

Desktop Java Live

April **2005**

The following is a sample chapter from SourceBeat's Desktop Java Live by Scott Delap. You can access more information about this book and our other titles at www.sourcebeat.com.

SourceBeat's publishing model is quite different than traditional publishers in that our books are updated monthly and accessed electronically via secure PDF files. Readers subscribe annually to each title for \$29.95 and receive updates throughout the subscription life.

Thank you for your interest in this title.

SourceBeat

Swing Threading

Desktop applications are often multi-threaded in nature. While adding benefits to responsiveness and performance, multi-threading adds coding complexity. You can write Swing applications without using multiple threads. However, once you cross the line into having a multi-threaded application, you should abide by Swing's threading rules. Otherwise, your application may behave in an unpredictable manner. In this chapter, you'll explore how Swing uses threads. You'll then view a number of threading-related resources in the Swing API. You'll also be introduced to three structured Swing threading solutions: SwingWorker, Foxtrot, and Spin. Finally, you'll learn how to detect incorrect thread usage in Swing with a custom repaint manager.

Swing, AWT, and the EDT

Swing is built on top of Java's *Abstract Windowing Toolkit* (AWT). AWT uses native code to access the underlying operating system's components. Each component in AWT, such as a button, has a corresponding native peer that requests use, for example when it needs to draw that it has been pressed. Swing, on the other hand, does not use native peer components except for a few rare exceptions. Instead, it paints components completely in Java on top of a canvas provided by AWT. An example of this is shown in Chapter 4; the `AnimatedGradientButtonUI` paints a special effect when a button has focus. Because Swing exists on top of AWT, it takes advantage of the AWT event-dispatching framework.

When you start a Swing application, three threads are executed: the *main application thread*, a *toolkit thread* and the *event dispatch thread*. The main application thread calls the `main()` method to start the application. The toolkit thread dispatches events from the native operating system, which make their way to the AWT event dispatch thread, called the *EDT*. The EDT handles two primary functions. First, it dispatches events, such as mouse clicks, to the appropriate Swing component. Second, it executes paint operations of Swing components.

Note: When a thread other than the EDT executes code, it is often described as “off the EDT.” Correspondingly, code executed by the EDT when it processes events is often described as “on the EDT.” Finally, using threads other than the EDT in your application causes your application to be “multi-threaded.”

When a mouse click occurs, for instance on a `JButton`, an event is generated. AWT dispatches this event to the `JButton`, which fires a mouse event to its listeners, one of which is its UI delegate. The delegate then paints the `JButton`'s visible state to appear pressed. From the time the event is dispatched until the paint operation is performed, the code is executed on the EDT.

You may be wondering what this has to do with threading, because AWT seems to have the situation handled. AWT, for the most part, has things under control; however, if you've written Swing code, you've probably encountered times when your application froze. Figure 5.1 and the subsequent code show a situation that will cause a freeze.

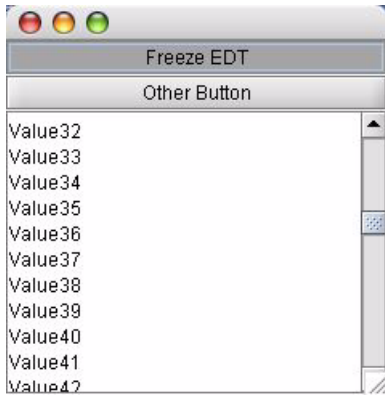


Figure 5.1: A JButton Experiencing a Freeze after Mouse Down

```

public JPanel createPanel() {
    JPanel panel = new JPanel();
    panel.setLayout(new FormLayout("p:g", "p, p, f:d:g"));
    CellConstraints cc = new CellConstraints();

    JButton button = new JButton(new FreezeAction());
    panel.add(button, cc.xy(1, 1));

    JButton button1 = new JButton("Other Button");
    panel.add(button1, cc.xy(1, 2));

    Vector values = new Vector();
    for (int i = 0; i < 100; i++) {
        values.add("Value" + i);
    }

    JList list = new JList(values);
    panel.add(new JScrollPane(list), cc.xy(1, 3));

    return panel;
}

private class FreezeAction extends AbstractAction {

```

```
public FreezeAction() {
    super("Freeze EDT");
}

public void actionPerformed(ActionEvent e) {
    try {
        Thread.sleep(5000);
    } catch (InterruptedException e1) {
    }
}
}
```

At first glance, you see two buttons and a list contained in a panel. However, notice the **Thread.sleep()** method calls in the appropriately named **FreezeAction**. Clicking the Freeze EDT button will result in the button staying “stuck” in a pressed state for five seconds. The button doesn’t return to its normal state because the **sleep()** method is applied to the EDT, and stops any further processing of input or paint events.

The Event Queue and Processing

Events, such as moving the mouse, are created and placed in a queue called the *EventQueue*. The EDT then processes these events sequentially. Figure 5.2 and the subsequent code expand on the previous example, illustrating some of the input events processed by the EDT.

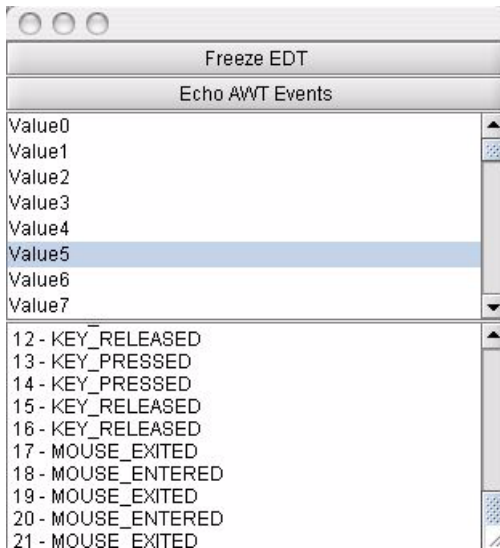


Figure 5.2: Echoing Events as They Are Processed by the EDT

In this example, an `AWTEventListener` is added to the default toolkit.

```
public JPanel createPanel() {
    JPanel panel = new JPanel();
    panel.setLayout(new FormLayout("p:g", "p, p, p, f:d:g"));
    CellConstraints cc = new CellConstraints();
    ...

    JTextArea textArea = new JTextArea();
    textArea.setAutoscrolls(false);

    JScrollPane textScrollPane = new JScrollPane(textArea);
    textScrollPane.setHorizontalScrollBarPolicy(
```

```
        JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
textScrollPane.setVerticalScrollBarPolicy(
        JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);

panel.add(textScrollPane, cc.xy(1, 4));

Toolkit.getDefaultToolkit().addAWTEventListener(
        new EchoingAWTListener(textArea),
        AWTEvent.KEY_EVENT_MASK | AWTEvent.MOUSE_EVENT_MASK);

return panel;
}

private class ToggleEchoAction extends AbstractAction {

    public ToggleEchoAction() {
        super("Echo AWT Events");
    }

    public void actionPerformed(ActionEvent e) {
        EDTListenerExample.this.echo = !EDTListenerExample.this.echo;
        if (EDTListenerExample.this.echo) {
            putValue(Action.NAME, "Disable Echo");
        } else {
            putValue(Action.NAME, "Echo AWT Events");
        }
    }
}

private class EchoingAWTListener implements AWTEventListener {

    private JTextArea textArea;

    private int count;

    public EchoingAWTListener(JTextArea textArea) {
        this.textArea = textArea;
    }
}
```

```
}

public void eventDispatched(AWTEvent event) {
    if (EDTListenerExample.this.echo) {
        String eventName = getEventType(event.getID());
        if (eventName != null) {
            this.count++;
            textArea.append("\n" +
                this.count + " - " + eventName);
        }
    }
}

private String getEventType(int id) {
    switch (id) {
        case KeyEvent.KEY_PRESSED:
            return "KEY_PRESSED";
        case KeyEvent.KEY_RELEASED:
            return "KEY_RELEASED";
        case MouseEvent.MOUSE_PRESSED:
            return "MOUSE_PRESSED";
        case MouseEvent.MOUSE_RELEASED:
            return "MOUSE_RELEASED";
        case MouseEvent.MOUSE_MOVED:
            return "MOUSE_MOVED";
        case MouseEvent.MOUSE_DRAGGED:
            return "MOUSE_DRAGGED";
        case MouseEvent.MOUSE_ENTERED:
            return "MOUSE_ENTERED";
        case MouseEvent.MOUSE_EXITED:
            return "MOUSE_EXITED";
        case MouseEvent.MOUSE_WHEEL:
            return "MOUSE_WHEEL";
        default:
```

```
        return null;
    }
}
}
```

In this example, an `AWTEventListener` is added to the default toolkit.

Note: An `AWTEventListener` can slow your application down because it is executed on every AWT event.

This listener is triggered every time an event is processed. Clicking the Echo AWT Events button lists (or echoes) all mouse and keyboard events in the text area. For instance, if you run this example and click the Freeze EDT button, ‘`MOUSE_PRESSED`’ will appear in the text area, which corresponds to clicking the mouse. At this point, the `Thread.sleep()` method begins to run, so the EDT can no longer process events. As a result, events stay queued; no events are processed until the `Thread.sleep()` method is finished. Upon the method’s completion, the queued events are processed sequentially (including the `MOUSE_RELEASED` event after clicking the button).

Freeze issues in most Swing applications occur when long running (or *heavyweight*) operations on the EDT are executed, which blocks the processing of other events. Often, it is not obvious that such operations are running on the EDT. By default, all Swing operations including, `actionPerformed()` methods in actions and listeners are executed on the EDT. You will see a number of ways to free up the EDT later in this chapter. First, you need to be aware of the *Single Threading Rule* of Swing.

The Single Threading Rule

The Swing Single Threading Rule states the following:

To avoid the possibility of deadlock, you must take extreme care that Swing components and models are created, modified, and queried only from the event-dispatching thread.¹

1. Sun Microsystems, Inc. “How to Use Threads.” Sun Microsystems, Inc. 1995-2005. <http://java.sun.com/docs/books/tutorial/uiswing/misc/threads.html>

Swing was designed with the assumption that all access to Swing components is executed on the EDT. Consequently, Swing components are not designed to be thread-safe. Therefore, any time you access Swing components from threads other than the EDT, common threading problems may exist, such as race conditions and deadlocks. The Single Threading Rule has evolved slightly over the years. It was once thought that components could be accessed from the EDT as long as you didn't modify them after they had been *realized*, that is, painted or ready to be painted. Using this definition, the following code was considered safe:

```
public static void main(String[] args) {  
    JFrame frame = new JFrame();  
    frame.getContentPane().add(new JButton("Button"), BorderLayout.CENTER);  
    frame.show();  
}
```

However, Sun's engineers found that in a few of the Swing tutorial examples, even unrealized components being accessed from the EDT caused problems. For instance, one of the official Swing tutorial examples intermittently had deadlock problems. Therefore, component creation should only occur on the EDT just like other Swing component manipulation. You will learn how to put code, such as the frame creation example, on the EDT from another thread later in this chapter.

Why Single Threading?

The single-threaded architecture of Swing is useful for a number of reasons.

- ▶ **Simplicity** – Since all painting and events are handled on the EDT, you don't have to worry about threading until you run into situations like the ones previously described. Instead of having to learn about interacting with multiple threads, you can code your application and it will simply run on the EDT. Having everything execute on the EDT also allows you to extend Swing components without having to worry about threading issues behind the scenes.
- ▶ **Reliability** – Since all events are queued, they are always dispatched in a predictable order.
- ▶ **Debugging/Testing** – If you've ever dealt with a multi-threaded application, you know the difficulties of debugging and testing it. Tracing execution and reproducible test cases is much more difficult with multiple threads interacting.

When To Multi-Thread

Now that you've seen the benefits of the single-threaded model, you may wonder when you should use another thread to execute code. The primary indicator is any long running process that prevents the EDT from processing events from the event queue in a timely manner. A specific example of this would be a code segment that causes extensive disk IO. Another example is a thread to wait for an event. If the EDT were to block and wait for the event, no painting or processing of events would occur.

How to Thread Your Application

This section will cover how to actually multi-thread your applications. First, common Swing methods related to interacting with threads will be discussed. Then three structured threading solutions will be covered: `SwingWorker`, `Foxtrot`, and `Spin`.

Non-EDT Safe Swing Methods

Before covering the proper way to access Swing components from non-EDT threads, it is important to mention a few methods in Swing that can be accessed from threads other than the EDT safely. These include `repaint()`, `revalidate()`, and `invalidate()`. You will learn how write safe methods using the `SwingUtilities.invokeLater()` method later in this chapter.

Note: It is also safe to add and remove listeners from threads other than the EDT.

Swing Threading Utility Methods

Swing provides a number of methods that are useful in thread-related programming via the `SwingUtilities` class. The methods, `invokeLater()` and `invokeAndWait()` form the foundation for allowing asynchronous operations in a Swing application. Execution can be forked off on another thread from the EDT. Interaction with Swing components can then be placed back in the `EventQueue` for execution on the EDT using these methods. Each posts a `Runnable` that is executed on the EDT.

InvokeLater

The `SwingUtilities.invokeLater(Runnable runnable)` method posts its `Runnable` parameter to the EDT, and then immediately continues execution on the thread it was called from. Figure 5.3 and subsequent code show an example of using `invokeLater()`.

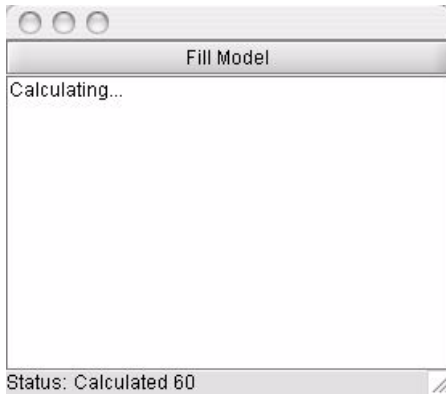


Figure 5.3: InvokeLater Filling the List Model in the Background While Updating Status to the User

```
private JLabel statusArea;
private DefaultListModel listModel;

public JPanel createPanel() {
    JPanel panel = new JPanel();
    panel.setLayout(new FormLayout("p, 2dlu, p:g", "p, f:d:g, p"));
    JButton button = new JButton(new LongRunningModelFillAction());
    CellConstraints cc = new CellConstraints();

    panel.add(button, cc.xywh(1, 1, 3, 1));

    JList list = new JList();
    this.listModel = new DefaultListModel();
    this.listModel.addElement("An Empty List Model");
    list.setModel(listModel);
    panel.add(new JScrollPane(list), cc.xywh(1, 2, 3, 1));

    panel.add(new JLabel("Status:"), cc.xy(1, 3));
    this.statusArea = new JLabel();
    panel.add(this.statusArea, cc.xy(3, 3));

    return panel;
}
```

```

}

private class LongRunningModelFillAction extends AbstractAction {
    public LongRunningModelFillAction() {
        super("Fill Model");
    }

    public void actionPerformed(ActionEvent e) {
        InvokeLaterExample.this.listModel.removeAllElements();
        InvokeLaterExample.this.listModel.addElement("Calculating...");
        PopulationRunnable populationRunnable = new PopulationRunnable();
        Thread populationThread = new Thread(populationRunnable);
        populationThread.start();
    }
}

private class PopulationRunnable implements Runnable {
    public void run() {
        final Object[] values = new Object[100];
        for (int i = 1; i <= 100; i++) {
            values[i - 1] = "Value" + i;

            if ((i % 10) == 0) {
                final int progress = i;
                SwingUtilities.invokeLater(new Runnable() {
                    'public void run() {
                        InvokeLaterExample.this.statusArea.setText("Calculated " + progress);
                    }
                });
            }
            try {
                Thread.sleep(50);
            } catch (InterruptedException e) {
            }
        }
    }
}

```

```
    }

    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            InvokeLaterExample.this.statusArea.setText("");

            InvokeLaterExample.this.listModel.removeAllElements();

            for (int i = 0; i < values.length; i++) {

                InvokeLaterExample.this.listModel.addElement(values[i]);
            }
        }
    });
}
```

The main components of the example are a button, list, and status area label. The button invokes the **LongRunningFillAction** when clicked. This action simulates a long running operation by starting a thread containing **PopulationRunnable**. This creates **String** values over the course of a few seconds and then populates them in the list.

A number of specifics are worth noting in relation to threading in this example. First, the list model is changed with the following code before the **PopulationRunnable** is started:

```
InvokeLaterExample.this.listModel.removeAllElements();
InvokeLaterExample.this.listModel.addElement("Calculating...");
```

This code is designed specifically to not violate the Single Threading Rule. Invoking these method calls before starting **PopulationRunnable** ensures that they will run on the EDT. When started, **PopulationRunnable** creates values while posting a **Runnable** to the EDT to update the status area label every 10 items. This update is placed back on the EDT for execution using **invokeLater()**. Since this method doesn't block execution as mentioned above, the **PopulationRunnable** continues creating the items that will be populated in the list model while the EDT is

processing events. Finally, you might wonder why this code is also executed using a **Runnable** and the **invokeLater()** method:

```
InvokeLaterExample.this.listModel.removeAllElements();

for (int i = 0; i < values.length; i++) {
    InvokeLaterExample.this.listModel.addElement(values[i]);
}
```

For each **addElement()** method call, the listeners of the model are notified. These listeners execute on the thread that caused the notification, because the model does not enforce execution on the EDT. The model has no way of knowing these notifications violate the Single Threading Rule. Specifically executing the model changes on the EDT solves this problem.

Note: A process still may cause issues even when it is executed on its own thread. All the threads are executed on the same computer, so they compete for the CPU's resources. Since CPU time is limited, the EDT may not receive as much time as it needs. If you experience problems, you can lower the priority of your worker threads. Here is how the priority of the example's worker thread is lowered.

```
populationThread.setPriority(Thread.MIN_PRIORITY);
```

InvokeAndWait

The **invokeAndWait()** method has a **Runnable** as a parameter just like **invokeLater()**. However, **invokeAndWait()** blocks execution of the thread invoking it until its **Runnable** finishes executing on the EDT.

Warning: In most cases, you should use **invokeLater()**. Since the calling thread is blocked until **invokeAndWait()** returns, it extends the period of any threading locks the calling thread has acquired. The more locks and time keeping them by threads in your application, the greater chance of threading issues like deadlock.

The example shown in Figures 5.4 and 5.5 and the subsequent code illustrate using **invokeAndWait()**.

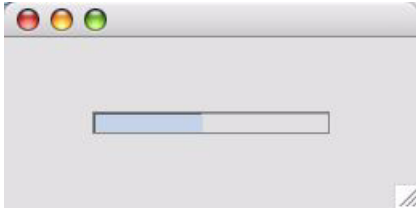


Figure 5.4: InvokeAndWait Progress Bar



Figure 5.5: InvokeAndWait Prompt

```
private Thread progressThread;
private JProgressBar progressBar;
private boolean keepRunning;

public JPanel createPanel() {
    this.keepRunning = true;
    this.progressThread = new ProgressThread();
    JPanel panel = new JPanel();
    panel.setLayout(new FormLayout("p:g, p, p:g", "p:g, p, p:g"));

    CellConstraints cc = new CellConstraints();

    this.progressBar = new JProgressBar();
    this.progressBar.setMaximum(100);

    panel.add(this.progressBar, cc.xy(2, 2));
    System.out.println(keepRunning);
    this.progressThread.start();
    return panel;
}
```

```
}

public String getTitle() {
    return "InvokeAndWait Example";
}

public void endExample() {
    this.keepRunning = false;
}

private class ProgressThread extends Thread {
    public void run() {
        int count = 0;
        while(keepRunning) {
            try {
                Thread.sleep(40);
            } catch (InterruptedException e) {
            }

            final int cval = count;
            SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    progressBar.setValue(cval);
                }
            });
            count++;

            if (count == 101) {
                count = 0;
            } else if (count == 50) {
                final int[] returnValue = new int[1];
                try {
                    SwingUtilities.invokeAndWait(new Runnable() {
                        public void run() {
```

```
        returnValue[0] = JOptionPane.  
            showConfirmDialog(  
                progressBar,  
                "Would you like  
                to reset progress?",  
                "Prompt",  
                JOptionPane.YES_NO_  
                OPTION);  
    }  
});  
if (returnValue[0] == JOptionPane.YES_OPTION) {  
    count = 0;  
}  
} catch (InterruptedException e) {  
    e.printStackTrace();  
} catch (InvocationTargetException e) {  
    e.printStackTrace();  
}  
}  
}  
}
```

This example starts a thread to update the state of the progress bar. Each update is posted on the EDT using `invokeLater()`. When the state reaches 50%, the thread prompts the user to continue updating the progress bar to 100% or to reset it to 0%. The creation of this option dialog is done with `invokeAndWait()`, which provides a number of advantages:

- ▶ The progress thread is suspended until the `invokeAndWait()` `Runnable` is processed. This prevents the progress bar from continuing to update towards 100%.
- ▶ Using `invokeAndWait()` allows you to gather input from the user on the EDT and then continue processing on the same non-EDT thread.
- ▶ The `ProgressThread` relies on input from the confirmation dialog. This input passes between the EDT and the `ProgressThread` using a final `int[]` variable. The main problem is finding a way to execute the call from `ProgressThread` so that it knows the `int[]` value has been filled. If you use `invokeLater()` instead, the `int[0]` value must be initialized to some indicator value like -1 and then polled by the `ProgressThread` until the value changes. Using

`invokeAndWait()` guarantees that the **Runnable** will be processed before the **ProgressThread** continues and that the `int[0]` value will be populated before it is needed.

The Inner Workings of InvokeLater and InvokeAndWait

So how do these methods actually execute the code on the EDT? Each method delegates to a **static** method on the class **EventQueue** with the same name. In the case of `invokeLater()`, the **Runnable** is wrapped in an **InvocationEvent** object and posted on the **EventQueue**. As the EDT processes events on the **EventQueue**, they are dispatched or sent to their appropriate destination. For example, **ComponentEvents** are dispatched to their respective component for processing. An **InvocationEvent** implements a special interface called **ActiveEvent**. Instead of the event being handed off for dispatching, **ActiveEvents** have a `dispatch()` method, which is invoked. In the case of **InvocationEvent**, the `run()` method of the **Runnable** it wraps is executed when `dispatch()` is called.

The `invokeAndWait()` method takes this execution a step further. Before the **Runnable** is posted to the event queue, a lock object is created. The **InvocationEvent** wrapper created contains both the **Runnable** and the lock object. After the **InvocationEvent** has been posted to the queue, the `wait()` method is called by the posting thread on the lock object. Invoking this method causes the posting thread to pause execution and wait. When the **InvocationEvent**'s **Runnable** is finished running, its `dispatch()` method calls `notifyAll()` on the same lock object. This causes the posting thread that had called the `wait()` method on the object to resume execution.

The Correct Way to Create Components in a Main Method

Earlier it was mentioned that accessing unrealized components from the EDT is no longer recommended. Now that you've been introduced to `invokeLater()` and `invokeAndWait()`, you can look at the recommended way to start your GUI in the main thread. The following code shows wrapping component creation in a **Runnable** and posting it on the EDT using `invokeLater()`.

```
public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            JFrame frame = new JFrame();
            frame.getContentPane().add(new JButton("Button"),
                BorderLayout.CENTER);
        }
    });
}
```

```
        frame.show();
    }
});
}
```

isEventDispatchThread()

Sometimes, when you write a multi-threaded application, you may not know the thread your code segments are executed on until runtime. You can use `SwingUtilities.isEventDispatchThread()` to make methods in your code EDT thread safe by checking the thread they are being executed on and responding appropriately. This **static** method will return a **boolean** value of true if it is executed from the EDT. Figure 5.6 and the subsequent code show an example using `isEventDispatchThread()`.



Figure 5.6: Random Colors in a Table Generated by Both a Thread and Button

```
public JPanel createPanel() {
    this.tableModel = new ColorTableModel();

    ...
}
```

```

panel.add(new JButton(new RandomColorAction()), cc.xy(3, 7));
this.keepRunning = true;
this.colorShadeThread = new Thread(new RandomColorShadeRunnable());
this.colorShadeThread.start();
return panel;
}

private class RandomColorAction extends AbstractAction {
    public RandomColorAction() {
        super("Create Random Color");
    }

    public void actionPerformed(ActionEvent e) {
        IsEDTExample.this.tableModel.generateRandomColor(VARIABLE);
    }
}

private class ColorTableModel extends AbstractTableModel {
    private Color[][] colors = new Color[3][3];

    ...

    public void generateRandomColor(int type) {
        Random random = new Random(System.currentTimeMillis());
        final int row = random.nextInt(SIZE);
        final int column = random.nextInt(SIZE);
        final Color color;
        if (type == RED) {
            color = new Color(random.nextInt(256), 0, 0);
        } else if (type == BLUE) {
            color = new Color(0, 0, random.nextInt(256));
        }
    }
}

```

```
        } else if (type == GREEN) {
            color = new Color(0, random.nextInt(256), 0);
        } else {
            color = new Color(random.nextInt(256),
                random.nextInt(256), random.nextInt(256));
        }

        if (SwingUtilities.isEventDispatchThread()) {
            colors[row][column] = color;
            fireTableCellUpdated(row, column);
        } else {
            SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    colors[row][column] = color;
                    fireTableCellUpdated(row, column);
                }
            });
        }
    }
}

private class ColorRenderer implements TableCellRenderer {
    private JLabel label;

    public ColorRenderer() {
        label = new JLabel();
        label.setOpaque(true);
        label.setPreferredSize(new Dimension(100, 100));
    }

    public Component getTableCellRendererComponent(JTable
        table, Object value, boolean isSelected,
        boolean hasFocus, int row, int column) {
```

```

        label.setBackground((Color) value);
        return label;
    }
}

private class RandomColorShadeRunnable implements Runnable {
    public void run() {
        while (keepRunning) {
            System.out.println("thread");
            tableModel.generateRandomColor(threadShade);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
            }
        }
    }
}

```

This example contains a **JTable** with a custom **ColorTableModel** and **ColorRenderer**. The **ColorTableModel** contains a special method called **generateRandomColor()**. This method will generate a random color based on the shade constant supplied as a parameter. It then updates the model and fires an update event to the table. Because this update event is accessing the table, it needs to be EDT thread safe. This is accomplished by using **SwingUtilities.isEventDispatchThread()** to check the thread that is invoking the **generateRandomColor()** method and posting a **Runnable** to the EDT if it is a non-EDT thread. This code structure then allows both the **RandomColorAction** and the **RandomColorShadeRunnable** to invoke the same method.

Timers

Another thread-related solution that is included with Swing is **javax.swing.Timer**. **Timer** allows you to fire action events at a specified relative time value. Figure 5.7 and subsequent code show an example using **Timer**.



Figure 5.7: A Clock and Progress Bar Driven by Timer

```
public class TimerExample extends AbstractCodeExample implements ActionListener {  
    private Timer timer = new Timer(100, this);  
    private JLabel clockLabel;  
    private Calendar calendar = Calendar.getInstance();  
    private SimpleDateFormat dateFormat = new SimpleDateFormat("h:mm ss a");  
    private JProgressBar secondsProgressBar = new JProgressBar(0, 1000);  
  
    public JPanel createPanel() {  
        JPanel panel = new JPanel();  
        panel.setLayout(new FormLayout("2dlu, p:g, 2dlu",  
            "2dlu, p, 2dlu, p, 2dlu, p"));  
        CellConstraints cc = new CellConstraints();  
        this.clockLabel = new JLabel("Clock Stopped");  
        this.clockLabel.setFont(this.clockLabel.getFont().  
            deriveFont(Font.BOLD, 16));  
        this.clockLabel.setHorizontalAlignment(JLabel.CENTER);  
  
        this.secondsProgressBar.setVisible(false);  
        this.secondsProgressBar.setForeground(Color.blue);  
    }  
}
```

```

panel.add(this.clockLabel,
          cc.xy(2, 2, CellConstraints.CENTER,
              CellConstraints.CENTER));
panel.add(secondsProgressBar, cc.xy(2, 4));
panel.add(new JButton(new ToggleClockAction()), cc.xy(2, 6));

return panel;
}

private class ToggleClockAction extends AbstractAction {
    private String startClock = "Start Clock";
    private String stopClock = "Stop Clock";
    private String clockStopped = "Clock Stopped";

    public ToggleClockAction() {
        super();
        putValue(Action.NAME, startClock);
    }

    public void actionPerformed(ActionEvent e) {
        if (TimerExample.this.timer.isRunning()) {
            putValue(Action.NAME, startClock);
            TimerExample.this.timer.stop();

            TimerExample.this.secondsProgressBar.setVisible(false);
            TimerExample.this.clockLabel.setText(clockStopped);
        } else {
            putValue(Action.NAME, stopClock);
            //Call Action Performed
            //To Initialize Time Before Timer is Started.
            // Null is ok since we ignore the event.
            TimerExample.this.actionPerformed(null);

            TimerExample.this.secondsProgressBar.
                setVisible(true);
        }
    }
}

```

```
        TimerExample.this.timer.start();
    }
}

public void actionPerformed(ActionEvent e) {
    this.calendar.setTimeInMillis(System.currentTimeMillis());

    this.clockLabel.setText(
        this.dateFormat.format(this.calendar.getTime()));
    int milliseconds = this.calendar.get(Calendar.MILLISECOND);

    this.secondsProgressBar.setValue(milliseconds);
}

public String getTitle() {
    return "Timer Example";
}
}
```

This example shows an instance of **Timer** constructed at the beginning. The first parameter passed into the constructor is the interval that **Timer** should fire on, which in this case is 100 milliseconds. The second parameter is an action listener for the **Timer** instance. The example implements **ActionListener** and is therefore used in the constructor as the argument. By default, a timer will loop when started. When the button is clicked, the **Timer** instance is started or stopped. Every 100 milliseconds, it will invoke the **actionPerformed()** method in the example. This method updates the clock label and the progress bar.

The Inner Workings of Timer

Timer is a particularly useful class in Swing in relation to threading. You may have noticed that the example contains no usages of **invokeLater()** or **invokeAndWait()**. This is because **Timer** instances automatically post calls to their action listeners back onto the EDT using **invokeLater()** internally. Another key feature of **Timer** is the threading behind it. At first glance, you might think that each **Timer** has a thread running behind it. This could get quite expensive from a resource perspective if you had many timers running in your application. Instead, all instances of **Timer** share one daemon thread called **TimerQueue**. This thread continuously waits

until the next **Timer** scheduled to fire is ready. At this time, its listeners are notified, and it is rescheduled in the queue if you set it to repeat. Timers also have a `coalesce` setting, which you can use if you have a busy EDT thread that cannot process **Timer** events as quickly as they are being posted. Instead of allowing the events to back up in the **TimerQueue** and continue to flood the EDT, the backed up events will be coalesced and only one will be fired. A new firing time will then be scheduled.

Structured Swing Threading Solutions

As you've just seen, you can use the static methods in **SwingUtilities** to handle the threading issues that you are likely to encounter in your applications. However, they don't naturally lead to elegantly structured code that is simple to write and understand. In this section, you will be introduced to the structured solutions of **SwingWorker**, **Foxtrot**, and **Spin** that make sure your applications follow the Single Threading Rule in Swing.

SwingWorker

The Swing Threading examples shown so far have handled threading solutions adequately, but they have lacked a formalized pattern of execution. Swing engineers recognized this fact, so they created the **SwingWorker** utility class. **SwingWorker** is not part of the standard Java distribution, and has had a number of homes and revisions throughout the years. The latest version currently lives as a part of the JDesktop Network Components (JDNC) project at <https://jdnc.dev.java.net>.

SwingWorker is an abstract class that provides a structure for asynchronous threading in an EDT-safe manner. It uses the **SwingUtilities** methods internally to provide a structure that hides the complexities of directly working with code on and off the EDT. The **SwingWorker** class defines the following public methods:

```
public void start();
```

This method starts a worker thread to perform a task. The worker thread then invokes the `construct()` method off the EDT.

```
public abstract Object construct();
```

This is an abstract method that must be implemented when extending **SwingWorker** with a concrete implementation. The code in this method carries out the execution that you want to do off the EDT thread. Any information that needs to be used back on the EDT can be returned in the form of an object.

public void interrupt();

Invoking this method will interrupt the worker thread.

public void finished();

This method is executed on the EDT after the worker thread has finished.

public Object get();

This method can be used by code in the **finish()** method on the EDT to get the value returned by the **construct()** method after **finish()** has executed off the EDT.

Here is an example of the action rewritten using `SwingWorker`.

```
private class LongRunningModelFillAction extends AbstractAction {
    public LongRunningModelFillAction() {
        super("Fill Model");
    }

    public void actionPerformed(ActionEvent e) {
        PopulationWorker populationWorker = new PopulationWorker();
        populationWorker.start();
    }
}

private class PopulationWorker extends SwingWorker {
    public Object construct() {
        Object[] values = new Object[100];
        for (int i = 1; i <=100; i++) {
            values[i - 1] = "Value" + i;
        }

        if ((i % 10) == 0) {
            final int progress = i;
            SwingUtilities.invokeLater(new Runnable() {
                public void run() {

```

```

        SwingWorkerExample.this.statusArea.setText("Calculated " + progress);
    }
    });
}
try {
    Thread.sleep(50);
} catch (InterruptedException e) {
}
}
return values;
}

public void start() {
    SwingWorkerExample.this.listModel.removeAllElements();
    SwingWorkerExample.this.listModel.addElement("Calculating...");
    super.start();
}

public void finished() {
    SwingWorkerExample.this.statusArea.setText("");
    SwingWorkerExample.this.listModel.removeAllElements();
    Object[] values = (Object[]) getValue();
    for (int i = 0; i < values.length; i++) {
        SwingWorkerExample.this.listModel.addElement(values[i]);
    }
}
}
}

```

A concrete implementation of the abstract **SwingWorker** class named **PopulationWorker** has been created. The **PopulationWorker**'s **start()** method prepares the list model for population and then calls **super.start()** to start the worker thread. The worker thread then calls **construct()**, which populates an **Object[]** with values and then returns it. This **Object[]** is now accessible by the **get()** method of the **PopulationWorker**. Before stopping, the worker thread internally posts a **Runnable** to the EDT that calls the **finished()** method of **PopulationWorker**. This method gets the value generated by the worker thread and uses it to populate the model.

A Synchronous Threading Solution for Swing

All the examples thus far have moved work off the EDT, which frees it up so that it can continue to process events. A synchronous threading solution blocks execution of code on the EDT, while work is done by another thread off the EDT. You are probably wondering how can you have a blocking synchronous threading solution. At first glance, you might think such an implementation would prevent the EDT from processing events; thus resulting in a UI freeze. Swing provides a hint to this solution with its implementation of modal dialogs. When the following code is executed, a dialog is displayed.

```
public void actionPerformed(ActionEvent e) {  
    JDialog dialog = new JDialog(JFrame)  
        SwingUtilities.getWindowAncestor(panel), true);  
    dialog.getContentPane().add(new JLabel("A Dialog"));  
    dialog.show();  
  
    //Isn't Executed Until Dialog is Closed ... How/Why?  
  
    putValue(Action.NAME, "Dialog Already Shown");  
}
```

The interesting part is that after the **Dialog.show()** method runs, the subsequent code isn't executed until after the dialog is closed. Yet the UI remains responsive while the dialog is displayed. How is the EDT thread being blocked and still processing events at the same time? Swing has an advanced technique in the implementation of **Dialog.show()**. If the thread executing **show()** is the EDT, it blocks the current EDT and starts a second EDT to process events while the current one is blocked.

Before showing you an API that takes advantage of this technique, here are a few reasons to consider synchronous threading as opposed to the asynchronous techniques that have been shown thus far:

- ▶ **Readability/Symmetry** – As you will see in the next section, code written with Foxtrot looks very similar to unthreaded code.
- ▶ **Exceptions** – Handling exceptions with asynchronous code is difficult. You have to write code that will manually check if an exception has occurred instead of just responding to one being thrown, which is normally the case in unthreaded code.

- ▶ **Maintainability/Debugging** – The possible problems greatly increase when your code forks in a different direction with an asynchronous solution.
- ▶ **Conversion of Legacy Code** – Since a synchronous threading solution is very similar to normal code from a structural standpoint, existing code can be modified to use such a solution quickly with minor effort.

Foxtrot

Foxtrot is an open source API (<http://foxtrot.sf.net>) that uses a technique similar to the one previously described to provide a synchronous threading solution for Swing. By moving the long running task off the EDT, it prevents a UI freeze. However, by blocking code execution on the EDT while still processing `EventQueue` events, the complexities of handling asynchronous execution of code are eliminated. The Foxtrot API has three main classes:

`foxtrot.Worker`

`foxtrot.Job`

`foxtrot.Task`

`Job` is an abstract class that is similar to a `Runnable` because it contains a unit of work. It contains one abstract method that you must implement:

```
public abstract java.lang.Object run()
```

Unlike a `Runnable`, a `Job`'s `run()` method returns an `Object`. A `Job` object is executed by calling the static method `Worker.post(Job job)`. This post method will block the EDT while the `Job` is being executed and return the `Object` that is returned from the `Job`'s `run()` method.

Here is the `LongRunningModelFillAction` from the `invokeLater()` example rewritten using a `Foxtrot Job()` instead.

```
private class LongRunningModelFillAction extends AbstractAction {
    public LongRunningModelFillAction() {
        super("Fill Model");
    }

    public void actionPerformed(ActionEvent e) {
        Foxtrot.JobExample.this.listModel.removeAllElements();
    }
}
```

```
FoxtrotJobExample.this.listModel.addElement("Calculating...");
Object[] values = (Object[]) Worker.post(new Job() {
    public Object run() {
        Object[] values = new Object[100];
        for (int i = 1; i <= 100; i++) {
            values[i - 1] = "Value" + i;

            if ((i % 10) == 0) {
                final int progress = i;
                //Only Needed Because of Intermediate
                //Status Otherwise this run() method
                //would not have any SwingUtilities
                //calls.

                SwingUtilities.invokeLater(new Runnable() {
                    public void run() {
                        FoxtrotJobExample.this.statusArea
                            .setText("Calculated "
                                + progress);
                    }
                });
            }
            try {
                Thread.sleep(50);
            } catch (InterruptedException ex) {
            }
        }

        return values;
    }
});

FoxtrotJobExample.this.statusArea.setText("");
FoxtrotJobExample.this.listModel.removeAllElements();
```

```

        for (int i = 0; i < values.length; i++) {
            FoxtrotJobExample.this.listModel.addElement(values[i]
        );
        }
    }
}

```

Using this structure, the `invokeLater()` method is now contained in an anonymous implementation of `Job`. Notice that the thread has been removed and all the code is now in the `LongRunningModelFillAction`'s `actionPerformed()` method. The code's structure and symmetry is improved by using synchronous threading. The code looks very similar to how a non-threaded implementation looks. When the `Job` is finished executing, it returns the `Object[]` of values, which is then returned by the `Worker.post()` method. At this point, blocking is stopped and execution continues on the EDT. The final part of the `actionPerformed()` method takes the `Object[]` and updates the `listModel`.

Besides the `Job` class, `Worker.post()` can also take a `Task` object as a parameter. A `Task` is just like a `Job` except that it can throw a checked `Exception`. The following code shows the example again with an `Exception` being thrown and handled in the `actionPerform()` method. Figure 5.8 shows the `Exception` being handled in the status area at the bottom of the window.

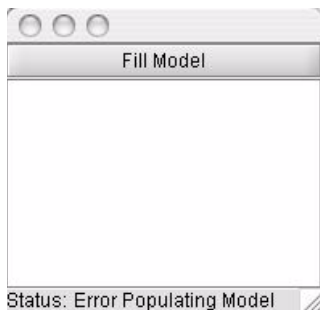


Figure 5.8: Foxtrot Task after an Exception Has Been Thrown

```

public void actionPerformed(ActionEvent e) {
    FoxtrotTaskExample.this.listModel.removeAllElements();
    FoxtrotTaskExample.this.listModel.addElement("Calculating...");
    try {
        Object[] values = (Object[]) Worker.post(new Task() {
            public Object run() throws Exception {

```

```
Object[] values = new Object[100];
for (int i = 1; i <= 100; i++) {
    values[i - 1] = "Value" + i;

    if ((i % 10) == 0) {
        final int progress = i;
        //Only Needed Because of Intermediate Status
        //Otherwise this run() method would not have any SwingUtilities calls.
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {

                FoxtrotTaskExample.this.statusArea.setText("Calculated " + progress);

            }
        });
    } else if (i == 51) {
        //Simulate And Exception
        throw new Exception("Error Populating Model");
    }
    try {
        Thread.sleep(50);
    } catch (InterruptedException ex) {
    }
}

return values;
});

FoxtrotTaskExample.this.statusArea.setText("");
FoxtrotTaskExample.this.listModel.removeAllElements();
for (int i = 0; i < values.length; i++) {
    FoxtrotTaskExample.this.listModel.addElement(values[i]);
}
} catch (Exception ex) {
    FoxtrotTaskExample.this.statusArea.
```

```

        setText("Error Populating Model");
        FoxtrotTaskExample.this.listModel.
            removeAllElements();
    }
}

```

Spin

The open source Spin API is the final structured threading solution discussed in this chapter (<http://spin.sf.net>). Spin takes a slightly different approach than the two solutions you have just seen. Instead of providing a structure for your code to conform to, Spin provides a mechanism to wrap your code with a proxy layer. This layer handles all the EDT interactions for you. However, Spin requires your non-EDT code to be in an implementation that has an interface. The following code shows how to proxy a bean with Spin:

```
bean = (Bean)Spin.off(bean);
```

Calls to the bean are “spun” off the EDT and onto a worker thread by a proxy that Spin wraps the bean in. This proxy uses the same sort of technique as Foxtrot to block execution while calling bean methods off the EDT.

If you want to have the bean call back to the UI, you can spin execution “over” to the EDT. The following code shows how to enable EDT-safe calls by adding a proxied **propertyChangeListener** to the bean:

```
bean.addPropertyChangeListener((PropertyChangeListener)Spin.over(gui));
```

The listener is proxied by Spin, which uses **invokeAndWait()** internally to call the UI.

Here is the **invokeLater()** example rewritten to take advantage of Spin.

```

private class LongRunningModelFillAction extends AbstractAction {
    public LongRunningModelFillAction() {
        super("Fill Model");
    }

    public void actionPerformed(ActionEvent e) {
        SpinExample.this.listModel.removeAllElements();
    }
}

```

```
SpinExample.this.listModel.addElement("Calculating...");

PopulationBean populationBean =
    (PopulationBean) Spin.off(new PopulationBeanImpl());

populationBean.addPopulationStateListener(
    (PopulationStateListener) Spin.over(
        new PopulationStateListenerImpl()));
Object[] values = populationBean.createValues();
SpinExample.this.statusArea.setText("");
SpinExample.this.listModel.removeAllElements();
for (int i = 0; i < values.length; i++) {
    SpinExample.this.listModel.addElement(values[i]);
}
}

}

public interface PopulationBean {
    Object[] createValues();
    void addPopulationStateListener(PopulationStateListener populationStateListener);
}

private class PopulationBeanImpl implements PopulationBean {
    private PopulationStateListener populationStateListener;

    public Object[] createValues() {
        Object[] values = new Object[100];
        for (int i = 1; i <=100; i++) {
            values[i - 1] = "Value" + i;

            if ((i % 10) == 0) {
                this.populationStateListener.updateState(i);
            }
        }
        try {
```

```

        Thread.sleep(50);
    } catch (InterruptedException e) {
    }
}
return values;
}

public void addPopulationStateListener(PopulationStateListener populationStateListener) {
    this.populationStateListener = populationStateListener;
}
}

public interface PopulationStateListener {
    void updateState(int string);
}

private class PopulationStateListenerImpl implements PopulationStateListener {
    public void updateState(int progress) {
        SpinExample.this.statusArea.setText("Calculated " + progress);
    }
}
}

```

The population code has been moved into a **PopulationBeanImpl** that implements the **PopulationBean** interface. A listener interface of **PopulationStateListener** has also been created with an implementation that updates the **statusArea** label. The **LongRunningModelFillAction** first creates a proxied version of the **PopulationBean** using **Spin.off()**. The listener is then proxied with **Spin.over()** and added to the bean. The action then calls **createValues()** on the **PopulationBean**. This call blocks the EDT while being executed off the EDT by Spin using the technique similar to Foxtrot. This keeps the UI responsive. While calculating values in the **createValues()** method, the **updateState()** method is called on the listener. Since it has been proxied by Spin, these calls are placed on the EDT for execution.

Note: The notification of the listener is done through **invokeAndWait()** internally by Spin instead of **invokeLater()**, which is used in the original version of this example. This difference will alter the speed of the **createValues()** method slightly because each notification must be processed in the event queue before the next value will be generated.

When the `createdValues()` method returns, the EDT is unblocked and the action updates the `ListModel`.

Foxtrot Verses Spin

Both Foxtrot and Spin provide synchronous solutions to the Single Threading Rule problem. However, they have a number of differences between them. Spin requires the use of interfaces while Foxtrot has a `Worker.post()` structure. Spin can actually simulate Foxtrot with code like this:

```
Object[] values = ((Job)Spin.off(new Job())
{
    public Object run()
    {
        return populationBean.createValues();
    }
}).run();
```

In this example, the `Job` object is proxied by Spin. As a result, calls to the `run()` method are executed off the EDT. The `run()` method in turn calls the `PopulationBean`'s `createValues()` method off the EDT just as Foxtrot's `Job` would.

Notice that the code in the Spin example returns an `Object[]`. Spin provides more specific return types and exceptions than Foxtrot. Because Spin proxies the object to implement its solution, it can return the same object type and throw the same type of exceptions as the original object. Foxtrot's `Worker.post()` method returns a generic `Object` and throws a generic `Exception`. Here is the Foxtrot task example (which throws an `Exception`) rewritten using Spin.

```
private class LongRunningModelFillAction extends AbstractAction {
    public LongRunningModelFillAction() {
        super("Fill Model");
    }

    public void actionPerformed(ActionEvent e) {

        ...
    }
}
```

```

        try {
            values = populationBean.createValues();

            SpinExceptionExample.this.statusArea.setText("");
            SpinExceptionExample.this.listModel.removeAllElements();
            for (int i = 0; i < values.length; i++) {

                SpinExceptionExample.this.listModel.addElement(values[i]);
            }
        } catch (PopulationException e1) {
            SpinExceptionExample.this.statusArea.setText("Error Populating Model");
            SpinExceptionExample.this.listModel.removeAllElements();
        }
    }
}

public interface PopulationBean {
    Object[] createValues() throws PopulationException;
    void addPopulationStateListener(PopulationStateListener
populationStateListener);
}

private class PopulationBeanImpl implements PopulationBean {
    private PopulationStateListener populationStateListener;

    public Object[] createValues() throws PopulationException {
        Object[] values = new Object[100];
        for (int i = 1; i <=100; i++) {
            values[i - 1] = "Value" + i;

            if ((i % 10) == 0) {
                this.populationStateListener.updateState(i);
            } else if (i == 51) {
                //Simulate And Exception
                throw new PopulationException("Error Populating Model");
            }
        }
    }
}

```

```
        }
        try {
            Thread.sleep(50);
        } catch (InterruptedException e) {
        }
    }
    return values;
}
...
}
```

Finally, Foxtrot's **DefaultWorkerThread** queues nested tasks that are posted to it. Spin, on the other hand, allows nested tasks. Here is an example of a nested execution path in Spin.

1. A long-running method A is invoked by the UI.
2. Spin runs method A off the EDT while keeping the UI responsive.
3. Method A prompts the user for input.
4. Spin transparently executes this request on the EDT.
5. This input must execute another long-running method B before returning to method A.
6. Spin invokes method B in a second invocation.
7. Method B returns and unblocks the EDT input request.
8. The EDT input response is returned to method A.
9. Method A finishes, and the original EDT call to method A unblocks execution.

Detecting Threading Issues

Often in a multi-threaded application, you can have Single Threading Rule violations without knowing it. I found an easy way to detect such problems. The result of most Swing method calls is a change in the painted state of a component. This painting is managed in Swing through a class called the **RepaintManager**. This seemed like an ideal place to check for EDT threading violations. If a method has been called on a component from a non-EDT thread that causes it to paint, the chain of executed methods still ends up accessing this class. Consequently, I wrote a custom thread checking **RepaintManager**. The Spin project enhanced my version and included theirs in the distribution of Spin. Its code follows:

```
public class CheckingRepaintManager extends RepaintManager {

    public synchronized void addInvalidComponent(JComponent component) {
        checkEDTRule(component);

        super.addInvalidComponent(component);
    }

    public synchronized void addDirtyRegion(JComponent component,
        int x, int y, int w, int h) {
        checkEDTRule(component);

        super.addDirtyRegion(component, x, y, w, h);
    }

    protected void checkEDTRule(Component component) {
        if (violatesEDTRule(component)) {
            EDTRuleViolation violation = new EDTRuleViolation(component);

            StackTraceElement[] stackTrace = violation.getStackTrace();
            try {
                for (int e = stackTrace.length - 1; e >= 0; e--) {
                    if (is LIABLEToEDTRule(stackTrace[e])) {
                        StackTraceElement[] subStackTrace =
                            new StackTraceElement[stackTrace.length - e];
```

```
        System.arraycopy(stackTrace, e, subStackTrace, 0,
            subStackTrace.length);

        violation.setStackTrace(subStackTrace);
    }
}
} catch (Exception ex) {
    // keep stackTrace
}

    indicate(violation);
}
}

protected boolean violatesEDTRule(Component component) {
    return !SwingUtilities.isEventDispatchThread() &&
        component.isShowing();
}

protected boolean isLiableToEDTRule(StackTraceElement element) throws Exception {
    return Component.class.isAssignableFrom(Class.forName(element.getClassName()));
}

protected void indicate(EDTRuleViolation violation)
    throws EDTRuleViolation {
    throw violation;
}
}
```

The two methods called during repaints are **addInvalidComponent()** and **addDirtyRegion()**. The **CheckingRepaintManager** checks that execution is occurring on the EDT using the **SwingUtilities.isEventDispatchThread()** method. If the calling method is on the EDT, the method call continues on to the parent version of the method. Otherwise, an **EDTRuleViolation** exception is thrown. You can create a simple logging version by overriding the **indicate()** method:

```
public class CheckAndLogRepaintManager extends CheckingRepaintManager {
    protected void indicate(EDTRuleViolation violation) throws
                                EDTRuleViolation {
        violation.printStackTrace();
    }
}
```

The final step is to put the new **RepaintManager** to use. You must install it by calling the static `setCurrentManager()` method on the **RepaintManager** class:

```
RepaintManager.setCurrentManager(new CheckAndLogRepaintManager());
```

Here is an example of a badly written Swing action. It adds elements to a list with a thread, which is a violation.

```
private class WorkerAction extends AbstractAction {
    private DefaultListModel listModel;
    private int val = 0;

    public WorkerAction(DefaultListModel listModel) {
        super("Do Work");
        this.listModel = listModel;
    }

    public void actionPerformed(ActionEvent event) {
        new Thread(new Runnable() {
            public void run() {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                for (int i = 0; i < 100; i++) {
                    val++;
                    listModel.addElement(new Integer(val));
                }
            }
        }).start();
    }
}
```

```
    }  
    }  
  }).start();  
  }  
}
```

When this action is executed with the **CheckAndLogRepaintManager** installed, the following stack-trace is printed:

```
spin.over.EDTRuleViolation  
    at javax.swing.JComponent.repaint(JComponent.java:4334)  
    at java.awt.Component.repaint(Component.java:2474)  
    at javax.swing.plaf.basic.BasicListUI.redrawList(BasicListUI.java:1511)  
    at javax.swing.plaf.basic.BasicListUI.access$300(BasicListUI.java:34)  
    at javax.swing.plaf.basic.BasicListUI$ListDataHandler.intervalAdded(BasicListUI.java:1553)  
    at javax.swing.AbstractListModel.fireIntervalAdded(AbstractListModel.java:130)  
    at javax.swing.DefaultListModel.addElement(DefaultListModel.java:348)  
    at com.sourcebeat.desktopjava.ch5.checkingrepaint.BadThreadSwingExample$1.run(BadThreadSwingExample.java:30)  
    at java.lang.Thread.run(Thread.java:552)
```

You can now see exactly where the violation originated.

Summary

This chapter covered the ins and outs of threading in respect to Swing and detailed how Swing and the EDT thread interact. Understanding this interaction pattern makes it much easier to write correctly threaded Swing code. After the discussion of the EDT, you were shown a number of structured threading solutions to help you design your threading code. These solutions provide a number of options to match your coding style. Finally, you saw how a custom RepaintManager facilitates finding incorrectly written threading code.