**Concurrency Utility**

# Executor

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 (with doWork(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 executing java.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 Executor implementation. The runnable task is thus able to execute via a new thread, a pooled thread, the calling thread, and so on.

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 most nThreads 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 the ScheduledExecutorService 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.

# ExecutorService

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 an ExecutorService 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.

**Example:**

import java.util.concurrent.ExecutorService**;**

import java.util.concurrent.Executors**;**

import java.util.concurrent.TimeUnit**;**

**class** **Processor** **implements** Runnable **{**

**private** **int** id**;**

**public** Processor**(int** id**)** **{**

**this.**id **=** id**;**

**}**

**public** **void** run**()** **{**

System**.**out**.**println**(**"Starting: " **+** id**);**

**try** **{**

Thread**.**sleep**(**5000**);**

**}** **catch** **(**InterruptedException e**)** **{**

**}**

System**.**out**.**println**(**"Completed: " **+** id**);**

**}**

**}**

**public** **class** **App** **{**

**public** **static** **void** main**(**String**[]** args**)** **{**

ExecutorService executor **=** Executors**.**newFixedThreadPool**(**2**);**

**for(int** i**=**0**;** i**<**5**;** i**++)** **{**

executor**.**submit**(new** Processor**(**i**));**

**}**

executor**.**shutdown**();**

System**.**out**.**println**(**"All tasks submitted."**);**

**try** **{**

executor**.**awaitTermination**(**1**,** TimeUnit**.**DAYS**);**

**}** **catch** **(**InterruptedException e**)** **{**

**}**

System**.**out**.**println**(**"All tasks completed."**);**

**}**

**All tasks submitted.**

**Starting: 1**

**Starting: 0**

**Completed: 1**

**Starting: 2**

**Completed: 0**

**Starting: 3**

**Completed: 2**

**Starting: 4**

**Completed: 3**

**Completed: 4**

**All tasks completed.**

# Future & Callable

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>'s V call() throws Exception, method can return a value and throw an exception.

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;

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.

# Semaphores:

The synchronization object that many readers will immediately recognize is Semaphore, which implements a classic semaphore. A semaphore controls access to a shared resource through the use of a counter. If the counter is greater than zero, then access is allowed. If it is zero, then access is denied. What the counter is counting are permits that allow access to the shared resource. Thus, to access the resource, a thread must be granted a permit from the semaphore.

In general, to use a semaphore, the thread that wants access to the shared resource tries to acquire a permit. If the semaphore’s count is greater than zero, then the thread acquires a permit, which causes the semaphore’s count to be decremented. Otherwise, the thread will be blocked until a permit can be acquired. When the thread no longer needs access to the shared resource, it releases the permit, which causes the semaphore’s count to be incremented.

If there is another thread waiting for a permit, then that thread will acquire a permit at that time. Java’s Semaphore class implements this mechanism.

Semaphore has the two constructors shown here:

-Semaphore(int num)

-Semaphore(int num, boolean how)

Here, num specifies the initial permit count. Thus, num specifies the number of threads that can access a shared resource at any one time. If num is one, then only one thread can access the resource at any one time. By default, waiting threads are granted a permit in an undefined order. By setting how to true, you can ensure that waiting threads are granted a permit in the order in which they requested access.

To acquire a permit, call the acquire( ) method, which has these two forms:

void acquire( ) throws InterruptedException

void acquire(int num) throws InterruptedException

The first form acquires one permit. The second form acquires num permits. Most often, the first form is used. If the permit cannot be granted at the time of the call, then the invoking thread suspends until the permit is available.

To release a permit, call release( ), which has these two forms:

void release( )

void release(int num)

The first form releases one permit. The second form releases the number of permits specified by num.

To use a semaphore to control access to a resource, each thread that wants to use that

resource must first call acquire( ) before accessing the resource. When the thread is done with

the resource, it must call release( ). Here is an example that illustrates the use of a semaphore:

// A simple semaphore example.

import java.util.concurrent.\*;

class SemDemo {

public static void main(String args[]) {

Semaphore sem = new Semaphore(1);

new IncThread(sem, "A");

new DecThread(sem, "B");

}

}

// A shared resource.

class Shared {

static int count = 0;

}

// A thread of execution that increments count.

class IncThread implements Runnable {

String name;

Semaphore sem;

IncThread(Semaphore s, String n) {

sem = s;

name = n;

new Thread(this).start();

}

public void run() {

System.out.println("Starting " + name);

try {

// First, get a permit.

System.out.println(name + " is waiting for a permit.");

sem.acquire();

System.out.println(name + " gets a permit.");

// Now, access shared resource.

for(int i=0; i < 5; i++) {

Shared.count++;

System.out.println(name + ": " + Shared.count);

// Now, allow a context switch -- if possible.

Thread.sleep(10);

}

} catch (InterruptedException exc) {

System.out.println(exc);

}

// Release the permit.

System.out.println(name + " releases the permit.");

sem.release();

}

}

// A thread of execution that decrements count.

class DecThread implements Runnable {

String name;

Semaphore sem;

DecThread(Semaphore s, String n) {

sem = s;

name = n;

new Thread(this).start();

}

public void run() {

System.out.println("Starting " + name);

try {

// First, get a permit.

System.out.println(name + " is waiting for a permit.");

sem.acquire();

System.out.println(name + " gets a permit.");

// Now, access shared resource.

for(int i=0; i < 5; i++) {

Shared.count--;

System.out.println(name + ": " + Shared.count);

// Now, allow a context switch -- if possible.

Thread.sleep(10);

}

} catch (InterruptedException exc) {

System.out.println(exc);

}

// Release the permit.

System.out.println(name + " releases the permit.");

sem.release();

}

}

The output from the program is shown here. (The precise order in which the threads

execute may vary.)

Starting A

A is waiting for a permit.

A gets a permit.

A: 1

Starting B

B is waiting for a permit.

A: 2

A: 3

A: 4

A: 5

A releases the permit.

B gets a permit.

B: 4

B: 3

B: 2

B: 1

B: 0

B releases the permit.

The program uses a semaphore to control access to the count variable, which is a static variable within the Shared class. Shared.count is incremented five times by the run( ) method of IncThread and decremented five times by DecThread. To prevent these two threads from accessing Shared.count at the same time, access is allowed only after a permit is acquired from the controlling semaphore. After access is complete, the permit is released. In this way, only one thread at a time will access Shared.count, as the output shows.

In both IncThread and DecThread, notice the call to sleep( ) within run( ). It is used to “prove” that accesses to Shared.count are synchronized by the semaphore. In run( ), the call to sleep( ) causes the invoking thread to pause between each access to Shared.count. This would normally enable the second thread to run. However, because of the semaphore, the second thread must wait until the first has released the permit, which happens only after all

accesses by the first thread are complete. Thus, Shared.count is first incremented five times by IncThread and then decremented five times by DecThread. The increments and decrements are not intermixed.

Without the use of the semaphore, accesses to Shared.count by both threads would have

occurred simultaneously, and the increments and decrements would be intermixed. To confirm

this, try commenting out the calls to acquire( ) and release( ). When you run the program,

you will see that access to Shared.count is no longer synchronized, and each thread accesses

it as soon as it gets a timeslice.

Although many uses of a semaphore are as straightforward as that shown in the preceding

program, more intriguing uses are also possible. Here is an example. The following program

reworks the producer/consumer program shown in Chapter 11 so that it uses two semaphores

to regulate the producer and consumer threads, ensuring that each call to put( ) is followed

by a corresponding call to get( ):

// An implementation of a producer and consumer

// that use semaphores to control synchronization.

import java.util.concurrent.Semaphore;

class Q {

int n;

// Start with consumer semaphore unavailable.

static Semaphore semCon = new Semaphore(0);

static Semaphore semProd = new Semaphore(1);

void get() {

try {

semCon.acquire();

} catch(InterruptedException e) {

System.out.println("InterruptedException caught");

}

System.out.println("Got: " + n);

semProd.release();

}

void put(int n) {

try {

semProd.acquire();

} catch(InterruptedException e) {

System.out.println("InterruptedException caught");

}

this.n = n;

System.out.println("Put: " + n);

semCon.release();

}

}

class Producer implements Runnable {

Q q;

Producer(Q q) {

this.q = q;

new Thread(this, "Producer").start();

}

public void run() {

for(int i=0; i < 20; i++) q.put(i);

}

}

class Consumer implements Runnable {

Q q;

Consumer(Q q) {

this.q = q;

new Thread(this, "Consumer").start();

}

public void run() {

for(int i=0; i < 20; i++) q.get();

}

}

class ProdCon {

public static void main(String args[]) {

Q q = new Q();

new Consumer(q);

new Producer(q);

}

}

Aportion of the output is shown here:

Put: 0

Got: 0

Put: 1

Got: 1

Put: 2

Got: 2

Put: 3

Got: 3

Put: 4

Got: 4

Put: 5

Got: 5

.

.

.

As you can see, the calls to put( ) and get( ) are synchronized. That is, each call to put( ) is followed by a call to get( ) and no values are missed. Without the semaphores, multiple calls to put( ) would have occurred without matching calls to get( ), resulting in values being missed. (To prove this, remove the semaphore code and observe the results.)

The sequencing of put( ) and get( ) calls is handled by two semaphores: semProd and semCon. Before put( ) can produce a value, it must acquire a permit from semProd. After it has set the value, it releases semCon. Before get( ) can consume a value, it must acquire a permit from semCon. After it consumes the value, it releases semProd. This “give and take” mechanism ensures that each call to put( ) must be followed by a call to get( ).

Notice that semCon is initialized with no available permits. This ensures that put( ) executes first. The ability to set the initial synchronization state is one of the more powerful aspects of a semaphore.

# CountDown Latch:

Sometimes you will want a thread to wait until one or more events have occurred. To handle such a situation, the concurrent API supplies CountDownLatch. A CountDownLatch is initially created with a count of the number of events that must occur before the latch is released. Each time an event happens, the count is decremented. When the count reaches zero, the latch opens.

CountDownLatch has the following constructor:

CountDownLatch(int num)

Here, num specifies the number of events that must occur in order for the latch to open.

To wait on the latch, a thread calls await( ), which has the forms shown here:

void await( ) throws InterruptedException

void await(long wait, TimeUnit tu) throws InterruptedException

The first form waits until the count associated with the invoking CountDownLatch reaches zero. The second form waits only for the period of time specified by wait. The units represented by wait are specified by tu, which is an object the TimeUnit enumeration. (TimeUnit is described later in this chapter.)

To signal an event, call the countDown( ) method, shown next:

void countDown( )

Each call to countDown( ) decrements the count associated with the invoking object.

The following program demonstrates CountDownLatch. It creates a latch that requires

five events to occur before it opens.

// An example of CountDownLatch.

import java.util.concurrent.CountDownLatch;

class CDLDemo {

public static void main(String args[]) {

CountDownLatch cdl = new CountDownLatch(5);

System.out.println("Starting");

new MyThread(cdl);

try {

cdl.await();

} catch (InterruptedException exc) {

System.out.println(exc);

}

System.out.println("Done");

}

}

class MyThread implements Runnable {

CountDownLatch latch;

MyThread(CountDownLatch c) {

latch = c;

new Thread(this).start();

}

public void run() {

for(int i = 0; i<5; i++) {

System.out.println(i);

latch.countDown(); // decrement count

}

}

}

The output produced by the program is shown here:

Starting

0

1

2

3

4

Done

Inside main( ), a CountDownLatch called cdl is created with an initial count of five.

Next, an instance of MyThread is created, which begins execution of a new thread. Notice

that cdl is passed as a parameter to MyThread’s constructor and stored in the latch instance

variable. Then, the main thread calls await( ) on cdl, which causes execution of the main thread

to pause until cdl’s count has been decremented five times.

Inside the run( ) method of MyThread, a loop is created that iterates five times. With

each iteration, the countDown( ) method is called on latch, which refers to cdl in main( ).

After the fifth iteration, the latch opens, which allows the main thread to resume.

CountDownLatch is a powerful yet easy-to-use synchronization object that is appropriate

whenever a thread must wait for one or more events to occur.

# Cyclic Barrier ???

# Exchanger

Perhaps the most interesting of the synchronization classes is Exchanger. It is designed to simplify the exchange of data between two threads. The operation of an Exchanger is astoundingly simple: it simply waits until two separate threads call its exchange( ) method.

When that occurs, it exchanges the data supplied by the threads. This mechanism is both elegant and easy to use. Uses for Exchanger are easy to imagine. For example, one thread might prepare a buffer for receiving information over a network connection. Another thread might fill that buffer with the information from the connection. The two threads work together so that each time a new buffer is needed, an exchange is made.

Exchanger is a generic class that is declared as shown here:

Exchanger<V>

Here, V specifies the type of the data being exchanged.

The only method defined by Exchanger is exchange( ), which has the two forms

shown here:

V exchange(V buffer) throws InterruptedException

V exchange(V buffer, long wait, TimeUnit tu)

throws InterruptedException, TimeoutException

Here, buffer is a reference to the data to exchange. The data received from the other thread is returned. The second form of exchange( ) allows a time-out period to be specified. The key point about exchange( ) is that it won’t succeed until it has been called on the same Exchanger object by two separate threads. Thus, exchange( ) synchronizes the exchange of the data.

Here is an example that demonstrates Exchanger. It creates two threads. One thread

creates an empty buffer that will receive the data put into it by the second thread. Thus,

the first thread exchanges an empty thread for a full one.

// An example of Exchanger.

import java.util.concurrent.Exchanger;

class ExgrDemo {

public static void main(String args[]) {

Exchanger<String> exgr = new Exchanger<String>();

new UseString(exgr);

new MakeString(exgr);

}

}

// A Thread that constructs a string.

class MakeString implements Runnable {

Exchanger<String> ex;

String str;

MakeString(Exchanger<String> c) {

ex = c;

str = new String();

new Thread(this).start();

}

public void run() {

char ch = 'A';

for(int i = 0; i < 3; i++) {

// Fill Buffer

for(int j = 0; j < 5; j++)

str += ch++;

try {

// Exchange a full buffer for an empty one.

str = ex.exchange(str);

} catch(InterruptedException exc) {

System.out.println(exc);

}

}

}

}

// A Thread that uses a string.

class UseString implements Runnable {

Exchanger<String> ex;

String str;

UseString(Exchanger<String> c) {

ex = c;

new Thread(this).start();

}

public void run() {

for(int i=0; i < 3; i++) {

try {

// Exchange an empty buffer for a full one.

str = ex.exchange(new String());

System.out.println("Got: " + str);

} catch(InterruptedException exc) {

System.out.println(exc);

}

}

}

}

Here is the output produced by the program:

Got: ABCDE

Got: FGHIJ

Got: KLMNO

In the program, the main( ) method creates an Exchanger for strings. This object is then used to synchronize the exchange of strings between the MakeString and UseString classes. The MakeString class fills a string with data. The UseString exchanges an empty buffer for a full one. It then displays the contents of the newly constructed string. The exchange of empty and full buffers is synchronized by the exchange( ) method, which is called by both class’ run( ) method.

# Locks

The java.util.concurrent.locks package provides support for locks, which are objects that offer an alternative to using synchronized to control access to a shared resource. In general, here is how a lock works. Before accessing a shared resource, the lock that protects that resource is acquired. When access to the resource is complete, the lock is released. If a second thread attempts to acquire the lock when it is in use by another thread, the second thread

will suspend until the lock is released. In this way, conflicting access to a shared resource is prevented.

Locks are particularly useful when multiple threads need to access the value of shared data. For example, an inventory application might have a thread that first confirms that an item is in stock and then decreases the number of items on hand as each sale occurs. If two or more of these threads are running, then without some form of synchronization, it would be possible for one thread to be in middle of a transaction when the second thread begins its transaction. The result could be that both threads would assume that adequate inventory exists, even if there is only sufficient inventory on hand to satisfy one sale. In this type of situation, a lock offers a convenient means of handling the needed synchronization.

All locks implement the Lock interface. In general, to acquire a lock, call lock( ). If the lock is unavailable, lock( ) will wait. To release a lock, call unlock( ). To see if a lock is available, and to acquire it if it is, call tryLock( ). This method will not wait for the lock if it is unavailable. Instead, it returns true if the lock is acquired and false otherwise. The newCondition( ) method returns a Condition object associated with the lock. Using a Condition, you gain detailed control of the lock through methods such as await( ) and signal( ), which provide functionality similar to Object.wait( ) and Object.notify( ).

java.util.concurrent.locks supplies an implementation of Lock called ReentrantLock. ReentrantLock implements a reentrant lock, which is a lock that can be repeatedly entered by the thread that currently holds the lock. (Of course, in the case of a thread reentering a lock, all calls to lock( ) must be offset by an equal number of calls to unlock( ).) Otherwise, a thread seeking to acquire the lock will suspend until the lock is not in use. The following program demonstrates the use of a lock. It creates two threads that access a shared resource called Shared.count. Before a thread can access Shared.count, it must obtain

a lock. After obtaining the lock, Shared.count is incremented and then, before releasing the lock, the thread sleeps. This causes the second thread to attempt to obtain the lock. However, because the lock is still held by the first thread, the second thread must wait until the first thread stops sleeping and releases the lock. The output shows that access to Shared.count is, indeed, synchronized by the lock.

// A simple lock example.

import java.util.concurrent.locks.\*;

class LockDemo {

public static void main(String args[]) {

ReentrantLock lock = new ReentrantLock();

new LockThread(lock, "A");

new LockThread(lock, "B");

}

}

// A shared resource.

class Shared {

static int count = 0;

}

// A thread of execution that increments count.

class LockThread implements Runnable {

String name;

ReentrantLock lock;

LockThread(ReentrantLock lk, String n) {

lock = lk;

name = n;

new Thread(this).start();

}

public void run() {

System.out.println("Starting " + name);

try {

// First, lock count.

System.out.println(name + " is waiting to lock count.");

lock.lock();

System.out.println(name + " is locking count.");

Shared.count++;

System.out.println(name + ": " + Shared.count);

// Now, allow a context switch -- if possible.

System.out.println(name + " is sleeping.");

Thread.sleep(1000);

} catch (InterruptedException exc) {

System.out.println(exc);

} finally {

// Unlock

System.out.println(name + " is unlocking count.");

lock.unlock();

}

}

}

The output is shown here. (The precise order in which the threads execute may vary.)

Starting A

A is waiting to lock count.

A is locking count.

A: 1

A is sleeping.

Starting B

B is waiting to lock count.

A is unlocking count.

B is locking count.

B: 2

B is sleeping.

B is unlocking count.

java.util.concurrent.locks also defines the ReadWriteLock interface. This interface specifies a reentrant lock that maintains separate locks for read and write access. This enables multiple locks to be granted for readers of a resource as long as the resource is not being written. ReentrantReadWriteLock provides an implementation of ReadWriteLock.

# concurrent collections

Several new Collection classes are added in Java 5 and Java 6 specially concurrent alternatives of standard synchronized ArrayList, Hashtable and synchronized HashMap collection classes. Many Java programmer still not familiar with these new collection classes from java.util.concurrent package and misses a whole new set of functionality which can be utilized to build more scalable and high performance Java application. In this Java tutorial we will some of useful collection classes e.g. ConcurrentHashMap, BlockingQueue which provides some of the very useful functionalities to build concurrent Java application. By the way this is not a comprehensive article explaining each feature of all these concurrent collections, Instead I will just try to list out why they are there, which Collection class they replace or provides alternative for. Idea is to keep it short and simple while highlighting key points of those useful java.util.concurrent collections.

## 1. ConcurrentHashMap

Java Concurrent Collections from JDK 5 and 6 Example TutorialConcurrentHashMap is undoubtedly most popular collection class introduced in Java 5 and most of us are already using it. ConcurrentHashMap provides a concurrent alternative of Hashtable or Synchronized Map classes with aim to support higher level of concurrency by implementing fined grained locking. Multiple reader can access the Map concurrently while a portion of Map gets locked for write operation depends upon concurrency level of Map. ConcurrentHashMap provides better scalability than there synchronized counter part. Iterator of ConcurrentHashMap are fail-safe iterators which doesn't throw ConcurrencModificationException thus eliminates another requirement of locking during iteration which result in further scalability and performance.

## 2. CopyOnWriteArrayList and CopyOnWriteArraySet

CopyOnWriteArrayList is a concurrent alternative of synchronized List. CopyOnWriteArrayList provides better concurrency than synchronized List by allowing multiple concurrent reader and replacing the whole list on write operation. Yes, write operation is costly on CopyOnWriteArrayList but it performs better when there are multiple reader and requirement of iteration is more than writing. Since CopyOnWriteArrayList Iterator also don't throw ConcurrencModificationException it eliminates need to lock the collection during iteration. Remember both ConcurrentHashMap and CopyOnWriteArrayList doesn't provides same level of locking as Synchronized Collection and achieves thread-safety by there locking and mutability strategy. So they perform better if requirements suits there nature. Similarly, CopyOnWriteArraySet is a concurrent replacement to Synchronized Set. See What is CopyOnWriteArrayList in Java for more details

## 3. BlockingQueue

BlockingQueue is also one of better known collection class in Java 5. BlockingQueue makes it easy to implement producer-consumer design pattern by providing inbuilt blocking support for put() and take() method. put() method will block if Queue is full while take() method will block if Queue is empty. Java 5 API provides two concrete implementation of BlockingQueue in form of ArrayBlockingQueue and LinkedBlockingQueue, both of them implement FIFO ordering of element. ArrayBlockingQueue is backed by Array and its bounded in nature while LinkedBlockingQueue is optionally bounded. Consider using BlockingQueue to solve producer Consumer problem in Java instead of writing your own wait-notify code. Java 5 also provides PriorityBlockingQueue, another implementation of BlockingQueue which is ordered on priority and useful if you want to process elements on order other than FIFO.

The BlockingQueue interface states that it is a Queue, meaning that its items are stored in first in, first out (FIFO) order. Items inserted in a particular order are retrieved in that same order — but with the added guarantee that any attempt to retrieve an item from an empty queue will block the calling thread until the item is ready to be retrieved. Likewise, any attempt to insert an item into a queue that is full will block the calling thread until space becomes available in the queue's storage.

BlockingQueue neatly solves the problem of how to "hand off" items gathered by one thread to another thread for processing, without explicit concern for synchronization issues.

### **When we should use java.util.concurrent.BlockingQueue?**

* When you want to throttle some sort of incoming request then you should use the same
* A producers can get far ahead of the consumers with an unbounded queue. If consumer is not catching up with producer then it may cause an OutOfMemoryError. In situations like these, it may be better to signal a would-be producer that the queue is full, and to give up quickly with a failure. In other words: the producers are naturally throttled.
* Blocking Queue is normally used in concurrent application
* It provides a correct, thread-safe implementation
* Memory consumption should be limited as well

import java.util.Random**;**

import java.util.concurrent.ArrayBlockingQueue**;**

import java.util.concurrent.BlockingQueue**;**

**public** **class** **App** **{**

**private** **static** BlockingQueue**<**Integer**>** queue **=** **new** ArrayBlockingQueue**<**Integer**>(**10**);**

**public** **static** **void** main**(**String**[]** args**)** **throws** InterruptedException **{**

Thread t1 **=** **new** Thread**(new** Runnable**()** **{**

**public** **void** run**()** **{**

**try** **{**

producer**();**

**}** **catch** **(**InterruptedException e**)** **{**

*// TODO Auto-generated catch block*

e**.**printStackTrace**();**

**}**

**}**

**});**

Thread t2 **=** **new** Thread**(new** Runnable**()** **{**

**public** **void** run**()** **{**

**try** **{**

consumer**();**

**}** **catch** **(**InterruptedException e**)** **{**

*// TODO Auto-generated catch block*

e**.**printStackTrace**();**

**}**

**}**

**});**

t1**.**start**();**

t2**.**start**();**

t1**.**join**();**

t2**.**join**();**

**}**

**private** **static** **void** producer**()** **throws** InterruptedException **{**

Random random **=** **new** Random**();**

**while(true)** **{**

queue**.**put**(**random**.**nextInt**(**100**));**

**}**

**}**

**private** **static** **void** consumer**()** **throws** InterruptedException **{**

Random random **=** **new** Random**();**

**while(true)** **{**

Thread**.**sleep**(**100**);**

**if(**random**.**nextInt**(**10**)** **==** 0**)** **{**

Integer value **=** queue**.**take**();**

System**.**out**.**println**(**"Taken value: " **+** value **+** "; Queue size is: " **+** queue**.**size**());**

**}**

**}**

**}**

**}**

**Taken value: 59; Queue size is: 10**

**Taken value: 40; Queue size is: 10**

**Taken value: 74; Queue size is: 9**

**Taken value: 40; Queue size is: 9**

**Taken value: 88; Queue size is: 9**

**Taken value: 74; Queue size is: 9**

**Taken value: 96; Queue size is: 9**

**Taken value: 88; Queue size is: 10**

**Taken value: 34; Queue size is: 10**

**Taken value: 81; Queue size is: 10**

**Taken value: 44; Queue size is: 10**

**Taken value: 84; Queue size is: 10**

**Taken value: 23; Queue size is: 10**

**Taken value: 77; Queue size is: 9**

**Taken value: 33; Queue size is: 9**

**Taken value: 45; Queue size is: 10**

**Taken value: 61; Queue size is: 9**

**Taken value: 24; Queue size is: 9**

**Taken value: 9; Queue size is: 9**

**Taken value: 70; Queue size is: 9**

**Taken value: 52; Queue size is: 9**

**Taken value: 70; Queue size is: 9**

**Taken value: 69; Queue size is: 9**

## 4. Deque and BlockingDeque

Deque interface is added in Java 6 and it extends Queue interface to support insertion and removal from both end of Queue referred as head and tail. Java6 also provides concurrent implementation of Deque like ArrayDeque and LinkedBlockingDeque. Deque Can be used efficiently to increase parallelism in program by allowing set of worker thread to help each other by taking some of work load from other thread by utilizing Deque double end consumption property. So if all Thread has there own set of task Queue and they are consuming from head; helper thread can also share some work load via consumption from tail.

## 5. ConcurrentSkipListMap and ConcurrentSkipListSet

Just like ConcurrentHashMap provides a concurrent alternative of synchronized HashMap. ConcurrentSkipListMap and ConcurrentSkipListSet provide concurrent alternative for synchronized version of SortedMap and SortedSet. For example instead of using TreeMap or TreeSet wrapped inside synchronized Collection, You can consider using ConcurrentSkipListMap or ConcurrentSkipListSet from java.util.concurrent package. They also implement NavigableMap and NavigableSet to add additional navigation method we have seen in our post How to use NavigableMap in Java.

That’s all on this list of concurrent Collection classes from Java 5 and 6. They are added on java.util.concurrent package as concurrent alternative of there synchronized counterpart. It’s good idea to learn these Collection classes along with other popular classes from Java Collection Framework.

## atomic variables ???