

Part II

Using Swing components

*P*art 2 of this book contains the details required to use each of the key Swing components. You will start by looking at Swing panels and panes (the most basic components) and then build on this knowledge to include buttons, menus, and more. Finally, we will examine more complex components, such as tables and hierarchical trees. When appropriate, we will also look at some of the more subtle aspects of Swing. The concepts you will learn in this part of the book can be applied to your own applications for either visual effect or performance enhancement.

3

Frames and panels

In this chapter

- Creating JFrames and JApplets
- Swing panels
- Additional Layout Manager provided by JFC

In this chapter, you will begin to learn about some of the common components built into Swing. If you are familiar with AWT, then you know that you require either an instance of `Frame` or `Applet` as a starting point, and from there you can add additional panels and components to complete the code.

With Swing, all user interface component class names begin with the letter `J`, and, where possible, the name is the same as the AWT class it replaces. So, the basic starting point for a Swing-based program is either `JFrame` or `JApplet`. We will examine both of these classes, followed by a discussion of `JPanel` (a replacement for the AWT `Panel` class), and the new layout managers supplied in the Swing class library.

3.1 JFrame

Simply put, if you know how to use the AWT `Frame` class, then you can use `JFrame`. `JFrame` is an extended version of `Frame` that adds support for special painting behavior, and for child components that are managed by a `JLayeredPane` (you will learn a little more about this class later in the chapter). Additionally, `JFrame` has support for Swing `MenuBar`s, allowing them to be placed not only at the top of the window, but also anywhere within the frame (though, if the menu is attached to the frame using the `setMenuBar()` method, it will always be located at the top of the window).

All objects associated with a `JFrame` are managed by its only child, an instance of `JRootPane`. `JRootPane` is a simple container for all other panes for the `JFrame` instance. The following hierarchy shows the nesting of objects within a `JFrame` instance:

```
JFrame
  JRootPane
    glassPane
    layeredPane
      [menuBar]
      contentPane
```

3.1.1 A JFrame application

As with the AWT `Frame` class, you use an instance of `JFrame` to build the primary window of applications rather than applets. Of course, a `JFrame` can also be used to create secondary windows to an application or an applet. The code below creates a very simple application class using `JFrame`.

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

class TestFrame
    extends JFrame
{
    public TestFrame()
    {
        setTitle( "Test Application" );
        setSize( 100, 100 );

        Panel topPanel = new Panel();
        topPanel.setLayout( new BorderLayout() );
        getContentPane().add( topPanel );

        // Create a label to look at
        Label labelHello = new Label( "Hello World!" );
        topPanel.add( labelHello, BorderLayout.NORTH );
    }

    // Main program started
    public static void main( String args[] )
    {
        // Create an instance of the Test application
        TestFrame mainFrame = new TestFrame();
        mainFrame.setVisible( true );
    }
}
```

Listing 3.1 “Hello World!” using JFrame

Listing 3.1 creates a very simple Java application using JFrame. Although Swing provides its own panel and label classes, they have been intentionally left out of this example. Instead, the AWT equivalents have been used. The result of this application is shown in figure 3.1.



Figure 3.1
The “Hello World!” application



It is possible to mix AWT components and Swing components within a Swing-based application; however, due to Z-order limitations, you may experience some annoyances during screen repaints. AWT components appear to get quickly painted in the upper-left corner of the frame before being correctly positioned, resulting in unexpected flicker.

When possible avoid using AWT components in Swing applications. Use the Swing equivalents instead.

The example in listing 3.1 appears pretty much as expected, containing code as it would exist using AWT. There is one significant difference, however. The line:

```
getContentPane().add( topPanel );
```

appears a bit odd, and, in particular, the call to `getContentPane()`. `JFrame` exhibits a slight incompatibility when compared to the AWT `Frame` class because it contains only a single child (which is an instance of `JRootPane`). In order to add any other components to the `JFrame` instance, they must be added to the root pane. This is unlike the AWT `Frame` class which creates an instance of a `Panel` automatically, allowing components to be added directly to the frame instance. To access the root pane, `JFrame` provides a method called `getContentPane()`.



It is impossible to add components directly to a `JFrame` using the following syntax:

```
frame.add( component );
```

Instead you must always use this notation:

```
frame.getContentPane().add( component );
```

Failure to add components using the `getContentPane()` mechanism will result in the generation of an exception. The best solution to this problem is to create a panel, add it to the content pane, then add all components to the new panel.

The `JFrame` class offers some other interesting features. In addition to the content pane, `JFrame` also provides two other panes: a layered pane and a glass pane. We will examine `JLayeredPane` in more detail later in this chapter.

The glass pane allows you to display components in front of the existing JFrame instance, which can be useful in some applications. For example, suppose you want to create a network application that allows users on different computers to draw on a common “white board.” You can display the local user’s drawing on the content pane of the application, while on the glass pane, you could display the remote user’s mouse pointer and any drawing he or she performs.

Another practical use for the glass pane is in game development. The background of your game can be shown in the content pane, and any animated items can be drawn on the glass pane. Building your game in this way greatly simplifies the redrawing you must do when an image moves on the screen.

3.1.2 JFrame hierarchy

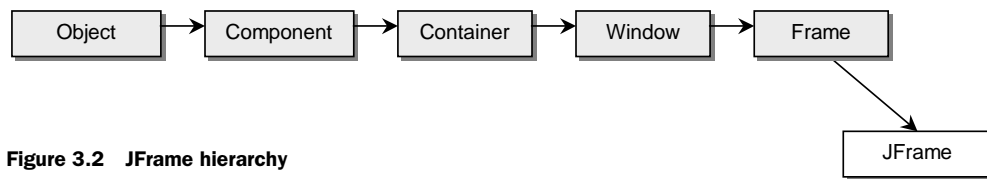


Figure 3.2 JFrame hierarchy

3.1.3 JFrame variables

```
protected JRootPane rootPane
```

This variable contains an instance of the root pane associated with the frame. Note that all components owned by the frame must be added to the root pane, which is unlike the technique familiar to AWT users.

```
protected boolean canAdd
```

This variable controls whether or not components can be added directly to the frame. Under most circumstances, this variable will contains a false value.

```
protected AccessibleContext accessibleContext
```

The `accessibleContext` variable contains an instance of the context used by the Assistive Technology mechanism of the component. In most situations, this variable will be unused.

3.1.4 JFrame constructors

```
JFrame()
```

This constructor creates a new JFrame instance that is initially invisible.

```
JFrame( String title )
```

This constructor creates a new JFrame instance that is initially invisible. The title of the frame (shown in the title bar of the frame) is assigned the text specified by the title parameter.

3.1.5 JFrame significant method groupings

```
public void setJMenuBar( JMenuBar menu );
```

Like its AWT cousin, JFrame will support a single application menu bar. We will see how menus are created in chapter 7; however, once created, they can be added with the `setMenuBar()` method.

```
public void setDefaultCloseOperation(int operation);  
public int getDefaultCloseOperation();
```

A unique feature of JFrame is the ability to assign how the window's close operation works. JFrame implements a method to set and get the value of the default close operation.

```
protected JRootPane createRootPane();  
protected void setRootPane(JRootPane root);  
public JRootPane getRootPane();  
public Container getContentPane();  
public void setLayeredPane(JLayeredPane layered);  
public JLayeredPane getLayeredPane();  
public void setGlassPane(Component glass);  
public Component getGlassPane();
```

A JFrame instance can contain three different panes: a layered pane, a glass pane, and a root pane. We've already talked about the root pane. The layered pane is the invisible pane on top of the frame's root pane. It can be accessed to display dynamic items (such as cursors) above the frame contents.

3.2 JApplet

JApplet is the Swing equivalent of the AWT Applet class. Much like JFrame, JApplet has extensions to allow for interposing input and special painting behavior, and also supports child components that are managed by a root pane (recall the root

pane description in the previous section) . Unlike the AWT applet class, JApplet permits the addition of menu bars and toolbars. This is a feature sorely lacking from AWT in previous versions of Java.

3.2.1 A JApplet sample applet

The code shown in listing 3.2 performs the same task as the example for JFrame shown previously; however, this code executes within the AppletViewer or a browser. Note that the restrictions of JFrame with regard to `getContentPane()` apply equally to JApplet. The output produced by executing the code in listing 3.2 is shown in figure 3.3.

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

public class TestApplet
    extends JApplet
{
    public TestApplet()
    {
    }

    public void init()
    {
        setSize( 100, 100 );

        JPanel topPanel = new JPanel();
        topPanel.setLayout( new BorderLayout() );
        getContentPane().add( topPanel );
        JLabel labelHello = new JLabel( "Hello World!" );
        topPanel.add( labelHello, BorderLayout.NORTH );
    }
}
```

Listing 3.2 “Hello World!” using JApplet

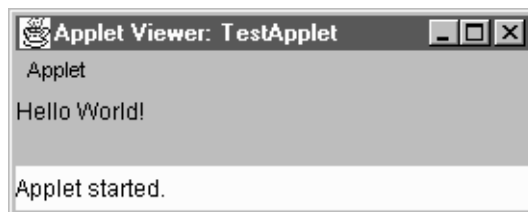


Figure 3.3
The “Hello World!” applet

3.2.2 JApplet hierarchy

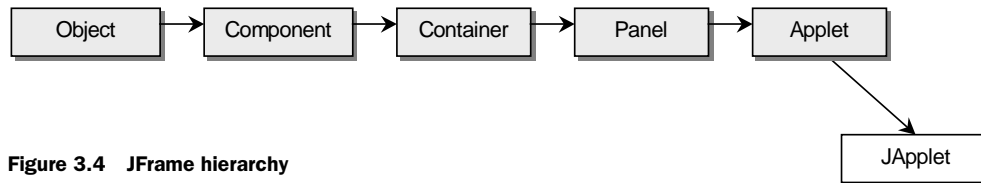


Figure 3.4 JFrame hierarchy

3.2.3 JApplet variables

```
protected boolean canAdd
```

This variable controls whether or not components can be added directly to the applet. Under most circumstances, this variable will contain a false value.

```
protected AccessibleContext accessibleContext
```

The `accessibleContext` variable contains an instance of the context used by the Assistive Technology mechanism of the component. In most situations, this variable will be unused.

3.2.4 JApplet constructors

```
JApplet()
```

This constructor creates a `JApplet` instance.

3.2.5 JApplet significant method groupings

```
public void setJMenuBar( JMenuBar menu );
public JMenuBar getJMenuBar();
```

If you have written AWT applets in the past, then you realize that it was impossible to assign a menu bar to an applet. With Swing, this limitation is a thing of the past.

```
public void setContentPane(Container contentPane);
public Container getContentPane();
public void setLayeredPane(JLayeredPane layered);
public JLayeredPane getLayeredPane();
public void setGlassPane(Component glass);
public Component getGlassPane();
protected void setRootPane(JRootPane root);
public JRootPane getRootPane();
protected JRootPane createRootPane();
```

A JApplet instance, like a JFrame, can contain three different panes: layered, pane, and root. The methods in this group manage these panes for the JApplet, allowing you to control where various panels, components, and pop-ups are placed within the hierarchy of the window.

3.3 Creating simple panels

The Swing equivalent of AWT's Panel class is JPanel. With few exceptions, everything you know about Panel applies equally to JPanel. JPanel supports all of the AWT layout managers and also the new layouts provided by Swing.

For example, if you recall the GridLayout source code from chapter 1, we can rewrite this as a Swing application in the following way:

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

class TestFrame
    extends JFrame
{
    public TestFrame()
    {
        setSize( 200, 200 );

        JPanel topPanel = new JPanel();
        topPanel.setLayout( new GridLayout( 3, 2 ) );
        getContentPane().add( topPanel );

        topPanel.add( new Button( "One" ) );
        topPanel.add( new Button( "Two" ) );
        topPanel.add( new Button( "Three" ) );
        topPanel.add( new Button( "Four" ) );
        topPanel.add( new Button( "Five" ) );
    }
}
```

Notice that the TestFrame class is now derived from JFrame rather than from Frame, and it creates a JPanel to which the GridLayout manager is applied. The buttons are now added to the JPanel instance, rather than to the main frame.

FYI

An instance of JPanel is double buffered by default. Developers of AWT code will appreciate this, since it simplifies dynamic aspects of a panel, such as animation, and reduces flicker while repainting. Double buffering can be turned off with a call to the following method from the JComponent class:

```
topPanel.setDoubleBuffered( false );
```

3.3.1 JPanel hierarchy

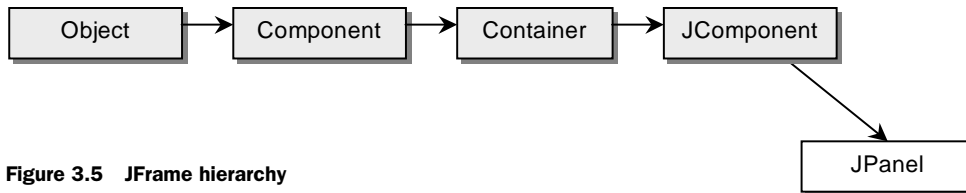


Figure 3.5 JFrame hierarchy

3.3.2 JPanel constructors

```
JPanel( LayoutManager layout, boolean isDoubleBuffered )
```

This constructor creates a JPanel instance with the specified layout. The double buffering capabilities of the panel are controlled by the specified boolean value.

```
JPanel( LayoutManager layout )
```

This constructor creates a JPanel instance with the specified layout.

```
JPanel( boolean isDoubleBuffered )
```

This constructor creates a JPanel instance with a default FlowLayout and the specified double buffering configuration.

```
JPanel()
```

This constructor creates a JPanel instance with a default FlowLayout and the double buffering enabled.

3.4 Simple border types

Borders can be applied to any Swing component, but in most instances it is a panel instance that gets a specific border style, so we will discuss these styles in this

section. All of the information in this section applies to all other Swing components; for example, you can apply a border to a label or text field using the techniques described in this section.

Swing implements several distinct border styles, shown in the following table:

Border	Description
BevelBorder	A 3-D border that supports a raised or lowered appearance
CompoundBorder	A border consisting of nested borders. The next section of this chapter discusses this special case.
EmptyBorder	A border permitting you to specify reserved space for an invisible border
EtchedBorder	A border that has the appearance of an etched line
LineBorder	A single color line of an arbitrary thickness
MatteBorder	A border allowing tiling of a specified icon or color
SoftBevelBorder	A border like <code>BevelBorder</code> , but having softer edges
TitledBorder	A border allowing a title string to be displayed on one of several orientations and alignments

Figure 3.6 shows all of the simple border styles provided by the Swing class library. These borders range from the ordinary to the obscure. To set the border of a `JPanel`, use code similar to the following fragment:

```
import javax.swing.*;
```

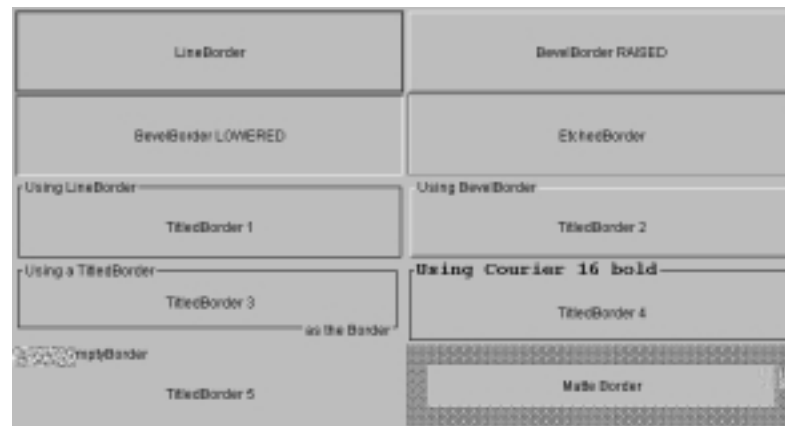


Figure 3.6 Simple border styles supported by Swing

```
import javax.swing.border.*;
{
    .
    .
    .
    JPanel myPanel = new JPanel();
    myPanel.setBorder( new BevelBorder ( BevelBorder.RAISED ) );
    .
    .
    .
}
```

Swing provides a shortcut class specifically for creating borders. This class, named `BorderFactory`, can be accessed by importing the `javax.swing.border` package. The following code fragment sets a raised, beveled border using `BorderFactory`:

```
import javax.swing.*;
import javax.swing.border.*;
{
    .
    .
    .
    JPanel myPanel = new JPanel();
    myPanel.setBorder( BorderFactory.createRaisedBevelBorder() );
    .
    .
    .
}
```



Use borders to create group boxes. Bordered panels may then be used to group functionality and visually segregate UI components. They could be used to enclose a set of related UI objects, such as radio buttons or check boxes. The contents of a group box are usually at a lower level of detail than is the label in the group box title. Use 3-D bevel border types to give the application a less dated look.

3.4.1 Creating a custom border class

The borders supplied by Swing are adequate (and standard) enough for most applications, but there may be times when a custom border is desirable. You can derive new borders by designing a class that implements the `Border` interface. This task is

relatively simple if you use the following code sample as a base. This example creates a class called `MyBorder`, that is instantiated by the main application class.

The main application class, shown in listing 3.3 creates a frame and adds some `JPanel` instances to it. The topmost panel has the normal default border style, the bottommost panel offers an etched border and the center panel shows what our custom border looks like. Notice in the output from this application, shown in figure 3.7, that the center panel includes a colored bevel around it. This simple example shows the instance of the `MyBorder` class and how it changes the appearance of the panel. Try modifying `MyBorder` to add a different look to the panel and you will see that creating your own custom borders is a simple exercise.

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

class MyBorder
    implements Border
{
    private Color    color;

    public MyBorder( Color color )
    {
        this.color = color;
    }

    public void paintBorder( Component component,
                            Graphics g, int iX, int iY,
                            int iWidth, int iHeight )
    {
        Insets insets = getBorderInsets( component );

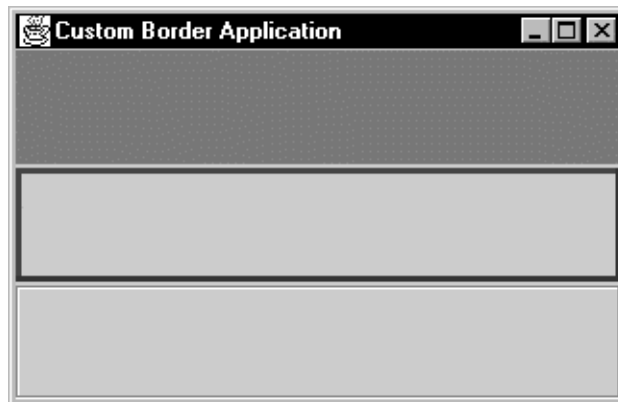
        g.setColor( color );
        g.fillRect( iX, iY, 3, iHeight );
        g.fillRect( iX, iY, iWidth, 3 );
        g.setColor( color.darker() );
        g.fillRect( iX + iWidth - insets.right, iY,
                    3, iHeight );
        g.fillRect( iX, iY + iHeight - insets.bottom,
                    iWidth, 3 );
    }

    public Insets getBorderInsets( Component component )
```

Listing 3.3 Custom border class

```
{
    return new Insets( 3, 3, 3, 3 );
}

public boolean isBorderOpaque()
{
    return false;
}
}
```

Listing 3.3 Custom border class**Figure 3.7** Custom border styles with Swing**FYI**

As we have already discussed, all Swing components extend the `JComponent` class. As shown in listing 3.3, we use the `JComponent.setBorder()` method to control the border style. This means that we can use our custom border on any Swing component—not just `JPanel` instances.

3.5 Compound border creation

Swing also supports compound borders (multiple borders applied to the same component). The need for this feature may not be immediately obvious, but there will certainly be times when you will want to create a component with a lowered border inside a raised border. Think of a status line, for example. You can create a status line by adding a text field to a raised panel, but this requires several lines of code and the creation of at least one superfluous panel. You will also find a compound border useful for applying an `EmptyBorder` around some other border type in order to create a larger-than-normal white space around a component. The code fragment below implements a compound border by combining a raised bevel with one that is lowered.

```
import javax.swing.*;

{
    .
    .
    .
    JPanel myPanel = new JPanel();
    myPanel.setBorder( BorderFactory.createCompoundBorder(
        new BevelBorder ( BevelBorder.RAISED ),
        new BevelBorder ( BevelBorder.LOWERED ) ) );
    .
    .
    .
}
```

FYI The first parameter to `createCompoundBorder()` represents the outside border, and the second parameter is the inside border.

Another example of a compound border is a border that possesses a visual style (an etched edge, for example) and also requires a title. Swing provides a slightly different interface to achieve this type of border. As shown in the following code fragment, the constructor of the `TitledBorder` class accepts a `Border` type as a parameter. This border can be any of the current borders supplied by Swing, or can be one of your own design if you create a class that extends `Border`.

```
import javax.swing.*;
import javax.swing.border.*;
```

```

{
.
.
.
myPanel.setBorder( new TitledBorder(
                    new EtchedBorder(), "Border Title" ) );
.
.
.
}

```

Finally, let's take a look at some code that generates a very complex border. Study the following code fragment to see if you can determine its results. Note that the code could have been written as one continuous line, but that the use of the temporary variable helps clarify its purpose.

```

import java.swing.*;
import javax.swing.border.*;

{
.
.
.
Border border1 = new TitledBorder( new EtchedBorder(),
                                "A very complex title" );
myPanel.setBorder( new TitledBorder( border1, "Another Title",
                                TitledBorder.RIGHT, TitledBorder.BOTTOM ) );
.
.
.
}

```

Did you get it right? Figure 3.8 shows the actual output from these two lines of code. As you can see, the `TitledBorder` class, and Swing border classes in general, provides the flexibility to create a border in almost any way you choose.



Figure 3.8
A very complex border

3.6 Swing layout managers

In chapter 1, we reviewed layout managers provided by AWT in Java 1.2. Though the AWT managers are completely compatible with Swing, and will be sufficient for most applications, Swing supports four additional layout managers to meet its own needs. The following table summarizes the Swing layout managers and their purposes:

Layout Manager	Description
BoxLayout	A layout manager that aligns components along the X- or Y- axis of a panel. It attempts to use the preferred width and height of components during the layout process.
OverlayLayout	Arranges components one on top of another, aligning the base point of each component in a single location
ScrollPaneLayout	A layout manager specific to scrolling panes
ViewportLayout	A layout manager specific to view ports within scrolling panes

Of the layout managers listed here, only `BoxLayout` will be useful in typical application development. The other layout managers discussed in the previous table may be useful to advanced Swing users, though these managers are quite specific and are usually applicable only to the components for which they are intended. Unless you are writing a new scrolling pane, you are unlikely to be able to apply the `ViewportLayout` manager to any other panel.

Since `BoxLayout` will be useful, we should look at it in a bit more detail. The `BoxLayout` organizes the components it manages along either the X- or Y-axis of the owner panel. The alignment of these components can be left or right justified, or centered (the default). The code in listing 3.4 creates two panels that support the `BoxLayout`. The upper panel aligns its components along the Y-axis, and the lower panel aligns along the X-axis. To each of these panels, we add three components of varying sizes so we can note the effects they have on the `BoxLayout` manager.

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

class TestFrame
    extends JFrame
    public TestFrame()
```

Listing 3.4 Sample using `BoxLayout`

```
{
    setTitle( "BoxLayout Application" );

    JPanel topPanel = new JPanel();
    topPanel.setLayout( new BorderLayout() );
    getContentPane().add( topPanel );

    // Create panels to display X- and Y-
    // axis box layouts
    JPanel yAxisPanel = createYAxisPanel();
    topPanel.add( yAxisPanel, BorderLayout.CENTER );
    JPanel xAxisPanel = createXAxisPanel();
    topPanel.add( xAxisPanel, BorderLayout.SOUTH );
}

public JPanel createYAxisPanel()
{
    JPanel panel = new JPanel();
    panel.setLayout( new BoxLayout( panel, BoxLayout.Y_AXIS ) );
    panel.setBackground( Color.lightGray );
    // Add some components to this panel
    panel.add( new JButton( "Button 1" ) );
    panel.add( new TextArea( "This is a text area" ) );
    panel.add( new JCheckBox( "Checkbox 1" ) );

    return panel;
}

public JPanel createXAxisPanel()
{
    JPanel panel = new JPanel();
    panel.setLayout( new BoxLayout( panel, BoxLayout.X_AXIS ) );
    panel.setBackground( Color.gray );

    // Add some components to this panel
    panel.add( new JButton( "Button 1" ) );
    panel.add( new TextArea( "This is a text area" ) );
    panel.add( new JCheckBox( "Checkbox 1" ) );

    return panel;
}

// Main program started
public static void main( String args[] )
{

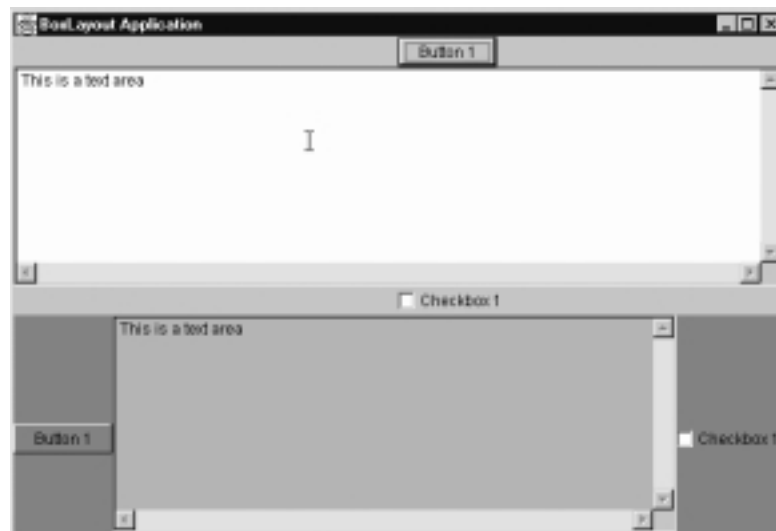
    // Create an instance of the test application
    TestFrame mainFrame = new TestFrame();
```

Listing 3.4 Sample using `BoxLayout` (continued)

```
mainFrame.pack();
mainFrame.setVisible( true );
}
}
```

Listing 3.4 Sample using BorderLayout (continued)

The output produced by this example is shown in figure 3.9. Notice how the components are laid out in each of the panels. In the upper panel, components are displayed vertically, and are centered horizontally. In the lower window (dark gray) the components are centered vertically, but are laid out in a horizontal direction,

**Figure 3.9 BorderLayout program output**

much like the FlowLayout manager would do it. Unlike FlowLayout, however, the BorderLayout manager will not wrap components to the next line. Instead, components will be reduced to the limits permitted by their specified minimum sizes, and if there is still insufficient panel real estate, the layout manager will simply clip components. This applies equally to panels with a vertical box layout.

Since some Swing components are ideally suited to the BorderLayout manager, we will examine a more practical implementation later. When adding Swing check boxes or radio buttons to a panel, it is highly recommended that you use the Box-

Layout, particularly if you have several components to lay out in tabular form. If you want to peek ahead to see an example using `BoxLayout`, you will find the source code in chapter 5, listing 5.5.

To make using the `BoxLayout` manager easier, Swing also provides a class named `Box` that creates a container with the `BoxLayout` manager applied. To quickly construct a panel of this type, use code similar to the following:

```
// Create a new panel
Box boxPanel = new Box(BoxLayout.Y_AXIS );

// Add components
boxPanel.add( new JButton( "Button 1" ) );
boxPanel.add( new TextArea( "This is a text area" ) );
boxPanel.add( new JCheckBox( "Checkbox 1" ) );
```

3.7 JRootPane in detail

We broached the concept of a root pane when we discussed `JFrames` and `JApplets`, but until now we haven't really gone into much detail on what the root pane actually is. The root pane is a component which is added to every frame and is managed such that it always covers the entire surface of the window, except for the title bar and the frame borders.

`JRootPane` contains four other components—a layered pane, an optional menu bar, a content pane, and a glass pane. Figure 3.10 shows a graphical rendition of a `JFrame` containing a `JRootPane` instance and its four standard components. In the example code shown in this chapter, the constructor for most frames includes statements like:

```
JPanel topPanel = new JPanel();
getContentPane().add( topPanel );
```

This is not the way you would have done this with AWT because that model did not share the `JRootPane` concept. In AWT you could create a frame and simply start adding components to it. The code fragment above is the Swing equivalent of:

```
Panel topPanel = new Panel();
add( topPanel );
```

in AWT. When building Swing applications and applets, you must reference the content pane when adding components—in fact the compiler will generate errors if you don't. So you will have to get used to it.

The glass pane is a special transparent component on top of every frame instance. Though you can't see the glass pane, you can add components to it, allowing you to create some special effects. For example, if you wanted to show some sort of animated Swing component on top of your frame the glass pane is the place to display it. You can add things to the glass pane in a similar way that components are added to the content pane:

```
JPanel floatingPanel = new JPanel();
getGlassPane().add( floatingPanel );
```

JRootPane, being part of every frame and applet, offers some interesting potential for building unique user interfaces when you want it, particularly when using multiple layers (which we'll discuss next) and adding components to the glass pane. Fortunately, when you are writing standard uneventful user interface code, the power of JRootPane stays out of your way.

3.7.1 JRootPane hierarchy

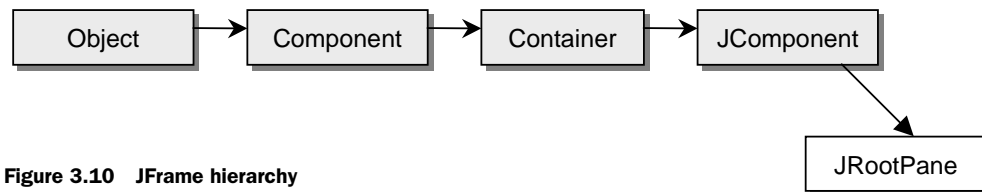


Figure 3.10 JFrame hierarchy

3.7.2 JRootPane variables

```
protected Container contentPane
```

This field holds a reference to the content pane used by the JRootPane.

```
protected Container glassPane
```

This field holds a reference to the glass pane used by the JRootPane.

```
protected JLayeredPane layeredPane
```

This field holds a reference to the layered pane used by the JRootPane.

```
protected JMenuBar menuBar
```

This field holds a reference to the optional menubar used by the JRootPane.

```
protected JButton defaultButton
```

This field holds a reference to the JButton instance in the object hierarchy that gets activated when the root pane has the focus and the user presses the Enter key.

```
protected JRootPane.DefaultAction defaultPressAction
```

This field holds a reference to the Swing action that is invoked when the user presses the default button.

```
protected JRootPane.DefaultAction defaultReleaseAction
```

This field holds a reference to the Swing action that is invoked when the user releases the default button.

3.7.3 JRootPane constructors

```
JRootPane()
```

This default constructor creates an instance of a Swing root pane. New JRootPane instances will never be created by a user application. You will likely never create instances of JRootPane unless you are developing a new type of frame object.

3.7.4 JRootPane significant method groupings

```
protected Container createContentPane()  
protected Component createGlassPane()  
protected JLayeredPane createLayeredPane()
```

The methods within this group are used to create instances of the internal pane objects used by the JRootPane. These methods are called by the JRootPane constructor.

```
Container getContentPane()  
Component getGlassPane()  
JMenuBar getJMenuBar()  
JLayeredPane getLayeredPane()  
void setContentPane(Container content)  
void setGlassPane(Component glass)  
void setJMenuBar(JMenuBar menu)  
void setLayeredPane(JLayeredPane layered)
```

These methods are used to access the four component parts of a JRootPane object. The `getMenuBar()` and `setMenuBar()` methods included in Swing 1.0 have been omitted from this group because they have been deprecated in Swing 1.1. Use the `getJMenuBar()` and `setJMenuBar()` methods instead.

3.8 Layered panes

The `JLayeredPane` class in Swing is a container which is provided by every `JFrame` or `JApplet`. It operates much like an AWT card layout in that child components are layered or stacked on top of one another. The primary difference is that a `JLayeredPane` is transparent, allowing the user to see more than one component at a time.

3.8.1 Layered pane basics

Swing provides a complete API for layered pane use and supports not only a series of standard layers, but a near infinite number of user configurable layers as well. The standard layers are shown in the following table in decreasing order of precedence:

Layer	Value	Description
<code>JLayeredPane.DRAG_LAYER</code>	400	Used to display components that are dragged around the frame. This is the topmost layer, so any component shown on this layer will display above all other components in the frame.
<code>JLayeredPane.POPUP_LAYER</code>	300	Used to display popup components such as a context menu.
<code>JLayeredPane.MODAL_LAYER</code>	200	Used to display components that are modal to a particular process. For example, use this layer to show a modal search window on top of other components in the frame.
<code>JLayeredPane.PALETTE_LAYER</code>	100	Use this layer to show palettelike objects such as color selectors or font choosers
<code>JLayeredPane.DEFAULT_LAYER</code>	0	This is the layer on which components are drawn by default. Use this layer for most component displays.
<code>JLayeredPane.FRAME_CONTENT_LAYER</code>	-30000	This is a special layer which is functionally equivalent to writing to a heavyweight component. This is the layer Swing uses to place the content pane and menu; its use should generally be avoided.

The layer attribute can be predefined for a `Component` by passing an `Integer` object during the add operation. For example:

```
layeredPane.add( component, JLayeredPane.DEFAULT_LAYER );
```

or

```
layeredPane.add( component, new Integer( 0 ) );
```

Additionally, the layer attribute can be set on a Component object by calling:

```
layeredPane.setLayer( component, JLayeredPane.DEFAULT_LAYER );
```



The layer of a component should be set before the component is added to the JLayeredPane instance.

The layer of a component can be determined anytime with the call:

```
int iLayer = layeredPane.getLayer( component );
```

JLayeredPane supports other methods for determining layers, component references, and so on, but we won't be going into any more detail here. The Swing online documentation describes the methods supported by this class.



Mixing AWT and Swing components in a JLayeredPane instance is generally a bad idea. AWT components will always appear on top of Swing components, destroying the layering effect.

In addition to defining the initial layer of a component, Swing supports a mechanism to move components from one layer to another. For example, to move a component to the back of the stack use:

```
layeredPane.moveToBack( component );
```

to bring a component to the front of the layer stack, use:

```
layeredPane.moveToFront( component );
```

The absolute layer position of a component can be set using:

```
layeredPane.setPosition( component, JLayeredPane.PALETTE_LAYER );
```

3.8.2 A Layered pane example

Before we move on to other topics, we should examine a simple example of layered pane use. This example, shown in listing 3.5, creates a simple frame and adds a simple label and a button (labeled “Move left”) to the default layer. To this frame, the example code adds a second button (labeled “Move right”), but this time it is

added to the drag layer. For effect, the code also creates a red panel on the palette layer. Finally, on the modal layer a label containing the string “Watch me” is added.

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

class TestFrame
    extends JFrame
{
    private JLayeredPane layerPane;
    private JLabel l2;
    private int position;

    public TestFrame()
    {
        position = 0;

        setTitle( "Layered Pane Application" );
        setSize( 400, 500 );

        JPanel topPanel = new JPanel();
        topPanel.setLayout( new BorderLayout() );
        getContentPane().add( topPanel );

        // Add some objects to the default layer
        JLabel l1 = new JLabel( "A JLabel" );
        l1.setPreferredSize( new Dimension( 80, 25 ) );
        topPanel.add( l1, BorderLayout.NORTH );

        JButton b1 = new JButton( "Move left" );
        b1.setPreferredSize( new Dimension( 100, 25 ) );
        topPanel.add( b1, BorderLayout.SOUTH );

        // Get the frame's layered pane
        layerPane = getLayeredPane();

        // Add a button to the drag layer
        JButton b2 = new JButton( "Move right" );
        b2.setBackground( Color.green );
        b2.setBounds( 10, 15, 150, 30 );
        layerPane.add( b2, JLayeredPane.DRAG_LAYER );
        // Add a panel to the palette layer
```

Listing 3.5 A layered pane example

```
JPanel p1 = new JPanel();
p1.setBackground( Color.blue );
p1.setBounds( 40, 20, 250, 60 );
layerPane.add( p1, JLayeredPane.PALETTE_LAYER );

// Add a moving lable to the modal layer
l2 = new JLabel( "Watch me" );
l2.setBounds( 0, 30, 75, 30 );
layerPane.add( l2, JLayeredPane.MODAL_LAYER );
// Anonymous listener for the first button
b1.addActionListener( new ActionListener()
{
    public void actionPerformed((ActionEvent e )
    {
        // Move the scrolling label left
        if( position >= 0 )
            position -= 5;
        l2.setLocation( position, 30 );
    }
});

// Anonymous listener for the second button
b2.addActionListener( new ActionListener()
{
    public void actionPerformed((ActionEvent e )
    {
        // Move the scrolling label left
        if( position < 300 )
            position += 5;
        l2.setLocation( position, 30 );
    }
});

//Anonymous inner class to terminate program.
this.addWindowListener( new WindowAdapter()
{
    public void windowClosing( WindowEvent e )
    {
        System.exit( 0 );
    }
} );//end addWindowListener

}

// Main method to get things started
public static void main( String args[] )
```

Listing 3.5 A layered pane example (continued)

```

{
    // Create an instance of the test application
    TestFrame mainFrame= new TestFrame();
    mainFrame.setVisible( true );
}

```

Listing 3.5 A layered pane example (continued)

Notice that clicking the left and right buttons moves the “Watch me” text around the frame; however notice that this text “slides” between the “Move right” button and the read panel. Also notice that if you resize the window similar to example output shown in figure 3.12, the “Move left” button moves underneath the red panel.

Of course this example isn’t very practical in the real world, but it does dem-



Figure 3.11 Layered pane program output

onstrate the concept of layering. You should use layering with caution, since hiding information can be very distracting to a user. Swing uses layering in a practical way to create internal frames (Microsoft Windows people refer to this as multidocument interface), which we will discuss in chapter 9.

3.8.3 JLayeredPane hierarchy

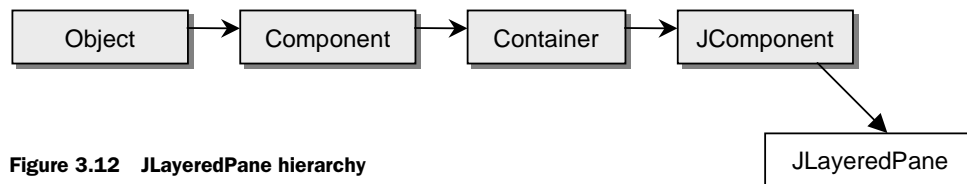


Figure 3.12 JLayeredPane hierarchy

3.8.4 JLayeredPane constants

```
public static final Integer DEFAULT_LAYER
public static final Integer PALETTE_LAYER
public static final Integer MODAL_LAYER
public static final Integer POPUP_LAYER
public static final Integer DRAG_LAYER
public static final Integer FRAME_CONTENT_LAYER
```

These constants are used to identify the layers within the layered pane. Each of these predefined layers has a specific purpose that is dependent on the type of pane over which it is applied.

3.8.5 JLayeredPane variables

```
protected boolean paintBackground
```

This boolean value controls the requirement to paint the background of the layered pane during a repaint. If the layered pane is transparent, this value will be false.

3.8.6 JLayeredPane constructors

```
JLayeredPane()
```

This default constructor creates a new instance of a layered pane.

3.8.7 JLayeredPane significant method groupings

```
public void setLayer(Component c, int layer)
public void setLayer(Component c, int layer, int position)
public int getLayer(Component c)
public int highestLayer()
public int lowestLayer()
public static void putLayer(JComponent c, int layer)
public static int getLayer(JComponent c)
public Component[] getComponentsInLayer(int layer)
protected int insertIndexForLayer(int layer, int position)
public int getComponentCountInLayer(int layer)
```

The methods within this group are used to manage the particular layers within the pane. With these methods, the current, highest, or lowest layers can be determined. Also, new components can be assigned to specific layers.

```
public int getIndexOf( Component c )
public void moveToFront( Component c )
public void moveToBack( Component c )
```

```
public void setPosition( Component c, int position )
public int getPosition( Component c )
```

This group of methods controls the position of components within the layers of the pane.

3.9 Chapter summary

This chapter has covered a lot of the preliminary ground required to understand Swing. You now know how Swing wraps the AWT application framework classes to create JFrame and JApplet, and have seen examples using these new classes.

This chapter also presented the JPanel class which, like AWT's Panel class, is used in almost every conceivable application. You saw how to take simple components (including AWT components) and add them to a JPanel instance to create forms, button arrays, and so on.

Also in this chapter we discussed the Swing Border class and its derivatives. We also examined ways in which you can use it to create your own borders to produce some interesting effects.

Finally, this chapter described some of the inner workings of Swing frames, including layered panes and the JRootPane instance contained within every Swing frame, applet and dialog. Every Swing program will use the content pane; however, you will find the presence of the layered pane can solve many problems which were extremely tedious or impossible with AWT.

