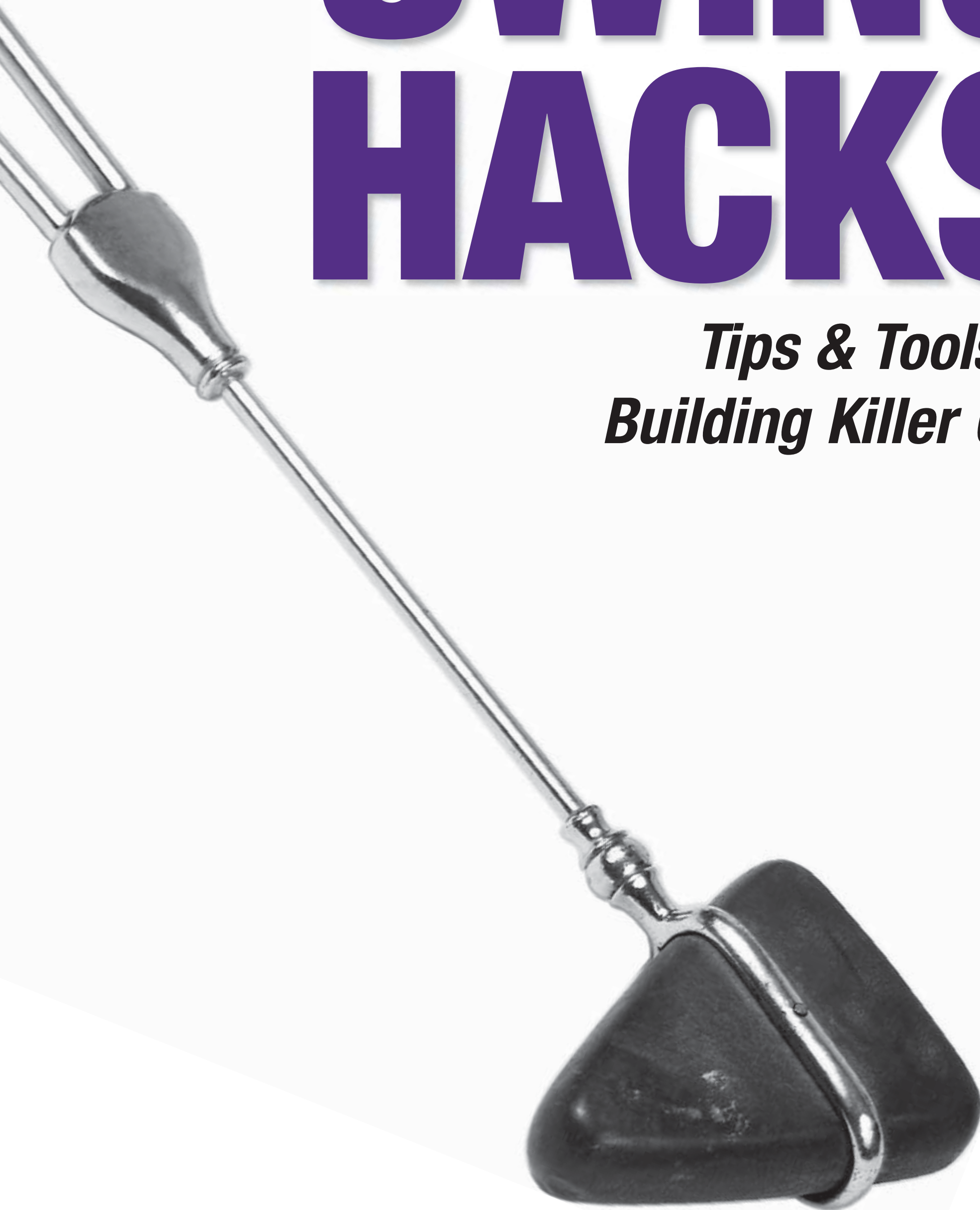
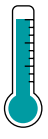
SWING HACKS™

Tips & Tools for Building Killer GUIs



O'REILLY®

Joshua Marinacci & Chris Adamson



HACK #8 Animate Transitions Between Tabs

This hack shows how to create animated transitions that play whenever the user switches tabs on a `JTabbedPane`.

One of Swing's great strengths is that you can hack into virtually anything. In particular, I love making changes to a component's painting code. The ability to do this is one of the reasons I prefer Swing over SWT. Swing gives me the freedom to create completely new UI concepts, such as transitions.

With the standard paint methods, Swing provides most of what you will need to build the transitions. You will have to put together three additional things, however. First, you need to find out when the user actually clicked on a tab to start a transition. Next, you need a thread to control the animation. Finally, since some animations might fade between the old and new tabs, you need a way to provide images of both tabs at the same time. With those three things, you can build any animation you desire.

Building a Basic Tabbed Pane

To keep things tidy, I have implemented this hack as a subclass of `JTabbedPane`, except for the actual animation drawing, which will be delegated to a further subclass. By putting all of the heavy lifting into the parent class, you will be able to create new animations easily.

Example 1-16 is the basic skeleton of the parent class.

Example 1-16. A skeleton for the transition manager

```
public class TransitionTabbedPane extends JTabbedPane
    implements ChangeListener, Runnable {

    protected int animation_length = 20;

    public TransitionTabbedPane( ) {
        super( );
        this.addChangeListener(this);
    }

    public int getAnimationLength( ) {
        return this.animation_length;
    }

    public void setAnimationLength(int length) {
        this.animation_length = length;
    }
```

`TransitionTabbedPane` extends the standard `JTabbedPane` and also implements `ChangeListener` and `Runnable`. `ChangeListener` allows you to learn

when the user has switched between tabs. Since the event is propagated *before* the new tab is painted, inserting the animation is very easy. `Runnable` is used for the animation thread itself.

You could have split the thread into a separate class, but I think that keeping all of the code together makes the system more encapsulated and easier to maintain.

`TransitionTabbedPane` adds one new property, the animation length. This defines the number of steps used for the transition, and it can be set by the subclass or external code.

Scheduling the Animation

Since the pane was added as a `ChangeListener` to itself, the `stateChanged( )` method will be called whenever the user switches tabs. This is the best place to start the animation thread. Once started, the thread will capture the previous tab into a buffer, loop through the animation, and control the repaint speed:

```
// threading code
public void stateChanged(ChangeEvent evt) {
    new Thread(this).start( );
}

protected int step;
protected BufferedImage buf = null;
protected int previous_tab = -1;

public void run( ) {
    step = 0;

    // save the previous tab
    if(previous_tab != -1) {
        Component comp = this.getComponentAt(previous_tab);
        buf = new BufferedImage(comp.getWidth( ),
            comp.getHeight( ),
            BufferedImage.TYPE_4BYTE_ABGR);
        comp.paint(buf.getGraphics( ));
    }
```

Notice that the `run( )` method grabs the previous tab component only when the `previous_tab` index isn't `-1`. The component will always have a valid value, except for the first time the pane is shown on screen, but that's OK because the user won't have really switched from anything anyway. If there is a previous tab, then the code grabs the component and paints it into a buffer image.

It's important to note that this is *not* thread-safe because the code is being executed on a custom thread, not the Swing thread. However, since the tab is about to be hidden anyway—and, in fact, the next real `paint()` call will only draw the new tab—you shouldn't have any problems. Any changes introduced by this extra `paint()` call won't show up on screen.

With the previous component safely saved away, you can now loop through the animation:

```
for(int i=0; i<animation_length; i++) {
    step = i;
    repaint();
    try {
        Thread.currentThread().sleep(100);
    } catch (Exception ex) {
        p("ex: " + ex);
    }
}

step = -1;
previous_tab = this.getSelectedIndex();
repaint();
```

This code shows a basic animation loop from 1 to `N`, with a 100-millisecond duration for each frame.

A more sophisticated version of the code could have dynamic frame rates to adjust for system speed.

Once the transition finishes, the animation step is set back to `-1`, the previous tab is stored, and the screen is repainted one last time, without the transition effects.

Drawing the Animation

The `TransitionTabbedPane` is now set up with the proper resources and repaints, but it still isn't drawing the animation. Because the animation is going to partially or completely obscure the tabs underneath, the best place to draw is right after the children are painted:

```
public void paintChildren(Graphics g) {
    super.paintChildren(g);

    if(step != -1) {
```

```
            Rectangle size = this.getComponentAt(0).getBounds( );
            Graphics2D g2 = (Graphics2D)g;
            paintTransition(g2, step, size, buf);
        }
    }

    public void paintTransition(Graphics2D g2, int step,
        Rectangle size, Image prev) {
    }
```

This code puts all of the custom drawing into the `paintTransition( )` method, currently empty. It will only be called if step isn't `-1`, meaning during a transition animation. The `paintTransition( )` method provides the drawing canvas, the current animation step, the size and position of the content area (excluding the tabs themselves), and the image buffer that stores the previous tab's content. By putting all of this in a single method, subclasses can build their own animations very easily. Example 1-17 is a simple transition with a white rectangle that grows out of the center, filling the screen, then shrinking again to reveal the new tab content.

Example 1-17. Setting up an animated transition

```
public class InOutPane extends TransitionTabbedPane {

    public void paintTransition(Graphics2D g2, int state,
            Rectangle size, Image prev) {

        int length = getAnimationLength( );
        int half = length/2;

        double scale = size.getHeight( )/length;
        int offset = 0;
        // calculate the fade out part
        if(state >= 0 && state < half) {
            // draw the saved version of the old tab component
            if(prev != null) {
                g2.drawImage(prev,(int)size.getX(),(int)size.getY( ),null);
            }
            offset = (int)((10-state)*scale);
        }

        // calculate the fade in part
        if(state >= half && state < length) {
            g2.setColor(Color.white);
            offset = (int)((state-10)*scale);
        }

        // do the drawing
        g2.setColor(Color.white);
        Rectangle area = new Rectangle((int)(size.getX( )+offset),
            (int)(size.getY( )+offset),
```

Example 1-17. Setting up an animated transition (continued)

```
            (int)(size.getWidth( )-offset*2),
            (int)(size.getHeight( )-offset*2));
        g2.fill(area);
    }
}
```

`InOutPane` implements only the `paintTransition( )` method, leaving all of the harder tasks to the parent class. First, it determines how long the animation will be, and then it calculates an offset to grow and shrink the white rectangle. If the drawing process is currently in the first half of the animation (`step < half`), then it draws the previous tab below the rectangle, creating the illusion that old tab content is still really on screen with the rectangle growing above it. For the second half of the animation, it just draws the rectangle, letting the real tab (the new one) shine through as the rectangle shrinks.

Putting It All Together

Because `TransitionTabbedPane` is just a `JTabbedPane` subclass, it can be used wherever the original would be. Example 1-18 creates a frame with two tabs, each containing a button. The running program looks like Figure 1-23. As you switch between the tabs, you will see an animation like that shown in Figure 1-24.

Example 1-18. Testing out tabbed animation transitions

```
public class TabFadeTest {

    public static void main(String[] args) {

        JFrame frame = new JFrame("Fade Tabs");

        JTabbedPane tab = new InOutPane( );
        tab.addTab("t1",new JButton("Test Button 1"));
        tab.addTab("t2",new JButton("Test Button 2"));

        frame.getContentPane( ).add(tab);
        frame.pack( );
        frame.show( );
    }

}
```

Another Example

Because `TransitionTabbedPane` makes it so easy to build new animations, I thought I'd add another one. This is the old venetian blinds effect, where

Figure 1-23. Two tabs, before transition effect begins

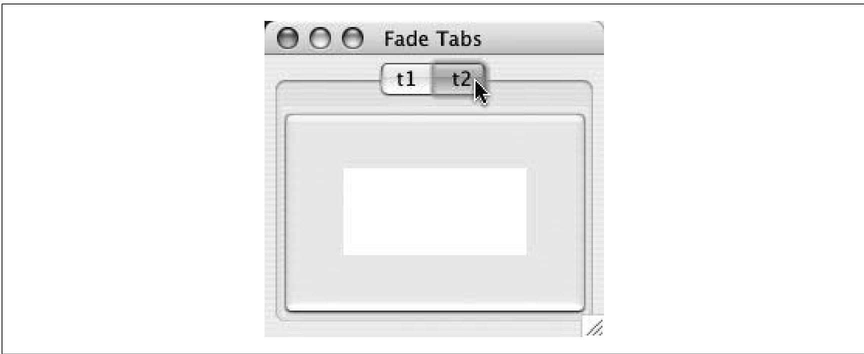


Figure 1-24. Tab transition at mid-point

vertical bars cover the old screen and uncover the new one; Example 1-19 puts it together.

Example 1-19. Creating a venetian blinds effect

```
public class VenetianPane extends TransitionTabbedPane {
    public void paintTransition(Graphics2D g2, int step,
            Rectangle size, Image prev) {

        int length = getAnimationLength( );
        int half = length/2;

        // create a blind
        Rectangle blind = new Rectangle( );

        // calculate the fade out part
        if(step >= 0 && step < half) {
            // draw the saved version of the old tab component
            if(prev != null) {
```

Example 1-19. Creating a venetian blinds effect (continued)

```
                g2.drawImage(prev,(int)size.getX(),(int)size.getY( ),null);
            }
            // calculate the growing blind
            blind = new Rectangle(
                (int)size.getX( ),
                (int)size.getY( ),
                step,
                (int)size.getHeight( ));
        }

        // calculate the fade in part
        if(step >= half && step < length) {
            // calculate the shrinking blind
            blind = new Rectangle(
                (int)size.getX( ),
                (int)size.getY( ),
                length-step,
                (int)size.getHeight( ));
            blind.translate(step-half,0);
        }

        // draw the blinds
        for(int i=0; i<size.getWidth( )/half; i++) {
            g2.setColor(Color.white);
            g2.fill(blind);
            blind.translate(half,0);
        }

    }
}
```

Just like `InOutPane`, `VenetianPane` selectively draws the old tab and then calculates the placement of animated rectangles. In this case, there is a blind rectangle that spans the entire screen from top to bottom, but has the width of the current step. As a result of the step growing, this rectangle gets bigger with each frame. For the second half of the animation, it shrinks and moves to the right, making it appear to fade into nothing. Once the blind is calculated, `VenetianPane` draws the blind multiple times to cover the entire tab content area, creating the effect seen in Figure 1-25.

This hack is quite extensible. With the power of Java2D you could add translucency, blurs, OS X-like genie effects, or anything else you can dream up. As a future enhancement, you could include more animation settings to control the frame rate and transition time. If you do create more, please post them on the Web for others to share.

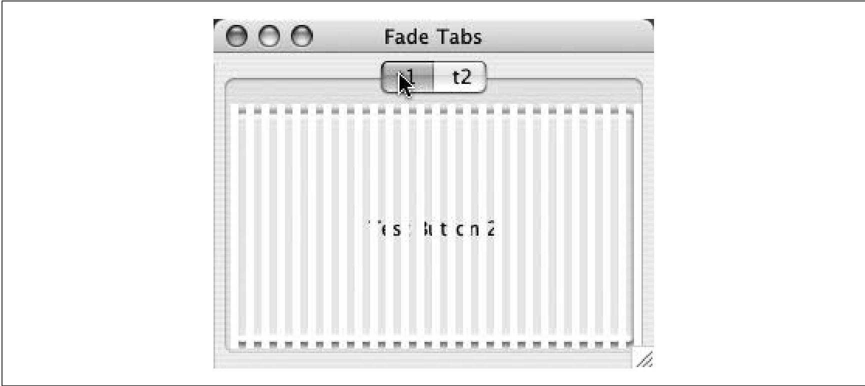


Figure 1-25. Tab transition with a venetian blinds effect