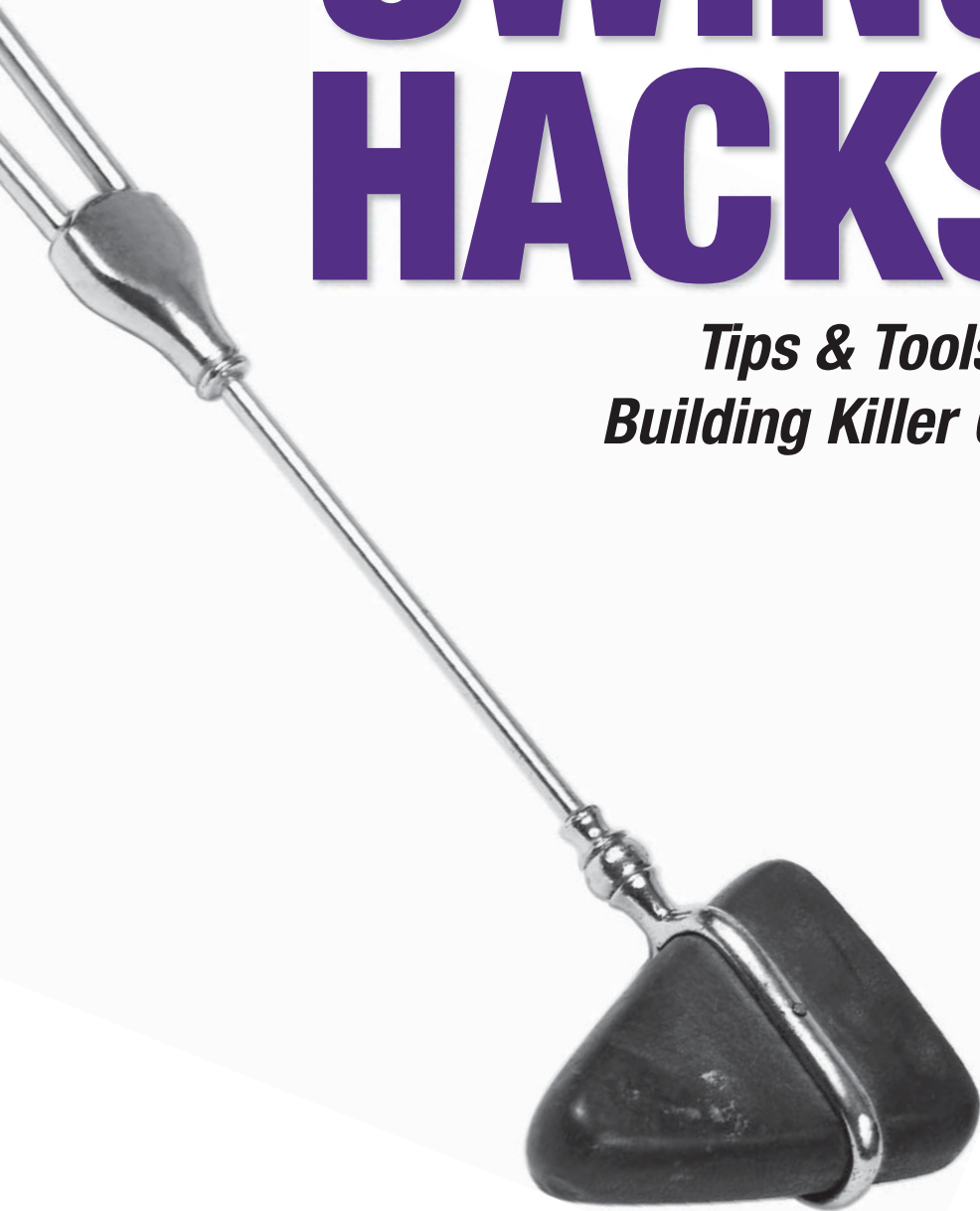


SWING HACKS™

*Tips & Tools for
Building Killer GUIs*



O'REILLY®

Joshua Marinacci & Chris Adamson

HACK
#44

Turn Dialogs into Frame-Anchored Sheets

One of Mac OS X's best ideas is binding the dialog to the window it blocks. This hack shows you how to mimic this in Swing.

One of my favorite features in Mac OS X is the *sheet*. This is a dialog box replacement that slides down from a window's titlebar. Figure 6-7 shows an example of a sheet in Apple's Safari web browser.



Figure 6-7. Sheet in Mac OS X Safari browser

Why Sheets Rock

Looking at it, you might think, “what’s the big deal” or “how is this any different than a regular dialog?” Oh, it’s far better:

A sheet is visually anchored to the window that it blocks

On platforms where dialogs have titlebars and close boxes, the relationship between a dialog and the window it blocks is not necessarily intuitive. On a related point....

A sheet doesn't have a close box

Dialog close boxes are one of the most hateful and stupid concepts in Windows and its many Linux imitators. What does the close box mean? Cancel? The default option? What does it mean when the dialog has multiple options of equal plausibility and thus no default? Perhaps the worst thing about the close box was back in the AWT era when Java developers—too lazy to add and wire-up an OK button to their dialogs—just figured users could dismiss the dialog with the close box. Mac OS 8 and 9 dialogs didn't have close boxes, so when a Java application brought up such a dialog, *the application blocked itself forever*. Duh. Sheets mean having to click one of the provided buttons, so the user's choices are unambiguous.

A sheet is used to block one window

This is an obvious side effect of being visually tied to a single window, but that's probably the most common case. As a side effect, this gives greater prominence to dialogs that block all windows for a single application (“Are you sure you want to Quit and lose all unsaved changes in all documents?”) and dialogs that block all applications (“Are you sure you want to Shut Down?”).

So, if you agree that it's an excellent GUI concept, the next question is “how do I mimic sheets in Swing?”

Use the Glass Pane

One way to imitate the Mac OS X sheet is to use the *glass pane*—a layer in the *LayeredPane* used by all *RootPaneContainers*, including *JApplets*, *JFrames*, *JInternalFrames*, *JWindows*, and *JDialogs*. In terms of z-order—the ordering of layers on “top” of one another from the user's perspective—the glass pane is “above” the content pane and the menu bar in the *LayeredPane*. It is usually empty and unfilled. One of the more typical uses for the glass pane is to add a *MouseListener* and *MouseMotionListener* to deny events to the content pane and thereby block it.

To imitate the sheet with the glass pane, the idea is to take the contents you'd usually put into a *JDialog* and place them instead into the glass pane. This will put them in the frame, above and in front of the frame's contents. To position your contents at the top center of the glass pane, you can use a *GridBagLayout* that gives the sheet a *NORTH* anchor, and then add a “glue” component in the next row that takes up as much vertical space as possible, pushing the sheet to the top of the pane.

Example 6-7 shows a subclass of *JFrame* that exposes a `showJDialogAsSheet()` method, which grabs the *JDialog*'s content pane *JComponent* and inserts it into the glass pane as the sheet.

Example 6-7. Adding a sheet in a JFrame's glass pane

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

public class SheetableJFrame extends JFrame {

    JComponent sheet;
    JPanel glass;
```

Example 6-7. Adding a sheet in a JFrame's glass pane (continued)

```
public SheetableJFrame (String name) {
    super(name);
    glass = (JPanel) getGlassPane();
}

public JComponent showJDialogAsSheet (JDialog dialog) {
    sheet = (JComponent) dialog.getContentPane();
    sheet.setBackground (Color.red);
    glass.setLayout (new GridBagLayout());
    sheet.setBorder (new LineBorder(Color.black, 1));
    glass.removeAll();
    GridBagConstraints gbc = new GridBagConstraints();
    gbc.anchor = GridBagConstraints.NORTH;
    glass.add (sheet, gbc);
    gbc.gridy=1;
    gbc.weighty = Integer.MAX_VALUE;
    glass.add (Box.createGlue(), gbc);
    glass.setVisible(true);
    return sheet;
}

public void hideSheet() {
    glass.setVisible(false);
}
}
```

Simple enough, isn't it? To test this component, just create a JFrame and a JDialog to insert into the frame. The SheetTest class in [Example 6-8](#), which exercises the SheetableJFrame, fills the frame with interesting content by making a JLabel from an image file. It then creates a JDialog the easy way—by making JOptionPane do it.



If you're used to calling JOptionPane's various showXXXDialog() methods, you might be unfamiliar with the idea of constructing and holding on to a JOptionPane and creating dialogs from it. This approach isn't common, but it's useful if you want to hold on to the JDialog, maybe for reuse (e.g., you have memory or performance concerns with repeatedly creating and disposing of dialogs) or because you don't want to block as soon as you show it. In the case of this demo, the advantage is to get a dialog to play with, without having to layout and wire up everything yourself.

Example 6-8. Testing the SheetableJFrame

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

Turn Dialogs into Frame-Anchored Sheets

Example 6-8. Testing the SheetableJFrame (continued)

```
public class SheetTest extends Object
    implements PropertyChangeListener {

    JOptionPane optionPane;
    SheetableJFrame frame;

    public static void main (String[] args) {
        new SheetTest();
    }

    public SheetTest () {
        frame = new SheetableJFrame ("Sheet test");
        // put an image in the frame's content pane
        ImageIcon icon = new ImageIcon ("keagy-lunch.png");
        JLabel label = new JLabel (icon);
        frame.getContentPane().add(label);
        // build JOptionPane dialog and hold onto it
        optionPane = new JOptionPane ("Do you want to save?",
                                     JOptionPane.QUESTION_MESSAGE,
                                     JOptionPane.YES_NO_OPTION);

        frame.pack();
        frame.setVisible(true);
        optionPane.addPropertyChangeListener (this);
        // pause for effect, then show the sheet
        try {Thread.sleep(1000);}
        catch (InterruptedException ie) {}
        JDialog dialog =
            optionPane.createDialog (frame, "irrelevant");
        frame.showJDialogAsSheet (dialog);
    }

    public void propertyChange (PropertyChangeEvent pce) {
        if (pce.getPropertyName().equals (JOptionPane.VALUE_PROPERTY)) {
            System.out.println ("Selected option " +
                               pce.getNewValue());
            frame.hideSheet();
        }
    }
}
```

The other thing that's interesting about holding onto a `JOptionPane` is that it—not the dialog—is what fires events when the user clicks one of the buttons. These are fired as `PropertyChangeEvents` with the property name value, which you should refer to as `JOptionPane.VALUE_PROPERTY`. In this case, what you want to listen for is any change in the value, which indicates that something has been clicked and means it's time to hide the sheet.

When you run `SheetTest`, the image comes up in a `SheetableJFrame` and, after a one-second pause for effect, the sheet appears at top center, as seen in [Figure 6-8](#).



Figure 6-8. *JDialog* shown as a “sheet” in the glass pane

When you click either of the options, the event listener hides the glass pane, which makes the sheet disappear.

This is a pretty simple case of using the glass pane—in fact, the test is a few lines longer than the implementation of the sheet. The only thing that’s missing is the [charming animation of the sheet sliding in \[Hack #45\]](#)....