In this chapter:
- *Managing a Desktop*
- *Building A Desktop*

9

Internal Frames

Managing a Desktop

Certain GUI applications need to simulate a desktop environment by allowing multiple "frames" to be displayed within a single root window. These frames typically look like the normal frames you'd see on a real desktop, but are not actually known to the window manager, because they are not really windows in the normal sense of the term. For some types of applications (word processors, IDEs, etc.), this can be a very powerful approach to UI design.

In this chapter, we'll look at a collection of classes Swing provides to allow you to create this type of application in Java. At the end of the chapter, we'll provide a large sample program that shows how to implement a variety of useful features.

Overview

Before looking at each of the classes involved in the Swing desktop/internal frame model, we'll take a moment for an overview of how they all work together. Figure 9-1 shows the relationships between the classes we'll be covering in this chapter.

A `JInternalFrame` is a container that looks much like a `JFrame`. The key difference is that internal frames can only exist within some other Java container. `JInternalFrame` implements the following six interfaces: `Accessible`, `MouseListener`, `MouseMotionListener`, `WindowConstants`, `RootPaneContainer`, `ComponentListener`.

Each internal frame keeps a reference to an instance of the static inner class called `JDesktopIcon`. Like real frames, `JInternalFrames` can be iconified. `JDesktop-`

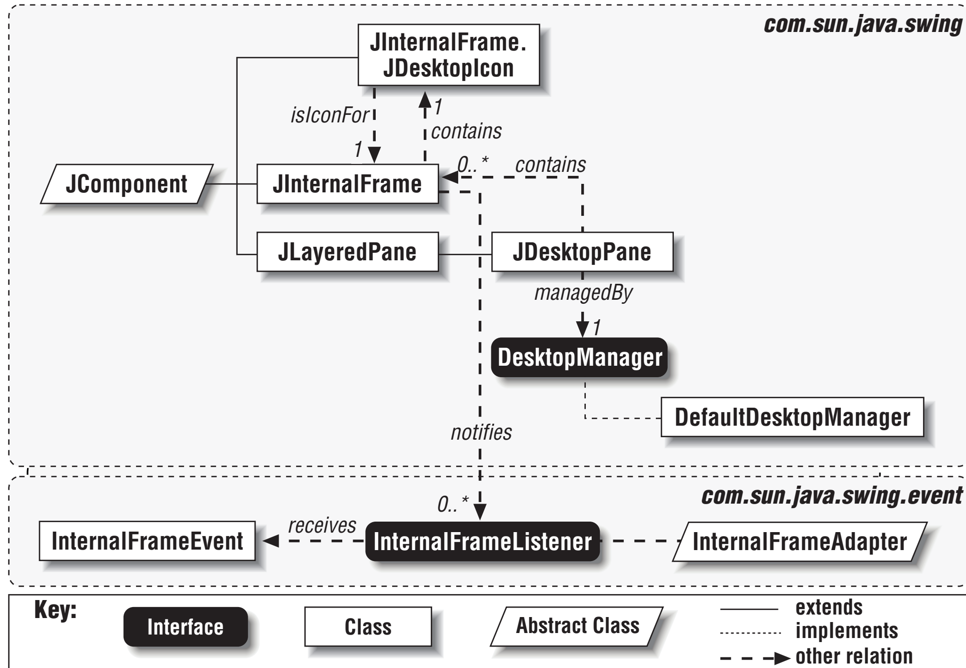

Figure 9-1. Internal Frame and Desktop class diagram

Icon is the class responsible for taking the place of the frame when it gets iconified.

Though it is not required, JInternalFrames are typically used inside of a JDesktopPane. JDesktopPane is an extension of JLayeredPane that adds direct support for managing a collection of JInternalFrames in layers. JDesktopPane uses an object called a DesktopManager to control how different behavior, like iconification or maximization, is carried out. A default implementation of this interface, DefaultDesktopManager, is provided. We'll see how all of this functionality is broken out as we cover the various classes and interfaces involved.

One more thing to notice about Figure 9-1 is that JInternalFrame supports a new type of listener called InternalFrameListener. This interface contains methods that match those defined by the AWT WindowListener class, but have slightly different names and take InternalFrameEvents, rather than WindowEvents, as input.

The JInternalFrame Class

JInternalFrame is a powerful addition to Java, providing the ability to create lightweight frames that exist inside other components. An internal frame is

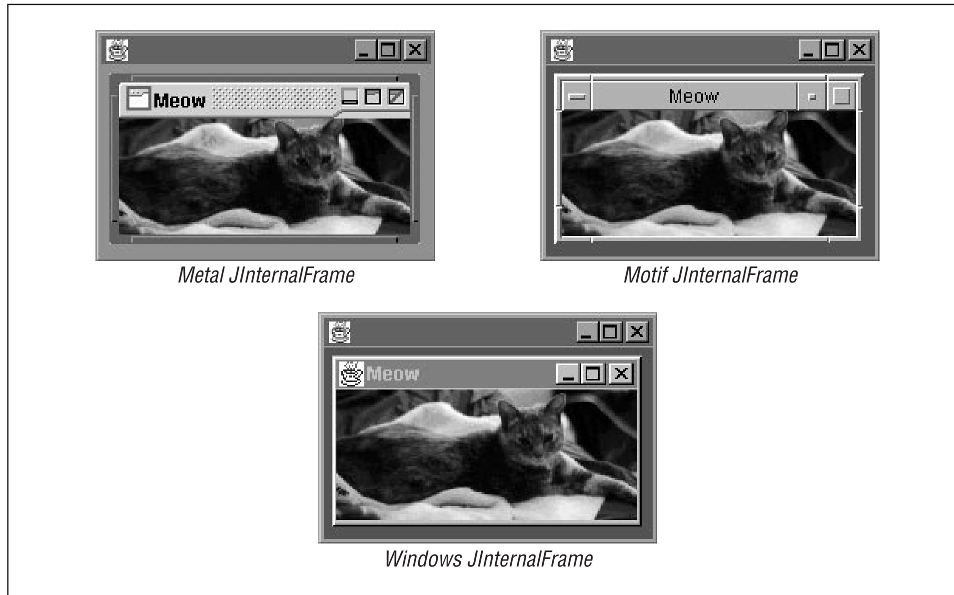

Figure 9-2. JInternalFrames

managed entirely within some other Java container, just like any other component, allowing the program complete control over iconification, maximization, resizing, etc. Despite looking like "real" windows, the underlying windowing system knows nothing of the existence of internal frames.* Figure 9-2 shows what an internal frame looks like in the different look-and-feels.

There's quite a lot to discuss about `JInternalFrames`, but most of their power comes when they are used inside a `JDesktopPane`. In this section, we will give a quick overview of the properties, constructors, and methods available in `JInternalFrame`, but we'll leave the more detailed discussion of using internal frames to the sections that follow.

Properties

`JInternalFrame` defines the properties and default values shown in Table 9-1.

The `accessibleContext` property is as expected. The `background` and `foreground` properties are delegated to the frame's content pane.

There are three pairs of properties that indicate whether or not something can be done to a frame and whether or not that thing is currently done to the frame.

* Note that `JInternalFrame` extends `JComponent`, not `JFrame` or `Frame`, so this statement should seem logical.

Table 9-1. JInternalFrame Properties

Property	Data Type	get	is	set	bound	Default Value
UI	InternalFrameUI	•		•		*from L&F*
UIClassID*	String	•				"InternalFrameUI"
accessibleContext*	AccessibleContext	•				AccessibleJInter-nalFrame
background*	Color	•		•		*from contentPane*
closable	boolean		•	•		false
closed[a]	boolean		•	•	•	false
contentPane*	Container	•		•	•	*from rootPane*
defaultCloseOperation	int	•		•		HIDE_ON_CLOSE
desktopIcon	JInternal-Frame.JDesktopIcon	•		•		JInternal-Frame.JDesktopIcon
desktopPane	JDesktopPane	•				null
foreground*	Color	•		•		*from contentPane*
frameIcon	Icon	•		•		null
glassPane*	Component	•		•	•	*from rootPane*
icon[a]	boolean		•	•	•	false
iconifiable	boolean		•	•		false
layer	int	•				0
layeredPane*	JLayeredPane	•		•	•	*from rootPane*
maximizable	boolean		•	•		false
maximum[a]	boolean		•	•	•	false
menuBar*	JMenuBar	•		•	•	*from rootPane*
resizable	boolean		•	•		false
rootPane*	JRootPane	•			•	JRootPane
selected[a]	boolean		•	•	•	false
title	String	•		•	•	""
warningString	String	•				null

See also properties from the JComponent class.

[a] These properties are unique in that they are the only properties in Swing which are constrained. The `set()` methods for each of them fire `PropertyChangeEvents` to any registered `VetoableProperty-ChangeListeners`. Consequently, calls to these `set()` methods must be wrapped in a try-catch block, checking for `PropertyVetoException`. See the example at the end of the chapter for more information.

They are: `closable/closed`, `iconifiable/icon`, and `maximizable/maximum`. Note that `closed`, `icon`, and `maximum` are constrained properties.

The contentPane, glassPane, layeredPane, and menuBar properties come from the RootPaneContainer interface and are taken directly from the frame's JRootPane. The rootPane property is set to a new JRootPane when the frame is constructed.

The value of the defaultCloseOperation property defaults to WindowConstants.HIDE_ON_CLOSE. This implies that when the frame is closed, its setClosed() method will be called. The frame could be reopened at a later time.

The desktopIcon reflects how the frame will be displayed when iconified. A JDesktopIcon (which leaves the rendering to the L&F) is created for the frame when it is instantiated. The desktopPane property provides a convenient way to access the JDesktopPane containing the frame, if there is one.

FrameIcon is the icon painted inside the frame's titlebar (usually on the far left). By default, there is no icon. However, the basic look-and-feel checks to see if a frameIcon has been set and, if not, paints the "java cup" icon. This explains why an icon appears in the Windows L&F frame shown in Figure 9-2, but not in the others (which provide their own paint() implementations, rather than using the one provided by the basic L&F).*

The layer property indicates the frame's current layer, if it has been placed in a JLayeredPane. The resizable property indicates whether or not the frame can be resized by dragging its edges or corners, and selected indicates whether or not the frame has been selected (this will typically determine the color of the titlebar). Note that selected is a constrained property. Title contains the string to appear on the titlebar.

The UI property holds the current L&F implementation for the frame, and UIClassID reflects the class ID for internal frames.

Finally, the warningString property, which is always null, is used to specify the string that should appear in contexts where the frame might be insecure. This is the technique used by java.awt.Window to display a string like "Warning: Applet Window" when a Java window is displayed from an applet. Since JInternalFrames are always fully enclosed by some other top-level container, this property is always null.

Events

Table 9-2 shows the types of events fired by JInternalFrames.

The following standard methods are provided for working with events.

* The BasicLookAndFeel is an abstract base class from which all the Swing L&Fs extend. For more information, see Chapter 26.

Table 9-2. JInternalFrame Events

Event	Description
`InternalFrameEvent`	The frame's state has changed. See the description of `InternalFrameListener` later in this chapter for more details.

public synchronized void addInternalFrameListener(InternalFrameListener l)
public synchronized void removeInternalFrameListener(InternalFrameListener l)
protected void processEvent(AWTEvent e)

This last method makes the appropriate listener method calls if the input event is an `InternalFrameEvent`. In the case of an `InternalFrameEvent.INTERNAL_FRAME_CLOSED` event ID, the default close action is performed.

Like all the other Swing classes, `JInternal Frame` fires `PropertyChangeEvents` when the value of any bound property is changed. `JInternalFrame` is unique in that it is the only Swing class that uses vetoable changes for some properties (`closed`, `icon`, `maximum`, and `selected`).

Constants

Table 9-3 shows the constants defined in this class. They are all strings and contain the names of the bound properties.

Table 9-3. JInternalFrame Constants

Constant	Property
`CONTENT_PANE_PROPERTY`	indicates that the content pane has changed
`GLASS_PANE_PROPERTY`	indicates that the glass pane has changed
`IS_CLOSED_PROPERTY`	indicates that the frame has been opened or closed
`IS_ICON_PROPERTY`	indicates that the frame as been iconified or deiconified
`IS_MAXIMUM_PROPERTY`	indicates that the frame has been maximized or minimized
`IS_SELECTED_PROPERTY`	indicates that the frame has been selected or deselected
`LAYERED_PANE_PROPERTY`	indicates that the layered pane has changed
`MENU_BAR_PROPERTY`	indicates that the menubar has changed
`ROOT_PANE_PROPERTY`	indicates that the root pane has changed
`TITLE_PROPERTY`	indicates that the frame's title has changed

Protected Fields

protected boolean closable
protected JInternalFrame.JDesktopIcon desktopIcon
protected Icon frameIcon
protected boolean iconable
protected boolean isClosed
protected boolean isIcon
protected boolean isMaximum
protected boolean isSelected
protected boolean maximizable
protected boolean resizable
protected JRootPane rootPane
protected String title

These fields are used to hold the values of many of the properties listed in Table 9-1. Subclasses should access them through the accessor methods, rather than using these fields directly.

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). By default, this is set to `true` once the frame has been built. Subclasses could change this property if necessary.

Constructors

`JInternalFrame` provides constructors that allow several of its boolean properties to be set at creation time. By default, `resizable`, `closable`, `maximizable`, and `iconifiable` are all set to `false`.

public JInternalFrame()
public JInternalFrame(String title)

Create a new frame with all four properties set to false.

public JInternalFrame(String title, boolean resizable)
public JInternalFrame(String title, boolean resizable, boolean closable)
public JInternalFrame(String title, boolean resizable, boolean closable, boolean maximizable)
public JInternalFrame(String title, boolean resizable, boolean closable, boolean maximizable, boolean iconifiable)

Allow one to four of the frame's boolean properties to be set at creation time.

JLayeredPane Methods

These methods are applicable only if the frame is contained by a `JLayeredPane` (otherwise, they do nothing).

public void moveToBack()
public void toBack()

Call the containing layered pane's `moveToBack()` method, causing the frame to be the first (bottom) component painted in its layer.

public void moveToFront()
public void toFront()

Call the containing layered pane's `moveToFront()` method, causing the frame to be the last (top) component painted in its layer.

MouseListener & MouseMotionListener Methods

`JInternalFrame` implements `MouseListener` and `MouseMotionListener` so it can control how mouse events are passed to its contained components. The frame is added as a listener to the root pane's glass pane.* The glass pane is made visible when the frame is not selected, forcing all mouse events to be captured by the internal frame's listeners. Once the internal frame is selected, the glass pane is made invisible and the listeners are removed, meaning that the frame's components will receive mouse events directly. All this is done so that the frame can be clicked anywhere to select it, without actually causing the underlying components to be clicked.

public void mousePressed(MouseEvent e)

This method is forwarded to the frame's UI, allowing the frame to be selected. Once the frame has been selected, the glass pane is made invisible and the mouse and mouse-motion listeners are removed. The glass pane becomes visible again when the frame becomes unselected.

public void mouseClicked(MouseEvent e)
public void mouseReleased(MouseEvent e)
public void mouseDragged(MouseEvent e)

These methods have empty implementations, meaning that the events will be ignored by the frame.

public void mouseEntered(MouseEvent e)
public void mouseExited(MouseEvent e)
public void mouseMoved(MouseEvent e)

These methods are forwarded to the appropriate underlying component.

ComponentListener Methods

`JInternalFrame` implements the `ComponentListener` interface so that maximized internal frames can be resized when the size of the frame's parent changes.

* For more information on the glass pane, see Chapter 8, *Swing Containers*.

The `setMaximum()` method adds the frame as a `ComponentListener` of the desktop when `maximum` is set to `true`, and removes the listener when `maximum` is set to `false`.

public void componentHidden(ComponentEvent e)
public void componentMoved(ComponentEvent e)
public void componentShown(ComponentEvent e)
These methods do nothing.

public void componentResized(ComponentEvent e)
Called when the component containing the `JInternalFrame` is resized, and the internal frame is maximized. The implementation of this method resizes the internal frame to fill the component.

Miscellaneous Public Methods

public void dispose()
Makes the frame invisible, unselected, and closed.

public void pack()
Works like `Frame`'s `pack()` method, causing the frame to be resized according to the preferred size of its components.

public void reshape(int x, int y, int width, int height)
Calls its superclass implementation and then forces a repaint of the frame, so that decorations such as the title bar will be painted.

public void show()
Makes the frame visible and selects it, bringing it to the front of its layer.

public void updateUI()
Called to indicate that the L&F for the frame has changed.

Protected Methods

protected void addImpl(Component comp, Object constraints, int index)
This is the method called by the various `add()` methods. If `rootPaneCheckingEnabled` is set to `true`, this method throws an `Error`, indicating that components should be added to the frame's content pane, not to the frame itself.

protected JRootPane createRootPane()
Used by the constructor to create a new `JRootPane`.

protected boolean isRootPaneCheckingEnabled()
Indicates the current value of `rootPaneCheckingEnabled`.

protected void setRootPane(JRootPane root)
Called by the constructor to set the frame's root pane.

The JInternalFrame.JDesktopIcon Class

JDesktopIcon is a static inner class of JInternalFrame, used to provide an iconified view of a frame. JInternalFrame instantiates a JDesktopIcon when the frame is created. The class extends JComponent and, like other Swing components, leaves all details of its visual appearance to its UI delegate.

Note that this class has no relation at all to the Swing Icon interface.

NOTE You should not work with the JDesktopIcon class directly—the javadoc for this inner class indicates that it will go away in a future Swing release. We are including this brief description of the class for completeness until the change is made.

Properties

JDesktopIcon defines the properties shown in Table 9-4.

Table 9-4. JDesktopIcon Properties

Property	Data Type	get	is	set	bound	Default Value
UI	DesktopIconUI	•		•		*from L&F*
UIClassID*	String	•				"DesktopIconUI"
accessibleContext*	AccessibleContext	•				AccessibleJDesktop-Icon
desktopPane	JDesktopPane	•				*from internal frame*
internalFrame	JInternalFrame	•		•		*from constructor*

See also properties from the JComponent class.

AccessibleContext is as expected. The desktopPane property simply provides convenient, direct access to the JDesktopPane, containing the icon, if applicable. This property comes directly form the icon's internal frame. The internalFrame property reflects the icon's tight coupling with the JInternalFrame class. The icon's frame is set when the icon is constructed and typically should not be changed. As usual, the UI provides access to the object's UI implementation, and UIClassID is set to the expected value.

Constructors

public JDesktopIcon(JInternalFrame f)
Creates an icon for the specified internal frame.

Methods

There is only one method other than the accessors for the icon's properties.

public void updateUI()

This method is called to indicate that the icon's L&F should be updated.

The InternalFrameEvent Class

As we described in the previous section, `JInternalFrames` fire `InternalFrameEvents` when the state of the frame changes. This is a standard `AWTEvent` subclass, providing a number of constants to define the type of change that was made to the frame.

Constants

Table 9-5 shows constants defined as possible values for the event ID.

Table 9-5. InternalFrameEvent Constants

Constant	Type	Description
INTERNAL_FRAME_ACTIVATED	int	the frame has been activated
INTERNAL_FRAME_CLOSED	int	the frame has been closed (sent any time the frame is closed)
INTERNAL_FRAME_CLOSING	int	the frame is about to be closed (sent when the user clicks the closebox on the frame)
INTERNAL_FRAME_DEACTIVATED	int	the frame has been deactivated
INTERNAL_FRAME_DEICONIFIED	int	the frame has been restored from an icon
INTERNAL_FRAME_ICONIFIED	int	the frame has been iconified
INTERNAL_FRAME_OPENED	int	the frame has been opened
INTERNAL_FRAME_FIRST	int	the first integer value used to represent the above event IDs
INTERNAL_FRAME_LAST	int	the last integer value used to represent the above event IDs

Constructors

public InternalFrameEvent(JInternalFrame source, int id)

Creates a new event. The *id* should be taken from the list of constants provided above.

Methods

public String paramString()

This method, used by `toString()`, returns a parameter string describing the event.

The InternalFrameListener Interface

`JInternalFrame` fires `InternalFrameEvents` to registered `InternalFrameListeners`. This interface defines the following set of methods (which have a one-to-one correspondence to the methods in the `java.awt.event.WindowListener` interface).

Methods

All of these methods except `internalFrameClosing()`, are called by the `JInternalFrame` when its properties are changed.

public abstract void internalFrameActivated(InternalFrameEvent e)
The frame has been activated, typically meaning that it will gain focus and be brought to the front.

public abstract void internalFrameClosed(InternalFrameEvent e)
The frame has been closed.

public abstract void internalFrameClosing(InternalFrameEvent e)
The frame is closing. This is actually called by the L&F when the close button is clicked.

public abstract void internalFrameDeactivated(InternalFrameEvent e)
The frame has been deactivated.

public abstract void internalFrameDeiconified(InternalFrameEvent e)
The frame has been restored from an icon.

public abstract void internalFrameIconified(InternalFrameEvent e)
The frame has been reduced to an icon.

public abstract void internalFrameOpened(InternalFrameEvent e)
A previously closed frame has been opened.

The InternalFrameAdapter Class

This class follows the standard AWT 1.1 listener/adapter pattern by providing empty implementations of the seven methods defined in the `InternalFrameListener` interface. If you are only interested in certain types of events, you can create a subclass of this adapter that implements only the methods you care about.

Methods

The following methods have empty implementations in this class:

public void internalFrameActivated(InternalFrameEvent e)
public void internalFrameClosed(InternalFrameEvent e)
public void internalFrameClosing(InternalFrameEvent e)

public void internalFrameDeactivated(InternalFrameEvent e)
public void internalFrameDeiconified(InternalFrameEvent e)
public void internalFrameIconified(InternalFrameEvent e)
public void internalFrameOpened(InternalFrameEvent e)

The JDesktopPane Class

`JDesktopPane` is an extension of `JLayeredPane`, which uses a `DesktopManager` to control the placement and movement of frames. Figure 9-3 shows what `JDesktopPane` looks like in the different look-and-feels. Like its superclass, `JLayeredPane` has a `null` layout manager. Components added to it must be placed at absolute locations with absolute sizes, because it is intended to be used to house `JInternalFrames`, which rely on the user to determine their placement.

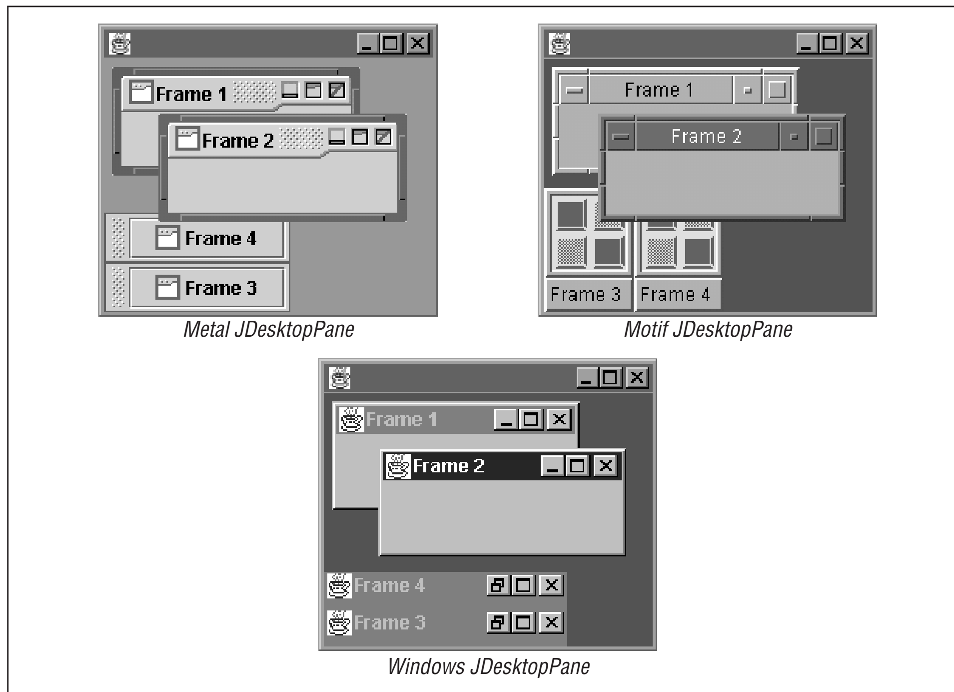

Figure 9-3. JDesktopPanes in the different look-and-feels

Another reason for using `JDesktopPane` is to allow popup dialog boxes to be displayed using `JInternalFrames`. This is discussed in detail in the next chapter.

Properties

Table 9-6 shows the properties defined by `JDesktopPane`.

Table 9-6. JDesktopPane properties

Property	Data Type	get	is	set	bound	Default Value
UI	DesktopPaneUI	•		•		*from L&F*
UIClassID*	String	•				"DesktopPaneUI"
accessibleContext*	AccessibleContext	•				AccessibleJDesktop-Pane
allFrames	JInternalFrame[]	•				*empty array*
desktopManager	DesktopManager	•		•		*from L&F*
opaque*	boolean		•	•		true

See also properties from the JLayeredPane class.

The accessibleContext property is as expected. AllFrames provides access to all JInternalFrames contained by the desktop. The desktopManager property holds the DesktopManager object supplied by the pane's L&F. We'll cover the responsibilities of the DesktopManager in the next section. The opaque property defaults to true for JDesktopPanes and isOpaque() is overridden so that it always returns true. UI contains the DesktopPaneUI implementation, and UIClassID contains the class ID for JDesktopPane.

One "property" not shown here is the allFramesInLayer property. Since the index to this property is a layer value, not a traditional array index, we don't list this as a property, and will discuss it later.

Constructors

public JDesktopPane()

Creates a new desktop and calls updateUI(), resulting in the L&F implementation installing a DesktopManager.

Methods

public JInternalFrame[] getAllFramesInLayer(int layer)

Returns all frames that have been added to the specified layer. This includes frames that have been iconified.

public void updateUI()

This method is called to indicate that the L&F for the desktop should be set.

The DesktopManager Interface

This is the interface responsible for much of the management of internal frames contained by JDesktopPanes. It allows a look-and-feel to define exactly how it

wants to manage things such as frame activation, movement, and iconification. Most of the methods in `InternalFrameUI` implementations should delegate to a `DesktopManager` object. As described earlier, `DesktopManagers` are contained by `JDesktopPane` objects and are intended to be set by the L&F.

Methods

The majority of the methods in this interface act on a given `JInternalFrame`. However, those methods that could be applied to other types of components do not restrict the parameter unnecessarily (they accept any `JComponent`), despite the fact that they will typically only be used with `JInternalFrames`. The exception to this in the current Swing release is a single use of `setBoundsForFrame()`, (called from `BasicDesktopIconUI`), which passes in a `JDesktopIcon` rather than a `JInternalFrame`. If you implement your own `DesktopManager` or other L&F classes, you may find a need for this flexibility.

public abstract void activateFrame(JInternalFrame f)
: Called to indicate that the specified frame should become active.

public abstract void beginDraggingFrame(JComponent f)
: Called to indicate that the specified frame is now being dragged. The input component will normally be a `JInternalFrame`.

public abstract void beginResizingFrame(JComponent f, int direction)
: Called to indicate that the specified frame is going to be resized. The direction comes from `SwingConstants` and must be `NORTH`, `SOUTH`, `EAST`, `WEST`, `NORTH_EAST`, `NORTH_WEST`, `SOUTH_EAST` or `SOUTH_WEST`. The input component will normally be a `JInternalFrame`. When resizing is complete, `endResizingFrame()` will be called.

public abstract void closeFrame(JInternalFrame f)
: Called to indicate that the specified frame should be closed.

public abstract void deactivateFrame(JInternalFrame f)
: Called to indicate that the specified frame is no longer active.

public abstract void deiconifyFrame(JInternalFrame f)
: Called to indicate that the specified frame should no longer be iconified.

public abstract void dragFrame(JComponent f, int newX, int newY)
: Called to indicate that the specified frame should be moved from its current location to the newly specified coordinates. The input component will normally be a `JInternalFrame`.

public abstract void endDraggingFrame(JComponent f)
: Called to indicate that the specified frame is no longer being dragged. The input component will normally be a `JInternalFrame`.

public abstract void endResizingFrame(JComponent f)

Called to indicate that the specified frame is no longer being resized. The input component will normally be a `JInternalFrame`.

public abstract void iconifyFrame(JInternalFrame f)

Called to indicate that the specified frame should be iconified.

public abstract void maximizeFrame(JInternalFrame f)

This method is called to indicate that the specified frame should be maximized.

public abstract void minimizeFrame(JInternalFrame f)

Called to indicate that the specified frame should be minimized. Note that this is not the same as iconifying the frame. Typically, calling this method will cause the frame to return to its size and position from before it was maximized.

public abstract void openFrame(JInternalFrame f)

Called to add a frame and display it at a reasonable location. This is not normally called, because frames are normally added directly to their parent.

public abstract void resizeFrame(JComponent f, int newX, int newY, int newWidth, int newHeight)

Called to indicate that the specified frame has been resized. Note that resizing is still in progress (many calls to this method may be made while the frame is being resized) after this method completes. The input component will normally be a `JInternalFrame`.

public abstract void setBoundsForFrame(JComponent f, int newX, int newY, int newWidth, int newHeight)

Called to set a new size and location for a frame. The input component will normally be a `JInternalFrame`.

The DefaultDesktopManager Class

`DefaultDesktopManager` is a default implementation of the `DesktopManager` interface used by all Swing-supplied look-and-feels except the `WindowsLookAndFeel` (which supplies its own). In this section, we'll give a brief explanation of how each of the methods in the interface is implemented by this class.

Methods

public void activateFrame(JInternalFrame f)

Calls `setSelected(false)` on all other `JInternalFrames` contained by the specified frame's parent that are in the same layer as the input frame. It then moves the given frame to the front of its layer, selecting it.

public void closeFrame(JInternalFrame f)

Removes the given frame from its parent. It also removes the frame's icon (if displayed). It sets the frame's previous bounds to `null`.

public void deiconifyFrame(JInternalFrame f)

Removes the input frame's icon from its parent and adds the frame. It then tries to select the input frame.

public void dragFrame(JComponent f, int newX, int newY)

Calls `setBoundsForFrame()` with the given location and current dimensions.

public void iconifyFrame(JInternalFrame f)

Removes the given frame from its parent and adds the frame's desktop icon. Before adding the icon, it checks to see if it has ever been iconified. If not, it calls `getBoundsForIconOf()` to set the icon's bounds. This is only done once for a given frame, ensuring that each time a frame is iconified, it returns to the same location on the desktop.

public void maximizeFrame(JInternalFrame f)

Maximizes the given frame so that it fills its parent. It also saves the frame's previous bounds for use in `minimizeFrame()`. Once the frame has been maximized, it is also selected.

This method can be called on an iconified frame, causing it to be deiconified and maximized.

public void minimizeFrame(JInternalFrame f)

Sets the frame's bounds to its previous bounds. If there are no previous bounds (previous bounds are set by calling `maximizeFrame()`), the frame is not resized.

public void openFrame(JInternalFrame f)

Gets the desktop icon for the input frame. If the icon's parent is non-`null`, the icon is removed from the parent, and the frame is added. If its parent is `null`, this method does nothing.

public void resizeFrame(JComponent f, int newX, int newY, int newWidth, int newHeight)

Calls `setBoundsForFrame()` with the given location and dimensions.

public void setBoundsForFrame(JComponent f, int newX, int newY, int newWidth, int newHeight)

Moves and resizes the given frame (using `setBounds()`) and validates the frame if the size was actually changed.

public void beginDraggingFrame(JComponent f)
public void beginResizingFrame(JComponent f, int direction)
public void deactivateFrame(JInternalFrame f)
public void endDraggingFrame(JComponent f)
public void endResizingFrame(JComponent f)

These methods have empty implementations in this class.

Protected Methods

This default implementation provides several convenience methods, which it uses in the methods described above. The methods relate to desktop icon management and the management of a frame's previous size (when maximized). If you subclass `DefaultDesktopManager`, these methods will probably be of use to you.

The frame's previous bounds and an indication of whether or not it has ever been iconified are stored in client properties on the frame itself.* The property names used are "previousBounds" (which holds a `Rectangle`) and "wasIconOnce" (which holds a `Boolean`).

protected Rectangle getBoundsForIconOf(JInternalFrame f)

Gets the bounds for the input frame's icon. The width and height are taken directly from the size of the icon. The icon's location will be the lower-left corner of the desktop. If an icon has already been placed in this corner, the icon is placed directly to the right, continuing until an unclaimed position along the bottom of the frame is found. If there is no space along the bottom, a new row of icons is started directly above the first row. Once a frame has been iconified, its icon's location is set and the icon will always return to the same spot (unless the icon is moved by the user).

protected Rectangle getPreviousBounds(JInternalFrame f)

Returns the frame's previous bounds (set when the frame is maximized). These bounds are retrieved from the frame's "previousBounds" client property.

protected void removeIconFor(JInternalFrame f)

Removes the input frame's icon from its parent and repaints the region under the icon.

protected void setPreviousBounds(JInternalFrame f, Rectangle r)

Saves the previous bounds of a frame. This is done by saving the frame's previous bounds in the frame itself, using the client property, "previousBounds." This is generally called by `maximizeFrame()` with the data being used in a subsequent `minimizeFrame()` call.

---

* See Chapter 3, *Swing Component Basics* for an explanation of JComponent's client property feature.

protected void setWasIcon(JInternalFrame f, Boolean value)

This method is called by `iconifyFrame()` to indicate whether or not the frame has, at some time, been iconified. This is done by saving the boolean value in the frame itself, using the client property "wasIconOnce." This is used to determine whether or not the icon's bounds have been defined.

protected boolean wasIcon(JInternalFrame f)

This method is called to determine whether or not a frame has ever been iconified (if it has, bounds will already be defined for the icon). This is done by returning the "wasIconOnce" client property on the frame.

Building A Desktop

In this section, we'll pull together some of the things we've discussed in the previous pages to create an application using `JDesktopPane`, `JInternalFrame`, and a custom `DesktopManager`. The example shows:

- The effect of adding frames to different layers of the desktop.
- How to display a background image ("wallpaper") on the desktop.
- How to keep frames from being moved outside of the desktop.
- How to deiconify, move, and resize internal frames by providing frame "tiling".
- How to take advantage of `JInternalFrame`'s constrained properties by requiring that there be at least one non-iconified frame on the desktop.

Here's what the application looks like when it's running.

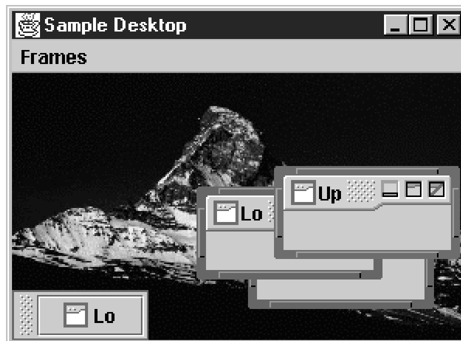

Figure 9-4. SampleDesktop layered frames and background image

Figure 9-4 shows the desktop with three frames, plus a fourth that has been iconified. The frames titled "Lo" are in a lower layer than the "Up" frame. No matter which frame is active or how the frames are arranged, the "Up" frame will always

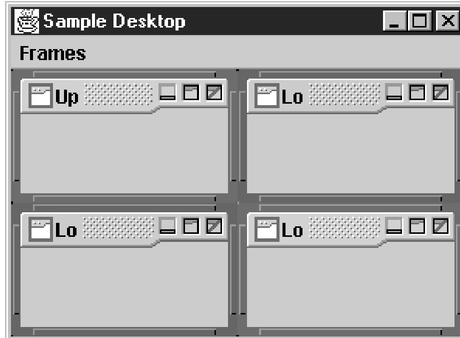

Figure 9-5. SampleDesktop with tiled frames

appear on top of the others. Frames in the same layer can be brought to the front of that layer by clicking on the frame. This display also shows the use of a background image (what good is a desktop if you can't put your favorite image on the background, right?). This image is added to a very low layer (the lowest possible integer, actually) to ensure that it is always painted behind anything else in the desktop. Figure 9-5 shows the same display after the frames have been "tiled."

Now, let's take a look at some of the code used to create this example. There are three primary classes:

SampleDesktop
: This is the main class, which we chose to create as a `JFrame` that uses a `JDesktopPane` as its content pane. `SampleDesktop` has two inner classes. `AddFrameAction` is an `Action` used to add frames to the desktop. Recall from Chapter 3 that actions are a nice way to encapsulate functionality that you might want to invoke from multiple locations. The other inner class, `IconPolice`, is responsible for ensuring that if there is only a single frame on the desktop, it cannot be iconified.

SampleDesktopManager
: An extension of `DefaultDesktopManager` that keeps frames from being moved outside the bounds of the desktop.

TileAction
: A generic action class that can be used to tile all frames on a given desktop.

Let's take a look at these classes piece by piece. The complete source listing is provided at the end of the chapter.

Setting Things Up

The first thing to look at is the `SampleDesktop` constructor:

```
  public SampleDesktop(String title) {
    super(title);
    addWindowListener(new BasicWindowMonitor());

    // Create a desktop and set it as the content pane. Don't set the layered
    // pane, since it needs to hold the menubar too.
    desk = new JDesktopPane();
    setContentPane(desk);

    // Install our custom desktop manager
    desk.setDesktopManager(new SampleDesktopMgr());
    createMenuBar();
    loadBackgroundImage();
  }
```

We set the frame's content pane to our new `JDesktopPane`. Since we won't be adding anything else to the body of the frame, this is a good idea. We could also have called `getContentPane().add(desk)`, but as we discussed in Chapter 8, this just introduces an unnecessary extra level (the content pane would then be a `JPanel` holding only our `JDesktopPane`). The more important thing to avoid is calling `setLayeredPane(desk)`. Remember, the layered pane is responsible for rendering the menubar too. If you did this, the menubar would still be drawn at the top of the frame, but your desktop would be filling the same space, allowing frames to be placed over the menu.

The `createMenuBar()` method called here just adds a few options to the frame's menubar. It uses instances of `AddFrameAction` for adding new frames (at "Up" and "Lo" levels), and it uses an instance of `TileAction` to support frame tiling. See the complete code listing at the end of this section for more details on this method.

The `loadBackgroundImage()` method looks like this:

```
protected void loadBackgroundImage() {
  ImageIcon icon = new ImageIcon("images/matterhorn.gif");
  JLabel l = new JLabel(icon);
  l.setBounds(0, 0, icon.getIconWidth(), icon.getIconHeight());

  desk.add(l, new Integer(Integer.MIN_VALUE));
}
```

This method just creates a large `JLabel` containing an image and adds this label to the lowest possible layer of the desktop. This ensures that nothing will ever be painted behind the background. In this example, we don't make any effort to resize or tile the background image, but it certainly could be done.

Adding Frames to the Desktop

The AddFrameAction class is an Action we've added to the menubar. When fired, AddFrameAction instantiates a JInternalFrame and adds it to the specified layer of the desktop. Here's the code for the actionPerformed() method of this class:

```
public void actionPerformed(ActionEvent ev) {
  JInternalFrame f = new JInternalFrame(name, true, true, true, true);
  f.addVetoableChangeListener(iconPolice);
  f.setBounds(0, 0, 120, 60);
  desk.add(f, layer);
}
```

The important thing to notice here is that we set the bounds, not just the size, of the new frame. If you don't specify a location (we've specified [0,0], the upper-left corner) for the frame, it won't appear on the desktop when you add it. Remember, there's no layout manager controlling the location of the components in a JDesktopPane.

Veto Power

In the previous code block, we added a VetoableChangeListener to each new frame we created. This listener is an instance of another inner class called IconPolice. The purpose of this class is to ensure that the last frame on the desktop cannot be iconified. This may not be the most useful thing in the world to do, but it serves as an example of how you can take advantage of the fact that certain JInternalFrame properties are constrained. Here's the code for this class.

```
class IconPolice implements VetoableChangeListener {
  public void vetoableChange(PropertyChangeEvent ev)
   throws PropertyVetoException {
    String name = ev.getPropertyName();
    if (name.equals(JInternalFrame.IS_ICON_PROPERTY)
    && (ev.getNewValue() == Boolean.TRUE)) {
      JInternalFrame[] frames = desk.getAllFrames();
      int count = frames.length;
      int nonicons = 0; // how many are not icons?
      for (int i=0; i<count; i++) {
        if (frames[i].isIcon() == false) {
          nonicons++;
        }
      }
      if (nonicons <= 1) {
        throw new PropertyVetoException("Invalid Iconification!", ev);
      }
    }
  }
```

```
}
```

If you haven't used constrained properties before, this code may look a little strange. The idea behind constrained properties is that before a property is changed, all registered listeners are given the opportunity to "veto" the change. This is done by throwing a `PropertyVetoException` from the `vetoableChange()` method as we've done here.

Bounding the Frames

The next class to look at is our custom desktop manager called `SampleDesktopMgr`. This class is an extension of `DefaultDesktopManager` which overrides the default implementation of `setBoundsForComponent()`. This is the method called any time the frame is moved or resized. The new implementation simply checks the new location of the frame to see if the requested change of bounds will result in part of the frame moving outside of the desktop. If so, it adjusts the coordinates so that the frame will only be moved to the edge of the desktop. The code for this method is included at the end of the chapter.

In order to correctly handle invalid bounds changes, we need to know if the frame is being moved or if it is being resized. This will affect how we adjust the frame's size and location in the `setBoundsForComponent()`. In this example, we do this by storing a client property* called `RESIZING` in each frame. In `beginResizingFrame()`, we set this property to `true`. When `endResizingFrame()` is called, we switch it to `false`. Here's how we do this:

```
protected static final String RESIZING = "RESIZING";

public void beginResizingFrame(JComponent f, int dir) {
  f.putClientProperty(RESIZING, Boolean.TRUE);
}

public void endResizingFrame(JComponent f) {
  f.putClientProperty(RESIZING, Boolean.FALSE);
}
```

This class is included only as a useful example of the type of thing you might want to do with a desktop manager. If you don't mind frames being moved off the desktop, you can always just use `DefaultDesktopManager` (the default).

Moving Things Around

The last class in this example is called `TileAction`. Its job is to resize all of the frames and lay them out in a grid on a desktop. There are a few interesting things

* For an explanation of client properties, see Chapter 3.

that take place in the actionPerformed() method of this class. First, we get all of the frames on the desktop and determine where each frame should be placed and how big it should be based on the size of the desktop and the total number of frames. For the details of how this is calculated, see the full code listing at the end of the chapter.

Next, we iterate over all of the frames on the desktop, deiconifying any iconified frames and then setting the size and location of each frame. Here's the block of code that does this work:

```
for (int i=0; i<rows; i++) {
  for (int j=0; j<cols && ((i*cols)+j<count); j++) {
    JInternalFrame f = allframes[(i*cols)+j];
    if ((f.isClosed() == false) && (f.isIcon() == true)) {
      try {
        f.setIcon(false);
      }
      catch (PropertyVetoException ex) {}
    }
    desk.getDesktopManager().resizeFrame(f, x, y, w, h);
    x += w;
  }
  y += h;     // start the next row
  x = 0;
}
```

We call setIcon() on the frame, rather than calling deiconifyFrame() on the DesktopManager. We do this because deiconifyFrame() does not actually change the state of the icon property in the frame, which can result in unexpected behavior down the road. Figure 9-6 shows the sequence of calls (only certain significant calls are identified) made when we call setIcon(false).

Note that the UI delegate is registered as a listener for property change events. When it hears that a frame is been deiconified, it calls deiconifyFrame() on the desktop manager. This object then adds the frame to its container (the desktop pane in this case), removes the icon, and selects the newly added frame.

Once we've got the frame deiconified, we relocate and resize it by calling the resizeFrame() method on the desktop manager:

```
desk.getDesktopManager().resizeFrame(f, x, y, w, h);
```

We call this method (instead of just calling setBounds() on the frame,) because it validates the frame after setting its bounds.

Source Code

Here's the complete source code (three files) for this example.

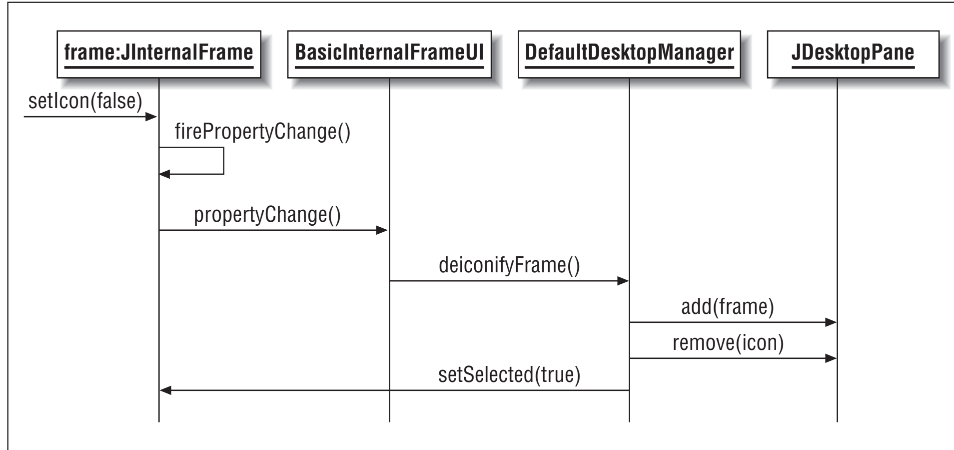

Figure 9-6. setIcon() sequence diagram

```
// SampleDesktop.java
//
import com.sun.java.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.util.*;
import java.beans.*;

// An example that shows how to do a few interesting things using
// JInternalFrames, JDesktopPane, and DesktopManager.
public class SampleDesktop extends JFrame {

  private JDesktopPane desk;
  private IconPolice iconPolice = new IconPolice();

  public SampleDesktop(String title) {
    super(title);
    addWindowListener(new BasicWindowMonitor());

    // Create a desktop and set it as the content pane. Don't set the layered
    // pane, since it needs to hold the menubar too.
    desk = new JDesktopPane();
    setContentPane(desk);

    // Install our custom desktop manager
    desk.setDesktopManager(new SampleDesktopMgr());

    createMenuBar();
    loadBackgroundImage();
  }
```

```
// Create a menubar to show off a few things.
protected void createMenuBar() {
  JMenuBar mb = new JMenuBar();
  JMenu menu = new JMenu("Frames");

  menu.add(new AddFrameAction(true)); // add "upper" frame
  menu.add(new AddFrameAction(false)); // add "lower" frame
  menu.add(new TileAction(desk)); // add tiling capability

  setJMenuBar(mb);
  mb.add(menu);
}

// Here we load a background image for our desktop.
protected void loadBackgroundImage() {
  ImageIcon icon = new ImageIcon("images/matterhorn.gif");
  JLabel l = new JLabel(icon);
  l.setBounds(0,0,icon.getIconWidth(),icon.getIconHeight());

  // Place the image in the lowest possible layer so nothing
  // can ever be painted under it.
  desk.add(l, new Integer(Integer.MIN_VALUE));
}

// This class will add a new JInternalFrame when requested.
class AddFrameAction extends AbstractAction {
  public AddFrameAction(boolean upper) {
    super(upper ? "Add Upper Frame" : "Add Lower Frame");
    if (upper) {
      this.layer = new Integer(2);
      this.name = "Up";
    }
    else {
      this.layer = new Integer(1);
      this.name = "Lo";
    }
  }

  public void actionPerformed(ActionEvent ev) {
    JInternalFrame f = new JInternalFrame(name,true,true,true,true);
    f.addVetoableChangeListener(iconPolice);

    f.setBounds(0, 0, 120, 60);
    desk.add(f, layer);
  }

  private Integer layer;
  private String name;
}
```

```
  // A simple vetoable change listener that insists that there is always at
  // least one noniconified frame (just as an example of the vetoable
  // properties).
  class IconPolice implements VetoableChangeListener {
    public void vetoableChange(PropertyChangeEvent ev)
     throws PropertyVetoException {

      String name = ev.getPropertyName();
      if (name.equals(JInternalFrame.IS_ICON_PROPERTY)
      && (ev.getNewValue() == Boolean.TRUE)) {
        JInternalFrame[] frames = desk.getAllFrames();
        int count = frames.length;
        int nonicons = 0; // how many are not icons?
        for (int i=0; i<count; i++) {
          if (frames[i].isIcon() == false) {
            nonicons++;
          }
        }
        if (nonicons <= 1) {
          throw new PropertyVetoException("Invalid Iconification!", ev);
        }
      }
    }
  }

  // A simple test program.
  public static void main(String[] args) {
    SampleDesktop td = new SampleDesktop("Sample Desktop");

    td.setSize(300, 220);
    td.setVisible(true);
  }
}

// SampleDesktopMgr.java
//
import com.sun.java.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.util.*;
import java.beans.*;

// A DesktopManager that keeps its frames inside the desktop.
public class SampleDesktopMgr extends DefaultDesktopManager {

  // We'll tag internal frames that are being resized using a client
  // property with the name RESIZING.  Used in setBoundsForFrame().
  protected static final String RESIZING = "RESIZING";
```

```
public void beginResizingFrame(JComponent f, int dir) {
  f.putClientProperty(RESIZING, Boolean.TRUE);
}

public void endResizingFrame(JComponent f) {
  f.putClientProperty(RESIZING, Boolean.FALSE);
}

// This is called any time a frame is moved or resized.  This
// implementation keeps the frame from leaving the desktop.
public void setBoundsForFrame(JComponent f, int x, int y, int w, int h) {
  if (f instanceof JInternalFrame == false) {
    super.setBoundsForFrame(f, x, y, w, h); // only deal w/internal frames
  }
  else {
    JInternalFrame frame = (JInternalFrame)f;

    // Figure out if we are being resized (otherwise it's just a move)
    boolean resizing = false;
    Object r = frame.getClientProperty(RESIZING);
    if (r != null && r instanceof Boolean) {
      resizing = ((Boolean)r).booleanValue();
    }

    JDesktopPane desk = frame.getDesktopPane();
    Dimension d = desk.getSize();

    // Nothing all that fancy below, just figuring out how to adjust
    // to keep the frame on the desktop.
    if (x < 0) {              // too far left?
      if (resizing)
        w += x;               // don't get wider!
      x=0;                    // flush against the left side
    }
    else {
      if (x+w>d.width) {      // too far right?
       if (resizing)
         w = d.width-x;       // don't get wider!
       else
         x = d.width-w;       // flush against the right side
      }
    }
    if (y < 0) {              // too high?
      if (resizing)
        h += y;               // don't get taller!
      y=0;                    // flush against the top
    }
    else {
```

```
        if (y+h > d.height) {   // too low?
          if (resizing)
            h = d.height - y;   // don't get taller!
          else
            y = d.height-h;     // flush against the bottom
        }
      }

      // Set 'em the way we like 'em
      super.setBoundsForFrame(f, x, y, w, h);
    }
  }
}

// TileAction.java
//
import com.sun.java.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.beans.*;

// An action that tiles all internal frames when requested.
public class TileAction extends AbstractAction {
  private JDesktopPane desk; // the desktop to work with

  public TileAction(JDesktopPane desk) {
    super("Tile Frames");
    this.desk = desk;
  }

  public void actionPerformed(ActionEvent ev) {

    // How many frames do we have?
    JInternalFrame[] allframes = desk.getAllFrames();
    int count = allframes.length;
    if (count == 0) return;

    // Determine the necessary grid size
    int sqrt = (int)Math.sqrt(count);
    int rows = sqrt;
    int cols = sqrt;
    if (rows*cols < count) {
      cols++;
      if (rows*cols < count) {
        rows++;
      }
    }

    // Define some initial values for size & location
```

```
    Dimension size = desk.getSize();

    int w = size.width/cols;
    int h = size.height/rows;
    int x = 0;
    int y = 0;

    // Iterate over the frames, deiconifying any iconified frames and then
    // relocating & resizing each
    for (int i=0; i<rows; i++) {
      for (int j=0; j<cols && ((i*cols)+j<count); j++) {
        JInternalFrame f = allframes[(i*cols)+j];

        if ((f.isClosed() == false) && (f.isIcon() == true)) {
          try {
            f.setIcon(false);
          }
          catch (PropertyVetoException ex) {}
        }

        desk.getDesktopManager().resizeFrame(f, x, y, w, h);
        x += w;
      }
      y += h; // start the next row
      x = 0;
    }
  }
}
```