Java Toolbox by Allen Holub

Programming Java threads in the real world, Part 4

Condition variables and counting semaphores -- filling in a few chinks in Java's threading model

Summary
This column continues where last month's column left off, presenting a few more implementations of classes that are useful when you're doing multithreading in Java. The two classes I'll discuss provide capabilities difficult to get using Java's threading primitives alone: a *condition variable* adds to `wait` the ability to not wait when the condition you're waiting for has already taken place; and a *counting semaphore* lets you control a pool of resources without sucking up machine cycles in polling loops. This month's column also discusses a few minor fixes to the code presented last month. *(6,250 words)*

This month's column adds a few more classes to the threading arsenal we started building in last month's **Java Toolbox** column. This time I'll discuss the following:

1. Roll-your-own versions of the *condition variable,* which replaces `wait()` and `notify()` in some situations

2. Djikstra's "counting" semaphore, which is used to manage pools of resources

(A *semaphore* is any of several mechanisms used to synchronize and communicate between threads. Think of the "semaphore" as in the flags that two boy scouts use to talk to each other from a distance -- different flag positions represent different letters of the alphabet. Napoleon's army used the vanes of windmills on mountain tops to send semaphore messages great distances very quickly. The mutex discussed last month, since it's a communications mechanism, is also a "semaphore.")

The condition variable can often be simulated using Java alone -- and I'll show you how -- while the counting semaphore can't.

you now -- while the counting semaphore can't.

Before I start, though, I'd like to go over a few loose ends from last month and fix a few bugs.

Oops! Could my code have bugs in it?

Before leaping into this month's meat (now there's a metaphor you can sink your teeth into), let's look at a few loose ends that either caught my eye (or were brought to my attention) after last month's article went live.

During one of my book's "peer reviews," an academic reviewer once took exception to the sentence "if you're anything like me, you'll forget to ... so you should write your code to do it automatically." His comment to me was: "I would never admit that in print." This guy was (and as far as I know, still is) a tenured professor at an Ivy League university, and I suppose his comment was correct in a literal sense: since he never had written any actual code, he never had any bugs to admit. I might as well say it up front, though: my code contains an occasional bug (gasp). Consequently, I expect an "Oops" section or its moral equivalent to become a regular feature of this column. There's nothing like having 100,000 people look over your code for problems to emerge and be highlighted.

Joe Bowbeer pointed out (quite correctly):

> Why not advise the use of `try`/`finally` to prevent an exception from gunking up the works?
>
> ```
> mutex.acquire();
> try
> {    doit();
> }
> finally
> {    mutex.release();
> }
> ```
>
> I prefer the [above] form of `try`/`finally` because it separates the exceptions that might occur in changing the state (`acquire`) from the exceptions that might occur when working in the new state (`doit`).
>
> The other (more intuitive?) form is
>
> ```
> try
> {    mutex.acquire();
>      doit();
> }
> finally
> {    mutex.release();
> }
> ```
>
> This requires more programming in `release()` to ensure that mutex is in a consistent state: if `release()` is called after an exception in `acquire()`, the mutex may not have been acquired, or [may have been] half-acquired, etc.

I should add that Joe's last point is important in the case of last month's `Mutex` class. The `acquire_without_blocking()` method, where the actual acquisition occurs, doesn't throw any exceptions at awkward times. The only exception that can be thrown from `acquire()`, in fact, is an

`InterruptedException`, thrown if a timeout is specified and the waiting thread is interrupted. This operation does not leave the mutex in an unstable state, however.

Be that as it may, while looking at my code to make sure Joe hadn't found a bug, I found a bug myself. (I'll show you the code in a moment, but let's discuss the problems in it first.) The `acquire()` method was using a standard spin lock while waiting to acquire the mutex, but this strategy is appropriate only when the timeout is infinitely large. I've fixed the problem by making the loop terminate for timeout values less than `Integer.MAX_VALUE` (the value I use for "forever"). It continues to use a spin lock in "forever" cases.

While I was at it, I also decided to have `acquire()` indicate whether or not it had timed out. The choices here are the usual ones: a return value or an exception toss. I opted for the latter because a timeout typically is an error condition, and I didn't want to clutter up the code with unnecessary tests for `false` return values.

I modified the definition for the `Semaphore` interface to incorporate this new exception:

```
package com.holub.asynch;

interface Semaphore
{
    int  id     ();
    void acquire(long timeout)  throws InterruptedException,
                                       Timed_out;
    void release();

    public static class Timed_out extends java.lang.RuntimeException
    {   Timed_out(){ super("Timed out while waiting to acquire semaphore"); };
    }
}
```

Note that `Semaphore.Timed_out` is a `RuntimeException`, so you don't have to catch it if the timeout is a fatal error (often the case).

The new (and this time, I hope, correct) version of `acquire()` now looks like this:

```
public synchronized void acquire( long timeout ) throws InterruptedException
{
    if( timeout == 0 )                         // don't wait at all
    {   acquire_without_blocking();
    }
    else if( timeout == Long.MAX_VALUE )       // wait forever
    {   while( !acquire_without_blocking() )
            this.wait( timeout );
    }
    else                                       // wait limited by timeout
    {   if( !acquire_without_blocking() )
        {   this.wait( timeout );
            if( !acquire_without_blocking() )
                throw new Semaphore.Timed_out();
        }
    }
}
```

Finally, in last month's column, I inadvertently used an outdated version of the `Lock_manager`'s comparator class. (It threw a `Not_Comparable` exception -- an artifact of my own *sort* implementation, which I abandoned when Java added an official one.) Anyway, the comparator class should look like this:

```
private static class Lock_comparator implements Comparator
{   public int compare(Object a, Object b)
    {   return ((Semaphore)a).id() - ((Semaphore)b).id();
    }
    public boolean equals(Object obj)
    {   return obj instanceof Lock_comparator;
    }
}
```

I've modified the code in the "Goodies" section of my Web site (see Resources) to incorporate all these changes.

Now, on to the meat of *this month's* article.

Condition variables

Forging ahead -- to boldly split infinitives no one has split before:

I've brought up "condition variables" before in the context of `wait()` and `notify()`. The central concept is that a thread will wait until some condition becomes true. For example, a thread may need to wait for somebody to push a button before proceeding with an action, or a thread may wait for something to appear in an empty queue (for the queue-not-empty condition to become true).

Using a condition variable to wait for events

The following code illustrates a classic problem that is easily solved with a condition variable: How do I wait for an event to occur without wasting machine cycles in a polling loop? The code sets up a simple `TextField` and an `ActionListener` that's notified when the user types the Enter key:

```
class Some_class extends Frame
{
    TextField input = new TextField();
    String    entered = "";

    public The_wrong_way()
    {   input.addActionListener
        (   new ActionListener()
            {   public void actionPerformed( ActionEvent e )
                {   entered = input.getText();
                }
            }
        );

        add(input);
        pack();
        show();
    }

    String read_line(){ return entered; }
    //...
}
```

So far, so good, but let's look at the situation in more detail.

When you display the `Frame`, AWT fires up a thread that monitors events coming in from the operating system, including key-press events. When the Enter key is pressed, for example, the AWT thread gets the key-press event from the OS and, in turn, calls the listener's `actionPerformed()` method. The "actionPerformed()" messages are coming in asynchronously from the AWT event-loop thread. Put another way: the "actionPerformed()" message is actually running on that AWT thread.

Meanwhile, a user thread (as differentiated from the AWT thread) calls `read_line()` to find out what the user has typed. The problem is that both the AWT and the user thread can access the `entered` field simultaneously -- a classic race condition. The second thread could call `read_line()` while the AWT thread is in the middle of `ActionPerformed()` and end up reading garbage.

Solve this first problem with synchronization:

```
class Some_class extends Frame
{
    TextField input = new TextField();
    String    entered = "";

    public The_wrong_way()
    {   input.addActionListener
        (   new ActionListener()
            {   public void actionPerformed( ActionEvent e )
                {   synchronized( Some_class.this )
                    {    entered = input.getText();
                    }
                }
            }
        );

        add(input);
        pack();
        show();
    }

    String synchronized read_line(){ return entered; }
    //...
}
```

Note that the inner-class method has to synchronize explicitly on the outer-class object. Simply synchronizing `actionPerformed()` doesn't work because you'll be synchronizing on the monitor of the anonymous inner-class object, and the field you want to guard is in the outer-class object.

Moving on, our user thread needs to know *when* an entire line has been typed to be sure that `read_line()` will return a complete line of input, but (and this is the big *but*), there's no direct communication between the two threads involved in this transaction. The code running on the AWT thread (`actionPerformed()`) doesn't tell the user thread that an entire-line-has-been-typed event has occurred.

So how does the caller of `read_line()` know the string has changed? It could sit in a tight polling loop calling `read_line()` and checking the current return value against the previously returned value, but that's an awful lot of machine cycles wasted on doing nothing useful.

Send in the cavalry

So what's one way for two threads to communicate with each other? (That's a rhetorical question.) Use a semaphore (think Napoleon, flags, mountain tops). To the rescue comes the semaphore known as a condition variable. To rehash from previous months' material: the basic notion of a condition variable is that some thread waits (is suspended) on the condition variable until the condition it represents becomes true. Every Java object has a condition variable associated with it -- in the same way it has the mutex used to guard the monitor. You wait for the condition to become true by calling `wait()`, and you set the condition to true by calling `notify()`. (The `notify()` call doesn't work in quite this way; I'll talk more about this in a moment.) It's easy enough to do a roll-your-own condition variable that solves the current thread-communication problem by using `wait()` and `notify()`.

Listing 1 demonstrates how to do this. A condition variable called `text_has_been_entered` is declared up at the top of the class definition. (We're going to wait for the text-has-been-entered condition to become true.) The `actionPerformed()` method doesn't read the text at all; rather, it simply notifies the condition variable, setting the condition to true. Note that Java requires you to be in the monitor for an object before you can call `wait()` or `notify()` on that object, so the `synchronized(text_has_been_entered)` statement is mandatory. (You must be holding the lock associated with the object, by being in either a `synchronized` function of that object's class or a standalone `synchronized` statement whose argument is the object on which you're synchronizing.)

The `synchronized(text_has_been_entered)` statement on line 15 is mandatory, since entering the `synchronized` block puts us into the monitor of the object referenced by `text_has_been_entered`.

Meanwhile, down in `read_line()` on line 33, the thread that calls `read_line()` is waiting for the condition to become true. When this happens, the new text value is read and returned. The `read_line()` method is itself synchronized so that two threads can't attempt to read the same line simultaneously. It's now possible to have a simple loop like the one in `main()`

```
while( (input = source.read_line()) != null )
System.out.println("Got: " + input );
```

which blocks until a new input line arrives.

Listing 1: Using wait() and notify() for a condition variable

```
01  import java.awt.*;
02  import java.awt.event.*;
03
04  public class Input_source extends Frame
05  {
06      int[]   text_has_been_entered = new int[1]; // The condition variable
07
08      TextField input = new TextField();
09
10      public Input_source()
11      {
12          input.addActionListener
13          (   new ActionListener()
14              {   public void actionPerformed( ActionEvent e )
15                  {   synchronized( text_has_been_entered )
16                      {   text_has_been_entered.notify();  // set the condition
17                      }
18                  }
```

```
            }
        );

        add(input);
        pack();
        show();
    }

    /** A blocking function that works like readLine(), but gets its
     *  text from the current window’s text area. The function doesn’t
     *  return until somebody types a line of text, whereupon it returns
     *  the line. Returns null if the user types an empty line.
     */

    synchronized String read_line() throws InterruptedException
    {   synchronized( text_has_been_entered )
        {   text_has_been_entered.wait(); // wait for the condition to become
        }

        String entered = input.getText();
        input.setText("");
        return (entered.length() == 0) ? null : entered;

    }


    static public void main( String[] args ) throws Exception
    {   Input_source source = new Input_source();
        String input;

        while( (input = source.read_line()) != null )
            System.out.println("Got: " + input );

        System.exit(0);         // kill the AWT Thread on exit
    }
}
```

But Wellington is coming

A subtle problem occurs when using Java’s built-in condition variable, however. What if you call `read_line()` just after the value has changed rather than before? You’ll just wait until the value is changed again, missing the first value entirely. The problem is that the built-in condition variable doesn’t really have a notion of state associated with it. What we really need, to solve the current problem, is a true condition variable -- one that blocks only if we try to `wait()` when the condition is false, and that doesn’t block at all if the condition is true.

Listing 2 shows a simple implementation of such a beast. There’s not much to it -- the vast majority of the file is comments. The main thing is the `_is_true` flag declared at the top of the class. The `_is_true` flag stores the state of the condition variable. You can set the condition to true or false by calling `set_true()` or `set_false()`. You can test the current state without blocking by calling `is_true()` or `is_false()`. You can block, waiting for the condition to become true by calling `wait_for_true()`, which doesn’t block at all if the condition happens to be true when you call it.

Listing 2: A condition-variable implementation

```
package com.holub.asynch;

//-------------------------------------------------------------------------
// This code (c) 1998 Allen I. Holub. All rights reserved.
//-------------------------------------------------------------------------
// This code may not be distributed by yourself except in binary form,
// incorporated into a Java .class file. You may use this code freely
// for personal purposes, but you may not incorporate it into any
// commercial product without express permission of Allen I. Holub in writing
//-------------------------------------------------------------------------

/**
 *  This class implements a simple "condition variable." The notion
 *  is that a thread waits for some condition to become true.
 *  If the condition is false, then no wait occurs.
 *
 *  Be very careful of nested-monitor-lockout here:
 * <PRE>
 *   class lockout
 *   {  Condition godot = new Condition(false);
 *
 *      synchronized void f()
 *      {
 *          some_code();
 *          godot.wait_for_true();
 *      }
 *
 *      synchronized void set() // Deadlock if another thread is in f()
 *      {   godot.set_true();
 *      }
 *   }
 * </PRE>
 *  You enter f(), locking the monitor, then block waiting for the
 *  condition to become true. Note that you have not released the
 *  monitor for the "lockout" object. [The only way to set godot true
 *  is to call set(), but you'll block on entry to set() because
 *  the original caller to f() has the monitor containing "lockout"
 *  object.]
 *  <p>Solve the problem by releasing the monitor before waiting:
 * <PRE>
 *   class okay
 *   {  Condition godot = new Condition(false);
 *
 *      void f()
 *      {   synchronized( this )
 *          {   some_code();
 *          }
 *          godot.wait_for_true();  // Move the wait outside the monitor
 *      }
 *
 *      synchronized void set()
 *      {   godot.set_true();
 *      }
 *   }
 * </PRE>
 * or by not synchronizing the `set()` method:
 * <PRE>
 *   class okay
 *   {  Condition godot = new Condition(false);
 *
 *      synchronized void f()
```

```
 *      {   some_code();
 *          godot.wait_for_true();
 *      }
 *
 *      void set()              // Remove the synchronized statement
 *      {   godot.set_true();
 *      }
 *  }
 * </PRE>
 * The normal wait()/notify() mechanism doesn't have this problem since
 * wait() releases the monitor, but you can't always use wait()/notify().
 */


public class Condition
{
    private boolean _is_true;

    /** Create a new condition variable in a known state.
     */
    public Condition( boolean is_true ){ _is_true = is_true; }

    /** See if the condition variable is true (without releasing).
     */
    public synchronized boolean is_true()  { return _is_true; }

    /** Set the condition to false. Waiting threads are not affected.
     */
    public synchronized void set_false(){ _is_true = false; }

    /** Set the condition to true. Waiting threads are not released.
     */
    public synchronized void set_true() { _is_true = true; notifyAll(); }

    /** Release all waiting threads without setting the condition true
     */
    public synchronized void release_all(){ notifyAll(); }

    /** Release one waiting thread without setting the condition true
     */
    public synchronized void release_one(){ notify(); }

    /** Wait for the condition to become true.
     *  @param timeout Timeout in milliseconds
     */
    public synchronized void wait_for_true( long timeout )
                                            throws InterruptedException
    {   if( !_is_true )
            wait( timeout );
    }

    /** Wait (potentially forever) for the condition to become true.
     */
    public synchronized void wait_for_true() throws InterruptedException
    {   if( !_is_true )
            wait();
    }
}
```

Listing 3 below is basically Listing 1 rewritten to use a `Condition` object. Now, a call to `read_line()` after the user enters the text works just fine because the condition will be in the true state, and `wait_for_true()` won't block. Notice that `read_line()` has to explicitly set the condition back to false after it has read the line.

There are still a few problems that have to be fixed to make this all work in the real world. For example, there's no way for a program to know if the user overtypes a string when nobody fetches the original string before it's overwritten. Input strings should be queued up as they come in and `read_line()` should return a string from the queue if there is one, blocking only if the queue is empty. Listing 3 serves to illustrate the problem at hand without addressing these other issues.

Listing 3: Using a Condition object

```
import java.awt.*;
import java.awt.event.*;
import com.holub.asynch.Condition;

public class Input_source_fixed extends Frame
{
    Condition   text_has_been_entered = new Condition(false); // Initial condi

    TextField input = new TextField();

    public Input_source_fixed()
    {
        input.addActionListener
        (   new ActionListener()
            {   public void actionPerformed( ActionEvent e )
                {   text_has_been_entered.set_true();  // set the condition tr
                }
            }
        );

        add(input);
        pack();
        show();
    }

    /** A blocking function that works like readLine(), but gets its
     *  text from the current window's text area. The function doesn't
     *  return until somebody types a line of text, whereupon it returns
     *  the line. Returns null if the user types an empty line.
     */

    synchronized String read_line() throws InterruptedException
    {
        text_has_been_entered.wait_for_true();
        text_has_been_entered.set_false();

        String entered = input.getText();
        input.setText("");
        return (entered.length() == 0) ? null : entered;

    }


    static public void main( String[] args ) throws Exception
```

```
    {   Input_source_fixed source = new Input_source_fixed();
        String input;

        while( (input = source.read_line()) != null )
            System.out.println("Got: " + input );

        System.exit(0);         // kill the AWT Thread on exit
    }
}
```

Counting semaphores

The other semaphore I want to look at this month is the "Djikstra" counting semaphore. This one has no direct analog in Java, so it's among the more useful of the *com.holub.asynch* classes.

Counting semaphores are used to keep track of the availability of a resource within a pool of limited size. For example, you might have four connections open to a database server that are simply recycled to perform multiple queries. This way, you won't incur the overhead of opening a connection every time you make a query. Threads seeking to make queries should block (should be suspended, waiting), if no connections are available. They should be reactivated (released from wait) when a connection becomes available. A counting semaphore can solve this problem (though other solutions -- such as a thread-safe stack with a pop method that blocks if the stack is empty -- are also possible).

Counting semaphores are initialized with a count -- typically the number of objects available in the pool. Every time you acquire the semaphore, the count is decremented; every time you release the semaphore, it's incremented. On acquisition, if the count (after the decrement) is non-0, nothing happens, and you get your slot in the pool. If the count is 0, however, the acquiring thread blocks until some other thread releases the semaphore, thereby incrementing the count.

Counting semaphores typically have maximum counts as well as initial counts. A semaphore initialized with a count of 0, but with a maximum of 10, is effectively saying that 10 objects can be in the pool, but none of them are available right now. A *reverse-sensing semaphore* (which I haven't implemented) is also occasionally useful. This one blocks unless the count is 0. It's useful if you need to acquire the entire pool before you can do anything useful, or if you need to do something when the pool becomes empty (such as add extra elements).

Listing 4, below, shows my `Counting_semaphore` implementation. It implements the `Semaphore` interface introduced last month, so slots in multiple pools can be acquired safely by using last month's lock-manager class. As was the case with the `Condition` class, the counting semaphore is built around Java's `wait()` and `notify()` methods. The `acquire()` method waits if enough slots aren't available; the `release()` method notifies any waiting threads when a slot becomes available. If multiple threads are waiting, they'll have to sort out amongst themselves which actually gets the slot -- all but one will go back to waiting, but you can't reliably predict which one will get the slot.

I've taken the coward's way out and haven't implemented a version of `acquire()` that lets you get multiple slots at once. You have to acquire slots one at a time. The problem is that the number of slots you need may become available one at a time, but they may also be grabbed by other threads before the total number you need becomes available. Being forced to acquire slots one at a time actually increases your odds of getting the total number of slots you need over the odds you'd get by waiting until the semaphore's internal count came up to the total. You'll be able to suck the slots up one at a time as they become available. I do have a version of `release()` that lets you free up multiple slots all at once, however.

Note that the "honor system" is in use in Figure 4. An individual slot doesn't have the concept of ownership associated with it. A thread is on its honor to free up only the slots it has previously acquired. A `Counting_semaphore.Too_many_releases` object will be thrown if a thread tries to bring the total available-slot count above the maximum, but a thread could still incorrectly release the wrong number of slots without triggering the exception toss.

Listing 4: A counting semaphore

```
//-------------------------------------------------------------------
// This code (c) 1998 Allen I. Holub. All rights reserved.
//-------------------------------------------------------------------
// This code may not be distributed by yourself except in binary form,
// incorporated into a java .class file. You may use this code freely
// for personal purposes, but you may not incorporate it into any
// commercial product without express permission of Allen I. Holub in writing
//-------------------------------------------------------------------

package com.holub.asynch;

import com.holub.tools.Comparable;
import com.holub.asynch.Semaphore;
import com.holub.asynch.Lock_manager;

class Counting_semaphore implements Semaphore
{
    private int         count;
    private final int   max_count;

    /*****************************************************************
     *  Create a counting semaphore with the specified initial and
     *  maximum counts. release(), which increments the count, is not
     *  permitted to increment it past the maximum. If the initial_count
     *  is larger than the max_count, it is silently truncated.
     *
     *  @see release
     */
    Counting_semaphore( int initial_count, int max_count )
    {
        this.max_count  = max_count;
        this.count      = (initial_count > max_count)
                                ? max_count : initial_count ;
    }

    /*****************************************************************
     *  Create a counting semaphore with a maximum count of
     *  Integer.MAX_VALUE
     */
    Counting_semaphore( int initial_count )
    {   this( initial_count, Integer.MAX_VALUE );
    }

    /*****************************************************************
     *  Required override of Semaphore.id(). Don't call this function.
```

```
     *  @see Lock_manager
     */

    public        int  id() { return _id; }
    private final int  _id = Lock_manager.new_id();

    /*****************************************************************
     *  Acquire the semaphore, decrementing the count. Block if the
     *  count goes to zero. Bug: It’s possible in some situations
     *  for the timeout to be exceeded.
     *
     *  <p>I have deliberately not implemented a variant that allows
     *  acquisition of multiple slots in a single call because it’s not
     *  clear what you’d do if all requested slots aren’t available.
     *
     *  @throws InterruptedException if interrupted while waiting
     *          for the semaphore.
     *  @return true if we got the slot.
     */
    public synchronized void acquire(long timeout)
                                        throws InterruptedException
    {   while( count <= 0 )
            this.wait( timeout );
        --count;
    }

    /*****************************************************************
     *  Release the semaphore and increment the count.
     *  This one is the generic release required by the Semaphore
     *  interface, so all it can do is throw an exception if
     *  there’s an error.
     *  @throws Counting_semaphore.TooManyReleases (a RuntimeException)
     *      if you try to release a semaphore whose count is already
     *      at the maximum value.
     */
    public synchronized void release(){ release(1); }

    /*****************************************************************
     *  Release "increment" slots in the semaphore all at once.
     *  @param increment The amount to increment the count.
     *      If this value is zero, the current count is returned and
     *      no threads are released.
     *  @throws Counting_semaphore.TooManyReleases (a RuntimeException)
     *      if the current value + count is greater than the maximum.
     *      The semaphore will not have been modified in this case.
     *  @return the value of the count after the increment is added.
     */
    public synchronized int release( int increment )
    {
        int current_count = count;
        int new_count     = count + increment;

        if( new_count > max_count )
            throw new TooManyReleases();

        count = new_count;
        if( current_count == 0 && count > 0 )
            notifyAll();

        return count;
    }
```

```

    /** Thrown if you try to release more than the maximum number
     * of slots.
     */
    public static class TooManyReleases extends RuntimeException
    {   private TooManyReleases()
        {   super("Released semaphore that was at capacity");
        }
    }

    /*****************************************************************
     *  A semaphore-specific release function, returns an error status
     *  if the count would go past the initially specified maximum.
     *  @returns true if the semaphore was successfully released, false
     *  otherwise.
     */
    public synchronized boolean notifying_release()
    {
        if( count >= max_count )
            return false;

        if( ++count == 1 )
            notifyAll();
        return true;
    }
}
```

Listing 4 is pretty useful as it stands, but there's one variations on the counting-semaphore theme that deserves mention. I sometimes use what I call a "reverse sensing" semaphore. This one blocks until the pool is empty. In the pooled-database-connection example I discussed earlier, a thread that created new database connections might wait on this reverse-sensing semaphore, waking up when it needed to open additional connections. I'm sure you can think of other variations as well.

Wrapping up

So that's it for this month. We've seen how to use the built-in condition variable to communicate between threads generally (and the AWT event thread in particular). We've also seen that, though you can do a lot of what a condition variable does just by using `wait()` and `notify()`, you can't do everything. The `Condition` class adds the essential ability to not wait when the condition is already true. The `Counting_semaphore` isn't implemented as a Java primitive at all, so it can be particularly useful when you really need it to manage pooled resources.

Next month's column will continue on the current theme, presenting a `Timer` class that makes it easy to fire events on a regular schedule. The `Timer` implementation also demonstrates how to write code to suspend, resume, and stop threads without using the (now deprecated) `suspend()`, `resume()`, and `stop()` methods. ■

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. The version on the Web site should be considered the definitive version -- at least it corrects any bugs I know about.
  http://www.holub.com
- Design patterns like "strategy" are discussed in the "Gang of Four" book: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides's *Design Patterns: Elements of Reusable Object-Oriented Software* (Reading: Addison Wesley, 1995). This stunningly good book is a 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 object-oriented 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 those 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 Gang of Four 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

  Several good books that discuss Java threading issues are mentioned in the first article in this series. For convenience, I've listed them again:
- 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, Calif.: 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/books/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/

About the author
Allen Holub has been working in the computer industry since 1979, currently doing object-oriented design and Java training, mentoring, and occasionally even programming. He is widely published in various 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 object-oriented 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

If you have problems with this magazine, contact webmaster@javaworld.com
URL: http://www.javaworld.com/javaworld/jw-12-1998/jw-12-toolbox.html
Last modified: Wednesday, January 20, 1999