The next generation: Java concurrency without the pain, Part 1

With the increasingly complexity of concurrent applications, many developers find that Java's low-level threading capabilities are insufficient to their programming needs. In that case, it might be time to discover the Java Concurrency Utilities. Get started with java.util.concurrent, with Jeff Friesen's detailed introduction to the Executor framework, synchronizer types, and the Java Concurrent Collections package.

The Java platform provides low-level threading capabilities that enable developers to write concurrent applications where different threads execute simultaneously. Standard Java threading has some downsides, however:

- Java's low-level concurrency primitives (synchronized, volatile, wait(), notify(), and notifyAll()) aren't easy to use correctly. Threading hazards like deadlock, thread starvation, and race conditions, which result from incorrect use of primitives, are also hard to detect and debug.
- Relying on synchronized to coordinate access between threads leads to performance issues that affect application scalability, a requirement for many modern applications.
- Java's basic threading capabilities are *too* low level. Developers often need higher level constructs like semaphores and thread pools, which Java's low-level threading capabilities don't offer. As a result, developers will build their own constructs, which is both time consuming and error prone.

The JSR 166: Concurrency Utilities framework was designed to meet the need for a high-level threading facility. Initiated in early 2002, the framework was formalized and implemented two years later in Java 5. Enhancements have followed in Java 6, Java 7, and the forthcoming Java 8.

This two-part *Java 101: The next generation* series introduces software developers familiar with basic Java threading to the Java Concurrency Utilities packages and framework. In Part 1, I present an overview of the Java Concurrency Utilities framework and introduce its Executor framework, synchronizer utilities, and the Java

Concurrent Collections package.

Inside the Java Concurrency Utilities

The Java Concurrency Utilities framework is a library of *types* that are designed to be used as building blocks for creating concurrent classes or applications. These types are thread-safe, have been thoroughly tested, and offer high performance.

Types in the Java Concurrency Utilities are organized into small frameworks; namely, Executor framework, synchronizer, concurrent collections, locks, atomic variables, and Fork/Join. They are further organized into a main package and a pair of subpackages:

- **java.util.concurrent** contains high-level utility types that are commonly used in concurrent programming. Examples include semaphores, barriers, thread pools, and concurrent hashmaps.
    1. The **java.util.concurrent.atomic** subpackage contains low-level utility classes that support lock-free thread-safe programming on single variables.
    2. The **java.util.concurrent.locks** subpackage contains low-level utility types for locking and waiting for conditions, which are different from using Java's low-level synchronization and monitors.

The Java Concurrency Utilities framework also exposes the low-level *compare-and-swap (CAS)* hardware instruction, variants of which are commonly supported by modern processors. CAS is much more lightweight than Java's monitor-based synchronization mechanism and is used to implement some highly scalable concurrent classes. The CAS-based java.util.concurrent.locks.ReentrantLockclass, for instance, is more performant than the equivalent monitor-basedsynchronized primitive. ReentrantLock offers more control over locking. (In Part 2 I'll explain more about how CAS works in java.util.concurrent.)

System.nanoTime()

The Java Concurrency Utilities framework includes long nanoTime(), which is a member of the java.lang.System class. This method enables access to a nanosecond-granularity

time source for making relative time measurements.

In the next sections I'll introduce three useful features of the Java Concurrency Utilities, first explaining why they're so important to modern concurrency and then demonstrating how they work to increase the speed, reliability, efficiency, and scalability of concurrent Java applications.

The Executor framework

In threading, a *task* is a unit of work. One problem with low-level threading in Java is that task submission is tightly coupled with a task-execution policy, as demonstrated by Listing 1.

Listing 1. Server.java (Version 1)

```
import java.io.IOException;

import java.net.ServerSocket;
import java.net.Socket;

class Server
{
  public static void main(String[] args) throws IOException
  {
    ServerSocket socket = new ServerSocket(9000);
    while (true)
    {
      final Socket s = socket.accept();
      Runnable r = new Runnable()
                {
                  @Override
                  public void run()
                  {
                    doWork(s);
```

```
                }
            };
        new Thread(r).start();
    }
  }

  static void doWork(Socket s)
  {
  }
}
```

The above code describes a simple server application (withdoWork(Socket) left empty for brevity). The server thread repeatedly calls socket.accept() to wait for an incoming request, and then starts a thread to service this request when it arrives.

Because this application creates a new thread for each request, it doesn't scale well when faced with a huge number of requests. For example, each created thread requires memory, and too many threads may exhaust the available memory, forcing the application to terminate.

You could solve this problem by changing the task-execution policy. Rather than always creating a new thread, you could use a thread pool, in which a fixed number of threads would service incoming tasks. You would have to rewrite the application to make this change, however.

java.util.concurrent includes the Executor framework, a small framework of types that decouple task submission from task-execution policies. Using the Executor framework, it is possible to easily tune a program's task-execution policy without having to significantly rewrite your code.

Inside the Executor framework

The Executor framework is based on the Executor interface, which describes an *executor* as any object capable of executingjava.lang.Runnable tasks. This interface declares the

following solitary method for executing a Runnable task:

```
void execute(Runnable command)
```

You submit a Runnable task by passing it to execute(Runnable). If the executor cannot execute the task for any reason (for instance, if the executor has been shut down), this method will throw a RejectedExecutionException.

The key concept is that *task submission is decoupled from the task-execution policy*, which is described by an Executorimplementation. The *runnable* task is thus able to execute via a new thread, a pooled thread, the calling thread, and so on.

Note that Executor is very limited. For example, you can't shut down an executor or determine whether an asynchronous task has finished. You also can't cancel a running task. For these and other reasons, the Executor framework provides anExecutorService interface, which extends Executor.

Five of ExecutorService's methods are especially noteworthy:

- **boolean awaitTermination(long timeout, TimeUnit unit)** blocks the calling thread until all tasks have completed execution after a shutdown request, the timeout occurs, or the current thread is interrupted, whichever happens first. The maximum time to wait is specified by timeout, and this value is expressed in the unit units specified by the TimeUnit enum; for example, TimeUnit.SECONDS. This method throws java.lang.InterruptedException when the current thread is interrupted. It returns *true* when the executor is terminated and *false* when the timeout elapses before termination.
- **boolean isShutdown()** returns *true* when the executor has been shut down.
- **void shutdown()** initiates an orderly shutdown in which previously submitted tasks are executed but no new tasks are accepted.
- **<T> Future<T> submit(Callable<T> task)** submits a value-returning task for execution and returns a Future representing the pending results of the task.
- **Future<?> submit(Runnable task)** submits a Runnable task for execution and

returns a Future representing that task.

The Future<V> interface represents the result of an asynchronous computation. The result is known as a *future*because it typically will not be available until some moment in the future. You can invoke methods to cancel a task, return a task's result (waiting indefinitely or for a timeout to elapse when the task hasn't finished), and determine if a task has been cancelled or has finished.

The Callable<V> interface is similar to the Runnable interface in that it provides a single method describing a task to execute. Unlike Runnable's void run() method, Callable<V>'sV call() throws Exception method can return a value and throw an exception.

Executor factory methods

At some point, you'll want to obtain an executor. The Executor framework supplies the Executors utility class for this purpose. Executors offers several factory methods for obtaining different kinds of executors that offer specific thread-execution policies. Here are three examples:

- **ExecutorService newCachedThreadPool()** creates a thread pool that creates new threads as needed, but which reuses previously constructed threads when they're available. Threads that haven't been used for 60 seconds are terminated and removed from the cache. This thread pool typically improves the performance of programs that execute many short-lived asynchronous tasks.
- **ExecutorService newSingleThreadExecutor()** creates an executor that uses a single worker thread operating off an unbounded queue -- tasks are added to the queue and execute sequentially (no more than one task is active at any one time). If this thread terminates through failure during execution before shutdown of the executor, a new thread will be created to take its place when subsequent tasks need to be executed.
- **ExecutorService newFixedThreadPool(int nThreads)** creates a thread pool that re-uses a fixed number of threads operating off a shared unbounded queue. At mostnThreads threads are actively processing tasks. If additional tasks are submitted when all threads are active, they wait in the queue until a thread is

available. If any thread terminates through failure during execution before shutdown, a new thread will be created to take its place when subsequent tasks need to be executed. The pool's threads exist until the executor is shut down.

The Executor framework offers additional types (such as theScheduledExecutorService interface), but the types you are likely to work with most often are ExecutorService, Future,Callable, and Executors.

See the java.util.concurrent Javadoc to explore additional types.

Working with the Executor framework

You'll find that the Executor framework is fairly easy to work with. In Listing 2, I've used Executor and Executors to replace the server example from Listing 1 with a more scalable thread pool-based alternative.

Listing 2. Server.java (Version 2)

```
import java.io.IOException;

import java.net.ServerSocket;
import java.net.Socket;

import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

class Server
{
  static Executor pool = Executors.newFixedThreadPool(5);

  public static void main(String[] args) throws IOException
  {
    ServerSocket socket = new ServerSocket(9000);
    while (true)
    {
```

```
        final Socket s = socket.accept();
        Runnable r = new Runnable()
                     {
                       @Override
                       public void run()
                       {
                         doWork(s);
                       }
                     };
        pool.execute(r);
      }
    }

    static void doWork(Socket s)
    {
    }
}
```

Listing 2 uses newFixedThreadPool(int) to obtain a thread pool-based executor that reuses five threads. It also replaces new Thread(r).start(); with pool.execute(r);for executing runnable tasks via any of these threads.

Listing 3 presents another example in which an application reads the contents of an arbitrary web page. It outputs the resulting lines or an error message if the contents aren't available within a maximum of five seconds.

Listing 3. ReadWebPage.java

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;
```

```java
import java.util.ArrayList;
import java.util.List;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Future;
import java.util.concurrent.TimeoutException;
import java.util.concurrent.TimeUnit;

public class ReadWebPage
{
  public static void main(final String[] args)
  {
    if (args.length != 1)
    {
      System.err.println("usage: java ReadWebPage url");
      return;
    }
    ExecutorService executor = Executors.newSingleThreadExecutor();
    Callable<List<String>> callable;
    callable = new Callable<List<String>>()
            {
              @Override
              public List<String> call()
                throws IOException, MalformedURLException
              {
                List<String> lines = new ArrayList<>();
                URL url = new URL(args[0]);
                HttpURLConnection con;
                con = (HttpURLConnection) url.openConnection();
```

```
                InputStreamReader isr;
                isr = new InputStreamReader(con.getInputStream());
                BufferedReader br;
                br = new BufferedReader(isr);
                String line;
                while ((line = br.readLine()) != null)
                  lines.add(line);
                return lines;
             }
          };
    Future<List<String>> future = executor.submit(callable);
    try
    {
      List<String> lines = future.get(5, TimeUnit.SECONDS);
      for (String line: lines)
        System.out.println(line);
    }
    catch (ExecutionException ee)
    {
      System.err.println("Callable through exception: "+ee.getMessage());
    }
    catch (InterruptedException | TimeoutException eite)
    {
      System.err.println("URL not responding");
    }
    executor.shutdown();
  }
}
```

Listing 3's main() method first verifies that a single (URL-based) command-line argument has been specified. It then creates a single-thread executor and a callable that tries to open a connection to this URL, read its contents line by line, and save these lines in a list, which it returns.

The callable is subsequently submitted to the executor and a future representing the list of strings is returned. main()invokes the future's V get(long timeout, TimeUnit unit)method to obtain this list.

get() throws TimeoutException when the callable doesn't finish within five seconds. It throws ExecutionException when the callable throws an exception (for instance, the callable will throw java.net.MalformedURLException when the URL is invalid).

Regardless of whether an exception is thrown or not, the executor must be shut down before the application exits. If the executor isn't shut down, the application won't exit because the non-daemon thread-pool threads are still executing.

Synchronizers

Synchronizers are high-level constructs that coordinate and control thread execution. The Java Concurrency Utilities framework provides classes that implement semaphore, cyclic barrier, countdown latch, exchanger, and phaser synchronizers. I'll introduce each of these synchronizer types and then show you how they'd work in a concurrent Java application.

Semaphores

A semaphore is a thread-synchronization construct for controlling thread access to a common resource. It's often implemented as a protected variable whose value is incremented by an *acquire* operation and decremented by a*release* operation.

The acquire operation either returns control to the invoking thread immediately or causes that thread to block when the semaphore's current value reaches a certain limit. The release operation decreases the current value, which causes a blocked thread to resume.

Semaphores whose current values can be incremented past 1 are known as counting semaphores, whereas semaphores whose current values can be only 0 or 1 are known as *binary semaphores* or mutexes. In either case, the current value cannot be negative.

The java.lang.concurrent.Semaphore class conceptualizes a semaphore as an object maintaining a set of *permits*. This class provides Semaphore(int permits)

andSemaphore(int permits, boolean fair) constructors for specifying the number of permits.

Each call to the Semaphore's void acquire() method takes one of the available permits or blocks the calling thread when one isn't available. Each call to Semaphore's void release() method returns an available permit, potentially releasing a blocking acquirer thread.

Working with semaphores

Semaphores are often used to restrict the number of threads that can access a resource. Listing 4 demonstrates this capability by using a semaphore to control access to a pool of string items -- the source code is based on Semaphore's Javadoc example code.

Listing 4. SemaphoreDemo.java

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Semaphore;

public class SemaphoreDemo
{
  public static void main(String[] args)
  {
    final Pool pool = new Pool();
    Runnable r = new Runnable()
                 {
                   @Override
                   public void run()
                   {
                     String name = Thread.currentThread().getName();
                     try
```

```
            {
              while (true)
              {
                String item;
                System.out.printf("%s acquiring %s%n", name,
                                  item = pool.getItem());
                Thread.sleep(200+(int)(Math.random()*100));
                System.out.printf("%s putting back %s%n",
name,
                                  item);
                pool.putItem(item);
              }
            }
            catch (InterruptedException ie)
            {
              System.out.printf("%s interrupted%n", name);
            }
          }
        };
    ExecutorService[] executors = new
ExecutorService[Pool.MAX_AVAILABLE+1];
    for (int i = 0; i < executors.length; i++)
    {
      executors[i] = Executors.newSingleThreadExecutor();
      executors[i].execute(r);
    }
  }
}

final class Pool
{
  public static final int MAX_AVAILABLE = 10;
```

```
  private Semaphore available = new Semaphore(MAX_AVAILABLE, true);
  private String[] items;
  private boolean[] used = new boolean[MAX_AVAILABLE];

  Pool()
  {
    items = new String[MAX_AVAILABLE];
    for (int i = 0; i < items.length; i++)
      items[i] = "ITEM"+i;
  }

  String getItem() throws InterruptedException
  {
    available.acquire();
    return getNextAvailableItem();
  }

  void putItem(String item)
  {
    if (markAsUnused(item))
      available.release();
  }

  private synchronized String getNextAvailableItem()
  {
    for (int i = 0; i < MAX_AVAILABLE; ++i)
    {
      if (!used[i])
      {
        used[i] = true;
        return items[i];
      }
    }
```

```
    return null; // not reached
  }

  private synchronized boolean markAsUnused(String item)
  {
    for (int i = 0; i < MAX_AVAILABLE; ++i)
    {
      if (item == items[i])
      {
        if (used[i])
        {
          used[i] = false;
          return true;
        }
        else
          return false;
      }
    }
    return false;
  }
}
```

Listing 4 presents SemaphoreDemo and Pool classes.SemaphoreDemo drives the application by creating executors and having them execute a runnable that repeatedly acquires string item resources from a pool (implemented by Pool) and then returns them.

Pool provides String getItem() and void putItem(String item) methods for obtaining and returning resources. Before obtaining an item in getItem(), a thread must acquire a permit from the semaphore, guaranteeing that an item is available for use. When the thread has finished with the item, it calls putItem(String), which returns the item to the pool and then returns a permit to the semaphore, which lets another thread acquire

that item.

No synchronization lock is held when acquire() is called because that would prevent an item from being returned to the pool. However, String getNextAvailableItem() and boolean markAsUnused(String item) are synchronizedto maintain pool consistency. (The semaphore encapsulates the synchronization needed to restrict access to the pool separately from the synchronization needed to maintain pool consistency.)

Compile Listing 4 (javac SemaphoreDemo.java) and run this application (java SemaphoreDemo). A prefix of the output generated from one run is shown below:

```
pool-1-thread-1 acquiring ITEM0
pool-9-thread-1 acquiring ITEM9
pool-7-thread-1 acquiring ITEM8
pool-5-thread-1 acquiring ITEM7
pool-3-thread-1 acquiring ITEM6
pool-10-thread-1 acquiring ITEM5
pool-8-thread-1 acquiring ITEM4
pool-6-thread-1 acquiring ITEM3
pool-4-thread-1 acquiring ITEM2
pool-2-thread-1 acquiring ITEM1
pool-5-thread-1 putting back ITEM7
pool-11-thread-1 acquiring ITEM7
pool-9-thread-1 putting back ITEM9
pool-5-thread-1 acquiring ITEM9
pool-7-thread-1 putting back ITEM8
pool-9-thread-1 acquiring ITEM8
pool-3-thread-1 putting back ITEM6
pool-7-thread-1 acquiring ITEM6
```

In the above output, eleven threads compete for ten resources. Thread pool-11-thread-1 is forced to wait when it attempts to acquire a resource. It resumes

with the ITEM7 resource when thread pool-5-thread-1 returns this resource to the pool.

Cyclic barriers

A cyclic barrier is a thread-synchronization construct that lets a set of threads wait for each other to reach a common barrier point. The barrier is called *cyclic* because it can be re-used after the waiting threads are released.

A cyclic barrier is implemented by thejava.lang.concurrent.CyclicBarrier class. This class provides the following constructors:

- CyclicBarrier(int nthreads, Runnable barrierAction) causes a maximum ofnthreads-1 threads to wait at the barrier. When one more thread arrives, it executes the *nonnull* barrierAction and then all threads proceed. This action is useful for updating shared state before any of the threads continue.
- CyclicBarrier(int nthreads) is similar to the previous constructor except that no runnable is executed when the barrier is tripped.

Either constructor throwsjava.lang.IllegalArgumentException when the value passed to nthreads is less than 1.

CyclicBarrier declares an int await() method that typically causes the calling thread to wait unless the thread is the final thread. If so, and if a nonnull Runnable was passed tobarrierAction, the final thread executes the runnable before the other threads continue.

await() throws InterruptedException when the thread that invoked this method is interrupted while waiting. This method throws BrokenBarrierException when another thread was interrupted while the invoking thread was waiting, the barrier was broken when await() was called, or the barrier action (when present) failed because an exception was thrown from the runnable's run() method.

Working with cyclic barriers

Cyclic barriers can be used to perform lengthy calculations by breaking them into smaller individual tasks (as demonstrated byCyclicBarrier's Javadoc example code). They're also used in multiplayer games that cannot start until the last player has joined, as shown in Listing 5.

Listing 5. CyclicBarrierDemo.java

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.TimeUnit;

public class CyclicBarrierDemo
{
  public static void main(String[] args)
  {
    Runnable action = new Runnable()
                {
                  @Override
                  public void run()
                  {
                    String name =
Thread.currentThread().getName();
                    System.out.printf("Thread %s "+
                                      "executing barrier
action.%n",
                                      name);
                  }
                };
    final CyclicBarrier barrier = new CyclicBarrier(3, action);
    Runnable task = new Runnable()
                {
```

```
                @Override
                public void run()
                {
                  String name = Thread.currentThread().getName();
                  System.out.printf("%s about to join game...%n",
                                    name);
                  try
                  {
                    barrier.await();
                  }
                  catch (BrokenBarrierException bbe)
                  {
                    System.out.println("barrier is broken");
                    return;
                  }
                  catch (InterruptedException ie)
                  {
                    System.out.println("thread interrupted");
                    return;
                  }
                  System.out.printf("%s has joined game%n", name);
                }
              };
    ExecutorService[] executors = new ExecutorService[]
                        {
                          Executors.newSingleThreadExecutor(),
                          Executors.newSingleThreadExecutor(),
                          Executors.newSingleThreadExecutor()
                        };
    for (ExecutorService executor: executors)
    {
      executor.execute(task);
      executor.shutdown();
```

```
    }
  }
}
```

The above main() method first creates a barrier action that's run by the last thread to reach the barrier. Next, a cyclic barrier is created. When three players arrive it trips and executes the barrier action.

Reusing a CyclicBarrier

To reuse a CyclicBarrier instance, invoke its void reset()method.

main() now creates a runnable that outputs various status messages and invokes await(), followed by a three-executor array. Each executor runs this runnable and shuts down after the runnable finishes.

Compile and run this application. You should see output similar to the following:

```
pool-1-thread-1 about to join game...
pool-3-thread-1 about to join game...
pool-2-thread-1 about to join game...
Thread pool-2-thread-1 executing barrier action.
pool-2-thread-1 has joined game
pool-3-thread-1 has joined game
pool-1-thread-1 has joined game
```

Countdown latches

A *countdown latch* is a thread-synchronization construct that causes one or more threads to wait until a set of operations being performed by other threads finishes. It consists of a count and "cause a thread to wait until the count reaches zero" and "decrement the count" operations.

The java.util.concurrent.CountDownLatch class implements a countdown latch. Its CountDownLatch(int count) constructor initializes the countdown latch to the specified count. A thread invokes the void await() method to wait until the count has reached zero (or the thread has been interrupted). Subsequent calls to await() for a zero count return immediately. A thread calls void countDown() to decrement the count.

Working with countdown latches

Countdown latches are useful for decomposing a problem into smaller pieces and giving a piece to a separate thread, as follows:

1. A main thread creates a countdown latch with a count of 1 that's used as a "starting gate" to start a group of worker threads simultaneously.
2. Each worker thread waits on the latch and the main thread decrements this latch to let all worker threads proceed.
3. The main thread waits on another countdown latch initialized to the number of worker threads.
4. When a worker thread completes, it decrements this count. After the count reaches zero (meaning that all worker threads have finished), the main thread proceeds and gathers the results.

Listing 6 demonstrates this scenario.

Listing 6. CountDownLatchDemo.java

```
import java.util.concurrent.CountDownLatch;

public class CountDownLatchDemo
{
  final static int N = 3;

  public static void main(String[] args) 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();
    System.out.println("about to let threads proceed");
    startSignal.countDown(); // let all threads proceed
    System.out.println("doing work");
    System.out.println("waiting for threads to finish");
    doneSignal.await(); // wait for all threads to finish
    System.out.println("main thread terminating");
```

```
  }
}

class Worker implements Runnable
{
  private final static int N = 5;

  private final CountDownLatch startSignal;
  private final CountDownLatch doneSignal;

  Worker(CountDownLatch startSignal, CountDownLatch doneSignal)
  {
    this.startSignal = startSignal;
    this.doneSignal = doneSignal;
  }

  @Override
  public void run()
  {
    try
    {
      String name = Thread.currentThread().getName();
      startSignal.await();
      for (int i = 0; i < N; i++)
      {
        System.out.printf("thread %s is working%n", name);
        try
        {
          Thread.sleep((int)(Math.random()*300));
        }
        catch (InterruptedException ie)
        {
        }
```

```
        }
        System.out.printf("thread %s finishing%n", name);
        doneSignal.countDown();
      }
      catch (InterruptedException ie)
      {
        System.out.println("interrupted");
      }
    }
}
```

Listing 6 presents CountDownLatchDemo and Worker classes.CountDownLatchDemo's main() method creates a startSignalcountdown latch initialized to 1 and a doneSignal countdown latch initialized to 3, the number of worker threads.

main() proceeds to create three worker threads described byWorker and then start these threads. After outputting a message, main() executes startSignal.countDown() to tell the worker threads that they can proceed.

After outputting a few more messages, main() executesdoneSignal.await() to wait until all worker threads have finished.

Worker's constructor saves these latches, and its run() method performs some work. Before performing this work, the thread executes startSignal.await() to block until the main thread allows it to proceed (by executing startSignal.countDown()).

The worker then enters a loop to simulate doing some work by alternately outputting messages and sleeping for random amounts of time. It then executesdoneSignal.countDown() to decrement the doneSignal countdown latch so that the main thread will eventually wake up.

Compile and run this application. You should see output similar to the following:

```
about to let threads proceed
doing work
```

```
waiting for threads to finish
thread Thread-1 is working
thread Thread-0 is working
thread Thread-2 is working
thread Thread-1 is working
thread Thread-0 is working
thread Thread-0 is working
thread Thread-1 is working
thread Thread-2 is working
thread Thread-1 is working
thread Thread-2 is working
thread Thread-0 is working
thread Thread-0 is working
thread Thread-1 is working
thread Thread-2 is working
thread Thread-1 finishing
thread Thread-0 finishing
thread Thread-2 is working
thread Thread-2 finishing
main thread terminating
```

Exchangers

An *exchanger* (also known as a *rendezvous*) is a thread-synchronization construct that lets a pair of threads exchange data items. An exchanger is similar to a cyclic barrier whose count is set to 2 but also supports exchange of data when both threads reach the barrier.

The java.util.concurrent.Exchanger<V> class implements an exchanger. This class provides an Exchanger() constructor for initializing an exchanger that describes an exchange point and a pair of exchange() methods for performing an exchange.

For example, V exchange(V x) throws InterruptedException waits for another thread to arrive at the exchange point (unless the current thread is interrupted) and then transfers the given object to it, receiving its object in return.

Working with exchangers

Exchanger's Javadoc states that this synchronizer may be useful in genetic algorithms and pipeline designs, where one thread fills a buffer and the other thread empties the buffer. When both threads meet at the exchange point, they swap their buffers. Listing 7 demonstrates.

Listing 7. ExchangerDemo.java

```
import java.util.ArrayList;
import java.util.List;

import java.util.concurrent.Exchanger;

public class ExchangerDemo
{
  static Exchanger<DataBuffer> exchanger = new
Exchanger<DataBuffer>();
  static DataBuffer initialEmptyBuffer = new DataBuffer();
  static DataBuffer initialFullBuffer = new DataBuffer("ITEM");

  public static void main(String[] args)
  {
    class FillingLoop implements Runnable
    {
      int count = 0;

      @Override
      public void run()
      {
        DataBuffer currentBuffer = initialEmptyBuffer;
        try
        {
          while (true)
          {
            addToBuffer(currentBuffer);
```

```
            if (currentBuffer.isFull())
             {
               System.out.println("filling loop thread wants to
exchange");
               currentBuffer = exchanger.exchange(currentBuffer);
               System.out.println("filling loop thread observes an
exchange");
             }
           }
         }
         catch (InterruptedException ie)
         {
           System.out.println("filling loop thread interrupted");
         }
       }

       void addToBuffer(DataBuffer buffer)
       {
         String item = "NEWITEM"+count++;
         System.out.printf("Adding %s%n", item);
         buffer.add(item);
       }
     }

     class EmptyingLoop implements Runnable
     {
       @Override
       public void run()
       {
         DataBuffer currentBuffer = initialFullBuffer;
         try
         {
           while (true)
```

```
            {
                takeFromBuffer(currentBuffer);
                if (currentBuffer.isEmpty())
                {
                    System.out.println("emptying loop thread wants to
exchange");
                    currentBuffer = exchanger.exchange(currentBuffer);
                    System.out.println("emptying loop thread observes an
exchange");
                }
            }
        }
        catch (InterruptedException ie)
        {
            System.out.println("emptying loop thread interrupted");
        }
    }

    void takeFromBuffer(DataBuffer buffer)
    {
        System.out.printf("taking %s%n", buffer.remove());
    }
  }

  new Thread(new EmptyingLoop()).start();
  new Thread(new FillingLoop()).start();
 }
}

class DataBuffer
{
 private final static int MAX = 10;
 private List<String> items = new ArrayList<>();
```

```
  DataBuffer()
  {
  }

  DataBuffer(String prefix)
  {
     for (int i = 0; i < MAX; i++)
     {
        String item = prefix+i;
        System.out.printf("Adding %s%n", item);
        items.add(item);
     }
  }

  void add(String s)
  {
     if (!isFull())
        items.add(s);
  }

  boolean isEmpty()
  {
     return items.size() == 0;
  }

  boolean isFull()
  {
     return items.size() == MAX;
  }

  String remove()
  {
```

```
    if (!isEmpty())
      return items.remove(0);
    return null;
  }
}
```

Listing 7 is based on the example code in Exchanger's Javadoc. One thread fills one buffer with strings while another thread empties another buffer. When the respective buffer is full or empty, these threads meet at an exchange point and swap buffers.

For example, when the filling thread'scurrentBuffer.isFull() expression is true, it executescurrentBuffer = exchanger.exchange(currentBuffer) and waits. The emptying thread continues untilcurrentBuffer.isEmpty() evaluates to true, and also invokesexchange(currentBuffer). At this point, the buffers are swapped and the threads continue.

Compile and run this application. Your initial output should be similar to the following prefix:

```
Adding ITEM0
Adding ITEM1
Adding ITEM2
Adding ITEM3
Adding ITEM4
Adding ITEM5
Adding ITEM6
Adding ITEM7
Adding ITEM8
Adding ITEM9
taking ITEM0
taking ITEM1
taking ITEM2
taking ITEM3
taking ITEM4
taking ITEM5
```

taking ITEM6
taking ITEM7
taking ITEM8
Adding NEWITEM0
taking ITEM9
Adding NEWITEM1
emptying loop thread wants to exchange
Adding NEWITEM2
Adding NEWITEM3
Adding NEWITEM4
Adding NEWITEM5
Adding NEWITEM6
Adding NEWITEM7
Adding NEWITEM8
Adding NEWITEM9
filling loop thread wants to exchange
filling loop thread observes an exchange
emptying loop thread observes an exchange
Adding NEWITEM10
Adding NEWITEM11
taking NEWITEM0
taking NEWITEM1
Adding NEWITEM12
taking NEWITEM2
taking NEWITEM3
Adding NEWITEM13
taking NEWITEM4
taking NEWITEM5
Adding NEWITEM14
taking NEWITEM6
taking NEWITEM7
Adding NEWITEM15
taking NEWITEM8

taking NEWITEM9
emptying loop thread wants to exchange
Adding NEWITEM16
Adding NEWITEM17
Adding NEWITEM18
Adding NEWITEM19
filling loop thread wants to exchange
filling loop thread observes an exchange
emptying loop thread observes an exchange
Adding NEWITEM20

Phasers

A *phaser* is a thread-synchronization construct that's similar to a cyclic barrier in that it lets a group of threads wait on a barrier and then proceed after the last thread arrives. It also offers the equivalent of a barrier action. However, a phaser is more flexible.

Unlike a cyclic barrier, which coordinates a fixed number of threads, a phaser can coordinate a variable number of threads, which can register at any time. To implement this capability, a phaser takes advantage of phases and phase numbers.

A *phase* is the phaser's current state, and this state is identified by an integer-based *phase number*. When the last of the registered threads arrives at the *phaser barrier*, a phaser advances to the next phase and increments its phase number by 1.

The java.util.concurrent.Phaser class implements a phaser. Because this class is thoroughly described in its Javadoc, I'll point out only a few constructors and methods:

- The Phaser(int threads) constructor creates a phaser that initially coordinatesnthreads threads (which have yet to arrive at the phaser barrier) and whose phase number is initially set to 0.
- The int register() method adds a new unarrived thread to this phaser and returns the phase number to which the arrival applies. This number is known as the *arrival phase number*.

- The int arriveAndAwaitAdvance() method records arrival and waits for the phaser to advance (which happens after the other threads have arrived). It returns the phase number to which the arrival applies.
- The int arriveAndDeregister() method arrives at this phaser and deregisters from it without waiting for others to arrive, reducing the number of threads required to advance in future phases.

Working with phasers

The small application in Listing 8 demonstrates the constructor and methods described above.

Listing 8. PhaserDemo.java

```
import java.util.ArrayList;
import java.util.List;

import java.util.concurrent.Phaser;

public class PhaserDemo
{
  public static void main(String[] args)
  {
    List<Runnable> tasks = new ArrayList<>();
    tasks.add(new Runnable()
          {
            @Override
            public void run()
            {
              System.out.printf("%s running at %d%n",
                        Thread.currentThread().getName(),
                        System.currentTimeMillis());
            }
          });
```

```
    tasks.add(new Runnable()
            {
                @Override
                public void run()
                {
                  System.out.printf("%s running at %d%n",
                                    Thread.currentThread().getName(),
                                    System.currentTimeMillis());
                }
              });
    runTasks(tasks);
  }

  static void runTasks(List<Runnable> tasks)
  {
    final Phaser phaser = new Phaser(1); // "1" to register self
    // create and start threads
    for (final Runnable task: tasks)
    {
      phaser.register();
      new Thread()
      {
        @Override
        public void run()
        {
          try
          {
            Thread.sleep(50+(int)(Math.random()*300));
          }
          catch (InterruptedException ie)
          {
            System.out.println("interrupted thread");
          }
```

```
            phaser.arriveAndAwaitAdvance(); // await all creation
            task.run();
          }
        }.start();
      }

      // allow threads to start and deregister self
      phaser.arriveAndDeregister();
    }
}
```

Listing 8 is based on the first code example in Phaser's Javadoc. This example shows how to use Phaser instead ofCountDownLatch to control a one-shot action serving a variable number of threads.

The application creates a pair of runnable tasks that each report the time (in milliseconds relative to the Unix epoch) at which its starts to run. Compile and run this application, and you should observe output that's similar to the following:

```
Thread-0 running at 1366315297635
Thread-1 running at 1366315297635
```

As you would expect from countdown latch behavior, both threads start running at (in this case) the same time even though a thread may have been delayed by as much as 349 milliseconds thanks to the presence of Thread.sleep().

Comment out phaser.arriveAndAwaitAdvance(); // await all creation and you should now observe the threads starting at radically different times, as illustrated below:

```
Thread-1 running at 1366315428871
Thread-0 running at 1366315429100
```

Concurrent collections

The Java Collections framework provides interfaces and classes in the java.utilpackage that facilitate working with collections of objects. Interfaces include List,Map, and Set. Classes include ArrayList, Vector, Hashtable, HashMap, and TreeSet.

Collections classes such as Vector and Hashtable are thread-safe. You can make other classes (like ArrayList) thread-safe by using synchronized wrapper factory methods such asCollections.synchronizedMap(),Collections.synchronizedList(), andCollections.synchronizedSet().

There are a couple of problems with the thread-safe collections:

1. Code that iterates over a collection that might be modified by another thread during the iteration requires a lock to avoid a thrown java.util.ConcurrentModificationException. This requirement is necessary because Collections framework classes return *fail-fast iterators*, which are iterators that throw ConcurrentModificationException when a collection is modified during iteration. Fail-fast iterators are often an inconvenience to concurrent applications.
2. Performance often suffers when these synchronized collections are accessed frequently from multiple threads; this is a performance problem that impacts an application's scalability.

The Java Concurrency Utilities framework overcomes these problems by introducing performant and highly-scalable collections-oriented types, which are part ofjava.util.concurrent. These collections-oriented classes return *weakly consistent iterators*, which have the following properties:

- When an element is removed after iteration starts, but hasn't yet been returned via the iterator's next()method, it won't be returned.
- When an element is added after iteration starts, it may or may not be returned.
- Regardless of changes to the collection, no element is returned more than once in an iteration.

The following list summarizes the Java Concurrency Utilities framework's collection-oriented types:

- BlockingDeque<E>: This interface extends BlockingQueueand java.util.Deque to describe a double-ended queue with additional support for blocking operations that wait for the deque to become non-empty when retrieving an element, and wait for space to become available in the deque when storing an element.
- BlockingQueue<E>: This interface extendsjava.util.Queue to describe a queue with additional support for operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.
- ConcurrentMap<K, V>: This interface extends java.util.Map to describe a map with additional atomic putIfAbsent, remove, and replace methods.
- ConcurrentNavigableMap<K, V>: This interface extends ConcurrentMap andjava.util.NavigableMap to describe a concurrent map with navigable operations.
- TransferQueue<E>: This interface extends BlockingQueue to describe a blocking queue in which producers may wait for consumers to receive elements.
- ArrayBlockingQueue<E>: This class describes a bounded blocking queue backed by an array.
- ConcurrentHashMap<K, V>: This class describes a hash table supporting full concurrency of retrievals and adjustable expected concurrency for updates.
- ConcurrentLinkedDeque<E>: This class describes an unbounded thread-safe deque based on linked nodes.
- ConcurrentLinkedQueue<E>: This class describes an unbounded thread-safe queue based on linked nodes.
- ConcurrentSkipListMap<K, V>: This class describes a scalable concurrentConcurrentNavigableMap implementation.
- ConcurrentSkipListSet<E>: This class describes a scalable concurrentjava.util.NavigableSet implementation based on a ConcurrentSkipListMap.
- CopyOnWriteArrayList<E>: This class describes a thread-safe variant of ArrayList in which all mutative operations (e.g., add and set) are implemented by making a fresh copy of the underlying array whenever an element is added or removed.

However, in-progress iterations continue to work on the previous copy (when the iterator was created). Although there's some cost to copying the array, this cost is acceptable in situations where there are many more iterations than modifications.

- CopyOnWriteArraySet<E>: This class describes a Set that uses an internalCopyOnWriteArrayList for all of its operations.
- DelayQueue<E extends Delayed>: This class describes an unbounded blocking queue of java.util.concurrent.Delayed elements, in which an element can only be taken when its delay has expired. (Delayed is an interface for marking objects that should be acted upon after a given delay.)
- LinkedBlockingDeque<E>: This class describes an optionally-bounded blocking deque based on linked nodes.
- LinkedBlockingQueue<E>: This class describes an optionally-bounded blocking queue based on linked nodes.
- LinkedTransferQueue<E>: This class describes an unbounded transfer queue based on linked nodes.
- PriorityBlockingQueue<E>: This class describes an unbounded blocking queue that uses the same ordering rules as java.util.PriorityQueue and supplies blocking retrieval operations.
- SynchronousQueue<E>: This class describes a blocking queue in which each insert operation must wait for a corresponding remove operation by another thread, and vice versa.

Working with Concurrent Collections

For an example of what you can do with Concurrent Collections, consider CopyOnWriteArrayList, as demonstrated in Listing 9.

Listing 9. CopyOnWriteArrayListDemo.java

```
import java.util.ArrayList;
import java.util.ConcurrentModificationException;
import java.util.Iterator;
import java.util.List;
```

```
import java.util.concurrent.CopyOnWriteArrayList;

public class CopyOnWriteArrayListDemo
{
  public static void main(String[] args)
  {
    List<String> empList = new ArrayList<>();
    empList.add("John Doe");
    empList.add("Jane Doe");
    empList.add("Rita Smith");
    Iterator<String> empIter = empList.iterator();
    while (empIter.hasNext())
      try
      {
        System.out.println(empIter.next());
        if (!empList.contains("Tom Smith"))
          empList.add("Tom Smith");
      }
      catch (ConcurrentModificationException cme)
      {
        System.err.println("attempt to modify list during iteration");
        break;
      }

    List<String> empList2 = new CopyOnWriteArrayList<>();
    empList2.add("John Doe");
    empList2.add("Jane Doe");
    empList2.add("Rita Smith");
    empIter = empList2.iterator();
    while (empIter.hasNext())
    {
      System.out.println(empIter.next());
      if (!empList2.contains("Tom Smith"))
```

```
            empList2.add("Tom Smith");
        }

    }
}
```

Listing 9 contrasts CopyOnWriteArrayListDemo with ArrayListfrom a ConcurrentModificationException perspective. During each iteration, an attempt is made to add a new employee name to the list. The ArrayList iteration fails with this exception, whereas the CopyOnWriteArrayList iteration ignores the addition.

If you compile and run this application you should see the following output:

```
John Doe
attempt to modify list during iteration
John Doe
Jane Doe
Rita Smith
```

In conclusion

The Java Concurrency Utilities framework offers a high-level alternative to Java's low-level threading capabilities. This library's thread-safe and high-performant types were designed to be used as the building blocks in concurrent classes and applications.

The Java Concurrency Utilities framework is organized into several smaller frameworks, with types stored in java.util.concurrent and two subpackages,java.util.concurrent.atomic and java.util.concurrent.locks.

In this article, I introduced three of these frameworks: the Executor framework, synchronizers, and the Java Concurrent Collections. You can learn more about them by exploring the exercises in this article's source code file. In Part 2 we'll explore locks, atomic variables, Fork/Join, and more.

Jeff Friesen is a freelance tutor and software developer with an emphasis on Java and Android. In addition to writing Java and Android books for Apress, Jeff has written numerous articles on Java and other technologies for JavaWorld, informIT,Java.net, DevSource, and SitePoint. Jeff can be contacted via his website atTutorTutor.ca.

Java 101: The next generation: Java concurrency without the pain, Part 2

Locking, atomic variables, Fork/Join, and what to expect in Java 8

The Java Concurrency Utilities are high-level concurrency types that facilitate threading tasks especially on multicore systems. Part 1 of this introduction featured java.util.concurrent's Executor framework, synchronizer types, and Java Concurrent Collections package. In Part 2, learn how the Java Concurrency Utilities handle locking, atomic variables, and fork/join operations. Then prepare for the future with an overview of seven anticipated changes to the Java Concurrency Utilities coming in Java 8.

The Locking framework

Java's low-level threading capabilities are famously hard to use and error-prone, which is why they're frequently associated with deadlock, thread starvation, race conditions, and other concurrency bugs. One alternative to losing your sleep and peace of mind is the Java Concurrency Utilities, introduced in JDK 5. The two articles in this short series are dedicated to exploring how Java developers use the packages and libraries injava.util.concurrent to work around common threading bugs and write cleaner, simpler programs.

In Part 1, I explored the Executor framework, synchronizer utilities, and the Java Concurrent Collections package. Part 2 is an in-depth look at the mechanics ofjava.util.concurrent's advanced locking mechanisms and atomic variables, as well as a short tutorial on the Fork/Join framework. I also discuss the new features and performance improvements coming to the Java Concurrency Utilities with Java 8.

The Java language lets threads use *synchronization* to update shared variables safely and ensure that one thread's updates are visible to other threads. In the Java language, you call synchronization via the synchronized keyword. The Java virtual machine (JVM) supports this mechanism via monitors and the associated monitorenter and monitorexit instructions.

Each Java object is associated with a *monitor*, which is a mutual exclusion mechanism that prevents multiple threads from concurrently executing in a critical section. Before a thread can enter this section, it must lock the monitor. If the monitor is already locked, the thread blocks until the monitor is unlocked.

Monitors also address the vagaries of memory caching and compiler optimizations that might otherwise prevent one thread from observing another thread's update of a shared variable. Before a thread leaves the critical section, the monitor ensures that the thread's updates are immediately visible, so that another thread about to enter the critical section will see those updates.

Synchronization vs volatile

Synchronization supports mutual exclusion and visibility. In contrast, thevolatile keyword only supports visibility.

Although adequate for simple applications, Java's low-level synchronization mechanism can be inconvenient for advanced applications that require additional capabilities such as timed waits and lock polling.

The Locking framework, which is found injava.util.concurrent.locks, addresses these limitations.

Locks

The Lock interface provides more extensive locking operations than it's possible to obtain via synchronized methods and statements. For instance, you could use Lock to immediately back out of a lock-acquisition attempt if a lock wasn't available, or you could wait indefinitely to acquire a lock and back out only if interrupted.

Lock declares the following methods:

- void lock() acquires a lock, disabling the current thread when the lock is not available. The thread remains dormant until the lock becomes available.

- void lockInterruptibly() is similar to void lock() but allows the disabled thread to be interrupted and resume execution through a thrownjava.lang.InterruptedException. (Note that interrupting lock acquisition is not supported in all cases.)
- Condition newCondition() returns a new Conditioninstance that's bound to a given Lock instance. If theLock implementation doesn't support conditions,java.lang.UnsupportedOperationException is thrown. (I discuss conditions later in this article.)
- void lock() acquires a lock, disabling the current thread when the lock is not available. The thread remains dormant until the lock becomes available.
- boolean tryLock() acquires a lock only when it's free at the time of invocation. This method returns true when the lock is acquired; otherwise, it returns false.
- boolean tryLock(long time, TimeUnit unit) is similar to boolean tryLock(); however, it lets you specify an amount of time to wait for the lock to become available. Pass the magnitude of the delay to time and the units represented by this delay to unit. For example, you might pass 2 to time and TimeUnit.SECONDS to unit. (The java.util.concurrent.TimeUnit enum also offers DAYS, HOURS, MICROSECONDS,MILLISECONDS, MINUTES, and NANOSECONDS.) This method throws InterruptedExceptionwhen the current thread is interrupted while acquiring the lock (in cases where interrupting lock acquisition is supported).
- void unlock() releases the lock.

It's important to always release a held lock. The Javadoc for the Lock interface presents the following idiom for locking a lock and ensuring that the lock is always unlocked:

```
Lock l = ...; // ... is a placeholder for code that obtains the lock
l.lock();
try
{
  // access the resource protected by this lock
}
catch(Exception ex)
```

```
{
  // restore invariants
}
finally
{
   l.unlock();
}
```

Lock's Javadoc also discusses the memory synchronization semantics that are expected of Lock implementations. Essentially, Lock implementations must behave as the built-in monitor lock, enforcing mutual exclusion and visibility.

Working with locks

The ReentrantLock class implements Lock and describes a *reentrant* mutual exclusion lock. The lock is associated with an acquisition count. When a thread holds the lock and re-acquires the lock, the acquisition count is incremented and the lock must be released twice.

ReentrantLock offers the same concurrency and memory semantics as the implicit monitor lock normally accessed using synchronized methods and statements. However, it has extended capabilities and offers better performance under high *thread contention* (that is, when threads are frequently asking to acquire a lock that is already held by another thread). When many threads attempt to access a shared resource, the JVM spends less time scheduling these threads and more time executing them.

ReentrantLock or synchronized?

ReentrantLock behaves like synchronized and you might wonder when it's appropriate to use one or the other. UseReentrantLock when you need timed or interruptible lock waits, non-block-structured locks (obtain a lock in one method; return the lock in another), multiple condition variables, or lock polling. Furthermore,

ReentrantLocksupports scalability and is useful where there is high contention among threads. If none of these factors come into play, use synchronized.

ReentrantLock declares the following constructors:

- ReentrantLock() creates a reentrant lock.
- ReentrantLock(boolean fair) creates a reentrant lock with the given fairness policy. Passing true to fair results in a lock that uses a *fair ordering policy*, which means that under contention, the lock favors granting access to the longest-waiting thread. The former constructor invokes this constructor, passing false to fair.

ReentrantLock implements Lock's methods: its implementation of unlock() throws java.lang.IllegalMonitorStateExceptionwhen the calling thread doesn't hold the lock. Additionally,ReentrantLock provides its own methods, including the following trio:

- int getHoldCount() returns the number of *holds* on this lock by the current thread: a thread has a hold on a lock for each lock action that isn't matched by an unlock action. When the lock() method is called and the current thread already holds the lock, the hold count is incremented by one and the method returns immediately.
- boolean isFair() returns the fairness setting.
- boolean isHeldByCurrentThread() queries if this lock is held by the current thread, returning true when this is the case. This method is often used for debugging and testing.

Listing 1 is a simple demonstration of ReentrantLock.

Listing 1. LockDemo.java

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.TimeUnit;

import java.util.concurrent.locks.ReentrantLock;

public class LockDemo
```

```
{
  public static void main(String[] args)
  {
    ExecutorService executor = Executors.newFixedThreadPool(2);
    final ReentrantLock rl = new ReentrantLock();

    class Worker implements Runnable
    {
      private String name;

      Worker(String name)
      {
        this.name = name;
      }

      @Override
      public void run()
      {
        rl.lock();
        try
        {
          if (rl.isHeldByCurrentThread())
            System.out.printf("Thread %s has entered its critical
section.%n",
                              name);
          System.out.printf("Thread %s is performing work for 2
seconds.%n", name);
          try
          {
            Thread.sleep(2000);
          }
          catch (InterruptedException ie)
          {
```

```
                ie.printStackTrace();
            }
            System.out.printf("Thread %s has finished working.%n",
name);
        }
        finally
        {
            rl.unlock();
        }
      }
    }

    executor.execute(new Worker("A"));
    executor.execute(new Worker("B"));

    try
    {
      executor.awaitTermination(5, TimeUnit.SECONDS);
    }
    catch (InterruptedException ie)
    {
      ie.printStackTrace();
    }
    executor.shutdownNow();
  }
}
```

Listing 1 creates two worker threads. Each thread first acquires a lock to ensure that it has complete access to the critical section. It then outputs some messages and sleeps for two seconds to simulate work. After outputting another message, it releases the lock.

Compile LockDemo.java and run the application. You should observe output similar to the following:

```
Thread A has entered its critical section.
Thread A is performing work for 2 seconds.
Thread A has finished working.
Thread B has entered its critical section.
Thread B is performing work for 2 seconds.
Thread B has finished working.
```

Comment out `rl.lock();` and `rl.unlock();` and you should observe interleaved output like what is shown below:

```
Thread A is performing work for 2 seconds.
Thread B is performing work for 2 seconds.
Thread A has finished working.
Thread B has finished working.
```

Conditions

The `Condition` interface factors out the `java.lang.Object`monitor methods (`wait()`, `notify()`, and `notifyAll()`) into distinct objects to give the effect of having multiple wait-sets per object, by combining them with the use of arbitrary `Lock`implementations. Where `Lock` replaces `synchronized` methods and statements, `Condition` replaces `Object` monitor methods.

`Condition` declares the following methods:

- `void await()` forces the current thread to wait until it's signalled or interrupted.
- `boolean await(long time, TimeUnit unit)` forces the current thread to wait until it's signalled or interrupted, or the specified waiting time elapses.
- `long awaitNanos(long nanosTimeout)` forces the current thread to wait until it's signalled or interrupted, or the specified waiting time elapses.
- `void awaitUninterruptibly()` forces the current thread to wait until it's signalled.
- `boolean awaitUntil(Date deadline)` forces the current thread to wait until it's signalled or interrupted, or the specified deadline elapses.
- `void signal()` wakes up one waiting thread.

- void signalAll() wakes up all waiting threads.

Working with conditions

The classic producer-consumer example nicely demonstrates conditions. In this example, a producer thread repeatedly produces items for consumption by a consumer thread.

The producer thread must not produce a new item until the previously produced item has been consumed. Similarly, the consumer thread must not consume an item that hasn't been produced. This is known as *lockstep synchronization*.

Listing 2 demonstrates conditions (and locks) in a producer-consumer context.

Listing 2. CondDemo.java

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

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

class Shared
{
   // Fields c and available are volatile so that writes to them are visible to
   // the various threads. Fields lock and condition are final so that they're
   // initial values are visible to the various threads. (The Java memory
model
```

```
  // promises that, after a final field has been initialized, any thread will
  // see the same [correct] value.)

  private volatile char c;
  private volatile boolean available;
  private final Lock lock;
  private final Condition condition;

  Shared()
  {
    c = '\u0000';
    available = false;
    lock = new ReentrantLock();
    condition = lock.newCondition();
  }

  Lock getLock()
  {
    return lock;
  }

  char getSharedChar()
  {
    lock.lock();
    try
    {
      while (!available)
        try
        {
          condition.await();
        }
        catch (InterruptedException ie)
        {
```

```
            ie.printStackTrace();
          }
        available = false;
        condition.signal();
      }
      finally
      {
        lock.unlock();
        return c;
      }
    }

    void setSharedChar(char c)
    {
      lock.lock();
      try
      {
        while (available)
          try
          {
            condition.await();
          }
          catch (InterruptedException ie)
          {
            ie.printStackTrace();
          }
        this.c = c;
        available = true;
        condition.signal();
      }
      finally
      {
        lock.unlock();
```

```
    }
  }
}

class Producer extends Thread
{
  // l is final because it's initialized on the main thread and accessed on the
  // producer thread.

  private final Lock l;

  // s is final because it's initialized on the main thread and accessed on the
  // producer thread.

  private final Shared s;

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

  @Override
  public void run()
  {
    for (char ch = 'A'; ch <= 'Z'; ch++)
    {
      l.lock();
      s.setSharedChar(ch);
      System.out.println(ch + " produced by producer.");
      l.unlock();
```

```
    }
  }
}
class Consumer extends Thread
{
  // l is final because it's initialized on the main thread and accessed on
the
  // consumer thread.

  private final Lock l;

  // s is final because it's initialized on the main thread and accessed on
the
  // consumer thread.

  private final Shared s;

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

  @Override
  public void run()
  {
    char ch;
    do
    {
      l.lock();
      ch = s.getSharedChar();
      System.out.println(ch + " consumed by consumer.");
      l.unlock();
```

```
    }
    while (ch != 'Z');
  }
}
```

Listing 2 presents four classes: CondDemo, Shared, Producer, andConsumer. CondDemo drives the application, Shared encapsulates the logic for setting and getting a shared variable's value,Producer describes the producer thread, and Consumer describes the consumer thread.

The mechanics of CondDemo

CondDemo's main() method instantiates Shared, Producer, andConsumer. It passes the Shared instance to the Producer andConsumer thread instance constructors and starts these threads.

The Producer and Consumer constructors are invoked on the main thread. Because the Shared instance is also accessed on the producer and consumer threads, it's necessary for this instance to be visible to them, especially when these threads run on different cores. Within each of Producer and Consumer, I accomplish this task by declaring s final. I could have declared this field volatile, but volatile suggests that there will be further writes to the field and s isn't supposed to change after being initialized.

Shared's constructor creates a lock (lock = new ReentrantLock();) and an associated condition (condition = lock.newCondition();). This lock is made available to the producer and consumer threads via the Lock getLock() method.

The producer thread always invokes the void setSharedChar(char c) method to generate a new character. This method locks the previously created lock object and then enters a while loop that repeatedly tests variable available -- this variable is true when a produced character hasn't yet been consumed.

As long as available is true, the producer invokes the condition's await() method to wait for available to become false. The consumer will signal the condition to wake up the

producer when it has consumed the character. (A loop is used instead of an if statement because spurious wakeups are possible and available might still be true.)

After exiting the loop, the producer records the new character, assigns true to available to indicate that a new character is available for consumption, and signals the condition to wake up a waiting consumer. Lastly, it unlocks the lock and returns from setSharedChar().

Locking controls output order

Why am I locking the get/print and set/print code blocks? Without this locking, you might observe consuming messages before producing messages, even though characters are produced before they're consumed. Locking these blocks prevents this strange output order.

The behavior of the consumer thread in the char getSharedChar() method is similar.

The mechanics of the producer and consumer threads are simpler. Each run() method first locks the lock, then sets or gets a character and outputs a message, and unlocks the lock. (I didn't use the try/finally idiom because an exception isn't thrown from this context.)

Compile CondDemo.java and run the application. You should observe the following output:

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

F consumed **by** consumer.
G produced **by** producer.
G consumed **by** consumer.
H produced **by** producer.
H consumed **by** consumer.
I produced **by** producer.
I consumed **by** consumer.
J produced **by** producer.
J consumed **by** consumer.
K produced **by** producer.
K consumed **by** consumer.
L produced **by** producer.
L consumed **by** consumer.
M produced **by** producer.
M consumed **by** consumer.
N produced **by** producer.
N consumed **by** consumer.
O produced **by** producer.
O consumed **by** consumer.
P produced **by** producer.
P consumed **by** consumer.
Q produced **by** producer.
Q consumed **by** consumer.
R produced **by** producer.
R consumed **by** consumer.
S produced **by** producer.
S consumed **by** consumer.
T produced **by** producer.
T consumed **by** consumer.
U produced **by** producer.
U consumed **by** consumer.
V produced **by** producer.
V consumed **by** consumer.

```
W produced by producer.
W consumed by consumer.
X produced by producer.
X consumed by consumer.
Y produced by producer.
Y consumed by consumer.
Z produced by producer.
Z consumed by consumer.
```

Read-write locks

You'll occasionally encounter a situation where data structures are read more often than they're modified. The Locking framework has a read-write locking mechanism for these situations that yields both greater concurrency when reading and the safety of exclusive access when writing.

The ReadWriteLock interface maintains a pair of associated locks, one for read-only operations and one for write operations. The *read lock* may be held simultaneously by multiple reader threads as long as there are no writers. The*write lock* is exclusive: only a single thread can modify shared data. (The lock that's associated with the synchronized keyword is also exclusive.)

ReadWriteLock declares the following methods:

- Lock readLock() returns the lock that's used for reading.
- Lock writeLock() returns the lock that's used for writing.

Working with read-write locks

The ReentrantReadWriteLock class implements ReadWriteLock and describes a read-write lock with similar semantics to a reentrant lock. Like ReentrantLock,ReentrantReadWriteLock declares a pair of constructors:

- ReentrantReadWriteLock() creates a reentrant read-write lock with default (nonfair) ordering properties.
- ReentrantReadWriteLock(boolean fair) creates a reentrant read-write lock with the given fairness policy.

ReentrantReadWriteLock implements ReadWriteLock's methods and provides additional methods, including the following trio:

- int getQueueLength() returns an estimate of the number of threads waiting to acquire either the read or write lock.
- int getReadHoldCount() returns the number of read holds on this lock by the current thread. A reader thread has a hold on a lock for each lock action that is not matched by an unlock action.
- boolean hasWaiters(Condition condition) returns true when there are threads waiting on the given condition associated with the write lock.

Listing 3 demonstrates ReentrantReadWriteLock.

LIsting 3. RWLockDemo.java

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.ThreadFactory;
import java.util.concurrent.TimeUnit;
```

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class RWLockDemo
{
  final static int DELAY = 80;
  final static int NUMITER = 5;

  public static void main(String[] args)
  {
    final Names names = new Names();

    class NamedThread implements ThreadFactory
    {
      private String name;

      NamedThread(String name)
      {
        this.name = name;
      }

      @Override
      public Thread newThread(Runnable r)
      {
        return new Thread(r, name);
      }
    }

    ExecutorService writer;
    writer = Executors.newSingleThreadExecutor(new
NamedThread("writer"));
    Runnable wrunnable = new Runnable()
                  {
```

```
                    @Override
                    public void run()
                    {
                      for (int i = 0; i < NUMITER; i++)
                      {
                        names.add(Thread.currentThread().getName(),
                                "A" + i);
                        try
                        {
                          Thread.sleep(DELAY);
                        }
                        catch (InterruptedException ie)
                        {
                        }
                      }
                    }
                  };
    writer.submit(wrunnable);

    ExecutorService reader1;
    reader1 = Executors.newSingleThreadExecutor(new
NamedThread("reader1"));
    ExecutorService reader2;
    reader2 = Executors.newSingleThreadExecutor(new
NamedThread("reader2"));
    Runnable rrunnable = new Runnable()
                  {
                    @Override
                    public void run()
                    {
                      for (int i = 0; i < NUMITER; i++)

names.dump(Thread.currentThread().getName());
```

```
                    }
                };
        reader1.submit(rrunnable);
        reader2.submit(rrunnable);

        reader1.shutdown();
        reader2.shutdown();
        writer.shutdown();
    }
}

class Names
{
    private final List<String> names;

    private final ReentrantReadWriteLock lock;
    private final Lock readLock, writeLock;

    Names()
    {
        names = new ArrayList<>();
        lock = new ReentrantReadWriteLock();
        readLock = lock.readLock();
        writeLock = lock.writeLock();
    }

    void add(String threadName, String name)
    {
        writeLock.lock();
        try
        {
            System.out.printf("%s: num waiting threads = %d%n",
                    threadName, lock.getQueueLength());
```

```
      names.add(name);
    }
    finally
    {
      writeLock.unlock();
    }
  }

  void dump(String threadName)
  {
    readLock.lock();
    try
    {
      System.out.printf("%s: num waiting threads = %d%n",
                        threadName, lock.getQueueLength());
      Iterator<String> iter = names.iterator();
      while (iter.hasNext())
      {
        System.out.printf("%s: %s%n", threadName, iter.next());
        try
        {
          Thread.sleep((int)(Math.random()*100));
        }
        catch (InterruptedException ie)
        {
        }
      }
    }
    finally
    {
      readLock.unlock();
    }
```

```
  }
}
```

Listing 3 describes an application where a writer thread appends names to a list of names and a pair of reader threads repeatedly dump this list to the standard output.

The mechanics of RWLockDemo

The main() method first instantiates the Names class, which stores the list of names and provides methods for adding names to and dumping the list. It then declares NamedThread.

NamedThread is a local class that is subsequently used in an executor context to provide a name for the executor's thread. It implements the java.util.concurrent.ThreadFactory interface and its Thread newThread(Runnable r) method, which returns a new thread whose name was previously passed to the NamedThread(String name) constructor.

Next, main() invokes java.util.concurrent.Executors's ExecutorService newSingleThreadExecutor(ThreadFactory threadFactory) method to create an executor for the writer thread. The name is obtained from the NamedThread instance.

A runnable for the writer thread is then created and submitted to the executor. The runnable repeatedly creates and adds a name to the list of names, and then delays for a short amount of time to give the reader threads a chance to run.

A pair of executors for the reader threads are now created along with a shared runnable for repeatedly dumping the names list. This runnable is submitted to each of the reader executors.

Lastly, main() invokes shutdown() on each executor to initiate an orderly shutdown of the executor as soon as it finishes.

About Names

Names is a simple class that demonstrates read-write locks. It first declares list-of-names, reentrant read-write lock, read-lock, and write-lock fields followed by a constructor that initializes these fields.

The void add(String threadName, String name) method is invoked by the writer thread to add a new name. The threadName argument is used to identify the writer thread (perhaps we might want to add more writer threads) and the name argument identifies the name to be added to the list.

This method first executes writeLock.lock(); to acquire the write lock and then outputs the number of threads waiting to acquire the read (0 to 2) or write (1) lock. After adding the name to the list, it executes writeLock.unlock(); to release the write lock.

The void dump(String threadName) method is similar to add()except for iterating over the list of names, outputting each name, and sleeping for a random amount of time.

Execute javac RWLockDemo.java to compile Listing 3. Then execute java RWLockDemo to run the application. On a Windows 7 platform, I observe something like the following output:

```
writer: num waiting threads = 0
reader1: num waiting threads = 1
reader2: num waiting threads = 0
reader2: A0
reader1: A0
reader2: num waiting threads = 0
reader2: A0
writer: num waiting threads = 1
reader2: num waiting threads = 1
reader2: A0
reader1: num waiting threads = 0
reader1: A0
reader2: A1
```

```
reader1: A1
writer: num waiting threads = 2
reader2: num waiting threads = 1
reader2: A0
reader1: num waiting threads = 0
reader1: A0
reader2: A1
reader1: A1
reader1: A2
reader2: A2
writer: num waiting threads = 2
reader2: num waiting threads = 1
reader2: A0
reader1: num waiting threads = 0
reader1: A0
reader2: A1
reader1: A1
reader2: A2
reader1: A2
reader2: A3
reader1: A3
writer: num waiting threads = 0
reader1: num waiting threads = 0
reader1: A0
reader1: A1
reader1: A2
reader1: A3
reader1: A4
```

The interleaved output for the reader threads demonstrates that the read lock may be held simultaneously by multiple reader threads.

Atomic variables

Multithreaded applications that run on multicore processors or multiprocessor systems can achieve good hardware utilization and be highly scalable. They can achieve these ends by having their threads spend most of their time performing work rather than waiting for work to accomplish, or waiting to acquire locks in order to access shared data structures.

However, Java's traditional synchronization mechanism, which enforces *mutual exclusion* (the thread holding the lock that guards a set of variables has exclusive access to them) and*visibility* (changes to the guarded variables become visible to other threads that subsequently acquire the lock), impacts hardware utilization and scalability, as follows:

- *Contended synchronization* (multiple threads constantly competing for a lock) is expensive and throughput suffers as a result. A major reason for the expense is the frequent context switching that takes place; a context switch operation can take many processor cycles to complete. In contrast, *uncontended synchronization* is inexpensive on modern JVMs.
- When a thread holding a lock is delayed (e.g., because of a scheduling delay), no thread that requires that lock makes any progress, and the hardware isn't utilized as well as it otherwise might be.

You might think that you can use volatile as a synchronization alternative. However, volatile variables only solve the visibility problem. They cannot be used to safely implement the atomic read-modify-write sequences that are necessary for safely implementing counters and other entities that require mutual exclusion.

Java 5 introduced a synchronization alternative that offers mutual exclusion combined with the performance of volatile. This *atomic variable* alternative is based on a microprocessor's compare-and-swap instruction and largely consists of the types in the java.util.concurrent.atomic package.

Understanding compare-and-swap

The *compare-and-swap (CAS)* instruction is an uninterruptible instruction that reads a memory location, compares the read value with an expected value, and stores a new value in the memory location when the read value matches the expected value. Otherwise, nothing is done. The actual microprocessor instruction may differ somewhat (e.g., return true if CAS succeeded or false otherwise instead of the read value).

Microprocessor CAS instructions

Modern microprocessors offer some kind of CAS instruction. For example, Intel microprocessors offer the cmpxchg family of instructions, whereas PowerPC

microprocessors offer load-link (e.g., lwarx) and store-conditional (e.g., stwcx) instructions for the same purpose.

CAS makes it possible to support atomic read-modify-write sequences. You would typically use CAS as follows:

1. Read value v from address X.
2. Perform a multistep computation to derive a new value v2.
3. Use CAS to change the value of X from v to v2. CAS succeeds when X's value hasn't changed while performing these steps.

To see how CAS offers better performance (and scalability) over synchronization, consider a counter example that lets you read its current value and increment the counter. The following class implements a counter based on synchronized:

Listing 4. Counter.java (version 1)

```
public class Counter
{
  private int value;

  public synchronized int getValue()
  {
    return value;
  }

  public synchronized int increment()
  {
    return ++value;
  }
}
```

High contention for the monitor lock will result in excessive context switching that can delay all of the threads and result in an application that doesn't scale well.

The CAS alternative requires an implementation of the compare-and-swap instruction. The following class emulates CAS. It uses synchronized instead of the actual hardware instruction to simplify the code:

Listing 5. EmulatedCAS.java

```
public class EmulatedCAS
{
  private int value;

  public synchronized int getValue()
  {
    return value;
  }

  public synchronized int compareAndSwap(int expectedValue, int
newValue)
  {
    int readValue = value;
    if (readValue == expectedValue)
      value = newValue;
    return readValue;
  }
}
```

Here, value identifies a memory location, which can be retrieved by getValue(). Also, compareAndSwap() implements the CAS algorithm.

The following class uses EmulatedCAS to implement a non-synchronized counter (pretend that EmulatedCAS doesn't require synchronized):

Listing 6. Counter.java (version 2)

```
public class Counter
{
  private EmulatedCAS value = new EmulatedCAS();

  public int getValue()
  {
    return value.getValue();
```

```
  }

  public int increment()
  {
    int readValue = value.getValue();
    while (value.compareAndSwap(readValue, readValue+1) != readValue)
      readValue = value.getValue();
    return readValue+1;
  }
}
```

Counter encapsulates an EmulatedCAS instance and declares methods for retrieving and incrementing a counter value with help from this instance. getValue() retrieves the instance's "current counter value" and increment() safely increments the counter value.

increment() repeatedly invokes compareAndSwap() untilreadValue's value doesn't change. It's then free to change this value. When no lock is involved, contention is avoided along with excessive context switching. Performance improves and the code is more scalable.

ReentrantLock and CAS

You previously learned that ReentrantLock offers better performance than synchronized under high thread contention. To boost performance, ReentrantLock's synchronization is managed by a subclass of the abstractjava.util.concurrent.locks.AbstractQueuedSynchronizerclass. In turn, this class leverages the undocumentedsun.misc.Unsafe class and its compareAndSwapInt() CAS method.

Exploring the atomic variables package

You don't have to implement compareAndSwap() via the nonportable Java Native Interface. Instead, Java 5 offers this support via java.util.concurrent.atomic: a toolkit of classes used for lock-free, thread-safe programming on single variables.

According to java.util.concurrent.atomic's Javadoc, these classes

extend the notion of volatile values, fields, and array elements to those that also provide an atomic conditional update operation of the form boolean

compareAndSet(expectedValue, updateValue). This method (which varies in argument types across different classes) atomically sets a variable to the updateValue if it currently holds the expectedValue, reporting true on success.

This package offers classes for Boolean (AtomicBoolean), integer (AtomicInteger), long integer (AtomicLong) and reference (AtomicReference) types. It also offers array versions of integer, long integer, and reference (AtomicIntegerArray, AtomicLongArray, andAtomicReferenceArray), markable and stamped reference classes for atomically updating a pair of values (AtomicMarkableReference and AtomicStampedReference), and more.

Implementing compareAndSet()

Java implements compareAndSet() via the fastest available native construct (e.g., cmpxchg or load-link/store-conditional) or (in the worst case) *spin locks*.

Consider AtomicInteger, which lets you update an int value atomically. We can use this class to implement the counter shown in Listing 6. Listing 7 presents the equivalent source code.

Listing 7. Counter.java (version 3)

```
import java.util.concurrent.atomic.AtomicInteger;

public class Counter
{
  private AtomicInteger value = new AtomicInteger();

  public int getValue()
  {
    return value.get();
  }

  public int increment()
  {
    int readValue = value.get();
    while (!value.compareAndSet(readValue, readValue+1))
      readValue = value.get();
    return readValue+1;
```

```
    }
}
```

Listing 7 is very similar to Listing 6 except that it replacesEmulatedCAS with AtomicInteger. Incidentally, you can simplifyincrement() because AtomicInteger supplies its own int getAndIncrement() method (and similar methods).

Fork/Join framework

Computer hardware has evolved significantly since Java's debut in 1995. Back in the day, single-processor systems dominated the computing landscape and Java's synchronization primitives, such as synchronized andvolatile, as well as its threading library (the Thread class, for example) were generally adequate.

Multiprocessor systems became cheaper and developers found themselves needing to create Java applications that effectively exploited the hardware parallelism that these systems offered. However, they soon discovered that Java's low-level threading primitives and library were very difficult to use in this context, and the resulting solutions were often riddled with errors.

What is parallelism?

Parallelism is the simultaneous execution of multiple threads/tasks via some combination of multiple processors and processor cores.

The Java Concurrency Utilities framework simplifies the development of these applications; however, the utilities offered by this framework do not scale to thousands of processors or processor cores. In our many-core era, we need a solution for achieving a finer-grained parallelism, or we risk keeping processors idle even when there is lots of work for them to handle.

Professor Doug Lea presented a solution to this problem in his paper introducing the idea for a Java-based fork/join framework. Lea describes a framework that supports "a style of parallel programming in which problems are solved by (recursively) splitting them into subtasks that are solved in parallel." The Fork/Join framework was eventually included in Java 7.

Overview of the Fork/Join framework

The Fork/Join framework is based on a special executor service for running a special kind of task. It consists of the following types that are located in thejava.util.concurrent package:

- ForkJoinPool: an ExecutorService implementation that runs ForkJoinTasks.ForkJoinPool provides task-submission methods, such as void

execute(ForkJoinTask<?> task), along with management and monitoring methods, such as int getParallelism() and long getStealCount().

- ForkJoinTask: an abstract base class for tasks that run within a ForkJoinPoolcontext. ForkJoinTask describes thread-like entities that have a much lighter weight than normal threads. Many tasks and subtasks can be hosted by very few actual threads in a ForkJoinPool instance.
- ForkJoinWorkerThread: a class that describes a thread managed by a ForkJoinPoolinstance. ForkJoinWorkerThread is responsible for executing ForkJoinTasks.
- RecursiveAction: an abstract class that describes a recursive resultlessForkJoinTask.
- RecursiveTask: an abstract class that describes a recursive result-bearingForkJoinTask.

The ForkJoinPool executor service is the entry-point for submitting tasks that are typically described by subclasses ofRecursiveAction or RecursiveTask. Behind the scenes, the task is divided into smaller tasks that are *forked* (distributed among different threads for execution) from the pool. A task waits until *joined* (its subtasks finish so that results can be combined).

ForkJoinPool manages a pool of worker threads, where each worker thread has its own double-ended work queue (deque). When a task forks a new subtask, the thread pushes the subtask onto the head of its deque. When a task tries to join with another task that hasn't finished, the thread pops another task off the head of its deque and executes the task. If the thread's deque is empty, it tries to steal another task from the tail of another thread's deque. This *work stealing* behavior maximizes throughput while minimizing contention.

Using the Fork/Join framework

Fork/Join was designed to efficiently execute *divide-and-conquer algorithms*, which recursively divide problems into sub-problems until they are simple enough to solve directly; for example, a merge sort. The solutions to these sub-problems are combined to provide a solution to the original problem. Each sub-problem can be executed independently on a different processor or core.

Lea's paper presents the following pseudocode to describe the divide-and-conquer behavior:

```
Result solve(Problem problem)
{
  if (problem is small)
    directly solve problem
  else
  {
```

```
    split problem into independent parts
    fork new subtasks to solve each part
    join all subtasks
    compose result from subresults
  }
}
```

The pseudocode presents a solve method that's called with some problem to solve and which returns a Result that contains the problem's solution. If the problem is too small to solve via parallelism, it's solved directly. (The overhead of using parallelism on a small problem exceeds any gained benefit.) Otherwise, the problem is divided into subtasks: each subtask independently focuses on part of the problem.

Operation fork launches a new fork/join subtask that will execute in parallel with other subtasks. Operation join delays the current task until the forked subtask finishes. At some point, the problem will be small enough to be executed sequentially, and its result will be combined along with other subresults to achieve an overall solution that's returned to the caller.

The Javadoc for the RecursiveAction and RecursiveTaskclasses presents several divide-and-conquer algorithm examples implemented as fork/join tasks. For RecursiveActionthe examples sort an array of long integers, increment each element in an array, and sum the squares of each element in an array of doubles. RecursiveTask's solitary example computes a Fibonacci number.

Listing 8 presents an application that demonstrates the sorting example in non-fork/join as well as fork/join contexts. It also presents some timing information to contrast the sorting speeds.

Listing 8. ForkJoinDemo.java

```
import java.util.Arrays;

import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;

public class ForkJoinDemo
{
  static class SortTask extends RecursiveAction
  {
```

```
    private final long[] array;
    private final int lo, hi;

    SortTask(long[] array, int lo, int hi)
    {
      this.array = array;
      this.lo = lo;
      this.hi = hi;
    }

    SortTask(long[] array)
    {
      this(array, 0, array.length);
    }

    private final static int THRESHOLD = 1000;

    @Override
    protected void compute()
    {
      if (hi-lo < THRESHOLD)
        sortSequentially(lo, hi);
      else
      {
        int mid = (lo+hi) >>> 1;
        invokeAll(new SortTask(array, lo, mid),
                  new SortTask(array, mid, hi));
        merge(lo, mid, hi);
      }
    }

    private void sortSequentially(int lo, int hi)
    {
```

```
      Arrays.sort(array, lo, hi);
    }

    private void merge(int lo, int mid, int hi)
    {
      long[] buf = Arrays.copyOfRange(array, lo, mid);
      for (int i = 0, j = lo, k = mid; i < buf.length; j++)
        array[j] = (k == hi || buf[i] < array[k]) ? buf[i++] : array[k++];
    }
  }

  public static void main(String[] args)
  {
    long[] array = new long[300000];
    for (int i = 0; i < array.length; i++)
      array[i] = (long) (Math.random()*10000000);
    long[] array2 = new long[array.length];
    System.arraycopy(array, 0, array2, 0, array.length);

    long startTime = System.currentTimeMillis();
    Arrays.sort(array, 0, array.length-1);
    System.out.printf("sequential sort completed in %d millis%n",
                      System.currentTimeMillis()-startTime);
    for (int i = 0; i < array.length; i++)
      System.out.println(array[i]);

    System.out.println();

    ForkJoinPool pool = new ForkJoinPool();
    startTime = System.currentTimeMillis();
    pool.invoke(new SortTask(array2));
    System.out.printf("parallel sort completed in %d millis%n",
                      System.currentTimeMillis()-startTime);
```

```
    for (int i = 0; i < array2.length; i++)
      System.out.println(array2[i]);
  }
}
```

Listing 8's ForkJoinDemo class declares a SortedTask nested class that describes a resultless fork/join task for sorting a long integer array. The key to this class is the overriding protected void compute() method, which is called by a worker thread to sort part of the array.

The mechanics of ForkJoinDemo

compute() first determines whether it should sort the array sequentially or divide the array among a pair of subtasks and invoke them recursively. As long as the difference between the lo and hi indexes is greater than or equal to THRESHOLD's value, the array is subdivided.

When a problem is very small, a sequential solution is often faster than a parallel solution because there's less overhead. The ideal threshold depends on the cost of coordinating tasks that run in parallel. The lower this cost, the lower the threshold value can be.

RecursiveAction inherits ForkJoinTask's void invokeAll(ForkJoinTask<?>... tasks)method for forking a variable number of tasks. It's passed two SortTask instances representing the low and high halves of the array to sort.

The invokeAll() call is followed by a call to merge(), which combines the results from the previously executed SortTask instances. This completes the behavior of this fork/join task.

Listing 8 also presents a main() method for running this application. This method first creates two large long integer arrays with identical content. It then sorts the first array sequentially, and sorts the second array via ForkJoinPool.

Compile Listing 8 (javac ForkJoinDemo.java) and run the application (java ForkJoinDemo). The following is a subset of the output that I observed during one run on my platform, a dual-core 64-bit AMD processor:

```
sequential sort completed in 94 millis
172
214
287
296
342
343
...

parallel sort completed in 78 millis
172
214
287
296
342
343
...
```

This output indicates that the parallel sorting is faster than sequential sorting.

Concurrency in Java 8

Java 8 is expected to reach general availability status in March 2014. Although this release will likely be celebrated (and remembered) for introducing Lambda expressions to the Java language, it includes other new features that will help to improve developer productivity. Consider the following enhancements to the Java Concurrency Utilities:

- **Improved ConcurrentHashMap class**: The`java.util.concurrent.ConcurrentHashMap` class has been improved to make it and classes that use it more useful as caches. New methods include sequential and parallel bulk operations (`forEach`, `search`, and `reduce`) methods.
- **Fence intrinsics**: This update exposes memory fence controls into Java code so that the Java Concurrency Utilities APIs can more accurately and efficiently control

memory ordering and bounding. This task is accomplished by adding "three memory-ordering intrinsics to the sun.misc.Unsafe class."

- **Changes to the Fork/Join framework**: The ForkJoinPool and ForkJoinTask classes have been updated to improve performance and supply additional functionality. "New features include support for completion-based designs that are often most appropriate for IO-bound usages, among others." Additionally, a newCountedCompleter class that subclasses ForkJoinTask and provides a completion action that's "performed when triggered and there are no remaining pending actions" has been introduced.
- **New CompletableFuture class**: The new java.util.concurrent.CompletableFutureclass is a "Future that may be explicitly completed (setting its value and status), and may include dependent functions and actions that trigger upon its completion." This class is associated with the newjava.util.concurrent.CompletableFuture.AsynchronousCompletionTask interface and the new java.util.concurrent.CompletionException exception. Check out Tomasz Nurkiewicz's Java 8: Definitive guide to CompletableFuture blog post for an extensive tutorial on how to use CompletableFuture.
- **New StampedLock class**: The new java.util.concurrent.locks.StampedLock class is "a capability-based lock with three modes for controlling read/write access." Check out Dr. Heinz Kabutz's Phaser and StampedLock Concurrency Synchronizers video presentation to learn about StampedLock. A PDF file of this presentation is also available.
- **Parallel array sorting**: The java.util.Arrays class has been augmented with severalparallel-prefixed class methods (such as void parallelSort(int[] a)) that leverage the Fork/Join framework to sort arrays in parallel.
- **Scalable updatable variables**: The java.util.concurrent.atomic package includes new DoubleAccumulator, DoubleAdder, LongAccumulator, and LongAdder classes that address a scalability problem in the context of maintaining a single count, sum, or some other value with the possibility of updates from many threads. These new classes "internally employ contention-reduction techniques that provide huge throughput improvements as compared to atomic variables. This is

made possible by relaxing atomicity guarantees in a way that is acceptable in most applications."

In conclusion

The Java Concurrency Utilities framework offers an alternative to Java's low-level threading capabilities. This article completes my two-part series on `java.util.concurrent` by focusing on the Locking framework, atomic variables, and the Fork/Join framework. I also introduced seven significant enhancements to the Java Concurrency Utilities, which are coming in Java 8.

The two articles in this series couldn't cover every API in the Java Concurrency Utilities, but there is more to explore. The code file for this article includes more demos, including exercises using the `java.util.concurrent.ThreadLocalRandom` and `java.util.concurrent.locks.LockSupport` classes, and more.

Jeff Friesen is a freelance tutor and software developer with an emphasis on Java and Android. In addition to writing Java and Android books for Apress, Jeff has written numerous articles on Java and other technologies for JavaWorld, informIT, Java.net, DevSource, and SitePoint. Jeff can be contacted via his website at TutorTutor.ca.

Learn more about this topic

Download the source code for this article.