

SWING HACKS™

*Tips & Tools for
Building Killer GUIs*



O'REILLY®

Joshua Marinacci & Chris Adamson

HACK
#89

Fun with Keyboard Lights

Flash the Caps Lock, Num Lock, and Scroll Lock keys for extra user feedback.

The AWT and Swing APIs are huge and full of robust components and frameworks for building big applications. They also have some dark corners where the lesser-known functions live. While cruising through the JavaDoc for `java.awt.Toolkit`, I ran across a function I had never noticed before, despite it being in the API for over four years. This hack explores building a keyboard busy indicator using the `Toolkit.setLockingKeyState()` function.

The root class of AWT, `Toolkit`, has a very interesting little function: `setLockingKeyState()`. You pass it the `KeyEvent` for the key you want to lock down and turn it on or off with the `boolean`. For most keyboards, this means the Caps Lock, Num Lock, and Scroll Lock keys (some keyboards may also have a Kana lock for Kanji support). Now that you have this nifty little function, what should you do with it?

My first thought was a busy cursor. If you've got three lights in a row, why not blink them off and on in sequence? The code in [Example 12-2](#) will flip each light on and off in order, creating a moving bar effect (depending on the order of your keyboard LEDs).

Example 12-2. Lights, camera, action

```
class SpinnerThread extends Thread {
    private boolean go;
    public void quit() {
        go = false;
    }
    public void run() {
        go = true;
        // get a toolkit
        Toolkit tk = Toolkit.getDefaultToolkit();

        // save the old key states
        boolean old_num, old_caps, old_scroll;
        old_num = tk.getLockingKeyState(KeyEvent.VK_NUM_LOCK);
        old_caps = tk.getLockingKeyState(KeyEvent.VK_CAPS_LOCK);
        old_scroll = tk.getLockingKeyState(KeyEvent.VK_SCROLL_LOCK);

        // set all keys to off
        tk.setLockingKeyState(KeyEvent.VK_NUM_LOCK, false);
        tk.setLockingKeyState(KeyEvent.VK_CAPS_LOCK, false);
        tk.setLockingKeyState(KeyEvent.VK_SCROLL_LOCK, false);
    }
}
```

Fun with Keyboard Lights

SpinnerThread is a Runnable implementation, meaning you can launch it with new Thread(new SpinnerThread()).start(). The run() method starts an infinite loop, ending only when the quit() method is called. The first thing to notice is that the code saves the existing state of the buttons so that it can restore them later. If you had Caps Lock on you'd probably still want it on once the busy cursor leaves. Next, it sets all of the key states to false. This puts them into a known position so that the animation looks right:

```
int key = -1;
boolean state = false;
// loop through 100 times
int counter = 0;
while(go) {
    // select each key every 3rd time
    if(counter%3 == 0) { key = KeyEvent.VK_NUM_LOCK; }
    if(counter%3 == 1) { key = KeyEvent.VK_CAPS_LOCK; }
    if(counter%3 == 2) { key = KeyEvent.VK_SCROLL_LOCK; }
    // flip the state
    state = tk.getLockingKeyState(key);
    tk.setLockingKeyState(key,!state);
    // sleep for 500 msec
    try { Thread.currentThread().sleep(500);
    } catch (InterruptedException ex) {}

    // increment counter
    counter++;
}

// restore the key settings
tk.setLockingKeyState(KeyEvent.VK_NUM_LOCK,old_num);
tk.setLockingKeyState(KeyEvent.VK_CAPS_LOCK,old_caps);
tk.setLockingKeyState(KeyEvent.VK_SCROLL_LOCK,old_scroll);
```

Next comes the loop, which flips one key each time through the loop, cycling between the three keys. This is what produces the actual animation. Once the loop is finished, the keyboard states are restored.

Revisiting this hack (I first wrote it for a blog on <http://www.java.net>), I have been looking for other interesting ideas. One popped out at me as being truly annoying, so I chose to include it here. The class in [Example 12-3](#) flashes the Scroll Lock key on every keystroke. Thus, your computer will keep in time with your typing. Is this the future of human-computer evolution, or just plain annoying?

The key, so to speak, of this hack is once again the Toolkit object. It contains a very interesting method, addAWTEventListener(), that lets you add a listener for every event dispatched throughout the JVM. This is equivalent to putting a listener on each component in your entire program. This method

is mainly intended for debuggers and testing tools, but I've used it to listen for application-wide keystroke events.

Example 12-3. Scroll Lock keeping up with typing

```
public class KeyboardFlasher implements AWTEventListener {

    public static void main(String[] args) {
        Toolkit tk = Toolkit.getDefaultToolkit();
        KeyboardFlasher flasher = new KeyboardFlasher();
        tk.addAWTEventListener(flasher, AWTEvent.KEY_EVENT_MASK);

        JFrame frame = new JFrame("Hack #89: Fun with Keyboard Lights");

        JTextField tf = new JTextField("this is some text");
        frame.getContentPane().add(tf);
        frame.pack();
        frame.setVisible(true);
    }
}
```

`KeyboardFlasher` implements the `AWTEventListener` interface and the `main` method adds a new flasher to the toolkit as a listener. `tk.addAWTEventListener`'s second argument, `AWTEvent.KEY_EVENT_MASK`, indicates that the listener should receive key events only. If I wanted the program to also look for mouse clicks, then I would do a bitwise OR (`|`) of `KEY_EVENT_MASK` with `MOUSE_EVENT_MASK`. Finally, the `main()` method creates a text field in a frame to test out generating keyboard events:

```
public void eventDispatched(AWTEvent evt) {
    if(evt instanceof KeyEvent) {
        KeyEvent kevt = (KeyEvent)evt;
        if(kevt.getID() == KeyEvent.KEY_PRESSED) {
            System.out.println("key event: " + evt);
            if(kevt.getKeyCode() != KeyEvent.VK_SCROLL_LOCK) {
                flipScrollLock();
            }
        }
    }
}

public void flipScrollLock() {
    Toolkit tk = Toolkit.getDefaultToolkit();
    boolean state = tk.getLockingKeyState(KeyEvent.VK_SCROLL_LOCK);
    tk.setLockingKeyState(KeyEvent.VK_SCROLL_LOCK, !state);
}
```

As an implementation of `AWTEventListener`, `KeyboardFlasher` has one required method only, `eventDispatched()`. This receives all events generated in the system. This implementation first checks that the event is indeed a keyboard event, and then checks if it is a `KEY_PRESSED` event. This will dis-

tinguish between the key going down and the key going up. Then, the method checks to make sure that the key pressed isn't the Scroll Lock key.



If I didn't add this check, the code would go into an infinite loop, as each flip of the Scroll Lock key would generate a new event and trigger a new flip.

Finally comes `flipScrollLock()`, which does exactly what it suggests: flips the state of the Scroll Lock key each time it is called. In other words, it takes two strokes to complete a cycle. For a faster effect, you could make the light turn on with the downward stroke and then off again with the upward stroke. You would just need to call `flipScrollLock()` twice, once for each event.

`setLockingKeyState()` is one of those nut-ball little functions that seems to have no purpose, but it sure lets us have some fun. What other uses can you think up? A new email alert? A three-bar sound meter? A Wi-Fi strength indicator?