CHAPTER

9

Threads and Swing

IN THIS CHAPTER

The Swing graphical toolkit brings a host of new components to the Java platform. There's a catch, though—Swing components are not designed for a multithreaded environment. In this chapter, I'll show how you how to safely interact with Swing components in a multithread-safe manner using `SwingUtilities.invokeAndWait()` and `SwingUtilities.invokeLater()`. I'll also show you some ways that animation can be achieved using Swing components and threads.

Why Isn't the Swing Toolkit Multithread-Safe?

After Swing components have been displayed on the screen, they should only be operated on by the event-handling thread. The event-handling thread (or just event thread) is started automatically by the Java VM when an application has a graphical interface. The event thread calls methods like `paint()` on `Component`, `actionPerformed()` on `ActionListener`, and all of the other event-handling methods.

Most of the time, modifications to Swing components are done in the event-handling methods. Because the event thread calls these methods, it is perfectly safe to directly change components in event-handling code. `SimpleEvent` (see Listing 9.1) shows safe Swing code.

LISTING 9.1 SimpleEvent.java—Safe Swing Code That Uses the Event Thread

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

public class SimpleEvent extends Object {
    private static void print(String msg) {
        String name = Thread.currentThread().getName();
        System.out.println(name + “: “ + msg);
    }

    public static void main(String[] args) {
         final JLabel label = new JLabel(“————”);
        JButton button = new JButton(“Click Here”);

        JPanel panel = new JPanel(new FlowLayout());
        panel.add(button);
        panel.add(label);

        button.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    print(“in actionPerformed()”);
                    label.setText(“CLICKED!”);
23:                 }
24:             });
25:
26:         JFrame f = new JFrame("SimpleEvent");
27:         f.setContentPane(panel);
28:         f.setSize(300, 100);
29:         f.setVisible(true);
30:     }
31: }
```

In `SimpleEvent`, two threads interact with the Swing components. First, the `main` thread creates the components (lines 12–15), adds them to `panel` (lines 16–17), and creates and configures a `JFrame` (lines 26–29). After `setVisible()` is invoked by `main` (line 29), it is no longer safe for any thread other than the event thread to make changes to the components.

When the button is clicked, the event thread invokes the `actionPerformed()` method (lines 20–23). In there, it prints a message to show which thread is running the code (line 21) and changes the text for `label` (line 22). This code is perfectly safe because it is the event thread that ends up calling `setText()`.

When `SimpleEvent` is run, the frame appears and the following output is printed to the console when the button is clicked:

```
AWT-EventQueue-0: in actionPerformed()
```

The thread named `AWT-EventQueue-0` is the event thread. This is the thread that can safely make changes through methods like `setText()`.

One of the goals for the developers of Swing was to make the toolkit as fast as possible. If the components had to be multithread-safe, there would need to be a lot of `synchronized` statements and methods. The extra overhead incurred acquiring and releasing locks all the time would have slowed the performance of the components. The developers made the choice for speed over safety. As a result, you need to be very careful when making modifications to Swing components that are initiated outside the event thread.

Using SwingUtilities.invokeAndWait()

The developers of the Swing toolkit realized that there would be times when an external thread would need to make changes to Swing components. They created a mechanism that puts a reference to a chunk of code on the event queue. When the event thread gets to this code block, it executes the code. This way, the GUI can be changed inside this block of code by the event thread.

The `SwingUtilities` class has a `static invokeAndWait()` method available to use to put references to blocks of code onto the event queue:

```
public static void invokeAndWait(Runnable target)
        throws InterruptedException,
               InvocationTargetException
```

The parameter `target` is a reference to an instance of `Runnable`. In this case, the `Runnable` will not be passed to the constructor of `Thread`. The `Runnable` interface is simply being used as a means to identify the entry point for the event thread. Just as a newly spawned thread will invoke `run()`, the event thread will invoke `run()` when it has processed all the other events pending in the queue.

An `InterruptedException` is thrown if the thread that called `invokeAndWait()` is interrupted before the block of code referred to by `target` completes. An `InvocationTargetException` (a class in the `java.lang.reflect` package) is thrown if an uncaught exception is thrown by the code inside `run()`.

NOTE

A new thread is *not* created when `Runnable` is used with `SwingUtilities.invokeAndWait()`. The event thread will end up calling the `run()` method of the `Runnable` when its turn comes up on the event queue.

Suppose a `JLabel` component has been rendered on screen with some text:

```
label = new JLabel( // ...
```

Now, if a thread *other than the event thread* needs to call `setText()` on `label` to change it, the following should be done. First, create an instance of `Runnable` to do the work:

```
Runnable setTextRun = new Runnable() {
        public void run() {
            label.setText( // ...
        }
    };
```

Then pass the `Runnable` instance referred to by `setTextRun` to `invokeAndWait()`:

```
try {
    SwingUtilities.invokeAndWait(setTextRun);
} catch ( InterruptedException ix ) {
    ix.printStackTrace();
} catch ( InvocationTargetException x ) {
    x.printStackTrace();
}
```

The try/catch block is used to catch the two types of exception that might be thrown while waiting for the code inside the `run()` method of `setTextRun` to complete.

`InvokeAndWaitDemo` (see Listing 9.2) is a complete example that demonstrates the use of `SwingUtilities.invokeAndWait()`.

LISTING 9.2 InvokeAndWaitDemo.java—Using SwingUtilities.invokeAndWait()

```
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;
import javax.swing.*;

public class InvokeAndWaitDemo extends Object {
    private static void print(String msg) {
        String name = Thread.currentThread().getName();
        System.out.println(name + “: “ + msg);
    }

    public static void main(String[] args) {
        final JLabel label = new JLabel(“————”);

        JPanel panel = new JPanel(new FlowLayout());
        panel.add(label);

        JFrame f = new JFrame(“InvokeAndWaitDemo”);
        f.setContentPane(panel);
        f.setSize(300, 100);
        f.setVisible(true);

        try {
            print(“sleeping for 3 seconds”);
            Thread.sleep(3000);

            print(“creating code block for event thread”);
            Runnable setTextRun = new Runnable() {
                    public void run() {
                        print(“about to do setText()”);
                        label.setText(“New text!”);
                    }
                };

```

continues

LISTING 9.2 Continued

```
35:             print(“about to invokeAndWait()”);
36:             SwingUtilities.invokeAndWait(setTextRun);
37:             print(“back from invokeAndWait()”);
38:         } catch ( InterruptedException ix ) {
39:             print(“interrupted while waiting on invokeAndWait()”);
40:         } catch ( InvocationTargetException x ) {
41:             print(“exception thrown from run()”);
42:         }
43:     }
44: }
```

Note that the `java.lang.reflect` package is imported (line 3) solely for `InvocationTargetException`. The `main` thread creates the GUI (lines 13–20) and invokes `setVisible()` on the `JFrame` (line 21). *From that point on, only the event thread should make changes to the GUI.*

After sleeping for `3` seconds (line 25), the `main` thread wants to change the text displayed in `label`. To safely do this, the `main` thread must pass this work off to the event-handling thread. The `main` thread creates a bundle of code in `setTextRun`, which is an instance of `Runnable` (lines 28–33). Inside the `run()` method, the `setText()` method is invoked on `label` (line 31). Ultimately, the event thread will end up invoking the `setText()` method inside this `run()` method.

The `main` thread then calls `SwingUtilities.invokeAndWait()` passing in `setTextRun` (line 36). Inside `invokeAndWait()`, the `setTextRun` reference is put onto the event queue. When all the events that were ahead of it in the queue have been processed, the event thread invokes the `run()` method of `setTextRun`. When the event thread returns from `run()`, it notifies the `main` thread that it has completed the work. The event thread then goes back to reading events from the event queue. At the same time, the `main` thread returns from `invokeAndWait()`, indicating that the code block inside `setTextRun` has been run by the event thread.

Listing 9.3 shows the output produced when `InvokeAndWaitDemo` is run. In addition, a GUI frame appears, but that doesn’t show anything other than the fact that the label changes when `setText()` is invoked.

LISTING 9.3 Output from InvokeAndWaitDemo

```
main: sleeping for 3 seconds
main: creating code block for event thread
main: about to invokeAndWait()
AWT-EventQueue-0: about to do setText()
main: back from invokeAndWait()
```

The `main` thread announces that it is about to call `invokeAndWait()` (line 3). Next, the event thread (`AWT-EventQueue-0`) reports that it is indeed the thread that is invoking `setText()` (line 4). The `main` thread then reports that it is done blocking and has returned from `invokeAndWait()` (line 5).

CAUTION

Do not call `SwingUtilities.invokeAndWait()` from the event thread. Doing so causes an instance of `Error` to be thrown. Even if this call were allowed, it would put the event thread into a deadlocked state. The event thread does not need the services of `invokeAndWait()` because it can make the changes directly.

Using SwingUtilities.invokeLater()

The `SwingUtilities` class has another `static` method available to use to put references to blocks of code onto the event queue:

```
public static void invokeLater(Runnable target)
```

The `SwingUtilities.invokeLater()` method works like `SwingUtilities.invokeAndWait()` except for the fact that it puts the request on the event queue and *returns right away*. The `invokeLater()` method does not wait for the block of code inside the `Runnable` referred to by `target` to execute. This allows the thread that posted the request to move on to other activities.

NOTE

Just as with `invokeAndWait()`, a new thread is *not* created when `Runnable` is used with `SwingUtilities.invokeLater()`.

This example is just like the one used for `invokeAndWait()`, but instead shows the changes necessary to use `invokeLater()`. Suppose a `JLabel` component has been rendered on screen with some text:

```
label = new JLabel( // ...
```

If a thread *other than the event thread* needs to call `setText()` on `label` to change it, you should do the following. First, create an instance of `Runnable` to do the work:

```
Runnable setTextRun = new Runnable() {
        public void run() {
            try {
                label.setText( // ...
            } catch ( Exception x ) {
                x.printStackTrace();
            }
        }
    };
```

Be sure to catch all exceptions inside `run()` because unlike `invokeAndWait()`, `invokeLater()` does not have an automatic mechanism to propagate the exception back to the thread that called `invokeLater()`. Instead of simply printing a stack trace, you could have the event thread store the exception and notify another thread that an exception occurred.

Next, pass the `Runnable` instance referred to by `setTextRun` to `invokeLater()`:

```
SwingUtilities.invokeLater(setTextRun);
```

This call returns right away and does not throw any exceptions. When the event thread has processed all of the pending events, it invokes the `run()` method of `setTextRun`.

`InvokeLaterDemo` (see Listing 9.4) is a complete example (based on `InvokeAndWaitDemo`) that demonstrates the use of `SwingUtilities.invokeLater()`.

LISTING 9.4 InvokeLaterDemo.java—Using SwingUtilities.invokeLater()

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

public class InvokeLaterDemo extends Object {
    private static void print(String msg) {
        String name = Thread.currentThread().getName();
        System.out.println(name + ": " + msg);
    }

    public static void main(String[] args) {
        final JLabel label = new JLabel("————");

        JPanel panel = new JPanel(new FlowLayout());
        panel.add(label);

        JFrame f = new JFrame("InvokeLaterDemo");
        f.setContentPane(panel);
        f.setSize(300, 100);
        f.setVisible(true);

22:         try {
23:             print(“sleeping for 3 seconds”);
24:             Thread.sleep(3000);
25:         } catch ( InterruptedException ix ) {
26:             print(“interrupted while sleeping”);
27:         }
28:
29:         print(“creating code block for event thread”);
30:         Runnable setTextRun = new Runnable() {
31:                 public void run() {
32:                     try {
33:                         Thread.sleep(100); // for emphasis
34:                         print(“about to do setText()”);
35:                         label.setText(“New text!”);
36:                     } catch ( Exception x ) {
37:                         x.printStackTrace();
38:                     }
39:                 }
40:             };
41:
42:         print(“about to invokeLater()”);
43:         SwingUtilities.invokeLater(setTextRun);
44:         print(“back from invokeLater()”);
45:     }
46: }
```

The `main` thread creates the GUI (lines 12–19) and invokes `setVisible()` on the `JFrame` (line 20). *From that point on, only the event thread should make changes to the GUI.*

After sleeping for `3` seconds (line 24), the `main` thread wants to change the text displayed in `label`. To safely do this, the `main` thread creates a bundle of code in `setTextRun` (lines 30–40). Inside the `run()` method, a try/catch block is used to capture any exceptions that might be thrown so that `run()` itself does not end up throwing any exceptions (lines 32–38). A very short sleep of 0.1 seconds (line 33) is used to momentarily slow the event thread to clearly show that the `invokeLater()` call returns right away. In real-world code there would not be any need for this sleep. Eventually, the event thread invokes the `setText()` method on `label` (line 35).

After setting up this code block, the `main` thread calls `SwingUtilities.invokeLater()`, passing in `setTextRun` (line 43). Inside `invokeLater()`, the `setTextRun` reference is put onto the event queue and then the `main` thread *returns right away*. When all of the events that were ahead of it in the queue have been processed, the event thread invokes the `run()` method of `setTextRun`.

Listing 9.5 shows the output produced when `InvokeLaterDemo` is run. Your output should match. In addition, a frame is drawn on the screen, but it doesn't show anything other than the fact that the label does indeed change.

LISTING 9.5 Output from InvokeLaterDemo

```
1: main: sleeping for 3 seconds
2: main: creating code block for event thread
3: main: about to invokeLater()
4: main: back from invokeLater()
5: AWT-EventQueue-0: about to do setText()
```

The `main` thread calls (line 3) and returns from (line 4) `invokeLater()` before the event thread gets a chance to invoke `setText()` (line 5). This is the exact asynchronous behavior that was desired.

NOTE

Unlike `SwingUtilities.invokeAndWait()`, the event thread *is* permitted to call `SwingUtilities.invokeLater()`. However, there isn't any value to doing so because the event thread can change the components directly.

Using SwingUtilities.isEventDispatchThread()

If you have code that must (or must not) be called by the event thread, you can use the `SwingUtilities.isEventDispatchThread()` method:

```
public static boolean isEventDispatchThread()
```

This `static` method returns `true` if the thread that invokes it is the event thread, and returns `false` if it is not.

If it is critical that only the event thread calls a particular method, you might want to put some code like this at the beginning of the method:

```
if ( SwingUtilities.isEventDispatchThread() == false ) {
    throw new RuntimeException(
        “only the event thread should invoke this method”);
}
```

This way if any thread other than the event thread calls the method, a `RuntimeException` is thrown. This step can help safeguard against dangerous code that works *most* of the time when called by a thread other than the event thread.

A downside to this method is that it takes a little bit of time to execute. If you have some code where performance is critical, you might want to skip this check.

When invokeAndWait() and invokeLater() Are Not Needed

It is not always necessary to use `invokeAndWait()` and `invokeLater()` to interact with Swing components. Any thread can safely interact with the components before they have been added to a visible container. You have seen this already in the examples: The `main` thread constructs the GUI and then invokes `setVisible()`. After the components have been drawn to the screen, only the event thread should make further changes to their appearance.

There are a couple of exceptions to this restriction. The adding and removing of event listeners can safely be done by any thread at any time. Also, any thread can invoke the `repaint()` method. The `repaint()` method has always worked asynchronously to put a repaint request onto the event queue. And finally, any method that *explicitly* indicates that it does not have to be called by the event thread is safe. The API documentation for the `setText()` method of `JTextComponent` explicitly states that `setText()` can be safely called by any thread. The `setText()` method is inherited by `JTextField` (a subclass of `JTextComponent`), so any thread can safely invoke `setText()` on a `JTextField` component at any time.

TIP

If you aren't sure whether a particular method on a Swing component can be invoked by any thread, use the `invokeAndWait()` or `invokeLater()` mechanism to be safe.

The Need for Worker Threads in a GUI Setting

The event thread plays a critical role in an application with a graphical interface. Code that will be executed by the event-handling thread should be relatively brief and nonblocking. If the event-handling thread is blocked in a section of code for a while, no other events can be processed!

This is especially important in a client/server application (even more so in an n-tier application). Imagine a situation where the client is a graphical application with a Search button. When this button is clicked, a request is made over the network to the server for the results. The server produces the results and sends this information back down to the client. The client then displays this result information on the GUI. To be safe, the event thread needs to be the thread that gathers the information from the GUI for the search. The event thread also needs to be the thread that displays the results. But does the event thread have to send the request over the network? No, it does not, and should not.

The `BalanceLookupCantCancel` class (see Listing 9.6) shows what happens when the event thread is used to fulfill a request that takes a long time. This simple graphical client simulates a call over the network by sleeping for five seconds before returning the account balance.

LISTING 9.6 BalanceLookupCantCancel.java—Overusing the Event Thread

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

public class BalanceLookupCantCancel extends JPanel {
    private JTextField acctTF;
    private JTextField pinTF;
    private JButton searchB;
    private JButton cancelB;
    private JLabel balanceL;

    public BalanceLookupCantCancel() {
        buildGUI();
        hookupEvents();
    }

    private void buildGUI() {
        JLabel acctL = new JLabel(“Account Number:”);
        JLabel pinL = new JLabel(“PIN:”);
        acctTF = new JTextField(12);
        pinTF = new JTextField(4);

        JPanel dataEntryP = new JPanel();
        dataEntryP.setLayout(new FlowLayout(FlowLayout.CENTER));
        dataEntryP.add(acctL);
        dataEntryP.add(acctTF);
        dataEntryP.add(pinL);
        dataEntryP.add(pinTF);

        searchB = new JButton(“Search”);
        cancelB = new JButton(“Cancel Search”);
        cancelB.setEnabled(false);

        JPanel innerButtonP = new JPanel();
        innerButtonP.setLayout(new GridLayout(1, -1, 5, 5));
        innerButtonP.add(searchB);
        innerButtonP.add(cancelB);

        JPanel buttonP = new JPanel();
        buttonP.setLayout(new FlowLayout(FlowLayout.CENTER));
        buttonP.add(innerButtonP);

        JLabel balancePrefixL = new JLabel("Account Balance:");
        balanceL = new JLabel("BALANCE UNKNOWN");

        JPanel balanceP = new JPanel();
        balanceP.setLayout(new FlowLayout(FlowLayout.CENTER));
        balanceP.add(balancePrefixL);
        balanceP.add(balanceL);

        JPanel northP = new JPanel();
        northP.setLayout(new GridLayout(-1, 1, 5, 5));
        northP.add(dataEntryP);
        northP.add(buttonP);
        northP.add(balanceP);

        setLayout(new BorderLayout());
        add(northP, BorderLayout.NORTH);
    }

    private void hookupEvents() {
        searchB.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    search();
                }
            });

        cancelB.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    cancelSearch();
                }
            });
    }

    private void search() {
        // better be called by event thread!
        searchB.setEnabled(false);
        cancelB.setEnabled(true);
```

continues

LISTING 9.6 Continued

```
        balanceL.setText(“SEARCHING ...”);

        // get a snapshot of this info in case it changes
        String acct = acctTF.getText();
        String pin = pinTF.getText();

        String bal = lookupBalance(acct, pin);
        setBalance(bal);
    }

    private String lookupBalance(String acct, String pin) {
        try {
            // Simulate a lengthy search that takes 5 seconds
            // to communicate over the network.
            Thread.sleep(5000);

            // result “retrieved”, return it
            return “1,234.56”;
        } catch ( InterruptedException x ) {
            return “SEARCH CANCELLED”;
        }
    }

    private void setBalance(String newBalance) {
        // better be called by event thread!
        balanceL.setText(newBalance);
        cancelB.setEnabled(false);
        searchB.setEnabled(true);
    }

    private void cancelSearch() {
        System.out.println(“in cancelSearch()”);
        // Here’s where the code to cancel would go if this
        // could ever be called!
    }

    public static void main(String[] args) {
        BalanceLookupCantCancel bl =
                new BalanceLookupCantCancel();

        JFrame f = new JFrame(“Balance Lookup - Can’t Cancel”);
        f.addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
                    System.exit(0);
                }
```

```
124:             });
125:
126:         f.setContentPane(bl);
127:         f.setSize(400, 150);
128:         f.setVisible(true);
129:     }
130: }
```

Most of `BalanceLookupCantCancel` (lines 1–73, 115–129) is dedicated to constructing the GUI. In `hookupEvents()` (lines 61–73), an event handler for each button is added. When the search button `searchB` is clicked, the `search()` method is called (lines 63–65). When the cancel button `cancelB` is clicked, `cancelSearch()` is called (lines 69–71).

Inside `search()` (lines 75–87), the Search button is disabled, the Cancel Search button is enabled, and the balance label is set to `SEARCHING ...` while the search is in progress (lines 77–78). The event thread is used to gather the account number and PIN number from the fields (lines 82–83). These strings are passed into `lookupBalance()`, and the balance found is returned and shown on the screen (lines 85–86).

The `lookupBalance()` method (lines 89–100) is used to simulate a lookup over a network connection. It sleeps for five seconds to simulate the delay for lookup and then returns `1,234.56` for every account. If the thread that called `lookupBalance()` is interrupted while the lookup is in progress (sleeping), it returns the `SEARCH CANCELLED` string instead of the balance. This is just a simulation; of course, a real system would do something more useful.

The `setBalance()` method (lines 102–107) is used to update the balance, disable the Cancel Search button, and enable the Search button again. The `cancelSearch()` method (lines 109–113) would normally be used to stop the search process, but in this example, it never gets called.

When the event thread calls `search()`, it blocks until the balance is retrieved and set. Keeping the event thread tied up for that long is a bad idea. And in this example, it prevents the Cancel Search button from being enabled.

Figure 9.1 shows how the application looks when it is first started. Notice that the Cancel Search button is disabled and that the balance label indicates that the balance is unknown.

After the user enters an account number and a PIN and clicks the Search button, the application looks like Figure 9.2. The window continues to look like that for about `5` seconds while the across-the-network lookup is simulated. Notice the following points:

- The `SEARCHING ...` message was not displayed in the balance label.
- The Cancel Search button was never enabled.
- The Search button stayed pressed in the whole time.

For the whole time that the lookup was going on, the GUI was unresponsive—the window couldn't even be closed. In particular, the Cancel Search button was never enabled. The event thread was tied up doing the long-running lookup and could not respond to user events. Obviously, this is not a good design.

Figure 9.3 shows what the application window looks like after the `5` seconds have elapsed. Here everything is as expected. The Search button is enabled, the Cancel Search button is disabled, and the balance label shows `1,234.56`.

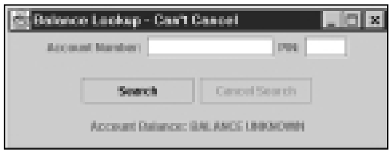

Balance Lookup - Can't Cancel
Account Number:
PIN:
Search
Cancel Search
Account Balance: BALANCE UNKNOWN

FIGURE 9.1
BalanceLookupCantCancel just after startup.

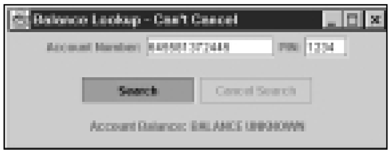

Balance Lookup - Can't Cancel
Account Number:
PIN: 1234
Search
Cancel Search
Account Balance: BALANCE UNKNOWN

FIGURE 9.2
BalanceLookupCantCancel after the Search button is clicked.

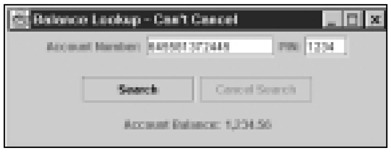


FIGURE 9.3
BalanceLookupCantCancel when the lookup finally completes.

Using a Worker Thread to Relieve the Event Thread

In `BalanceLookupCantCancel`, it became apparent that tying up the event thread to do an extensive operation was a bad idea. This was a problem especially because there was no way to signal that the search should be canceled. Another thread is needed to do the lookup so that the event thread can get back to the business of handling events.

BalanceLookup (see Listing 9.7) uses a worker thread to do the lengthy lookup and frees the event thread from this delay. This technique makes it possible to use the Cancel Search button to stop a search.

LISTING 9.7 BalanceLookup.java—Using a Worker Thread to Relieve the Event Thread

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

public class BalanceLookup extends JPanel {
    private JTextField acctTF;
    private JTextField pinTF;
    private JButton searchB;
    private JButton cancelB;
    private JLabel balanceL;

    private volatile Thread lookupThread;

    public BalanceLookup() {
        buildGUI();
        hookupEvents();
    }

    private void buildGUI() {
        JLabel acctL = new JLabel("Account Number:");
        JLabel pinL = new JLabel("PIN:");
        acctTF = new JTextField(12);
        pinTF = new JTextField(4);

        JPanel dataEntryP = new JPanel();
        dataEntryP.setLayout(new FlowLayout(FlowLayout.CENTER));
        dataEntryP.add(acctL);
        dataEntryP.add(acctTF);
        dataEntryP.add(pinL);
        dataEntryP.add(pinTF);

        searchB = new JButton("Search");
        cancelB = new JButton("Cancel Search");
        cancelB.setEnabled(false);

        JPanel innerButtonP = new JPanel();
```

continues

LISTING 9.7 Continued

```
        innerButtonP.setLayout(new GridLayout(1, -1, 5, 5));
        innerButtonP.add(searchB);
        innerButtonP.add(cancelB);

        JPanel buttonP = new JPanel();
        buttonP.setLayout(new FlowLayout(FlowLayout.CENTER));
        buttonP.add(innerButtonP);

        JLabel balancePrefixL = new JLabel(“Account Balance:”);
        balanceL = new JLabel(“BALANCE UNKNOWN”);

        JPanel balanceP = new JPanel();
        balanceP.setLayout(new FlowLayout(FlowLayout.CENTER));
        balanceP.add(balancePrefixL);
        balanceP.add(balanceL);

        JPanel northP = new JPanel();
        northP.setLayout(new GridLayout(-1, 1, 5, 5));
        northP.add(dataEntryP);
        northP.add(buttonP);
        northP.add(balanceP);

        setLayout(new BorderLayout());
        add(northP, BorderLayout.NORTH);
    }

    private void hookupEvents() {
        searchB.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    search();
                }
            });

        cancelB.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    cancelSearch();
                }
            });
    }

    private void search() {
        // better be called by event thread!
        ensureEventThread();

```

```
            searchB.setEnabled(false);
            cancelB.setEnabled(true);
            balanceL.setText("SEARCHING ...");

            // get a snapshot of this info in case it changes
            String acct = acctTF.getText();
            String pin = pinTF.getText();

            lookupAsync(acct, pin);
        }

        private void lookupAsync(String acct, String pin) {
            // Called by event thread, but can be safely
            // called by any thread.
            final String acctNum = acct;
            final String pinNum = pin;

            Runnable lookupRun = new Runnable() {
                    public void run() {
                        String bal = lookupBalance(acctNum, pinNum);
                        setBalanceSafely(bal);
                    }
                };

            lookupThread = new Thread(lookupRun, "lookupThread");
            lookupThread.start();
        }

        private String lookupBalance(String acct, String pin) {
            // Called by lookupThread, but can be safely
            // called by any thread.
            try {
                // Simulate a lengthy search that takes 5 seconds
                // to communicate over the network.
                Thread.sleep(5000);

                // result "retrieved", return it
                return "1,234.56";
            } catch ( InterruptedException x ) {
                return "SEARCH CANCELLED";
            }
        }

```

continues

LISTING 9.7 Continued

```
    private void setBalanceSafely(String newBal) {
        // Called by lookupThread, but can be safely
        // called by any thread.
        final String newBalance = newBal;

        Runnable r = new Runnable() {
                public void run() {
                    try {
                        setBalance(newBalance);
                    } catch ( Exception x ) {
                        x.printStackTrace();
                    }
                }
            };

        SwingUtilities.invokeLater(r);
    }

    private void setBalance(String newBalance) {
        // better be called by event thread!
        ensureEventThread();

        balanceL.setText(newBalance);
        cancelB.setEnabled(false);
        searchB.setEnabled(true);
    }

    private void cancelSearch() {
        // better be called by event thread!
        ensureEventThread();

        cancelB.setEnabled(false); //prevent additional requests

        if ( lookupThread != null ) {
            lookupThread.interrupt();
        }
    }

    private void ensureEventThread() {
        // throws an exception if not invoked by the
        // event thread.
        if ( SwingUtilities.isEventDispatchThread() ) {
            return;
```

```
167:         }
168:
169:         throw new RuntimeException("only the event " +
170:             "thread should invoke this method");
171:     }
172:
173:     public static void main(String[] args) {
174:         BalanceLookup bl = new BalanceLookup();
175:
176:         JFrame f = new JFrame("Balance Lookup");
177:         f.addWindowListener(new WindowAdapter() {
178:                 public void windowClosing(WindowEvent e) {
179:                     System.exit(0);
180:                 }
181:             });
182:
183:         f.setContentPane(bl);
184:         f.setSize(400, 150);
185:         f.setVisible(true);
186:     }
187: }
```

The code for `BalanceLookup` is based on `BalanceLookupCantCancel` but includes a few key changes to support a worker thread. Now, when the Search button is clicked and the `search()` method is called, `lookupAsync()` is invoked instead of looking up the balance directly.

The event thread invokes `lookupAsync()` (lines 92–107), passing in the account number and PIN strings. A new `Runnable` is created (lines 98–103). Inside the `run()` method, the slow `lookupBalance()` method is called. When `lookupBalance()` finally returns the balance, it is passed to the `setBalanceSafely()` method. A new `Thread` named `lookupThread` is constructed and started (lines 105–106). The event thread is now free to handle other events and `lookupThread` takes care of searching for the account information.

This time, the `lookupBalance()` method (lines 109–122) gets called by `lookupThread` instead of the event thread. `lookupThread` proceeds to sleep for `5` seconds to simulate the slow lookup on the server. If `lookupThread` is not interrupted while sleeping, it returns `1,234.56` for the balance (line 118). If it was interrupted, it returns `SEARCH CANCELLED` (line 120).

The `String` returned from `lookupBalance()` is taken by the `lookupThread` and passed to `setBalanceSafely()` (lines 124–140). Inside `setBalanceSafely()`, a `Runnable` is created that calls `setBalance()` inside its `run()` method (lines 129–137). This `Runnable` is passed to `SwingUtilities.invokeLater()` so that the event thread is the one that ultimately calls the `setBalance()` method.

Inside `setBalance()` (lines 142–149), a check is done by calling `ensureEventThread()` to be sure that it is indeed the event thread that has called the method. If it is, the balance label is updated with the information, the Cancel Search button is disabled again, and the Search button is enabled again.

The `cancelSearch()` method (lines 151–160) is called by the event thread when the Cancel Search button is clicked. Inside, it disables the Cancel Search button and interrupts `lookupThread`. This causes `lookupThread` to throw an `InterruptedException` and return the `SEARCH CANCELLED` message.

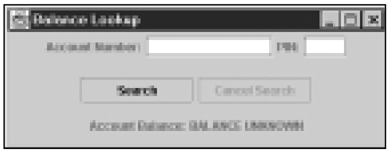
The `ensureEventThread()` method (lines 162–171) checks to see if the current thread is the event thread by using the `SwingUtilities.isEventDispatchThread()` method. If it is not, a `RuntimeException` is thrown. Several methods in `BalanceLookup` use `ensureEventThread()` to make sure that only the event thread is allowed to proceed.

Figure 9.4 shows how `BalanceLookup` looks just after startup. Notice that the Cancel Search button is disabled and that the balance label is `BALANCE UNKNOWN`.

After an account number and PIN are entered and the Search button is clicked, the application window looks like Figure 9.5. Notice that the Search button is disabled, the Cancel Search button is enabled, and the balance label is `SEARCHING ...`. It remains like this for about `5` seconds while the lookup is simulated.

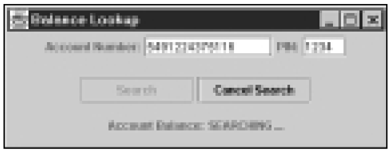
When the search finally completes, the application looks like Figure 9.6. Notice that the balance label is `1,234.56` (the fake balance), the Search button is enabled again, and the Cancel Search button is disabled again.

If you click on the Cancel Search button during the `5` seconds while the search is in progress, the window looks like Figure 9.7. Notice that that the balance label is `SEARCH CANCELLED`, indicating that the search did not get a chance to complete. As before, the Search button is enabled, and the Cancel Search button is disabled.



Account Balance: BALANCE UNKNOWN

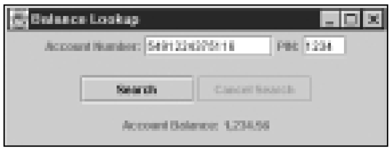
FIGURE 9.4
BalanceLookup just after startup.



Account Number: 5491224375118 PIN: 1234

Account Balance: SEARCHING ...

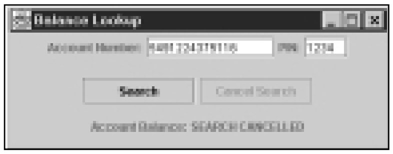
FIGURE 9.5
BalanceLookup after the Search button is clicked.



Account Number: 5491224375118 PIN: 1234

Account Balance: 1,234.56

FIGURE 9.6
BalanceLookup after the search has completed.



Account Number: 5491224319118 PIN: 1234

Account Balance: SEARCH CANCELLED

FIGURE9.7
BalanceLookup after the Cancel Search button is clicked during a search.

TIP

Rather than spawning a new thread every time the Search button is clicked, a better design would be to have a thread up and running and waiting to do the work. The event thread would gather the information, pass it to the waiting worker thread using synchronization, and signal the worker through the wait-notify mechanism that a new request was pending. When the worker thread had fulfilled the request, it would update the GUI through the `invokeLater()` mechanism. The worker would then go back to waiting for another notification. To simplify the synchronization and notification of the handoff, an `ObjectFIFO` with a capacity of `1` could be used (see Chapter 18, "First-In-First-Out (FIFO) Queue"). Also look at the thread-pooling techniques in Chapter 13, "Thread Pooling," for an example of how to do this type of handoff from one thread to another through a First-In-First-Out queue.

Scrolling Text in a Custom Component

`ScrollText` (see Listing 9.8) is a custom `JComponent` that takes the text passed to its constructor and scrolls it from left to right across the face of the component. At the time of construction, an off-screen image is prepared with the specified text and an internal thread is started to scroll this image. `ScrollText` is a self-running object and uses some of the techniques from Chapter 11, “Self-Running Objects,” to manage its internal thread.

LISTING 9.8 ScrollText.java—Scroll Text Across the Face of the Component

```
import java.awt.*;
import java.awt.image.*;
import java.awt.font.*;
import java.awt.geom.*;
import javax.swing.*;

public class ScrollText extends JComponent {
    private BufferedImage image;
    private Dimension imageSize;
    private volatile int currOffset;

    private Thread internalThread;
    private volatile boolean noStopRequested;

    public ScrollText(String text) {
        currOffset = 0;
        buildImage(text);

        setMinimumSize(imageSize);
        setPreferredSize(imageSize);
        setMaximumSize(imageSize);
        setSize(imageSize);

        noStopRequested = true;
        Runnable r = new Runnable() {
                public void run() {
                    try {
                        runWork();
                    } catch ( Exception x ) {
                        x.printStackTrace();
                    }
                }
            };

        internalThread = new Thread(r, “ScrollText”);
        internalThread.start();
    }

    private void buildImage(String text) {
        // Request that the drawing be done with anti-aliasing
        // turned on and the quality high.
        RenderingHints renderHints = new RenderingHints(
            RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON);

        renderHints.put(
            RenderingHints.KEY_RENDERING,
            RenderingHints.VALUE_RENDER_QUALITY);

        // Create a scratch image for use in determining
        // the text dimensions.
        BufferedImage scratchImage = new BufferedImage(
                1, 1, BufferedImage.TYPE_INT_RGB);

        Graphics2D scratchG2 = scratchImage.createGraphics();
        scratchG2.setRenderingHints(renderHints);

        Font font =
            new Font(“Serif”, Font.BOLD | Font.ITALIC, 24);

        FontRenderContext frc = scratchG2.getFontRenderContext();
        TextLayout tl = new TextLayout(text, font, frc);
        Rectangle2D textBounds = tl.getBounds();
        int textWidth = (int) Math.ceil(textBounds.getWidth());
        int textHeight = (int) Math.ceil(textBounds.getHeight());

        int horizontalPad = 10;
        int verticalPad = 6;

        imageSize = new Dimension(
                textWidth + horizontalPad,
                textHeight + verticalPad
            );

        // Create the properly-sized image
        image = new BufferedImage(
```

continues

Listing 9.8 Continued

```
                imageSize.width,
                imageSize.height,
                BufferedImage.TYPE_INT_RGB);

        Graphics2D g2 = image.createGraphics();
        g2.setRenderingHints(renderHints);

        int baselineOffset =
            ( verticalPad / 2 ) - ( (int) textBounds.getY());

        g2.setColor(Color.white);
        g2.fillRect(0, 0, imageSize.width, imageSize.height);

        g2.setColor(Color.blue);
        tl.draw(g2, 0, baselineOffset);

        // Free-up resources right away, but keep “image” for
        // animation.
        scratchG2.dispose();
        scratchImage.flush();
        g2.dispose();
    }

    public void paint(Graphics g) {
        // Make sure to clip the edges, regardless of curr size
        g.setClip(0, 0, imageSize.width, imageSize.height);

        int localOffset = currOffset; // in case it changes
        g.drawImage(image, -localOffset, 0, this);
        g.drawImage(
            image, imageSize.width - localOffset, 0, this);

        // draw outline
        g.setColor(Color.black);
        g.drawRect(
            0, 0, imageSize.width - 1, imageSize.height - 1);
    }

    private void runWork() {
        while ( noStopRequested ) {
            try {
                Thread.sleep(100);  // 10 frames per second

                // adjust the scroll position
```

```
121:                 currOffset =
122:                     ( currOffset + 1 ) % imageSize.width;
123:
124:                 // signal the event thread to call paint()
125:                 repaint();
126:             } catch ( InterruptedException x ) {
127:                 Thread.currentThread().interrupt();
128:             }
129:         }
130:     }
131:
132:     public void stopRequest() {
133:         noStopRequested = false;
134:         internalThread.interrupt();
135:     }
136:
137:     public boolean isAlive() {
138:         return internalThread.isAlive();
139:     }
140:
141:     public static void main(String[] args) {
142:         ScrollText st =
143:             new ScrollText("Java can do animation!");
144:
145:         JPanel p = new JPanel(new FlowLayout());
146:         p.add(st);
147:
148:         JFrame f = new JFrame("ScrollText Demo");
149:         f.setContentPane(p);
150:         f.setSize(400, 100);
151:         f.setVisible(true);
152:     }
153: }
```

In `main()` (lines 141–152), a new `ScrollText` instance is constructed (lines 142–143) and put into a `JPanel` with a `FlowLayout` to let the instance of `ScrollText` take on its preferred size (lines 145–146). This `JPanel` is put into a `JFrame` and set visible.

Inside the constructor (lines 15–37), `currOffset` is set to initially be `0` pixels. `currOffset` is the x-position of the image relative to the component's coordinate system. Because `currOffset` is set by the internal thread and read by `paint()`, it is `volatile` (line 10). The `buildImage()` method is called to create the off-screen image that scrolls (line 17). The rest of the constructor sets the dimensions of the component and starts up the internal thread.

The buildImage() method (lines 39–98) is used to prepare the off-screen image with the desired text. Because the text will be drawn to the image only once, the rendering hints are set for quality and anti-alias (lines 42–48). A scratch image is created first and used in determining the exact pixel dimensions needed for the specified text (lines 52–65). A little bit of horizontal and vertical padding is added and the real off-screen image is created (lines 67–79). A graphics context is created from the image and used for drawing the text (lines 81–91).

Whenever the event-handling thread calls paint() (lines 100–113), the off-screen image is redrawn onto the component. The value of currOffset is captured in the local variable localOffset in case currOffset is changed while paint() is in progress (line 104). Actually, the image is drawn twice. It is drawn once off to the left of the component by the localOffset (line 105). And it is drawn a second time by the same offset from the right side of the component (lines 106–107). Parts of the images will be automatically clipped because they extend off the sides of the component (line 102). After the image is in place, a black outline is drawn around the edge (lines 110–112).

The runWork() method (lines 115–130) is invoked by the internal thread that was started in the constructor. It loops continuously until another thread invokes the stopRequest() method. In the while loop, the internal thread sleeps for 0.1 seconds (line 118), increments currOffset (lines 121–122), and puts a request onto the event queue for the paint() method to be called (line 125). The value of currOffset is kept between 0 and the width of the off-screen image (line 122). currOffset is volatile so that the event thread sees the changes in value being made by the internal thread.

Figure 9.8 shows a snapshot of ScrollText in action. Figure 9.9 shows the same component a few seconds later. The text “Java can do animation!” is scrolling from right to left across the component. The main() method of ScrollText is simply used for demonstration purposes. ScrollText can be used as a component anywhere. You might want to enhance ScrollText so that the colors, font, scroll rate, and size of the scroll window can be specified for a more real-world application. You can speed up the scrolling by moving more than one pixel at a time, or by moving more than 10 times per second. Keep in mind that increasing the number of advances per second will use more processor resources.

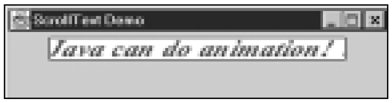


FIGURE 9.8
ScrollText—snapshot of text scrolling in progress.

FIGURE 9.9
ScrollText—another snapshot a few seconds later.

NOTE

Beginning with JDK 1.2, there is a class `javax.swing.Timer` that can be used to simplify animation. After being started, `Timer` calls the `actionPerformed()` method on the registered object at regular intervals. The event-handling thread is used to invoke `actionPerformed()`, so direct modification of visible components from within `actionPerformed()` is safe. If the action is nonblocking and brief, the use of `Timer` might be an appropriate substitute for the techniques I've shown you here.

Animating a Set of Images

Instead of scrolling one image across the face of a component as `ScrollText` does, a component can use an internal thread to step through a set of different images one image at a time. This set of images can be considered frames or slides. By flipping through the slides (or frames), the internal thread creates animation.

In `SlideShow` (see Listing 9.9), a set of images is created and an internal thread loops through them at a rate of 10 images per second. In this case, an expanding yellow circle is drawn on a blue background, but you could use any set of images.

LISTING 9.9 SlideShow.java—Animation of a Set of Images

```
1: import java.awt.*;
2: import java.awt.image.*;
3: import javax.swing.*;
4:
5: public class SlideShow extends JComponent {
6:     private BufferedImage[] slide;
7:     private Dimension slideSize;
8:     private volatile int currSlide;
9:
```

continues

LISTING 9.9 Continued

```
    private Thread internalThread;
    private volatile boolean noStopRequested;

    public SlideShow() {
        currSlide = 0;
        slideSize = new Dimension(50, 50);
        buildSlides();

        setMinimumSize(slideSize);
        setPreferredSize(slideSize);
        setMaximumSize(slideSize);
        setSize(slideSize);

        noStopRequested = true;
        Runnable r = new Runnable() {
                public void run() {
                    try {
                        runWork();
                    } catch ( Exception x ) {
                        // in case ANY exception slips through
                        x.printStackTrace();
                    }
                }
            };

        internalThread = new Thread(r, “SlideShow”);
        internalThread.start();
    }

    private void buildSlides() {
        // Request that the drawing be done with anti-aliasing
        // turned on and the quality high.
        RenderingHints renderHints = new RenderingHints(
            RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON);

        renderHints.put(
            RenderingHints.KEY_RENDERING,
            RenderingHints.VALUE_RENDER_QUALITY);

        slide = new BufferedImage[20];

        Color rectColor = new Color(100, 160, 250);   // blue
        Color circleColor = new Color(250, 250, 150); // yellow
```

```java

        for ( int i = 0; i < slide.length; i++ ) {
            slide[i] = new BufferedImage(
                    slideSize.width,
                    slideSize.height,
                    BufferedImage.TYPE_INT_RGB);

            Graphics2D g2 = slide[i].createGraphics();
            g2.setRenderingHints(renderHints);

            g2.setColor(rectColor);
            g2.fillRect(0, 0, slideSize.width, slideSize.height);

            g2.setColor(circleColor);

            int diameter = 0;
            if ( i < ( slide.length / 2 ) ) {
                diameter = 5 + ( 8 * i );
            } else {
                diameter = 5 + ( 8 *  ( slide.length - i ) );
            }

            int inset = ( slideSize.width - diameter ) / 2;
            g2.fillOval(inset, inset, diameter, diameter);

            g2.setColor(Color.black);
            g2.drawRect(
                0, 0, slideSize.width - 1, slideSize.height - 1);

            g2.dispose();
        }
    }

    public void paint(Graphics g) {
        g.drawImage(slide[currSlide], 0, 0, this);
    }

    private void runWork() {
        while ( noStopRequested ) {
            try {
                Thread.sleep(100);  // 10 frames per second

                // increment the slide pointer
```

continues

LISTING 9.9 Continued

```
 97:                 currSlide = ( currSlide + 1 ) % slide.length;
 98:
 99:                 // signal the event thread to call paint()
100:                 repaint();
101:             } catch ( InterruptedException x ) {
102:                 Thread.currentThread().interrupt();
103:             }
104:         }
105:     }
106:
107:     public void stopRequest() {
108:         noStopRequested = false;
109:         internalThread.interrupt();
110:     }
111:
112:     public boolean isAlive() {
113:         return internalThread.isAlive();
114:     }
115:
116:     public static void main(String[] args) {
117:         SlideShow ss = new SlideShow();
118:
119:         JPanel p = new JPanel(new FlowLayout());
120:         p.add(ss);
121:
122:         JFrame f = new JFrame("SlideShow Demo");
123:         f.setContentPane(p);
124:         f.setSize(250, 150);
125:         f.setVisible(true);
126:     }
127: }
```

In `main()` (lines 116–126), a new `SlideShow` instance is constructed (line 117) and put into a `JPanel` with a `FlowLayout` to let the instance of `SlideShow` take on its preferred size (lines 119–120). This `JPanel` is put into a `JFrame` and set visible.

Inside the constructor (lines 13–37), `currSlide` is set to initially be `0`. `currSlide` is the index into the `BufferedImage[]` referred to by `slide` indicating the current slide to display. Because `currSlide` is set by one thread (the internal thread) and read by another in `paint()` (the event thread), it must be `volatile` to ensure that the event thread sees the changes in value (line 8). The `buildSlides()` method is called to create the set of images used for the animation. The rest of the constructor sets the dimensions of the component and starts up the internal thread.

The `buildSlides()` method (lines 39–85) is used to construct an array of 20 images (line 50) to loop through. High-quality rendering hints are used because the images are drawn on only once and are displayed over and over (lines 42–48, 62). Each of the images is constructed and drawn on in the `for` loop (lines 55–84). First, a blue rectangle is filled in (lines 52, 64–65). Then a yellow circle of varying diameter is drawn in the center (lines 53, 67–77). The last shape drawn onto each image is a black rectangle to outline the slide (lines 79–81). Each graphics context is disposed of immediately when it is no longer needed (line 83).

Whenever the `paint()` method (lines 87–89) is called by the event thread, the current slide is drawn onto the component. Because `currSlide` is `volatile`, the event thread always sees the most recent index value.

The internal thread invokes the `runWork()` method (lines 91–105). Inside, it continues to execute the `while` loop until another thread comes along and invokes `stopRequest()`. Each time through, the internal thread sleeps for 0.1 seconds, increments the frame number, and requests that the event thread repaint the component as soon as possible (lines 94–100). The `slide` indexed by `currSlide` is kept in the range `0` to `(slide.length - 1)` (line 97). The internal thread loops through all of the slides over and over until `stopRequest()` is called.

Figure 9.10 catches `SlideShow` just as the yellow circle is beginning to expand. Figure 9.11 shows it when the yellow circle has expanded almost enough to touch the edges of the component. Figure 9.12 shows it when the yellow circle has grown to almost big enough to eclipse the entire blue region. After the yellow circle has grown to fill the component, it begins to shrink until it is a tiny circle again. This animation loop continues until `stopRequest()` is called. In this example I used simple drawing to keep the code size down, but you can feel free to use images of any complexity in this animation component.

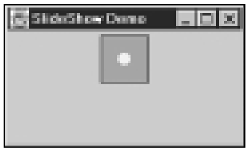


FIGURE 9.10
SlideShow when the yellow circle is just beginning to expand.

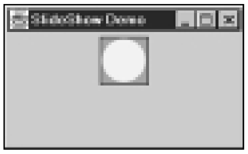


FIGURE 9.11
SlideShow when the yellow circle has almost expanded to the edges.

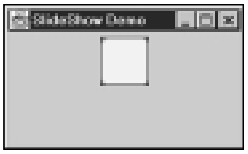


FIGURE 9.12
SlideShow when the yellow circle has almost engulfed the whole component.

Displaying Elapsed Time on a JLabel

`DigitalTimer` (see Listing 9.10) extends `JLabel` and uses an internal thread to update the text on the label with the elapsed time since the component was constructed. In this class it is important to use `SwingUtilities.invokeAndWait()` to have the event thread actually update the text on the label.

LISTING 9.10 DigitalTimer.java—Extending JLabel to Continually Display Elapsed Time

```
 1: import java.awt.*;
 2: import java.text.*;
 3: import java.lang.reflect.*;
 4: import javax.swing.*;
 5:
 6: public class DigitalTimer extends JLabel {
 7:     private volatile String timeText;
 8:
 9:     private Thread internalThread;
10:     private volatile boolean noStopRequested;
11:
12:     public DigitalTimer() {
13:         setBorder(BorderFactory.createLineBorder(Color.black));
14:         setHorizontalAlignment(SwingConstants.RIGHT);
```

```
        setFont(new Font(“SansSerif”, Font.BOLD, 16));
        setText(“00000.0”); // use to size component
        setMinimumSize(getPreferredSize());
        setPreferredSize(getPreferredSize());
        setSize(getPreferredSize());

        timeText = “0.0”;
        setText(timeText);

        noStopRequested = true;
        Runnable r = new Runnable() {
                public void run() {
                    try {
                        runWork();
                    } catch ( Exception x ) {
                        x.printStackTrace();
                    }
                }
            };

        internalThread = new Thread(r, “DigitalTimer”);
        internalThread.start();
    }

    private void runWork() {
        long startTime = System.currentTimeMillis();
        int tenths = 0;
        long normalSleepTime = 100;
        long nextSleepTime = 100;
        DecimalFormat fmt = new DecimalFormat(“0.0”);

        Runnable updateText = new Runnable() {
                public void run() {
                    setText(timeText);
                }
            };

        while ( noStopRequested ) {
            try {
                Thread.sleep(nextSleepTime);

                tenths++;
```

continues

LISTING 9.10 Continued

```
                long currTime = System.currentTimeMillis();
                long elapsedTime = currTime - startTime;

                nextSleepTime = normalSleepTime +
                    ( ( tenths * 100 ) - elapsedTime );

                if ( nextSleepTime < 0 ) {
                    nextSleepTime = 0;
                }

                timeText = fmt.format(elapsedTime / 1000.0);
                SwingUtilities.invokeAndWait(updateText);
            } catch ( InterruptedException ix ) {
                // stop running
                return;
            } catch ( InvocationTargetException x ) {
                // If an exception was thrown inside the
                // run() method of the updateText Runnable.
                x.printStackTrace();
            }
        }
    }

    public void stopRequest() {
        noStopRequested = false;
        internalThread.interrupt();
    }

    public boolean isAlive() {
        return internalThread.isAlive();
    }

    public static void main(String[] args) {
        DigitalTimer dt = new DigitalTimer();

        JPanel p = new JPanel(new FlowLayout());
        p.add(dt);

        JFrame f = new JFrame(“DigitalTimer Demo”);
        f.setContentPane(p);
        f.setSize(250, 100);
        f.setVisible(true);
    }
}
```

In `main()` (lines 89–99), a new `DigitalTimer` instance is constructed (line 90) and put into a `JPanel` with a `FlowLayout` to let it take on its preferred size (lines 92–93). This `JPanel` is put into a `JFrame` and set visible.

Inside the constructor (lines 12–37), the border, alignment, and font for the label are set. A sample text string of `00000.0` is used to initially size the component (lines 16–19). `timeText` is initialized to be `0.0`. `timeText` is declared to be `volatile` (line 7) because (after construction) it is set by the internal thread and read by the event thread. The rest of the constructor gets the internal thread up and running.

The internal thread invokes `runWork()` (lines 39–78) to keep track of time and update the label. Much of the work inside this method is done to keep the elapsed time as accurate as possible. (See Chapter 4, “Implementing Runnable Versus Extending Thread,” for a more in-depth discussion of the accuracy issues and techniques used.) The `Runnable` instance referred to by `updateText` (lines 46–50) is used by `SwingUtilities.invokeAndWait()` to get the event thread to update the text on the label. Notice that the same `Runnable` instance is used over and over inside the `while` loop. The `Runnable` reads the `volatile` member variable `timeText` to find out what text should be displayed.

In the `while` loop (lines 52–77), the internal thread sleeps for a while (about 0.1 seconds), increments the `tenths` counter, and calculates the elapsed time (lines 54–58). The `nextSleepTime` is calculated to keep the clock from running too fast or too slow (lines 60–65). The elapsed time is converted into seconds (from milliseconds) and formatted into a `String` that is stored in `timeText` (line 67). Next, `SwingUtilities.invokeAndWait()` is used to get the event thread to update the text currently displayed on the label (line 68). `SwingUtilities.invokeAndWait()` was used instead of `SwingUtilities.invokeLater()` so that the internal thread would not get ahead of the event thread.

Figure 9.13 shows how `DigitalTimer` appears after 15.5 seconds have elapsed.

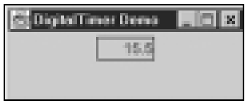


FIGURE 9.13
DigitalTimer after 15.5 seconds have elapsed.

Floating Components Around Inside a Container

`CompMover` (see Listing 9.11) is a utility that takes a component and an initial position and moves the component around inside its container. This is basically a demonstration of how animation can be achieved by moving components.

LISTING 9.11 CompMover.java—A Utility to Float Components Around Inside a Container

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

public class CompMover extends Object {
    private Component comp;
    private int initX;
    private int initY;
    private int offsetX;
    private int offsetY;
    private boolean firstTime;
    private Runnable updatePositionRun;

    private Thread internalThread;
    private volatile boolean noStopRequested;

    public CompMover(Component comp,
                int initX, int initY,
                int offsetX, int offsetY
            ) {

        this.comp = comp;
        this.initX = initX;
        this.initY = initY;
        this.offsetX = offsetX;
        this.offsetY = offsetY;

        firstTime = true;

        updatePositionRun = new Runnable() {
                public void run() {
                    updatePosition();
                }
            };

        noStopRequested = true;
        Runnable r = new Runnable() {
                public void run() {
                    try {
                        runWork();
                    } catch ( Exception x ) {
                        // in case ANY exception slips through
                        x.printStackTrace();
                    }
                }
            };

        internalThread = new Thread(r);
        internalThread.start();
    }

    private void runWork() {
        while ( noStopRequested ) {
            try {
                Thread.sleep(200);
                SwingUtilities.invokeAndWait(updatePositionRun);
            } catch ( InterruptedException ix ) {
                // ignore
            } catch ( Exception x ) {
                x.printStackTrace();
            }
        }
    }

    public void stopRequest() {
        noStopRequested = false;
        internalThread.interrupt();
    }

    public boolean isAlive() {
        return internalThread.isAlive();
    }

    private void updatePosition() {
        // should only be called by the *event* thread

        if ( !comp.isVisible() ) {
            return;
        }

        Component parent = comp.getParent();
        if ( parent == null ) {
            return;
        }

        Dimension parentSize = parent.getSize();
        if ( ( parentSize == null ) &&
```

continues

LISTING 9.11 Continued

```
         ( parentSize.width < 1 ) &&
         ( parentSize.height < 1 )
       ) {

        return;
      }

      int newX = 0;
      int newY = 0;

      if ( firstTime ) {
        firstTime = false;
        newX = initX;
        newY = initY;
      } else {
        Point loc = comp.getLocation();
        newX = loc.x + offsetX;
        newY = loc.y + offsetY;
      }

      newX = newX % parentSize.width;
      newY = newY % parentSize.height;

      if ( newX < 0 ) {
        // wrap around other side
        newX += parentSize.width;
      }

      if ( newY < 0 ) {
        // wrap around other side
        newY += parentSize.height;
      }

      comp.setLocation(newX, newY);
      parent.repaint();
    }

    public static void main(String[] args) {
      Component[] comp = new Component[6];

      comp[0] = new ScrollText("Scrolling Text");
      comp[1] = new ScrollText("Java Threads");
      comp[2] = new SlideShow();
      comp[3] = new SlideShow();
      comp[4] = new DigitalTimer();
```

```
132:         comp[5] = new DigitalTimer();
133:
134:         JPanel p = new JPanel();
135:         p.setLayout(null); // no layout manager
136:
137:         for ( int i = 0; i < comp.length; i++ ) {
138:             p.add(comp[i]);
139:
140:             int x = (int) ( 300 * Math.random() );
141:             int y = (int) ( 200 * Math.random() );
142:             int xOff = 2 - (int) ( 5 * Math.random() );
143:             int yOff = 2 - (int) ( 5 * Math.random() );
144:
145:             new CompMover(comp[i], x, y, xOff, yOff);
146:         }
147:
148:         JFrame f = new JFrame("CompMover Demo");
149:         f.setContentPane(p);
150:         f.setSize(400, 300);
151:         f.setVisible(true);
152:     }
153: }
```

The constructor for `CompMover` (lines 16–49) takes a component, an initial position, and x and y offset information. A `Runnable` is created (lines 29–33) to be passed to `SwingUtilities.invokeAndWait()`. This `Runnable` is referred to by `updatePositionRun` and invokes the `updatePosition()` when called by the event thread. The rest of the constructor gets the internal thread up and running.

The internal thread invokes `runWork()` (lines 51–62), where it loops inside the `while` until another thread invokes `stopRequest()`. Inside the `while` loop, the thread sleeps for 0.2 seconds and then invokes `SwingUtilities.invokeAndWait()` passing in `updatePositionRun` (lines 54–55). `updatePositionRun` causes the event thread to invoke `updatePosition()`.

Each time that the event thread calls `updatePosition()` (lines 73–122), the event thread attempts to move the component a little. Several checks are done to be sure that the parent container is accessible (lines 76–92). The current location of the component is retrieved and the x and y offsets are added to determine the new location (lines 97–118). The event thread proceeds to invoke `setLocation()` on the component to move it to its new position (line 120). The event thread then invokes `repaint()` on the parent container to get the move to show up (line 121).

In `main()` (lines 124–152), a number of components are constructed: two instances of `ScrollText`, two instances of `SlideShow`, and two instances of `DigitalTimer` (lines 125–132). A panel is created to house these components, and it has its layout manager set to `null` because `CompMover` is taking care of component positions (lines 134–135). Inside the `for` loop (lines 137–146), each component is added to the panel (line 138) and has its initial position and x and y offsets randomly determined (lines 140–143). Each component also gets handed off to a new instance of `CompMover` to handle its positioning (line 145).

Each of the six components has an internal thread running within it to handle its animation. In addition, each of the six instances of `CompMover` also has an internal thread running to handle the component movement. All 12 of these threads perform many operations per second and can bog down the processor. If you don't have a really fast machine, you might notice some sluggishness when you run this example. As you can see, animation is very processor-intensive.

Figure 9.14 shows how `CompMover` looks after running for about 75 seconds. Figure 9.15 shows how it looks after about 136 seconds. Each of the components travels around in a different direction. When a component moves off one side of the screen, it returns on the other side. Your output will differ significantly because the initial positions and directions of movement for the components are randomly determined.

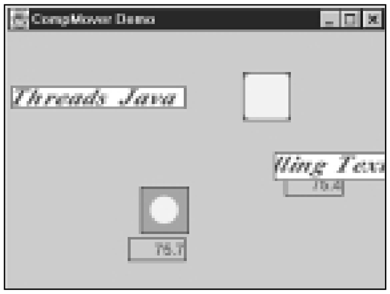


FIGURE 9.14
A snapshot of CompMover in action.

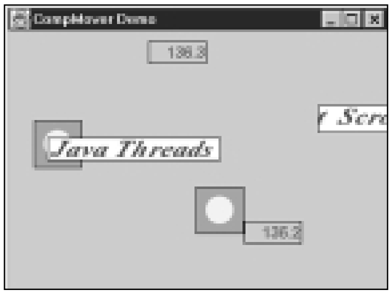


FIGURE 9.15
Another snapshot of CompMover after more time has passed.

Summary

In this chapter, you saw how it is important that the event thread be the only thread that makes direct modifications to Swing components after they have been added to a visible container. The `SwingUtilities.invokeAndWait()` and `SwingUtilities.invokeLater()` methods provide a mechanism for any thread to put a block of code onto the event queue. When the event thread gets to the block of code, it executes it and safely makes changes to Swing components. Using these tools, threads were added to components to provide animation capabilities. Additionally, a worker thread was able to take the results of a long-running search and safely update the graphical interface of an application.