Understanding Java threads, Part 1: Introducing threads and runnables

Learn how to improve Java application performance using Java threads

This article is the first in a four-part *Java 101* series exploring Java threads. Although you might think threading in Java would be challenging to grasp, I intend to show you that threads are easy to understand. In this article, I introduce you to Java threads and runnables. In subsequent articles, we'll explore synchronization (via locks), synchronization problems (such as deadlock), the wait/notify mechanism, scheduling (with and without priority), thread interruption, timers, volatility, thread groups, and thread local variables.

What is a thread?

Conceptually, the notion of a *thread* is not difficult to grasp: it's an independent path of execution through program code. When multiple threads execute, one thread's path through the same code usually differs from the others. For example, suppose one thread executes the byte code equivalent of an if-else statement's `if`part, while another thread executes the byte code equivalent of the `else` part. How does the JVM keep track of each thread's execution? The JVM gives each thread its own method-call stack. In addition to tracking the current byte code instruction, the method-call stack tracks local variables, parameters the JVM passes to a method, and the method's return value.

When multiple threads execute byte-code instruction sequences in the same program, that action is known as *multithreading*. Multithreading benefits a program in various ways:

- Multithreaded GUI (graphical user interface)-based programs remain responsive to users while performing other tasks, such as repaginating or printing a document.

- Threaded programs typically finish faster than their nonthreaded counterparts. This is especially true of threads running on a multiprocessor machine, where each thread has its own processor.

Java accomplishes multithreading through its java.lang.Threadclass. Each Thread object describes a single thread of execution. That execution occurs in Thread's run() method. Because the defaultrun() method does nothing, you must subclass Thread and override run() to accomplish useful work. For a taste of threads and multithreading in the context ofThread, examine Listing 1:

Listing 1. ThreadDemo.java

```
// ThreadDemo.java
class ThreadDemo
{
   public static void main (String [] args)
   {
      MyThread mt = new MyThread ();
      mt.start ();
      for (int i = 0; i < 50; i++)
           System.out.println ("i = " + i + ", i * i = " + i * i);
   }
}
class MyThread extends Thread
{
   public void run ()
   {
      for (int count = 1, row = 1; row < 20; row++, count++)
      {
           for (int i = 0; i < count; i++)
                System.out.print ('*');
           System.out.print ('\n');
      }
   }}
```

Listing 1 presents source code to an application consisting of classes ThreadDemo and MyThread. Class ThreadDemo drives the application by creating a MyThread object, starting a thread that associates with that object, and executing some code to print a table of squares. In contrast, MyThread overridesThread's run() method to print (on the standard output stream) a right-angle triangle composed of asterisk characters.

Thread scheduling and the JVM

Most (if not all) JVM implementations use the underlying platform's threading capabilities. Because those capabilities are platform-specific, the order of your multithreaded programs' output might differ from the order of someone else's output. That difference results from scheduling, a topic I explore later in this series.

When you type java ThreadDemo to run the application, the JVM creates a starting thread of execution, which executes the main() method. By executing mt.start ();, the starting thread tells the JVM to create a second thread of execution that executes the byte code instructions comprising the MyThread object's run() method. When the start() method returns, the starting thread executes its for loop to print a table of squares, while the new thread executes the run() method to print the right-angle triangle.

What does the output look like? Run ThreadDemo to find out. You will notice each thread's output tends to intersperse with the other's output. That results because both threads send their output to the same standard output stream.

The Thread class

To grow proficient at writing multithreaded code, you must first understand the various methods that make up the Threadclass. This section explores many of those methods. Specifically, you learn about methods for starting threads, naming threads, putting threads to sleep, determining whether a thread is alive, joining one thread to another thread, and enumerating all active threads in the current thread's thread group and subgroups. I also discuss Thread's debugging aids and user threads versus daemon threads.

I'll present the remainder of Thread's methods in subsequent articles, with the exception of Sun's deprecated methods.

Deprecated methods

Sun has deprecated a variety of Thread methods, such assuspend() and resume(), because they can lock up your programs or damage objects. As a result, you should not call them in your code. Consult the SDK documentation for workarounds to those methods. I do not cover deprecated methods in this series.

Constructing threads

Thread has eight constructors. The simplest are:

- Thread(), which creates a Thread object with a default name
- Thread(String name), which creates a Thread object with a name that the name argument specifies

The next simplest constructors are Thread(Runnable target)and Thread(Runnable target, String name). Apart from theRunnable parameters, those constructors are identical to the aforementioned constructors. The difference: The Runnable parameters identify objects outside Thread that provide the run() methods. (You learn about Runnablelater in this article.) The final four constructors resemble Thread(String name),Thread(Runnable target), and Thread(Runnable target, String name); however, the final constructors also include a ThreadGroup argument for organizational purposes.

One of the final four constructors, Thread(ThreadGroup group, Runnable target, String name, long stackSize), is interesting in that it lets you specify the desired size of the thread's method-call stack. Being able to specify that size proves helpful in programs with methods that utilize recursion—an execution technique whereby a method repeatedly calls itself—to elegantly solve certain problems. By explicitly setting the stack size, you can sometimes prevent

StackOverflowErrors. However, too large a size can result in OutOfMemoryErrors. Also, Sun regards the method-call stack's size as platform-dependent. Depending on the platform, the method-call stack's size might change. Therefore, think carefully about the ramifications to your program before writing code that calls Thread(ThreadGroup group, Runnable target, String name, long stackSize).

Start your vehicles

Threads resemble vehicles: they move programs from start to finish. Thread and Thread subclass objects are not threads. Instead, they describe a thread's attributes, such as its name, and contain code (via a run() method) that the thread executes. When the time comes for a new thread to execute run(), another thread calls the Thread's or its subclass object's start()method. For example, to start a second thread, the application's starting thread—which executes main()—calls start(). In response, the JVM's thread-handling code works with the platform to ensure the thread properly initializes and calls aThread's or its subclass object's run() method.

Once start() completes, multiple threads execute. Because we tend to think in a linear fashion, we often find it difficult to understand the*concurrent* (simultaneous) activity that occurs when two or more threads are running. Therefore, you should examine a chart that shows where a thread is executing (its position) versus time. The figure below presents such a chart.

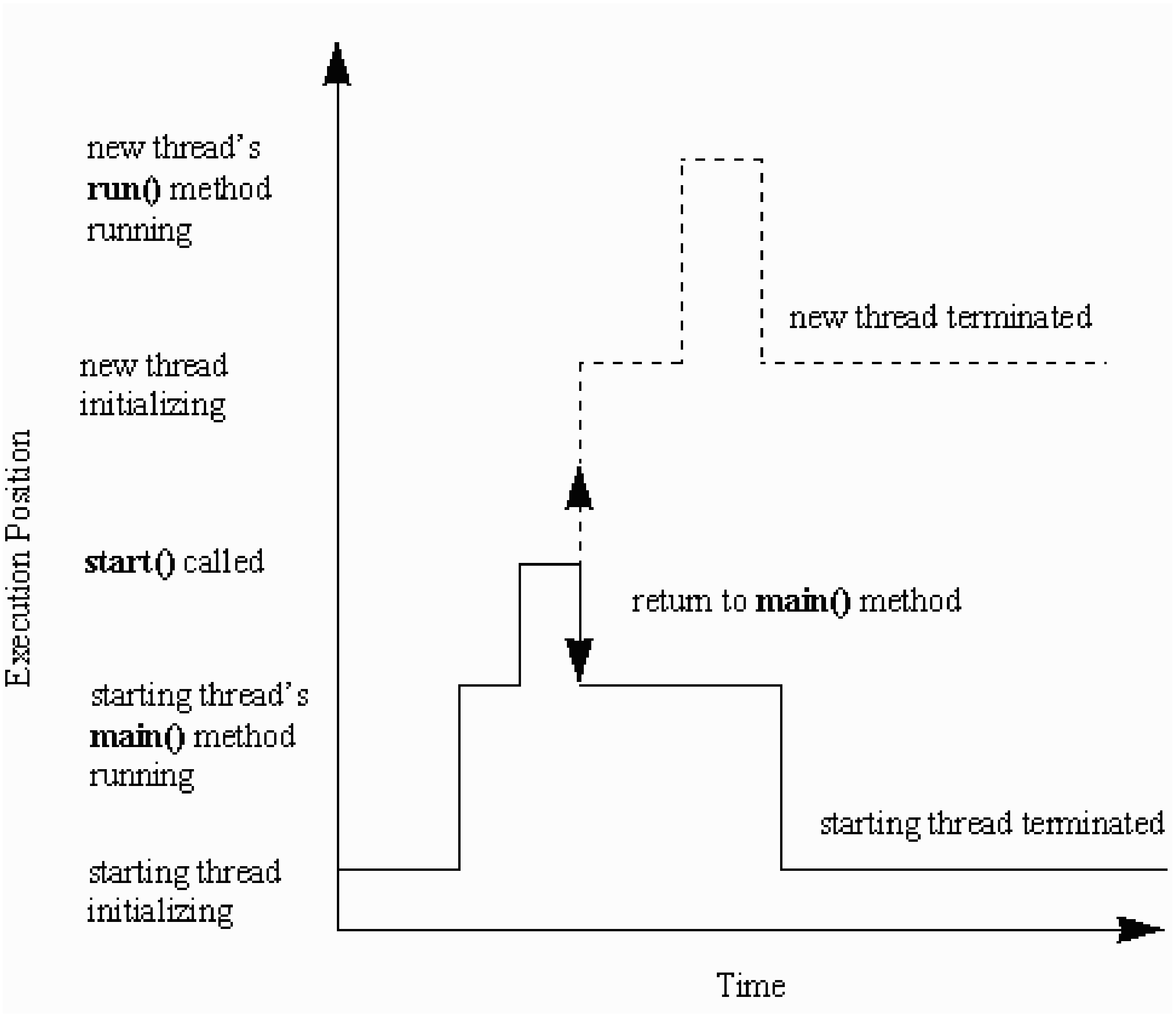
new thread's **run()** method running
new thread initializing
Execution Position
start() called
starting thread's **main()** method running
starting thread initializing
new thread terminated
return to **main()** method
starting thread terminated
Time

Figure 1. The behaviors of a starting thread's and a newly created thread's execution positions versus time

The chart shows several significant time periods:

- The starting thread's initialization
- The moment that thread begins to execute main()
- The moment that thread begins to execute start()
- The momentstart() creates a new thread and returns to main()
- The new thread's initialization
- The moment the new thread begins to execute run()
- The different moments each thread terminates

Note that the new thread's initialization, its execution ofrun(), and its termination happen simultaneously with the starting thread's execution. Also note that after a thread callsstart(), subsequent calls to that method before the run()method exits cause start() to throw ajava.lang.IllegalThreadStateException object.

What's in a name?

During a debugging session, distinguishing one thread from another in a user-friendly fashion proves helpful. To differentiate among threads, Java associates a name with a thread. That name defaults to Thread, a hyphen character, and a zero-based integer number. You can accept Java's default thread names or you can choose your own. To accommodate custom names, Thread provides constructors that take name arguments and a setName(String name) method. Thread also provides a getName() method that returns the current name. Listing 2 demonstrates how to establish a custom name via the Thread(String name) constructor and retrieve the current name in the run()method by calling getName():

Listing 2. NameThatThread.java

```
// NameThatThread.java
class NameThatThread
{
   public static void main (String [] args)
   {
      MyThread mt;
      if (args.length == 0)
          mt = new MyThread ();
      else
          mt = new MyThread (args [0]);
      mt.start ();
   }
}
class MyThread extends Thread
{
   MyThread ()
   {
      // The compiler creates the byte code equivalent of super ();
   }
   MyThread (String name)
   {
      super (name); // Pass name to Thread superclass
   }
   public void run ()
   {
      System.out.println ("My name is: " + getName ());
   }
}
```

You can pass an optional name argument to MyThread on the command line. For example, java NameThatThread X establishesX as the thread's name. If you fail to specify a name, you'll see the following output:

```
My name is: Thread-1
```

If you prefer, you can change the super (name); call in theMyThread (String name) constructor to a call to setName (String name)—as in setName (name);. That latter method call achieves the same objective—establishing the thread's name—as super (name);. I leave that as an exercise for you.

Naming main

Java assigns the name main to the thread that runs the main() method, the starting thread. You typically see that name in the Exception in thread "main" message that the JVM's default exception handler prints when the starting thread throws an exception object.

To sleep or not to sleep

Later in this column, I will introduce you to *animation*— repeatedly drawing on one surface images that slightly differ from each other to achieve a movement illusion. To accomplish animation, a thread must pause during its display of two consecutive images. Calling Thread's static sleep(long millis) method forces a thread to pause for millis milliseconds. Another thread could possibly interrupt the sleeping thread. If that happens, the sleeping thread awakes and throws an InterruptedExceptionobject from the sleep(long millis) method. As a result, code that calls sleep(long millis) must appear within a try block—or the code's method must includeInterruptedException in its throws clause.

To demonstrate sleep(long millis), I've written a CalcPI1application. That application starts a new thread that uses a mathematic algorithm to calculate the value of the mathematical constant pi. While the new thread calculates, the starting thread pauses for 10 milliseconds by callingsleep(long millis). After the starting thread awakes, it prints the pi value, which the new thread stores in variable pi. Listing 3 presents CalcPI1's source code:

Listing 3. CalcPI1.java

```
// CalcPI1.java
class CalcPI1
{
   public static void main (String [] args)
   {
      MyThread mt = new MyThread ();
      mt.start ();
      try
      {
          Thread.sleep (10); // Sleep for 10 milliseconds
      }
      catch (InterruptedException e)
      {
      }
      System.out.println ("pi = " + mt.pi);
   }
}
class MyThread extends Thread
{
   boolean negative = true;
```

```
  double pi; // Initializes to 0.0, by default
  public void run ()
  {
     for (int i = 3; i < 100000; i += 2)
     {
          if (negative)
              pi -= (1.0 / i);
          else
              pi += (1.0 / i);
          negative = !negative;
     }
     pi += 1.0;
     pi *= 4.0;
     System.out.println ("Finished calculating PI");
  }
}
```

If you run this program, you will see output similar (but probably not identical) to the following:

```
pi = -0.2146197014017295
Finished calculating PI
```

Why is the output incorrect? After all, pi's value is roughly equivalent to 3.14159. The answer: The starting thread awoke too soon. Just as the new thread was beginning to calculate pi, the starting thread woke up, read pi's current value, and printed that value. We can compensate by increasing the delay from 10 milliseconds to a longer value. That longer value, which (unfortunately) is platform dependent, will give the new thread a chance to complete its calculations before the starting thread awakes. (Later, you will learn about a platform-independent technique that prevents the starting thread from waking until the new thread finishes.)

Sleeping threads don't lie

`Thread` also supplies a `sleep(long millis, int nanos)`method, which puts the thread to sleep for `millis`milliseconds and `nanos` nanoseconds. Because most JVM-based platforms do not support resolutions as small as a nanosecond, JVM thread-handling code rounds the number of nanoseconds to the nearest number of milliseconds. If a platform does not support a resolution as small as a millisecond, JVM thread-handling code rounds the number of milliseconds to the nearest multiple of the smallest resolution that the platform supports.

Is it dead or alive?

When a program calls `Thread`'s `start()` method, a time period (for initialization) passes before a new thread calls `run()`. After `run()` returns, a time period passes before the JVM cleans up the thread. The JVM considers the thread to be alive immediately prior to the thread's call to `run()`, during the thread's execution of`run()`, and immediately after `run()` returns. During that interval, `Thread`'s `isAlive()`method returns a Boolean true value. Otherwise, that method returns false.

`isAlive()` proves helpful in situations where a thread needs to wait for another thread to finish its `run()` method before the first thread can examine the other thread's results. Essentially, the thread that needs to wait enters a `while` loop. While`isAlive()` returns true for the other thread, the waiting thread calls `sleep(long millis)` (or `sleep(long millis, int nanos)`) to periodically sleep (and avoid wasting many CPU cycles). Once`isAlive()` returns false, the waiting thread can examine the other thread's results.

Where would you use such a technique? For starters, how about a modified version of CalcPI1, where the starting thread waits for the new thread to finish before printing pi's value? Listing 4's CalcPI2 source code demonstrates that technique:

Listing 4. CalcPI2.java

```java
// CalcPI2.java
class CalcPI2
{
   public static void main (String [] args)
   {
      MyThread mt = new MyThread ();
      mt.start ();
      while (mt.isAlive ())
        try
        {
            Thread.sleep (10); // Sleep for 10 milliseconds
        }
        catch (InterruptedException e)
        {
        }
      System.out.println ("pi = " + mt.pi);
   }
}
class MyThread extends Thread
{
   boolean negative = true;
   double pi; // Initializes to 0.0, by default
   public void run ()
   {
      for (int i = 3; i < 100000; i += 2)
      {
           if (negative)
```

```
                pi -= (1.0 / i);
            else
                pi += (1.0 / i);
            negative = !negative;
        }
        pi += 1.0;
        pi *= 4.0;
        System.out.println ("Finished calculating PI");
    }
}
```

CalcPI2's starting thread sleeps in 10 millisecond intervals, until mt.isAlive () returns false. When that happens, the starting thread exits from its while loop and prints pi's contents. If you run this program, you will see output similar (but probably not identical) to the following:

```
Finished calculating PI
pi = 3.1415726535897894
```

Now doesn't that look more accurate?

Is it alive?

A thread could possibly call the isAlive() method on itself. However, that does not make sense becauseisAlive() will always return true.

Joining forces

Because the while loop/isAlive() method/sleep() method technique proves useful, Sun packaged it into a trio of methods: join(), join(long millis), and join(long millis, int nanos). The current thread calls join(), via another thread's thread object reference when it wants to wait for that other thread to terminate. In contrast, the current thread calls join(long millis) or join(long millis, int nanos)when it wants to either wait for that other thread to terminate or wait until a combination of millis millseconds and nanos nanoseconds passes. (As with the sleep() methods, the JVM thread-handling code will round up the argument values of the join(long millis) and join(long millis, int nanos) methods.) Listing 5's CalcPI3 source code demonstrates a call to join():

Listing 5. CalcPI3.java

```
// CalcPI3.java
class CalcPI3
{
   public static void main (String [] args)
   {
      MyThread mt = new MyThread ();
      mt.start ();
      try
      {
          mt.join ();
      }
      catch (InterruptedException e)
      {
      }
      System.out.println ("pi = " + mt.pi);
   }
}
class MyThread extends Thread
{
   boolean negative = true;
   double pi; // Initializes to 0.0, by default
   public void run ()
   {
      for (int i = 3; i < 100000; i += 2)
      {
           if (negative)
               pi -= (1.0 / i);
           else
               pi += (1.0 / i);
           negative = !negative;
      }
      pi += 1.0;
      pi *= 4.0;
      System.out.println ("Finished calculating PI");
   }
}
```

CalcPI3's starting thread waits for the thread that associates with the MyThread object, referenced by mt, to terminate. The starting thread then prints pi's value, which is identical to the value that CalcPI2 outputs.

Do not attempt to join the current thread to itself because the current thread will wait forever.)

Census taking

In some situations, you might want to know which threads are actively running in your program. Thread supplies a pair of methods to help you with that task: activeCount() and enumerate(Thread [] thdarray). But those methods work only in the context of the current thread's thread group. In other words, those methods identify only active threads that belong to the same thread group as the current thread. (I discuss the thread group—an organizational mechanism—concept in a future series article.)

The static activeCount() method returns a count of the threads actively executing in the current thread's thread group. A program uses this method's integer return value to size an array of Thread references. To retrieve those references, the program must call the static enumerate(Thread [] thdarray) method. That method's integer return value identifies the total number of Thread references thatenumerate(Thread []thdarray) stores in the array. To see how these methods work together, check out Listing 6:

Listing 6. Census.java

```
// Census.java
class Census
{
  public static void main (String [] args)
  {
    Thread [] threads = new Thread [Thread.activeCount ()];
    int n = Thread.enumerate (threads);
    for (int i = 0; i < n; i++)
        System.out.println (threads [i].toString ());
  }
}
```

When run, this program produces output similar to the following:

```
Thread[main,5,main]
```

The output shows that one thread, the starting thread, is running. The leftmost mainidentifies that thread's name. The 5 indicates that thread's priority, and the rightmost main identifies that thread's thread group. You might be disappointed that you cannot see any system threads, such as the garbage collector thread, in the output. That limitation results from Thread's enumerate(Thread [] thdarray)method, which interrogates only the current thread's thread group for active threads. However, the ThreadGroup class contains multiple enumerate() methods that allow you to capture references to all active threads, regardless of thread group. Later in this series, I will show you how to enumerate all references when I explore ThreadGroup.

activeCount() and NullPointerException

Do not depend on activeCount()'s return value when iterating over an array. If you do, your program runs the risk of throwing NullPointerException objects. Why? Between the calls to activeCount() and enumerate(Thread [] thdarray), one or more threads might possibly terminate. As a result, enumerate(Thread [] thdarray)would copy fewer thread references into its array. Therefore, think of activeCount()'s return value as a maximum value for array-sizing purposes only. Also, think of enumerate(Thread [] thdarray)'s return value as representing the number of active threads at the time of a program's call to that method.

Antibugging

If your program malfunctions, and you suspect that the problem lies with a thread, you can learn details about that thread by calling Thread's dumpStack() andtoString() methods. The static dumpStack() method, which provides a wrapper around new Exception ("Stack trace").printStackTrace ();, prints a stack trace for the current thread. toString() returns a String object that describes the thread's name, priority, and thread group according to the following format: Thread[thread-name,priority,thread-group]. (You will learn more about priority later in this series.)

The caste system

Not all threads are created equal. They divide into two categories: user and daemon. A *user thread* performs important work for the program's user, work that must finish before the application terminates. In contrast, a *daemon thread* performs housekeeping (such as garbage collection) and other background tasks that probably do not contribute to the application's main work but are necessary for the application to continue its main work. Unlike user threads, daemon threads do not need to finish before the application terminates. When an application's starting thread (which is a user thread) terminates, the JVM checks whether any other user threads are running. If some are, the JVM prevents the application from terminating. Otherwise, the JVM terminates the application regardless of whether daemon threads are running.

When to use currentThread()

In several places, this article refers to the concept of a *current thread.* If you need access to a Thread object that describes the current thread, call Thread's static currentThread() method. Example: Thread current = Thread.currentThread ();.

When a thread calls a thread object's start() method, the newly started thread is a user thread. That is the default. To establish a thread as a daemon thread, the program must call Thread's setDaemon(boolean isDaemon) method with a Boolean true argument value prior to the call to start(). Later, you can check if a thread is daemon by calling Thread's isDaemon() method. That method returns a Boolean true value if the thread is daemon.

To let you play with user and daemon threads, I wroteUserDaemonThreadDemo:

Listing 7. UserDaemonThreadDemo.java

```
// UserDaemonThreadDemo.java
class UserDaemonThreadDemo
{
   public static void main (String [] args)
   {
      if (args.length == 0)
         new MyThread ().start ();
      else
      {
         MyThread mt = new MyThread ();
         mt.setDaemon (true);
         mt.start ();
      }
      try
      {
         Thread.sleep (100);
      }
      catch (InterruptedException e)
      {
      }
   }
}
class MyThread extends Thread
```

```
{
  public void run ()
  {
    System.out.println ("Daemon is " + isDaemon ());
    while (true);
  }
}
```

After compiling the code, run UserDaemonThreadDemo via the Java 2 SDK's java command. If you run the program with no command-line arguments, as in java UserDaemonThreadDemo, for example, new MyThread ().start (); executes. That code fragment starts a user thread that prints Daemon is false prior to entering an infinite loop. (You must press Ctrl-C or an equivalent keystroke combination to terminate that infinite loop.) Because the new thread is a user thread, the application keeps running after the starting thread terminates. However, if you specify at least one command-line argument, as in java UserDaemonThreadDemo x, for example, mt.setDaemon (true);executes, and the new thread will be a daemon. As a result, once the starting thread awakes from its 100-millisecond sleep and terminates, the new daemon thread will also terminate.

A setDaemon() exception

Note that the setDaemon(boolean isDaemon) method throws anIllegalThreadStateException object if a call is made to that method after the thread starts execution.

Runnables

After studying the previous section's examples, you might think that introducing multithreading into a class always requires you to extend Thread and have your subclass override Thread's run() method. That is not always an option, however. Java's enforcement of implementation inheritance prohibits a class from extending two or more superclasses. As a result, if a class extends a non-Thread class, that class cannot also extend Thread. Given that restriction, how is it possible to introduce multithreading into a class that already extends some other class? Fortunately, Java's designers realized that situations would arise where subclassingThread wouldn't be possible. That realization led to the java.lang.Runnableinterface and Thread constructors with Runnable parameters, such asThread(Runnable target).

The Runnable interface declares a single method signature: void run();. That signature is identical to Thread's run() method signature and serves as a thread's entry of execution. Because Runnable is an interface, any class can implement that interface by attaching an implements clause to the class header and by providing an appropriate run() method. At execution time, program code can create an object, or*runnable,* from that class and pass the runnable's reference to an appropriate Threadconstructor. The constructor stores that reference within the Thread object and ensures that a new thread calls the runnable's run() method after a call to theThread object's start() method, which Listing 8 demonstrates:

Listing 8. RunnableDemo.java

```
// RunnableDemo.java

public class RunnableDemo extends java.applet.Applet implements
Runnable
{
   private Thread t;

   public void run ()
   {
      while (t == Thread.currentThread ())
      {
           int width = rnd (30);
           if (width < 2)
               width += 2;

           int height = rnd (10);
           if (height < 2)
```

```
               height += 2;

            draw (width, height);
      }
   }

   public void start ()
   {
      if (t == null)
      {
         t = new Thread (this);
         t.start ();
      }
   }

   public void stop ()
   {
      if (t != null)
         t = null;
   }

   private void draw (int width, int height)
   {
      for (int c = 0; c < width; c++)
           System.out.print ('*');

      System.out.print ('\n');

      for (int r = 0; r < height - 2; r++)
      {
           System.out.print ('*');

           for (int c = 0; c < width - 2; c++)
```

```
                System.out.print (' ');

           System.out.print ('*');

           System.out.print ('\n');
      }

      for (int c = 0; c < width; c++)
           System.out.print ('*');

      System.out.print ('\n');
   }

   private int rnd (int limit)
   {
      // Return a random number x in the range 0 <= x < limit.

      return (int) (Math.random () * limit);
   }
}
```

RunnableDemo describes an applet for repeatedly outputting asterisk-based rectangle outlines on the standard output. To accomplish this task, Runnable must extend thejava.applet.Applet class (java.applet identifies the package in which Applet is located -- I discuss packages in a future article) and implement the Runnable interface.

An applet provides a public void start() method, which is called (typically by a Web browser) when an applet is to start running, and provides a public void stop() method, which is called when an applet is to stop running.

The start() method is the perfect place to create and start a thread, andRunnableDemo accomplishes this task by executing t = new Thread (this); t.start ();. I pass this to

Thread's constructor because the applet is a runnable due toRunnableDemo implementing Runnable.

The stop() method is the perfect place to stop a thread, by assigning null to theThread variable. I cannot use Thread's public void stop() method for this task because this method has been deprecated -- it's unsafe to use.

The run() method contains an infinite loop that runs for as long asThread.currentThread() returns the same Thread reference as located in Threadvariable t. The reference in this variable is nullified when the applet's stop()method is called.

Because RunnableDemo's new output would prove too lengthy to include with this article, I suggest you compile and run that program yourself.

You will need to use the appletviewer tool and an HTML file to run the applet. Listing 9 presents a suitable HTML file -- the width and height are set to 0 because no graphical output is generated.

Listing 9. RunnableDemo.html

```
<applet code="RunnableDemo" width="0" height="0"></applet>
```

Specify appletviewer RunnableDemo.html to run this applet.

Thread vs Runnable?

When you face a situation where a class can either extend Thread or implement Runnable, which approach do you choose? If the class already extends another class, you must implement Runnable. However, if that class extends no other class, think about the class name. That name will suggest that the class's objects are either active or passive. For example, the nameTicker suggests that its objects are active—they tick. Thus, the Ticker class would extend Thread, and Ticker objects would be specialized Thread objects.

Review

Users expect programs to achieve strong performance. One way to accomplish that task is to use threads. A thread is an independent path of execution through program code. Threads benefit GUI-based programs because they allow those programs to remain responsive to users while performing other tasks. In addition, threaded programs typically finish faster than their nonthreaded counterparts. This is especially true of threads running on a multiprocessor machine, where each thread has its own processor. The`Thread` and `Thread` subclass objects describe threads and associate with those entities. For those classes that cannot extend `Thread`, you must create a runnable to take advantage of multithreading.

Next month, I continue this series by showing you how to synchronize access to shared data.

Jeff Friesen has been involved with computers for the past 20 years. He holds a degree in computer science and has worked with many computer languages. Jeff has also taught introductory Java programming at the college level. In addition to writing for JavaWorld, he has written his own Java book for beginners— Java 2 by Example, Second Edition (Que Publishing, 2001; ISBN: 0789725932)—and helped write Using Java 2 Platform, Special Edition (Que Publishing, 2001; ISBN: 0789724685). Jeff goes by the nickname Java Jeff (or JavaJeff). To see what he's working on, check out his Website at http://www.javajeff.com.

Understanding Java threads, Part 2: Thread synchronization

Use synchronization to serialize thread access to critical code sections

Last month I showed you how easy it is to create thread objects, start threads that associate with those objects by calling `Thread`'s `start()` method, and perform simple thread operations by calling other `Thread` methods such as the three overloaded `join()` methods. This month we're taking on multithreaded Java programs, however, which are more complex.

Multithreaded programs often function erratically or produce erroneous values due to the lack of thread *synchronization*. Synchronization is the act of *serializing* (or ordering one at a time) thread access to those code sequences that let multiple threads manipulate class and instance field variables, and other shared resources. I call those code sequences *critical code sections.*. This month's column is all about using synchronization to serialize thread access to critical code sections in your programs.

I begin with an example that illustrates why some multithreaded programs must use synchronization. I next explore Java's synchronization mechanism in terms of monitors and locks, and the `synchronized` keyword. Because incorrectly using the synchronization mechanism negates its benefits, I conclude by investigating two problems that result from such misuse.

Tip: Unlike class and instance field variables, threads cannot share local variables and parameters. The reason: Local variables and parameters allocate on a thread's method-call stack. As a result, each thread receives its own copy of those variables. In contrast, threads can share class fields and instance fields because those variables do not allocate on a thread's method-call stack. Instead, they allocate in shared heap memory—as part of classes (class fields) or objects (instance fields).

The need for synchronization

Why do we need synchronization? For an answer, consider this example: You write a Java program that uses a pair of threads to simulate withdrawal/deposit of financial transactions. In that program, one thread performs deposits while the other performs withdrawals. Each thread manipulates a pair of *shared variables,* class and instance field variables, that identifies the financial transaction's name and amount. For a correct financial transaction, each thread must finish assigning values to the`name` and `amount` variables (and print those values, to simulate saving the transaction) before the other thread starts assigning values to name and amount (and also printing those values). After some work, you end up with source code that resembles Listing 1:

Listing 1. NeedForSynchronizationDemo.java

```
// NeedForSynchronizationDemo.java
class NeedForSynchronizationDemo
{
   public static void main (String [] args)
   {
      FinTrans ft = new FinTrans ();
      TransThread tt1 = new TransThread (ft, "Deposit Thread");
      TransThread tt2 = new TransThread (ft, "Withdrawal Thread");
      tt1.start ();
      tt2.start ();
   }
}
class FinTrans
{
   public static String transName;
   public static double amount;
}
class TransThread extends Thread
{
   private FinTrans ft;
   TransThread (FinTrans ft, String name)
   {
      super (name); // Save thread's name
      this.ft = ft; // Save reference to financial transaction object
   }
   public void run ()
   {
    for (int i = 0; i < 100; i++)
    {
        if (getName ().equals ("Deposit Thread"))
        {
            // Start of deposit thread's critical code section
            ft.transName = "Deposit";
            try
            {
               Thread.sleep ((int) (Math.random () * 1000));
            }
            catch (InterruptedException e)
            {
            }
            ft.amount = 2000.0;
            System.out.println (ft.transName + " " + ft.amount);
            // End of deposit thread's critical code section
        }
        else
        {
            // Start of withdrawal thread's critical code section
            ft.transName = "Withdrawal";
            try
            {
               Thread.sleep ((int) (Math.random () * 1000));
            }
            catch (InterruptedException e)
            {
            }
            ft.amount = 250.0;
            System.out.println (ft.transName + " " + ft.amount);
            // End of withdrawal thread's critical code section
        }
    }
```

```
  }
}
```

NeedForSynchronizationDemo's source code has two critical code sections: one accessible to the deposit thread, and the other accessible to the withdrawal thread. Within the deposit thread's critical code section, that thread assigns the DepositString object's reference to shared variable transName and assigns 2000.0 to shared variable amount. Similarly, within the withdrawal thread's critical code section, that thread assigns the Withdrawal String object's reference to transName and assigns 250.0 to amount. Following each thread's assignments, those variables' contents print. When you runNeedForSynchronizationDemo, you might expect output similar to a list of interspersed Withdrawal 250.0 and Deposit 2000.0lines. Instead, you receive output resembling the following:

```
Withdrawal 250.0
Withdrawal 2000.0
Deposit 2000.0
Deposit 2000.0
Deposit 250.0
```

The program definitely has a problem. The withdrawal thread should not be simulating $2000 withdrawals, and the deposit thread should not be simulating $250 deposits. Each thread produces inconsistent output. What causes those inconsistencies? Consider the following:

- On a single-processor machine, threads share the processor. As a result, one thread can only execute for a certain time period. At that time, the JVM/operating system pauses that thread's execution and allows another thread to execute—a manifestation of thread scheduling, a topic I discuss in Part 3. On a multiprocessor machine, depending on the number of threads and processors, each thread can have its own processor.
- On a single-processor machine, a thread's execution period might not last long enough for that thread to finish executing its critical code section before another

thread begins executing its own critical code section. On a multiprocessor machine, threads can simultaneously execute code in their critical code sections. However, they might enter their critical code sections at different times.

- On either single-processor or multiprocessor machines, the following scenario can occur: Thread A assigns a value to shared variable X in its critical code section and decides to perform an input/output operation that requires 100 milliseconds. Thread B then enters its critical code section, assigns a different value to X, performs a 50-millisecond input/output operation, and assigns values to shared variables Y and Z. Thread A's input/output operation completes, and that thread assigns its own values to Y and Z. Because X contains a B-assigned value, whereas Y and Z contain A-assigned values, an inconsistency results.

How does an inconsistency arise in `NeedForSynchronizationDemo`? Suppose the deposit thread executes `ft.transName = "Deposit";` and then calls `Thread.sleep()`. At that point, the deposit thread surrenders control of the processor for the time period it must sleep, and the withdrawal thread executes. Assume the deposit thread sleeps for 500 milliseconds (a randomly selected value, thanks to`Math.random()`, from the inclusive range 0 through 999 milliseconds; I explore `Math`and its `random()` method in a future article). During the deposit thread's sleep time, the withdrawal thread executes `ft.transName = "Withdrawal";`, sleeps for 50 milliseconds (the withdrawal thread's randomly selected sleep value), awakes, executes `ft.amount = 250.0;`, and executes `System.out.println (ft.transName + " " + ft.amount);`—all before the deposit thread awakes. As a result, the withdrawal thread prints `Withdrawal 250.0`, which is correct. When the deposit thread awakes, it executes `ft.amount = 2000.0;`, followed by `System.out.println (ft.transName + " " + ft.amount);`. This time, `Withdrawal 2000.0` prints, which is not correct. Although the deposit thread previously assigned the `"Deposit"`'s reference to`transName`, that reference subsequently disappeared when the withdrawal thread assigned the `"Withdrawal"`'s reference to that shared variable. When the deposit thread awoke, it failed to restore the correct reference to `transName`, but continued its execution by assigning `2000.0` to `amount`. Although neither variable has an invalid value, the combined values of both variables represent an inconsistency. In this case, their values represent an attempt to withdraw ,000.

Long ago, computer scientists invented a term to describe the combined behaviors of multiple threads that lead to inconsistencies. That term is *race condition*—the act of each thread racing to complete its critical code section before some other thread enters that same critical code section. AsNeedForSynchronizationDemo demonstrates, threads' execution orders are unpredictable. There is no guarantee that a thread can complete its critical code section before some other thread enters that section. Hence, we have a race condition, which causes inconsistencies. To prevent race conditions, each thread must complete its critical code section before another thread enters either the same critical code section or another related critical code section that manipulates the same shared variables or resources. With no means of serializing access—that is, allowing access to only one thread at a time —to a critical code section, you can't prevent race conditions or inconsistencies. Fortunately, Java provides a way to serialize thread access: through its synchronization mechanism.

Note: Of Java's types, only long integer and double-precision floating-point variables are prone to inconsistencies. Why? A 32-bit JVM typically accesses a 64-bit long integer variable or a 64-bit double-precision floating-point variable in two adjacent 32-bit steps. One thread might complete the first step and then wait while another thread executes both steps. Then, the first thread might awake and complete the second step, producing a variable with a value different from either the first or second thread's value. As a result, if at least one thread can modify either a long integer variable or a double-precision floating-point variable, all threads that read and/or modify that variable must use synchronization to serialize access to the variable.

Java's synchronization mechanism

Java provides a synchronization mechanism for preventing more than one thread from executing code in one or more critical code sections at any point in time. That mechanism bases itself on the concepts of monitors and locks. Think of a *monitor* as a protective wrapper around a critical code section and a *lock* as a software entity that a monitor uses to prevent multiple threads from entering the monitor. The idea is this: When a thread wishes to enter a monitor-guarded critical code section, that thread must acquire the lock associated with an object that associates with the monitor. (Each object has its own lock.) If some other thread holds that lock, the JVM forces the requesting thread to wait in a waiting area associated with the monitor/lock. When the thread in the monitor releases the lock, the JVM removes the waiting thread from the monitor's waiting area and allows that thread to acquire the lock and proceed to the monitor's critical code section.

To work with monitors/locks, the JVM provides the monitorenter and monitorexitinstructions. Fortunately, you do not need to work at such a low level. Instead, you can use Java's synchronized keyword in the context of the synchronized statement and synchronized methods.

The synchronized statement

Some critical code sections occupy small portions of their enclosing methods. To guard multiple thread access to such critical code sections, you use thesynchronized statement. That statement has the following syntax:

```
'synchronized' '(' objectidentifier ')'
'{'
  // Critical code section
'}'
```

The synchronized statement begins with keyword synchronized and continues with an *objectidentifier,* which appears between a pair of round brackets. The*objectidentifier* references an object whose lock associates with the monitor that the synchronized statement represents. Finally, the Java statements' critical code section appears between a pair of brace characters. How do you interpret thesynchronized statement? Consider the following code fragment:

```
synchronized ("sync object")
{
  // Access shared variables and other shared resources
}
```

From a source code perspective, a thread attempts to enter the critical code section that the synchronized statement guards. Internally, the JVM checks if some other thread holds the lock associated with the "sync object" object. (Yes, "sync object"is an object. You will understand why in a future article.) If no other thread holds the lock, the JVM gives the lock to the requesting thread and allows that thread to enter the critical code section between the brace characters. However, if some other thread holds the lock, the JVM forces the requesting thread to wait in a private waiting area until the thread currently within the critical code section finishes executing the final statement and transitions past the final brace character.

You can use the synchronized statement to eliminateNeedForSynchronizationDemo's race condition. To see how, examine Listing 2:

Listing 2. SynchronizationDemo1.java

```
// SynchronizationDemo1.java
class SynchronizationDemo1
{
   public static void main (String [] args)
   {
      FinTrans ft = new FinTrans ();
      TransThread tt1 = new TransThread (ft, "Deposit Thread");
      TransThread tt2 = new TransThread (ft, "Withdrawal Thread");
      tt1.start ();
      tt2.start ();
   }
}
class FinTrans
{
   public static String transName;
   public static double amount;
}
class TransThread extends Thread
{
   private FinTrans ft;
   TransThread (FinTrans ft, String name)
```

```
  {
     super (name); // Save thread's name
     this.ft = ft; // Save reference to financial transaction object
  }
  public void run ()
  {
     for (int i = 0; i < 100; i++)
     {
          if (getName ().equals ("Deposit Thread"))
          {
              synchronized (ft)
              {
                 ft.transName = "Deposit";
                 try
                 {
                    Thread.sleep ((int) (Math.random () * 1000));
                 }
                 catch (InterruptedException e)
                 {
                 }
                 ft.amount = 2000.0;
                 System.out.println (ft.transName + " " + ft.amount);
              }
          }
          else
          {
              synchronized (ft)
              {
                 ft.transName = "Withdrawal";
                 try
                 {
                    Thread.sleep ((int) (Math.random () * 1000));
                 }
```

```
                  catch (InterruptedException e)
                  {
                  }
                  ft.amount = 250.0;
                  System.out.println (ft.transName + " " + ft.amount);
               }
           }
      }
   }
}
```

Look carefully at SynchronizationDemo1; the run() method contains two critical code sections sandwiched betweensynchronized (ft) { and }. Each of the deposit and withdrawal threads must acquire the lock that associates with the FinTransobject that ft references before either thread can enter its critical code section. If, for example, the deposit thread is in its critical code section and the withdrawal thread wants to enter its own critical code section, the withdrawal thread attempts to acquire the lock. Because the deposit thread holds that lock while it executes within its critical code section, the JVM forces the withdrawal thread to wait until the deposit thread executes that critical code section and releases the lock. (When execution leaves the critical code section, the lock releases automatically.)

Tip: When you need to determine if a thread holds a given object's associated lock, call Thread's static boolean holdsLock(Object o) method. That method returns a Boolean true value if the thread calling that method holds the lock associated with the object that o references; otherwise, false returns. For example, if you were to place System.out.println (Thread.holdsLock (ft));at the end of SynchronizationDemo1's main() method, holdsLock() would return false. False would return because the main thread executing themain() method does not use the synchronization mechanism to acquire any lock. However, if you were to place System.out.println (Thread.holdsLock (ft)); in either of run()'s synchronized (ft) statements, holdsLock() would return true because either the deposit thread or the withdrawal thread had to acquire the lock associated with the FinTrans object that ft references before that thread could enter its critical code section.

Synchronized methods

You can employ synchronized statements throughout your program's source code. However, you might run into situations where excessive use of such statements leads to inefficient code. For example, suppose your program contains a method with two successive synchronized statements that each attempt to acquire the same common object's associated lock. Because acquiring and releasing the object's lock eats up time, repeated calls (in a loop) to that method can degrade the program's performance. Each time a call is made to that method, it must acquire and release two locks. The greater the number of lock acquisitions and releases, the more time the program spends acquiring and releasing the locks. To get around that problem, you might consider using a synchronized method.

A synchronized method is either an instance or class method whose header includes the synchronized keyword. For example: synchronized void print (String s). When you synchronize an entire instance method, a thread must acquire the lock associated with the object on which the method call occurs. For example, given anft.update("Deposit", 2000.0); instance method call, and assuming that update() is synchronized, a thread must acquire the lock associated with the object that ftreferences. To see a synchronized method version of theSynchronizationDemo1 source code, check out Listing 3:

Listing 3. SynchronizationDemo2.java

```
// SynchronizationDemo2.java
class SynchronizationDemo2
{
   public static void main (String [] args)
   {
      FinTrans ft = new FinTrans ();
      TransThread tt1 = new TransThread (ft, "Deposit Thread");
      TransThread tt2 = new TransThread (ft, "Withdrawal Thread");
      tt1.start ();
      tt2.start ();
   }
}
class FinTrans
{
   private String transName;
   private double amount;
   synchronized void update (String transName, double amount)
   {
      this.transName = transName;
      this.amount = amount;
      System.out.println (this.transName + " " + this.amount);
   }
}
class TransThread extends Thread
{
   private FinTrans ft;
   TransThread (FinTrans ft, String name)
   {
      super (name); // Save thread's name
      this.ft = ft; // Save reference to financial transaction object
   }
   public void run ()
   {
      for (int i = 0; i < 100; i++)
          if (getName ().equals ("Deposit Thread"))
             ft.update ("Deposit", 2000.0);
      else
         ft.update ("Withdrawal", 250.0);
  }
}
```

Though slightly more compact than Listing 2, Listing 3 accomplishes the same purpose. If the deposit thread calls the`update()` method, the JVM checks to see if the withdrawal thread has acquired the lock associated with the object that`ft` references. If so, the deposit thread waits. Otherwise, that thread enters the critical code section.

`SynchronizationDemo2` demonstrates a synchronized instance method. However, you can also synchronize *class* methods. For example, the `java.util.Calendar` class declares a `public static synchronized Locale [] getAvailableLocales()`method. Because class methods have no concept of a `this`reference, from where does the class method acquire its lock? Class methods acquire their locks from class objects—each loaded class associates with a `Class`object, from which the loaded class's class methods obtain their locks. I refer to such locks as *class locks.*

Caution: Don't synchronize a thread object's `run()` method because situations arise where multiple threads need to execute `run()`. Because those threads attempt to synchronize on the same object, only one thread at a time can execute `run()`. As a result, each thread must wait for the previous thread to terminate before it can access `run()`.

Some programs intermix synchronized instance methods and synchronized class methods. To help you understand what happens in programs where synchronized class methods call synchronized instance methods and vice-versa (via object references), keep the following two points in mind:

1. Object locks and class locks do not relate to each other. They are different entities. You acquire and release each lock independently. A synchronized instance method calling a synchronized class method acquires both locks. First, the synchronized instance method acquires its object's object lock. Second, that method acquires the synchronized class method's class lock.

2. Synchronized class methods can call an object's synchronized methods or use the object to lock a synchronized block. In that scenario, a thread initially acquires the synchronized class method's class lock and subsequently acquires the object's object lock. Hence, a synchronized class method calling a synchronized instance method also acquires two locks.

The following code fragment illustrates the second point:

```
class LockTypes
{
  // Object lock acquired just before execution passes into 
instanceMethod()
  synchronized void instanceMethod ()
  {
    // Object lock released as thread exits instanceMethod()
  }
  // Class lock acquired just before execution passes into classMethod()
  synchronized static void classMethod (LockTypes lt)
  {
    lt.instanceMethod ();
    // Object lock acquired just before critical code section executes

    synchronized (lt)
    {
      // Critical code section
      // Object lock released as thread exits critical code section
    }
    // Class lock released as thread exits classMethod()
  }
}
```

The code fragment demonstrates synchronized class methodclassMethod() calling synchronized instance methodinstanceMethod(). By reading the comments, you see thatclassMethod() first acquires its class lock and then acquires the object lock associated with the LockTypes object that ltreferences.

Two problems with the synchronization mechanism

Despite its simplicity, developers often misuse Java's synchronization mechanism, which causes problems ranging from no synchronization to deadlock. This section examines these problems and provides a pair of recommendations for avoiding them.

Note: A third problem related to the synchronization mechanism is the time cost associated with lock acquisition and release. In other words, it takes time for a thread to acquire or release a lock. When acquiring/releasing a lock in a loop, individual time costs add up, which can degrade performance. For older JVMs, the lock-acquisition time cost often results in significant performance penalties. Fortunately, Sun Microsystems' HotSpot JVM (which ships with Sun's Java 2 Platform, Standard Edition (J2SE) SDK) offers fast lock acquisition and release, greatly reducing this problem's impact.

No synchronization

After a thread voluntarily or involuntarily (through an exception) exits a critical code section, it releases a lock so another thread can gain entry. Suppose two threads want to enter the same critical code section. To prevent both threads from entering that critical code section simultaneously, each thread must attempt to acquire the same lock. If each thread attempts to acquire a different lock and succeeds, both threads enter the critical code section; neither thread has to wait for the other thread to release its lock because the other thread acquires a different lock. The end result: no synchronization, as demonstrated in Listing 4:

Listing 4. NoSynchronizationDemo.java

```
// NoSynchronizationDemo.java
class NoSynchronizationDemo
{
   public static void main (String [] args)
   {
```

```java
      FinTrans ft = new FinTrans ();
      TransThread tt1 = new TransThread (ft, "Deposit Thread");
      TransThread tt2 = new TransThread (ft, "Withdrawal Thread");
      tt1.start ();
      tt2.start ();
   }
}
class FinTrans
{
   public static String transName;
   public static double amount;
}
class TransThread extends Thread
{
   private FinTrans ft;
   TransThread (FinTrans ft, String name)
   {
      super (name); // Save thread's name
      this.ft = ft; // Save reference to financial transaction object
   }
   public void run ()
   {
      for (int i = 0; i < 100; i++)
      {
           if (getName ().equals ("Deposit Thread"))
           {
               synchronized (this)
               {
                  ft.transName = "Deposit";
                  try
                  {
                     Thread.sleep ((int) (Math.random () * 1000));
                  }
```

```
                 catch (InterruptedException e)
                 {
                 }
                 ft.amount = 2000.0;
                 System.out.println (ft.transName + " " + ft.amount);
              }
          }
          else
          {
              synchronized (this)
              {
                 ft.transName = "Withdrawal";
                 try
                 {
                    Thread.sleep ((int) (Math.random () * 1000));
                 }
                 catch (InterruptedException e)
                 {
                 }
                 ft.amount = 250.0;
                 System.out.println (ft.transName + " " + ft.amount);
              }
          }
     }
   }
}
```

When you run NoSynchronizationDemo, you will see output resembling the following excerpt:

```
Withdrawal 250.0
Withdrawal 2000.0
Deposit 250.0
```

```
Withdrawal 2000.0
Deposit 2000.0
```

Despite the use of synchronized statements, no synchronization takes place. Why? Examine synchronized (this). Because keyword this refers to the current object, the deposit thread attempts to acquire the lock associated with the TransThreadobject whose reference initially assigns to tt1 (in the main()method). Similarly, the withdrawal thread attempts to acquire the lock associated with the TransThread object whose reference initially assigns to tt2. We have two differentTransThread objects, and each thread attempts to acquire the lock associated with its respective TransThread object before entering its own critical code section. Because the threads acquire different locks, both threads can be in their own critical code sections at the same time. The result is no synchronization.

Tip: To avoid a no-synchronization scenario, choose an object common to all relevant threads. That way, those threads compete to acquire the same object's lock, and only one thread at a time can enter the associated critical code section.

Deadlock

In some programs, the following scenario might occur: Thread A acquires a lock that thread B needs before thread B can enter B's critical code section. Similarly, thread B acquires a lock that thread A needs before thread A can enter A's critical code section. Because neither thread has the lock it needs, each thread must wait to acquire its lock. Furthermore, because neither thread can proceed, neither thread can release the other thread's lock, and program execution freezes. This behavior is known as *deadlock,* which Listing 5 demonstrates:

Listing 5. DeadlockDemo.java

```
// DeadlockDemo.java
class DeadlockDemo
{
   public static void main (String [] args)
   {
      FinTrans ft = new FinTrans ();
      TransThread tt1 = new TransThread (ft, "Deposit Thread");
      TransThread tt2 = new TransThread (ft, "Withdrawal Thread");
      tt1.start ();
      tt2.start ();
   }
}
class FinTrans
{
   public static String transName;
   public static double amount;
}
class TransThread extends Thread
{
   private FinTrans ft;
   private static String anotherSharedLock = "";
   TransThread (FinTrans ft, String name)
   {
      super (name); // Save thread's name
      this.ft = ft; // Save reference to financial transaction object
   }
   public void run ()
   {
      for (int i = 0; i < 100; i++)
      {
           if (getName ().equals ("Deposit Thread"))
           {
               synchronized (ft)
               {
                  synchronized (anotherSharedLock)
                  {
                     ft.transName = "Deposit";
                     try
```

```
            {
              Thread.sleep ((int) (Math.random () * 1000));
            }
            catch (InterruptedException e)
            {
            }
            ft.amount = 2000.0;
            System.out.println (ft.transName + " " + ft.amount);
          }
        }
      }
      else
      {
        synchronized (anotherSharedLock)
        {
          synchronized (ft)
          {
            ft.transName = "Withdrawal";
            try
            {
              Thread.sleep ((int) (Math.random () * 1000));
            }
            catch (InterruptedException e)
            {
            }
            ft.amount = 250.0;
            System.out.println (ft.transName + " " + ft.amount);
          }
        }
      }
    }
```

```
   }
}
```

If you run DeadlockDemo, you will probably see only a single line of output before the application freezes. To unfreezeDeadlockDemo, press Ctrl-C (assuming you are using Sun's SDK 1.4 toolkit at a Windows command prompt).

What causes the deadlock? Look carefully at the source code; the deposit thread must acquire two locks before it can enter its innermost critical code section. The outer lock associates with the FinTrans object that ft references, and the inner lock associates with the String object that anotherSharedLockreferences. Similarly, the withdrawal thread must acquire two locks before it can enter its own innermost critical code section. The outer lock associates with the String object thatanotherSharedLock references, and the inner lock associates with the FinTrans object that ft references. Suppose both threads' execution orders are such that each thread acquires its outer lock. Thus, the deposit thread acquires its FinTranslock, and the withdrawal thread acquires its String lock. Now that both threads possess their outer locks, they are in their appropriate outer critical code section. Both threads then attempt to acquire the inner locks, so they can enter the appropriate inner critical code sections.

The deposit thread attempts to acquire the lock associated with theanotherSharedLock-referenced object. However, the deposit thread must wait because the withdrawal thread holds that lock. Similarly, the withdrawal thread attempts to acquire the lock associated with the ft-referenced object. But the withdraw thread cannot acquire that lock because the deposit thread (which is waiting) holds it. Therefore, the withdrawal thread must also wait. Neither thread can proceed because neither thread releases the lock it holds. And neither thread can release the lock it holds because each thread is waiting. Each thread deadlocks, and the program freezes.

Tip: To avoid deadlock, carefully analyze your source code for situations where threads might attempt to acquire each others' locks, such as when a synchronized method calls another synchronized method. You must do that because a JVM cannot detect or prevent deadlock.

Review

To achieve strong performance with threads, you will encounter situations where your multithreaded programs need to serialize access to critical code sections. Known as synchronization, that activity prevents inconsistencies resulting in strange program behavior. You can use eithersynchronized statements to guard portions of a method, or synchronize the entire method. But comb your code carefully for glitches that can result in failed synchronization or deadlocks.

In Part 3, I will introduce you to thread scheduling, thread interruption, and Java's wait/notify mechanism.

Jeff Friesen has been involved with computers for the past 20 years. He holds a degree in computer science and has worked with many computer languages. Jeff has also taught introductory Java programming at the college level. In addition to writing for JavaWorld, he has written his own Java book for beginners— Java 2 by Example, Second Edition (Que Publishing, 2001; ISBN: 0789725932)—and helped write Using Java 2 Platform, Special Edition (Que Publishing, 2001; ISBN: 0789724685). Jeff goes by the nickname Java Jeff (or JavaJeff). To see what he's working on, check out his Website at http://www.javajeff.com.

Understanding Java threads, Part 3: Thread scheduling and wait/notify

Learn about the mechanisms that help you set and manage thread priority

This month, I continue my four-part introduction to Java threads by focusing on thread scheduling, the wait/notify mechanism, and thread interruption. You'll investigate how either a JVM or an operating-system thread scheduler chooses the next thread for execution. As you'll discover, priority is important to a thread scheduler's choice. You'll examine how a thread waits until it receives notification from another thread before it continues execution and learn how to use the wait/notify mechanism for coordinating the execution of two threads in a producer-consumer relationship. Finally, you'll learn how to prematurely awaken either a sleeping or a waiting thread for thread termination or other tasks. I'll also teach you how a thread that is neither sleeping nor waiting detects an interruption request from another thread.

Thread scheduling

In an idealized world, all program threads would have their own processors on which to run. Until the time comes when computers have thousands or millions of processors, threads often must share one or more processors. Either the JVM or the underlying platform's operating system deciphers how to share the processor resource among threads—a task known as *thread scheduling*. That portion of the JVM or operating system that performs thread scheduling is a *thread scheduler*.

Note: To simplify my thread scheduling discussion, I focus on thread scheduling in the context of a single processor. You can extrapolate this discussion to multiple processors; I leave that task to you.

Remember two important points about thread scheduling:

1. Java does not force a VM to schedule threads in a specific manner or contain a thread scheduler. That implies platform-dependent thread scheduling. Therefore, you must exercise care when writing a Java program whose behavior depends on how threads are scheduled and must operate consistently across different platforms.
2. Fortunately, when writing Java programs, you need to think about how Java schedules threads only when at least one of your program's threads heavily uses the processor for long time periods and intermediate results of that thread's execution prove important. For example, an applet contains a thread that dynamically creates an image. Periodically, you want the painting thread to draw that image's current contents so the user can see how the image progresses. To ensure that the calculation thread does not monopolize the processor, consider thread scheduling.

Examine a program that creates two processor-intensive threads:

Listing 1. SchedDemo.java

```
// SchedDemo.java
class SchedDemo
{
   public static void main (String [] args)
   {
      new CalcThread ("CalcThread A").start ();
      new CalcThread ("CalcThread B").start ();
   }
}
class CalcThread extends Thread
{
   CalcThread (String name)
   {
      // Pass name to Thread layer.
      super (name);
   }
   double calcPI ()
   {
      boolean negative = true;
      double pi = 0.0;
      for (int i = 3; i < 100000; i += 2)
      {
           if (negative)
               pi -= (1.0 / i);
           else
               pi += (1.0 / i);
           negative = !negative;
      }
      pi += 1.0;
      pi *= 4.0;
      return pi;
   }
   public void run ()
   {
      for (int i = 0; i < 5; i++)
         System.out.println (getName () + ": " + calcPI ());
   }
}
```

SchedDemo creates two threads that each calculate the value of pi (five times) and print each result. Depending upon how your JVM implementation schedules threads, you might see output resembling the following:

```
CalcThread A: 3.1415726535897894
CalcThread B: 3.1415726535897894
CalcThread A: 3.1415726535897894
CalcThread A: 3.1415726535897894
CalcThread B: 3.1415726535897894
CalcThread A: 3.1415726535897894
CalcThread A: 3.1415726535897894
CalcThread B: 3.1415726535897894
CalcThread B: 3.1415726535897894
CalcThread B: 3.1415726535897894
```

According to the above output, the thread scheduler shares the processor between both threads. However, you could see output similar to this:

```
CalcThread A: 3.1415726535897894
CalcThread A: 3.1415726535897894
CalcThread A: 3.1415726535897894
CalcThread A: 3.1415726535897894
CalcThread A: 3.1415726535897894
CalcThread B: 3.1415726535897894
CalcThread B: 3.1415726535897894
CalcThread B: 3.1415726535897894
CalcThread B: 3.1415726535897894
CalcThread B: 3.1415726535897894
```

The above output shows the thread scheduler favoring one thread over another. The two outputs above illustrate two general categories of thread schedulers: green and native. I'll explore their behavioral differences in upcoming sections. While discussing each category, I refer to *thread states,* of which there are four:

1. **Initial state:** A program has created a thread's thread object, but the thread does not yet exist because the thread object's `start()` method has not yet been called.
2. **Runnable state:** This is a thread's default state. After the call to `start()` completes, a thread becomes runnable whether or not that thread is running, that is, using the processor. Although many threads might be runnable, only one currently runs. Thread schedulers determine which runnable thread to assign to the processor.
3. **Blocked state:** When a thread executes the `sleep()`,`wait()`, or `join()` methods, when a thread attempts to read data not yet available from a network, and when a thread waits to acquire a lock, that thread is in the blocked state: it is neither running nor in a position to run. (You can probably think of other times when a thread would wait for something to happen.) When a blocked thread unblocks, that thread moves to the runnable state.
4. **Terminating state:** Once execution leaves a thread's `run()` method, that thread is in the terminating state. In other words, the thread ceases to exist.

How does the thread scheduler choose which runnable thread to run? I begin answering that question while discussing green thread scheduling. I finish the answer while discussing native thread scheduling.

Green thread scheduling

Not all operating systems, the ancient Microsoft Windows 3.1 perating system, for example, support threads. For such systems, Sun Microsystems can design a JVM that divides its sole thread of execution into multiple threads. The JVM (not the underlying platform's operating system) supplies the threading logic and contains the thread scheduler. JVM threads are *green threads,* or *user threads*.

A JVM's thread scheduler schedules green threads according to *priority*—a thread's relative importance, which you express as an integer from a well-defined range of values. Typically, a JVM's thread scheduler chooses the highest-priority thread and allows that thread to run until it either terminates or blocks. At that time, the thread scheduler chooses a thread of the next highest priority. That thread (usually) runs until it terminates or blocks. If, while a thread runs, a thread of higher priority unblocks (perhaps the higher-priority thread's sleep time expired), the thread scheduler *preempts,* or interrupts, the lower-priority thread and assigns the unblocked higher-priority thread to the processor.

Note: A runnable thread with the highest priority will not always run. Here's the *Java Language Specification's* take on priority:

Every thread has a *priority.* When there is competition for processing resources, threads with higher priority are generally executed in preference to threads with lower priority. Such preference is not, however, a guarantee that the highest priority thread will always be running, and thread priorities cannot be used to reliably implement mutual exclusion.

That admission says much about the implementation of green thread JVMs. Those JVMs cannot afford to let threads block because that would tie up the JVM's sole thread of execution. Therefore, when a thread must block, such as when that thread is reading data slow to arrive from a file, the JVM might stop the thread's execution and use a polling mechanism to determine when data arrives. While the thread remains stopped, the JVM's thread scheduler might schedule a lower-priority thread to run. Suppose data arrives while the lower-priority thread is running. Although the higher-priority thread should run as soon as data arrives, that doesn't happen until the JVM next polls the operating system and discovers the arrival. Hence, the lower-priority thread runs even though the higher-priority thread should run. ou need to worry about this situation only when you need real-time behavior from Java. But then Java is not a real-time operating system, so why worry?

To understand which runnable green thread becomes the currently running green thread, consider the following. Suppose your application consists of three threads: the main thread that runs the `main()` method, a calculation thread, and a thread that reads keyboard input. When there is no keyboard input, the reading thread blocks. Assume the reading thread has the highest priority and the calculation thread has the lowest priority. (For simplicity's sake, also assume that no other internal JVM threads are available.) Figure 1 illustrates the execution of these three threads.

Figure 1. Thread-scheduling diagram for different priority threads

At time T0, the main thread starts running. At time T1, the main thread starts the calculation thread. Because the calculation thread has a lower priority than the main thread, the calculation thread waits for the processor. At time T2, the main thread starts the reading thread. Because the reading thread has a higher priority than the main thread, the main thread waits for the processor while the reading thread runs. At time T3, the reading thread blocks and the main thread runs. At time T4, the reading thread unblocks and runs; the main thread waits. Finally, at time T5, the reading thread blocks and the main thread runs. This alternation in execution between the reading and main threads continues as long as the program runs. The calculation thread never runs because it has the lowest priority and thus starves for processor attention, a situation known as *processor starvation*.

We can alter this scenario by giving the calculation thread the same priority as the main thread. Figure 2 shows the result, beginning with time T2. (Prior to T2, Figure 2 is identical to Figure 1.)

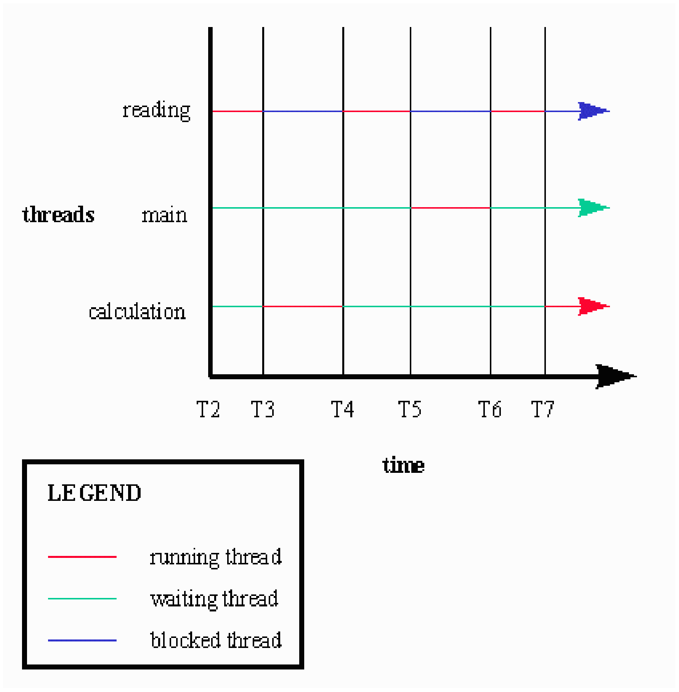

Figure 2. Thread-scheduling diagram for equal priority main and calculation threads, and a different priority reading thread

At time T2, the reading thread runs while the main and calculation threads wait for the processor. At time T3, the reading thread blocks and the calculation thread runs, because the main thread ran just before the reading thread. At time T4, the reading thread unblocks and runs; the main and calculation threads wait. At time T5, the reading thread blocks and the main thread runs, because the calculation thread ran just before the reading thread. This alternation in execution between the main and calculation threads continues as long as the program runs and depends on the higher-priority thread running and blocking.

We must consider one last item in green thread scheduling. What happens when a lower-priority thread holds a lock that a higher-priority thread requires? The higher-priority thread blocks because it cannot get the lock, which implies that the higher-priority thread effectively has the same priority as the lower-priority thread. For example, a priority 6 thread attempts to acquire a lock that a priority 3 thread holds. Because the priority 6 thread must wait until it can acquire the lock, the priority 6 thread ends up with a 3 priority—a phenomenon known as *priority inversion*.

Priority inversion can greatly delay the execution of a higher-priority thread. For example, suppose you have three threads with priorities of 3, 4, and 9. Priority 3 thread is running and the other threads are blocked. Assume that the priority 3 thread grabs a lock, and the priority 4 thread unblocks. The priority 4 thread becomes the currently running thread. Because the priority 9 thread requires the lock, it continues to wait until the priority 3 thread releases the lock. However, the priority 3 thread cannot release the lock until the priority 4 thread blocks or terminates. As a result, the priority 9 thread delays its execution.

A JVM's thread scheduler usually solves the priority inversion problem through *priority inheritance:* The thread scheduler silently raises the priority of the thread holding the lock when a higher-priority thread requests the lock. As a result, both the thread holding the lock and the thread waiting for the lock temporarily have equal priorities. Using the previous example, the priority 3 thread (holding the lock) would temporarily become a priority 9 thread as soon as the priority 9 thread attempts to acquire the lock and is blocked. As a result, the priority 9 thread (holding the lock) would become the currently running thread (even when the priority 4 thread unblocks). The priority 9 thread would finish its execution and release the lock, allowing the waiting priority 9 thread to acquire the lock and continue execution. The priority 4 thread would lack the chance to become the currently running thread. Once the thread with its silently raised priority releases the lock, the thread scheduler restores the thread's priority to its original priority. Therefore, the thread scheduler would restore the priority 9 thread previously holding the lock to priority 3, once it releases the lock. Thus, priority inheritance ensures that a lower-priority thread holding a lock is not preempted by a thread whose priority exceeds the lock-holding thread's priority but is less than the priority of the thread waiting for the lock to release.

Native thread scheduling

Most JVMs rely on the underlying operating system (such as Linux or Microsoft Windows XP) to provide a thread scheduler. When an operating system handles thread scheduling, the threads are *native threads*. As with green thread scheduling, priority proves important to native thread scheduling: higher-priority threads typically preempt lower-priority threads. But native thread schedulers often introduce an additional detail: *time-slicing.*

Note: Some green thread schedulers also support time-slicing. And many native thread schedulers support priority inheritance. As a result, green thread schedulers and native thread schedulers normally differ only in their thread scheduler's source: JVM or operating system.

Native thread schedulers typically introduce time-slicing to prevent processor starvation of equal-priority threads. The idea is to give each equal-priority thread the same amount of time, known as a *quantum.* A timer tracks each quantum's remaining time and alerts the thread scheduler when the quantum expires. The thread scheduler then schedules another equal-priority thread to run, unless a higher-priority thread unblocks.

Time-slicing complicates the writing of those platform-independent multithreaded programs that depend on consistent thread scheduling, because not all thread schedulers implement time-slicing. Without time-slicing, an equal-priority runnable thread will keep running (assuming it is the currently running thread) until that thread terminates, blocks, or is replaced by a higher-priority thread. Thus, the thread scheduler fails to give all equal-priority runnable threads the chance to run. Though complicated, time-slicing does not prevent you from writing platform-independent multithreaded programs. The`setPriority(int priority)` and `yield()` methods influence thread scheduling so a program behaves fairly consistently (as far as thread scheduling is concerned) across platforms.

Note: To prevent lower-priority threads from starving, some thread schedulers, such as Windows schedulers, give temporary priority boosts to threads that have not run in a long time. When the thread runs, that priority boost decays. Thread schedulers still give higher-priority threads preference over lower-priority threads, but at least all threads receive a chance to run.

Schedule with the setPriority(int priority) method

Enough theory! Let's learn how to influence thread scheduling at the source code level. One way is to use Thread's void setPriority(int priority); method. When called,setPriority(int priority) sets the priority of a thread associated with the specified thread object (as inthd.setPriority (7);), to priority. If priority is not within the range of priorities that Thread's MIN_PRIORITY andMAX_PRIORITY constants specify, setPriority(int priority)throws an IllegalArgumentException object.

Note: If you call setPriority(int priority) with a priority value that exceeds the maximum allowed priority for the respective thread's thread group, this method silently lowers the priority value to match the thread group's maximum priority. (I'll discuss thread groups next month.)

When you must determine a thread's current priority, call Thread's int getPriority() method, via that thread's thread object. The getPriority() method returns a value between MIN_PRIORITY (1) and MAX_PRIORITY (10). One of those values might be 5—the value that assigns to the NORM_PRIORITY constant, which represents a thread's default priority.

The setPriority(int priority) method proves useful in preventing processor starvation. For example, suppose your program consists of a thread that blocks and a calculation thread that doesn't block. By assigning a higher priority to the thread that blocks, you ensure the calculation thread will not starve the blocking thread. Because the blocking thread periodically blocks, the calculation thread will not starve. Of course, we assume the thread scheduler does not support time-slicing. If the thread scheduler does supports time-slicing, you will probably see no difference between calls tosetPriority(int priority) and no calls to that method, depending on what the affected threads are doing. However, you will at least ensure that your code ports across thread schedulers. To demonstrate setPriority(int priority), I wrote PriorityDemo:

Listing 2. PriorityDemo.java

```
// PriorityDemo.java
class PriorityDemo
{
   public static void main (String [] args)
   {
      BlockingThread bt = new BlockingThread ();
      bt.setPriority (Thread.NORM_PRIORITY + 1);
      CalculatingThread ct = new CalculatingThread ();
      bt.start ();
      ct.start ();
      try
      {
          Thread.sleep (10000);
      }
      catch (InterruptedException e)
      {
      }
      bt.setFinished (true);
      ct.setFinished (true);
   }
}
class BlockingThread extends Thread
{
   private boolean finished = false;
   public void run ()
   {
      while (!finished)
      {
         try
         {
            int i;
            do
            {
               i = System.in.read ();
               System.out.print (i + " ");
            }
            while (i != '\n');
            System.out.print ('\n');
         }
         catch (java.io.IOException e)
         {
         }
      }
   }
   public void setFinished (boolean f)
   {
      finished = f;
   }
}
class CalculatingThread extends Thread
{
   private boolean finished = false;
   public void run ()
   {
      int sum = 0;
      while (!finished)
         sum++;
   }
   public void setFinished (boolean f)
   {
      finished = f;
```

```
  }
}
```
PriorityDemo has a blocking thread and a calculating thread in addition to the main thread. Suppose you ran this program on a platform where the thread scheduler did not support time-slicing. What would happen? Consider two scenarios:

1. **Assume no** bt.setPriority (Thread.NORM_PRIORITY + 1);**method call:** The main thread runs until it sleeps. At that point, assume the thread scheduler starts the blocking thread. That thread runs until it calls System.in.read(), which causes the blocking thread to block. The thread scheduler then assigns the calculating thread to the processor (assuming the main thread has not yet unblocked from its sleep). Because the blocking, main, and calculating threads all have the same priority, the calculating thread continues to run in an infinite loop.
2. **Assume** bt.setPriority (Thread.NORM_PRIORITY + 1); **method call:** The blocking thread gets the processor once it unblocks. Then assume that the thread scheduler arbitrarily chooses either the calculating thread or the main thread (assuming the main thread has unblocked from its sleep) when the blocking thread blocks upon its next call to System.in.read(). As a result, the program should eventually end. If the thread scheduler always picks the calculating thread over the main thread, consider boosting the main thread's priority to ensure eventual termination.

If you run PriorityDemo with time-slicing, you have the following two scenarios:

1. **Assume no** bt.setPriority (Thread.NORM_PRIORITY + 1);**method call:** Time-slicing ensures that all equal-priority threads have a chance to run. The program eventually terminates.
2. **Assume** bt.setPriority (Thread.NORM_PRIORITY + 1);**method call:** The blocking thread will run more often because of its higher priority. But because it blocks periodically, the blocking thread does not cause significant

disruption to the calculation and main threads. The program eventually terminates.

Schedule with the yield() method

Many developers prefer the alternative to the setPriority(int priority) method, Thread's static void yield();, because of its simplicity. When the currently running thread calls Thread.yield ();, the thread scheduler keeps the currently running thread in the runnable state, but (usually) picks another thread of equal priority to be the currently running thread, unless a higher-priority thread has just been made runnable, in which case the higher-priority thread becomes the currently running thread. If you have no higher-priority thread and no other equal-priority threads, the thread scheduler immediately reschedules the thread calling yield() as the currently running thread. Furthermore, when the thread scheduler picks an equal-priority thread, the picked thread might be the thread that called yield()—which means that yield()accomplishes nothing except delay. This behavior typically happens under a time-slicing thread scheduler. Listing 3 demonstrates the yield() method:

Listing 3. YieldDemo.java

```
// YieldDemo.java
class YieldDemo extends Thread
{
   static boolean finished = false;
   static int sum = 0;
   public static void main (String [] args)
   {
      new YieldDemo ().start ();
      for (int i = 1; i <= 50000; i++)
      {
           sum++;
           if (args.length == 0)
```

```
            Thread.yield ();
      }
      finished = true;
   }
   public void run ()
   {
      while (!finished)
         System.out.println ("sum = " + sum);
   }
}
```

From a logical perspective, YieldDemo's main thread starts a new thread (of the same NORM_PRIORITY priority) that repeatedly outputs the value of instance field sum until the value of instance field finished is true. After starting that thread, the main thread enters a loop that repeatedly increments sum's value. If no arguments pass to YieldDemo on the command line, the main thread calls Thread.yield ();after each increment. Otherwise, no call is made to that method. Once the loop ends, the main thread assigns true tofinished, so the other thread will terminate. After that, the main thread terminates.

Now that you know what YieldDemo should accomplish, what kind of behavior can you expect? That answer depends on whether the thread scheduler uses time-slicing and whether calls are made to yield(). We have four scenarios to consider:

1. **No time-slicing and no yield() calls:** The main thread runs to completion. The thread scheduler won't schedule the output thread once the main thread exits. Therefore, you see no output.
2. **No time-slicing and yield() calls:** After the first yield()call, the output thread runs forever because finishedcontains false. You should see the same sum value printed repeatedly in an endless loop (because the main thread does not run and increment sum). To counteract this problem, the output thread should also call yield() during each while loop iteration.

3. **Time-slicing and no `yield()` calls:** Both threads have approximately equal amounts of time to run. However, you will probably see very few lines of output because each `System.out.println ("sum =" + sum);` method call occupies a greater portion of a quantum than a`sum++;` statement. (Many processor cycles are required to send output to the standard output device, while (relatively) few processor cycles are necessary for incrementing an integer variable.) Because the main thread accomplishes more work by the end of a quantum than the output thread and because that activity brings the program closer to the end, you observe fewer lines of output.
4. **Time-slicing and `yield()` calls:** Because the main thread yields each time it increments `sum`, the main thread completes less work during a quantum. Because of that, and because the output thread receives additional quantums, you see many more output lines.

Note: Should you call `setPriority(int priority)` or `yield()`? Both methods affect threads similarly. However, `setPriority(int priority)` offers flexibility, whereas `yield()` offers simplicity. Also, `yield()` might immediately reschedule the yielding thread, which accomplishes nothing. I prefer`setPriority(int priority)`, but you must make your own choice.

To support the wait/notify mechanism, `Object` declares the `void wait();` method (to force a thread to wait) and the `void notify();` method (to notify a waiting thread that it can continue execution). Because every object inherits `Object`'s methods, `wait()` and `notify()` are available to all objects. Both methods share a common feature: they are synchronized. A thread must call `wait()` or `notify()` from within a synchronized context because of a race condition inherent to the wait/notify mechanism. Here is how that race condition works:

1. Thread A tests a condition and discovers it must wait.
2. Thread B sets the condition and calls `notify()` to inform A to resume execution. Because A is not yet waiting, nothing happens.
3. Thread A waits, by calling `wait()`.

4. Because of the prior notify() call, A waits indefinitely.

To solve the race condition, Java requires a thread to enter a synchronized context before it calls either wait() or notify(). Furthermore, the thread that calls wait() (the waiting thread) and the thread that calls notify() (the notification thread) must compete for the same lock. Either thread must call wait()or notify() via the same object on which they enter their synchronized contexts because wait() tightly integrates with the lock. Prior to waiting, a thread executingwait() releases the lock, which allows the notification thread to enter its synchronized context to set the condition and notify the waiting thread. Once notification arrives, the JVM wakens the waiting thread, which then tries to reacquire the lock. Upon successfully reacquiring the lock, the previously waiting thread returns from wait(). Confused? The following code fragment offers clarification:

```
// Condition variable initialized to false to indicate condition has not
occurred.
boolean conditionVar = false;
// Object whose lock threads synchronize on.
Object lockObject = new Object ();
// Thread A waiting for condition to occur...
synchronized (lockObject)
{
  while (!conditionVar)
    try
    {
      lockObject.wait ();
    }
    catch (InterruptedException e) {}
}
// ... some other method
// Thread B notifying waiting thread that condition has now occurred...
synchronized (lockObject)
{
  conditionVar = true;
  lockObject.notify ();
}
```

The wait/notify mechanism

As you learned last month, each object's associated lock and waiting area allow the JVM to synchronize access to critical code sections. For example: When thread X tries to acquire a lock before entering a synchronized context guarding a critical code section from concurrent thread access, and thread Y is executing within that context (and holding the lock), the JVM places X in a waiting area. When Y exits the synchronized context (and releases the lock), the JVM removes X from the waiting area, assigns the lock to X, and allows that thread to enter the synchronized context. In addition to its use in synchronization, the waiting area serves a second purpose: it is part of the wait/notify mechanism, the mechanism that coordinates multiple threads' activities.

The idea behind the wait/notify mechanism is this: A thread forces itself to wait for some kind of *condition,* a prerequisite for continued execution, to exist before it continues. The waiting thread assumes that some other thread will create that condition and then notify the waiting thread to continue execution. Typically, a thread examines the contents of a *condition variable*—a Boolean variable that determines whether a thread will wait—to confirm that a condition does not exist. If a condition does not exist, the thread waits in an object's waiting area. Later, another thread will set the condition by modifying the condition variable's contents and then notifying the waiting thread that the condition now exists and the waiting thread can continue execution.

Tip: Think of a condition as the reason one thread waits and another thread notifies the waiting thread.

the code fragment introduces condition variable `conditionVar`, which threads A and B use to test and set a condition, and lock variable `lockObject`, which both threads use for synchronization purposes. The condition variable initializes to false because the condition does not exist when the code starts execution. When A needs to wait for a condition, it enters a synchronized context (provided B is not in its synchronized context). Once inside its context, A executes a `while` loop statement whose Boolean expression tests `conditionVar`'s value and waits (if the value is false) by calling `wait()`. Notice that `lockObject` appears as part of `synchronized (lockObject)` and `lockObject.wait ();`—that is no coincidence. From inside `wait()`, A releases the lock associated with the object on which the call to `wait()` is made—the object associated with `lockObject` in the`lockObject.wait ();` method call. This allows B to enter its `synchronized (lockObject)` context, set `conditionVar` to true, and call `lockObject.notify ();` to notify A that the condition now exists. Upon receiving notification, A attempts to reacquire its lock. That does not occur until B leaves its synchronized context. Once A reacquires its lock, it returns from wait() and retests the condition variable. If this variable's value is true, A leaves its synchronized context.

Caution: If a call is made to `wait()` or `notify()` from outside a synchronized context, either call results in an `IllegalMonitorStateException`.

Apply wait/notify to the producer-consumer relationship

To demonstrate wait/notify's practicality, I introduce you to the producer-consumer relationship, which is common among multithreaded programs where two or more threads must coordinate their activities. The producer-consumer relationship demonstrates coordination between a pair of threads: a producer thread (producer) and a consumer thread (consumer). The producer produces some item that a consumer consumes. For example, a producer reads items from a file and passes those items to a consumer for processing. The producer cannot produce an item if no room is available for storing that item because the consumer has not finished consuming its item(s). Also, a consumer cannot consume an item that does not exist. Those restrictions prevent a producer from producing items that a consumer never receives for consumption, and prevents a consumer from attempting to consume more items than are available. Listing 4 shows the architecture of a producer-/consumer-oriented program:

Listing 4. ProdCons1.java

```
// ProdCons1.java
class ProdCons1
{
   public static void main (String [] args)
   {
      Shared s = new Shared ();
      new Producer (s).start ();
      new Consumer (s).start ();
   }
}
class Shared
{
   private char c = '\u0000';
   void setSharedChar (char c) { this.c = c; }
   char getSharedChar () { return c; }
}
class Producer extends Thread
{
   private Shared s;
   Producer (Shared s)
   {
      this.s = s;
   }
   public void run ()
   {
      for (char ch = 'A'; ch <= 'Z'; ch++)
      {
           try
           {
              Thread.sleep ((int) (Math.random () * 4000));
           }
```

```
            catch (InterruptedException e) {}
            s.setSharedChar (ch);
            System.out.println (ch + " produced by producer.");
      }
   }
}
class Consumer extends Thread
{
   private Shared s;
   Consumer (Shared s)
   {
      this.s = s;
   }
   public void run ()
   {
      char ch;
      do
      {
         try
         {
            Thread.sleep ((int) (Math.random () * 4000));
         }
         catch (InterruptedException e) {}
         ch = s.getSharedChar ();
         System.out.println (ch + " consumed by consumer.");
      }
      while (ch != 'Z');
   }
}
```
ProdCons1 creates producer and consumer threads. The producer passes uppercase letters individually to the consumer by calling s.setSharedChar (ch);. Once the producer finishes, that thread terminates. The consumer receives uppercase characters, from within a loop, by callings.getSharedChar (). The loop's duration

depends on that method's return value. When Z returns, the loop ends, and, thus, the producer informs the consumer when to finish. To make the code more representative of real-world programs, each thread sleeps for a random time period (up to four seconds) before either producing or consuming an item.

Because the code contains no race conditions, thesynchronized keyword is absent. Everything seems fine: the consumer consumes every character that the producer produces. In reality, some problems exist, which the following partial output from one invocation of this program shows:

```
consumed by consumer.
A produced by producer.
B produced by producer.
B consumed by consumer.
C produced by producer.
C consumed by consumer.
D produced by producer.
D consumed by consumer.
E produced by producer.
F produced by producer.
F consumed by consumer.
```

The first output line, consumed by consumer., shows the consumer trying to consume a nonexisting uppercase letter. The output also shows the producer producing a letter (A) that the consumer does not consume. Those problems do not result from lack of synchronization. Instead, the problems result from lack of coordination between the producer and the consumer. The producer should execute first, produce a single item, and then wait until it receives notification that the consumer has consumed the item. The consumer should wait until the producer produces an item. If both threads coordinate their activities in that manner, the aforementioned problems will disappear. Listing 5 demonstrates that coordination, which the wait/notify mechanism initiates:

Listing 5. ProdCons2.java

```
// ProdCons2.java

class ProdCons2
{
   public static void main (String [] args)
   {
      Shared s = new Shared ();
      new Producer (s).start ();
      new Consumer (s).start ();
   }
}

class Shared
{
   private char c = '\u0000';
   private boolean writeable = true;

   synchronized void setSharedChar (char c)
   {
      while (!writeable)
         try
         {
            wait ();
         }
         catch (InterruptedException e) {}

      this.c = c;
      writeable = false;
      notify ();
   }
   synchronized char getSharedChar ()
   {
      while (writeable)
         try
         {
            wait ();
         }
         catch (InterruptedException e) { }

      writeable = true;
      notify ();

      return c;
   }
}

class Producer extends Thread
{
   private Shared s;

   Producer (Shared s)
   {
      this.s = s;
   }

   public void run ()
   {
      for (char ch = 'A'; ch <= 'Z'; ch++)
      {
           try
           {
              Thread.sleep ((int) (Math.random () * 4000));
           }
         catch (InterruptedException e) {}

         s.setSharedChar (ch);
         System.out.println (ch + " produced by producer.");
      }
   }
}

class Consumer extends Thread
{
   private Shared s;

   Consumer (Shared s)
   {
      this.s = s;
   }

   public void run ()
   {
      char ch;

      do
      {
         try
         {
            Thread.sleep ((int) (Math.random () * 4000));
         }
         catch (InterruptedException e) {}

         ch = s.getSharedChar ();
         System.out.println (ch + " consumed by consumer.");
      }
      while (ch != 'Z');
```

```
  }
}
</code>
```

When you run ProdCons2, you should see the following output (abbreviated for brevity):

```
A produced by producer.
A consumed by consumer.
B produced by producer.
B consumed by consumer.
C produced by producer.
C consumed by consumer.
D produced by producer.
D consumed by consumer.
E produced by producer.
E consumed by consumer.
F produced by producer.
F consumed by consumer.
G produced by producer.
G consumed by consumer.
```

The problems disappeared. The producer always executes before the consumer and never produces an item before the consumer has a chance to consume it. To produce this output,ProdCons2 uses the wait/notify mechanism.

The wait/notify mechanism appears in the Shared class. Specifically, wait() and notify() appear in Shared'ssetSharedChar(char c) and getSharedChar() methods. Sharedalso introduces a writeable instance field, the condition variable that works with wait() and notify() to coordinate the execution of the producer and consumer. Here is how that coordination works, assuming the consumer executes first:

1. The consumer executes s.getSharedChar ().

2. Within that synchronized method, the consumer calls`wait()` (because `writeable` contains true). The consumer waits until it receives notification.
3. At some point, the producer calls `s.setSharedChar (ch);`.
4. When the producer enters that synchronized method (possible because the consumer released the lock inside the `wait()` method just before waiting), the producer discovers `writeable`'s value as true and does not call `wait()`.
5. The producer saves the character, sets `writeable` to false (so the producer must wait if the consumer has not consumed the character by the time the producer next invokes `setSharedChar(char c)`), and calls `notify()` to waken the consumer (assuming the consumer is waiting).
6. The producer exits `setSharedChar(char c)`.
7. The consumer wakens, sets `writeable` to true (so the consumer must wait if the producer has not produced a character by the time the consumer next invokes`getSharedChar()`), notifies the producer to awaken that thread (assuming the producer is waiting), and returns the shared character.

Note: To write more reliable programs that use wait/notify, think about what conditions exist in your program. For example, what conditions exist in `ProdCons2`? Although `ProdCons2` contains only one condition variable, there are two conditions. The first condition is the producer waiting for the consumer to consume a character and the consumer notifying the producer when it consumes the character. The second condition represents the consumer waiting for the producer to produce a character and the producer notifying the consumer when it produces the character.

The rest of the family

In addition to `wait()` and `notify()`, three other methods make up the wait/notify mechanism's method family: `void wait(long millis);`, `void wait(long millis, int nanos);`, and `void notifyAll();`.

The overloaded wait(long millis) and wait(long millis, int nanos) methods allow you to limit how long a thread must wait. wait(long millis) limits the waiting period to millis milliseconds, and wait(long millis, int nanos) limits the waiting period to a combination of millis milliseconds and nanos nanoseconds. As with the no-argument wait() method, code must call these methods from within a synchronized context.

You use wait(long millis) and wait(long millis, int nanos) in situations where a thread must know when notification arrives. For example, suppose your program contains a thread that connects to a server. That thread might be willing to wait up to 45 seconds to connect. If the connection does not occur in that time, the thread must attempt to contact a backup server. By executing wait (45000);, the thread ensures it will wait no more than 45 seconds.

notifyAll() wakens all waiting threads associated with a given lock—unlike thenotify() method, which awakens only a single thread. Although all threads wake up, they must still reacquire the object lock. The JVM selects one of those threads to acquire the lock and allows that thread to run. When that thread releases the lock, the JVM automatically selects another thread to acquire the lock. That continues until all threads have run. Examine Listing 6 for an example ofnotifyAll():

Listing 6. WaitNotifyAllDemo.java

```
// WaitNotifyAllDemo.java
class WaitNotifyAllDemo
{
  public static void main (String [] args)
  {
    Object lock = new Object ();
    MyThread mt1 = new MyThread (lock);
    mt1.setName ("A");
    MyThread mt2 = new MyThread (lock);
    mt2.setName ("B");
    MyThread mt3 = new MyThread (lock);
    mt3.setName ("C");
    mt1.start ();
    mt2.start ();
    mt3.start ();
    System.out.println ("main thread sleeping");
    try
    {
      Thread.sleep (3000);
    }
    catch (InterruptedException e)
    {
    }
    System.out.println ("main thread awake");
    synchronized (lock)
    {
      lock.notifyAll ();
    }
  }
}
class MyThread extends Thread
{
  private Object o;
  MyThread (Object o)
  {
    this.o = o;
  }
  public void run ()
  {
    synchronized (o)
    {
      try
      {
         System.out.println (getName () + " before wait");
         o.wait ();
```

```
            System.out.println (getName () + " after wait");
        }
        catch (InterruptedException e)
        {
        }
      }
   }
}
```

WaitNotifyAllDemo's main thread creates three MyThread objects and assigns names A, B, and C to the associated threads, which subsequently start. The main thread then sleeps for three seconds to give the newly created threads time to wait. After waking up, the main thread calls notifyAll() to awaken those threads. One by one, each thread leaves its synchronizedstatement and run() method, then terminates.

Tip: This article demonstrates notify() in the ProdCons2program because only one thread waits for a condition to occur. In your programs, where more than one thread might simultaneously wait for the same condition to occur, consider using notifyAll(). That way, no waiting thread waits indefinitely.

Thread interruption

One thread can interrupt another thread that is either waiting or sleeping by callingThread's void interrupt(); method. In response, the waiting/sleeping thread resumes execution by creating an object from InterruptedException and throwing that object from the wait() or sleep() methods.

Note: Because the join() methods call sleep() (directly or indirectly), thejoin() methods can also throw InterruptedException objects.

When a call is made to interrupt(), that method either allows a waiting/sleeping thread to resume execution via a thrown exception object or sets a Boolean flag to true (somewhere in the appropriate thread object) to indicate that an executing thread has been interrupted. The method sets the flag only if the thread is neither waiting nor sleeping. An executing thread can determine the Boolean flag's state by calling one of the following Thread methods: static boolean interrupted(); (for the current thread) or boolean isInterrupted(); (for a specific thread). These methods feature two differences:

1. Because interrupted() is a static method, you do not need a thread object before you call it. For example:System.out.println (Thread.interrupted ()); // Display Boolean flag value for current thread. In contrast, because isInterrupted() is a nonstatic method, you need a thread object before you call that method.
2. The interrupted() method clears the Boolean flag to false, whereas the isInterrupted() method does not modify the Boolean flag.

In a nutshell, interrupt() sets the Boolean flag in a thread object and interrupted()/isInterrupted() returns that flag's state. How do we use this capability? Examine Listing 7's ThreadInterruptionDemosource code:

Listing 7. ThreadInterruptionDemo.java

```
// ThreadInterruptionDemo.java
class ThreadInterruptionDemo
{
   public static void main (String [] args)
   {
      ThreadB thdb = new ThreadB ();
      thdb.setName ("B");
      ThreadA thda = new ThreadA (thdb);
      thda.setName ("A");
      thdb.start ();
      thda.start ();
   }
}
class ThreadA extends Thread
{
   private Thread thdOther;
   ThreadA (Thread thdOther)
   {
      this.thdOther = thdOther;
   }
   public void run ()
   {
      int sleepTime = (int) (Math.random () * 10000);
      System.out.println (getName () + " sleeping for " + sleepTime +
                          " milliseconds.");
      try
      {
         Thread.sleep (sleepTime);
      }
      catch (InterruptedException e)
      {
      }
      System.out.println (getName () + " waking up, interrupting other " +
                          "thread and terminating.");
      thdOther.interrupt ();
   }
}
class ThreadB extends Thread
{
   int count = 0;
   public void run ()
   {
      while (!isInterrupted ())
      {
         try
         {
            Thread.sleep ((int) (Math.random () * 10));
         }
         catch (InterruptedException e)
         {
            System.out.println (getName () + " about to terminate...");
            // Because the Boolean flag in the consumer thread's thread
            // object is clear, we call interrupt() to set that flag.
            // As a result, the next consumer thread call to isInterrupted()
            // retrieves a true value, which causes the while loop statement
            // to terminate.
            interrupt ();
         }
         System.out.println (getName () + " " + count++);
      }
   }
}
```

ThreadInterruptionDemo starts a pair of threads: A and B. Asleeps for a random amount of time (up to 10 seconds) before calling interrupt() on B's thread object. B continually checks for interruption by calling its thread object's isInterrupted()method. As long as that method returns false, B executes the statements within the while loop statement. Those statements cause B to sleep for a random amount of time (up to 10 milliseconds), print variable count's value, and increment that value.

When A calls interrupt(), B is either sleeping or not sleeping. If B is sleeping, B wakes up and throws anInterruptedException object from the sleep() method. Thecatch clause then executes, and B calls interrupt() on its thread object to set B's Boolean flag to true. (That flag clears to false when the exception object is thrown.) The next call to isInterrupted() causes execution to leave the while loop statement because isInterrupted() returns true. The result:B terminates. If B is not sleeping, consider two scenarios. First, B has just called isInterrupted() and is about to callsleep() when A calls interrupt(). B's call to sleep() results in that method immediately throwing an InterruptedException object. This scenario is then identical to when B was sleeping: B eventually terminates. Second, B is executingSystem.out.println (getName () + " " + count++); when A calls interrupt(). Bcompletes that method call and calls isInterrupted(). That method returns true, Bbreaks out of the while loop statement, and B terminates.

Review

This article continued to explore Java's threading capabilities by focusing on thread scheduling, the wait/notify mechanism, and thread interruption. You learned that thread scheduling involves either the JVM or the underlying platform's operating system deciphering how to share the processor resource among threads. Furthermore, you learned that the wait/notify mechanism makes it possible for threads to coordinate their executions—to achieve ordered execution, as in the producer-consumer relationship. Finally, you learned that thread interruption allows one thread to prematurely awaken a sleeping or waiting thread.

This article's material proves important for three reasons. First, thread scheduling helps you write platform-independent programs where thread scheduling is an issue. Second, situations (such as the producer-consumer relationship) arise where you must order thread execution. The wait/notify mechanism helps you accomplish that task. Third, you can interrupt threads when your program must terminate even though other threads are waiting or sleeping.

In next month's article, I'll conclude this series by exploring thread groups, volatility, thread local variables, and timers.

Jeff Friesen has been involved with computers for the past 20 years. He holds a degree in computer science and has worked with many computer languages. Jeff has also taught introductory Java programming at the college level. In addition to writing for JavaWorld, he has written his own Java book for beginners— Java 2 by Example, Second Edition (Que Publishing, 2001; ISBN: 0789725932)—and helped write Using Java 2 Platform, Special Edition (Que Publishing, 2001; ISBN: 0789724685). Jeff goes by the nickname Java Jeff (or JavaJeff). To see what he's working on, check out his Website at http://www.javajeff.com.

Understanding Java threads, Part 4: Thread groups, volatility, and thread-local variables

Final concepts for improving Java application programming with Java threads

Thread groups

In a network server program, one thread waits for and accepts requests from client programs to execute, for example, database transactions or complex calculations. The thread usually creates a new thread to handle the request. Depending on the request volume, many different threads might be simultaneously present, complicating thread management. To simplify thread management, programs organize their threads with *thread groups*—`java.lang.ThreadGroup` objects that group related threads' `Thread` (and `Thread`subclass) objects. For example, your program can use `ThreadGroup` to group all printing threads into one group.

Note: To keep the discussion simple, I refer to thread groups as if they organize threads. In reality, thread groups organize `Thread` (and `Thread`subclass) objects associated with threads.

Java requires every thread and every thread group—save the root thread group,`system`—to join some other thread group. That arrangement leads to a hierarchical thread-group structure, which the figure below illustrates in an application context.

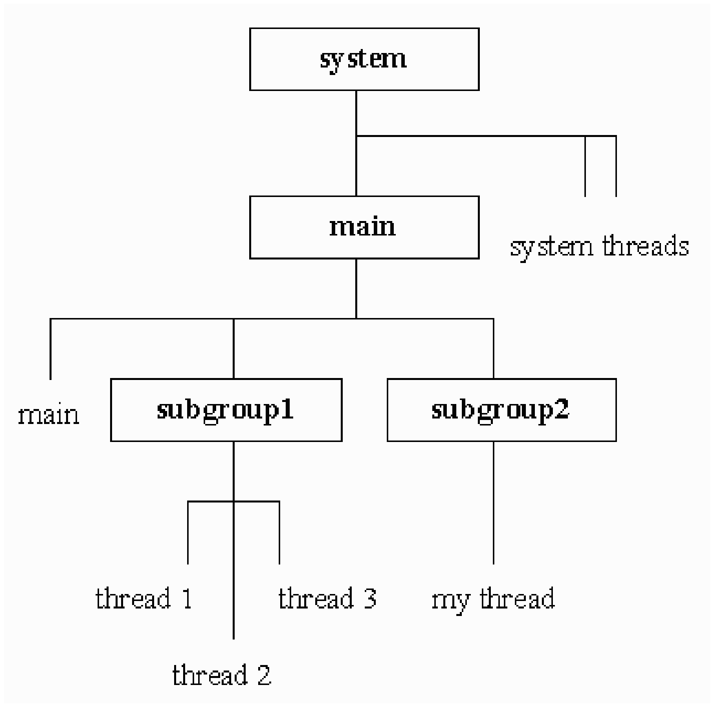

Figure 1. An application's hierarchical thread-group structure begins with a main thread group just below the system thread group

At the top of the figure's structure is the system thread group. The JVM-created system group organizes JVM threads that deal with object finalization and other system tasks, and serves as the root thread group of an application's hierarchical thread-group structure. Just below system is the JVM-created mainthread group, which is system's subthread group (subgroup, for short). main contains at least one thread—the JVM-created main thread that executes byte-code instructions in the main()method.

Below the main group reside the subgroup 1 and subgroup 2 subgroups, application-created subgroups (which the figure's application creates). Furthermore, subgroup 1groups three application-created threads: thread 1, thread 2, and thread 3. In contrast, subgroup 2 groups one application-created thread: my thread.

Now that you know the basics, let's start creating thread groups.

Create thread groups and associate threads with those groups

The ThreadGroup class's SDK documentation reveals two constructors:ThreadGroup(String name) and ThreadGroup(ThreadGroup parent, String name). Both constructors create a thread group and give it a name, as the name parameter specifies. The constructors differ in their choice of what thread group serves as parent to the newly created thread group. Each thread group, except system, must have a parent thread group. For ThreadGroup(String name), the parent is the thread group of the thread that calls ThreadGroup(String name). As an example, if the main thread calls ThreadGroup(String name), the newly created thread group has the main thread's group as its parent—main. For ThreadGroup(ThreadGroup parent, String name), the parent is the group that parent references. The following code shows how to use these constructors to create a pair of thread groups:

```java
public static void main (String [] args)
{
  ThreadGroup tg1 = new ThreadGroup ("A");
  ThreadGroup tg2 = new ThreadGroup (tg1, "B");
}
```

In the code above, the main thread creates two thread groups:A and B. First, the main thread creates A by callingThreadGroup(String name). The tg1-referenced thread group's parent is main because main is the main thread's thread group. Second, the main thread creates B by callingThreadGroup(ThreadGroup parent, String name). The tg2-referenced thread group's parent is A because tg1's reference passes as an argument to ThreadGroup (tg1, "B") and A associates with tg1.

Tip: Once you no longer need a hierarchy of ThreadGroup objects, callThreadGroup's void destroy() method via a reference to the ThreadGroupobject at the top of that hierarchy. If the top ThreadGroup object and all subgroup objects lack thread objects, destroy() prepares those thread group objects for garbage collection. Otherwise, destroy() throws anIllegalThreadStateException object. However, until you nullify the reference to the top ThreadGroup object (assuming a field variable contains that reference), the garbage collector cannot collect that object. Referencing the top object, you can determine if a previous call was made to the destroy()method by calling ThreadGroup's boolean isDestroyed() method. That method returns true if the thread group hierarchy was destroyed.

By themselves, thread groups are useless. To be of any use, they must group threads. You group threads into thread groups by passing ThreadGroup references to appropriateThread constructors:

```
ThreadGroup tg = new ThreadGroup ("subgroup 2");
Thread t = new Thread (tg, "my thread");
```

The code above first creates a subgroup 2 group with main as the parent group. (I assume the main thread executes the code.) The code next creates a my thread Thread object in the subgroup 2 group.

Now, let's create an application that produces our figure's hierarchical thread-group structure:

Listing 1. ThreadGroupDemo.java

```
// ThreadGroupDemo.java
class ThreadGroupDemo
{
  public static void main (String [] args)
  {
    ThreadGroup tg = new ThreadGroup ("subgroup 1");
    Thread t1 = new Thread (tg, "thread 1");
    Thread t2 = new Thread (tg, "thread 2");
    Thread t3 = new Thread (tg, "thread 3");
    tg = new ThreadGroup ("subgroup 2");
    Thread t4 = new Thread (tg, "my thread");
    tg = Thread.currentThread ().getThreadGroup ();
    int agc = tg.activeGroupCount ();
    System.out.println ("Active thread groups in " + tg.getName () +
                        " thread group: " + agc);
    tg.list ();
  }
}
```

ThreadGroupDemo creates the appropriate thread group and thread objects to mirror what you see in the figure above. To prove that the subgroup 1 and subgroup 2 groups are main's only subgroups, ThreadGroupDemo does the following:

1. Retrieves a reference to the main thread's ThreadGroupobject by calling Thread's static currentThread() method (which returns a reference to the main thread's Threadobject) followed by Thread's ThreadGroup getThreadGroup() method.
2. Calls ThreadGroup's int activeGroupCount() method on the just-returned ThreadGroup reference to return an estimate of active groups within the main thread's thread group.
3. Calls ThreadGroup's String getName () method to return the main thread's thread group name.
4. Calls ThreadGroup's void list () method to print on the standard output device details on the main thread's thread group and all subgroups.

When run, ThreadGroupDemo displays the following output:

```
Active thread groups in main thread group: 2
java.lang.ThreadGroup[name=main,maxpri=10]
    Thread[main,5,main]
    Thread[Thread-0,5,main]
    java.lang.ThreadGroup[name=subgroup 1,maxpri=10]
        Thread[thread 1,5,subgroup 1]
        Thread[thread 2,5,subgroup 1]
        Thread[thread 3,5,subgroup 1]
    java.lang.ThreadGroup[name=subgroup 2,maxpri=10]
        Thread[my thread,5,subgroup 2]
```

Output that begins with Thread results from list()'s internal calls to Thread's toString() method, an output format I described in Part 1. Along with that output, you see output beginning with java.lang.ThreadGroup. That output identifies the thread group's name followed by its maximum priority.

Priority and thread groups

A thread group's maximum priority is the highest priority any of its threads can attain. Consider the aforementioned network server program. Within that program, a thread waits for and accepts requests from client programs. Before doing that, the wait-for/accept-request thread might first create a thread group with a maximum priority just below that thread's priority. Later, when a request arrives, the wait-for/accept-request thread creates a new thread to respond to the client request and adds the new thread to the previously created thread group. The new thread's priority automatically lowers to the thread group's maximum. That way, the wait-for/accept-request thread responds more often to requests because it runs more often.

Java assigns a maximum priority to each thread group. When you create a group, Java obtains that priority from its parent group. Use ThreadGroup's void setMaxPriority(int priority)method to subsequently set the maximum priority. Any threads that you add to the group after setting its maximum priority cannot have a priority that exceeds the maximum. Any thread with a higher priority automatically lowers when it joins the thread group. However, if you usesetMaxPriority(int priority) to lower a group's maximum priority, all threads added to the group prior to that method call keep their original priorities. For example, if you add a priority 8 thread to a maximum priority 9 group, and then lower that group's maximum priority to 7, the priority 8 thread remains at priority 8. At any time, you can determine a thread group's maximum priority by callingThreadGroup's int getMaxPriority() method. To demonstrate priority and thread groups, I wrote MaxPriorityDemo:

Listing 2. MaxPriorityDemo.java

```java
// MaxPriorityDemo.java
class MaxPriorityDemo
{
   public static void main (String [] args)
   {
      ThreadGroup tg = new ThreadGroup ("A");
      System.out.println ("tg maximum priority = " + tg.getMaxPriority ());
      Thread t1 = new Thread (tg, "X");
      System.out.println ("t1 priority = " + t1.getPriority ());
      t1.setPriority (Thread.NORM_PRIORITY + 1);
      System.out.println ("t1 priority after setPriority() = " +
                          t1.getPriority ());
      tg.setMaxPriority (Thread.NORM_PRIORITY - 1);
      System.out.println ("tg maximum priority after setMaxPriority() = " +
                          tg.getMaxPriority ());
      System.out.println ("t1 priority after setMaxPriority() = " +
                          t1.getPriority ());
      Thread t2 = new Thread (tg, "Y");
      System.out.println ("t2 priority = " + t2.getPriority ());
      t2.setPriority (Thread.NORM_PRIORITY);
      System.out.println ("t2 priority after setPriority() = " +
                          t2.getPriority ());
   }
}
```

When run, MaxPriorityDemo produces the following output:

```
tg maximum priority = 10
t1 priority = 5
t1 priority after setPriority() = 6
tg maximum priority after setMaxPriority() = 4
t1 priority after setMaxPriority() = 6
t2 priority = 4
t2 priority after setPriority() = 4
```

Thread group A (which tg references) starts with the highest priority (10) as its maximum. Thread X, whose Thread object t1references, joins the group and receives 5 as its priority. We change that thread's priority to 6, which succeeds because 6 is less than 10. Subsequently, we callsetMaxPriority(int priority) to reduce the group's maximum priority to 4. Although thread X remains at priority 6, a newly-added Y thread receives 4 as its priority. Finally, an attempt to increase thread Y's priority to 5 fails, because 5 is greater than 4.

Note: setMaxPriority(int priority) automatically adjusts the maximum priority of a thread group's subgroups.

In addition to using thread groups to limit thread priority, you can accomplish other tasks by calling various ThreadGroup methods that apply to each group's thread. Methods include void suspend(), void resume(), void stop(), and void interrupt(). Because Sun Microsystems has deprecated the first three methods (they are unsafe), we examine only interrupt().

Interrupt a thread group

ThreadGroup's interrupt() method allows a thread to interrupt a specific thread group's threads and subgroups. This technique would prove appropriate in the following scenario: Your application's main thread creates multiple threads that each perform a unit of work. Because all threads must complete their respective work units before any thread can examine the results, each thread waits after completing its work unit. The main thread monitors the work state. Once all other threads are waiting, the main thread calls interrupt()to interrupt the other threads' waits. Then those threads can examine and process the results. Listing 3 demonstrates thread group interruption:

Listing 3. InterruptThreadGroup.java

```
// InterruptThreadGroup.java
class InterruptThreadGroup
{
   public static void main (String [] args)
   {
      MyThread mt = new MyThread ();
      mt.setName ("A");
      mt.start ();
      mt = new MyThread ();
      mt.setName ("B");
      mt.start ();
      try
      {
         Thread.sleep (2000); // Wait 2 seconds
      }
      catch (InterruptedException e)
      {
      }
      // Interrupt all methods in the same thread group as the main
      // thread
      Thread.currentThread ().getThreadGroup ().interrupt ();
   }
}
class MyThread extends Thread
{
   public void run ()
   {
      synchronized ("A")
      {
         System.out.println (getName () + " about to wait.");
         try
```

```
            {
               "A".wait ();
            }
            catch (InterruptedException e)
            {
               System.out.println (getName () + " interrupted.");
            }
            System.out.println (getName () + " terminating.");
         }
      }
   }
```

The main thread creates and starts threads A and B before sleeping for 2,000 milliseconds to give A and B a chance to wait. Upon waking, the main thread assumes A and B are waiting, and executes Thread.currentThread ().getThreadGroup ().interrupt (); to interrupt those threads. Because A and Bexecute in a synchronized context, one thread will throw anInterruptedException object and finish its processing before the other thread does the same. When run,InterruptThreadGroup produces the following output (for one invocation):

```
A about to wait.
B about to wait.
A interrupted.
A terminating.
B interrupted.
B terminating.
```

A is interrupted and terminates before B is interrupted and terminates.

You'll occasionally want to enumerate all threads or subgroups that comprise a thread group. The next section explores that activity.

Enumerate threads and subgroups

Part 1 introduced you to Thread's activeCount() and enumerate(Thread [] thdarray)methods. activeCount() calls ThreadGroup's int activeCount() method via the current thread's group reference to return an estimate of the active threads in the current thread's group and subgroups. enumerate(Thread [] thdarray) callsThreadGroup's int enumerate(Thread [] thdarray) method via the current thread's group reference. enumerate(Thread [] thdarray) is just one of four enumeration methods in ThreadGroup:

1. **int enumerate(Thread [] thdarray)** copies into thdarray references to every active thread in the current thread group and all subgroups.
2. **int enumerate(Thread [] thdarray, boolean recurse)** copies into thdarrayreferences to every active thread in the current thread group only if recurse is false. Otherwise, this method includes active threads from subgroups.
3. **int enumerate(ThreadGroup [] tgarray)** copies into tgarray references to every active subgroup in the current thread group.
4. **int enumerate(ThreadGroup [] tgarray, boolean recurse)** copies into tgarrayreferences to every active subgroup in the current thread group only if recurse is false. Otherwise, this method includes all active subgroups of active subgroups, active subgroups of active subgroups of active subgroups, and so on.

You can use ThreadGroup's activeCount and enumerate(Thread [] thdarray) methods to enumerate all program threads. First, you find the system thread group. Then you call ThreadGroup'sactiveCount() method to retrieve an active thread count for array-sizing purposes. Next, you call ThreadGroup'senumerate(Thread [] thdarray) method to populate that array with Thread references, as Listing 4 demonstrates:

Listing 4. EnumThreads.java

```
// EnumThreads.java
class EnumThreads
{
   public static void main (String [] args)
```

```
  {
     // Find system thread group
     ThreadGroup system = null;
     ThreadGroup tg = Thread.currentThread ().getThreadGroup ();
     while (tg != null)
     {
        system = tg;
        tg = tg.getParent ();
     }
     // Display a list of all system and application threads, and their
     // daemon status
     if (system != null)
     {
        Thread [] thds = new Thread [system.activeCount ()];
        int nthds = system.enumerate (thds);
        for (int i = 0; i < nthds; i++)
             System.out.println (thds [i] + " " + thds [i].isDaemon ());
     }
  }
}
```

When run, EnumThreads produces the following output (on my platform):

```
Thread[Reference Handler,10,system] true
Thread[Finalizer,8,system] true
Thread[Signal Dispatcher,10,system] true
Thread[CompileThread0,10,system] true
Thread[main,5,main] false
```

Apart from a main thread, all other threads belong to thesystem thread group.

Tip: You can easily determine a thread group's parent group by callingThreadGroup's ThreadGroup getParent() method. For all thread groups, savesystem, this method returns a nonnull reference. For system, this method returns null. You can also find out if a thread group is the parent, grandparent, and so forth of another thread group by calling ThreadGroup's boolean parentOf(ThreadGroup tg) method. That method returns true if a thread, whose reference you use to call parentOf(ThreadGroup tg), is a parent (or other ancestor) of the group that tg references—or is the same group as thetg-referenced group. Otherwise, the method returns false.

Volatility

Volatility, that is, changeability, describes the situation where one thread changes a shared field variable's value and another thread sees that change. You expect other threads to always see a shared field variable's value, but that is not necessarily the case. For performance reasons, Java does not require a JVM implementation to read a value from or write a value to a shared field variable in *main memory,* or object heap memory. Instead, the JVM might read a shared field variable's value from a processor register or cache, collectively known as *working memory*. Similarly, the JVM might write a shared field variable's value to a processor register or cache. That capability affects how threads share field variables, as you will see.

Suppose a program creates a shared integer-field variable xwhose initial value in main memory is 10. This program starts two threads; one thread writes to x, and the other reads x's value. Finally, this program runs on a JVM implementation that assigns each thread its own private working memory, meaning each thread has its own private copy of x. When the writing thread writes 6 to x, the writing thread only updates its private working-memory copy of x; the thread does not update the main-memory copy. Also, when the reading thread reads from x, the returned value comes from the reading thread's private copy. Hence, the reading thread returns 10 (because a shared field variable's private working-memory copies initialize to values taken from the main-memory counterpart), not 6. As a result, one thread is unaware of another's change to a shared field variable.

A thread's inability to observe another thread's modification to a shared field variable can cause serious problems. For example, last month's YieldDemoapplication contained a pair of shared field variables: finished and sum. For JVMs that support separate working memory for each thread, the main and main-createdYieldDemo threads can have their own copies of finished and sum. As a result, the main thread's execution of finished = true; would not affect the YieldDemothread's copy of that variable—and the YieldDemo thread would never terminate.

When you ran YieldDemo, you probably discovered that program eventually terminated. That implies your JVM implementation reads/writes main memory instead of working memory. But if you found that the program did not terminate, you probably encountered a situation where the main thread set its working memory copy of finished to true, not the equivalent main-memory copy. Also, the YieldDemo thread read its own working-memory copy of finished, and never saw true.

To fix YieldDemo's visibility problem (on those JVMs that support working memory), include Java's volatile keyword in the finished and sum declarations: static volatile boolean finished = false; andstatic volatile int sum = 0;. The volatile keyword ensures that when a thread writes to a volatile shared field variable, the JVM modifies the main-memory copy, not the thread's working-memory copy. Similarly, the JVM ensures that a thread always reads from the main-memory copy.

Caution: The volatile and final keywords cannot appear together in a shared field variable declaration. Any attempt to include both keywords forces the compiler to report an error.

The visibility problem does not occur when threads use synchronization to access shared field variables. When a thread acquires a lock, the thread's working-memory copies of shared field variables reload from their main-memory counterparts. Similarly, when a thread releases a lock, the working-memory copies flush back to the main-memory shared field variables. For example, in last month's ProdCons2 application, the producer and consumer threads read from/wrote to thewriteable shared field variable's main-memory copy because all access to that shared field variable happened within synchronized contexts. As a result, synchronization allows threads to communicate via shared field variables.

Tip: To ensure that a read/write operation (outside a synchronized context) on either a long-integer shared field variable or a double-precision floating-point shared field variable succeeds, prefix the shared field variable's declaration with keyword volatile.

New developers sometimes think volatility replaces synchronization. Although volatility, through keyword volatile, lets you assign values to long-integer or double-precision floating-point shared field variables outside a synchronized context, volatility cannot replace synchronization. Synchronization lets you group several operations into an indivisible unit, which you cannot do with volatility. However, because volatility is faster than synchronization, use volatility in situations where multiple threads must communicate via a single shared field variable.

Thread-local variables

Sun's Java 2 Platform, Standard Edition (J2SE) SDK 1.2 introduced the java.lang.ThreadLocal class, which developers use to create *thread-local variables*—ThreadLocal objects that store values on a per-thread basis. Each ThreadLocal object maintains a separate value (such as a user ID) for each thread that accesses the object. Furthermore, a thread manipulates its own value and can't access other values in the same thread-local variable.

ThreadLocal has three methods:

1. **Object get ()**: Returns the calling thread's value from the thread-local variable. Because this method is thread-safe, you can call get() from outside a synchronized context.
2. **Object initialValue ()**: Returns the calling thread's initial value from the thread-local variable. Each thread's first call to either get() or set(Object value) results in an indirect call to initialValue() to initialize that thread's value in the thread-local variable. Because ThreadLocal's default implementation of initialValue() returns null, you must subclass ThreadLocal and override this method to return a nonnull initial value.
3. **void set (Object value)**: Sets the current thread's value in the thread-local variable to value. Use this method to replace the value that initialValue() returns.

Listing 5 shows you how to use ThreadLocal:

Listing 5. ThreadLocalDemo1.java

```
// ThreadLocalDemo1.java
class ThreadLocalDemo1
{
   public static void main (String [] args)
   {
      MyThread mt1 = new MyThread ("A");
      MyThread mt2 = new MyThread ("B");
      MyThread mt3 = new MyThread ("C");
      mt1.start ();
      mt2.start ();
      mt3.start ();
   }
}
class MyThread extends Thread
{
   private static ThreadLocal tl =
      new ThreadLocal ()
          {
             protected synchronized Object initialValue ()
             {
                return new Integer (sernum++);
             }
          };
   private static int sernum = 100;
   MyThread (String name)
   {
      super (name);
   }
   public void run ()
```

```
  {
     for (int i = 0; i < 10; i++)
          System.out.println (getName () + " " + tl.get ());
  }
}
```

ThreadLocalDemo1 creates a thread-local variable. That variable associates a unique serial number with each thread that accesses the thread-local variable by callingtl.get (). When a thread first calls tl.get (), ThreadLocal's get() method calls the overridden initialValue() method in the anonymous ThreadLocal subclass. The following output results from one program invocation:

```
A 100
A 100
A 100
A 100
A 100
A 100
A 100
A 100
A 100
A 100
B 101
B 101
B 101
B 101
B 101
B 101
B 101
B 101
B 101
B 101
```

```
C 102
C 102
C 102
C 102
C 102
C 102
C 102
C 102
C 102
C 102
```

The output associates each thread name (A, B, or C) with a unique serial number. If you run this program a second time, you might see a different serial number associate with a thread name. Though the number differs, it always associates with a single thread name.

Note: To allow multiple threads access to the sameThreadLocal object, ThreadLocalDemo1 uses the statickeyword. Without that keyword, each thread accesses its own ThreadLocal object, with each object containing only a value for one thread, not a separate value for each thread. Because thread-local variables store values on a per-thread basis, failing to use static in a thread-local variable declaration serves little purpose.

An alternative to overriding ThreadLocal's initialValue()method is calling that class's set(Object value) method to provide an initial value:

Listing 6. ThreadLocalDemo2.java

```java
// ThreadLocalDemo2.java
class ThreadLocalDemo2
{
  public static void main (String [] args)
  {
    MyThread mt1 = new MyThread ("A");
    MyThread mt2 = new MyThread ("B");
```

```
      MyThread mt3 = new MyThread ("C");
      mt1.start ();
      mt2.start ();
      mt3.start ();
   }
}
class MyThread extends Thread
{
   private static ThreadLocal tl = new ThreadLocal ();
   private static int sernum = 100;
   MyThread (String name)
   {
      super (name);
   }
   public void run ()
   {
      synchronized ("A")
      {
         tl.set ("" + sernum++);
      }
      for (int i = 0; i < 10; i++)
           System.out.println (getName () + " " + tl.get ());
   }
}
```

ThreadLocalDemo2 is nearly identical to ThreadLocalDemo1. However, instead of overriding initialValue() to establish each thread's initial value to a unique serial number, ThreadLocalDemo2 uses a tl.set ("" + sernum++); method call. If you run this program, your output will be more or less identical (on an invocation-by-invocation basis) to ThreadLocalDemo1's output.

Before leaving this section, we must consider one other topic: inheritance and thread-local variables. When a thread creates another thread, the creating thread is the parent thread and the created thread is the child thread. For example, the main thread that executes the main() method's byte-code instructions is the parent of all threads that those instructions create. A child cannot inherit a parent's thread-local values that the parent thread establishes via the ThreadLocal class. However, a parent can use java.lang.InheritableThreadLocal(which extends ThreadLocal) to pass the values of *inheritable thread-local variables* to a child, as Listing 7 demonstrates:

Listing 7. InheritableThreadLocalDemo.java

```
// InheritableThreadLocalDemo.java
class InheritableThreadLocalDemo implements Runnable
{
   static InheritableThreadLocal itl = new InheritableThreadLocal ();
   static ThreadLocal tl = new ThreadLocal ();
   public static void main (String [] args)
   {
      itl.set ("parent thread thread-local value passed to child thread");
      tl.set ("parent thread thread-local value not passed to child
thread");
      InheritableThreadLocalDemo itld;
      itld = new InheritableThreadLocalDemo ();
      Thread child1 = new Thread (itld);
      Thread child2 = new Thread (itld);
      child1.start ();
      child2.start ();
   }
   public void run ()
   {
      System.out.println (itl.get ());
      System.out.println (tl.get ());
   }
}
```

InheritableThreadLocalDemo creates an inheritable thread-local variable via the InheritableThreadLocal class and a thread-local variable via the ThreadLocal class. Furthermore, the main thread calls each variable's set(Object value) method to establish an initial value before creating and starting two child threads. When each child calls get() to retrieve that value, only the inheritable thread-local variable's value returns, as the following output (from one invocation) demonstrates:

```
parent thread thread-local value passed to child thread
null
parent thread thread-local value passed to child thread
null
```

The output shows each child thread printing parent thread thread-local value passed to child thread, which is the inheritable thread-local variable's value. However, null prints as the ThreadLocal variable's value.

Tip: Override InheritableThreadLocal's childValue(Object parentvalue)method to make the child's inheritable thread-local value a function of the parent's inheritable thread-local value.

Timers

Programs occasionally need a *timer* mechanism to execute code either once or periodically, and either at some specified time or after a time interval. Before Sun released J2SE 1.3, a developer either created a custom timer mechanism or relied on another's mechanism. Incompatible timer mechanisms led to difficult-to-maintain source code. Recognizing a need to standardize timer mechanisms, Sun introduced two timer classes in SDK 1.3: java.util.Timer and java.util.TimerTask.

Note: Sun also introduced the javax.swing.Timer class in the 1.3 SDK. I don't present that class here because that discussion requires Swing knowledge. After I introduce you to Swing in a future article, I'll explore Swing's Timerclass.

According to the SDK, use a Timer object to schedule *tasks*—TimerTask and subclass objects—for execution. That execution relies on a thread associated with the Timer object. To create a Timer object, call either the Timer() orTimer(boolean isDaemon) constructor. The constructors differ in the threads they create to execute tasks: Timer() creates a nondaemon thread, whereas Timer(boolean isDaemon) creates a daemon thread, when isDaemon contains true. The following code demonstrates both constructors creating Timer objects:

```
Timer t1 = new Timer (); // Create a nondaemon thread to execute all tasks
Timer T2 = new Timer (true); // Create a daemon thread to execute all tasks
```

Once you create a Timer object, you need a TimerTask to execute. Do that by subclassing TimerTask and overriding TimerTask's run() method. (TimerTaskimplements Runnable, which specifies the run() method.) The following code fragment demonstrates:

```
class MyTask extends TimerTask
{
  public void run ()
  {
    System.out.println ("MyTask task is running.");
  }
}
```

Now that you have a Timer object and a TimerTask subclass, to schedule a TimerTask object for one-time or repeated execution, call one of Timer's four schedule() methods:

1. **void schedule(TimerTask task, Date time)**: Schedulestask for one-time execution at the specified time.
2. **void schedule(TimerTask task, Date firstTime, long interval)**: Schedules task for repeated execution at the specified firstTime and at interval millisecond intervals following firstTime. This execution is known as *fixed-delay execution* because each subsequent taskexecution occurs relative to the previous taskexecution's actual execution time. Furthermore, if an execution delays because of garbage collection or some other background activity, all subsequent executions also delay.
3. **void schedule(TimerTask task, long delay)**: Schedules task for one-time execution after delay milliseconds pass.
4. **void schedule(TimerTask task, long delay, long interval)**: Schedules task for repeated execution after delay milliseconds pass and at interval millisecond intervals following firstTime. This method employs fixed-delay execution.

The following code fragment, which assumes a t1-referencedTimer object, creates a MyTask object and schedules that object using the fourth method in the above list for repeated execution (every second), following an initial delay of zero milliseconds:

```
t1.schedule (new MyTask (), 0, 1000);
```

Every second (that is, 1,000 milliseconds), the Timer's thread executes the MyTask's run() method. For a more useful example of task execution, check out Listing 8:

Listing 8. Clock1.java

```
// Clock1.java
// Type Ctrl+C (or equivalent keystroke combination on non-Windows
platform)
// to terminate
import java.util.*;
class Clock1
```

```
{
  public static void main (String [] args)
  {
    Timer t = new Timer ();
    t.schedule (new TimerTask ()
                {
                    public void run ()
                    {
                      System.out.println (new Date ().toString ());
                    }
                },
                0,
                1000);
  }
}
```

Clock1 creates a Timer object and calls schedule(TimerTask task, long delay, long interval) to schedule fixed-delay executions of an anonymous TimerTask subclass object's run()method. That method retrieves the current date by callingjava.util.Date's Date() constructor and converts the Dateobject's contents to a human-readable String, which subsequently prints. The following partial output shows the results of running this program:

```
Mon Jul 01 16:23:49 CDT 2002
Mon Jul 01 16:23:51 CDT 2002
Mon Jul 01 16:23:52 CDT 2002
Mon Jul 01 16:23:53 CDT 2002
Mon Jul 01 16:23:54 CDT 2002
Mon Jul 01 16:23:55 CDT 2002
Mon Jul 01 16:23:56 CDT 2002
Mon Jul 01 16:23:57 CDT 2002
```

In addition to the four schedule() methods, Timer includes twoscheduleAtFixedRate() methods:

1. **void scheduleAtFixedRate(TimerTask task, Date firstTime, long interval):**Schedules task for repeated execution at the specified firstTime and at intervalmillisecond intervals following firstTime. This execution is known as *fixed-rate execution* because each subsequent task execution occurs relative to the initialtask execution. Furthermore, if an execution delays because of garbage collection or some other background activity, two or more executions occur in rapid succession to maintain the execution frequency.
2. **void scheduleAtFixedRate(TimerTask task, long delay, long interval):**Schedules task for repeated execution after delay milliseconds pass and at intervalmillisecond intervals following firstTime. This method employs fixed-rate execution.

Two of the four schedule() methods use fixed-delay execution, whereas both scheduleAtFixedRate() methods use fixed-rate execution. How do these execution styles differ? Fixed-delay execution promotes an accurate frequency in the short run versus the long run. This execution style is appropriate for tasks that must operate smoothly, such as many animation tasks and a blinking cursor, where erratic movements interrupt the programs' fluidity. In contrast to fixed-delay execution, fixed-rate execution promotes total frequency accuracy at the expense of execution smoothness. This style is appropriate for counters and clocks that should not miss a single execution. Why is that important? In Clock1's output, you see the time moving from Mon Jul 01 16:23:49 CDT 2002 to Mon Jul 01 16:23:51 CDT 2002, with no intermediate Mon Jul 01 16:23:50 CDT 2002. Clock1's fixed-delay execution results in a loss of total frequency accuracy. We can correct that problem by using fixed-rate execution:

Listing 9. Clock2.java

```
// Clock2.java
// Type Ctrl+C (or equivalent keystroke combination on non-Windows 
platform)
// to terminate
```

```
import java.util.*;
class Clock2
{
  public static void main (String [] args)
  {
    Timer t = new Timer ();
    t.scheduleAtFixedRate (new TimerTask ()
                         {
                           public void run ()
                           {
                             System.out.println (new Date ().
                                             toString ());
                           }
                         },
                         0,
                         1000);
  }
}
```

With Clock2, you are less likely to miss seeing a single second because scheduleAtFixedRate(TimerTask task, long delay, long interval) promotes extra task executions when a delay occurs. But if the delay is excessive, the output can still omit various times.

Clock1 and Clock2 have a problem: Neither program provides platform-independent termination. Depending on the platform used, a user must type some keystroke combination to exit either program. To solve this problem, you could construct a daemon task execution thread by calling Timer (true). That way, the program ends when the main thread ends. However, because this technique does not allow the currently executing task to finish, it is problematic. Imagine what would happen if the task is writing to a file when the application ends. A better approach allows the main thread to initiate a thread that checks for user input and waits for the input thread to terminate once input occurs. The main thread then calls Timer's void cancel()method to terminate the timer and discard all scheduled tasks after the currently executing task leaves its run() method. Listing 10 demonstrates this approach:

Listing 10. Clock3.java

```
// Clock3.java
// Type Ctrl+C (or equivalent keystroke combination on non-Windows 
platform)
// to terminate
import java.util.*;
class Clock3
{
   public static void main (String [] args)
   {
      Timer t = new Timer ();
      t.scheduleAtFixedRate (new TimerTask ()
                             {
                                public void run ()
                                {
                                   System.out.println (new Date ().
                                                       toString ());
                                }
                             },
                             0,
                             1000);
      InputThread it = new InputThread ();
      it.start ();
      try
      {
          // Wait for input thread to terminate
          it.join ();
      }
      catch (InterruptedException e)
      {
    }
    // Terminate the timer and discard all scheduled tasks after the
    // currently executing task leaves its run() method
    t.cancel ();
  }
}
class InputThread extends Thread
{
  public void run ()
  {
    try
    {
      // Wait for user to type Enter key
      System.in.read ();
    }
    catch (java.io.IOException e)
    {
    }
  }
}
```

Tip: To terminate the currently running task without affecting other tasks, call TimerTask's boolean cancel()method. That method cancels the current task so that it will never run again (assuming the task is repeating) after it finishes its current execution and returns a Boolean true value if either the task is a one-time task that has not yet run or a repeating task. False returns if a one-time task has already run, if it was never scheduled, or if it was cancelled. TimerTask's cancel() method does not cancel any other tasks.

Thread death

Prior to the deprecation of Thread's stop() method, developers often called that method to terminate a thread. stop() throws a ThreadDeath object, causing the thread to exit from any point within run()'s execution by unwinding the thread's method-call stack. The JVM catches the thrown ThreadDeath object, yet does not display a stack trace.

Though you shouldn't use stop() anymore, you might need to stop a thread during its execution. Although thread termination normally involves returning from therun() method, that might prove difficult to accomplish if the thread's execution is deep within a nested set of method calls. By throwing a ThreadDeath object, a thread can unwind the method-call stack and terminate gracefully as Listing 11 shows:

Listing 11. ThreadDeathDemo.java

```
// ThreadDeathDemo.java
class ThreadDeathDemo
{
   public static void main (String [] args)
   {
      MyThreadGroup mtg = new MyThreadGroup ("My Group");
      new MyThread (mtg, "My Thread").start ();
   }
}
class MyThread extends Thread
{
   MyThread (ThreadGroup tg, String name)
   {
      super (tg, name);
   }
   public void run ()
   {
      System.out.println ("About to do something.");
      doSomething ();
      System.out.println ("Something done.");
   }
   void doSomething ()
   {
      doSomethingHelper ();
   }
   void doSomethingHelper ()
   {
      throw new MyThreadDeath (MyThreadDeath.REASON2);
   }
}
class MyThreadDeath extends ThreadDeath
{
   final static int REASON1 = 1;
   final static int REASON2 = 2;
   final static int REASON3 = 3;
   int reason;
   MyThreadDeath (int reason)
   {
      this.reason = reason;
   }
}
class MyThreadGroup extends ThreadGroup
{
   MyThreadGroup (String name)
   {
      super (name);
   }
   public void uncaughtException (Thread t, Throwable e)
   {
      if (e instanceof MyThreadDeath)
      {
          reportError (t, e);
          cleanup ();
      }
      super.uncaughtException (t, e);
   }
   void reportError (Thread t, Throwable e)
   {
      System.out.print (t.getName () + " unable to do something. Reason: 
");
      switch (((MyThreadDeath) e).reason)
      {
         case MyThreadDeath.REASON1:
            System.out.println ("First reason.");
            break;
         case MyThreadDeath.REASON2:
            System.out.println ("Second reason.");
            break;
         case MyThreadDeath.REASON3:
            System.out.println ("Third reason.");
      }
   }
   void cleanup ()
   {
      System.out.println ("Cleaning up");
   }
}
```

ThreadDeathDemo's main thread executes main()'s byte-code instructions, which create a MyThreadGroup object and aMyThread object that groups into MyThreadGroup. The main thread then starts a thread associated with MyThread.

The MyThread thread enters its run() method, where it prints some text and calls the doSomething() method. That method subsequently calls doSomethingHelper(), which throws aMyThreadDeath object. (I subclass ThreadDeath so I can assign a reason for the death of a thread.) At some point, the thrownMyThreadDeath object results in a call to MyThreadGroup's uncaughtException(Thread t, Throwable e) method, which determines if a MyThreadDeath object was thrown. If so, calls are made to methods reportError(Thread t, Throwable e) and cleanup(), to print error information and perform cleanup operations, respectively. When run,ThreadDeathDemo produces the following output:

```
About to do something.
My Thread unable to do something. Reason: Second reason.
Cleaning up
```

Caution: ThreadDeath is a powerful tool for causing a thread to terminate its execution. However, this tool is dangerous, and is the reason Sun deprecated the stop() method. When a thread throws a ThreadDeath object, all locked monitors unlock as ThreadDeath propagates up the method-call stack. Objects protected by these monitors become accessible to other threads. If those objects are in an inconsistent state, a program can experience erratic behavior, a database or file can corrupt, and so on. However, if you know that the thread is not holding any locks, you can safely throw ThreadDeath.

Review

This article completes my coverage of threads by exploring thread groups, volatility, thread-local variables, timers, andThreadDeath. You learned to use thread groups to group related threads, to use volatility to allow threads access to main-memory copies of shared field variables, to use thread-local variables to give threads their own independently initialized values, to use timers to schedule the execution of tasks either periodically or for a one-time execution, and to use ThreadDeath to let a thread prematurely exit from itsrun() method.

Next month I'll show you how to use packages to organize your classes and interfaces.

Jeff Friesen has been involved with computers for the past 20 years. He holds a degree in computer science and has worked with many computer languages. Jeff has also taught introductory Java programming at the college level. In addition to writing for JavaWorld, he has written his own Java book for beginners— Java 2 by Example, Second Edition (Que Publishing, 2001; ISBN: 0789725932)—and helped write Using Java 2 Platform, Special Edition (Que Publishing, 2001; ISBN: 0789724685). Jeff goes *by the nickname Java Jeff (or JavaJeff). To see what he's working on, check out his Website at http://www.javajeff.com.*