
8

In this chapter:

- *A Simple Container*
- *The JPanel Class*
- *The JRootPane Class*
- *The RootPaneContainer Interface*
- *The JLayeredPane Class*
- *The WindowConstants Interface*
- *Basic RootPaneContainers*
- *The JFrame Class*
- *The JWindow Class*
- *The JApplet Class*

Swing Containers

In this chapter, we'll take a look at a number of components Swing provides for grouping other components together. In AWT, such components extended `java.awt.Container` and included `Panel`, `Window`, `Frame`, and `Dialog`. With Swing, you get a whole new set of options, providing greater flexibility and power. Consequently, most containers in Swing are a bit more complex than they were with AWT. However, there's an exception, the `JPanel`, which is very simple. We'll take that as our starting point.

A Simple Container

Not everything in this chapter is more complex than its AWT counterpart. As proof of this claim, we'll start the chapter with a look at the `JPanel` class, a very simple Swing container.

The JPanel Class

`JPanel` is an extension of `JComponent` (which, remember, extends `java.awt.Container`) used for grouping together other components. It gets most of its implementation from its superclasses. Typically, using `JPanel` amounts to instantiating it, setting a layout manager (this can be set in the constructor and defaults to a `FlowLayout`), and adding components to it via the `add()` method inherited from `Container`.

Properties

`JPanel` does not define any new properties. Table 8-1 shows the default values that differ from those provided by `JComponent`.

Table 8-1. *JPanel* Properties

Property	Data Type	get	is	set	bound	Default Value
<code>accessibleContext*</code>	<code>AccessibleContext</code>	•				<code>AccessibleJPanel</code>
<code>background*</code>	<code>Color</code>	•		•		
<code>doubleBuffered*</code>	<code>true</code>		•	•		<code>true</code>
<code>font*</code>	<code>Font</code>	•		•		
<code>foreground*</code>	<code>Color</code>	•		•		
<code>layout*</code>	<code>LayoutManager</code>	•		•		<code>FlowLayout</code>
<code>opaque*</code>	<code>boolean</code>	•	•		•	<code>true</code>

See also properties from the `JComponent` class.

`AccessibleContext` is as expected. The `doubleBuffered` and `opaque` properties default to `true` while the `layoutManager` defaults to a new `FlowLayout`.

The other three properties shown in the table are set in the constructor by calling the accessor for each property, allowing subclasses to define custom values. If these accessors return either `null` or a `UIResource`, the values are determined by calling `UIManager.getFont()` and `UIManager.getColor()`. See Chapter 26, *Look & Feel*, for more information on `UIManager`, `UIResource`, and the rest of the pluggable look-and-feel architecture.

Constructors

public JPanel()

Creates a new panel with a `FlowLayout` and double buffering.

public JPanel(boolean isDoubleBuffered)

Creates a new panel with a `FlowLayout` and double buffering according to the input parameter.

public JPanel(LayoutManager layout)

Creates a new panel with the specified layout manager and double buffering.

public JPanel(LayoutManager layout, boolean isDoubleBuffered)

This constructor (called by all the others) creates a new panel with the specified layout manager and double-buffering policy. It sets the panel's foreground, background, and font as described in the "Properties" section.

User Interface Methods

public void paintComponent(Graphics g)

Paints the panel's background if it is opaque. Otherwise, it does nothing.

public void updateUI()

Called to indicate that the L&F has changed. Unlike most Swing components, there is no UI delegate for JPanel so this method essentially does nothing.

Opacity

Here's a simple program showing what it means for a JPanel to be opaque. All we do here is create two JPanels. Inside the first JPanel, we place another JPanel, which is opaque. In the second, we place a transparent (non-opaque) JPanel. In both cases, we set the background of the outer panel to white and the background of the inner panel to black. We'll place a JButton inside each inner panel to give it some size. Figure 8-1 shows the result.

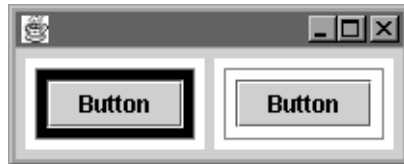


Figure 8-1. Non-opaque and opaque JPanels

On the left, we see the black panel inside the white one. But on the right, since the inner panel is not opaque, its black background is never painted and the background of the outer panel shows through. Here's the code:

```
// OpaqueExample.java
//
import com.sun.java.swing.*;
import java.awt.*;

public class OpaqueExample {
    public static void main(String[] args) {

        // Create 2 JPanels (opaque), one containing another opaque JPanel, and
        // the other containing a nonopaque JPanel
        JPanel opaque = createNested(true);
        JPanel notOpaque = createNested(false);

        // Throw it all together in a JFrame
        JFrame f = new JFrame();
        f.addWindowListener(new BasicWindowMonitor());
        f.getContentPane().setLayout(new FlowLayout());
        f.getContentPane().add(opaque);
```

```

        f.getContentPane().add(notOpaque);
        f.pack();
        f.setVisible(true);
    }

    // Create a JPanel containing another JPanel. The inner JPanel's opacity
    // is set according to the parameter. A JButton is placed inside the inner
    // JPanel to give it some content.
    public static JPanel createNested(boolean opaque) {
        JPanel outer = new JPanel(new FlowLayout());
        JPanel inner = new JPanel(new FlowLayout());
        outer.setBackground(Color.white);
        inner.setBackground(Color.black);

        inner.setOpaque(opaque);

        inner.setBorder(BorderFactory.createLineBorder(Color.gray));

        inner.add(new JButton("Button"));
        outer.add(inner);

        return outer;
    }
}

```

The Box Class

Another very useful Swing container is the `Box`. A `Box` makes it easy to lay out a series of components end-to-end. Since it's supported by a special new layout manager (`BoxLayout`), we'll cover it later in Chapter 11, *Specialty Panes and Layout Managers*.

The Root Pane

Now that we've seen the simplest example of a Swing container, we'll move on to something a bit more powerful. Most of the other Swing containers (`JFrame`, `JApplet`, `JWindow`, `JDialog`, and even `JInternalFrame`) contain an instance of another class, `JRootPane`, as their only component, and implement a common interface, `RootPaneContainer`. In this section, we'll look at `JRootPane` and `RootPaneContainer`, as well as another class `JRootPane` uses, `JLayeredPane`.

Before jumping into the descriptions of these classes, let's take a look at how the classes and interfaces that make up the Swing root containers fit together.

Figure 8-2 shows that `JApplet`, `JFrame`, `JDialog` and `JWindow` do not extend `JComponent` like the other Swing components. Instead, they extend their AWT counterparts, serving as top-level user interface windows. This implies that these

components (unlike the lightweight Swing components) have native AWT peer objects.

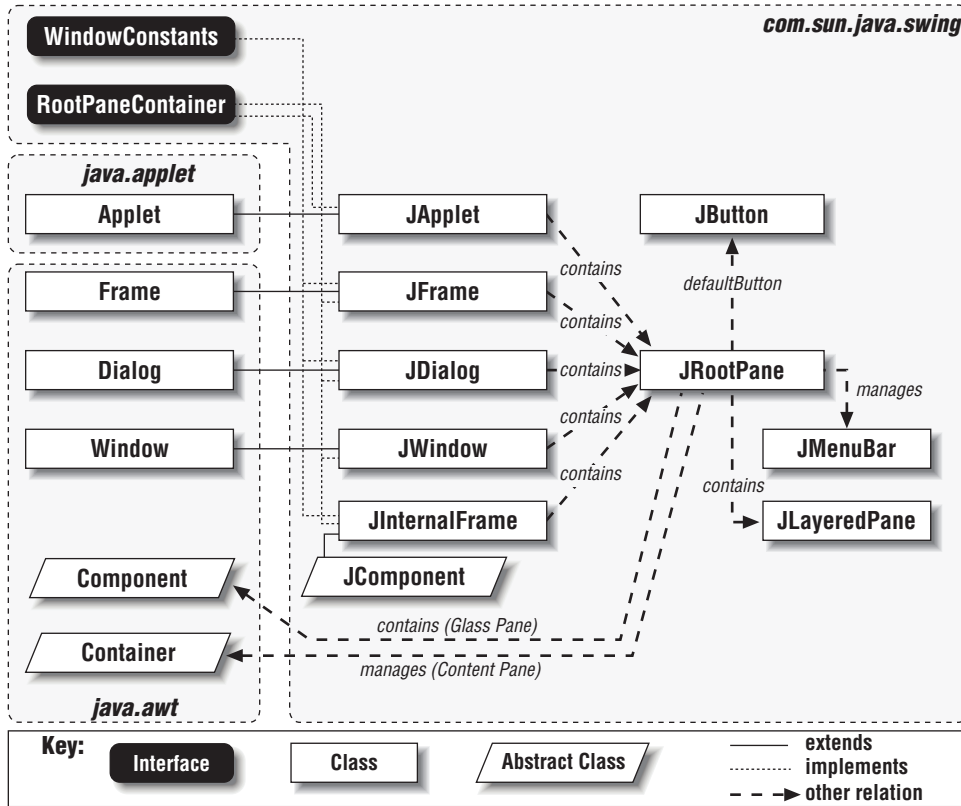


Figure 8-2. Swing “Root” containers

Notice that these Swing containers (as well as `JInternalFrame`) implement a common interface, `RootPaneContainer`. This is an interface that gives access to the `JRootPane`’s properties. Furthermore, each of the five containers uses a `JRootPane` as the “true” container of child components managed by the container. This class is discussed later in this chapter.

The `JRootPane` Class

`JRootPane` is a special container that extends `JComponent` and is used by many of the other Swing containers. It’s quite different from most containers you’re probably used to using. The first thing to understand about `JRootPane` is that it contains a fixed set of components: a `Component` called the glass pane and a `JLayeredPane` called, logically enough, the layered pane. Furthermore, the layered pane contains two more components: a `JMenuBar` and a `Container` called

the content pane.* Figure 8-3 shows a schematic view of the makeup of a `JRootPane`.

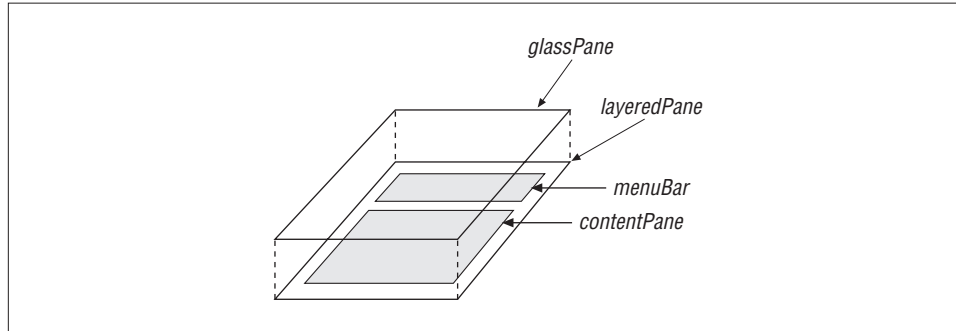


Figure 8-3. `JRootPane`

Attempts to add additional components to a `JRootPane` are ignored by its custom layout manager (a protected inner class called `RootLayout`).[†] Instead, children of the root pane should typically be added to its content pane. In fact, for most uses of `JRootPane`, all you'll need to do is get the content pane and add your components to it. Here's a simple example (using a `JFrame`) that adds a single button to the content pane:

```
// RootExample.java
//
import com.sun.java.swing.*;
import java.awt.*;

public class RootExample {
    public static void main(String[] args) {
        JFrame f = new JFrame();
        f.addWindowListener(new BasicWindowMonitor());
        JRootPane root = f.getRootPane(); // XXX
        Container content = root.getContentPane(); // XXX
        content.add(new JButton("Hello")); // XXX
        f.pack();
        f.setVisible(true);
    }
}
```

* Note that in general, `JLayeredPanels` can contain any components they wish. This is why Figure 8-2 does not show `JLayeredPane` as containing the menu bar and content pane. In the case of the `JRootPane`, a `JLayeredPane` is used to hold these two specific components.

† It is possible to change the layout manager to one of your own choosing, but it would be responsible for handling all details of laying out the `JRootPane`. Using any of the other AWT or Swing layouts will not work properly.

This may seem like a lot of complexity just to add something to a frame. Thankfully, as we'll see in the next section, each of the containers that use `JRootPane` implement the `RootPaneContainer` interface, which provides direct access to each of the root's subcomponents. This allows the three lines marked with 'XXX' to be replaced with:

```
f.getContentPane().add(new JButton("Hello"));
```

In the next example, we see how to add a menu to a root pane, producing a display like the one in Figure 8-4.



Figure 8-4. `JRootPane` with a `JMenuBar`

```
// RootExample2.java
//
import com.sun.java.swing.*;
import java.awt.*;

public class RootExample2 {
    public static void main(String[] args) {
        JFrame f = new JFrame();
        f.addWindowListener(new BasicWindowMonitor());
        JRootPane root = f.getRootPane();

        // Create a menu bar
        JMenuBar bar = new JMenuBar();
        JMenu menu = new JMenu("File");
        bar.add(menu);
        menu.add("Open");
        menu.add("Close");
        root.setMenuBar(bar);

        // Add a button to the content pane
        root.getContentPane().add(new JButton("Hello World"));

        // Display the UI
        f.pack();
        f.setVisible(true);
    }
}
```

In this case, the `getRootPane()` and `setMenuBar()` calls could have been replaced with a single `f.setJMenuBar(bar)` call. Note that the menubar prop-

erty on the Swing containers is called `JMenuBar`, while the name used in `JRootPane` is `menuBar`. This unusual name in the Swing components is used to avoid a name clash with the existing AWT methods for working with menubars.

The previous two root pane examples were intended to give you an understanding of how the `JRootPane` really works. Typically, however, your code will not be working with `JRootPane` directly. We'll get a better understanding of why when we get to the discussion of `RootPaneContainer`. For now, here's a version of the last example that shows how you'd really write that code:

```
// RootExample3.java
//
import com.sun.java.swing.*;
import java.awt.*;

public class RootExample3 {
    public static void main(String[] args) {
        JFrame f = new JFrame();
        f.addWindowListener(new BasicWindowMonitor());
        JMenuBar bar = new JMenuBar();
        JMenu menu = new JMenu("File");
        bar.add(menu);
        menu.add("Open");
        menu.add("Close");

        // Shortcuts provided by JFrame (and other RootPaneContainers)...
        f.setJMenuBar(bar);
        f.getContentPane().add(new JButton("Hello World"));

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

Understanding the Glass Pane

`JRootPane` may seem a bit confusing at first glance. The important thing to remember is that in most cases, all you'll need to worry about is adding your component to the content pane and possibly setting a menubar. In this section, we'll explain the other component contained by `JRootPane`: the "glass pane."

The glass pane is a component that is laid out to fill the entire pane. By default, it is an instance of `JPanel`, but it can be replaced with any `Component`. `JRootPane`'s implementation of the `addImpl()` method ensures that the glass pane is the first component in the container, meaning that it will be painted last. In other words, the glass pane allows you to place components "above" any other components in the pane. Because of this, it generally makes sense for the glass pane to be

non-opaque; otherwise it will cover everything in the layered pane. It's important to remember that when the layout of the `JRootPane` is performed, the placement of the contents of the glass pane will have no effect on the placement of the contents of the layered pane (and its content pane). Both sets of components are placed within the same component space, overlapping each other as necessary. It's also important to realize that the components in the various panes are all equal when it comes to receiving input: mouse events are sent to any component in the `JRootPane`, whatever part of the pane it happens to be in.

This last note brings us a common use of the glass pane – blocking mouse events from the other components. As a rule, mouse events are sent to the “top” component if components are positioned on top of each other. If the top component has registered mouse listeners, the events will not be sent to the covered components. In the next chapter, we'll see how `JInternalFrame` takes advantage of this technique.

For now, we'll look at an example in which we use the glass pane to display a single button above the rest of the root pane's content. The panel will listen for all mouse events (and do nothing with them). Once the “start” button in the glass pane is pressed, the glass pane will be removed, allowing the underlying components to be used again. This example is not intended to be particularly useful. It's here for the purpose of demonstrating how the glass pane works.

```
// GlassExample.java
//
import com.sun.java.swing.*;
import java.awt.*;
import java.awt.event.*;

// Show how a glass pane can be used to block mouse events
public class GlassExample {
    public static void main(String[] args) {

        // Create a frame and its content pane's contents
        JFrame f = new JFrame();
        f.addWindowListener(new BasicWindowMonitor());
        final JPanel p1 = new JPanel();
        p1.add(new JLabel("Foo, Bar & Baz", JLabel.CENTER));
        p1.add(new JButton("Foo"));
        p1.add(new JButton("Bar"));
        f.getContentPane().add(p1);

        // Place a button in the center of the glass pane and make it visible
        final JPanel glass = (JPanel)f.getGlassPane();

        glass.setVisible(true);
        glass.setLayout(new GridBagLayout());
```

```

    JButton glassButton = new JButton("Start");
    glass.add(glassButton);

    // Register for all the mouse events in the glass pane (and do nothing).
    // This registration keeps the components in the content pane from being
    // clickable. We could have created our own panel that called
    // enableEvents(AWTEvent.MOUSE_EVENT_MASK |
    // AWTEvent.MOUSE_MOTION_EVENT_MASK) to get the same effect.
    glass.addMouseListener(new MouseAdapter() {});
    glass.addMouseMotionListener(new MouseMotionAdapter() {});

    // Add a listener to the glass pane's button that will make the glass
    // pane invisible when the button is clicked.
    glassButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ev) {
            glass.setVisible(false);
            pl.repaint();
        }
    });

    // Display the example . . .
    f.setSize(150, 80);
    f.setVisible(true);
}
}

```

Note that the lines:

```

glass.addMouseListener(new MouseAdapter() {});
glass.addMouseMotionListener(new MouseMotionAdapter() {});

```

block mouse events from reaching the hidden components (remember, the glass pane fills the entire frame), because the events are sent to the first component (starting at the top) with registered listeners. Any time a mouse event method is called, it will do nothing, since we just extended the empty-implementation adapter classes.

Once the start button is clicked, the glass pane is no longer visible, so the previously hidden components will now receive the events.

Figure 8-5 shows how the glass pane is drawn above the content pane when this example is run. If you were to call `setOpaque(true)` on the glass pane, only the start button would be visible.

Avoiding Unnecessary Layers

The following code fragment shows a common mistake that's easy to make:

```

JPanel panel = new JPanel();
panel.add(someStuff);

```



Figure 8-5. Glass pane

```
JFrame f = new JFrame();
f.getContentPane().add(panel);
```

There's nothing fundamentally wrong with this code. It will work just fine. However, there's an extra layer added here that's just not necessary. Recall from the beginning of this section that the content pane is initialized to an instance of `JPanel`. There's nothing special about that panel, and you should feel free to use it. A better implementation of the code fragment we just looked at would be:

```
JFrame f = new JFrame();
Container panel = f.getContentPane(); // cast to JPanel if you want to
panel.add(someStuff);
```

It's also important to keep in mind that the content pane can be any arbitrary container—it doesn't have to be a `JPanel`. If you want to fill the content pane with a scrollable region, or perhaps with a tabbed pane, you can replace the content pane with a `JScrollPane` or `JTabbedPane`. For example:

```
JScrollPane scroll = new JScrollPane(new JTextPane());
JFrame f = new JFrame();
f.setContentPane(scroll); // not f.getContentPane().add(scroll);
```

A reasonable rule of thumb is that if you are only going to add a single component to the content pane, and you want it to fill the entire pane, don't add to the content pane—replace it.

Properties

Table 8-2 shows the properties and default values defined by `JRootPane`.

Table 8-2. *JRootPane* Properties

Property	Data Type	get	is	set	bound	Default Value
<code>accessibleContext*</code>	<code>AccessibleContext</code>	•				<code>AccessibleJRootPane</code>
<code>background*</code>	<code>Color</code>	•		•		<code>UIManager.getColor("control")</code>
<code>contentPane</code>	<code>Container</code>	•		•		<code>JPanel</code>

See also properties from the `JComponent` class.

Table 8-2. *JRootPane Properties*

Property	Data Type	get	is	set	bound	Default Value
defaultButton	JButton	•		•	•	null
doubleBuffered*	boolean	•		•		true
glassPane	Component	•		•		JPanel
layeredPane	JLayeredPane	•		•		JLayeredPane
layout*	LayoutManager	•		•		RootLayout
menuBar	JMenuBar	•		•		null
validateRoot*	boolean		•			true

See also properties from the JComponent class.

The `accessibleContext` property follows the normal accessibility pattern. The `background` property is set to the default “control” color defined in the `UIManager`.

The `contentPane` is initially set to a `JPanel` with a `BorderLayout`, while `glassPane` is set to a non-opaque, invisible `JPanel` with a default (`FlowLayout`) layout manager. A new instance of `JLayeredPane` is the default value for `layeredPane`, and by default the `menuBar` property is set to `null`. The `contentPane` is contained by the layered pane’s `FRAME_CONTENT_LAYER` (see the section on `JLayeredPane` for further explanation).

Note that the `set()` methods for the `menuBar` and `contentPane` properties take care of placing these components within the `JLayeredPane`, so you don’t typically have to worry about the layered pane at all.

The inherited `doubleBuffered` property is `true` by default and `layout` defaults to a new instance of the protected inner class `RootLayout`.

The `defaultButton` property was introduced back in Chapter 5, *Buttons*. This property allows a `JButton` to be specified as the default for the container. The default button is the button that will be “pressed” if the user presses “enter” (or some other UI-defined key) while the pane has focus (unless some other focused component, like a `JTextField`, handles the key). This is a very convenient feature when presenting a user with information to be viewed and acknowledged, because it keeps the user from having to use the mouse.

Revalidate

The last property listed in Table 8-2 is the `revalidate` property. `JRootPane` overrides `isValidateRoot()` to return `true`. This causes the container to be validated (meaning that its contents will be redisplayed) as a result of any call to

`revalidate()` on one of its children or their descendants. This simplifies the process of dealing with components that change dynamically.

Previously, if the font size (for example) of a component changed, you needed to call `invalidate()` on the component and then `validate()` on its container to ensure that the component would be resized appropriately. Using `revalidate()`, only one call is necessary. Furthermore, the way `revalidate()` is implemented allows multiple `revalidate()` calls to be handled at once, much like multiple `repaint()` calls are handled at the same time by the AWT.

Here's a simple example that shows how `revalidate()` can be used:

```
// RevalidateExample.java
//
import com.sun.java.swing.*;
import java.awt.*;
import java.awt.event.*;

public class RevalidateExample {
    public static void main(String[] args) {
        JFrame f = new JFrame();
        f.addWindowListener(new BasicWindowMonitor());

        // Create a single button
        Font font = new Font("Dialog", Font.PLAIN, 10);
        final JButton b = new JButton("Add");
        b.setFont(font);

        Container c = f.getContentPane();
        c.setLayout(new FlowLayout());
        c.add(b);

        // Increase the size of the button's font each time it's clicked
        b.addActionListener(new ActionListener() {
            int size = 10;

            public void actionPerformed(ActionEvent ev) {
                b.setFont(new Font("Dialog", Font.PLAIN, ++size));
                b.revalidate(); // invalidates the button & validates its root pane
            }
        });

        f.setSize(150, 120);
        f.setVisible(true);
    }
}
```

In this example, we create a single button and add it to the content pane of a `JFrame` (which uses a `JRootPane`). Each time the button is clicked, we increase

the size of the button's font. As a result, the button needs to be resized to accommodate the larger label. To make this happen, we simply call `revalidate()` on the button. Note that the button could have been nested inside any number of other containers below the root pane and this would still work properly. As long as there is an ancestor of the revalidated component that returns `true` to `isValidateRoot()`, the container will be validated.

Protected Fields

The following fields are available to subclasses of `JRootPane`:

protected JMenuBar menuBar
protected Container contentPane
protected JLayeredPane layeredPane
protected Component glassPane
protected JButton defaultButton

These fields just hold the components described in the Properties section.

protected JRootPane.DefaultAction defaultPressAction
protected JRootPane.DefaultAction defaultReleaseAction

These fields contain the `Actions` used to implement the behavior associated with the `defaultButton` property. `DefaultAction` is a package-level inner class.

Constructors

public JRootPane()

Creates a new pane with the default property values specified in Table 8-2. It uses the four protected *create* methods listed below, so subclasses may choose to override certain aspects of the creation process.

Protected Methods

Other than the accessors for the pane's properties, the only methods defined in `JRootPane` are a series of protected methods used primarily to create the various components that make up the pane.

protected void addImpl(Component comp, Object constraints, int index)

Overrides the default implementation from `Component` to ensure that the glass pane will always be the first component in the container.

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

Called by the constructor to build the pane's subcomponents and set the default layout manager. The default implementations set the default properties as described in the "Properties" section.

Inner Classes

```
protected class RootLayout implements LayoutManager2, Serializable
```

The default layout manager for `JRootPane`. It places the glass pane so that it covers the entire pane (remember, the glass pane usually should not be opaque). It also lays out the menubar and content pane, even though they are technically part of the layered pane, by placing the menubar (if one has been set) at the top of the pane, with the content pane just below it, filling the remainder of the pane.

The RootPaneContainer Interface

As we've already mentioned, the top-level Swing containers all use the `JRootPane` class as their single child component. In order to make it easier to work with these containers, Swing provides a common interface, that each of them implement. This interface, `RootPaneContainer`, defines methods for accessing the common properties available in `JRootPane`, as well as for the root pane itself. This is what allows the shortcuts we described in the previous section.

The classes that implement this interface typically just delegate the methods to their contained `JRootPane`. For example, `getContentPane()` would typically be implemented like this:

```
public Container getContentPane {  
    return getRootPane().getContentPane();  
}
```

Properties

This interface is made up entirely of accessors for the `JRootPane` and its properties, shown in Table 8-3.

Notice that the root pane's `JMenuBar` is not available in this interface. This is because certain containers (`JWindow`, specifically) don't typically contain menus. Not to say that you couldn't use one if you really wanted to (accessing it from the `JRootPane`), but access to the menu is not directly supported by the interface.

Table 8-3. *RootPaneContainer Properties*

Property	Data Type	get	is	set	bound
contentPane	Container	•		•	
glassPane	Component	•		•	
layeredPane	JLayeredPane	•		•	
rootPane	JRootPane	•			

See also properties from the JComponent class.

The JLayeredPane Class

Though it didn't make much use of it directly, JRootPane introduced a class called JLayeredPane. JLayeredPane is a container that manages its components via layers so that components in the upper layers are painted on top of components in the lower layers. This gives you something that was difficult to get with AWT: complete control over which components are painted on top and which are hidden.

The easiest way to understand how this works is to look at a very simple example.:

```
// SimpleLayers.java
//
import com.sun.java.swing.*;
import java.awt.Color;

public class SimpleLayers {
    public static void main(String[] args) {

        // Create a frame & gets its layered pane
        JFrame f = new JFrame();
        f.addWindowListener(new BasicWindowMonitor());
        JLayeredPane lp = f.getLayeredPane();

        // Create 3 buttons
        JButton top = new JButton();
        top.setBackground(Color.white);
        top.setBounds(20, 20, 50, 50);
        JButton middle = new JButton();
        middle.setBackground(Color.gray);
        middle.setBounds(40, 40, 50, 50);
        JButton bottom = new JButton();
        bottom.setBackground(Color.black);
        bottom.setBounds(60, 60, 50, 50);

        // Place the buttons in different layers
        lp.add(middle, new Integer(2));
```

```

lp.add(top, new Integer(3));
lp.add(bottom, new Integer(1));

// Show the frame
f.setSize(150, 150);
f.setVisible(true);
}
}

```

In this example, we add three colored buttons to a `JLayeredFrame`. The top button is placed in layer 3, the middle in layer 2, and the bottom in layer 1. Recall that the `Component.add()` method takes an `Object` as a second parameter, so we must create `Integer` objects to identify the layers, rather than just passing in `ints`. When we run this example, we see (in Figure 8-6) that the white button (the one with the highest layer—3) is drawn above the gray button (in layer 2), which is drawn above the black button (layer 1). Note that the order in which the buttons were added has no significance.

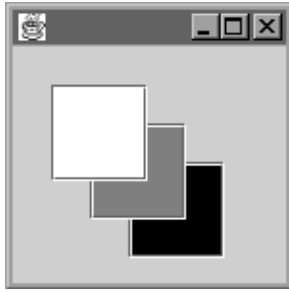


Figure 8-6. *JLayeredFrame* example

The actual values used for the layers are not important, only their relative ordering. We could just as easily have used 10, 20, and 30 as the layer values.

Properties

`JLayeredPane` defines default values for properties listed in Table 8-4.

Table 8-4. *JLayeredPane* Properties

Property	Data Type	get	is	set	bound	Default Value
<code>accessibleContext*</code>	<code>AccessibleContext</code>	•				<code>AccessibleJLayeredPane</code>
<code>layout*</code>	<code>LayoutManager</code>	•		•		<code>null</code>
<code>optimizedDrawingEnabled*</code>	<code>boolean</code>	•				<code>true</code>

See also properties from the `JComponent` class.

The `accessibleContext` property is as expected.

`Layout` is set to `null` by default. This works fine when the pane's layers are containers themselves, each managing the layout of a particular layer, or when only a single component is added to each layer. If multiple components are added to a single layer, however, they will be laid out with no layout manager. This is why the `RootLayout` class described in the `JRootPane` section explicitly lays out the components it adds to a single layer of its layered pane.

`OptimizedDrawingEnabled` is a property defined in `JComponent` that allows a component's children to be drawn more efficiently if they can be guaranteed not to overlap. In `JComponent`, this property is always `true`. In `JLayeredPane`, it is `true` only if the components in the pane do not overlap.

Constants

`JLayeredPane` defines several constants. The first six shown in Table 8-5 (and in Figure 8-7) are `Integer` objects, used to define specific layers available to users of the class.

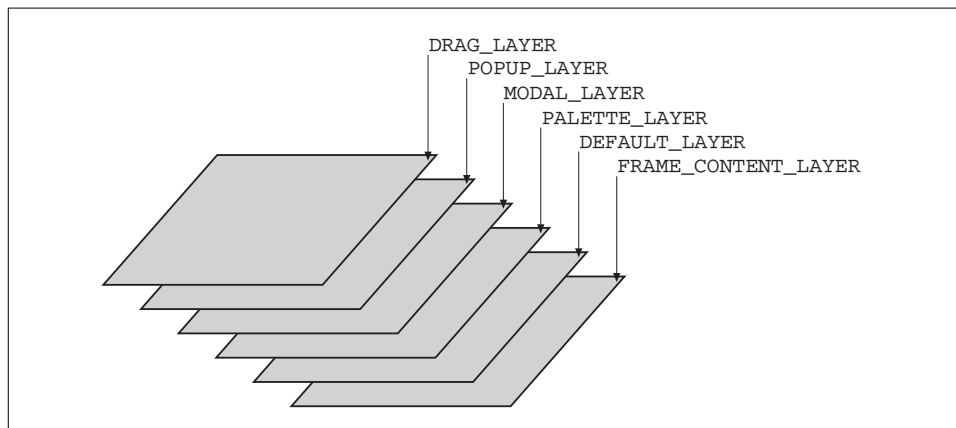


Figure 8-7. Predefined layers

Remember, any number can be used as a layer number; these are just provided as useful defaults. However, it's generally not a good idea to mix your own values with these constants, since there's no guarantee they won't change (this would be very unlikely, but it's definitely poor coding practice to assume the exact values of symbolic constants). Instead, you should choose to either use these constants or define your own layer values.

The last constant in this table, `LAYER_PROPERTY`, is used as a client property name on `JComponents` added to the pane. A property with this name will be set on any

Table 8-5. *JLayeredPane* Constants

Constant	Type	Description
DEFAULT_LAYER	Integer	Used for most components (0)
DRAG_LAYER	Integer	Used when dragging objects on the screen to ensure that they appear on top of everything else as they are being dragged (400)
FRAME_CONTENT_LAYER	Integer	Used only for the content pane and menu bar (-30000)
MODAL_LAYER	Integer	Used to display modal popup windows above other components (200)
PALETTE_LAYER	Integer	Used to display floating toolbars or palettes (100)
POPUP_LAYER	Integer	Used to ensure that popups (including tooltips) are displayed above the components that generate them (300)
LAYER_PROPERTY	String	The name of the layer client property

`JComponent` added to the pane. The property value will be an `Integer` representing the component's layer.

Constructors

```
public JLayeredPane()
```

This constructor creates a new pane with a `null` layout manager.

Adding Components to Layers

The `add()` methods described below (implemented in `java.awt.Container`) are not actually reimplemented in this class, but it's important to understand how they can be used with `JLayeredPane`. In order to gain this understanding, we'll first explain the use of the term *position* with respect to this class.

A component's position in a layer determines the order in which it will be drawn. This is no different from a component's position in a simple container. Components with the lowest position numbers are drawn last (on top). Components with a position of -1 are added with the next highest position number, so they will draw first (on bottom). This is best understood by looking at a quick example. Assume we have three components in a layer at positions 0, 1, and 2. We have:

A B C

Now, if we add D to position 1, we have:

A D B C

Adding E to position -1 yields:

A D B C E

Adding F to position 5 gives us:

A D B C E F

If we paint these components, they will be painted in the following order:

F E C B D A

That is, F will be drawn first (on bottom) and A will be drawn last.

When working with multiple layers, nothing changes. The only difference is that all components in a given layer are painted before any components in the next layer, regardless of their positions within a layer. Note that the ordering of layers places the components in the *highest* numbered layer on top, while the ordering of positions places the component with the *lowest* numbered position on top. So, if we have:

Layer 1: A B (A is at position 0, B is at position 1)

Layer 2: C D

Layer 3: E F

The components will be painted in this order:

B A D C F E

The component (E) with the highest layer (3) and lowest position (0) is painted last (on top), as shown in Figure 8-8.

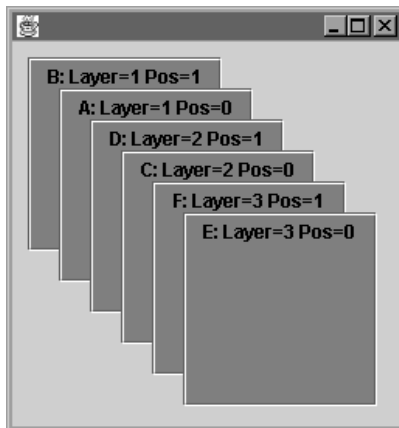


Figure 8-8. Paint order of layered components

Here's how the various versions of `Component.add()` work with `JLayeredPane`. Again, these `add()` methods are *not* reimplemented in `JLayeredPane`, they're

covered here only for the purpose of explaining how they work in this context. Each version of `add()` is explained in terms of how it will call `addImpl()`, a protected method that is implemented in this class and is also described below.

public Component add(Component comp)

Results in a call to `addImpl(comp, null, -1)`.

public Component add(Component comp, int index)

Results in a call to `addImpl(comp, null, index)`.

public void add(Component comp, Object constraints)

Results in a call to `addImpl(comp, constraints, -1)`. The input object should be an `Integer` specifying which layer to add the component to.

public void add(Component comp, Object constraints, int index)

Results in a call to `addImpl(comp, constraints, index)`. The input object should be an `Integer` specifying the layer to add the component to.

public Component add(String name, Component comp)

Should not be used with `JLayeredPane`. If it is, it results in a call to `addImpl(comp, name, -1)`. Since `name` is not an `Integer`, it is ignored.

protected void addImpl(Component comp, Object constraints, int index)

This implementation of `addImpl` checks to see if the input constraint object is an `Integer` and if so, uses it as the component's layer. If the constraint object is `null` (or anything other than an `Integer`), the component's layer is set by calling `getLayer()`, (described later in this chapter).

Layer Management Methods

`JLayeredPane` makes it easy to manipulate layers and the components within them by providing the following methods:

public int getComponentCountInLayer(int layer)

Returns the number of components currently in the specified layer.

public Component[] getComponentsInLayer(int layer)

Returns an array containing the `Components` currently in the specified layer.

public int getIndexOf(Component c)

Returns the absolute index of the given component. This ignores the pane's layers completely. The component with the highest index is the first component painted, meaning it will appear under all other components (which are painted in decreasing order). Since this method ignores the abstractions the layered pane provides, you will not typically use it in application code.

public int getLayer(Component c)

Returns the layer in which the given component has been placed. If the input component is a `JComponent`, the layer is determined by getting its `LAYER_`

PROPERTY client property. If it is not a `JComponent`, it is looked up in an internal hash table used for mapping non-`JComponent`s to layers. In either case, if the layer cannot be determined as described, the `DEFAULT_LAYER` is returned.

public int getPosition(Component c)

Returns a component's position within its layer.

public int highestLayer()

Returns the highest numbered layer in which a child is contained. If there are no children, zero is returned.

public int lowestLayer()

Returns the lowest numbered layer in which a child is contained. If there are no children, zero is returned.

public void moveToBack(Component c)

Moves the specified component to the "back" of its layer.

public void moveToFront(Component c)

Moves the specified component to the "front" of its layer (position 0).

public void remove(int index)

Removes the specified component (the index is an absolute index, not layer-based) from the pane.

public void setLayer(Component c, int layer)

public void setLayer(Component c, int layer, int position)

Set the layer and position (which defaults to -1 in the first case) for the given component and repaints the component. Note that this does not add the component to the pane; `add()` must still be called. Alternatively, a single call to `add(c, new Integer(layer))` or `add(c, new Integer(layer), position)` could be made. If the input component is a `JComponent`, its layer is stored by setting the `LAYER_PROPERTY` client property on the component itself. If not, the component's layer is stored in an internal hash table that maps from non-`JComponent`s to layers.

public void setPosition(Component c, int position)

Sets a component's position within its layer (determined by calling `getLayer(c)`).

Other Public Methods

public void paint(Graphics g)

This method is overridden to explicitly paint the background rectangle if the pane is opaque. It also calls `super.paint()`.

Static Methods

public static int getLayer(JComponent c)

Uses the `LAYER_PROPERTY` to get the layer for a given swing component. Normally, the `getLayer()` instance method should be used.

public static JLayeredPane getLayeredPaneAbove(Component c)

Searches the component hierarchy from the given component upward, returning the first `JLayeredPane` it finds. This allows you to find the layered pane in which a component has been placed. If none is found, it will return `null`.

public static void putLayer(JComponent c, int layer)

Sets a component's layer by assigning a value to its `LAYER_PROPERTY` client property. It does not cause a repaint like the `setLayer()` instance method does. Normally, `setLayer()` should be used.

Protected Methods

In addition to the `addImpl()` method already described, `JLayeredPane` defines the following protected methods:

protected Hashtable getComponentToLayer()

Provides access to an internal table, mapping child components to layers.

protected Integer getObjectForLayer(int layer)

Returns an `Integer` object for the given layer number.

protected int insertIndexForLayer(int layer, int position)

Determines the absolute index for a component to be added to the specified layer at the specified position.

Basic RootPaneContainers

For the rest of this chapter, we'll look at some basic containers (`JFrame`, `JApplet`, and `JWindow`), which implement `RootPaneContainer` and use `JRootPane`. First, we'll take a quick look at a simple interface called `WindowConstants`.

The WindowConstants Interface

`WindowConstants` is a simple interface containing only constants. It is implemented by `JFrame` and `JDialog`.

Constants

The constants defined in `WindowConstants` specify possible behaviors in response to a window being closed. These values are shown in Table 8-6.

Table 8-6. WindowConstants Constants

Constant	Type	Description
DISPOSE_ON_CLOSE	int	Dispose window when closed.
DO_NOTHING_ON_CLOSE	int	Do nothing when closed.
HIDE_ON_CLOSE	int	Hide window when closed.

In the next section, we'll look at a strategy for exiting the application in response to a frame being closed.

The JFrame Class

The most common Swing container for Java applications is the `JFrame`. Like `java.awt.Frame`, `JFrame` provides a top-level window with a title, border, and other platform-specific adornments (e.g., minimize, maximize, and close buttons). Because it uses a `JRootPane` as its only child, working with a `JFrame` is slightly different than working with an AWT `Frame`. An empty `JFrame` is shown in Figure 8-9.



Figure 8-9. JFrame

The primary difference is that calls to `add()` must be replaced with calls to `getContentPane().add()`. In fact, the `addImpl()` method is implemented so that a call made directly to `add()` throws an `Error`. For more information, please refer to the previous sections of this chapter.

Properties

`JFrame` defines the properties shown in Table 8-7.

The `accessibleContext` property is as expected. `ContentPane`, `glassPane`, `layeredPane`, and `JMenuBar` are really properties of `JRootPane`, described earlier in the chapter. `JFrame` provides direct access to these panes, as required by the `RootPaneContainer` interface.

Table 8-7. *JFrame* properties

Property	Data Type	get	is	set	bound	Default Value
<code>accessibleContext*</code>	<code>AccessibleContext</code>	•				<code>AccessibleJFrame</code>
<code>background*</code>	<code>Color</code>	•		•		<code>UIManager.getColor("control")</code>
<code>contentPane*</code>	<code>Container</code>	•		•		from <code>rootPane</code>
<code>defaultCloseOperation</code>	<code>int</code>	•		•		<code>HIDE_ON_CLOSE</code>
<code>glassPane*</code>	<code>Component</code>	•		•		from <code>rootPane</code>
<code>layeredPane*</code>	<code>JLayeredPane</code>	•		•		from <code>rootPane</code>
<code>layout*</code>	<code>LayoutManager</code>	•		•		<code>BorderLayout</code>
<code>JMenuBar*</code>	<code>JMenuBar</code>	•		•		from <code>rootPane</code>
<code>rootPane*</code>	<code>JRootPane</code>	•				<code>JRootPane</code>
<code>title*</code>	<code>String</code>	•	•			<code>" "</code>

See also properties from `java.awt.Frame` class (not covered in this book).

The `defaultCloseOperation` is set to `HIDE_ON_CLOSE`, a value taken from `WindowConstants`. This indicates that closing a `JFrame` window will result in a call to `setVisible(false)`.

The `layout` property is listed here because `JFrame` overrides `setLayout()` to throw an `Error` if an attempt is made to change the layout manager, rather than set the layout manager of the frame's content pane.

The `rootPane` property is set to a new instance of `JRootPane` when the frame is created and cannot be changed (via public methods).

The accessors for the `title` property are inherited from `Frame`. This property can be set in the `JFrame` constructor.

Protected Fields

The following fields are available to subclasses of `JFrame`:

protected AccessibleContext `accessibleContext`

Contains the `AccessibleJFrame` for this frame.

protected JRootPane `rootPane`

Contains the frame's root pane.

protected boolean rootPaneCheckingEnabled

Indicates whether the frame will throw an `Error` if an attempt is made to add components directly to the frame (rather than to its content pane) or to set the layout manager. By default, this is set to `true` once the frame has been built. Subclasses could change this property if necessary.

Constructors

public JFrame()

Creates a new unnamed, invisible frame.

public JFrame(String title)

Creates an invisible frame with the specified name.

User Interface Methods

public void update(Graphics g)

Overrides `Container.update()`, to do nothing but call `paint()`. This is consistent with the implementation of `update()` provided by `JComponent`.

Protected Methods

protected void addImpl(Component comp, Object constraints, int index)

This method (called by `add()`) is overridden to throw an `Error` when an attempt is made to add a component directly to the `JFrame`. The only component allowed to be added is the `JRootPane`, which fills the entire frame (using `BorderLayout.CENTER`).

protected JRootPane createRootPane()

Called by the constructor to create the frame's `JRootPane`.

protected void frameInit()

Called by the constructor to enable key and window events, set the root pane, and set the background color. The last thing this method does is set the `rootPaneCheckingEnabled` field to `true`.

protected boolean isRootPaneCheckingEnabled()

Indicates whether the frame will throw an `Error` if an attempt is made to add a component directly to the frame.

protected void processKeyEvent(KeyEvent e)

Forwards the event to `JComponent`'s `processKeyBindingsForAllComponents` static method.

protected void processWindowEvent(WindowEvent e)

This method first allows the superclass implementation to process the event. It then handles window closing events based on the current default close opera-

tion for the frame. For `HIDE_ON_CLOSE`, the frame is made invisible, for `DISPOSE_ON_CLOSE`, the frame is made invisible and disposed, and for `DO_NOTHING_ON_CLOSE`, nothing is done.

protected void setRootPane(JRootPane root)

Used internally to set the root pane. It temporarily allows components (the root pane) to be added to the frame (keeping `addImpl()` from throwing an `Error`).

protected void setRootPaneCheckingEnabled(boolean enabled)

Sets the `rootPaneCheckingEnabled` field.

Exiting Frames

In many applications, closing the main application frame should cause the program to exit (shutting down the virtual machine). The default implementation, however, is only to hide the frame when it is closed, leaving the VM running with no visible frame. We'll briefly look at two simple ways to get the program to exit when the frame is closed.

The simplest thing to do is to add a `WindowListener` to the frame, calling `System.exit()` in the `windowClosing()` method. Here's a simple example:

```
// FrameClose1.java
//
import com.sun.java.swing.JFrame;
import java.awt.event.*;

public class FrameClose1 {
    public static void main(String[] args) {
        JFrame mainFrame = new JFrame();

        // Exit app when frame is closed.
        mainFrame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent ev) {
                System.exit(0);
            }
        });

        mainFrame.setSize(320, 240);
        mainFrame.setVisible(true);
    }
}
```

If you get tired of writing this same block of code in every frame that needs to close properly, you might want to use an extension of `JFrame` that supports this feature. Here's one possible implementation of such a class:

```
// ExitFrame.java
```

```
//
import com.sun.java.swing.JFrame;
import java.awt.event.WindowEvent;

// A very simple extension of JFrame that adds another option for the
// defaultCloseOperation called EXIT_ON_CLOSE. This is the default
// for this class, but it can be changed just as it is changed with JFrame.
public class ExitFrame extends JFrame {

    // Exit the VM when the frame is closed
    public static int EXIT_ON_CLOSE = 100;

    protected int closeOp;

    public ExitFrame() {
        super();
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    public ExitFrame(String title) {
        super(title);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    // Overrides JFrame implementation to store the operation locally
    public void setDefaultCloseOperation(int operation) {
        super.setDefaultCloseOperation(operation);
        closeOp = operation;
    }

    // Override JFrame implementation to exit if the close operation is set
    // to EXIT_ON_CLOSE
    protected void processWindowEvent(WindowEvent e) {
        if (e.getID() == WindowEvent.WINDOW_CLOSING) {
            if (closeOp == EXIT_ON_CLOSE)
                System.exit(0);
        }
        super.processWindowEvent(e);
    }
}
```

You can use this class just like you'd use a `JFrame`. If you don't want the program to exit when the user closes the frame, just change the default close action to one of the values defined in `WindowConstants`.

A more common strategy is to display a dialog box asking something like *are you sure?* when the user tries to close the frame. `JOptionPane` (which we'll get to in detail in Chapter 10, *Swing Dialogs*) makes this very easy to do. All you need to do is reimplement your `processWindowEvent()` method like this:

```
protected void processWindowEvent(WindowEvent e) {
    if (e.getID() == WindowEvent.WINDOW_CLOSING) {
        if (closeOp == MAYBE_EXIT_ON_CLOSE) {
            int exit = JOptionPane.showConfirmDialog(this, "Are you sure?");
            if (exit == JOptionPane.YES_OPTION) {
                System.exit(0);
            }
        }
    }
    super.processWindowEvent(e);
}
```

The JWindow Class

JWindow is an extension of `java.awt.Window` that uses a `JRootPane` as its single component. Other than this core distinction, JWindow does not change anything defined by the `Window` class.

In AWT, one common reason for using the `Window` class was to create a popup menu. Since Swing explicitly provides a `JPopupMenu` class (see Chapter 14, *Menus and Toolbars* with `JPopup`), there is no need to extend `JWindow` for this purpose. The only time you'll use `JWindow` is if you've got something that needs to be displayed in its own window without the adornments added by `JFrame`. Remember, this means that the window can only be moved or closed programmatically (or via the user's platform-specific window manager controls, if available).

One possible use for `JWindow` would be to display a "splash" screen when an application is starting up. Many programs display such a screen, possibly containing copyright information, resource loading status, etc. Here's such a program.

```
// Splash.java
//
import com.sun.java.swing.*;
import java.awt.*;

public class Splash {
    public static void main(String[] args) {

        // Throw a nice little title page up on the screen first . . .
        showSplash(10000);

        System.exit(0); // replace with application code!
    }

    // A simple little method to show a title screen in the
    // center of the screen for a given amount of time.
    public static void showSplash(int duration) {
        JWindow splash = new JWindow();
```

```

JPanel content = (JPanel)splash.getContentPane();

// set the window's bounds, centering the window
int width = 240;
int height = 120;
Dimension screen = Toolkit.getDefaultToolkit().getScreenSize();
int x = (screen.width-width)/2;
int y = (screen.height-height)/2;
splash.setBounds(x,y,width,height);

// build the splash screen . . .
JLabel label = new JLabel(new ImageIcon("splash.gif"));
JLabel copyrt = new JLabel
    ("Copyright 1998, PussinToast Inc.", JLabel.CENTER);
copyrt.setFont(new Font("Sans-Serif", Font.BOLD, 12));
content.add(label, BorderLayout.CENTER);
content.add(copyrt, BorderLayout.SOUTH);
content.setBorder(BorderFactory.createLineBorder(Color.red, 10));

// display it . . .
splash.setVisible(true);

// Wait a little while, maybe while loading resources
try { Thread.sleep(duration); } catch (Exception e) {}

splash.setVisible(false);
}
}

```

All this program does is create a `JWindow` containing a pair of labels and display it in the center of the screen. In a real application, the title screen might be displayed while various system resources are being loaded (consider using a `ProgressMonitor` in this case). When run, this example displays a simple window in the center of the screen, as shown in Figure 8-10.



Figure 8-10. *JWindow* used as a splash screen

Properties

`JWindow` defines the properties shown in Table 8-8.

Table 8-8. *JWindow* properties

Property	Data Type	get	is	set	bound	Default Value
<code>accessibleContext*</code>	<code>AccessibleContext</code>	•				<code>AccessibleJWindow</code>
<code>contentPane*</code>	<code>Container</code>	•		•		from <code>rootPane</code>
<code>glassPane*</code>	<code>Component</code>	•		•		from <code>rootPane</code>
<code>layeredPane*</code>	<code>JLayeredPane</code>	•		•		from <code>rootPane</code>
<code>layout*</code>	<code>LayoutManager</code>	•		•		<code>BorderLayout</code>
<code>rootPane*</code>	<code>JRootPane</code>	•				<code>JRootPane</code>

See also the `java.awt.Window` class (not covered in this book)

The `accessibleContext` property is as expected. `ContentPane`, `glassPane`, and `layeredPane` are really properties of `JRootPane`, as described earlier in the chapter. Direct access is provided for convenience. Note that unlike `JFrame` (and `JApplet`, below), `JWindow` does not provide direct access to the root pane's menubar. This is just an indication of `JWindow`'s intended usage. If you have some compelling reason to display a menubar on a `JWindow`, you can always access it via the root pane.

The `layout` property is listed here because `JWindow` overrides `setLayout()` to throw an `Error` if an attempt is made to change the layout manager, rather than set the layout manager of the window's content pane.

The `rootPane` property is set to a new instance of `JRootPane` when the frame is created and cannot be changed via public methods.

Protected Fields

The following fields are available to subclasses of `JWindow`:

protected AccessibleContext accessibleContext

Contains the `AccessibleJWindow` for this window.

protected JRootPane rootPane

Contains the window's root pane.

protected boolean rootPaneCheckingEnabled

Indicates whether the window will throw an `Error` if an attempt is made to add components directly to the window (rather than to its content pane) or to set the layout manager. By default, this is set to `true` once the window has been built. Subclasses could change this property if necessary.

Constructors

public JWindow()

Creates a new, invisible window associated with no particular owner. This makes use of `SwingUtilities.getSharedOwnerFrame()`.

public JWindow(JFrame frame)

Creates a new, invisible window associated with the given frame.

Protected Methods

protected void addImpl(Component comp, Object constraints, int index)

This method (called by `add()`) is overridden to throw an `Error` when an attempt is made to add a component directly to the `JWindow`. The only component allowed to be added is the `JRootPane`, which fills the entire window (using `BorderLayout.CENTER`).

protected JRootPane createRootPane()

Called by the constructor to create the window's `JRootPane`.

protected boolean isRootPaneCheckingEnabled()

Indicates whether the window will throw an `Error` if an attempt is made to add a component directly to the window.

protected void setRootPane(JRootPane root)

Used internally to set the root pane.

protected void setRootPaneCheckingEnabled(boolean enabled)

Sets the `rootPaneCheckingEnabled` field.

protected void windowInit()

Called by the constructor set the root pane and set the `rootPaneCheckingEnabled` field to `true`.

The JApplet Class

`JApplet` is a simple extension of `java.applet.Applet`, for use when creating Swing programs designed to be used in a web browser (or `appletviewer`). As a direct subclass of `Applet`, `JApplet` is used in much the same way, with the `init()`, `start()`, and `stop()` methods still playing critical roles. The primary thing `JApplet` provides over `Applet` is the use of a `JRootPane` as its single display component. The properties and methods described below should look a lot like those described in the previous sections on `JFrame` and `JWindow`. Figure 8-11 shows a `JApplet` running in `appletviewer`.



Figure 8-11. A JApplet running in JDK appletviewer

Hiding the Warning Message

At the time of this writing, the current popular browsers do not allow applets to access the system event queue. As a result, a warning message is printed to the Java console, indicating that the applet attempted to access the system event queue and failed. If you find this warning sufficiently annoying, Swing provides a workaround that allows you to suppress it. Just implement a constructor for your applet with the following code:

```
getRootPane().putClientProperty("defeatSystemEventQueueCheck", Boolean.TRUE);
```

In AWT, applets rarely (if ever) had constructors. With Swing, a constructor (which must have no arguments) is a good place to set client properties like this one.

Threading Issues

Technically speaking, all UI setup done in a JApplet should be done in the event thread, because applets are started in an existing virtual machine and are technically just panels in a larger application. To deal with this issue, simply wrap any UI-related setup code using the `SwingUtilities.invokeLater()` or `invokeAndWait()` constructs. Here's a sample applet that uses this technique: we also use a constructor to suppress the warning message.

```
// SimpleApplet.java
//
import com.sun.java.swing.*;
import java.awt.*;

public class SimpleApplet extends JApplet {
    public SimpleApplet() {

        // supress warning msg
        getRootPane().putClientProperty("defeatSystemEventQueueCheck",
```

```

        Boolean.TRUE);
    }

    public void init() {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() { // run in the event thread . . .
                JPanel p = new JPanel();
                p.setLayout(new GridLayout(2, 2, 2, 2));
                p.add(new JLabel("Username"));
                p.add(new JTextField());
                p.add(new JLabel("Password"));
                p.add(new JPasswordField());
                Container content = getContentPane();
                content.setLayout(new GridBagLayout()); // used to center the panel
                content.add(p);
                validate();
            }
        });
    }
}

```

For more information on threading issues in Swing, please see Chapter 1, *Introducing Swing* (for an introduction) and Chapter 28, *Swing Under the Hood*.

Properties

JApplet defines the properties and default values shown in Table 8-9.

Table 8-9. JApplet properties

Property	Data Type	set	is	set	bound	Default Value
accessibleContext*	AccessibleContext	•				AccessibleJApplet
contentPane*	Container	•		•		from rootPane
glassPane*	Component	•		•		from rootPane
layeredPane*	JLayeredPane	•		•		from rootPane
layout*	LayoutManager	•		•		BorderLayout
JMenuBar*	JMenuBar	•		•		from rootPane
rootPane*	JRootPane	•				JRootPane

See also the java.applet.Applet class (not covered in this book)

The accessibleContext property is as expected. ContentPane, glassPane, layeredPane, and JMenuBar are really properties of JRootPane, as described earlier in the chapter. Direct access is provided to them for convenience.

The `layout` property is listed here because `JApplet` overrides `setLayout()` to throw an `Error` if an attempt is made to change the layout manager, rather than set the layout manager of the applet's content pane.

The `rootPane` property is set when the applet is created. It cannot be changed via public methods.

Protected Fields

The following fields are available to subclasses of `JApplet`.

protected AccessibleContext accessibleContext

Contains the `AccessibleJApplet` for this window.

protected JRootPane rootPane

Contains the applet's root pane.

protected boolean rootPaneCheckingEnabled

Indicates whether the applet will throw an `Error` if an attempt is made to add components directly to the applet (rather than to its content pane) to set the layout manager. By default, this is set to `true` once the applet has been built. Subclasses could change this property if necessary.

Constructors

public JApplet()

Creates a new applet. This is how browsers and appletviewers create new applets. If you supply a constructor with an applet, perhaps to disable event queue checking, remember that browsers expect an applet constructor to have no arguments.

User Interface Methods

public void update(Graphics g)

Overrides `Container.update()` to do nothing but call `paint()`. This is consistent with the implementation of `update()` provided by `JComponent` (and the implementation used by `JFrame`).

Notify Methods

public void addNotify()

This method (defined in `java.awt.Component`) is called when the applet is added to a container (such as the appletviewer's main frame). It is overridden to initialize the system event queue, call its superclass implementation, and enable key events.

public void removeNotify()

This method (defined in `java.awt.Component`) is called when the applet is removed from a container. This implementation calls the superclass implementation and removes the applet timer.

Protected Methods

protected void addImpl(Component comp, Object constraints, int index)

This method (called by `add()`) is overridden to throw an `Error` when an attempt is made to add a component directly to the `JApplet`. The only component allowed to be added is the `JRootPane`, which fills the entire applet (using `BorderLayout.CENTER`).

protected JRootPane createRootPane()

Called by the constructor to create the applet's `JRootPane`.

protected boolean isRootPaneCheckingEnabled()

Indicates whether the applet will throw an `Error` if an attempt is made to add a component directly to the applet.

protected void processKeyEvent(KeyEvent e)

Forwards the event to `JComponent`'s `processKeyBindingsForAllComponents` static method.

protected void setRootPane(JRootPane root)

Used internally to set the root pane. It temporarily allows components (the root pane) to be added to the applet, ensuring that `addImpl()` will not throw an `Error`.

protected void setRootPaneCheckingEnabled(boolean enabled)

Sets the `rootPaneCheckingEnabled` field.