This story appeared on JavaWorld at
http://www.javaworld.com/javaworld/jw-06-2012/120626-modern-threading.html

Modern threading: A Java concurrency primer

Understanding Java threads in a post-JDK 1.4 world

By Cameron Laird, JavaWorld.com, 06/26/12

Much of what there is to learn about programming with Java threads hasn't changed dramatically over the evolution of the Java platform, but it has changed incrementally. In this Java threads primer, Cameron Laird hits some of the high (and low) points of threads as a concurrent programming technique. Get an overview of what's perennially challenging about multithreaded programming and find out how the Java platform has evolved to meet some of the challenges.

Concurrency is among the greatest worries for newcomers to Java programming but there's no reason to let it daunt you. Not only is excellent documentation available (we'll explore several sources in this article) but Java threads have become easier to work with as the Java platform has evolved. In order to learn how to do multithreaded programming in Java 6 and 7, you really just need some building blocks. We'll start with these:

- A simple threaded program
- Threading is all about speed, right?
- Challenges of Java concurrency
- When to use Runnable
- When good threads go bad
- What's new in Java 6 and 7
- What's next for Java threads

This article is a beginner's survey of Java threading techniques, including links to some of JavaWorld's most frequently read introductory articles about multithreaded programming. Start your engines and follow the links above if you're ready to start learning about Java threading today.

A simple threaded program

Consider the following Java source.

Listing 1. FirstThreadingExample

```
class FirstThreadingExample {
   public static void main (String [] args) {
    // The second argument is a delay between
    //   successive outputs.  The delay is
    //   measured in milliseconds.  "10", for
    //   instance, means, "print a line every
    //   hundredth of a second".
      ExampleThread mt = new ExampleThread("A", 31);
      ExampleThread mt2 = new ExampleThread("B", 25);
      ExampleThread mt3 = new ExampleThread("C", 10);
      mt.start();
      mt2.start();
      mt3.start();
   }
}

class ExampleThread extends Thread {
   private int delay;
   public ExampleThread(String label, int d) {
         // Give this particular thread a
         //   name:  "thread 'LABEL'".
      super("thread '" + label + "'");
      delay = d;
   }
   public void run () {
      for (int count = 1, row = 1; row < 20; row++, count++) {
         try {
            System.out.format("Line #%d from %s\n",
                                          count, getName());
            Thread.currentThread().sleep(delay);
         }
         catch (InterruptedException ie) {
            // This would be a surprise.
         }
      }
   }
}
```

Now compile and run this source as you would any other Java command-line application. You'll see output that looks something like this:

Listing 2. Output of a threaded program

```
Line #1 from thread 'A'
Line #1 from thread 'C'
Line #1 from thread 'B'
Line #2 from thread 'C'
Line #3 from thread 'C'
Line #2 from thread 'B'
Line #4 from thread 'C'
   ...
Line #17 from thread 'B'
Line #14 from thread 'A'
Line #18 from thread 'B'
Line #15 from thread 'A'
Line #19 from thread 'B'
Line #16 from thread 'A'
Line #17 from thread 'A'
Line #18 from thread 'A'
Line #19 from thread 'A'
```

That's it -- you're a Java `Thread` programmer!

Well, okay, maybe not so fast. As small as the program in Listing 1 is, it contains some subtleties that merit our attention.

Threads and indeterminacy

A typical learning cycle with programming consists of four stages: (1) Study new concept; (2) execute sample program; (3) compare output to expectation; and (4) iterate until the two match. Note, though, that I previously said the output for `FirstThreadingExample` would look "something like" Listing 2. So, that means your output could be different from mine, line by line. What's *that* about?

In the simplest Java programs, there is a guarantee of order-of-execution: the first line in `main()` will be executed first, then the next, and so on, with appropriate tracing in and out of other methods. `Thread` weakens that guarantee.

Threading brings new power to Java programming; you can achieve results with threads that you couldn't do without them. But that power comes at the cost of *determinacy*. In the simplest Java programs, there is a guarantee of order-of-execution: the first line in `main()` will be executed first, then the next, and so on, with appropriate tracing in and out of other methods. `Thread` weakens that guarantee. In a multithreaded program, "`Line #17 from thread B`" might appear on your screen before or after "`Line #14 from thread A`," and the order might differ on successive executions of the same program, even on the same computer.

Indeterminacy may be unfamiliar, but it needn't be disturbing. Order-of-execution *within* a thread remains predictable, and there are also advantages associated with indeterminacy. You might have experienced something similar when working with graphical user interfaces (GUIs). Event listeners in Swing or event handlers in HTML are examples.

While a full discussion of thread synchronization is outside the scope of this introduction, it's easy to explain the basics.

For instance, consider the mechanics of how HTML specifies `... onclick = "myFunction();" ...` to determine the action that will happen after the user clicks. This familiar case of indeterminacy illustrates some of its advantages. In this case, `myFunction()` isn't executed at a definite time with respect to other elements of source code, but *in relation to the end-user's action*. So indeterminacy isn't just a weakness in the system; it's also an *enrichment* of the model of execution, one that gives the programmer new opportunities to determine sequence and dependency.

Execution delays and Thread subclassing

You can learn from `FirstThreadingExample` by experimenting with it on your own. Try adding or removing `ExampleThreads` -- that is, constructor invocations like `... new ExampleThread(label, delay);` -- and tinkering with the `delays`. The basic idea is that the program starts three separate `Threads`, which then run independently until completion. To make their execution more instructive, each one delays slightly between the successive lines it writes to output; this gives the other threads a chance to write *their* output.

Note that `Thread`-based programming does not, in general, require handling an `InterruptedException`. The one shown in `FirstThreadingExample` has to do with `sleep()`, rather than being directly related to `Thread`. Most `Thread`-based source does not include a `sleep()`; the purpose of `sleep()` here is to model, in a simple way, the behavior of long-running methods found "in the wild."

Something else to notice in Listing 1 is that `Thread` is an *abstract* class, designed to be subclassed. Its default `run()` method does nothing, so must be overridden in the subclass definition to accomplish anything useful.

This is all about speed, right?

So by now you can see a little bit of what makes programming with threads complex. But the main point of enduring all these difficulties *isn't* to gain speed.

Multithreaded programs *do not*, in general, complete faster than single-threaded ones -- in fact they can be significantly slower in pathologic cases. The fundamental added value of multithreaded programs is *responsiveness*. When multiple processing cores are available to the JVM, or when the program spends significant time waiting on multiple external resources such as network responses, then multithreading can help the program complete faster.

Think of a GUI application: if it still responds to end-user points and clicks while searching "in the background" for a matching fingerprint or re-calculating the calendar for next year's tennis tournament, then it was built with concurrency in mind. A typical concurrent application architecture puts recognition and response to user actions in a thread separate from the computational thread assigned to handle the big back-end load. (See "Swing threading and the event-dispatch thread" for further illustration of these principles.)

In your own programming, then, you're most likely to consider using `Threads` in one of these circumstances:

1. An existing application has correct functionality but is unresponsive at times. These "blocks" often have to do with external resources outside your control: time-consuming database queries, complicated calculations, multimedia playback, or networked responses with uncontrollable latency.
2. A computationally-intense application could make better use of multicore hosts. This might be the case for someone rendering complex graphics or simulating an involved scientific model.
3. `Thread` naturally expresses the application's required programming model. Suppose, for instance, that you were modeling the behavior of rush-hour automobile drivers or bees in a hive. To implement each driver or bee as a `Thread`-related object might be convenient from a programming standpoint, apart from any considerations of speed or responsiveness.

Challenges of Java concurrency

Experienced programmer Ned Batchelder recently quipped

> Some people, when confronted with a problem, think, "I know, I'll use threads," and then two they hav erpoblesms.

That's funny because it so well models the problem with concurrency. As I already mentioned, multithreaded programs are likely to give different results in terms of the exact sequence or timing of thread execution. That's troubling to programmers, who are trained to think in terms of reproducible results, strict determinacy, and invariant sequence.

It gets worse. Different threads might not only produce results in different orders, but they can *contend* at more essential levels for results. It's easy for a newcomer to multithreading to `close()` a file handle in one `Thread` before a different `Thread` has finished everything it needs to write.

Testing concurrent programs

Ten years ago on JavaWorld, Dave Dyer noted that the Java language had one feature so "pervasively used incorrectly" that he ranked it as a serious design flaw. That feature was multithreading.

Dyer's comment highlights the challenge of testing multithreaded programs. When you can no longer easily specify the output of a program in terms of a definite sequence of characters, there will be an impact on how effectively you can test your threaded code.

The correct starting point to resolving the intrinsic difficulties of concurrent programming was well stated by Heinz Kabutz in his Java Specialist newsletter: recognize that concurrency is a topic that you should understand and study it systematically. There are of course tools such as diagramming techniques and formal languages that will help. But the first step is to sharpen your intuition by practicing with simple programs like `FirstThreadingExample` in Listing 1. Next, learn as much as you can about threading fundamentals like these:

- Synchronization and immutable objects
- Thread scheduling and wait/notify
- Race conditions and deadlock
- Thread monitors for exclusive access, conditions, and assertions
- JUnit best practices -- testing multithreaded code

When to use Runnable

Object orientation in Java defines singly inherited classes, which has consequences for multithreading coding. To this point, I have only described a use for `Thread` that was based on subclasses with an overridden `run()`. In an object design that already involved inheritance, this simply wouldn't work. You cannot simultaneously inherit from `RenderedObject` or `ProductionLine` or `MessageQueue` alongside `Thread`!

This constraint affects many areas of Java, not just multithreading. Fortunately, there's a classical solution for the problem, in the form of the `Runnable` interface. As explained by Jeff Friesen in his 2002 introduction to threading, the `Runnable` interface is made for situations where subclassing `Thread` isn't possible:

> 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 `Thread` constructor.

So for those classes that cannot extend `Thread`, you must create a runnable to take advantage of multithreading.

Semantically, if you're doing system-level programming and your class is in an is-a relation to `Thread`, then you should subclass directly from `Thread`. But most application-level use of multithreading relies on composition, and thus defines a `Runnable` compatible with the application's class diagram. Fortunately, it takes only an extra line or two to code using the `Runnable` interface, as shown in Listing 3 below.

Listing 3. MonitorModel

```
import java.util.Random;

class MonitorModel {
   public static void main (String [] args) {
      new RemoteHost("example 1");
      new RemoteHost("example 2");
   }
}

class MonitoredObject {
}

// This models a simple monitor of a remote host, as an
// administrative program might use.  A new value from
// the RemoteHost appears every 'delay' milliseconds, and
// is reported to the screen.
class RemoteHost extends MonitoredObject implements Runnable {
    String name;
    int delay;
    RemoteHost(String n) {
        name = n;
        new Thread(this).start();
    }
    public void run() {
      int count = 0;
      Random r = new Random();

        // Continue indefinitely, until a keyboard interrupt.
      while (count++) {
         try {
            int delay = r.nextInt(1000);
            System.out.format(
              "Line #%d from RemoteHost '%s', after %d-milliseconds.\n",
                                               count, name, delay);
            Thread.currentThread().sleep(delay);
         }
         catch (InterruptedException ie) {
            // This would be a surprise.
         }
     count++;
      }
    }
}
```

When you run this example, you'll see output from two different threads interleave unpredictably, as is likely to happen in a real-world system monitor. Output is likely to include "runs" where one `RemoteHost` reports repeatedly while the other is silent. Listing 4 shows three notices from `'example-1'` in succession:

Listing 4. Output from MonitorModel

```
.
```

```
      .
      .
Line #54 from RemoteHost 'example 1', after 875-milliseconds.
Line #59 from RemoteHost 'example 2', after 964-milliseconds.
Line #55 from RemoteHost 'example 1', after 18-milliseconds.
Line #56 from RemoteHost 'example 1', after 261-milliseconds.
Line #57 from RemoteHost 'example 1', after 820-milliseconds.
Line #60 from RemoteHost 'example 2', after 807-milliseconds.
Line #58 from RemoteHost 'example 1', after 525-milliseconds.
      .
      .
      .
```

Using `Runnable` is thus nearly as succinct as directly subclassing from `Thread`. (Remember that most application code relies on `Runnable` definitions rather than `Thread` subclassing, if only to avoid a multiple-inheritance conflict.)

About monitors

Note that *monitor* in Listing 3 stems from the vocabulary of system administrators or network operators, who monitor objects -- often physical ones -- for which they're responsible. Coincidentally, programming theory applies the same word to a specific concept in concurrency. In a classic JavaWorld article from 1997, Bill Venners explained monitors thus:

> A *monitor* is basically a guardian in that it watches over a sequence of code, making sure that only one thread at a time executes the code ... Each monitor is associated with an object reference. When a thread arrives at the first instruction in a block of code that is under the watchful eye of a monitor, the thread must obtain a lock on the referenced object. The thread is not allowed to execute the code until it obtains the lock. Once it has obtained the lock, the thread enters the block of protected code.

The examples of multithreading to this point have involved little coordination between threads; there's been no particular consequence for one thread executing before another, nor much teamwork required between them. A programming monitor is one of several mechanisms for managing thread synchronization. Locking is another concept, which I'll discuss shortly.

When good threads go bad ... good programmers make them better

> *There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies. -- C.A.R. Hoare*

Hoare's words about the complications of coding seems to apply with special force to multithreading, with the result that many programmers simply shun thread-based code. Most of us can't do without the responsiveness and performance improvements of concurrency, however, especially as software systems evolve onto multicore architectures.

There's no getting around the reality that multithreading is hard. Worse, you might have heard it said that threads are a bad idea, meaning, prone to difficulty.

Some Java developers try to mitigate the hazards of thread programming by trading it in for newer (or just different) styles. Message-passing models expressed in terms of actors and agents eliminate many of the difficulties of shared state and synchronization. So, in a different way, do closures, which come baked into Java 7. The popularity of these alternative concurrency models in Java-related languages like Clojure has amplified their familiarity in the Java community proper.

Still, you might prefer to work with the standard Java threading architecture. As your applications grow more sophisticated, you'll need to learn not only how to launch runnables, but also to synchronize thread execution, communicate between threads, and manage thread lifetimes. With a little effort and the well-established design techniques discussed below you can make your multithreaded code as understandable and reliable as any other Java source.

Loose coupling

If you are going to stick with the Java Threads API and `java.util.concurrent`, what can you do to reduce the hazards of multithreading? First, develop your own sense of functional style. You know to minimize use of `goto` and global variables; in much the same way, design your `Runnables` to be as simple and loosely coupled as possible. "Loose coupling" here has several aspects:

- Share as little state as possible
- Minimize side-effects and emphasize immutable objects
- Use standard design patterns (like Subscribe-Publish) to manage shared resources
- Minimize inter-thread coordination
- Simplify the lifespans of threads

Write testable code

Make your threads *testable*. A conventional application might build in facilities to ensure not just code coverage, but the precise results of "edge cases": does the program behave correctly when a limit is reached? When within one unit of the limit? In the same way, construct your multithreaded application to exercise threads under reproducible, salient conditions: with only a single worker thread operating; with two threads, started in either order; with more threads than the test host has cores; with controllable simulated loads; and so on.

Debug your code

Next, learn how to debug multithreaded applications. Studying best practices for debugging will improve your code measurably. See Esther Schindler's "Learning and improving your debugging skills" for a collection of valuable tips and exercises that will improve your Java debugging techniques.

FInally, there are tools. In the early years of Java, standalone utilities to help with debugging of thread problems were popular; more recently, the leading debuggers have incorporated essentially all of this functionality. (For instance, nearly every debugger, including `jdb`, supports introspection on threads.) Know what your favorite debugger offers and be sure that it compares well with the alternatives.

What's new in Java 6 and 7

The basics of multithreading have changed remarkably little since the early days of the Java platform. One fortunate change, however, is that multithreading has become *easier* in several important regards.

First, when Java 6 was released at the end of 2006 it brought considerable attention to *locking*. Locking is an important technique because threads sometimes contend for resources: two different threads might need to update a global variable that tracks how many users are active, or to update a bank account balance. It's very bad, though, to allow the threads to interfere with each other. In the case of an accounting program, imagine a starting balance of $100. One thread tries to debit $10, the other to credit $20. Depending on the exact sequence of operations, it's easy to end up with a final balance of $120 or $90 rather than the correct combination of $110.

Software applications need to *guarantee* correct results. The most common way to achieve this is with *locks* or other means of synchronizing the access of different threads. One thread -- the credit one, say, in the example of the last paragraph -- locks the account balance, completes its operation, sets the balance to $120. At that point, it releases the lock to the debit thread, which itself locks the balance, until it has successfully updated the balance to the correct sum of $110.

Perils of locking

Locks have the reputation of being difficult and costly, and occasionally they're simply wrong. Improvements to Java 6 and the JVMs that implement locks improved performance and refined programmatic control over them. The result: the speed of multithreaded applications generally improved, sometimes dramatically. (See "Do Java 6 threading optimizations actually work?" for more about threading optimization in Java 6.)

With the release of Java 7 in the summer of 2011, we saw a different kind of improvement to thread programming: removal! One common application of multithreading has been to answer questions such as, "is there a new file in this FTP repository?" or "have any results been appended to this logfile lately?" Java 7 includes a filesystem monitor and support for asynchronous input/output in `java.nio.file`. So the `nio` API allows us to directly code file update functionality, thus eliminating one need for programmers to write multithreaded source.

Closures also reduced the pressure to code with threads. For instance, the `java.util.concurrent` package includes support for writing concurrent loops (Neal Gafter, 2006).

Conclusion: What's next for threads

Current public plans for Java 8 and Java 9 appear to have little direct impact on threaded programming. A major goal of Java 8, scheduled for mid-2013, is more effective use of multicore environments. Expansion of `java.util.concurrent.Callable` closures will encourage their use (under the formal language label of "lambda") as an alternative to multithreading.

One of the goals for Java 9 is "massive multicore" compatibility. It's not yet clear how this will be effected at the level of language syntax.

One of the most interesting conclusions of the last fifteen years of Java concurrency programming is that threads are not so much to be avoided as handled with respect. Some teams do best with new `java.util.concurrent` techniques, or even some of the alternative models I've mentioned. For many teams, though, the best results come from using threads, but using them with the wisdom of the last decade. This generally involves simple synchronization models, an emphasis on testability, small and understandable class definitions, and teams that carefully review each other's source.

About the author

Cameron Laird began writing Java code before it was called Java and has contributed occasionally to *JavaWorld* in the years since then. Keep up with his coding and writing through Twitter as @Phaseit.

Read more about Core Java in JavaWorld's Core Java section.