Java Toolbox by Allen Holub

Programming Java threads in the real world, Part 5

Has Sun abandoned 'run anywhere'? Plus: Threads and Swing, timers, and getting around stop(), suspend(), and resume() deprecation

Summary
In an aside to his running series on programming Java threads in the real world, Allen prefaces this month's column with a look at one dangerous new "feature" of the Java 2 platform: the ability to produce noncompliant class files that will not run on a 1.1 Java virtual machine (JVM).

From there, he picks up the theme from Part 4 of this series. This month's main topic is timers, both the one in the Swing library (called `Timer`) and also a roll-your-own variety called `Alarm`, which is useful when the Swing `Timer` isn't available or appropriate.

Timers let you perform fixed-interval operations, such as animation refreshes. The `Alarm` implementation included here is by far the most complicated of the classes you've looked at so far, and it demonstrates several important Java programming techniques, such as how to write code that needs to suspend and resume threads without using the (now deprecated) `suspend()` and `resume()` methods. Along the way, Allen reveals how to kill a thread gracefully, without using the (also deprecated) `stop()` method. *(6,000 words)*

I'm going to start off this month on a deliberately alarmist note. Feel free to skip to the next section if my foaming at the mouth starts to bother you. Bear in mind, as you read this, that I'm one of the "good guys." I think Java is the best thing since sliced bread. In fact, I've abandoned C++ for Java. Virtually all of my work nowadays centers around Java programming -- either doing it or teaching it. Nonetheless, I'm hoping that some public criticism, as in the following, might encourage Sun to operate in a more responsible fashion than is described here.

Whatever happened to 'run anywhere'?

So, what's so upsetting? It seems Sun has abandoned the "run anywhere" part of its oft-quoted write-once, run-anywhere mantra. I'll explain. The ability to "run its oft-quoted write-once, run-anywhere mantra. I'll explain. The ability to "run anywhere" is predicated on a class file being able to run on any machine that has a Java virtual machine (JVM) on it, and on that VM being able to run any class file it's given. If different implementations of the JVM recognize different class file formats, platform independence is a pipe dream. Microsoft understands this issue very well. That's why it tried to push a nonconforming VM onto Windows. Programs that leverage Microsoft's nonstandard extensions simply are not platform-independent.

This standardization is particularly important in a network environment, where an application installed on a server and distributed automatically must be able to run on all the client boxes, regardless of the VM that's installed on a given box. You can always download a new library -- that's one of the reasons Swing is in the `javax` (rather than `java`) package. Updating the VM itself is another matter. Sun is well aware of this issue. James Gosling himself has often justified changes to the language at the source-code level by saying that no VM changes would be mandated.

In the face of this fact, I was horrified to find that the Java 2 platform (formerly known as JDK 1.2) compiler can create a nonstandard class file format recognizable only to the Java 2 VM. The class file generated by `java -target 1.2` *will not run* on a 1.1 VM. Since you have absolutely no control over which version of the VM is on the client platform, using this switch effectively throws away any hope of platform independence. You'll have to ship your Java application with a version of the Java 2 VM for every platform on which you'd hope to run, and we're back in DLL hell.

Fortunately, the switch is off by default, and you don't have to use it. But even the presence of this switch indicates that Sun is perhaps not as trustworthy as the Java community has been assuming. At a recent Java SIGs meeting hosted by San Jose's Software Development Forum, Gosling stated that even though Java has maintained backward compatibility as it has evolved, Sun has no commitment to this compatibility. He implied that there would eventually come a time where we would all be forced to adopt the new class file format. I can see it now: a Microsoft PR guy waving around a disk, shouting "Here's an official Java application, compiled with Sun's own Java compiler, and it won't run on these Windows, Solaris, or Mac boxes unless you spend hours upgrading an obscure piece of software called the virtual machine. In fact, there's no guarantee that you can even get such an upgrade if you're not running Solaris. Platform independence? Ha!"

The fact that the new nonstandard class file format was sprung on the Java community with no advanced warning or feedback seems to me the worst problem of all. There must be a way to move forward, of course. It's inevitable that the VM spec will change as Java evolves. However, this change should come about through an open process that involves the whole Java community, not a unilateral move on the part of Sun. This no-questions-asked imposition of a new standard strikes me as a betrayal of Sun's commitment to Java as an open standard, and calls Sun's legitimacy as a standards-setting body into question. It seems silly to gripe about Microsoft creating a noncompliant compiler/VM when Sun itself has now done exactly that. I suppose I've been deluding myself into thinking Sun was acting as the steward of an open standard. It turns out that Sun owns the standard, and like Microsoft, does what it feels like whenever it feels like doing it.

Frankly, I can't imagine what would be so important that Sun would be willing to change the VM spec at this point in time. As far as I can tell from experimenting, the generated bytecode is identical to the 1.1 bytecode except for the new version number. (This one difference is enough to prevent a 1.1 VM from loading a Java 2 class file, however.) Speaking as a compiler writer, there are a few annoyances in the bytecode -- mainly workarounds for VM bugs -- but these are trivial matters. I've asked both Gosling and Tim Lindholm (the latter is one of the coauthors of the VM) why the change was made, but neither has given me an answer. Perhaps this ploy is intended solely to render the Microsoft VM irrelevant, since it won't conform to the Java 2 spec. If one of you readers can enlighten me, please send me e-mail; my address is available in the bio below.

Back to threads

On a more prosaic (or perhaps Prozac) note, let's get back to the subject at hand: threads. This month, I want to talk about *timers* -- objects that help execute some operation at a fixed interval. I'll discuss both the timer that's part of the Swing package and a roll-your-own timer that I've found more useful in some situations.

It's often the case that programs need to execute some operation at a regular interval. Animation loops come to mind immediately, but other applications are legion. Consider the case of a thread that must notify a large list of observers that some event has occurred. Using a timer to notify one observer from the list on each "tick" of the timer can give other threads a chance to run while notifications are in progress.

A simple timed wait doesn't often do the trick, though. For example, each loop iteration in the following example could takes a different amount of time to execute, depending on which way the test goes and how long the database server takes to process requests:

```
while( running )
{
    if( some_condition )
        notify_any_observers( FAILED );
    else
    {
        open_a_database();
        make_1000_queries();
        process_the_results():
        generate_a_billion_reports();
        notify_any_observers( SUCCESS );
    }

    wait( FIXED_INTERVAL ); // There is no matching notify(). Just time out.
}
```

Swingin' threads

The first solution to this problem is the new `javax.swing.Timer` class. The `Timer` is designed to be a Swing-specific solution (it's only available if Swing is installed on the client system) to a Swing-specific problem: Swing itself is not thread-safe.

The thread-safety issue is an important one. Swing's threading support is described in depth in a Tech note on the Sun Web site, but to summarize: Swing, like AWT before it, uses a single thread to field all user interface (UI) events that come in from the operating system. This thread dequeues OS events from some sort of event queue, figures out what the OS is trying to tell it, and then notifies any listeners interested in the event. For example, the event thread could dequeue a `WM_MOUSEMOVE` message from the

Windows message queue, and then send out `mouseMoved()` messages to all the interested `MouseMotionListener` objects.

These event handler methods (such as `mouseMoved()`) *actually run on the thread that's dequeueing the OS-level events.* This single-threaded approach is a big issue, because the UI is effectively locked while the event handler messages are executing. That is, no OS-level messages (such as button presses) are serviced while listener notifications are in progress. To keep your UI responsive to user input, these event handler functions should be very short and very fast. They must spawn off threads to do time-consuming operations.

To make matters more interesting, none of Swing class's methods are thread safe -- they are simply not designed to handle the case of two threads accessing them simultaneously. The performance hit required to make Swing thread-safe (not to mention the extra complexity) is just unacceptable. The lack of thread safety means that once you start up the Swing event handler thread (by calling `setVisible()`, `pack()`, or any other method that makes a window visible), you cannot safely call a Swing method from anywhere but Swing's own event handler thread. (Note that you can safely manipulate Swing objects *before* the event handling thread starts -- before you call `setVisible()` or `pack()`, for example.)

The lack of thread safety is often not an issue because Swing methods are, more often than not, called from Swing's own event handler thread -- remember, all the listener methods run on that thread. For example, a listener's event handler like `mouseMoved()` might pass a `repaint()` message to some `Component`. This operation is safe because `mouseMoved()` is running on Swing's own thread.

The invokeLater() and invokeAndWait() methods

There are often cases where some thread of your own devising needs to communicate with Swing, however. For example, the main thread of an application could fire up a window, do some stuff, then want to modify the window. Since you obviously need to pass messages to Swing at times other than initialization time, a mechanism has been provided to ask the Swing event-handling thread to execute a block of code for you. You need to encapsulate your code into a `Runnable` object, and then pass that object to Swing by calling `SwingUtilities.invokeLater()` or `SwingUtilities.invokeAndWait()`. For example, a thread could force a window to redraw itself like this:

```
Component some_window = ... ;

Swingutilities.invokeLater( new Runnable()
                            {   public void run()
                                {   some_window.repaint();
                                }
                            }
                          );
```

The asynchronous `invokeLater()` message just puts the `Runnable` object onto the event queue and then returns. The object's `run()` method is executed when Swing gets around to it. The `invokeAndWait()` method blocks until the `run()` function has been executed. Since the associated `run()` methods run on the Swing event handler thread, synchronously with other Swing methods, the thread safety of the individual Swing methods is irrelevant.

Don't be misled by the fact that you're using a `Runnable` object to pass a request to Swing. The `run()` method doesn't run on its own thread; it uses the Swing event queue. If `run()` takes a long time to execute, your program's user interface will appear to lock up -- it won't respond at all to any user input.

It's best for these methods either to be very short or to fire up their own threads to do time-consuming background operations.

Using the Swing Timer

One common situation where a standalone thread needs to talk to Swing is an animation loop. Listing 1 below shows how you might be tempted to implement an animation loop. This is typical AWT-style code, where the `run()` method of a thread (line 42) forces a repaint at a regular interval. The thread is fired up in `start()` (line 23). Unfortunately, this code only *appears* to work -- it will probably run fine until you try to actually use an `Animator1` object in a real application that uses Swing intensively, then it will blow up. The `run()` method on line 42 of Listing 1 runs on the thread created on line 24, not on the Swing thread. Consequently, the call to `repaint` on line 51 is not safe. We're calling an unsynchronized Swing method from somewhere other than a listener method or the Swing event processing thread.

Also, the animation loop is a bit jerky because the repaint doesn't actually happen on a fixed interval. There's a difference between repainting every 100 milliseconds and waiting 100 milliseconds between repaint requests. In the latter case, which is what I'm doing here, the actual time between refreshes can vary, depending on how long it takes for `repaint()` to execute. That's not really the behavior I want.

Listing 1. (Animator1.java) The wrong way to do animation

```
01  import java.awt.*;
02  import javax.swing.*;
03
04  public class Animator1 extends JComponent implements Runnable
05  {
06      private final int     refresh_rate;
07      private final Image[] frames;
08      private       int     current_frame = 0;
09      private       Thread  driver;
10      //-------------------------------------------------------------------
11      public Animator1( Image[] frames, int refresh_rate )
12                                             throws InterruptedException
13      {
14          this.frames       = frames;
15          this.refresh_rate = refresh_rate;
16
17          MediaTracker tracker = new MediaTracker(this);  // wait for all
18          for( int i = 0; i < frames.length; ++i )        // images to
19              tracker.addImage( frames[i], 0 );           // load
20          tracker.waitForAll();
21      }
22      //-------------------------------------------------------------------
23      public void start()
24      {   driver = new Thread( this );
25          driver.start();
26      }
27      //-------------------------------------------------------------------
28      public void stop()
29      {   driver.interrupt();
30      }
31      //-------------------------------------------------------------------
32      public Dimension getPreferredSize()
33      {   return new Dimension(   frames[0].getWidth(this),
34                                  frames[0].getHeight(this) );
35      }
```

```
    //------------------------------------------------------------------
    public void paint( Graphics g )
    {   g.drawImage( frames[ current_frame ], 0, 0,
                        getBounds().width, getBounds().height, this );
    }
    //------------------------------------------------------------------
    public void run()
    {   try
        {   while( !Thread.interrupted() )
            {
                Thread.currentThread().sleep( refresh_rate );

                ++current_frame;
                current_frame %= frames.length;

                Animator1.this.repaint();
            }
        }
        catch( InterruptedException e ){/*terminate thread by returning*/}
    }
    //==================================================================
    public static class Test extends JFrame
    {
        static public void main( String[] s ) throws Exception
        {
            Toolkit kit     = Toolkit.getDefaultToolkit();
            Image[] images  = new Image[]
            {
                kit.getImage("5.images/Juggler0.gif"),
                kit.getImage("5.images/Juggler1.gif"),
                kit.getImage("5.images/Juggler2.gif"),
                kit.getImage("5.images/Juggler3.gif"),
                kit.getImage("5.images/Juggler4.gif")
            };

            Animator1 animator = new Animator1( images, 100 );

            JFrame frame = new Test();
            frame.getContentPane().add( animator );
            frame.pack();
            frame.show();

            animator.start();
            Thread.currentThread().sleep( 5 * 1000 );
            animator.stop();
            System.exit(0);
        }
    }
}
```

Killing threads

You should pay attention to how run() terminates in Listing 1. Jeff Richter, in his book, *Advanced Windows Programming,* says that there are three ways in which threads can die: they can die a natural death, commit suicide, or be murdered! Java’s `stop()` method, use of which falls in both the murder and suicide category, is now deprecated because it can leave NT DLLs in an unstable state, so "murder" and "suicide" aren’t options. The remaining option, a "natural death," is brought about by returning from `run()`. If you want to shut a thread down, you have to ask the thread to shut itself down.

A simple approach might simply call a method that sets an internal flag that's tested in the main `run()` loop of the thread. This approach doesn't usually work, however, because most threads spend the bulk of their time blocked, waiting for some operation like a blocking read to complete. A suspended thread can't test a flag. However, sending a blocked thread an `interrupt()` message will bump it out of the wait (with an exception toss). The `stop()` method (line 28) does just that: it sends an `interrupt()` message to the driver thread in order to stop it. (Remember, `stop()` itself runs on whatever thread calls it, not on the animator thread. The only methods that run on the animator thread are the ones that `Animator.run()` calls.) Since I intend for an interrupt operation to kill the thread, the `run()` method (line 42) catches the exception and ignores it.

What if the thread is interrupted, rather than blocked, while it's working? The `Thread` class provides two methods to check whether or not a thread has been interrupted. `Thread.interrupted()`, which I've used here, returns true if the current thread has been interrupted. A call to `some_thread.isInterrupted()` returns true if `some_thread` has been interrupted. It's a good idea to call one or the other of these periodically as the thread works.

Unit tests
The final thing to look at in Listing 1 is the unit-test class: `Test`, on line 57. A *unit test* is a piece of code that tests one thing, in this case a class. It's a good idea to associate a unit test with every class you write, and there are several ways to do that. You could pair a test file with every class file, but you'd end up with too many files to maintain if you did that, and keeping the test routine in phase with the class as it evolves would be difficult since you could recompile the class without recompiling the test.

Another strategy is to put a `main()` in every class. I don't do that because I don't want to carry around the baggage of a unit test in every class file that comprises the program. The tests are often bigger than the class I'm testing, and I don't want to inflate the application size with all this test code.

A third option -- putting the test code in a separate class, but in the same file as the class being tested -- doesn't work out very well because the package-level name space becomes corrupted with all the names of the test classes. Just managing these names can become a big problem.

The strategy I eventually hit on is to test a class using an inner class of the class I'm testing. I always call this class `Test`. This inner class resides in its own class file, and I don't ship it with the application, so it doesn't increase the application's size. The test class's name is within the name space of the outer class, so the global name space isn't corrupted with stuff the user of the class couldn't care less about. Given a set of classes listed in a makefile, I can easily run a whole battery of tests from a makefile by deriving the test class name from the source file name using macros. For example, I can test the current class by executing:

```
java Animator1\$Test
```

(You don't need the backslash on Windows machines.) You can't write `java Animator.Test` because the VM will think you're trying to run the `Test` class in the `Animator` package.

Use Timer
Getting back to animations, Swing solves the it's-not-safe-to-call-a-Swing-method-from-another-thread problem with the `Timer` class, which executes requests on Swing's own event processing thread at fixed intervals. I think of this kind of timer as "proactive." You set up a timer interval and register one or more

ActionListener objects with it. The Timer then sends an actionPerformed() message to each listener when the interval elapses. (All the listeners are notified at once, not one per interval.) These notifications are executed from the Swing event loop, so they can happily call Swing methods without worrying about thread synchronization.

Listing 2 below demonstrates how to use a timer by printing the time once per second. The Timer is initialized with a time interval (1000 milliseconds) and is provided an ActionListener to notify when the interval expires. The listener's actionPerformed() method both prints the current time and advances the elapsed-time counter. After five seconds (and five calls to actionPerformed()), the condition variable (which I discussed last month) is set to true, thereby releasing any threads that are waiting for the timer to finish. The main thread creates a Swing_timer, the constructor of which starts up the clock, then blocks, waiting for the done condition to become true. The net effect is that the constructor doesn't return until five seconds have elapsed.

Sun's Timer is a bit more flexible than I've shown; it can also work in one-shot mode, where the listeners are notified only once, for example. The Timer also supports methods that let you add additional listeners, change the timer interval, and so forth.

Listing 2. (Swing_timer.java) Using javax.swing.Timer

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

import com.holub.asynch.Condition; // Discussed last month.

public class Swing_timer
{
    private Condition done          = new Condition(false);
    private int       elapsed_time  = 0;

    // Create a one-second timer:

    Timer clock = new Timer( 1000,
                        new ActionListener()
                        {   public void actionPerformed( ActionEvent e )
                            {   synchronized( Swing_timer.this )
                                {   System.out.println( (new Date()).toString()
                                    if( ++elapsed_time == 5 )
                                        done.set_true();
                                }
                            }
                        }
                    );

    Swing_timer() throws InterruptedException
    {   clock.start();
        done.wait_for_true();
    }

    public static void main(String[] args) throws InterruptedException
    {   new Swing_timer();
    }
}
```

So, how does it work?
It's interesting to delve into the inner workings of Swing's `Timer`. (The sources are in the JDK's src.jar file, installed in the JDK root directory if you tell the installer program to install it.)

You would expect a timer to fire off a thread whose `run()` method would first wait for the time interval to elapse, then notify listeners. This thread would then either terminate or loop back up and wait again, depending on the timer's mode. That's more or less how my `Alarm` class (which I'll present in a moment) works, and that's how our failed attempt at an animation loop worked as well.

Swing's `Timer` doesn't work this way, however. What you don't see on the surface is a second package-access class, called a `TimerQueue`. This class is a Singleton -- only one instance of it exists. It's the `TimerQueue` object that encapsulates the "wait" thread. The `Timer` object actually doesn't do any timing at all. It just stores its time interval internally. When you start it up, the `Timer` object adds itself to the `TimerQueue`. The queue keeps the timers ordered by expiration time, and puts itself to sleep until that time arrives. The `TimerQueue` object then passes the ball back to the expired timer, asking it to notify its listeners.

At this juncture, the `Timer` object calls Swing's `invokeLater()` method (discussed earlier) and passes it a `Runnable` object whose `run()` method sends `ActionPerformed()` messages to the timer's `ActionListener` objects. It's important to observe that this notification happens on Swing's event processing thread, so the `ActionPerformed()` method can happily call Swing functions without worrying about thread safety.

By default, if several references to the *same* timer object are waiting in the notification queue, all but one are ignored. (Notification requests can build up in this way if the application is very busy servicing normal OS-event handlers.) This way, a listener doesn't get multiple notifications coming one on top of the other. This behavior can be suppressed by passing the `Timer` object a `setCoalesce(false)` message.

In Listing 3, below, I've rewritten the earlier animation example to use a Swing `Timer`. The new code is in boldface, and the main difference between this version and the previous one is the replacement of the `run()` method with an `actionPerformed()`, which is invoked from the Swing event thread when the timer expires.

Listing 3. (Animator.java) Animation with a Swing Timer

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

public class Animator extends JComponent  implements ActionListener //
{
    Timer   timer;

    private final int     refresh_rate;
    private final Image[] frames;
    private       int     current_frame = 0;
    private       Thread  driver;
    //-----------------------------------------------------------------
    public Animator( Image[] frames, int refresh_rate )
                                        throws InterruptedException
    {
        this.frames       = frames;
        this.refresh_rate = refresh_rate;

        timer= new Timer(refresh_rate, this);
        timer.setInitialDelay(0);
        timer.setCoalesce(true);

        MediaTracker tracker = new MediaTracker(this);  // wait for all
        for( int i = 0; i < frames.length; ++i )        // images to
            tracker.addImage( frames[i], 0 );           // load
        tracker.waitForAll();
    }
    //-----------------------------------------------------------------
    public void start() //
    {   timer.start();
    }
    //-----------------------------------------------------------------
    public void stop()      //
    {   timer.stop();
    }
    //-----------------------------------------------------------------
```

```
    public Dimension getPreferredSize()
    {   return new Dimension(   frames[0].getWidth(this),
                                frames[0].getHeight(this) );
    }
    //------------------------------------------------------------------
    public void paint( Graphics g )
    {   g.drawImage( frames[ current_frame ], 0, 0,
                        getBounds().width, getBounds().height, this );
    }
    //------------------------------------------------------------------
    public void actionPerformed( ActionEvent e )
    {   ++current_frame;
        current_frame %= frames.length;
        repaint();
    }
    //==================================================================
    public static class Test extends JFrame
    {
        static public void main( String[] s ) throws Exception
        {
            Toolkit kit     = Toolkit.getDefaultToolkit();
            Image[] images  = new Image[]
            {
                kit.getImage("5.images/Juggler0.gif"),
                kit.getImage("5.images/Juggler1.gif"),
                kit.getImage("5.images/Juggler2.gif"),
                kit.getImage("5.images/Juggler3.gif"),
                kit.getImage("5.images/Juggler4.gif")
            };

            Animator animator = new Animator( images, 100 );

            JFrame frame = new Test();
            frame.getContentPane().add( animator );
            frame.pack();
            frame.show();

            animator.start();
            Thread.currentThread().sleep( 5 * 1000 );
            animator.stop();
            System.exit(0);
        }
    }
}
```

Why use a Swing Timer?

The main advantage to Swing's approach (over and above the thread-safety issue) is that the number of threads that exist at a given moment is small, since all the real work is done on the Swing event thread, and all timers share the same `TimerQueue`.

There are lots of disadvantages, though:

- If your program doesn't have a GUI, or if it's using 1.1 AWT instead of Swing, it can't use a `Timer`.
- The notification is performed on Swing's event thread. This is great in the context of sending

messages to Swing components but is a problem everywhere else. If the event queue is full of operating-system events like mouse moves, it could take quite a while for your `ActionPerfomed()` method to be called.

- Since `ActionPerformed()` executes on the event thread, the UI is locked up while the `ActionPerformed()` method is executing. It's a bad idea to lock up the UI by doing something unrelated to the UI.

The main problems, however, are the synchronization problems discussed last month in the context of AWT: the `actionPerformed()` methods run on Swing's event thread, which typically is not the thread that's waiting for the time to expire. This means you must remember to synchronize the listener on the outer-class object (as I've done on line 17 of Listing 2). You must also use condition variables or a wait-notify strategy to synchronize the waiting thread and the notifying thread. For all the reason I discussed last month, it's this last set of problems that are the most bothersome in practice. That's why I didn't use Swing's architecture when I came up with my own timer class.

Roll your own

My own timer class, called `Alarm`, is different from the Swing `Timer` in several ways. First, it doesn't have to use the "observer" pattern at all. You can simply wait for the timer to expire by calling a blocking function that suspends the calling thread until the next timer tick occurs. This strategy has two advantages when compared to Swing's `Timer`: First, it's more in line with an object-oriented way of thinking. In object-oriented systems, you tend to think in terms of objects sending each other synchronous messages (which don't return until they've finished doing whatever they do) or asynchronous messages (which return immediately, while whatever they do goes on in the background). You're thinking *messages* here, not threads. Second, it's easier to code my `Alarm` than a Swing `Timer`. You don't have to worry about the listener-related synchronization problems, and you don't have to mess with `wait()` and `notify()` or a condition variable to find out when the timer has ticked. The Swing approach makes more sense when you can do everything you need to do in the `actionPerformed()` method, but I've found that most of the applications for timers I've come up with lend themselves more to a blocking-wait strategy.

Here's the earlier print-the-time-once-a-second example from Listing 2, rewritten to use my `Alarm` class:

```
static public void main( String[] args )
{
    Alarm clock = new Alarm(1000, Alarm.CONTINUOUS);
    clock.start();

    for( int i = 3; --i>= 0; )
    {   System.out.println( new Date().toString() );
        clock.await( Alarm.FOREVER );
    }

    t.stop();
}
```

Note that the clock is running while the `Date()` object is being created and printed. The time required for this operation isn't added to the 1000-millisecond interval specified in the constructor call. If the loop takes longer than 1,000 milliseconds to execute, the program will miss a timer tick. Contrast this with the Swing approach, where timer ticks can be missed because the I/O system is too busy to notify the listeners. It's not that one situation is better or worse than the other, but there is a difference.

To my mind, the code that uses an `Alarm` is both easier to write and easier to read than that of Listing 2. It also doesn't require any condition variables for synchronization, because the `Alarm` object isn't sending messages to the main thread asynchronously; rather, the main thread simply waits for the timer to expire with a blocking call. There are still two threads, but the timer thread is doing nothing but timing -- it has no access to the code in the class that contains `main()`. Therefore, the main thread does not have to synchronize access to any fields because only one thread accesses those fields.

If you really need Swing-style event-based notification, the `Alarm` class supports that as well. Just add an `ActionListener` to the thread with a call to `addActionListener()`, and remove the listener with a call to `removeActionListener()`. Here's an example:

```
clock = new Alarm( 1000 );  // 1-second continuous timer
Alarm.Notifier notifier = new Alarm.Notifier( clock );

notifier.addActionListener
(   new ActionListener()
    {   public void ActionPerformed()
        {   System.out.println( new Date().toString() );
        }
    }
);

clock.start();
```

When you use an `ActionListener`, a second notification thread is created solely to notify the listeners. This second thread sleeps, waiting for the `Alarm` to go off, then notifies the listeners, then goes back to sleep. The notification thread destroys itself automatically when there are no more listeners to notify. Since a second thread is used for notification, the time spent executing `actionPerformed()` methods will not impact the `Alarm`'s timing. Since the `Alarm` doesn't use the Swing thread to run these listener methods, you don't have to worry about locking up the UI if a listener takes too long to do its work. But, by the same token, you do have to call `SwingUtilities.invokeLater()` or `SwingUtilities.invokeAndWait()` from your `ActionPerformed()` method to safely talk to Swing.

The notification thread is a "daemon." The fact that it's running will not keep the program alive. The `Alarm` itself has a reference to it, however, so as long as you have a reference to the `Alarm` object with which listeners have been registered, the notifier thread will hang around.

How does an Alarm work?

The source code for the `Alarm` class is in Listing 4, below. There's a lot to talk about here. Starting from the top, I've declared references for the time between ticks, and the actual timer thread, `clock`Listing 4, line 24). This thread spends most of its time suspended, waiting for the time interval to expire. I also declare a flag `is_stopped` (line 25) so I can tell the difference between a timer that's expired and one that's been stopped by the user. I need this flag because threads that are waiting on the `Alarm` are notified both on expiration and when the `Alarm` is sent a `stop()` method. This flag lets me return different values from `wait()`, so the waiting threads can distinguish the normal-expiration case from the timer-being-stopped case.

The next few lines define constants that control the way an `Alarm` works. `Alarm`s run in one of three modes: `CONTINUOUS`, `ONE_SHOT`, and `MULTI_SHOT`, as defined symbolically on lines 29, 31, and 33, respectively. In `CONTINUOUS` mode, the `Alarm` runs continuously, firing off events (or releasing any waiting threads) at regular intervals. A `ONE_SHOT Alarm` runs only once, then shuts itself down, freeing all support-level threads. You can restart the timer by calling `start()` a second time. A `MULTI_SHOT` timer works like just like a `ONE_SHOT`; it doesn't free the underlying timer thread, however. Use a `MULTI_SHOT` when you know you're going to fire up the `Alarm` again and don't want to incur the thread-creation overhead.

All three of these symbolic constants are instances of the `Mode` class, defined on line 27. This class demonstrates the proper way to do an enumerated type in Java. Because of the private constructor, only three instances of `Mode` will ever exist: `Alarm.CONTINUOUS`, `Alarm.ONE_SHOT`, and `Alarm.MULTI_SHOT`. Consequently, the values of a `Mode` are constrained, as is only proper for an enumerated type. Since the `Mode` class itself is `public`, other classes can access a `Mode`, declare references to `Mode` objects, and so on. But the private constructor prevents other classes from creating a new `Mode`. A method can now take a `Mode` argument (as does the constructor on line 48), and the compiler -- not the runtime system -- can guarantee that that argument will be either a legitimate mode or `null`. There are no other possibilities. Had I used a `static final int` to define the modes, it would be possible to pass the constructor an illegal value for the `Mode` argument.

Two more symbolic constants, `STOPPED` and `EXPIRED`, are defined on lines 42 and 44. These are the "action command" Strings attached to the `ActionEvent` objects sent to registered listeners when the timer goes off. The action commands are outputs from the `Alarm` (as compared to inputs), the values are effectively constants, and a full-blown enumeration like `Mode` isn't required here.

To understand `Alarm`'s methods, we need to skip ahead a moment and talk about structure. Three classes are used here, shown in Figure 1. (This is a UML static-model diagram. See the Resources for more information if you've never seen one of these before.)

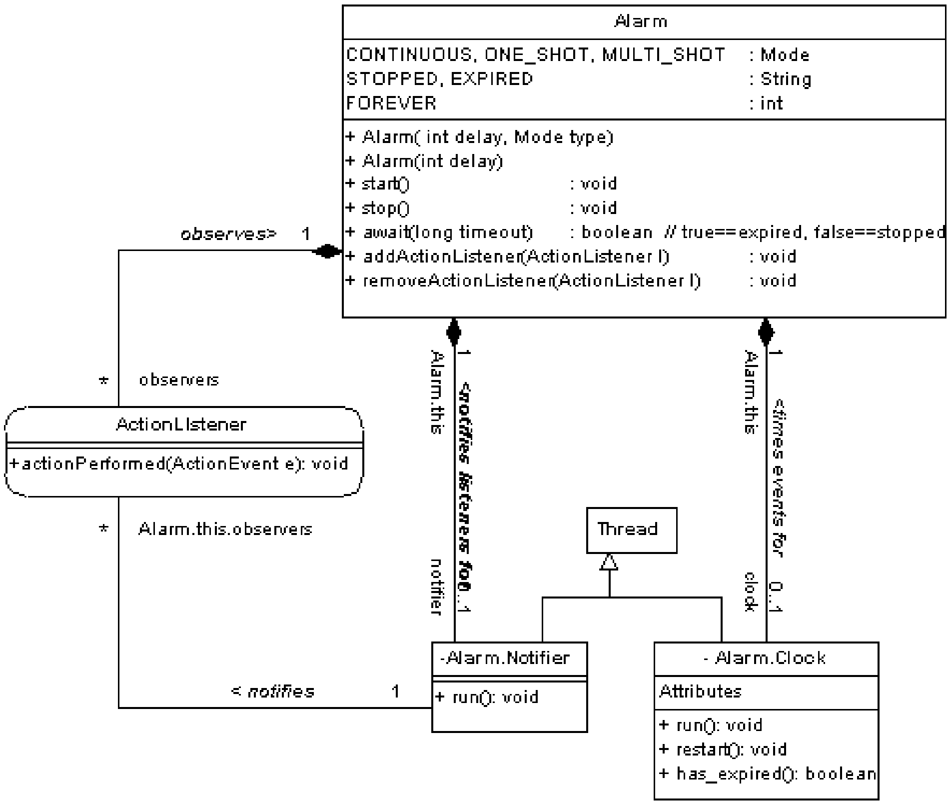

Figure 1: Static model for the Alarm class

What time is it?

The main class is the `Alarm` itself, and it's this class that the client waits on. The `Alarm` isn't actually a thread, however, though it uses two threads (`Clock` and `Notifier`) to get its work done. The `Clock` (Listing 4, line 156) is what's actually doing the timing. It's `run()` method (line 166) loops, sleeping for the amount of time specified in `delay` (line 23. When it wakes up, it sends a `notifyAll()` message to the `Alarm` object, which in turn causes `await()` (line 97) to be released from its `wait()` and return. The `Clock` object then either goes back to sleep, destroys itself (line 197), or suspends itself (line 185), depending on the operating mode.

There are a couple of reasons for using a second class for the timer thread (as compared to making the `Alarm` itself a `Thread`). The main reason is that occasionally it's convenient to wait on a timer that hasn't been started yet. For example, you might have a thread that performs some sort of timeout action that permanently waits on a `MULTI_SHOT Alarm` whose delay interval is the timeout time. The `Alarm` isn't fired up until the event is initiated, however. Using two objects lets me hold off on creating the timing thread until it's actually needed.

The second reason for the two-object approach is that it makes it much easier to restart the timer before it's expired. Let's say you have a "dead-man" `Alarm` (my term) that works like a "dead-man switch." The thread that's waiting on the timer wakes up only if nothing's happened in the last *N* minutes. You can use this sort of thing for automatic saves to file, bringing up screen savers, and so on. You can implement a dead man by creating an `Alarm` with an *N*-second interval, then firing off a thread that waits on the alarm. The main program sends a `start()` message to the `Alarm` every time it does something. The `Alarm` doesn't go off until *N* seconds have passed without anything having been done.

Implementing this sort of restart with a single class is difficult because there's no way to change the time associated with a `sleep()` call without breaking out of the sleep. Using two classes is easy, though. The `Alarm`'s `start()` method (line 59) simply interrupts the existing `Clock`, then creates a new one with the original delay. The `notifications_off` flag (line 159) is used by `run()` to make sure waiting threads aren't notified when a `Clock` is being killed during a reset operation.

There is one tricky synchronization issue in this code. The `Clock` removes all traces of itself as it shuts down by setting the `Alarm`'s `Clock` reference to null, as seen on line 197. Meanwhile, several methods of the `Alarm` itself use or modify `Clock`. All the relevant `Alarm` methods are synchronized, so there's nothing to worry about there. The `Clock` object's `run()` method must synchronize on the outer-class object with a `synchronized(Alarm.this)` before it can safely be modified on line 197. By the same token, the `Clock` has to be synchronized on the outer-class object when it notifies the waiting threads on line 180. However, the `Clock` *cannot* hold the monitor lock on the `Alarm` object while it's suspended, as on line 185. That's a classic nested-monitor lockout (discussed back in Part 2 of this series.) The only way to bump the `Clock` out of the wait is to call the `Alarm`'s `start()` method, which is synchronized on the same lock that's held by the dormant `Clock` thread. The solution is to release the outer-class object's lock while we're waiting on line 185. That's why I need two `synchronized(Alarm.this)` blocks.

Also note that I've used a `wait()` on line 185 rather than a (deprecated) `suspend()` call. The matching `notify()` is down on line 215. A `wait()`/`notify()` strategy can almost always replace calls to `suspend()` and `resume()`, without any of the monitor-related problems caused by the now-deprecated methods. I discussed these back in Part 2, but by way of reminder, `suspend()`, unlike `wait()`, does not give up the monitor's lock when it's called. This was the guy-goes-into-the-bathroom-and-falls-into-a-drug-induced-coma scenario. As a consequence, it's possible to get into deadlock scenarios if you use `suspend()` that you simply can't get into if you use `wait()`, which gives up the monitor's lock before it goes off to wait on the associated condition variable.

Also, note that the lock on the `Clock` itself (acquired by the `synchronized(this)` statement on line 175) is released by `wait()` when the thread is suspended on line 185 -- that's just the way `wait()` works. Consequently, the fact that restart() is synchronized is irrelevant. (In fact, it has to be synchronized to call `notify()`.)

The final `Clock`-related thing to point out is the test on line 196. This test is necessary because of the following scenario.

1. The clock expires, gets as far as line 194, and is preempted.

2. Another thread comes along and stops the clock. This in itself is harmless because the `stop()` method (Listing 4, line 78) just sets the `clock` reference to null and interrupts the `Clock`. Neither operation will cause difficulty.

3. Another thread comes along and sends the `Clock` a `start()` message (line 59), which creates a new clock and starts it up.

4. The original `Clock` thread now wakes up and executes line 197, setting the newly minted `Clock` reference to null.

The `if` statement on line 196 solves the delete-an-active-`Clock` problem by assuring that a given `Clock` object can remove only a reference to itself.

Hey you! Time to wake up!

The job of the second thread (`Notifier` on line 125) is to notify the listeners when the `Alarm` goes off. This thread is created when a listener is added to `addActionListener()` on line 110. The thread shuts down automatically when there are no observers to notify. (The `while` loop on line 131 uses `observers != null` as its termination condition.)

Getting the synchronization right is tricky here, too. The notification loop on line 131 must synchronize on the outer-class object (`Alarm.this`) in order to wait on that object. You have no idea how long it's going to take to get through the notifications, however, and you don't want to lock the `Alarm` during this period. Fortunately, the `AWTEventMulticaster` (which will be the subject of a future column) is designed with this problem in mind, and it can be used safely to notify listeners without locking the notifying object.

The test on line 142 is essential, because the observers might have been removed while we were waiting. It's possible, though, for run() to be preempted between lines 142 and 143. If `observers` itself was used in the test rather than the copy I made on line 138, the preempting thread could remove all the `Notifier` objects, setting the `observers` reference to null. The subsequent call to `actionPerformed()` would then throw a `NullPointerException`.

The main problem with my approach is that the thread could be preempted after the copy was made. The preempting thread could then remove listeners that will, nonetheless, be notified that the alarm goes off. This is a generic problem with all AWT listeners, by the way. If notifications are in progress when a listener is removed, it's still possible for that listener to be notified. You have to plan for this eventuality when you write the code. Observers added after the notifications are started are also not notified, but that's as it should be.

The next problem is creating the thread. The test in addActionListener() starts up a new notifier thread if one doesn't exist, but as usual, things are not as simple as they first appear. There are two problems: you can end up with too many `Notifier` objects or you can end up with no `Notifier` objects at all. The first problem is easily solved by synchronizing `addActionListener()`. Since only one thread at a time can be in the method, we don't have to worry about being preempted after the test but before the `new`. The second problem can occur when we're killing the thread. The simple strategy, of setting `notifier` to null as `run()` exits, doesn't work because `run()` could be preempted while notifications are in progress but before `notifier` is set to null. The preempting thread would assume a notifier exists (because we haven't set it to null yet), and not create a new notifier. Control would then transfer back to the notifier thread, which would then kill itself and set `notifier` null. The net effect is that no notification thread would exist.

I've solved the problem by setting `notifier` to null on line 139, while the notifier is still synchronized with the `Alarm`. Now if somebody comes along and adds an observer while notifications are in progress, we'll end up (briefly) with two threads. But the original one will safely go out of existence when it's done with the current batch of notifications.

Initialization is also tricky. The addActionListener() method must first add the notifier, then create the thread. Otherwise the test on line 131 will fail immediately. The `addActionListener()` method must also be synchronized so that we aren’t interrupted after creating the observer but in the middle of initialization. Also, we have to make a decision about whether or not to synchronize removeActionListener(). Consider the following scenario:

1. `addActionListener()` is called and starts up the thread. `run()` is preempted just before entering the `while` loop.

2. `removeActionListener()` (unsynchronized in this scenario) is called and removes a listener. It is preempted as `AwtEventMulticaster.remove()` is returning, after the listener has been removed but before the control is passed back to the caller. `AwtEventMulticaster.remove()` is thread-safe, so we don’t have to worry about it, at least.

3. `run()` reactivates, and the `while` loop is executed because `observers` has not been modified yet, even though there are no observers.

At this juncture, the notification thread exists, but it will harmlessly do nothing (and terminate itself) when the `Alarm` goes off. If `removeActionListener()` had been synchronized, the superfluous notifier thread would still be created, but this time we’d get all the way through to the `synchronized` statement that guards the `wait()` on line 134 before blocking. The listener would now be removed, and we’re back where we were before.

Given that there’s no advantage to synchronization, I decided not to synchronize `removeActionListener()`. No point in incurring the extra overhead unnecessarily.

This is not a crash-and-burn sort of failure, because eventually the timer will go off, no `Notifier` objects will be found, and the thread will terminate. Nonetheless, I don’t like threads to be kicking around unnecessarily.

Listing 4. (/src/com/holub/asynch/Alarm.java) Roll Your Own Timer

```
001 //----------------------------------------------------------------------
002 // This code (c) 1998 Allen I. Holub. All Rights Reserved.
003 //----------------------------------------------------------------------
004 // This code may not be distributed by yourself except in binary form,
005 // incorporated into a java .class file. You may use this code freely
006 // for personal purposes, but you may not incorporate it into any
007 // commercial product without express permission of Allen I. Holub in writing.
008 //
009 // Sources code for this class can be found in the "Goodies" section of
010 // <http://www.holub.com>.
011 //----------------------------------------------------------------------
012
013 package com.holub.asynch;
014
015 import java.util.*;          // Need Date for testing
016 import java.awt.event.*;     // For ActionListener, etc.
017 import java.awt.*;           // For ActionListener, etc.
018
019
```

/**

An interval timer. Once you "start" it, it runs for a predetermined amount of time. The timer runs on its own relat high-priority thread while one or more other threads block, waiting for the timer to "expire." Three sorts of timer

A "one-shot" timer
: runs once, then expires. May be started again, but generally isn't.

A "multi-shot" timer
: works like a "one-shot" but expects to be started again so is more efficient in this situation. Must be stopp

A "continuous" (oscillator-style) timer
: runs continuously. Starts up again automatically when it expires. Must be stopped explicitly.

All timers may be restarted (have the time interval set back to the original value) while they are running. **Warnin**

1. It's easy to forget to stop() a multi-shot or continuous timer, and neglecting to do so can create a memory consequence. (The VM will keep the `Thread` object alive but suspended, even if there are no external refe finalizer is provided to throw an exception in this situation, but since the finalizer may not be called, that's solution.
2. Calling wait() on a timer is possible, but it only works correctly for CONTINUOUS timers. The await() m correctly for all types of timers, and is easier to use since you don't have to synchronize first. The only tin raw wait() makes sense if if you're interested in the InterruptedException, which is silently absorbed by a

```
**/

public class Alarm
{                                           // A compiler bug (1.1x, 1.2rc1)
    private int             delay;          // permit blank finals.
    private Alarm.Clock     clock       = null;
    private boolean         is_stopped  = false;

    public static class Mode { private Mode(){} }

    /****************************************************************
```

CONTINUOUS alarms run continuosly (until stopped), sending out notification messages and releasing wa a regular interval.

```
     */

    public static final Mode CONTINUOUS  = new Mode();

    /****************************************************************
```

ONE_SHOT alarms run once, then stop automatically. [You do not have to call stop().] Can be started agai calling start(), but a new internal thread must be created to do so.

```
     */

    public static final Mode ONE_SHOT    = new Mode();

    /****************************************************************
```

MULTI_SHOT alarms work just like ONE_SHOT alarms, but they can be restarted more efficiently. (The a MULTI_SHOT alarm is not destroyed until you stop() the alarm.) Use a MULTI_SHOT when you know one-shot will be restarted at regular intervals and don't mind having an extra thread kicking around in a sus

```
     */

    public static final Mode MULTI_SHOT  = new Mode();

    private Mode type;

    //------------------------------------------------------------------

    private ActionListener  observers   = null;
    private Thread          notifier;
```

```
/**************************************************************
```

"Action command" sent to the ActionListeners when the timer has been stopped manually.

```
 */

public static final String STOPPED = "stopped";

/**************************************************************
```

"Action command" sent to the ActionListeners when the timer has expired or "ticked."

```
 */

public static final String EXPIRED = "expired";

/**************************************************************
```

Create a timer that expires after the indicated delay in milliseconds. Generally, timers must run at high pric the waiting threads from starting up immediately on being notified, thereby messing up the timing. If a higl is waiting on a recurring timer, however, the next "tick" could be delayed by the amount of time the high-p spends doing whatever it does after being notified. Alarms run at the highest priority permitted by the threa which they are a member, but spend most of their time sleeping.

@param ***delay*** Time to expiration (nominal) in milliseconds.

@param ***type*** One of the following symbolic constants (see):

`CONTINUOUS`	The timer runs continuously Must call stop() when you're done with it.
`ONE_SHOT`	The timer runs once, then stops automatically. You do not have to call stop
`MULTI_SHOT`	Like ONE_SHOT, but can be restarted more efficiently.

If this argument is `null`, `CONTINUOUS` is used.

```
 **/

public Alarm( int delay, Mode type )
{   this.delay  = delay;
    this.type   = (type == null) ? CONTINUOUS : type ;
}

/**************************************************************
```

Make a continuous timer.

```
 */

public Alarm( int delay )
{   this( delay, CONTINUOUS );
}

/**************************************************************
```

Start up a timer or restart an expired timer. If the timer is running, it is set back to the original count withou of the threads that are waiting for it to expire. (For example, if you start up a 10-second timer and then rest: seconds, the waiting threads won't be notified until 10 seconds after the restart---15 seconds after the origi Starting a running timer causes a new thread to be created.

```
 **/

public synchronized void start()
{   if( clock != null )
```

```
        {   if( type == MULTI_SHOT && clock.has_expired() )
            {
                clock.restart();
                return;
            }
            else
            {   clock.no_notifications();
                clock.interrupt();
            }
        }

        clock = new Clock();
        clock.start();
        is_stopped = false;
    }

    /*****************************************************************
     | Stops the current timer abruptly, releasing all waiting threads. The timer can be restarted by calling start().
       way for a thread to determine if it was notified as the result of a stop() or a normal expiration.
     **/


    public synchronized void stop()
    {   if( clock != null )
            clock.interrupt();

        clock      = null;
        is_stopped = true;
        notifyAll();
    }

    /*****************************************************************
     | It is a bug not to stop() a CONTINUOUS or MULTI_SHOT timer when you're done with it. This finalizer
       the bug by throwing an Error() if the timer that's being destroyed is still running.
     **/


    public void finalize()
    {   if( clock != null )
            throw new Error("Alarm was not stopped before being destroyed");
    }

    /*****************************************************************
     | A long time (roughly 292,271,023 years) that you can use for the timeout value in await():{@link await()
     **/


    public static final long FOREVER = Long.MAX_VALUE;

    /*****************************************************************
     | Wait for the timer to expire. Returns immediately in the case of an expired ONE_SHOT or MULTI_SHOT
       until the timer expires in all other situations (including any sort of timer that has not yet been started).
       @return false if the method returned because the timer was stopped, true if the timer simply expired.
       @see FOREVER
     **/
```

```
    public synchronized boolean await( long timeout )
    {
        if( clock == null || !clock.has_expired() )
        {   try                                 { wait( timeout );  }
            catch( InterruptedException e ) { /*do nothing*/    }
        }
        return !is_stopped;
    }

    /*****************************************************************
      Same as await(Alarm.FOREVER)
     **/

    public boolean await(){ return await( FOREVER ); }

    /*****************************************************************
      Add a listener that will be notified the next time the Alarm goes off. The listeners are notified on a thread t
      for that purpose, rather than being notified from the timer thread. This way the time spent doing notificatio
      the time interval used by the timer. The "action command" in the ActionEvent object will be either the Stri
      "expired" (which are also defined in the symbolic constants {@link Alarm.STOPPED} and {@link Alarm.
      depending on whether this notification occured because the timer was stopped manually, or because it expi
      normal way.
     */

    public synchronized void addActionListener(ActionListener l)
    {   observers = AWTEventMulticaster.add(observers, l);
        if( notifier == null )
        {   notifier = new Alarm.Notifier();
            notifier.setDaemon( true );
            notifier.start();
        }
    }

    /*****************************************************************
      Remove a listener
     */


    public void removeActionListener(ActionListener l)
    {   observers = AWTEventMulticaster.remove(observers, l);
    }

    private class Notifier extends Thread
    {
        public void run()
        {
            ActionListener copy;

            while( observers != null )
            {   synchronized( Alarm.this )
                {   try
                    {   Alarm.this.wait();
                    }
```

```
                    catch(InterruptedException e){/*ignore*/}

                    if( (copy = observers) == null)
                        notifier = null;
            }

                if( copy != null )
                    copy.actionPerformed(
                            new ActionEvent(this,0,
                                    is_stopped ? STOPPED : EXPIRED));
            }

            // can't set notifier=null here
        }
    }

    //================================================================
    // Support classes:
    //================================================================

    private final class Clock extends Thread
    {
        private boolean expired             = false;    // continuous timers c
        private boolean notifications_off   = false;

        Clock()
        {   setPriority( getThreadGroup().getMaxPriority() );
        }

        public void run()
        {
            while( !isInterrupted() )
            {   try
                {   sleep(delay); // release the monitor while sleeping

                    if( isInterrupted() )   // don't notify waiting threads if
                        break;              // we've been stopped by the
                                            // Alarm object.
                    synchronized( this )
                    {
                        expired = true;
                        if( !notifications_off )
                        {   synchronized( Alarm.this )
                            {   Alarm.this.notifyAll();
                            }
                        }

                        if( type == MULTI_SHOT )
                        {   wait();                     // suspend
                        }
                        else if( type == ONE_SHOT )
                        {
                            // Set the outer-class reference to the
                            // current clock to null so that the
                            // memory can be reclaimed. Note that
                            // we're still synchronized on the outer-
                            // class object.

                            synchronized( Alarm.this)
                            {   if( Alarm.this.clock == this )
                                    Alarm.this.clock = null;
```

```
                            }
                            break;
                        }
                    }
                }
                catch(InterruptedException e) // don’t notify the waiting
                {   break;                    // threads because an
                }                             // interrupt is used to stop
            }                                 // the timer.
        }

        public void no_notifications()
        {   notifications_off = true;
        }

        public synchronized void restart()
        {   expired = false;
            notify();                       // resume
        }

        public boolean has_expired()                // CONTINUOUS
        {   return (type != CONTINUOUS) && expired; // timers never
        }                                           // expire.
    };

    //=================================================================
    // Unit test:
    //=================================================================
    static public class Test
    {
        public static void main( String[] args ) throws Exception
        {
            // A recurring timer, runs until it is stoped manually.

            Alarm clock = new Alarm(1000, Alarm.CONTINUOUS );
            clock.start();

            System.out.println("Print time 3 times at 1-second intervals");

            for( int i = 3; --i>= 0; )
            {
                System.out.println( new Date().toString() );
                clock.await( Alarm.FOREVER );
            }

            clock.stop();   // It is essential to stop the timer manually.
                        // Otherwise, the memory for it might never be
                        // reclaimed.

            System.out.println("\nOne-shot:\n");

            // A One-shot. Fire it manually. You don’t need to stop()
            // it explicitly---it automatically frees up all threads
            // when it expires.

            clock = new Alarm(1000, Alarm.ONE_SHOT);
            clock.start();
            for( int i = 3; --i>= 0; )
            {
                System.out.println( new Date().toString() + "\r" );
                clock.await( Alarm.FOREVER );
```

```
                clock.start();
            }

            System.out.println("\n Multi-shot:\n");

            // A Multi-shot is much like a one-shot. Fire it manually,
            // but you must stop() it explicitly. The main difference
            // is that a MULTI_SHOT timer doesn’t recreate the timer
            // thread when it’s restarted. The one-shot timer creates
            // the thread anew every time it’s started.

            clock = new Alarm(1000, Alarm.MULTI_SHOT);
            clock.start();
            for( int i = 3; --i>= 0; )
            {
                System.out.println( new Date().toString() + "\r" );
                clock.await( Alarm.FOREVER );
                clock.start();
            }
            clock.stop();

            System.out.println( "\n Notifier\n" );

            clock = new Alarm( 1000 );  // 1-second continuous timer
            clock.addActionListener
            (   new ActionListener()
                {   public void actionPerformed( ActionEvent e )
                    {   System.out.println( new Date().toString()
                                            + e.getActionCommand() );
                    }
                }
            );
            clock.start();

            System.out.println("Sleeping");
            Thread.currentThread().sleep( 5000 );
            System.out.println("Waking");

            clock.stop();
            System.exit(0);
        }
    }
}

```

Happy trails

Frankly, I’m not completely convinced that all this will indeed work correctly in all situations. I’ve stared at the code, made diagrams, and analyzed it up the wazoo, but if past experience is any indication, I’ve probably still missed something. Welcome to the world of thread programming. If you find a flaw in my reasoning (and some of you probably will -- sigh), please send me e-mail.

So that’s it for this month. A `Timer` is a useful thing to have around, not just for animation loops, but for any application that needs to do things at regular intervals or during idle time. Between the new Swing `Timer` and my own `Alarm`, all the applications I can think of are covered. The synchronization issues in this code are hairier than usual, but it does serve as a good example of threading in the real world.

Next month, I'll continue with even more threading stuff by delving into implementations of the Observer and Singleton design patterns that work in multithreaded environments. I'll look at the inner workings of the `AWTEventMulticaster` class and other ways of sending notifications efficiently in a thread-safe way. I'll also look at how to create a Singleton efficiently, and how to work around a 1.1 VM bug that makes it difficult to produce Singletons. ■

Also this month in JavaWorld
Go

Resources

- All the real code discussed in this article (the stuff in the `com.holub.asynch` package) is available in the "Goodies" section on my Web site, http://www.holub.com. The version on the Web site should be considered the definitive version -- at least it corrects any bugs I know about.
- An article describing Swing's support for threads is at http://java.sun.com/products/jfc/tsc/archive/tech_topics_arch/threads/threads.html
- You can find documentation for the Swing `Timer` class in the JDK docs and at http://java.sun.com/products/jdk/1.2/docs/api/javax/swing/Timer.html
- UML (the Unified Modeling Language) amalgamates the notations of Grady Booch, James Rumbaugh, and Ivar Jacobson. The following three references are recommended:
  - Martin Fowler and Kendall Scott's *UML Distilled: Applying the the Object Modeling Language* (Reading: Addison Wesley, 1997 [ISBN: 0-201-32563-2]) is a good, quick
  - introduction to UML for those who already know an object-oriented notation: http://cseng.awl.com/bookdetail.qry?ISBN=0-201-32563-2&ptype=0
  - A more in-depth introduction to UML is Grady Booch, James Rumbaugh, Ivar Jacobson's *The Unified Modeling Language User Guide* (Reading: Addison Wesley, 1999 [ISBN: 0-201-57168-4]): http://cseng.aw.com/bookdetail.qry?ISBN=0-201-57168-4&ptype=0
  - The actual technical documentation for UML is truly turgid prose, useful if you're
  - suffering from insomnia, but it is accurate. Because the UML docs are distributed on a commercial site, I feel compelled to point out that even though Booch, et al., are movers and shakers in the OO world, their OO-design tool -- Rational Rose -- is not wonderful. It's UI is miserable, it doesn't do nearly enough for you in the code-generation department, it's way too expensive for the work it does, and it's much to tightly coupled to Microsoft technology for me to feel comfortable entrusting a real project to it. If you do download the trial version of Rose from the Rational site, bear in mind that there are better alternatives: http://www.rational.com/uml/documentation.html
- Design patterns like Observer are discussed in the so-called Gang of Four (G4) book: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides's *Design Patterns: Elements of Reusable Object-Oriented Software* (Reading: Addison Wesley, 1995). This is a stunningly good cookbook, documenting many patterns of object/class interaction that recur regularly in good object-oriented designs. All experienced object-oriented designers will recognize these patterns immediately from their own work, and folks who are learning OO design can save themselves a lot of hard work by using the patterns described here from the beginning. Though any cookbook approach to design is ultimately limiting (to use a cookbook, you have to know how to cook), this

book is invaluable to people learning the design process or wanting a common vocabulary to describe parts of their designs to other designers. It can save you an enormous amount of time.
http://www.awl.com/cseng/swpatterns

- A good complement to the G4 book is John Vlissides's *Pattern Hatching: Design Patterns Applied* (Reading: Addison Wesley, 1998 [ISBN 0-201-43293-5]). The book talks about a few problems that didn't occur to the authors while writing the first book, and also introduces a new pattern.
  http://www.awl.com/cseng/swpatterns
- The Sun Web site has a page on multithreading in its "Technical Articles" section.
  http://developer.javasoft.com/developer/technicalArticles/#thread
- Prashant Jain and Douglas C. Schmidt have a good article contrasting C++ to Java that discusses many of the thread-related problems inherent in the language. The article can be found at
  http://www.cs.wustl.edu/%7Eschmidt/C++2java.html
- Doug Lea has a bunch of Mutex and Condition-variable classes in his `util.concurrent` package. See
  http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html
- For a great in-depth look at multithreading in general and the implementation of multithreading both in and with Java in particular, this is a must. It's required reading if you're using threads heavily: Doug Lea, *Concurrent Programming in Java: Design Principles and Patterns* (Reading: Addison Wesley, 1997)
  http://java.sun.com/docs/books/cp/
- For an intro-level book on Java threading that is less technical but more readable than Lea, see: Scott Oaks and Henry Wong, *Java Threads* (Sebastopol, CA: O'Reilly, 1997)
  http://www.oreilly.com/catalog/jthreads/
- This book is good for the general subject of multithreading but doesn't have a Java slant: Bill Lewis and Daniel J. Berg, *Threads Primer: A Guide to Multithreaded Programming* (Englewood Cliffs: Prentice Hall/SunSoft Press, ISBN 0-13-443698-9)
  http://www.sun.com/books/catalog/Lewis/Lewis.html
- Doug Schmidt's "Ace" Framework is a good, though complex, attempt at a truly platform-independent threading system.
  http://www.cs.wustl.edu/~schmidt/">http://www.cs.wustl.edu/~schmidt/">http://www.cs.wustl.edu

About the author
Allen Holub has been working in the computer industry since 1979. He is widely published in magazines (*Dr. Dobb's Journal, Programmers Journal, Byte*, and *MSJ*, among others). He has seven books to his credit, and is currently working on an eighth that will present the complete sources for a Java compiler written in Java.

After eight years as a C++ programmer, Allen abandoned C++ for Java in early 1996. He now looks at C++ as a bad dream, the memory of which is mercifully fading. He's been teaching programming (first C, then C++ and MFC, now OO-Design and Java) both on his own and for the University of California Berkeley Extension since 1982. Allen offers both public classes and in-house training in Java and Object-Oriented Design topics. He also does object-oriented design consulting and contract Java programming. Get information, and contact Allen, via his Web site http://www.holub.com. Reach Allen at allen.holub@javaworld.com.

What did you think of this article?

-Very worth reading	-Too long	-Too technical
-Worth reading	-Just right	-Just right
-Not worth reading	-Too short	-Not technical enough

Comments:

Name:

Email:

Company Name: Send data