

Concurrency in JDK 5.0

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.

| | |
|--|----|
| 1. About this tutorial | 2 |
| 2. Concurrency basics | 5 |
| 3. Thread-safe collections | 10 |
| 4. Task management | 15 |
| 5. Synchronizer classes | 25 |
| 6. Low-level facilities -- Lock and Atomic | 30 |
| 7. Performance and scalability | 33 |
| 8. Wrap-up and resources | 37 |

Section 1. About this tutorial

What is this tutorial about?

JDK 5.0 is a major step forward for the creation of highly scalable concurrent applications in the Java language. The JVM has been improved to allow classes to take advantage of hardware-level concurrency support, and a rich set of new concurrency building blocks has been provided to make it easier to develop concurrent applications.

This tutorial covers the new utility classes for concurrency provided by JDK 5.0 and demonstrates how these classes offer improved scalability compared to the existing concurrency primitives (`synchronized`, `wait()`, and `notify()`).

Should I take this tutorial?

While this tutorial is aimed at a wide range of levels, it is assumed that readers have a basic understanding of threads, concurrency, and the concurrency primitives provided by the Java language, particularly the semantics and correct use of synchronization.

Beginning readers may wish to first consult the "Introduction to Java Threads" tutorial (see [Resources](#) on page 37), or read the concurrency chapter of a general purpose introductory text on the Java language.

Many of the classes in the `java.util.concurrent` package use generics, as `java.util.concurrent` has other strong dependencies on the JDK 5.0 JVM. Users not familiar with generics may wish to consult resources on the new generics facility in JDK 5.0. (For those not familiar with generics, you may find it useful to simply ignore whatever is inside the angle brackets in class and method signatures in your first pass through this tutorial.)

What's new in JDK 5.0 for concurrency

The `java.util.concurrent` package contains a wealth of thread-safe, well-tested, high-performance concurrent building blocks. The goal for the creation of `java.util.concurrent` was, quite immodestly, to do for concurrency what the Collections framework did for data structures. By providing a set of reliable, high-performance concurrency building blocks,

developers can improve the thread safety, scalability, performance, readability, and reliability of their concurrent classes.

If some of the class names look familiar, it is probably because many of the concepts in `java.util.concurrent` are derived from Doug Lea's `util.concurrent` library (see [Resources](#) on page 37).

The improvements for concurrency in JDK 5.0 can be divided into three groups:

- **JVM-level changes.** Most modern processors have some hardware-level support for concurrency, usually in the form of a *compare-and-swap* (CAS) instruction. CAS is a low-level, fine-grained technique for allowing multiple threads to update a single memory location while being able to detect and recover from interference from other threads. It is the basis for many high-performance concurrent algorithms. Prior to JDK 5.0, the only primitive in the Java language for coordinating access between threads was synchronization, which was more heavyweight and coarse-grained. Exposing CAS makes it possible to develop highly scalable concurrent Java classes. These changes are intended primarily for use by JDK library classes, not by developers.
- **Low-level utility classes -- locking and atomic variables.** Using CAS as a concurrency primitive, the `ReentrantLock` class provides identical locking and memory semantics as the `synchronized` primitive, while offering better control over locking (such as timed lock waits, lock polling, and interruptible lock waits) and better scalability (higher performance under contention). Most developers will not use the `ReentrantLock` class directly, but instead will use the high-level classes built atop it.
- **High-level utility classes.** These are classes that implement the concurrency building blocks described in every computer science text -- semaphores, mutexes, latches, barriers, exchangers, thread pools, and thread-safe collection classes. Most developers will be able to use these classes to replace many, if not all, uses of synchronization, `wait()`, and `notify()` in their applications, likely to the benefit of performance, readability, and correctness.

Roadmap

This tutorial will focus primarily on the higher-level utility classes provided by the `java.util.concurrent` package -- thread-safe collections, thread pools, and synchronization utilities. These are classes that both novices and experts can use "out of the box."

In the first section, we'll review the basics of concurrency, although it should not substitute for an understanding of threads and thread safety. Readers who are not familiar with threading at all should probably first consult an introduction to threads, such as the "Introduction to Java Threads" tutorial (see [Resources](#) on page 37).

The next several sections explore the high-level utility classes in `java.util.concurrent` -- thread-safe collections, thread pools, semaphores, and synchronizers.

The final sections cover the low-level concurrency building blocks in `java.util.concurrent`, and offer some performance measurements showing the improved scalability of the new `java.util.concurrent` classes.

Environmental requirements

The `java.util.concurrent` package is tightly tied to JDK 5.0; there is no backport to previous JVM versions. The code examples in this tutorial will not compile or run on JVMs prior to 5.0, and many of the code examples use generics, enhanced-for, or other new language features from JDK 5.0.

About the author

Brian Goetz is a regular columnist on the developerWorks Java zone and has been a professional software developer and manager for the past 18 years. He is a Principal Consultant at Quiotix, a software development and consulting firm in Los Altos, California.

See Brian's [published and upcoming articles](#) (<http://www.briangoetz.com/pubs.html>) in popular industry publications.

Contact Brian at brian@quietix.com.

Section 2. Concurrency basics

What are threads?

All nontrivial operating systems support the concept of processes -- independently running programs that are isolated from each other to some degree.

Threads are sometimes referred to as *lightweight processes*. Like processes, they are independent, concurrent paths of execution through a program, and each thread has its own program counter, call stack, and local variables. However, threads exist within a process, and they share memory, file handles, and per-process state with other threads within the same process.

Nearly every operating system today also supports threads, allowing multiple, independently schedulable threads of execution to coexist within a single process. Because threads within a process execute within the same address space, multiple threads can simultaneously access the same objects, and they allocate objects from the same heap. While this makes it easier for threads to share information with each other, it also means that you must take care to ensure that threads do not interfere with each other.

When used correctly, threads enable a variety of benefits, including better resource utilization, simplified development, higher throughput, more responsive user interfaces, and the ability to perform asynchronous processing.

The Java language includes primitives for coordinating the behavior of threads so that shared variables can be accessed and modified safely without violating design invariants or corrupting data structures.

What are threads good for?

Many reasons exist for using threads in Java programs, and nearly every Java application uses threads, whether the developer knows it or not. Many J2SE and J2EE facilities create threads, such as RMI, Servlets, Enterprise JavaBeans components, and the Swing GUI toolkit.

Reasons for using threads include:

- **More responsive user interfaces.** Event-driven GUI toolkits, such as AWT or Swing, use a separate event thread to process GUI events. Event listeners, registered with GUI objects, are called from within the event

thread. However, if an event listener were to perform a lengthy task (such as spell-checking a document), the UI would appear to freeze, as the event thread would not be able to process other events until the lengthy task completed. By executing lengthy operations in a separate thread, the UI can continue to be responsive while lengthy background tasks are executing.

- **Exploiting multiple processors.** Multiprocessor (MP) systems are becoming cheaper and more widespread every year. Because the basic unit of scheduling is usually the thread, a single-threaded application can only run on a single processor at once, no matter how many processors are available. In a well-designed program, multiple threads can improve throughput and performance by better utilizing available computing resources.
- **Simplicity of modeling.** Using threads effectively can make your programs simpler to write and maintain. By the judicious use of threads, individual classes can be insulated from details of scheduling, interleaved operations, asynchronous IO and resource waits, and other complications. Instead, they can focus exclusively on the domain requirements, simplifying development and improving reliability.
- **Asynchronous or background processing.** Server applications may serve many simultaneous remote clients. If an application goes to read from a socket, and there is no data available to read, the call to `read()` will block until data is available. In a single-threaded application, this means that not only will processing the corresponding request stall, but processing all requests will stall while that single thread is blocked. However, if each socket had its own IO thread, then one thread blocking would have no effect on the behavior of other concurrent requests.

Thread safety

Ensuring that classes are thread-safe is difficult but necessary if those classes are to be used in a multithreaded environment. One of the goals of the `java.util.concurrent` specification process was to provide a set of thread-safe, high-performance concurrent building blocks, so that developers are relieved of some of the burden of writing thread-safe classes.

Defining thread-safety clearly is surprisingly hard, and most definitions seem downright circular. A quick Google search reveals the following examples of typical but unhelpful definitions (or, rather, descriptions) of thread-safe code:

- ... can be called from multiple programming threads without unwanted

interaction between the threads.

- . . . may be called by more than one thread at a time without requiring any other action on the caller's part.

With definitions like these, it's no wonder we're so confused by thread safety. These definitions are no better than saying "a class is thread-safe if it can be called safely from multiple threads." Which is, of course, what it means, but that doesn't help us tell a thread-safe class from an unsafe one. What do we mean by "safe"?

For a class to be thread-safe, it first must behave correctly in a single-threaded environment. If a class is correctly implemented, which is another way of saying that it conforms to its specification, no sequence of operations (reads or writes of public fields, and calls to public methods) on objects of that class should be able to put the object into an invalid state; observe the object to be in an invalid state; or violate any of the class's invariants, preconditions, or postconditions.

Furthermore, for a class to be thread-safe, it must continue to behave correctly (in the sense described above) when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment and without any additional synchronization on the part of the calling code. The effect is that operations on a thread-safe object will appear to all threads to occur in a fixed, globally consistent order.

In the absence of some sort of explicit coordination between threads, such as locking, the runtime is free to interleave the execution of operations in multiple threads as it sees fit.

Until JDK 5.0, the primary mechanism for ensuring thread safety was the `synchronized` primitive. Threads that access shared variables (those that are reachable by more than one thread) must use synchronization to coordinate both read and write access to shared variables. The `java.util.concurrent` package offers some alternate concurrency primitives, as well as a set of thread-safe utility classes that require no additional synchronization.

Concurrency, reluctantly

Even if your program never explicitly creates a thread, threads may be created on your behalf by a variety of facilities or frameworks, requiring that classes called from these threads be thread-safe. This can place a significant design and implementation burden on developers, as developing thread-safe classes requires more care and analysis than developing non-thread-safe classes.

AWT and Swing

These GUI toolkits create a background thread, called the event thread, from which listeners registered with GUI components will be called. Therefore, the classes that implement these listeners must be thread-safe.

TimerTask

The `TimerTask` facility, introduced in JDK 1.3, allows you to execute a task at a later time or schedule tasks for periodic execution. `TimerTask` events execute in the `Timer` thread, which means that tasks executed as `TimerTasks` must be thread-safe.

Servlets and JavaServer Pages technology

Servlet containers create multiple threads, and may call a given servlet simultaneously for multiple requests in multiple threads. Servlet classes must therefore be thread-safe.

RMI

The remote method invocation (RMI) facility allows you to invoke operations running in other JVMs. The most common way to implement a remote object is by extending `UnicastRemoteObject`. When a `UnicastRemoteObject` is instantiated, it is registered with the RMI dispatcher, which may create one or more threads in which remote methods will be executed. Therefore, remote classes must be thread-safe.

As you can see, many situations occur in which classes may be called from other threads, even if your application never explicitly creates a thread. Fortunately, the classes in `java.util.concurrent` can greatly simplify the task of writing thread-safe classes.

Example -- a non-thread-safe servlet

The following servlet looks like a harmless guestbook servlet, which saves the name of every visitor. However, this servlet is not thread-safe, and servlets are supposed to be thread-safe. The problem is that it uses a `HashSet` to store the name of the visitors, and `HashSet` is not a thread-safe class.

When we say this servlet is not thread-safe, the downside is not limited to losing a guestbook entry. In the worst case, our guestbook data structure could be irretrievably corrupted.

```
public class UnsafeGuestbookServlet extends HttpServlet {
```

```
private Set visitorSet = new HashSet();

protected void doGet(HttpServletRequest httpServletRequest,
    HttpServletResponse httpServletResponse) throws ServletException, IOException {
    String visitorName = httpServletRequest.getParameter("NAME");
    if (visitorName != null)
        visitorSet.add(visitorName);
    }
}
```

The class could be made thread-safe by changing the definition of visitorSet to

```
private Set visitorSet = Collections.synchronizedSet(new HashSet());
```

Examples like this one show how the built-in support for threading is a double-edged sword -- while it makes it easier to build multithreaded applications, it also requires developers to be more aware of concurrency issues, even when developing something as mundane as a guestbook servlet.

Section 3. Thread-safe collections

Introduction

The Collections framework, introduced in JDK 1.2, is a highly flexible framework for representing collections of objects, using the basic interfaces `List`, `Set`, and `Map`. Several implementations of each are provided by the JDK (`HashMap`, `Hashtable`, `TreeMap`, `WeakHashMap`, `HashSet`, `TreeSet`, `Vector`, `ArrayList`, `LinkedList`, and so on). Some of these are already thread-safe (`Hashtable` and `Vector`), and the remainder can be rendered thread-safe by the synchronized wrapper factories (`Collections.synchronizedMap()`, `synchronizedList()`, and `synchronizedSet()`).

The `java.util.concurrent` package adds several new thread-safe collection classes (`ConcurrentHashMap`, `CopyOnWriteArrayList`, and `CopyOnWriteArraySet`). The purpose of these classes is to provide high-performance, highly scalable, thread-safe versions of the basic collection types.

The thread-safe collections in `java.util` still have some drawbacks. For example, it is generally necessary to hold the lock on a collection while iterating it, otherwise you risk throwing `ConcurrentModificationException`. (This characteristic is sometimes called *conditional thread-safety*; see [Resources](#) on page 37 for more explanation.) Further, these classes often perform poorly if the collection is accessed frequently from multiple threads. The new collection classes in `java.util.concurrent` enable higher concurrency at the cost of some small changes in semantics.

JDK 5.0 also offers two new collection interfaces -- `Queue` and `BlockingQueue`. The `Queue` interface is similar to `List`, but permits insertion only at the tail and removal only from the head. By eliminating the random-access requirements from `List`, it becomes possible to create `Queue` implementations with better performance than the existing `ArrayList` and `LinkedList` implementations. Because many applications of `List` do not in fact need random access, `Queue` can often be substituted for `List`, with the result being better performance.

Weakly consistent iterators

The collections classes in the `java.util` package all return *fail-fast* iterators, which means that they assume a collection will not change its contents during the time a thread is iterating through its contents. If a fail-fast iterator detects

that a modification has been made during iteration, it throws `ConcurrentModificationException`, which is an unchecked exception.

The requirement that a collection not change during iteration is often inconvenient for many concurrent applications. Instead, it may be preferable to allow concurrent modification and ensure that iterators simply make a reasonable effort to provide a consistent view of the collection, as the iterators in the `java.util.concurrent` collection classes do.

The iterators returned by `java.util.concurrent` collections are called *weakly consistent* iterators. For these classes, if an element has been removed since iteration began, and not yet returned by the `next()` method, it will not be returned to the caller. If an element has been added since iteration began, it may or may not be returned to the caller. And no element will be returned twice in a single iteration, regardless of how the underlying collection has changed.

CopyOnWriteArrayList and CopyOnWriteArraySet

You can create a thread-safe array-backed `List` in two ways -- `Vector`, or wrapping an `ArrayList` with `Collections.synchronizedList()`. The `java.util.concurrent` package adds the awkwardly named `CopyOnWriteArrayList`. Why would we want a new thread-safe `List` class? Why isn't `Vector` sufficient?

The simple answer has to do with the interaction between iteration and concurrent modification. With `Vector` or with the synchronized `List` wrapper, the iterators returned are fail-fast, meaning that if any other threads modify the `List` during iteration, iteration may fail.

A very common application for `Vector` is to store a list of listeners registered with a component. When a suitable event occurs, the component will iterate through the list of listeners, calling each one. To prevent `ConcurrentModificationException`, the iterating thread must either copy the list or lock the list for the entire iteration -- both of which have a significant performance cost.

The `CopyOnWriteArrayList` class avoids this problem by creating a new copy of the backing array every time an element is added or removed, but iterations in progress keep working on the copy that was current at the time the iterator was created. While the copying has some cost as well, in many situations iterations greatly outnumber modifications, and in these cases the copy-on-write offers better performance and concurrency than the alternatives.

If your application requires the semantics of `Set` instead of `List`, there is a `Set` version as well -- `CopyOnWriteArraySet`.

ConcurrentHashMap

Just as there already exists implementations of `List` that are thread-safe, you can create a thread-safe hash-based `Map` in several ways -- `Hashtable` and wrapping a `HashMap` with `Collections.synchronizedMap()`. JDK 5.0 adds the `ConcurrentHashMap` implementation, which offers the same basic thread-safe `Map` functionality, but greatly improved concurrency.

The simple approach to synchronization taken by both `Hashtable` and `synchronizedMap` -- synchronizing each method on the `Hashtable` or the `synchronizedMap` wrapper object -- has two principal deficiencies. It is an impediment to scalability, because only one thread can access the hash table at a time. At the same time, it is insufficient to provide true thread safety, in that many common compound operations still require additional synchronization. While simple operations such as `get()` and `put()` can complete safely without additional synchronization, several common sequences of operations exist, such as iteration or `put-if-absent`, which still require external synchronization to avoid data races.

`Hashtable` and `Collections.synchronizedMap` achieve thread safety by synchronizing every method. This means that when one thread is executing one of the `Map` methods, other threads cannot until the first thread is finished, regardless of what they want to do with the `Map`.

By contrast, `ConcurrentHashMap` allows multiple reads to almost always execute concurrently, reads and writes to usually execute concurrently, and multiple simultaneous writes to often execute concurrently. The result is much higher concurrency when multiple threads need to access the same `Map`.

In most cases, `ConcurrentHashMap` is a drop-in replacement for `Hashtable` or `Collections.synchronizedMap(new HashMap())`. However, there is one significant difference -- synchronizing on a `ConcurrentHashMap` instance does not lock the map for exclusive use. In fact, there is no way to lock a `ConcurrentHashMap` for exclusive use -- it is designed to be accessed concurrently. To make up for the fact that the collection cannot be locked for exclusive use, additional (atomic) methods for common compound operations, such as `put-if-absent`, are provided. The iterators returned by `ConcurrentHashMap` are weakly consistent, meaning that they will not throw `ConcurrentModificationException` and will make "reasonable efforts" to reflect modifications to the `Map` that are made by other threads during iteration.

Queue

The original collections framework included three interfaces -- `List`, `Map`, and `Set`. `List` described an ordered collection of elements, supporting full random access -- an element could be added, fetched, or removed from any position.

The `LinkedList` class is often used to store a list, or queue, of work elements -- tasks waiting to be executed. However, the `List` interface offers far more flexibility than is needed for this common application, which in general only inserts elements at the tail and removes elements from the head. But the requirement to support the full `List` interface means that `LinkedList` is not as efficient for this task as it might otherwise be. The `Queue` interface is much simpler than `List` -- it includes only `put()` and `take()` methods, and enables more efficient implementations than `LinkedList`.

The `Queue` interface also allows the implementation to determine the order in which elements are stored. The `ConcurrentLinkedQueue` class implements a first-in-first-out (FIFO) queue, whereas the `PriorityQueue` class implements a priority queue (also called a heap), which is useful for building schedulers that must execute tasks in order of priority or desired execution time.

```
interface Queue<E> extends Collection<E> {
    boolean offer(E x);
    E poll();
    E remove() throws NoSuchElementException;
    E peek();
    E element() throws NoSuchElementException;
}
```

The classes that implement `Queue` are:

- **`LinkedList`** Has been retrofitted to implement `Queue`
- **`PriorityQueue`** A non-thread-safe priority queue (heap) implementation, returning elements according to natural order or a comparator
- **`ConcurrentLinkedQueue`** A fast, thread-safe, non-blocking FIFO queue

BlockingQueue

`Queues` can be bounded or unbounded. Attempting to modify a bounded queue will fail when you attempt to add an element to an already full queue, or when you attempt to remove an element from an empty queue.

Sometimes, it is more desirable to cause a thread to block when a queue operation would otherwise fail. In addition to not requiring the calling class to

deal with failure and retry, blocking also has the benefit of flow control -- when a consumer is removing elements from a queue more slowly than the producers are putting them on the queue, forcing the producers to block will throttle the producers. Contrast this with an unbounded, nonblocking queue -- if the imbalance between producer and consumer is long-lived, the system may run out of memory as the queue length grows without bound. A bounded, blocking queue allows you to limit the resources used by a given queue in a graceful and natural way.

The classes that implement `BlockingQueue` are:

- **`LinkedBlockingQueue`** A bounded or unbounded FIFO blocking queue implemented like a linked list
- **`PriorityBlockingQueue`** An unbounded blocking priority queue
- **`ArrayBlockingQueue`** A bounded FIFO blocking queue backed by an array
- **`SynchronousQueue`** Not really a queue at all, but facilitates synchronous handoff between cooperating threads

Section 4. Task management

Thread creation

One of the most common applications for threads is to create one or more threads for the purpose of executing specific types of tasks. The `Timer` class creates a thread for executing `TimerTask` objects, and `Swing` creates a thread for processing UI events. In both of these cases, the tasks that are executing in the separate thread are supposed to be short-lived -- these threads exist to service a potentially large number of short-lived tasks.

In each of these cases, these threads generally have a very simple structure:

```
while (true) {
    if (no tasks)
        wait for a task;
    execute the task;
}
```

Threads are created by instantiating an object that derives from `Thread` and calling the `Thread.start()` method. You can create a thread in two ways -- by extending `Thread` and overriding the `run()` method, or by implementing the `Runnable` interface and using the `Thread(Runnable)` constructor:

```
class WorkerThread extends Thread {
    public void run() { /* do work */ }
}
Thread t = new WorkerThread();
t.start();
```

or:

```
Thread t = new Thread(new Runnable() {
    public void run() { /* do work */ }
});
t.start();
```

Reusing threads

Frameworks like the `Swing` GUI framework create a single thread for event tasks instead of spawning a new thread for each task for several reasons. The first is that there is some overhead to creating threads, so creating a thread to

execute a simple task would be a waste of resources. By reusing the event thread to process multiple events, the startup and teardown cost (which varies by platform) are amortized over many events.

Another reason that Swing uses a single background thread for events is to ensure that events will not interfere with each other, because the next event will not start being processed until the previous event is finished. This approach simplifies the writing of event handlers. With multiple threads, it would take more work to ensure that only one thread is executing thread-sensitive code at a time.

How not to manage tasks

Most server applications, such as Web servers, POP servers, database servers, or file servers, process requests on behalf of remote clients, which usually use a socket to connect to the server. For each request, there is generally a small amount of processing (go get this block of this file and send it back down the socket), but a potentially large (and unbounded) number of clients requesting service.

A simplistic model for building a server application would be to spawn a new thread for every request. The following code fragment implements a simple Web server, which accepts socket connections on port 80 and spawns a new thread to handle the request. Unfortunately, this code would not be a good way to implement a Web server, as it will fail under heavy load, taking down the entire server.

```
class UnreliableWebServer {
    public static void main(String[] args) {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable r = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            // Don't do this!
            new Thread(r).start();
        }
    }
}
```

The `UnreliableWebServer` class deals poorly with the situation where the server is overwhelmed by requests. Every time a request comes in, a new thread is created. Depending on your operating system and available memory,

the number of threads you can create is limited. Unfortunately, you don't always know what that limit is -- you only find out when your application crashes with an `OutOfMemoryError`.

If you throw HTTP requests at this server fast enough, eventually one of the thread creations will fail, with the resulting `Error` taking down the entire application. And there's no reason to create a thousand threads when you can only service a few dozen of them effectively at a time -- such a use of resources will likely hurt performance anyway. Creating a thread uses a fair bit of memory -- there are two stacks (Java and C), plus per-thread data structures. And if you create too many threads, each of them will get very little CPU time anyway, with the result being that you are using a lot of memory to service a large number of threads, each of which are running very slowly. This isn't a good use of computing resources.

Thread pools to the rescue

Creating a new thread for a task is not necessarily bad, but if the frequency of task creation is high and the mean task duration is low, we can see how spawning a new thread per task will create performance (and, if the load is unpredictable, stability) problems.

If it is not to create a new thread per task, a server application must have some means of limiting how many requests are being processed at one time. This means that it cannot simply call

```
new Thread(runnable).start()
```

every time it needs to start a new task.

The classic mechanism for managing a large group of small tasks is to combine a *work queue* with a *thread pool*. A work queue is simply a queue of tasks to be processed, and the `Queue` classes described earlier fit the bill exactly. A thread pool is a collection of threads that each feed off of the common work queue. When one of the worker threads completes the processing of a task, it goes back to the queue to see if there are more tasks to process. If there are, it dequeues the next task and starts processing it.

A thread pool offers a solution to both the problem of thread life-cycle overhead and the problem of resource thrashing. By reusing threads for multiple tasks, the thread-creation overhead is spread over many tasks. As a bonus, because the thread already exists when a request arrives, the delay introduced by thread creation is eliminated. Thus, the request can be serviced immediately, rendering the application more responsive. Furthermore, by properly tuning the number of

threads in the thread pool, you can prevent resource thrashing by forcing any requests in excess of a certain threshold to wait until a thread is available to process it, where they will consume less resources while waiting than an additional thread would.

The Executor framework

The `java.util.concurrent` package contains a flexible thread pool implementation, but even more valuable, it contains an entire framework for managing the execution of tasks that implement `Runnable`. This framework is called the Executor framework.

The `Executor` interface is quite simple. It describes an object whose job it is to run `Runnable`s:

```
public interface Executor {
    void execute(Runnable command);
}
```

Which thread the task runs in is not specified by the interface -- that depends on which implementation of `Executor` you are using. It could run in a background thread, like the Swing event thread, or in a pool of threads, or in the calling thread, or a new thread, or even in another JVM! By submitting the task through the standardized `Executor` interface, the task submission is decoupled from the task execution policy. The `Executor` interface concerns itself solely with task submission -- it is the choice of `Executor` implementation that determines the *execution policy*. This makes it much easier to tune the execution policy (queue bounds, pool size, prioritization, and so on) at deployment time, with minimal code changes.

Most of the `Executor` implementations in `java.util.concurrent` also implement the `ExecutorService` interface, an extension of `Executor` that also manages the lifecycle of the execution service. This makes it easier for them to be managed and to provide services to an application whose lifetime may be longer than that of an individual `Executor`.

```
public interface ExecutorService extends Executor {
    void shutdown();
    List<Runnable> shutdownNow();
    boolean isShutdown();
    boolean isTerminated();
    boolean awaitTermination(long timeout,
                             TimeUnit unit);
    // other convenience methods for submitting tasks
}
```

Executors

The `java.util.concurrent` package contains several implementations of `Executor`, each of which implement different execution policies. What is an execution policy? An execution policy defines when and in what thread a task will run, what level of resources (threads, memory, and so on) the execution service may consume, and what to do if the executor is overloaded.

Rather than being instantiated through constructors, executors are generally instantiated through factory methods. The `Executors` class contains static factory methods for constructing a number of different kinds of `Executor` implementations:

- **`Executors.newCachedThreadPool()`** Creates a thread pool that is not limited in size, but which will reuse previously created threads when they are available. If no existing thread is available, a new thread will be created and added to the pool. Threads that have not been used for 60 seconds are terminated and removed from the cache.
- **`Executors.newFixedThreadPool(int n)`** Creates a thread pool that reuses a fixed set of threads operating off a shared unbounded queue. If any thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks.
- **`Executors.newSingleThreadExecutor()`** Creates an `Executor` that uses a single worker thread operating off an unbounded queue, much like the Swing event thread. Tasks are guaranteed to execute sequentially, and no more than one task will be active at any given time.

A more reliable Web server -- using Executor

The code in [How not to manage tasks](#) on page 16 earlier showed how *not* to write a reliable server application. Fortunately, fixing this example is quite easy -- replace the `Thread.start()` call with submitting a task to an `Executor`:

```
class ReliableWebServer {
    Executor pool =
        Executors.newFixedThreadPool(7);
    public static void main(String[] args) {
```

```
ServerSocket socket = new ServerSocket(80);
while (true) {
    final Socket connection = socket.accept();
    Runnable r = new Runnable() {
        public void run() {
            handleRequest(connection);
        }
    };
    pool.execute(r);
}
}
```

Note that the only difference between this example and the previous example is the creation of the `Executor` and how tasks are submitted for execution.

Customizing ThreadPoolExecutor

The `Executors` returned by the `newFixedThreadPool` and `newCachedThreadPool` factory methods in `Executors` are instances of the class `ThreadPoolExecutor`, which is highly customizable.

The creation of a pool thread can be customized by using a version of the factory method or constructor that takes a `ThreadFactory` argument. A `ThreadFactory` is a factory object that constructs new threads to be used by an executor. Using a customized thread factory gives you the opportunity to create threads that have a useful thread name, are daemon threads, belong to a specific thread group, or have a specific priority.

The following is an example of a thread factory that creates daemon threads instead of user threads:

```
public class DaemonThreadFactory implements ThreadFactory {
    public Thread newThread(Runnable r) {
        Thread thread = new Thread(r);
        thread.setDaemon(true);
        return thread;
    }
}
```

Sometimes an `Executor` cannot execute a task, either because it has been shut down, or because the `Executor` uses a bounded queue for storing waiting tasks, and the queue is full. In that case, the executor's `RejectedExecutionHandler` is consulted to determine what to do with the task -- throw an exception (the default), discard the task, execute the task in the caller's thread, or discard the oldest task in the queue to make room for the new task. The rejected execution handler can be set by

```
ThreadPoolExecutor.setRejectedExecutionHandler.
```

You can also extend `ThreadPoolExecutor`, and override the methods `beforeExecute` and `afterExecute`, to add instrumentation, add logging, add timing, reinitialize thread-local variables, or make other execution customizations.

Special considerations

Using the `Executor` framework decouples task submission from execution policy, which in the general case is more desirable as it allows us to flexibly tune the execution policy without having to change the code in hundreds of places. However, several situations exist when the submission code implicitly assumes a certain execution policy, in which case it is important that the selected `Executor` implement a consistent execution policy.

One such case is when tasks wait synchronously for other tasks to complete. In that case, if the thread pool does not contain enough threads, it is possible for the pool to deadlock, if all currently executing tasks are waiting for another task, and that task cannot execute because the pool is full.

A similar case is when a group of threads must work together as a cooperating group. In that case, you will want to ensure that the thread pool is large enough to accommodate all the threads.

If your application makes certain assumptions about a specific executor, these should be documented near the definition and initialization of the `Executor` so that well-intentioned changes do not subvert the correct functioning of your application.

Tuning thread pools

A common question asked when creating `Executors` is "How big should the thread pool be?" The answer, of course, depends on your hardware (how many processors do you have?) and the type of tasks that are going to be executed (are they compute-bound or IO-bound?).

If thread pools are too small, the result may be incomplete resource utilization -- there may be idle processors while tasks are still on the work queue waiting to execute.

On the other hand, if the thread pool is too large, then there will be many active threads, and performance may suffer due to the memory utilization of the large number of threads and active tasks, or because there will be more context switches per task than with a smaller number of threads.

So what's the right size for a thread pool, assuming the goal is to keep the processors fully utilized? Amdahl's law gives us a good approximate formula, if we know how many processors our system has and the approximate ratio of compute time to wait time for the tasks.

Let `WT` represent the average wait time per task, and `ST` the average service time (computation time) per task. Then `WT/ST` is the percentage of time a task spends waiting. For an `N` processor system, we would want to have approximately $N \cdot (1 + WT/ST)$ threads in the pool.

The good news is that you don't have to estimate `WT/ST` exactly. The range of "good" pool sizes is fairly large; you just want to avoid the extremes of "much too big" and "much too small."

The Future interface

The `Future` interface allows you to represent a task that may have completed, may be in the process of being executed, or may not yet have started execution. Through the `Future` interface, you can attempt to cancel a task that has not yet completed, inquire whether the task has completed or cancelled, and fetch (or wait for) the task's result value.

The `FutureTask` class implements `Future`, and has constructors that allow you to wrap a `Runnable` or `Callable` (a result-bearing `Runnable`) with a `Future` interface. Because `FutureTask` also implements `Runnable`, you can then simply submit `FutureTask` to an `Executor`. Some submission methods (like `ExecutorService.submit()`) will return a `Future` interface in addition to submitting the task.

The `Future.get()` method retrieves the result of the task computation (or throws `ExecutionException` if the task completed with an exception). If the task has not yet completed, `Future.get()` will block until the task completes; if it has already completed, the result will be returned immediately.

Building a cache with Future

This code example ties together several classes from `java.util.concurrent`, prominently showcasing the power of `Future`. It implements a cache, and uses `Future` to describe a cached value that may already be computed or that may be "under construction" in another thread.

It takes advantage of the atomic `putIfAbsent()` method in `ConcurrentHashMap`, ensuring that only one thread will try to compute the value for a given key. If another thread subsequently requests the value for that same key, it simply waits (with the help of `Future.get()`) for the first thread to complete. As a result, two threads will not try to compute the same value.

```
public class Cache<K, V> {
    ConcurrentMap<K, FutureTask<V>> map = new ConcurrentHashMap();
    Executor executor = Executors.newFixedThreadPool(8);

    public V get(final K key) {
        FutureTask<V> f = map.get(key);
        if (f == null) {
            Callable<V> c = new Callable<V>() {
                public V call() {
                    // return value associated with key
                }
            };
            f = new FutureTask<V>(c);
            FutureTask old = map.putIfAbsent(key, f);
            if (old == null)
                executor.execute(f);
            else
                f = old;
        }
        return f.get();
    }
}
```

CompletionService

`CompletionService` combines an execution service with a `Queue`-like interface, so that the processing of task results can be decoupled from task execution. The `CompletionService` interface includes `submit()` methods for submitting tasks for execution, and `take()/poll()` methods for asking for the next completed task.

`CompletionService` allows the application to be structured using the Producer/Consumer pattern, where producers create tasks and submit them, and consumers request the results of a complete task and then do something with those results. The `CompletionService` interface is implemented by the `ExecutorCompletionService` class, which uses an `Executor` to process

the tasks and exports the submit/poll/take methods from `CompletionService`.

The following example uses an `Executor` and a `CompletionService` to start a number of "solver" tasks, and uses the result of the first one that produces a non-null result, and cancels the rest:

```
void solve(Executor e, Collection<Callable<Result>> solvers)
    throws InterruptedException {
    CompletionService<Result> ecs = new ExecutorCompletionService<Result>(e);
    int n = solvers.size();
    List<Future<Result>> futures = new ArrayList<Future<Result>>(n);
    Result result = null;
    try {
        for (Callable<Result> s : solvers)
            futures.add(ecs.submit(s));
        for (int i = 0; i < n; ++i) {
            try {
                Result r = ecs.take().get();
                if (r != null) {
                    result = r;
                    break;
                }
            } catch (ExecutionException ignore) {}
        }
    } finally {
        for (Future<Result> f : futures)
            f.cancel(true);
    }

    if (result != null)
        use(result);
}
```

Section 5. Synchronizer classes

Synchronizers

Another useful category of classes in `java.util.concurrent` is the synchronizers. This set of classes coordinates and controls the flow of execution for one or more threads.

The `Semaphore`, `CyclicBarrier`, `CountdownLatch`, and `Exchanger` classes are all examples of synchronizers. Each of these has methods that threads can call that may or may not block based on the state and rules of the particular synchronizer being used.

Semaphores

The `Semaphore` class implements a classic Dijkstra counting semaphore. A counting semaphore can be thought of as having a certain number of permits, which can be acquired and released. If there are permits left, the `acquire()` method will succeed, otherwise it will block until one becomes available (by another thread releasing the permit). A thread can acquire more than one permit at a time.

Counting semaphores can be used to restrict the number of threads that have concurrent access to a resource. This approach is useful for implementing resource pools or limiting the number of outgoing socket connections in a Web crawler.

Note that the semaphore does not keep track of which threads own how many permits; it is up to the application to ensure that when a thread releases a permit, that it either owns the permit or it is releasing it on behalf of another thread, and that the other thread realizes that its permit has been released.

Mutex

A special case of counting semaphores is the *mutex*, or mutual-exclusion semaphore. A mutex is simply a counting semaphore with a single permit, meaning that only one thread can hold a permit at a given time (also called a *binary semaphore*). A mutex can be used to manage exclusive access to a shared resource.

While mutexes have a lot in common with locks, mutexes have one additional feature that locks generally do not have, and that is the ability for the mutex to be released by a different thread than the one holding the permit. This may be useful in deadlock recovery situations.

CyclicBarrier

The `CyclicBarrier` class is a synchronization aid that allows a set of threads to wait for the entire set of threads to reach a common barrier point.

`CyclicBarrier` is constructed with an integer argument, which determines the number of threads in the group. When one thread arrives at the barrier (by calling `CyclicBarrier.await()`), it blocks until all threads have arrived at the barrier, at which point all the threads are then allowed to continue executing. This action is similar to what many families (try to) do at the mall -- family members go their separate ways, and everyone agrees to meet at the movie theater at 1:00. When you get to the movie theater and not everyone is there, you sit and wait for everyone else to arrive. Then everyone can leave together.

The barrier is called cyclic because it is reusable; once all the threads have met up at the barrier and been released, the barrier is reinitialized to its initial state.

You can also specify a timeout when waiting at the barrier; if by that time the rest of the threads have not arrived at the barrier, the barrier is considered broken and all threads that are waiting receive a `BrokenBarrierException`.

The code example below creates a `CyclicBarrier` and launches a set of threads that will each compute a portion of a problem, wait for all the other threads to finish, and then check to see if the solution has converged. If not, each worker thread will begin another iteration. This example uses a variant of `CyclicBarrier` that lets you register a `Runnable` that is executed whenever all the threads arrive at the barrier but before any of them are released.

```
class Solver { // Code sketch
    void solve(final Problem p, int nThreads) {
        final CyclicBarrier barrier =
            new CyclicBarrier(nThreads,
                new Runnable() {
                    public void run() { p.checkConvergence(); }}
            );
        for (int i = 0; i < nThreads; ++i) {
            final int id = i;
            Runnable worker = new Runnable() {
                final Segment segment = p.createSegment(id);
                public void run() {
                    try {
                        while (!p.converged()) {
```

```
        segment.update();
        barrier.await();
    }
}
catch(Exception e) { return; }
}
};
new Thread(worker).start();
}
}
```

CountdownLatch

The `CountdownLatch` class is similar to `CyclicBarrier`, in that its role is to coordinate a group of threads that have divided a problem among themselves. It is also constructed with an integer argument, indicating the initial value of the count, but, unlike `CyclicBarrier`, is not reusable.

Where `CyclicBarrier` acts as a gate to all the threads that reach the barrier, allowing them through only when all the threads have arrived at the barrier or the barrier is broken, `CountdownLatch` separates the arrival and waiting functionality. Any thread can decrement the current count by calling `countDown()`, which does not block, but merely decrements the count. The `await()` method behaves slightly differently than `CyclicBarrier.await()` -- any threads that call `await()` will block until the latch count gets down to zero, at which point all threads waiting will be released, and subsequent calls to `await()` will return immediately.

`CountdownLatch` is useful when a problem has been decomposed into a number of pieces, and each thread has been given a piece of the computation. When the worker threads finish, they decrement the count, and the coordination thread(s) can wait on the latch for the current batch of computations to finish before moving on to the next batch.

Conversely, a `CountdownLatch` class with a count of 1 can be used as a "starting gate" to start a group of threads at once; the worker threads wait on the latch, and the coordinating thread decrements the count, which releases all the worker threads at once. The following example uses two `CountdownLatches`; ones as a starting gate, and one that releases when all the worker threads are finished:

```
class Driver { // ...
    void main() throws InterruptedException {
        CountdownLatch startSignal = new CountdownLatch(1);
        CountdownLatch doneSignal = new CountdownLatch(N);
```

```
    for (int i = 0; i < N; ++i) // create and start threads
        new Thread(new Worker(startSignal, doneSignal)).start();

    doSomethingElse();           // don't let them run yet
    startSignal.countDown();     // let all threads proceed
    doSomethingElse();
    doneSignal.await();         // wait for all to finish
}
}

class Worker implements Runnable {
    private final CountdownLatch startSignal;
    private final CountdownLatch doneSignal;
    Worker(CountdownLatch startSignal, CountdownLatch doneSignal) {
        this.startSignal = startSignal;
        this.doneSignal = doneSignal;
    }
    public void run() {
        try {
            startSignal.await();
            doWork();
            doneSignal.countDown();
        } catch (InterruptedException ex) {} // return;
    }
}
```

Exchanger

The `Exchanger` class facilitates a two-way exchange between two cooperating threads; in this way, it is like a `CyclicBarrier` with a count of two, with the added feature that the two threads can "trade" some state when they both reach the barrier. (The `Exchanger` pattern is also sometimes called a rendezvous.)

A typical use for `Exchanger` would be where one thread is filling a buffer (by reading from a socket) and the other thread is emptying the buffer (by processing the commands received from the socket). When the two threads meet at the barrier, they swap buffers. The following code demonstrates this technique:

```
class FillAndEmpty {
    Exchanger<DataBuffer> exchanger = new Exchanger<DataBuffer>();
    DataBuffer initialEmptyBuffer = new DataBuffer();
    DataBuffer initialFullBuffer = new DataBuffer();

    class FillingLoop implements Runnable {
        public void run() {
            DataBuffer currentBuffer = initialEmptyBuffer;
            try {
                while (currentBuffer != null) {
                    addToBuffer(currentBuffer);
                }
            }
        }
    }
}
```

```
        if (currentBuffer.full())
            currentBuffer = exchanger.exchange(currentBuffer);
    }
} catch (InterruptedException ex) { ... handle ... }
}
}

class EmptyingLoop implements Runnable {
    public void run() {
        DataBuffer currentBuffer = initialFullBuffer;
        try {
            while (currentBuffer != null) {
                takeFromBuffer(currentBuffer);
                if (currentBuffer.empty())
                    currentBuffer = exchanger.exchange(currentBuffer);
            }
        } catch (InterruptedException ex) { ... handle ...}
    }
}

void start() {
    new Thread(new FillingLoop()).start();
    new Thread(new EmptyingLoop()).start();
}
}
```

Section 6. Low-level facilities -- Lock and Atomic

Lock

The Java language has a built-in locking facility -- the `synchronized` keyword. When a thread acquires a monitor (built-in lock), other threads will block when trying to acquire the same lock, until the first thread releases it. Synchronization also ensures that the values of any variables modified by a thread while it holds a lock are visible to a thread that subsequently acquires the same lock, ensuring that if classes properly synchronize access to shared state, threads will not see "stale" values of variables that are the result of caching or compiler optimization.

While there is nothing wrong with synchronization, it has some limitations that can prove inconvenient in some advanced applications. The `Lock` interface is a generalization of the locking behavior of built-in monitor locks, which allow for multiple lock implementations, while providing some features that are missing from built-in locks, such as timed waits, interruptible waits, lock polling, multiple condition-wait sets per lock, and non-block-structured locking.

```
interface Lock {
    void lock();
    void lockInterruptibly() throws IE;
    boolean tryLock();
    boolean tryLock(long time,
                    TimeUnit unit) throws IE;
    void unlock();
    Condition newCondition() throws
        UnsupportedOperationException;
}
```

ReentrantLock

`ReentrantLock` is an implementation of `Lock` with the same basic behavior and semantics as the implicit monitor lock accessed using `synchronized` methods and statements, but with extended capabilities.

As a bonus, the implementation of `ReentrantLock` is far more scalable under contention than the current implementation of `synchronized`. (It is likely that there will be improvements to the contended performance of `synchronized` in a future version of the JVM.) This means that when many threads are all contending for the same lock, the total throughput is generally going to be better with `ReentrantLock` than with `synchronized`. In other words, when many

threads are attempting to access a shared resource protected by a `ReentrantLock`, the JVM will spend less time scheduling threads and more time executing them.

While it has many advantages, the `ReentrantLock` class has one major disadvantage compared to synchronization -- it is possible to forget to release the lock. It is recommended that the following structure be used when acquiring and releasing a `ReentrantLock`:

```
Lock lock = new ReentrantLock();
...
lock.lock();
try {
    // perform operations protected by lock
}
catch(Exception ex) {
    // restore invariants
}
finally {
    lock.unlock();
}
```

Because the risk of fumbling a lock (forgetting to release it) is so severe, it is recommended that you continue to use `synchronized` for basic locking unless you really need the additional flexibility or scalability of `ReentrantLock`. `ReentrantLock` is an advanced tool for advanced applications -- sometimes you need it, but sometimes the trusty old hammer does just fine.

Conditions

Just as the `Lock` interface is a generalization of synchronization, the `Condition` interface is a generalization of the `wait()` and `notify()` methods in `Object`. One of the methods in `Lock` is `newCondition()` -- this asks the lock to return a new `Condition` object bound to this lock. The `await()`, `signal()`, and `signalAll()` methods are analogous to `wait()`, `notify()`, and `notifyAll()`, with the added flexibility that you can create more than one condition variable per `Lock`. This simplifies the implementation of some concurrent algorithms.

ReadWriteLock

The locking discipline implemented by `ReentrantLock` is quite simple -- one thread at a time holds the lock, and other threads must wait for it to be

available. Sometimes, when data structures are more commonly read than modified, it may be desirable to use a more complicated lock structure, called a read-write lock, which allows multiple concurrent readers but also allows for exclusive locking by a writer. This approach offers greater concurrency in the common case (read only) while still offering the safety of exclusive access when necessary. The `ReadWriteLock` interface and the `ReentrantReadWriteLock` class provide this capability -- a multiple-reader, single-writer locking discipline that can be used to protect shared mutable resources.

Atomic variables

Even though they will rarely be used directly by most users, the most significant new concurrent classes may well be the atomic variable classes (`AtomicInteger`, `AtomicLong`, `AtomicReference`, and so on). These classes expose the low-level improvements to the JVM that enable highly scalable atomic read-modify-write operations. Most modern CPUs have primitives for atomic read-modify-write, such as compare-and-swap (CAS) or load-linked/store-conditional (LL/SC). The atomic variable classes are implemented with whatever is the fastest concurrency construct provided by the hardware.

Many concurrent algorithms are defined in terms of compare-and-swap operations on counters or data structures. By exposing a high-performance, highly scalable CAS operation (in the form of atomic variables), it becomes practical to implement high performance, wait-free, lock-free concurrent algorithms in the Java language.

Nearly all of the classes in `java.util.concurrent` are built on top of `ReentrantLock`, which itself is built on top of the atomic variable classes. So while they may only be used by a few concurrency experts, it is the atomic variable classes that provide much of the scalability improvement of the `java.util.concurrent` classes.

The primary use for atomic variables is to provide an efficient, fine-grained means of atomically updating "hot" fields -- fields that are frequently accessed and updated by multiple threads. In addition, they are a natural mechanism for counters or generating sequence numbers.

Section 7. Performance and scalability

Performance vs. scalability

While the overriding goal of the `java.util.concurrent` effort was to make it easier to write correct, thread-safe classes, a secondary goal was to improve scalability. Scalability is not exactly the same thing as performance -- in fact, sometimes scalability comes at the cost of performance.

Performance is a measure of "how fast can you execute this task." Scalability describes how an application's throughput behaves as its workload and available computing resources increase. A scalable program can handle a proportionally larger workload with more processors, memory, or I/O bandwidth. When we talk about scalability in the context of concurrency, we are asking how well a given class performs when many threads are accessing it simultaneously.

The low-level classes in `java.util.concurrent` -- `ReentrantLock` and the atomic variable classes -- are far more scalable than the built-in monitor (synchronization) locks. As a result, classes that use `ReentrantLock` or atomic variables for coordinating shared access to state will likely be more scalable as well.

Hashtable vs. ConcurrentHashMap

As an example of scalability, the `ConcurrentHashMap` implementation is designed to be far more scalable than its thread-safe uncle, `Hashtable`. `Hashtable` only allows a single thread to access the `Map` at a time; `ConcurrentHashMap` allows for multiple readers to execute concurrently, readers to execute concurrently with writers, and some writers to execute concurrently. As a result, if many threads are accessing a shared map frequently, overall throughput will be better with `ConcurrentHashMap` than with `Hashtable`.

The table below gives a rough idea of the scalability differences between `Hashtable` and `ConcurrentHashMap`. In each run, `N` threads concurrently executed a tight loop where they retrieved random key values from either a `Hashtable` or a `ConcurrentHashMap`, with 60 percent of the failed retrievals performing a `put()` operation and 2 percent of the successful retrievals performing a `remove()` operation. Tests were performed on a dual-processor Xeon system running Linux. The data shows run time for 10,000,000 iterations, normalized to the 1-thread case for `ConcurrentHashMap`. You can see that the performance of `ConcurrentHashMap` remains scalable up to many

threads, whereas the performance of `Hashtable` degrades almost immediately in the presence of lock contention.

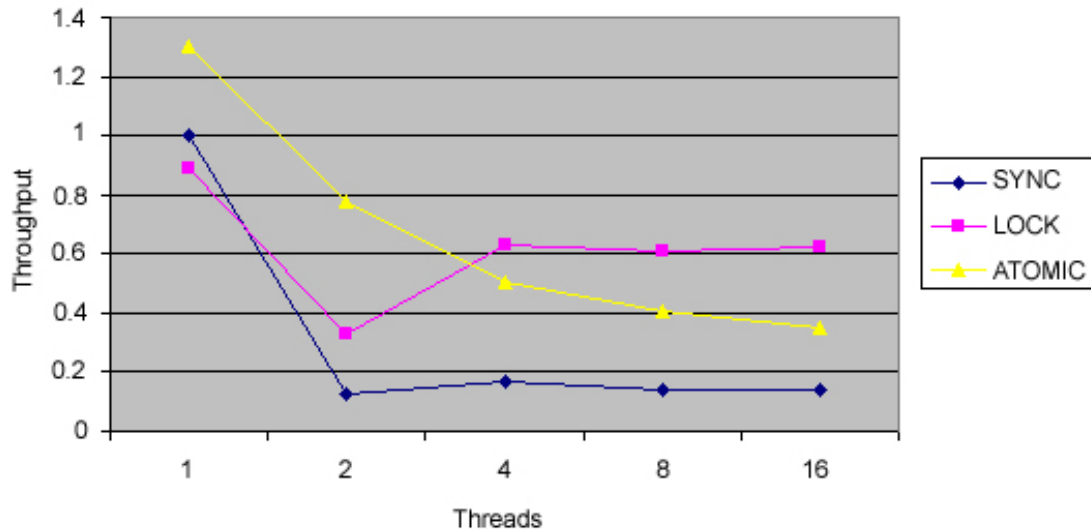
The number of threads in this test may look small compared to typical server applications. However, because each thread is doing nothing but repeatedly hitting on the table, this simulates the contention of a much larger number of threads using the table in the context of doing some amount of real work.

| Threads | <code>ConcurrentHashMap</code> | <code>Hashtable</code> |
|---------|--------------------------------|------------------------|
| 1 | 1.0 | 1.51 |
| 2 | 1.44 | 17.09 |
| 4 | 1.83 | 29.9 |
| 8 | 4.06 | 54.06 |
| 16 | 7.5 | 119.44 |
| 32 | 15.32 | 237.2 |

Lock vs. synchronized vs. Atomic

Another example of the scalability improvements possible with `java.util.concurrent` is evidenced by the following benchmark. This benchmark simulates rolling a die, using a linear congruence random number generator. Three implementations of the random number generator are available: one that uses synchronization to manage the state of the generator (a single variable), one that uses `ReentrantLock`, and one that uses `AtomicLong`. The graph below shows the relative throughput of the three versions with increasing numbers of threads, on an 8-way Ultrasparc3 system. (The graph probably understates the scalability of the atomic variable approach.)

Figure 1. Relative throughput using synchronization, Lock, and AtomicLong



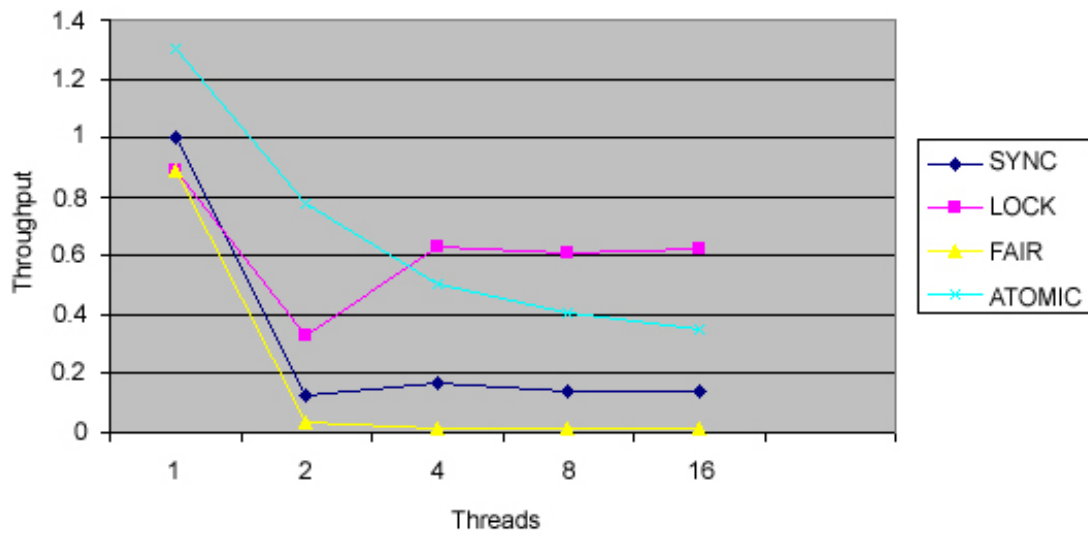
Fair vs. unfair

One additional element of customization in many of the classes in `java.util.concurrent` is the question of "fairness." A fair lock, or fair semaphore, is one where threads are granted the lock or semaphore on a first-in, first-out (FIFO) basis. The constructors for `ReentrantLock`, `Semaphore`, and `ReentrantReadWriteLock` all can take arguments that determine whether the lock is fair, or whether it permits *barging* (threads to acquire the lock even if they have not been waiting the longest).

While the idea of barging locks may seem ridiculous and, well, unfair, barging locks are in fact quite common, and usually preferable. The built-in locks accessed with synchronization are not fair locks (and there is no way to make them fair). Instead, they provide weaker liveness guarantees that require that all threads will eventually acquire the lock.

The reason most applications choose (and should choose) barging locks over fair locks is performance. In most cases, exact fairness is not a requirement for program correctness, and the cost of fairness is quite high indeed. The table below adds a fourth dataset to the table from the previous panel, where access to the PRNG state is managed by a fair lock. Note the large difference in throughput between barging locks and fair locks.

Figure 2. Relative throughput using synchronization, Lock, fair Lock, and AtomicLong



Section 8. Wrap-up and resources

Summary

The `java.util.concurrent` package contains a wealth of useful building blocks for improving the performance, scalability, thread-safety, and maintainability of concurrent classes. With them, you should be able to eliminate most uses of synchronization, `wait/notify`, and `Thread.start()` in your code, replacing them with higher-level, standardized, high-performance concurrency utilities.

Resources

- Review the basics of multithreading in the Java language by taking the "[Introduction to Java Threads](#)" tutorial, also by Brian Goetz
- For questions related to concurrency, visit the [Multithreaded Java programming discussion forum](#), where you can get an answer or share your knowledge.
- Many of the concepts from `java.util.concurrent` came from Doug Lea's [util.concurrent](#) package.
- Doug Lea's [Concurrent Programming in Java, Second Edition](#) (Addison-Wesley, 1999) is a masterful book on the subtle issues surrounding multithreaded programming in the Java language.
- The `java.util.concurrent` package was formalized under Java Community Process [JSR 166](#) (<http://www.jcp.org/jsr/detail/166.jsp>).
- The [Java theory and practice](#) column, also by Brian Goetz, frequently covers the topic of multithreading and concurrency. the following installments may be of particular interest:
 - "[Thread pools and work queues](#)" (July 2002)
 - "[Building a better HashMap](#)" (August 2003)
 - "[Characterizing thread safety](#)" (September 2003)
 - "[More flexible, scalable locking in JDK 5.0](#)" (October 2004)
 - "[Going Atomic](#)" (November 2004)

Your feedback

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.)

For more information about the Toot-O-Matic, visit www-106.ibm.com/developerworks/xml/library/x-toot/ .