feel, then before the first component's UI is created, the UI manager tests whether the user specified a look and feel in a file named swing.properties. It looks for the file in the lib directory of the Java release. For example, if you're using the Java interpreter in *javaHomeDirectory*\bin, then the swing.properties file (if it exists) is in *javaHomeDirectory*\lib. If the user specified a look and feel, then again the UI manager tries to instantiate the specified class. Here is an example of the contents of a swing.properties file:
   ```
   # Swing properties
   swing.defaultlaf=com.sun.java.swing.plaf.motif.MotifLookAndFeel
   ```
3. If neither the program nor the user successfully specifies a look and feel, then the program uses the Java Look & Feel.

Changing the Look and Feel After Startup

You can change the look and feel with setLookAndFeel even after the program's GUI is visible. To make existing components reflect the new look and feel, invoke the SwingUtilities updateComponentTreeUI method once per top-level container. Then you might wish to resize each top-level container to reflect the new sizes of its contained components. For example:

```
UIManager.setLookAndFeel(lnfName);
SwingUtilities.updateComponentTreeUI(frame);
frame.pack();
```