Introduction to Java threads

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. About this tutorial

What is this tutorial about?

This tutorial explores the basics of threads -- what they are, why they are useful, and how to get started writing simple programs that use them.

We will also explore the basic building blocks of more sophisticated threading applications -- how to exchange data between threads, how to control threads, and how threads can communicate with each other.

Should I take this tutorial?

This tutorial is for Java programmers who have a good working knowledge of the Java language, but who have limited experience with multithreading or concurrency.

At the completion of this tutorial, you should be able to write simple programs that use threads. You should also be able to read and understand programs that use threads in straightforward ways.

About the author

Brian Goetz is a regular columnist on the *developerWorks* Java technology zone and has been a professional software developer for the past 15 years. He is a Principal Consultant at *Quiotix*, a software development and consulting firm located in Los Altos, California.

See Brian's *published and upcoming articles* in popular industry publications.

Contact Brian at *brian@quiotix.com*.

Section 2. Thread basics

What are threads?

Nearly every operating system supports the concept of processes -- independently running programs that are isolated from each other to some degree.

Threading is a facility to allow multiple activities to coexist within a single process. Most modern operating systems support threads, and the concept of threads has been around in various forms for many years. Java is the first mainstream programming language to explicitly include threading within the language itself, rather than treating threading as a facility of the underlying operating system.

Threads are sometimes referred to as *lightweight processes*. Like processes, threads are independent, concurrent paths of execution through a program, and each thread has its own stack, its own program counter, and its own local variables. However, threads within a process are less insulated from each other than separate processes are. They share memory, file handles, and other per-process state.

A process can support multiple threads, which appear to execute simultaneously and asynchronously to each other. Multiple threads within a process share the same memory address space, which means they have access to the same variables and objects, and they allocate objects from the same heap. While this makes it easy for threads to share information with each other, you must take care to ensure that they do not interfere with other threads in the same process.

The Java thread facility and API is deceptively simple. However, writing complex programs that use threading effectively is not quite as simple. Because multiple threads coexist in the same memory space and share the same variables, you must take care to ensure that your threads don't interfere with each other.

Every Java program uses threads

Every Java program has at least one thread -- the main thread. When a Java program starts, the JVM creates the main thread and calls the program's `main()` method within that thread.

The JVM also creates other threads that are mostly invisible to you -- for example, threads associated with garbage collection, object finalization, and other JVM housekeeping tasks. Other facilities create threads too, such as the AWT (Abstract Windowing Toolkit) or Swing UI toolkits, servlet containers, application servers, and RMI (Remote Method Invocation).

Why use threads?

There are many reasons to use threads in your Java programs. If you use Swing, servlets, RMI, or Enterprise JavaBeans (EJB) technology, you may already be using threads without

realizing it.

Some of the reasons for using threads are that they can help to:

- Make the UI more responsive
- Take advantage of multiprocessor systems
- Simplify modeling
- Perform asynchronous or background processing

More responsive UI

Event-driven UI toolkits, such as AWT and Swing, have an event thread that processes UI events such as keystrokes and mouse clicks.

AWT and Swing programs attach event listeners to UI objects. These listeners are notified when a specific event occurs, such as a button being clicked. Event listeners are called from within the AWT event thread.

If an event listener were to perform a lengthy task, such as checking spelling in a large document, the event thread would be busy running the spelling checker, and thus would not be able to process additional UI events until the event listener completed. This would make the program appear to freeze, which is disconcerting to the user.

To avoid stalling the UI, the event listener should hand off long tasks to another thread so that the AWT thread can continue processing UI events (including requests to cancel the long-running task being performed) while the task is in progress.

Take advantage of multiprocessor systems

Multiprocessor (MP) systems are much more common than they used to be. Once they were found only in large data centers and scientific computing facilities. Now many low-end server systems -- and even some desktop systems -- have multiple processors.

Modern operating systems, including Linux, Solaris, and Windows NT/2000, can take advantage of multiple processors and schedule threads to execute on any available processor.

The basic unit of scheduling is generally the thread; if a program has only one active thread, it can only run on one processor at a time. If a program has multiple active threads, then multiple threads may be scheduled at once. In a well-designed program, using multiple threads can improve program throughput and performance.

Simplicity of modeling

In some cases, using threads can make your programs simpler to write and maintain. Consider a simulation application, where you simulate the interaction between multiple entities. Giving each entity its own thread can greatly simplify many simulation and modeling applications.

Another example where it is convenient to use separate threads to simplify a program is when an application has multiple independent event-driven components. for example, an application might have a component that counts down the number of seconds since some event and updates a display on the screen. Rather than having a main loop check the time periodically and update the display, it is much simpler -- and less error-prone -- to have a thread that does nothing but sleep until a certain amount of time has elapsed and then update the on-screen counter. This way the main thread doesn't need to worry about the timer at all.

Asynchronous or background processing

Server applications get their input from remote sources, such as sockets. When you read from a socket, if there is no data currently available, the call to `SocketInputStream.read()` will block until data is available.

If a single-threaded program were to read from the socket, and the entity on the other end of the socket were never to send any data, the program would simply wait forever, and no other processing would get done. On the other hand, the program could poll the socket to see if data was available, but this is often undesirable for performance reasons.

If, instead, you created a thread to read from the socket, the main thread could perform other tasks while the other thread waited for input from the socket. You can even create multiple threads so you can read from multiple sockets at once. In this way, you are notified quickly when data is available (because the waiting thread is awakened) without having to poll frequently to check if data is available. The code to wait on a socket using threads is also much simpler and less error-prone than polling would be.

Simple, but sometimes risky

While the Java thread facility is very easy to use, there are several risks you should try to avoid when you create multithreaded programs.

When multiple threads access the same data item, such as a static field, an instance field of a globally accessible object, or a shared collection, you need to make sure that they coordinate their access to the data so that both see a consistent view of the data and neither steps on the other's changes. The Java language provides two keywords for this purpose: `synchronized` and `volatile`. We will explore the use and meaning of these keywords later in this tutorial.

When accessing variables from more than one thread, you must ensure that the access is properly synchronized. For simple variables, it may be enough to declare the variable `volatile`, but in most situations, you will need to use synchronization.

If you are going to use synchronization to protect access to shared variables, you must make sure to use it *everywhere* in your program where the variable is accessed.

Don't overdo it

While threads can greatly simplify many types of applications, overuse of threads can be hazardous to your program's performance and its maintainability. Threads consume resources. Therefore, there is a limit on how many threads you can create without degrading performance.

In particular, using multiple threads will *not* make a CPU-bound program run any faster on a single-processor system.

Example: Using a thread for timing and a thread to do work

The following example uses two threads, one for timing and one to do actual work. The main thread calculates prime numbers using a very straightforward algorithm.

Before it starts, it creates and starts a timer thread, which will sleep for ten seconds, and then set a flag that the main thread will check. After ten seconds, the main thread will stop. Note that the shared flag is declared `volatile`.

```
/**
 * CalculatePrimes -- calculate as many primes as we can in ten seconds
 */

public class CalculatePrimes extends Thread {

    public static final int MAX_PRIMES = 1000000;
    public static final int TEN_SECONDS = 10000;

    public volatile boolean finished = false;

    public void run() {
        int[] primes = new int[MAX_PRIMES];
        int count = 0;

        for (int i=2; count<MAX_PRIMES; i++) {

            // Check to see if the timer has expired
            if (finished) {
                break;
            }

            boolean prime = true;
            for (int j=0; j<count; j++) {
                if (i % primes[j] == 0) {
                    prime = false;
```

```
                    break;
                }
            }

            if (prime) {
                primes[count++] = i;
                System.out.println("Found prime: " + i);
            }
        }
    }

    public static void main(String[] args) {
        CalculatePrimes calculator = new CalculatePrimes();
        calculator.start();
        try {
            Thread.sleep(TEN_SECONDS);
        }
        catch (InterruptedException e) {
            // fall through
        }

        calculator.finished = true;
    }
}
```

Summary

The Java language includes a powerful threading facility built into the language. You can use the threading facility to:

- Increase the responsiveness of GUI applications
- Take advantage of multiprocessor systems
- Simplify program logic when there are multiple independent entities
- Perform blocking I/O without blocking the entire program

When you use multiple threads, you must be careful to follow the rules for sharing data between threads, which we'll cover in Sharing access to data on page 16 . All these rules boil down to one basic principle: **Don't forget to synchronize**.

Section 3. A thread's life

Creating threads

There are several ways to create a thread in a Java program. Every Java program contains at least one thread: the main thread. Additional threads are created through the `Thread` constructor or by instantiating classes that extend the `Thread` class.

Java threads can create other threads by instantiating a `Thread` object directly or an object that extends `Thread`. In the example in Thread basics on page 3 , in which we calculated as many primes as we could in ten seconds, we created a thread by instantiating an object of type `CalculatePrimes`, which extends `Thread`.

When we talk about threads in Java programs, there are two related entities we may be referring to: the actual thread that is doing the work or the `Thread` object that represents the thread. The running thread is generally created by the operating system; the `Thread` object is created by the Java VM as a means of controlling the associated thread.

Creating threads and starting threads are not the same

A thread doesn't actually begin to execute until another thread calls the `start()` method on the `Thread` object for the new thread. The `Thread` object exists before its thread actually starts, and it continues to exist after its thread exits. This allows you to control or obtain information about a thread you've created, even if the thread hasn't started yet or has already completed.

It's generally a bad idea to `start()` threads from within a constructor. Doing so could expose partially constructed objects to the new thread. If an object owns a thread, then it should provide a `start()` or `init()` method that will start the thread, rather than starting it from the constructor. (See Resources on page 30 for links to articles that provide a more detailed explanation of this concept.)

Ending threads

A thread will end in one of three ways:

- The thread comes to the end of its `run()` method.
- The thread throws an `Exception` or `Error` that is not caught.
- Another thread calls one of the deprecated `stop()` methods. Deprecated means they still exist, but you shouldn't use them in new code and should strive to eliminate them in existing code.

When all the threads within a Java program complete, the program exits.

Joining with threads

The Thread API contains a method for waiting for another thread to complete: the `join()` method. When you call `Thread.join()`, the calling thread will block until the target thread completes.

`Thread.join()` is generally used by programs that use threads to partition large problems into smaller ones, giving each thread a piece of the problem. The example at the end of this section creates ten threads, starts them, then uses `Thread.join()` to wait for them all to complete.

Scheduling

Except when using `Thread.join()` and `Object.wait()`, the timing of thread scheduling and execution is nondeterministic. If two threads are running at the same time and neither is waiting, you must assume that between any two instructions, other threads may be running and modifying program variables. If your thread will be accessing data that may be visible to other threads, such as data referenced directly or indirectly from static fields (global variables), you must use synchronization to ensure data consistency.

In the simple example below, we'll create and start two threads, each of which prints two lines to `System.out`:

```
public class TwoThreads {

    public static class Thread1 extends Thread {
        public void run() {
            System.out.println("A");
            System.out.println("B");
        }
    }

    public static class Thread2 extends Thread {
        public void run() {
            System.out.println("1");
            System.out.println("2");
        }
    }

    public static void main(String[] args) {
        new Thread1().start();
        new Thread2().start();
    }
}
```

We have no idea in what order the lines will execute, except that "1" will be printed before "2"

and "A" before "B." The output could be any one of the following:

- 1 2 A B
- 1 A 2 B
- 1 A B 2
- A 1 2 B
- A 1 B 2
- A B 1 2

Not only may the results vary from machine to machine, but running the same program multiple times on the same machine may produce different results. Never assume one thread will do something before another thread does, unless you've used synchronization to force a specific ordering of execution.

Sleeping

The Thread API includes a `sleep()` method, which will cause the current thread to go into a wait state until the specified amount of time has elapsed or until the thread is interrupted by another thread calling `Thread.interrupt()` on the current thread's `Thread` object. When the specified time elapses, the thread again becomes runnable and goes back onto the scheduler's queue of runnable threads.

If a thread is interrupted by a call to `Thread.interrupt()`, the sleeping thread will throw an `InterruptedException` so that the thread will know that it was awakened by an interrupt and won't have to check to see if the timer expired.

The `Thread.yield()` method is like `Thread.sleep()`, but instead of sleeping, it simply pauses the current thread momentarily so that other threads can run. In most implementations, threads with lower priority will not run when a thread of higher priority calls `Thread.yield()`.

The `CalculatePrimes` example used a background thread to calculate primes, then slept for ten seconds. When the timer expired, it set a flag to indicate that the ten seconds had expired.

Daemon threads

We mentioned that a Java program exits when all of its threads have completed, but this is not exactly correct. What about the hidden system threads, such as the garbage collection thread and others created by the JVM? We have no way of stopping these. If those threads are running, how does any Java program ever exit?

These system threads are called *daemon* threads. A Java program actually exits when all its non-daemon threads have completed.

Any thread can become a daemon thread. You can indicate a thread is a daemon thread by

calling the `Thread.setDaemon()` method. You might want to use daemon threads for background threads that you create in your programs, such as timer threads or other deferred event threads, which are only useful while there are other non-daemon threads running.

Example: Partitioning a large task with multiple threads

In this example, `TenThreads` shows a program that creates ten threads, each of which do some work. It waits for them all to finish, then gathers the results.

```
/**
 * Creates ten threads to search for the maximum value of a large matrix.
 * Each thread searches one portion of the matrix.
 */
public class TenThreads {

    private static class WorkerThread extends Thread {
        int max = Integer.MIN_VALUE;
        int[] ourArray;

        public WorkerThread(int[] ourArray) {
            this.ourArray = ourArray;
        }

        // Find the maximum value in our particular piece of the array
        public void run() {
            for (int i = 0; i < ourArray.length; i++)
                max = Math.max(max, ourArray[i]);
        }

        public int getMax() {
            return max;
        }
    }

    public static void main(String[] args) {
        WorkerThread[] threads = new WorkerThread[10];
        int[][] bigMatrix = getBigHairyMatrix();
        int max = Integer.MIN_VALUE;

        // Give each thread a slice of the matrix to work with
        for (int i=0; i < 10; i++) {
            threads[i] = new WorkerThread(bigMatrix[i]);
            threads[i].start();
        }

        // Wait for each thread to finish
        try {
            for (int i=0; i < 10; i++) {
                threads[i].join();
                max = Math.max(max, threads[i].getMax());
            }
        }
        catch (InterruptedException e) {
            // fall through
```

```
        }

        System.out.println("Maximum value was " + max);
    }
}
```

Summary

Like programs, threads have a life cycle: they start, they execute, and they complete. One program, or process, may contain multiple threads, which appear to execute independently of each other.

A thread is created by instantiating a `Thread` object, or an object that extends `Thread`, but the thread doesn't start to execute until the `start()` method is called on the new `Thread` object. Threads end when they come to the end of their `run()` method or throw an unhandled exception.

The `sleep()` method can be used to wait for a certain amount of time; the `join()` method can be used to wait until another thread completes.

Section 4. Threads everywhere

Who creates threads?

Even if you never explicitly create a new thread, you may find yourself working with threads anyway. Threads are introduced into our programs from a variety of sources.

There are a number of facilities and tools that create threads for you, and you should understand how threads interact and how to prevent threads from getting in the way of each other if you're going to use these facilities.

AWT and Swing

Any program that uses AWT or Swing must deal with threads. The AWT toolkit creates a single thread for handling UI events, and any event listeners called by AWT events execute in the AWT event thread.

Not only do you have to worry about synchronizing access to data items shared between event listeners and other threads, but you have to find a way for long-running tasks triggered by event listeners -- such as checking spelling in a large document or searching a file system for a file -- to run in a background thread so the UI doesn't freeze while the task is running (which would also prevent the user from canceling the operation). A good example of a framework for doing this is the `SwingWorker` class (see Resources on page 30).

The AWT event thread is not a daemon thread; this is why `System.exit()` is often used to end AWT and Swing apps.

Using TimerTask

The `TimerTask` facility was introduced to the Java language in JDK 1.3. This convenient facility allows you to execute a task at a later time (that is, for example, run a task once ten seconds from now), or to execute a task periodically (that is, run a task every ten seconds).

Implementing the `Timer` class is quite straightforward: it creates a timer thread and builds a queue of waiting events sorted by execution time.

The `TimerTask` thread is marked as a daemon thread so it doesn't prevent the program from exiting.

Because timer events execute in the timer thread, you must make sure that access to any data items used from within a timer task is properly synchronized.

In the `CalculatePrimes` example, instead of having the main thread sleep, we could have used a `TimerTask` as follows:

```
public static void main(String[] args) {
    Timer timer = new Timer();

    final CalculatePrimes calculator = new CalculatePrimes();
    calculator.start();

    timer.schedule(
            new TimerTask() {
                public void run()
                {
                    calculator.finished = true;
                }
            }, TEN_SECONDS);
}
```

Servlets and JavaServer Pages technology

Servlet containers create multiple threads in which servlet requests are executed. As the servlet writer, you have no idea (nor should you) in what thread your request will be executed; the same servlet could be active in multiple threads at once if multiple requests for the same URL come in at the same time.

When writing servlets or JavaServer Pages (JSP) files, you must assume at all times that the same servlet or JSP file may be executing concurrently in multiple threads. Any shared data accessed by a servlet or JSP file must be appropriately synchronized; this includes fields of the servlet object itself.

Implementing an RMI object

The RMI facility allows you to invoke operations on objects running in other JVMs. When you call a remote method, the RMI stub, created by the RMI compiler, packages up the method parameters and sends them over the network to the remote system, which unpacks them and calls the remote method.

Suppose you create an RMI object and register it in the RMI registry or a Java Naming and Directory Interface (JNDI) namespace. When a remote client invokes one of its methods, in what thread does that method execute?

The common way to implement an RMI object is to extend `UnicastRemoteObject`. When a `UnicastRemoteObject` is constructed, the infrastructure for dispatching remote method calls is initialized. This includes a socket listener to receive remote invocation requests, and one or more threads to execute remote requests.

So when you receive a request to execute an RMI method, these methods will execute in an RMI-managed thread.

Summary

Threads enter Java programs through several mechanisms. In addition to creating threads explicitly with the `Thread` constructors, threads are created by a variety of other mechanisms:

- AWT and Swing
- RMI
- The `java.util.TimerTask` facility
- Servlets and JSP technology

Section 5. Sharing access to data

Sharing variables

For multiple threads to be useful in a program, they have to have some way to communicate or share their results with each other.

The simplest way for threads to share their results is to use shared variables. They should also use synchronization to ensure that values are propagated correctly from one thread to another and to prevent threads from seeing inconsistent intermediate results while another thread is updating several related data items.

The example that calculated prime numbers in Thread basics on page 3 used a shared boolean variable to indicate that the specified time period had elapsed. This illustrates the simplest form of sharing data between threads: polling a shared variable to see if another thread has finished performing a certain task.

All threads live in the same memory space

As we discussed earlier, threads have a lot in common with processes, except that they share the same process context, including memory, with other threads in the same process. This is a tremendous convenience, but also a significant responsibility. Threads can easily exchange data among themselves simply by accessing shared variables (static or instance fields), but threads must also ensure that they access shared variables in a controlled manner, lest they step on each other's changes.

Any reachable variable is accessible to any thread, just as it is accessible to the main thread. The prime numbers example used a public instance field, called `finished`, to indicate that the specified time had elapsed. One thread wrote to this field when the timer expired; the other read from this field periodically to check if it should stop. Note that this field was declared `volatile`, which is important to the proper functioning of this program. We'll see why later in this section.

Synchronization for controlled access

The Java language provides two keywords for ensuring that data can be shared between threads in a controlled manner: `synchronized` and `volatile`.

`Synchronized` has two important meanings: it ensures that only one thread executes a protected section of code at one time (mutual exclusion or *mutex*), and it ensures that data changed by one thread is visible to other threads (visibility of changes).

Without synchronization, it is easy for data to be left in an inconsistent state. For example, if one thread is updating two related values (say, the position and velocity of a particle), and another thread is reading those two values, it is possible that the second thread could be

scheduled to run after the first thread has written one value but not the other, thus seeing one old and one new value. Synchronization allows us to define blocks of code that must run *atomically*, in which they appear to execute in an all-or-nothing manner, as far as other threads can tell.

The atomic execution or mutual exclusion aspect of synchronization is similar to the concept of *critical sections* in other operating environments.

Ensuring visibility of changes to shared data

Synchronization allows us to ensure that threads see consistent views of memory.

Processors can use caches to speed up access to memory (or compilers may store values in registers for faster access). On some multiprocessor architectures, if a memory location is modified in the cache on one processor, it is not necessarily visible to other processors until the writer's cache is flushed *and* the reader's cache is invalidated.

This means that on such systems, it is possible for two threads executing on two different processors to see two different values for the same variable! This sounds scary, but it *is* normal. It just means that you have to follow some rules when accessing data used or modified by other threads.

`Volatile` is simpler than synchronization and is suitable only for controlling access to single instances of primitive variables -- integers, booleans, and so on. When a variable is declared `volatile`, any write to that variable will go directly to main memory, bypassing the cache, while any read of that variable will come directly from main memory, bypassing the cache. This means that all threads see the same value for a `volatile` variable at all times.

Without proper synchronization, it is possible for threads to see stale values of variables or experience other forms of data corruption.

Atomic code blocks protected by locks

`Volatile` is useful for ensuring that each thread sees the most recent value for a variable, but sometimes we need to protect access to larger sections of code, such as sections that involve updating multiple variables.

Synchronization uses the concepts of *monitors*, or locks, to coordinate access to particular blocks of code.

Every Java object has an associated lock. Java locks can be held by no more than one thread at a time. When a thread enters a `synchronized` block of code, the thread blocks and waits until the lock is available, acquires the lock when it becomes available, and then executes the block of code. It releases the lock when control exits the protected block of code, either by reaching the end of the block or when an exception is thrown that is not caught within the

`synchronized` block.

In this way, only one thread can execute a block protected by a given monitor at one time. The block can be considered *atomic* because, from the perspective of other threads, it appears to either have executed entirely or not at all.

A simple synchronization example

Using `synchronized` blocks allows you to perform a group of related updates as a set without worrying about other threads interrupting or seeing the intermediate results of a computation. The following example code will either print "1 0" or "0 1." In the absence of synchronization, it could also print "1 1" (or even "0 0," believe it or not).

```
public class SyncExample {
  private static lockObject = new Object();
  private static class Thread1 extends Thread {
    public void run() {
      synchronized (lockObject) {
        x = y = 0;
        System.out.println(x);
      }
    }
  }

  private static class Thread2 extends Thread {
    public void run() {
      synchronized (lockObject) {
        x = y = 1;
        System.out.println(y);
      }
    }
  }

  public static void main(String[] args) {
    new Thread1().run();
    new Thread2().run();
  }
}
```

You must use synchronization in *both* threads for this program to work properly.

Java locking

Java locking incorporates a form of mutual exclusion. Only one thread may hold a lock at one time. Locks are used to protect blocks of code or entire methods, but it is important to remember that it is the identity of the lock that protects a block of code, not the block itself. One lock may protect many blocks of code or methods.

Conversely, just because a block of code is protected by a lock does not mean that two threads cannot execute that block at once. It only means that two threads cannot execute that block at once *if they are waiting on the same lock.*

In the following example, the two threads are free to execute the `synchronized` block in `setLastAccess()` simultaneously because each thread has a different value for `thingie`. Therefore, the `synchronized` block is protected by different locks in the two executing threads.

```
public class SyncExample {
  public static class Thingie {

    private Date lastAccess;

    public synchronized void setLastAccess(Date date) {
      this.lastAccess = date;
    }
  }

  public static class MyThread extends Thread {
    private Thingie thingie;

    public MyThread(Thingie thingie) {
      this.thingie = thingie;
    }

    public void run() {
      thingie.setLastAccess(new Date());
    }
  }

  public static void main() {
    Thingie thingie1 = new Thingie(),
      thingie2 = new Thingie();

    new MyThread(thingie1).start();
    new MyThread(thingie2).start();
  }
}
```

Synchronized methods

The simplest way to create a `synchronized` block is to declare a method as `synchronized`. This means that before entering the method body, the caller must acquire a lock:

```
public class Point {
  public synchronized void setXY(int x, int y) {
    this.x = x;
    this.y = y;
  }
```

```
}
```

For ordinary `synchronized` methods, this lock will be the object on which the method is being invoked. For static `synchronized` methods, this lock will be the monitor associated with the `Class` object in which the method is declared.

Just because `setXY()` is declared as `synchronized` doesn't mean that two different threads can't still execute `setXY()` at the same time, as long as they are invoking `setXY()` on different `Point` instances. Only one thread can execute `setXY()`, or any other `synchronized` method of `Point`, on a single `Point` instance at one time.

Synchronized blocks

The syntax for `synchronized` blocks is a little more complicated than for `synchronized` methods because you also need to explicitly specify what lock is being protected by the block. The following version of `Point` is equivalent to the version shown in the previous panel:

```
public class Point {
  public void setXY(int x, int y) {
    synchronized (this) {
      this.x = x;
      this.y = y;
    }
  }
}
```

It is common, but not required, to use the `this` reference as the lock. This means that the block will use the same lock as `synchronized` methods in that class.

Because synchronization prevents multiple threads from executing a block at once, it has performance implications, even on uniprocessor systems. It is a good practice to use synchronization around the smallest possible block of code that needs to be protected.

Access to local (stack-based) variables never need to be protected, because they are only accessible from the owning thread.

Most classes are not synchronized

Because synchronization carries a small performance penalty, most general-purpose classes, like the Collection classes in `java.util`, do not use synchronization internally. This means that classes like `HashMap` cannot be used from multiple threads without additional synchronization.

You can use the Collections classes in a multithreaded application by using synchronization every time you access a method in a shared collection. For any given collection, you must

synchronize on the same lock each time. A common choice of lock would be the collection object itself.

The example class `SimpleCache` in the next panel shows how you can use a `HashMap` to provide caching in a thread-safe way. Generally, however, proper synchronization doesn't just mean synchronizing every method.

The `Collections` class provides us with a set of convenience wrappers for the `List`, `Map`, and `Set` interfaces. You can wrap a `Map` with `Collections.synchronizedMap` and it will ensure that all access to that map is properly synchronized.

If the documentation for a class does not say that it is thread-safe, then you must assume that it is not.

Example: A simple thread-safe cache

As shown in the following code sample, `SimpleCache.java` uses a `HashMap` to provide a simple cache for an object loader. The `load()` method knows how to load an object by its key. After an object is loaded once, it is stored in the cache so subsequent accesses will retrieve it from the cache instead of loading it all over each time. Each access to the shared cache is protected by a `synchronized` block. Because it is properly synchronized, multiple threads can call the `getObject` and `clearCache` methods simultaneously without risk of data corruption.

```
public class SimpleCache {
  private final Map cache = new HashMap();

  public Object load(String objectName) {
    // load the object somehow
  }

  public void clearCache() {
    synchronized (cache) {
      cache.clear();
    }
  }

  public Object getObject(String objectName) {
    synchronized (cache) {
      Object o = cache.get(objectName);
      if (o == null) {
        o = load(objectName);
        cache.put(objectName, o);
      }
    }

    return o;
  }
}
```

Summary

Because the timing of thread execution is nondeterministic, we need to be careful to control a thread's access to shared data. Otherwise, multiple concurrent threads could step on each other's changes and result in corrupted data, or changes to shared data might not be made visible to other threads on a timely basis.

By using synchronization to protect access to shared variables, we can ensure that threads interact with program variables in predictable ways.

Every Java object can act as a lock, and `synchronized` blocks can ensure that only one thread executes `synchronized` code protected by a given lock at one time.

Section 6. Synchronization details

Mutual exclusion

In Sharing access to data on page 16 , we discussed the characteristics of `synchronized` blocks and described them as implementing a *classic mutex* (that is, mutual exclusion or critical section), in which only one thread can execute a block protected by a given lock at one time.

Mutual exclusion is an important part of what synchronization does, but synchronization also has several other characteristics important to achieve correct results on multiprocessor systems.

Visibility

In addition to mutual exclusion, synchronization, like `volatile`, enforces certain visibility constraints. When an object acquires a lock, it first invalidates its cache, so that it is guaranteed to load variables directly from main memory.

Similarly, before an object releases a lock, it flushes its cache, forcing any changes made to appear in main memory.

In this way, two threads that synchronize on the same lock are guaranteed to see the same values in variables modified inside a `synchronized` block.

When do you have to synchronize?

To maintain proper visibility across threads, you have to use `synchronized` (or `volatile`) to ensure that changes made by one thread are visible by another whenever a non-final variable is shared among several threads.

The basic rule for synchronizing for visibility is that you must synchronize whenever you are:

- Reading a variable that may have been last written by another thread
- Writing a variable that may be read next by another thread

Synchronization for consistency

In addition to synchronizing for visibility, you must also synchronize to ensure that consistency is maintained from an application perspective. When modifying multiple related values, you want other threads to see that set of changes atomically -- either all of the changes or none of them. This applies to related data items (such as the position and velocity of a particle) and metadata items (such as the data values contained in a linked list and the chain of entries in

the list itself).

Consider the following example, which implements a simple (but not thread-safe) stack of integers:

```
public class UnsafeStack {
  public int top = 0;
  public int[] values = new int[1000];

  public void push(int n) {
    values[top++] = n;
  }

  public int pop() {
    return values[--top];
  }
}
```

What happens if more than one thread tries to use this class at the same time? It could be a disaster. Because there's no synchronization, multiple threads could execute `push()` and `pop()` at the same time. What if one thread calls `push()` and another thread calls `push()` right between the time `top` is incremented and it is used as an index into `values`? Then both threads would store their new value in the same location! This is just one of many forms of data corruption that can occur when multiple threads rely on a known relationship between data values, but don't ensure that only one thread is manipulating those values at a given time.

In this case, the cure is simple: synchronize both `push()` and `pop()`, and you'll prevent one thread from stepping on another.

Note that using `volatile` would not have been enough -- you need to use `synchronized` to ensure that the relationship between `top` and `values` remains consistent.

Incrementing a shared counter

In general, if you're protecting a single primitive variable, such as an integer, you can sometimes get away with just using `volatile`. However, if new values of the variable are derived from previous values, you will have to use synchronization. Why? Consider this class:

```
public class Counter {
  private int counter = 0;

  public int  get()      { return counter; }
  public void set(int n) { counter = n; }
  public void increment() {
    set(get() + 1);
  }
}
```

What happens when we want to increment the counter? Look at the code for `increment()`. It is clear, but it is not thread-safe. What happens if two threads try to execute `increment()` at the same time? The counter might be incremented by 1 or by 2. Surprisingly, marking `counter` as `volatile` won't help, nor will making both `get()` and `set()` `synchronized`.

Imagine that the counter is zero and two threads execute the increment code at exactly the same time. Both threads call `Counter.get()` and see that the counter is zero. Now both add one to this and then call `Counter.set()`. If our timing is unlucky, neither will see the other's update, even if `counter` is `volatile` or `get()` and `set()` are `synchronized`. Now, even though we've incremented the counter twice, the resulting value may only be one instead of two.

For increment to work properly, not only do `get()` and `set()` have to be `synchronized`, but `increment()` needs to be `synchronized`, too! Otherwise, a thread calling `increment()` could interrupt another thread calling `increment()`. If you're unlucky, the end result would be that the counter is incremented once instead of twice. Synchronizing `increment()` prevents this from happening, because the entire increment operation is atomic.

The same is true when you are iterating through the elements of a `Vector`. Even though the methods of `Vector` are synchronized, the contents of `Vector` could still change while you are iterating. If you want to ensure that the contents of `Vector` don't change while you're iterating through it, you have to wrap the entire block with synchronization on it.

Immutability and final fields

Many Java classes, including `String`, `Integer`, and `BigDecimal`, are immutable: once constructed, their state never changes. A class would be immutable if all of its fields were declared `final`. (In practice, many immutable classes have non-final fields to cache previously computed method results, like `String.hashCode()`, but this is invisible to the caller.)

Immutable classes make concurrent programming substantially simpler. Because their fields do not change, you do not need to worry about propagating changes in state from one thread to another. Once the object is properly constructed, it can be considered constant.

Similarly, final fields are also more thread-friendly. Because final fields cannot change their value once initialized, you do not need to synchronize access when sharing final fields across threads.

When you don't need to synchronize

There are a few cases where you do not have to synchronize to propagate data from one thread to another, because the JVM is implicitly performing the synchronization for you. These cases include:

- When data is initialized by a static initializer (an initializer on a static field or in a `static{}` block)
- When accessing final fields
- When an object is created before a thread is created
- When an object is already visible to a thread that it is then joined with

Deadlock

Whenever you have multiple processes contending for exclusive access to multiple locks, there is the possibility of deadlock. A set of processes or threads is said to be *deadlocked* when each is waiting for an action that only one of the others can perform.

The most common form of deadlock is when Thread 1 holds a lock on Object A and is waiting for the lock on Object B, and Thread 2 holds the lock on Object B and is waiting for the lock on Object A. Neither thread will ever acquire the second lock or relinquish the first lock. They will simply wait forever.

To avoid deadlock, you should ensure that when you acquire multiple locks, you always acquire the locks in the same order in all threads.

Performance considerations

There has been a lot written -- much of it wrong -- on the performance costs of synchronization. It is true that synchronization, especially contended synchronization, has performance implications, but these may not be as large as is widely suspected.

Many people have gotten themselves in trouble by using fancy but ineffective tricks to try to avoid having to synchronize. One classic example is the double-checked locking pattern (see Resources on page 30 for several articles on what's wrong with it). This harmless-looking construct purported to avoid synchronization on a common code path, but was subtly broken, and all attempts to fix it were also broken.

When writing concurrent code, don't worry so much about performance until you've actually seen evidence of performance problems. Bottlenecks appear in the places we often least suspect. Speculatively optimizing one code path that may not even turn out to be a performance problem -- at the cost of program correctness -- is a false economy.

Guidelines for synchronization

There are a few simple guidelines you can follow when writing `synchronized` blocks that will go a long way toward helping you to avoid the risks of deadlock and performance hazards:

- **Keep blocks short.** `Synchronized` blocks should be short -- as short as possible while

still protecting the integrity of related data operations. Move thread-invariant preprocessing and postprocessing out of `synchronized` blocks.

- **Don't block.** Don't ever call a method that might block, such as `InputStream.read()`, inside a `synchronized` block or method.

- **Don't invoke methods on other objects while holding a lock.** This may sound extreme, but it eliminates the most common source of deadlock.

Section 7. Additional thread API details

wait(), notify(), and notifyAll() methods

In addition to using polling, which can consume substantial CPU resources and has imprecise timing characteristics, the `Object` class includes several methods for threads to signal events from one thread to another.

The `Object` class defines the methods `wait()`, `notify()`, and `notifyAll()`. To execute any of these methods, you must be holding the lock for the associated object.

`Wait()` causes the calling thread to sleep until it is interrupted with `Thread.interrupt()`, the specified timeout elapses, or another thread wakes it up with `notify()` or `notifyAll()`.

When `notify()` is invoked on an object, if there are any threads waiting on that object via `wait()`, then one thread will be awakened. When `notifyAll()` is invoked on an object, all threads waiting on that object will be awakened.

These methods are the building blocks of more sophisticated locking, queuing, and concurrency code. However, the use of `notify()` and `notifyAll()` is complicated. In particular, using `notify()` instead of `notifyAll()` is risky. Use `notifyAll()` unless you really know what you're doing.

Rather than use `wait()` and `notify()` to write your own schedulers, thread pools, queues, and locks, you should use the `util.concurrent` package (see Resources on page 30), a widely used open source toolkit full of useful concurrency utilities. JDK 1.5 will include the `java.util.concurrent` package; many of its classes are derived from `util.concurrent`.

Thread priorities

The Thread API allows you to associate an execution priority with each thread. However, how these are mapped to the underlying operating system scheduler is implementation-dependent. In some implementations, multiple -- or even all -- priorities may be mapped to the same underlying operating system priority.

Many people are tempted to tinker with thread priorities when they encounter a problem like deadlock, starvation, or other undesired scheduling characteristics. More often than not, however, this just moves the problem somewhere else. Most programs should simply avoid changing thread priority.

Thread groups

The `ThreadGroup` class was originally intended to be useful in structuring collections of threads into groups. However, it turns out that `ThreadGroup` is not all that useful. You are

better off simply using the equivalent methods in `Thread`.

`ThreadGroup` does offer one useful feature not (yet) present in `Thread`: the `uncaughtException()` method. When a thread within a thread group exits because it threw an uncaught exception, the `ThreadGroup.uncaughtException()` method is called. This gives you an opportunity to shut down the system, write a message to a log file, or restart a failed service.

SwingUtilities

Although it is not part of the `Thread` API, the `SwingUtilities` class deserves a brief mention.

As mentioned earlier, Swing applications have a single UI thread (sometimes called the event thread) in which all UI activity must occur. Sometimes another thread may want to update the appearance of something on the screen or fire an event on a Swing object.

The `SwingUtilities.invokeLater()` method allows you to pass a `Runnable` object to it, and the specified `Runnable` will be executed in the event thread. Its cousin, `invokeAndWait()`, will invoke `Runnable` in the event thread, but `invokeAndWait()` will block until `Runnable` is finished executing.

```
void showHelloThereDialog() throws Exception {
    Runnable showModalDialog = new Runnable() {
        public void run() {
            JOptionPane.showMessageDialog(myMainFrame, "Hello There");
        }
    };
    SwingUtilities.invokeLater(showModalDialog);
}
```

For AWT applications, `java.awt.EventQueue` also provides `invokeLater()` and `invokeAndWait()`.

Section 8. Wrapup and resources

Summary

Every Java program uses threads, whether you know it or not. If you are using either of the Java UI toolkits (AWT or Swing), Java Servlets, RMI, or JavaServer Pages or Enterprise JavaBeans technologies, you may be using threads without realizing it.

There are a number of situations where you might want to explicitly use threads to improve the performance, responsiveness, or organization of your programs. These include:

- Making the user interface more responsive when performing long tasks
- Exploiting multiprocessor systems to handle multiple tasks in parallel
- Simplifying modeling of simulations or agent-based systems
- Performing asynchronous or background processing

While the thread API is simple, writing thread-safe programs is not. When variables are shared across threads, you must take great care to ensure that you have properly synchronized both read and write access to them. When writing a variable that may next be read by another thread, or reading a variable that may have been written by another thread, you must use synchronization to ensure that changes to data are visible across threads.

When using synchronization to protect shared variables, you must ensure that not only are you using synchronization, but the reader and writer are synchronizing on the *same* monitor. Furthermore, if you rely on an object's state remaining the same across multiple operations, or rely on multiple variables staying consistent with each other (or consistent with their own past values), you must use synchronization to enforce this. But simply synchronizing every method in a class does not make it thread safe -- it just makes it more prone to deadlock.

Resources

Downloads

- Explore Doug Lea's `util.concurrent` package (http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html), which contains a wealth of useful classes for building efficient concurrent applications.

Articles and tutorials

- "*Synchronization and the Java Memory Model*" (http://gee.cs.oswego.edu/dl/cpj/jmm.html) is an excerpt from Doug Lea's book that focuses on the actual meaning of `synchronized`.

- In his article "*Writing multithreading Java applications*" (*developerWorks*, February 2001,

http://www-106.ibm.com/developerworks/library/j-thread.html), Alex Roetter outlines issues involved in Java multithreading and offers solutions to common problems.

- "*Threading lightly, Part 1: Synchronization is not the enemy*" by Brian Goetz (*developerWorks*, July 2001, http://www-106.ibm.com/developerworks/library/j-threads1/) explores how to manage the performance of concurrent applications.

- "*Achieve strong performance with threads*" by Jeff Friesen (*JavaWorld*, May 2002, http://www.javaworld.com/javaworld/jw-05-2002/jw-0503-java101.html) is a four-part tutorial on using threads.

- "*Double-checked locking: Clever, but broken* " (*JavaWorld*, February 2001, http://www.javaworld.com/jw-02-2001/jw-0209-double.html), explores the Java Memory Model in detail and the surprising consequences of failing to synchronize in certain situations.

- Thread safety is tricky stuff. "*Java theory and practice: Safe construction techniques*" (*developerWorks*, June 2002, http://www-106.ibm.com/developerworks/library/j-jtp0618.html) offers some tips for safely constructing objects.

- In "*Threads and Swing*" (http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html), the Sun folks explore the rules of safe Swinging and introduce the useful `SwingWorker` class.

Recommended books

- Doug Lea's *Concurrent Programming in Java, Second Edition* (Addison-Wesley, 1999, http://www.amazon.com/exec/obidos/ASIN/0201310090/none0b69) is a masterful book on the subtle issues surrounding multithreaded programming in Java applications.

- Paul Hyde's *Java Thread Programming* (http://www.amazon.com/exec/obidos/ASIN/0672315858/none0b69) is a nice tutorial and reference for many real-world multithreading issues.

- Allen Holub's book *Taming Java Threads* (http://www.amazon.com/exec/obidos/ASIN/1893115100/none0b69) is an enjoyable introduction to the challenges of Java thread programming.

Additional resources

- The `util.concurrent` package is being formalized under Java Community Process *JSR 166* (http://www.jcp.org/jsr/detail/166.jsp) for inclusion in the 1.5 release of the JDK.

- The *Foxtrot project* (http://foxtrot.sourceforge.net/) is an alternative -- and possibly simpler -- approach to threading in Swing applications.

- You'll find hundreds of articles about every aspect of Java programming in the *developerWorks Java technology zone* (http://www-106.ibm.com/developerworks/java/).

- See the *developerWorks Java technology tutorials page* (http://www-105.ibm.com/developerworks/education.nsf/dw/java-onlinecourse-bytitle?OpenDocument&Coun for a complete listing of free tutorials from developerWorks.

Feedback

Please send us your feedback on this tutorial. We look forward to hearing from you!

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.