**Java Concurrency / Multithreading**

Back in the old days a computer had a single CPU, and was only capable of executing a single program at a time. Later came multitasking which meant that computers could execute multiple programs (AKA tasks or processes) at the same time. It wasn't really "at the same time" though. The single CPU was shared between the programs. The operating system would switch between the programs running, executing each of them for a little while before switching.

Along with multitasking came new challenges for software developers. Programs can no longer assume to have all the CPU time available, nor all memory or any other computer resources. A "good citizen" program should release all resources it is no longer using, so other programs can use them.

Later yet came multithreading which mean that you could have multiple threads of execution inside the same program. A thread of execution can be thought of as a CPU executing the program. When you have multiple threads executing the same program, it is like having multiple CPU's execute within the same program.

Mulithreading is even more challenging than multitasking. The threads are executing within the same program and are hence reading and writing the same memory simultanously. This can result in errors not seen in a singlethreaded program. Some of these errors may not be seen on single CPU machines, because two threads never really execute "simultanously". Modern computers, though, come with multi core CPU's. This means that separate threads can be executed by separate cores simultanously.

If a thread reads a memory location while another thread writes to it, what value will the first thread end up reading? The old value? The value written by the second thread? Or a value that is a mix between the two? Or, if two threads are writing to the same memory location simultanously, what value will be left when they are done? The value written by the first thread? The value written by the second thread? Or a mix of the two values written? Without proper precautions any of these outcomes are possible. The behaviour would not even be predictable. The outcome could change from time to time.

The reason multithreading is still used in spite of its challenges is that multithreading can have several benefits. Some of these benefits are:

* Better resource utilization.
* Simpler program design in some situations.
* More responsive programs.

# Multithreading Costs

Going from a singlethreaded to a multithreaded application doesn't just provide benefits. It also has some costs. Don't just multithread-enable an application just because you can. You should have a good idea that the benefits gained by doing so, are larger than the costs. When in doubt, try measuring the performance or responsiveness of the application, instead of just guessing.

## More complex design

Though some parts of a multithreaded applications is simpler than a singlethreaded application, other parts are more complex. Code executed by multiple threads accessing shared data need special attention. Thread interaction is far from always simple. Errors arising from incorrect thread synchronization can be very hard to detect, reproduce and fix.

## Context Switching Overhead

When a CPU switches from executing one thread to executing another, the CPU needs to save the local data, program pointer etc. of the current thread, and load the local data, program pointer etc. of the next thread to execute. This switch is called a "context switch". The CPU switches from executing in the context of one thread to executing in the context of another.

Context switching isn't cheap. You don't want to switch between threads more than necessary.

You can read more about context switching on Wikipedia:

<http://en.wikipedia.org/wiki/Context_switch>

## Increased Resource Consumption

A thread needs some resources from the computer in order to run. Besides CPU time a thread needs some memory to keep its local stack. It may also take up some resources inside the operating system needed to manage the thread. Try creating a program that creates 100 threads that does nothing but wait, and see how much memory the application takes when running.

Java threads are objects like any other Java objects. Threads are instances of class java.lang.Thread, or instances of subclasses of this class. In addition to being objects, java threads can also execute code.

## Creating and Starting Threads

Creating a thread in Java is done like this:

Thread thread = new Thread();

To start the thread you will call its start() method, like this:

thread.start();

This example doesn't specify any code for the thread to execute. It will stop again right away.

There are two ways to specify what code the thread should execute. The first is to create a subclass of Thread and override the run() method. The second method is to pass an object that implements Runnable to the Thread constructor. Both methods are covered below.

## Thread Subclass

The first way to specify what code a thread is to run, is to create a subclass of Thread and override the run() method. The run() method is what is executed by the thread after you call start(). Here is an example:

public class MyThread extends Thread {

public void run(){

System.out.println("MyThread running");

}

}

To create and start the above thread you can do like this:

MyThread myThread = new MyThread();

myTread.start();

The start() call will return as soon as the thread is started. It will not wait until the run() method is done. The run() method will execute as if executed by a different CPU. When the run() method executes it will print out the text "MyThread running".

You can also create an anonymous subclass of Thread like this:

Thread thread = new Thread(){

public void run(){

System.out.println("Thread Running");

}

}

thread.start();

This example will print out the text "Thread running" once the run() method is executed by the new thread.

## Runnable Interface Implemention

The second way to specify what code a thread should run is by creating a class that implements java.lang.Runnable. The Runnable object can be executed by a Thread.

Here is a Java Runnable example:

public class MyRunnable implements Runnable {

public void run(){

System.out.println("MyRunnable running");

}

}

To have the run() method executed by a thread, pass an instance of MyRunnable to a Thread in its constructor. Here is how that is done:

Thread thread = new Thread(new MyRunnable());

thread.start();

When the thread is started it will call the run() method of the MyRunnable instance instead of executing it's own run() method. The above example would print out the text "MyRunnable running".

You can also create an anonymous implementation of Runnable, like this:

Runnable myRunnable = new Runnable(){

public void run(){

System.out.println("Runnable running");

}

}

Thread thread = new Thread(myRunnable);

thread.start();

## Subclass or Runnable?

There are no rules about which of the two methods that is the best. Both methods works. Personally though, I prefer implementing Runnable, and handing an instance of the implementation to a Thread instance. When having the Runnable's executed by a thread pool it is easy to queue up the Runnable instances until a thread from the pool is idle. This is a little harder to do with Thread subclasses.

Sometimes you may have to implement Runnable as well as subclass Thread. For instance, if creating a subclass of Thread that can execute more than one Runnable. This is typically the case when implementing a thread pool.

## Common Pitfall: Calling run() instead of start()

When creating and starting a thread a common mistake is to call the run() method of the Thread instead of start(), like this:

Thread newThread = new Thread(MyRunnable());

thread.run(); //should be start();

At first you may not notice anything because the Runnable's run() method is executed like you expected. However, it is NOT executed by the new thread you just created. Instead the run() method is executed by the thread that created the thread. In other words, the thread that executed the above two lines of code. To have the run() method of the MyRunnable instance called by the new created thread, newThread, you MUST call the newThread.start() method.

## Thread Names

When you create a thread you can give it a name. The name can help you distinguish different threads from each other. For instance, if multiple threads write to System.out it can be handy to see which thread wrote the text. Here is an example:

Thread thread = new Thread("New Thread") {

public void run(){

System.out.println("run by: " + getname());

} };

thread.start();

System.out.println(thread.getName());

Notice the string "New Thread" passed as parameter to the Thread constructor. This string is the name of the thread. The name can be obtained by the Thread's getName() method. You can also pass a name to a Thread when using a Runnable implementation. Here is how that looks:

MyRunnable runnable = new MyRunnable();

Thread thread = new Thread(runnable, "New Thread");

thread.start();

System.out.println(thread.getName());

Notice however, that since the MyRunnable class is not a subclass of Thread, it does not have access to the getName() method of the thread executing it. A reference to the currently executing thread can be obtained using the call

Thread.currentThread();

Getting the name of the thread currently executing the code can therefore be done like this:

String threadName = Thread.currentThread().getName();

## Java Thread Example

Here is a small example. First it prints out the name of the thread executing the main() method. This thread is assigned by the JVM. Then it starts up 10 threads and give them all a number as name ("" + i). Each thread then prints its name out, and then stops executing.

public class ThreadExample {

public static void main(String[] args){

System.out.println(Thread.currentThread().getName());

for(int i=0; i<10; i++){

new Thread("" + i){

public void run(){

System.out.println("Thread: " + getName() + " running");

}

}.start();

}

}

}

Note that even if the threads are started in sequence (1, 2, 3 etc.) they may not execute sequentially, meaning thread 1 may not be the first thread to write its name to System.out. This is because the threads are in principle executing in parallel and not sequentially. The JVM and/or operating system determines the order in which the threads are executed. This order does not have to be the same order in which they were started.

# Race Conditions and Critical Sections

Running more than one thread inside the same application does not by itself cause problems. The problems arise when multiple threads access the same resources. For instance the same memory (variables, arrays, or objects), systems (databases, web services etc.) or files. In fact, problems only arise if one or more of the threads write to these resources. It is safe to let multiple threads read the same resources, as long as the resources do not change.

Here is a code example that may fail if executed by multiple threads simultaneously:

public class Counter {

protected long count = 0;

public void add(long value){

this.count = this.count + value;

}

}

Imagine if two threads, A and B, are executing the add method on the same instance of the Counter class. There is no way to know when the operating system switches between the two threads. The code is not executed as a single instruction by the Java virtual machine. Rather it is executed along the lines of:

get this.count from memory into register

add value to register

write register to memory

Observe what happens with the following mixed execution of threads A and B:

this.count = 0;

A: reads this.count into a register (0)

B: reads this.count into a register (0)

B: adds value 2 to register

B: writes register value (2) back to memory. this.count now equals 2

A: adds value 3 to register

A: writes register value (3) back to memory. this.count now equals 3

The two threads added the values 2 and 3 to the counter. Thus the value should have been 5 after the two threads complete execution. However, since the execution of the two threads is interleaved, both threads read the value 0 from memory. Then they add their individual values, 2 and 3, to the value, and write the result back to memory. Instead of 5, the value left in this.count will be the value written by the last thread to write its value. In the above case it is thread A, but it could as well have been thread B. Without proper thread synchronization mechanisms there is no way to know exactly how the thread execution is interleaved.

## Race Conditions & Critical Sections

The situation where two threads compete for the same resource, where the sequence in which the resource is accessed is significant, is called race conditions. A code section that leads to race conditions is called a critical section. In the previous example the method add() is a critical section, leading to race conditions. Race conditions can be avoided by proper thread synchronization in critical sections.

# Thread Safety and Shared Resources

Code that is safe to call by multiple threads simultanously is called thread safe. If a piece of code is thread safe, then it contains no race conditions. Race condition only occur when multiple threads update shared resources. Therefore it is important to know what resources Java threads share when executing.

## Local Variables

Local variables are stored in each thread's own stack. That means that local variables are never shared between threads. That also means that all local primitive variables are thread safe. Here is an example of a thread safe local primitive variable:

public void someMethod(){

long threadSafeInt = 0;

threadSafeInt++;

}

## Local Object References

Local references to objects are a bit different. The reference itself is not shared. The object referenced however, is not stored in each threads's local stack. All objects are stored in the shared heap. If an object created locally never escapes the method it was created in, it is thread safe. In fact you can also pass it on to other methods and objects as long as none of these methods or objects make the passed object available to other threads. Here is an example of a thread safe local object:

public void someMethod(){

LocalObject localObject = new LocalObject();

localObject.callMethod();

method2(localObject);

}

public void method2(LocalObject localObject){

localObject.setValue("value");

}

The LocalObject instance in this example is not returned from the method, nor is it passed to any other objects that are accessible from outside the someMethod() method. Each thread executing the someMethod() method will create its own LocalObject instance and assign it to the localObject reference. Therefore the use of the LocalObject here is thread safe. In fact, the whole method someMethod() is thread safe. Even if the LocalObject instance is passed as parameter to other methods in the same class, or in other classes, the use of it is thread safe. The only exception is of course, if one of the methods called with the LocalObject as parameter, stores the LocalObject instance in a way that allows access to it from other threads.

## Object Members

Object members are stored on the heap along with the object. Therefore, if two threads call a method on the same object instance and this method updates object members, the method is not thread safe. Here is an example of a method that is not thread safe:

public class NotThreadSafe{

StringBuilder builder = new StringBuilder();

public add(String text){

this.builder.append(text); }}

If two threads call the add() method simultanously **on the same NotThreadSafe instance** then it leads to race conditions. For instance:

NotThreadSafe sharedInstance = new NotThreadSafe();

new Thread(new MyRunnable(sharedInstance)).start();

new Thread(new MyRunnable(sharedInstance)).start();

public class MyRunnable implements Runnable{

NotThreadSafe instance = null;

public MyRunnable(NotThreadSafe instance){

this.instance = instance;

}

public void run(){

this.instance.add("some text");

}}

Notice how the two MyRunnable instances share the same NotThreadSafe instance. Therefore, when they call the add() method on the NotThreadSafe instance it leads to race condition.

However, if two threads call the add() method simultanously **on different instances** then it does not lead to race condition. Here is the example from before, but slightly modified:

new Thread(new MyRunnable(new NotThreadSafe())).start();

new Thread(new MyRunnable(new NotThreadSafe())).start();

Now the two threads have each their own instance of NotThreadSafe so their calls to the add method doesn't interfere with each other. The code does not have race condition anymore. So, even if an object is not thread safe it can still be used in a way that doesn't lead to race condition.

## The Thread Control Escape Rule

When trying to determine if your code's access of a certain resource is thread safe you can use the thread control escape rule:

If a resource is created, used and disposed within

the control of the same thread,

and never escapes the control of this thread,

the use of that resource is thread safe.

Resources can be any shared resource like an object, array, file, database connection, socket etc. In Java you do not always explicitly dispose objects, so "disposed" means losing or null'ing the reference to the object.

Even if the use of an object is thread safe, if that object points to a shared resource like a file or database, your application as a whole may not be thread safe. For instance, if thread 1 and thread 2 each create their own database connections, connection 1 and connection 2, the use of each connection itself is thread safe. But the use of the database the connections point to may not be thread safe. For example, if both threads execute code like this:

check if record X exists

if not, insert record X

If two threads execute this simultanously, and the record X they are checking for happens to be the same record, there is a risk that both of the threads end up inserting it. This is how:

Thread 1 checks if record X exists. Result = no

Thread 2 checks if record X exists. Result = no

Thread 1 inserts record X

Thread 2 inserts record X

This could also happen with threads operating on files or other shared resources. Therefore it is important to distinguish between whether an object controlled by a thread **is** the resource, or if it merely **references** the resource

# Thread Safety and Immutability

[Race conditions](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html) occur only if multiple threads are accessing the same resource, **and** one or more of the threads **write**to the resource. If multiple threads read the same resource [race conditions](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html) do not occur.

We can make sure that objects shared between threads are never updated by any of the threads by making the shared objects immutable, and thereby thread safe. Here is an example:

public class ImmutableValue{

private int value = 0;

public ImmutableValue(int value){

this.value = value;

}

public int getValue(){

return this.value;

}

}

Notice how the value for the ImmutableValue instance is passed in the constructor. Notice also how there is no setter method. Once an ImmutableValue instance is created you cannot change its value. It is immutable. You can read it however, using the getValue() method.

If you need to perform operations on the ImmutableValue instance you can do so by returning a new instance with the value resulting from the operation. Here is an example of an add operation:

public class ImmutableValue{

private int value = 0;

public ImmutableValue(int value){

this.value = value;

}

public int getValue(){

return this.value;

}

**public ImmutableValue add(int valueToAdd){**

**return new ImmutableValue(this.value + valueToAdd);**

**}**

}

Notice how the add() method returns a new ImmutableValue instance with the result of the add operation, rather than adding the value to itself.

## The Reference is not Thread Safe!

It is important to remember, that even if an object is immutable and thereby thread safe, the reference to this object may not be thread safe. Look at this example:

public void Calculator{

private ImmutableValue currentValue = null;

public ImmutableValue getValue(){

return currentValue; }

public void setValue(ImmutableValue newValue){

this.currentValue = newValue;

}

public void add(int newValue){

this.currentValue = this.currentValue.add(newValue);

}

}

The Calculator class holds a reference to an ImmutableValue instance. Notice how it is possible to change that reference through both the setValue() and add() methods. Therefore, even if the Calculator class uses an immutable object internally, it is not itself immutable, and therefore not thread safe. In other words: TheImmutableValue class is thread safe, but the **use of it** is not. This is something to keep in mind when trying to achieve thread safety through immutability.

To make the Calculator class thread safe you could have declared the getValue(), setValue(), and add()methods synchronized. That would have done the trick.

# Java Synchronized Blocks

A Java synchronized block marks a method or a block of code as synchronized. Java synchronized blocks can be used to avoid [race conditions](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html).

## The Java synchronized Keyword

Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block in Java is synchronized on some object. All synchronized blocks synchronized on the same object can only have one thread executing inside them at the same time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

The synchronized keyword can be used to mark four different types of blocks:

1. Instance methods
2. Static methods
3. Code blocks inside instance methods
4. Code blocks inside static methods

These blocks are synchronized on different objects. Which type of synchronized block you need depends on the concrete situation.

## Synchronized Instance Methods

Here is a synchronized instance method:

public **synchronized** void add(int value){

this.count += value;

}

Notice the use of the synchronized keyword in the method declaration. This tells Java that the method is synchronized.

A synchronized instance method in Java is synchronized on the instance (object) owning the method. Thus, each instance has its synchronized methods synchronized on a different object: the owning instance. Only one thread can execute inside a synchronized instance method. If more than one instance exist, then one thread at a time can execute inside a synchronized instance method per instance. One thread per instance.

## Synchronized Static Methods

Static methods are marked as synchronized just like instance methods using the synchronized keyword. Here is a Java synchronized static method example:

public **static synchronized** void add(int value){

count += value;

}

Also here the synchronized keyword tells Java that the method is synchronized.

Synchronized static methods are synchronized on the class object of the class the synchronized static method belongs to. Since only one class object exists in the Java VM per class, only one thread can execute inside a static synchronized method in the same class.

If the static synchronized methods are located in different classes, then one thread can execute inside the static synchronized methods of each class. One thread per class regardless of which static synchronized method it calls.

## Synchronized Blocks in Instance Methods

You do not have to synchronize a whole method. Sometimes it is preferable to synchronize only part of a method. Java synchronized blocks inside methods makes this possible.

Here is a synchronized block of Java code inside an unsynchronized Java method:

public void add(int value){

**synchronized(this){**

this.count += value;

**}**

}

This example uses the Java synchronized block construct to mark a block of code as synchronized. This code will now execute as if it was a synchronized method.

Notice how the Java synchronized block construct takes an object in parentheses. In the example "this" is used, which is the instance the add method is called on. The object taken in the parentheses by the synchronized construct is called a monitor object. The code is said to be synchronized on the monitor object. A synchronized instance method uses the object it belongs to as monitor object.

Only one thread can execute inside a Java code block synchronized on the same monitor object.

The following two examples are both synchronized on the instance they are called on. They are therefore equivalent with respect to synchronization:

public class MyClass {

public **synchronized** void log1(String msg1, String msg2){

log.writeln(msg1);

log.writeln(msg2);

}

public void log2(String msg1, String msg2){

**synchronized(this){**

log.writeln(msg1);

log.writeln(msg2);

**}**

} }

Thus only a single thread can execute inside either of the two synchronized blocks in this example.

Had the second synchronized block been synchronized on a different object than this, then one thread at a time had been able to execute inside each method.

## Synchronized Blocks in Static Methods

Here are the same two examples as static methods. These methods are synchronized on the class object of the class the methods belong to:

public class MyClass {

public static synchronized void log1(String msg1, String msg2){

log.writeln(msg1);

log.writeln(msg2);

}

public static void log2(String msg1, String msg2){

synchronized(MyClass.class){

log.writeln(msg1);

log.writeln(msg2);

}

}

}

Only one thread can execute inside any of these two methods at the same time.

Had the second synchronized block been synchronized on a different object than MyClass.class, then one thread could execute inside each method at the same time.

**Java Synchronized Example**

Here is an example that starts 2 threads and have both of them call the add method on the same instance of Counter. Only one thread at a time will be able to call the add method on the same instance, because the method is synchronized on the instance it belongs to.

public class Counter{

long count = 0;

public synchronized void add(long value){

this.count += value;

}

}

public class CounterThread extends Thread{

protected Counter counter = null;

public CounterThread(Counter counter){

this.counter = counter;

}

public void run() {

for(int i=0; i<10; i++){

counter.add(i);

}

}

}

public class Example {

public static void main(String[] args){

Counter counter = new Counter();

Thread threadA = new CounterThread(counter);

Thread threadB = new CounterThread(counter);

threadA.start();

threadB.start();

}

}

Two threads are created. The same Counter instance is passed to both of them in their constructor. TheCounter.add() method is synchronized on the instance, because the add method is an instance method, and marked as synchronized. Therefore only one of the threads can call the add() method at a time. The other thread will wait until the first thread leaves the add() method, before it can execute the method itself.

If the two threads had referenced two separate Counter instances, there would have been no problems calling the add() methods simultaneously. The calls would have been to different objects, so the methods called would also be synchronized on different objects (the object owning the method). Therefore the calls would not block. Here is how that could look:

public class Example {

public static void main(String[] args){

Counter counterA = new Counter();

Counter counterB = new Counter();

Thread threadA = new CounterThread(counterA);

Thread threadB = new CounterThread(counterB);

threadA.start();

threadB.start();

}

}

Notice how the two threads, threadA and threadB, no longer reference the same counter instance. The add method of counterA and counterB are synchronized on their two owning instances. Calling add() on counterA will thus not block a call to add() on counterB.

## Java Concurrency Utilities

The synchronized mechanism was Java's first mechanism for synchronizing access to objects shared by multiple threads. The synchronized mechanism isn't very advanced though. That is why Java 5 got a whole set of[concurrency utility classes](http://tutorials.jenkov.com/java-util-concurrent/index.html) to help developers implement more fine grained concurrency control than what you get with synchronized.

# Thread Signaling

The purpose of thread signaling is to enable threads to send signals to each other. Additionally, thread signaling enables threads to wait for signals from other threads. For instance, a thread B might wait for a signal from thread A indicating that data is ready to be processed.

## Signaling via Shared Objects

A simple way for threads to send signals to each other is by setting the signal values in some shared object variable. Thread A may set the boolean member variable hasDataToProcess to true from inside a synchronized block, and thread B may read the hasDataToProcess member variable, also inside a synchronized block. Here is a simple example of an object that can hold such a signal, and provide methods to set and check it:

public class MySignal{

protected boolean hasDataToProcess = false;

public synchronized boolean hasDataToProcess(){

return this.hasDataToProcess;

}

public synchronized void setHasDataToProcess(boolean hasData){

this.hasDataToProcess = hasData;

}

}

Thread A and B must have a reference to a shared MySignal instance for the signaling to work. If thread A and B has references to different MySignal instance, they will not detect each others signals. The data to be processed can be located in a shared buffer separate from the MySignal instance.

## Busy Wait

Thread B which is to process the data is waiting for data to become available for processing. In other words, it is waiting for a signal from thread A which causes hasDataToProcess() to return true. Here is the loop that thread B is running in, while waiting for this signal:

protected MySignal sharedSignal = ...

while(!sharedSignal.hasDataToProcess()){

//do nothing... busy waiting

}

Notice how the while loop keeps executing until hasDataToProcess() returns true. This is called busy waiting. The thread is busy while waiting.

## wait(), notify() and notifyAll()

Busy waiting is not a very efficient utilization of the CPU in the computer running the waiting thread, except if the average waiting time is very small. Else, it would be smarter if the waiting thread could somehow sleep or become inactive until it receives the signal it is waiting for.

Java has a builtin wait mechanism that enable threads to become inactive while waiting for signals. The class java.lang.Object defines three methods, wait(), notify(), and notifyAll(), to facilitate this.

A thread that calls wait() on any object becomes inactive until another thread calls notify() on that object. In order to call either wait() or notify the calling thread must first obtain the lock on that object. In other words, the calling thread must call wait() or notify() from inside a synchronized block. Here is a modified version of MySignal called MyWaitNotify that uses wait() and notify().

public class MonitorObject{

}

public class MyWaitNotify{

MonitorObject myMonitorObject = new MonitorObject();

public void doWait(){

synchronized(myMonitorObject){

try{

myMonitorObject.wait();

} catch(InterruptedException e){...}

}

}

public void doNotify(){

synchronized(myMonitorObject){

myMonitorObject.notify();

}

}

}

The waiting thread would call doWait(), and the notifying thread would call doNotify(). When a thread calls notify() on an object, one of the threads waiting on that object are awakened and allowed to execute. There is also a notifyAll() method that will wake all threads waiting on a given object.

As you can see both the waiting and notifying thread calls wait() and notify() from within a synchronized block. This is mandatory! A thread cannot call wait(), notify() or notifyAll() without holding the lock on the object the method is called on. If it does, an IllegalMonitorStateException is thrown.

But, how is this possible? Wouldn't the waiting thread keep the lock on the monitor object (myMonitorObject) as long as it is executing inside a synchronized block? Will the waiting thread not block the notifying thread from ever entering the synchronized block in doNotify()? The answer is no. Once a thread calls wait() it releases the lock it holds on the monitor object. This allows other threads to call wait() or notify() too, since these methods must be called from inside a synchronized block.

Once a thread is awakened it cannot exit the wait() call until the thread calling notify() has left its synchronized block. In other words: The awakened thread must reobtain the lock on the monitor object before it can exit the wait() call, because the wait call is nested inside a synchronized block. If multiple threads are awakened using notifyAll() only one awakened thread at a time can exit the wait() method, since each thread must obtain the lock on the monitor object in turn before exiting wait().

## Missed Signals

The methods notify() and notifyAll() do not save the method calls to them in case no threads are waiting when they are called. The notify signal is then just lost. Therefore, if a thread calls notify() before the thread to signal has called wait(), the signal will be missed by the waiting thread. This may or may not be a problem, but in some cases this may result in the waiting thread waiting forever, never waking up, because the signal to wake up was missed.

To avoid losing signals they should be stored inside the signal class. In the MyWaitNotify example the notify signal should be stored in a member variable inside the MyWaitNotify instance. Here is a modified version of MyWaitNotify that does this:

public class MyWaitNotify2{

MonitorObject myMonitorObject = new MonitorObject();

boolean wasSignalled = false;

public void doWait(){

synchronized(myMonitorObject){

if(!wasSignalled){

try{

myMonitorObject.wait();

} catch(InterruptedException e){...}

}

//clear signal and continue running.

wasSignalled = false;

}

}

public void doNotify(){

synchronized(myMonitorObject){

wasSignalled = true;

myMonitorObject.notify();

}

}

}

Notice how the doNotify() method now sets the wasSignalled variable to true before calling notify(). Also, notice how the doWait() method now checks the wasSignalled variable before calling wait(). In fact it only calls wait() if no signal was received in between the previous doWait() call and this.

## Spurious Wakeups

For inexplicable reasons it is possible for threads to wake up even if notify() and notifyAll() has not been called. This is known as spurious wakeups. Wakeups without any reason.

If a spurious wakeup occurs in the MyWaitNofity2 class's doWait() method the waiting thread may continue processing without having received a proper signal to do so! This could cause serious problems in your application.

To guard against spurious wakeups the signal member variable is checked inside a while loop instead of inside an if-statement. Such a while loop is also called a spin lock. The thread awakened spins around until the condition in the spin lock (while loop) becomes false. Here is a modified version of MyWaitNotify2 that shows this:

public class MyWaitNotify3{

MonitorObject myMonitorObject = new MonitorObject();

boolean wasSignalled = false;

public void doWait(){

synchronized(myMonitorObject){

while(!wasSignalled){

try{

myMonitorObject.wait();

} catch(InterruptedException e){...}

}

//clear signal and continue running.

wasSignalled = false;

}

}

public void doNotify(){

synchronized(myMonitorObject){

wasSignalled = true;

myMonitorObject.notify();

}

}

}

Notice how the wait() call is now nested inside a while loop instead of an if-statement. If the waiting thread wakes up without having received a signal, the wasSignalled member will still be false, and the while loop will execute once more, causing the awakened thread to go back to waiting.

## Multiple Threads Waiting for the Same Signals

The while loop is also a nice solution if you have multiple threads waiting, which are all awakened using notifyAll(), but only one of them should be allowed to continue. Only one thread at a time will be able to obtain the lock on the monitor object, meaning only one thread can exit the wait() call and clear the wasSignalled flag. Once this thread then exits the synchronized block in the doWait() method, the other threads can exit the wait() call and check the wasSignalled member variable inside the while loop. However, this flag was cleared by the first thread waking up, so the rest of the awakened threads go back to waiting, until the next signal arrives.

## Don't call wait() on constant String's or global objects

An earlier version of this text had an edition of the MyWaitNotify example class which used a constant string ( "" ) as monitor object. Here is how that example looked:

public class MyWaitNotify{

**String myMonitorObject = "";**

boolean wasSignalled = false;

public void doWait(){

synchronized(myMonitorObject){

while(!wasSignalled){

try{

myMonitorObject.wait();

} catch(InterruptedException e){...}

}

//clear signal and continue running.

wasSignalled = false;

}

}

public void doNotify(){

synchronized(myMonitorObject){

wasSignalled = true;

myMonitorObject.notify();

}

}}

The problem with calling wait() and notify() on the empty string, or any other constant string is, that the JVM/Compiler internally translates constant strings into the same object. That means, that even if you have two different MyWaitNotify instances, they both reference the same empty string instance. This also means that threads calling doWait() on the first MyWaitNotify instance risk being awakened by doNotify() calls on the second MyWaitNotify instance.

The situation is sketched in the diagram below:

|  |
| --- |
| Calling wait()/notify() on string constants |

Remember, that even if the 4 threads call wait() and notify() on the same shared string instance, the signals from the doWait() and doNotify() calls are stored individually in the two MyWaitNotify instances. A doNotify() call on the MyWaitNotify 1 may wake threads waiting in MyWaitNotify 2, but the signal will only be stored in MyWaitNotify 1.

At first this may not seem like a big problem. After all, if doNotify() is called on the second MyWaitNotify instance all that can really happen is that Thread A and B are awakened by mistake. This awakened thread (A or B) will check its signal in the while loop, and go back to waiting because doNotify() was not called on the first MyWaitNotify instance, in which they are waiting. This situation is equal to a provoked spurious wakeup. Thread A or B awakens without having been signaled. But the code can handle this, so the threads go back to waiting.

The problem is, that since the doNotify() call only calls notify() and not notifyAll(), only one thread is awakened even if 4 threads are waiting on the same string instance (the empty string). So, if one of the threads A or B is awakened when really the signal was for C or D, the awakened thread (A or B) will check its signal, see that no signal was received, and go back to waiting. Neither C or D wakes up to check the signal they had actually received, so the signal is missed. This situation is equal to the missed signals problem described earlier. C and D were sent a signal but fail to respond to it.

If the doNotify() method had called notifyAll() instead of notify(), all waiting threads had been awakened and checked for signals in turn. Thread A and B would have gone back to waiting, but one of either C or D would have noticed the signal and left the doWait() method call. The other of C and D would go back to waiting, because the thread discovering the signal clears it on the way out of doWait().

You may be tempted then to always call notifyAll() instead notify(), but this is a bad idea performance wise. There is no reason to wake up all threads waiting when only one of them can respond to the signal.

So: Don't use global objects, string constants etc. for wait() / notify() mechanisms. Use an object that is unique to the construct using it. For instance, each MyWaitNotify3 (example from earlier sections) instance has its own MonitorObject instance rather than using the empty string for wait() / notify() calls.

# Deadlock

## Thread Deadlock

A deadlock is when two or more threads are blocked waiting to obtain locks that some of the other threads in the deadlock are holding. Deadlock can occur when multiple threads need the same locks, at the same time, but obtain them in different order.

For instance, if thread 1 locks A, and tries to lock B, and thread 2 has already locked B, and tries to lock A, a deadlock arises. Thread 1 can never get B, and thread 2 can never get A. In addition, neither of them will ever know. They will remain blocked on each their object, A and B, forever. This situation is a deadlock.

The situation is illustrated below:

Thread 1 locks A, waits for B

Thread 2 locks B, waits for A

Here is an example of a TreeNode class that call synchronized methods in different instances:

public class TreeNode {

TreeNode parent = null;

List children = new ArrayList();

public synchronized void addChild(TreeNode child){

if(!this.children.contains(child)) {

this.children.add(child);

child.setParentOnly(this);

} }

public synchronized void addChildOnly(TreeNode child){

if(!this.children.contains(child){

this.children.add(child);

}

}

public synchronized void setParent(TreeNode parent){

this.parent = parent;

parent.addChildOnly(this);

}

public synchronized void setParentOnly(TreeNode parent){

this.parent = parent;

}

}

If a thread (1) calls the parent.addChild(child) method at the same time as another thread (2) calls the child.setParent(parent) method, on the same parent and child instances, a deadlock can occur. Here is some pseudo code that illustrates this:

Thread 1: parent.addChild(child); //locks parent

--> child.setParentOnly(parent);

Thread 2: child.setParent(parent); //locks child

--> parent.addChildOnly()

First thread 1 calls parent.addChild(child). Since addChild() is synchronized thread 1 effectively locks the parent object for access from other treads.

Then thread 2 calls child.setParent(parent). Since setParent() is synchronized thread 2 effectively locks the child object for acces from other threads.

Now both child and parent objects are locked by two different threads. Next thread 1 tries to call child.setParentOnly() method, but the child object is locked by thread 2, so the method call just blocks. Thread 2 also tries to call parent.addChildOnly() but the parent object is locked by thread 1, causing thread 2 to block on that method call. Now both threads are blocked waiting to obtain locks the other thread holds.

Note: The two threads must call parent.addChild(child) and child.setParent(parent) at the same time as described above, and on the same two parent and child instances for a deadlock to occur. The code above may execute fine for a long time until all of a sudden it deadlocks.

The threads really need to take the locks \*at the same time\*. For instance, if thread 1 is a bit ahead of thread2, and thus locks both A and B, then thread 2 will be blocked already when trying to lock B. Then no deadlock occurs. Since thread scheduling often is unpredictable there is no way to predict \*when\* a deadlock occurs. Only that it \*can\* occur.

## More Complicated Deadlocks

Deadlock can also include more than two threads. This makes it harder to detect. Here is an example in which four threads have deadlocked:

Thread 1 locks A, waits for B

Thread 2 locks B, waits for C

Thread 3 locks C, waits for D

Thread 4 locks D, waits for A

Thread 1 waits for thread 2, thread 2 waits for thread 3, thread 3 waits for thread 4, and thread 4 waits for thread 1.

## Database Deadlocks

A more complicated situation in which deadlocks can occur, is a database transaction. A database transaction may consist of many SQL update requests. When a record is updated during a transaction, that record is locked for updates from other transactions, until the first transaction completes. Each update request within the same transaction may therefore lock some records in the database.

If multiple transactions are running at the same time that need to update the same records, there is a risk of them ending up in a deadlock.

For example

Transaction 1, request 1, locks record 1 for update

Transaction 2, request 1, locks record 2 for update

Transaction 1, request 2, tries to lock record 2 for update.

Transaction 2, request 2, tries to lock record 1 for update.

Since the locks are taken in different requests, and not all locks needed for a given transaction are known ahead of time, it is hard to detect or prevent deadlocks in database transactions.

# Deadlock Prevention

## Lock Ordering

Deadlock occurs when multiple threads need the same locks but obtain them in different order.

If you make sure that all locks are always taken in the same order by any thread, deadlocks cannot occur. Look at this example:

Thread 1:

lock A

lock B

Thread 2:

wait for A

lock C (when A locked)

Thread 3:

wait for A

wait for B

wait for C

If a thread, like Thread 3, needs several locks, it must take them in the decided order. It cannot take a lock later in the sequence until it has obtained the earlier locks.

For instance, neither Thread 2 or Thread 3 can lock C until they have locked A first. Since Thread 1 holds lock A, Thread 2 and 3 must first wait until lock A is unlocked. Then they must succeed in locking A, before they can attempt to lock B or C.

Lock ordering is a simple yet effective deadlock prevention mechanism. However, it can only be used if you know about all locks needed ahead of taking any of the locks. This is not always the case.

## Lock Timeout

Another deadlock prevention mechanism is to put a timeout on lock attempts meaning a thread trying to obtain a lock will only try for so long before giving up. If a thread does not succeed in taking all necessary locks within the given timeout, it will backup, free all locks taken, wait for a random amount of time and then retry. The random amount of time waited serves to give other threads trying to take the same locks a chance to take all locks, and thus let the application continue running without locking.

Here is an example of two threads trying to take the same two locks in different order, where the threads back up and retry:

Thread 1 locks A

Thread 2 locks B

Thread 1 attempts to lock B but is blocked

Thread 2 attempts to lock A but is blocked

Thread 1's lock attempt on B times out

Thread 1 backs up and releases A as well

Thread 1 waits randomly (e.g. 257 millis) before retrying.

Thread 2's lock attempt on A times out

Thread 2 backs up and releases B as well

Thread 2 waits randomly (e.g. 43 millis) before retrying.

In the above example Thread 2 will retry taking the locks about 200 millis before Thread 1 and will therefore likely succeed at taking both locks. Thread 1 will then wait already trying to take lock A. When Thread 2 finishes, Thread 1 will be able to take both locks too (unless Thread 2 or another thread takes the locks in between).

An issue to keep in mind is, that just because a lock times out it does not necessarily mean that the threads had deadlocked. It could also just mean that the thread holding the lock (causing the other thread to time out) takes a long time to complete its task.

Additionally, if enough threads compete for the same resources they still risk trying to take the threads at the same time again and again, even if timing out and backing up. This may not occur with 2 threads each waiting between 0 and 500 millis before retrying, but with 10 or 20 threads the situation is different. Then the likeliness of two threads waiting the same time before retrying (or close enough to cause problems) is a lot higher.

A problem with the lock timeout mechanism is that it is not possible to set a timeout for entering a synchronized block in Java. You will have to create a custom lock class or use one of the Java 5 concurrency constructs in the java.util.concurrency package. Writing custom locks isn't difficult but it is outside the scope of this text. Later texts in the Java concurrency trails will cover custom locks.

## Deadlock Detection

Deadlock detection is a heavier deadlock prevention mechanism aimed at cases in which lock ordering isn't possible, and lock timeout isn't feasible.

Every time a thread **takes** a lock it is noted in a data structure (map, graph etc.) of threads and locks. Additionally, whenever a thread **requests** a lock this is also noted in this data structure.

When a thread requests a lock but the request is denied, the thread can traverse the lock graph to check for deadlocks. For instance, if a Thread A requests lock 7, but lock 7 is held by Thread B, then Thread A can check if Thread B has requested any of the locks Thread A holds (if any). If Thread B has requested so, a deadlock has occurred (Thread A having taken lock 1, requesting lock 7, Thread B having taken lock 7, requesting lock 1).

Of course a deadlock scenario may be a lot more complicated than two threads holding each others locks. Thread A may wait for Thread B, Thread B waits for Thread C, Thread C waits for Thread D, and Thread D waits for Thread A. In order for Thread A to detect a deadlock it must transitively examine all requested locks by Thread B. From Thread B's requested locks Thread A will get to Thread C, and then to Thread D, from which it finds one of the locks Thread A itself is holding. Then it knows a deadlock has occurred.

Below is a graph of locks taken and requested by 4 threads (A, B, C and D). A data structure like this that can be used to detect deadlocks.

|  |
| --- |
| Deadlock Detection Data Structure |

So what do the threads do if a deadlock is detected?

One possible action is to release all locks, backup, wait a random amount of time and then retry. This is similar to the simpler lock timeout mechanism except threads only backup when a deadlock has actually occurred. Not just because their lock requests timed out. However, if a lot of threads are competing for the same locks they may repeatedly end up in a deadlock even if they back up and wait.

A better option is to determine or assign a priority of the threads so that only one (or a few) thread backs up. The rest of the threads continue taking the locks they need as if no deadlock had occurred. If the priority assigned to the threads is fixed, the same threads will always be given higher priority. To avoid this you may assign the priority randomly whenever a deadlock is detected.

# Starvation and Fairness

If a thread is not granted CPU time because other threads grab it all, it is called "starvation". The thread is "starved to death" because other threads are allowed the CPU time instead of it. The solution to starvation is called "fairness" - that all threads are fairly granted a chance to execute.

## Causes of Starvation in Java

The following three common causes can lead to starvation of threads in Java:

1. Threads with high priority swallow all CPU time from threads with lower priority.
2. Threads are blocked indefinately waiting to enter a synchronized block, because other threads are constantly allowed access before it.
3. Threads waiting on an object (called wait() on it) remain waiting indefinitely because other threads are constantly awakened instead of it.

### Threads with high priority swallow all CPU time from threads with lower priority

You can set the thread priority of each thread individually. The higher the priority the more CPU time the thread is granted. You can set the priority of threads between 1 and 10. Exactly how this is interpreted depends on the operating system your application is running on. For most applications you are better off leaving the priority unchanged.

### Threads are blocked indefinitely waiting to enter a synchronized block

Java's synchronized code blocks can be another cause of starvation. Java's synchronized code block makes no guarantee about the sequence in which threads waiting to enter the synchronized block are allowed to enter. This means that there is a theoretical risk that a thread remains blocked forever trying to enter the block, because other threads are constantly granted access before it. This problem is called "starvation", that a thread is "starved to death" by because other threads are allowed the CPU time instead of it.

### Threads waiting on an object (called wait() on it) remain waiting indefinitely

The notify() method makes no guarantee about what thread is awakened if multiple thread have called wait() on the object notify() is called on. It could be any of the threads waiting. Therefore there is a risk that a thread waiting on a certain object is never awakened because other waiting threads are always awakened instead of it.

## Implementing Fairness in Java

While it is not possible to implement 100% fairness in Java we can still implement our synchronization constructs to increase fairness between threads.

First lets study a simple synchronized code block:

public class Synchronizer{

public synchronized void doSynchronized(){

//do a lot of work which takes a long time

}

}

If more than one thread call the doSynchronized() method, some of them will be blocked until the first thread granted access has left the method. If more than one thread are blocked waiting for access there is no guarantee about which thread is granted access next.

### Using Locks Instead of Synchronized Blocks

To increase the fairness of waiting threads first we will change the code block to be guarded by a lock rather than a synchronized block:

public class Synchronizer{

Lock lock = new Lock();

public void doSynchronized() throws InterruptedException{

this.lock.lock();

//critical section, do a lot of work which takes a long time

this.lock.unlock();

}

}

Notice how the doSynchronized() method is no longer declared synchronized. Instead the critical section is guarded by the lock.lock() and lock.unlock() calls.

A simple implementation of the Lock class could look like this:

public class Lock{

private boolean isLocked = false;

private Thread lockingThread = null;

public synchronized void lock() throws InterruptedException{

while(isLocked){

wait();

}

isLocked = true;

lockingThread = Thread.currentThread();

}

public synchronized void unlock(){

if(this.lockingThread != Thread.currentThread()){

throw new IllegalMonitorStateException(

"Calling thread has not locked this lock");

}

isLocked = false;

lockingThread = null;

notify();

}

}

If you look at the Synchronizer class above and look into this Lock implementation you will notice that threads are now blocked trying to access the lock() method, if more than one thread calls lock() simultanously. Second, if the lock is locked, the threads are blocked in the wait() call inside the while(isLocked) loop in the lock() method. Remember that a thread calling wait() releases the synchronization lock on the Lock instance, so threads waiting to enter lock() can now do so. The result is that multiple threads can end up having called wait() inside lock().

If you look back at the doSynchronized() method you will notice that the comment between lock() and unlock() states, that the code in between these two calls take a "long" time to execute. Let us further assume that this code takes long time to execute compared to entering the lock() method and calling wait() because the lock is locked. This means that the majority of the time waited to be able to lock the lock and enter the critical section is spent waiting in the wait() call inside the lock() method, not being blocked trying to enter the lock() method.

As stated earlier synchronized blocks makes no guarantees about what thread is being granted access if more than one thread is waiting to enter. Nor does wait() make any guarantees about what thread is awakened when notify() is called. So, the current version of the Lock class makes no different guarantees with respect to fairness than synchronized version of doSynchronized(). But we can change that.

The current version of the Lock class calls its own wait() method. If instead each thread calls wait() on a separate object, so that only one thread has called wait() on each object, the Lock class can decide which of these objects to call notify() on, thereby effectively selecting exactly what thread to awaken.

### A Fair Lock

Below is shown the previous Lock class turned into a fair lock called FairLock. You will notice that the implementation has changed a bit with respect to synchronization and wait() / notify() compared to the Lock class shown earlier.

Exactly how I arrived at this design beginning from the previous Lock class is a longer story involving several incremental design steps, each fixing the problem of the previous step: [Nested Monitor Lockout](http://tutorials.jenkov.com/java-concurrency/nested-monitor-lockout.html), [Slipped Conditions](http://tutorials.jenkov.com/java-concurrency/slipped-conditions.html), and [Missed Signals](http://tutorials.jenkov.com/java-concurrency/thread-signaling.html#missedsignals). That discussion is left out of this text to keep the text short, but each of the steps are discussed in the appropriate texts on the topic ( see the links above). What is important is, that every thread calling lock() is now queued, and only the first thread in the queue is allowed to lock the FairLock instance, if it is unlocked. All other threads are parked waiting until they reach the top of the queue.

public class FairLock {

private boolean isLocked = false;

private Thread lockingThread = null;

private List<QueueObject> waitingThreads =

new ArrayList<QueueObject>();

public void lock() throws InterruptedException{

QueueObject queueObject = new QueueObject();

boolean isLockedForThisThread = true;

synchronized(this){

waitingThreads.add(queueObject);

}

while(isLockedForThisThread){

synchronized(this){

isLockedForThisThread =

isLocked || waitingThreads.get(0) != queueObject;

if(!isLockedForThisThread){

isLocked = true;

waitingThreads.remove(queueObject);

lockingThread = Thread.currentThread();

return;

}

}

try{

queueObject.doWait();

}catch(InterruptedException e){

synchronized(this) { waitingThreads.remove(queueObject); }

throw e;

}

}

}

public synchronized void unlock(){

if(this.lockingThread != Thread.currentThread()){

throw new IllegalMonitorStateException(

"Calling thread has not locked this lock");

}

isLocked = false;

lockingThread = null;

if(waitingThreads.size() > 0){

waitingThreads.get(0).doNotify();

}

}

}

public class QueueObject {

private boolean isNotified = false;

public synchronized void doWait() throws InterruptedException {

while(!isNotified){

this.wait();

}

this.isNotified = false;

}

public synchronized void doNotify() {

this.isNotified = true;

this.notify();

}

public boolean equals(Object o) {

return this == o;

}

}

First you might notice that the lock() method is no longer declared synchronized. Instead only the blocks necessary to synchronize are nested inside synchronized blocks.

FairLock creates a new instance of QueueObject and enqueue it for each thread calling lock(). The thread callingunlock() will take the top QueueObject in the queue and call doNotify() on it, to awaken the thread waiting on that object. This way only one waiting thread is awakened at a time, rather than all waiting threads. This part is what governs the fairness of the FairLock.

Notice how the state of the lock is still tested and set within the same synchronized block to avoid slipped conditions.

Also notice that the QueueObject is really a semaphore. The doWait() and doNotify() methods store the signal internally in the QueueObject. This is done to avoid missed signals caused by a thread being preempted just before calling queueObject.doWait(), by another thread which calls unlock() and therebyqueueObject.doNotify(). The queueObject.doWait() call is placed outside the synchronized(this) block to avoid nested monitor lockout, so another thread can actually call unlock() when no thread is executing inside thesynchronized(this) block in lock() method.

Finally, notice how the queueObject.doWait() is called inside a try - catch block. In case an InterruptedException is thrown the thread leaves the lock() method, and we need to dequeue it.

### A Note on Performance

If you compare the Lock and FairLock classes you will notice that there is somewhat more going on inside thelock() and unlock() in the FairLock class. This extra code will cause the FairLock to be a sligtly slower synchronization mechanism than Lock. How much impact this will have on your application depends on how long time the code in the critical section guarded by the FairLock takes to execute. The longer this takes to execute, the less significant the added overhead of the synchronizer is. It does of course also depend on how often this code is called.

# Slipped Conditions

## What is Slipped Conditions?

Slipped conditions means, that from the time a thread has checked a certain condition until it acts upon it, the condition has been changed by another thread so that it is errornous for the first thread to act. Here is a simple example:

public class Lock {

private boolean isLocked = true;

public void lock(){

synchronized(this){

while(isLocked){

try{

this.wait();

} catch(InterruptedException e){

//do nothing, keep waiting

}

}

}

synchronized(this){

isLocked = true;

}

}

public synchronized void unlock(){

isLocked = false;

this.notify();

}

}

Notice how the lock() method contains two synchronized blocks. The first block waits until isLocked is false. The second block sets isLocked to true, to lock the Lock instance for other threads.

Imagine that isLocked is false, and two threads call lock() at the same time. If the first thread entering the first synchronized block is preempted right after the first synchronized block, this thread will have checked isLocked and noted it to be false. If the second thread is now allowed to execute, and thus enter the first synchronized block, this thread too will see isLocked as false. Now both threads have read the condition as false. Then both threads will enter the second synchronized block, set isLocked to true, and continue.

This situation is an example of slipped conditions. Both threads test the condition, then exit the synchronized block, thereby allowing other threads to test the condition, before any of the two first threads change the conditions for subsequent threads. In other words, the condition has slipped from the time the condition was checked until the threads change it for subsequent threads.

To avoid slipped conditions the testing and setting of the conditions must be done atomically by the thread doing it, meaning that no other thread can check the condition in between the testing and setting of the condition by the first thread.

The solution in the example above is simple. Just move the line isLocked = true; up into the first synchronized block, right after the while loop. Here is how it looks:

public class Lock {

private boolean isLocked = true;

public void lock(){

synchronized(this){

while(isLocked){

try{

this.wait();

} catch(InterruptedException e){

//do nothing, keep waiting

}

}

isLocked = true;

}

}

public synchronized void unlock(){

isLocked = false;

this.notify();

}

}

Now the testing and setting of the isLocked condition is done atomically from inside the same synchronized block.

## A More Realistic Example

You may rightfully argue that you would never implement a Lock like the first implementation shown in this text, and thus claim slipped conditions to be a rather theoretical problem. But the first example was kept rather simple to better convey the notion of slipped conditions.

A more realistic example would be during the implementation of a fair lock, as discussed in the text on [Starvation and Fairness](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html). If we look at the naive implementation from the text [Nested Monitor Lockout](http://tutorials.jenkov.com/java-concurrency/nested-monitor-lockout.html), and try to remove the nested monitor lock problem it, it is easy to arrive at an implementation that suffers from slipped conditions. First I'll show the example from the nested monitor lockout text:

//Fair Lock implementation with nested monitor lockout problem

public class FairLock {

private boolean isLocked = false;

private Thread lockingThread = null;

private List<QueueObject> waitingThreads =

new ArrayList<QueueObject>();

public void lock() throws InterruptedException{

QueueObject queueObject = new QueueObject();

**synchronized(this){**

waitingThreads.add(queueObject);

while(isLocked || waitingThreads.get(0) != queueObject){

**synchronized(queueObject){**

try{

queueObject.wait();

}catch(InterruptedException e){

waitingThreads.remove(queueObject);

throw e;

}

**}**

}

waitingThreads.remove(queueObject);

isLocked = true;

lockingThread = Thread.currentThread();

**}**

}

public **synchronized** void unlock(){

if(this.lockingThread != Thread.currentThread()){

throw new IllegalMonitorStateException(

"Calling thread has not locked this lock");

}

isLocked = false;

lockingThread = null;

if(waitingThreads.size() > 0){

QueueObject queueObject = waitingThread.get(0);

**synchronized(queueObject){**

queueObject.notify();

**}**

}

}

}

public class QueueObject {}

Notice how the synchronized(queueObject) with its queueObject.wait() call is nested inside thesynchronized(this) block, resulting in the nested monitor lockout problem. To avoid this problem thesynchronized(queueObject) block must be moved outside the synchronized(this) block. Here is how that could look:

//Fair Lock implementation with slipped conditions problem

public class FairLock {

private boolean isLocked = false;

private Thread lockingThread = null;

private List<QueueObject> waitingThreads =

new ArrayList<QueueObject>();

public void lock() throws InterruptedException{

QueueObject queueObject = new QueueObject();

synchronized(this){

waitingThreads.add(queueObject);

}

boolean mustWait = true;

while(mustWait){

synchronized(this){

mustWait = isLocked || waitingThreads.get(0) != queueObject;

}

synchronized(queueObject){

if(mustWait){

try{

queueObject.wait();

}catch(InterruptedException e){

waitingThreads.remove(queueObject);

throw e;

}

}

}

}

synchronized(this){

waitingThreads.remove(queueObject);

isLocked = true;

lockingThread = Thread.currentThread();

}

}

}

Note: Only the lock() method is shown, since it is the only method I have changed.

Notice how the lock() method now contains 3 synchronized blocks.

The first synchronized(this) block checks the condition by setting mustWait = isLocked || waitingThreads.get(0) != queueObject.

The second synchronized(queueObject) block checks if the thread is to wait or not. Already at this time another thread may have unlocked the lock, but lets forget that for the time being. Let's assume that the lock was unlocked, so the thread exits the synchronized(queueObject) block right away.

The third synchronized(this) block is only executed if mustWait = false. This sets the condition isLockedback to true etc. and leaves the lock() method.

Imagine what will happen if two threads call lock() at the same time when the lock is unlocked. First thread 1 will check the isLocked conditition and see it false. Then thread 2 will do the same thing. Then neither of them will wait, and both will set the state isLocked to true. This is a prime example of slipped conditions.

### Removing the Slipped Conditions Problem

To remove the slipped conditions problem from the example above, the content of the last synchronized(this)block must be moved up into the first block. The code will naturally have to be changed a little bit too, to adapt to this move. Here is how it looks:

//Fair Lock implementation without nested monitor lockout problem,

//but with missed signals problem.

public class FairLock {

private boolean isLocked = false;

private Thread lockingThread = null;

private List<QueueObject> waitingThreads =

new ArrayList<QueueObject>();

public void lock() throws InterruptedException{

QueueObject queueObject = new QueueObject();

synchronized(this){

waitingThreads.add(queueObject);

}

boolean mustWait = true;

while(mustWait){

**synchronized(this){**

**mustWait = isLocked || waitingThreads.get(0) != queueObject;**

**if(!mustWait){**

**waitingThreads.remove(queueObject);**

**isLocked = true;**

**lockingThread = Thread.currentThread();**

**return;**

**}**

**}**

synchronized(queueObject){

if(mustWait){

try{

queueObject.wait();

}catch(InterruptedException e){

waitingThreads.remove(queueObject);

throw e;

}

}

}

}

}

}

Notice how the local variable mustWait is tested and set within the same synchronized code block now. Also notice, that even if the mustWait local variable is also checked outside the synchronized(this) code block, in thewhile(mustWait) clause, the value of the mustWait variable is never changed outside thesynchronized(this). A thread that evaluates mustWait to false will atomically also set the internal conditions (isLocked) so that any other thread checking the condition will evaluate it to true.

The return; statement in the synchronized(this) block is not necessary. It is just a small optimization. If the thread must not wait (mustWait == false), then there is no reason to enter thesynchronized(queueObject) block and execute the if(mustWait) clause.

The observant reader will notice that the above implementation of a fair lock still suffers from a missed signal problem. Imagine that the FairLock instance is locked when a thread calls lock(). After the firstsynchronized(this) block mustWait is true. Then imagine that the thread calling lock() is preempted, and the thread that locked the lock calls unlock(). If you look at the unlock() implementation shown earlier, you will notice that it calls queueObject.notify(). But, since the thread waiting in lock() has not yet calledqueueObject.wait(), the call to queueObject.notify() passes into oblivion. The signal is missed. When the thread calling lock() right after calls queueObject.wait() it will remain blocked until some other thread callsunlock(), which may never happen.

The missed signals problems is the reason that the FairLock implementation shown in the text [Starvation and Fairness](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html) has turned the QueueObject class into a semaphore with two methods: doWait() and doNotify(). These methods store and react the signal internally in the QueueObject. That way the signal is not missed, even ifdoNotify() is called before doWait().

# Locks in Java

A lock is a thread synchronization mechanism like synchronized blocks except locks can be more sophisticated than Java's synchronized blocks. Locks (and other more advanced synchronization mechanisms) are created using synchronized blocks, so it is not like we can get totally rid of the synchronized keyword.

From Java 5 the package java.util.concurrent.locks contains several lock implementations, so you may not have to implement your own locks. But you will still need to know how to use them, and it can still be useful to know the theory behind their implementation. For more details, see my tutorial on the[java.util.concurrent.locks.Lock](http://tutorials.jenkov.com/java-util-concurrent/lock.html) interface.

## A Simple Lock

Let's start out by looking at a synchronized block of Java code:

public class Counter{

private int count = 0;

public int inc(){

synchronized(this){

return ++count;

}

}

}

Notice the synchronized(this) block in the inc() method. This block makes sure that only one thread can execute the return ++count at a time. The code in the synchronized block could have been more advanced, but the simple ++count suffices to get the point across.

The Counter class could have been written like this instead, using a Lock instead of a synchronized block:

public class Counter{

private Lock lock = new Lock();

private int count = 0;

public int inc(){

lock.lock();

int newCount = ++count;

lock.unlock();

return newCount;

}

}

The lock() method locks the Lock instance so that all threads calling lock() are blocked until unlock() is executed.

Here is a simple Lock implementation:

public class Lock{

private boolean isLocked = false;

public synchronized void lock()

throws InterruptedException{

while(isLocked){

wait();

}

isLocked = true;

}

public synchronized void unlock(){

isLocked = false;

notify();

}

}

Notice the while(isLocked) loop, which is also called a "spin lock". Spin locks and the methods wait() andnotify() are covered in more detail in the text [Thread Signaling](http://tutorials.jenkov.com/java-concurrency/thread-signaling.html). While isLocked is true, the thread callinglock() is parked waiting in the wait() call. In case the thread should return unexpectedly from the wait() call without having received a notify() call (AKA a [Spurious Wakeup](http://tutorials.jenkov.com/java-concurrency/thread-signaling.html#spuriouswakeups)) the thread re-checks the isLocked condition to see if it is safe to proceed or not, rather than just assume that being awakened means it is safe to proceed. IfisLocked is false, the thread exits the while(isLocked) loop, and sets isLocked back to true, to lock the Lockinstance for other threads calling lock().

When the thread is done with the code in the [critical section](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html) (the code between lock() and unlock()), the thread calls unlock(). Executing unlock() sets isLocked back to false, and notifies (awakens) one of the threads waiting in the wait() call in the lock() method, if any.

## Lock Reentrance

Synchronized blocks in Java are reentrant. This means, that if a Java thread enters a synchronized block of code, and thereby take the lock on the monitor object the block is synchronized on, the thread can enter other Java code blocks synchronized on the same monitor object. Here is an example:

public class Reentrant{

public synchronized outer(){

inner();

}

public synchronized inner(){

//do something

}

}

Notice how both outer() and inner() are declared synchronized, which in Java is equivalent to asynchronized(this) block. If a thread calls outer() there is no problem calling inner() from inside outer(), since both methods (or blocks) are synchronized on the same monitor object ("this"). If a thread already holds the lock on a monitor object, it has access to all blocks synchronized on the same monitor object. This is called reentrance. The thread can reenter any block of code for which it already holds the lock.

The lock implementation shown earlier is not reentrant. If we rewrite the Reentrant class like below, the thread calling outer() will be blocked inside the lock.lock() in the inner() method.

public class Reentrant2{

Lock lock = new Lock();

public outer(){

lock.lock();

inner();

lock.unlock();

}

public synchronized inner(){

lock.lock();

//do something

lock.unlock();

}

}

A thread calling outer() will first lock the Lock instance. Then it will call inner(). Inside the inner() method the thread will again try to lock the Lock instance. This will fail (meaning the thread will be blocked), since the Lockinstance was locked already in the outer() method.

The reason the thread will be blocked the second time it calls lock() without having called unlock() in between, is apparent when we look at the lock() implementation:

public class Lock{

boolean isLocked = false;

public synchronized void lock()

throws InterruptedException{

while(isLocked){

wait();

}

isLocked = true;

}

...

}

It is the condition inside the while loop (spin lock) that determines if a thread is allowed to exit the lock() method or not. Currently the condition is that isLocked must be false for this to be allowed, regardless of what thread locked it.

To make the Lock class reentrant we need to make a small change:

public class Lock{

boolean isLocked = false;

Thread lockedBy = null;

int lockedCount = 0;

public synchronized void lock()

throws InterruptedException{

Thread callingThread = Thread.currentThread();

while(isLocked && lockedBy != callingThread){

wait();

}

isLocked = true;

lockedCount++;

lockedBy = callingThread;

}

public synchronized void unlock(){

if(Thread.curentThread() == this.lockedBy){

lockedCount--;

if(lockedCount == 0){

isLocked = false;

notify();

}

}

}

...

}

Notice how the while loop (spin lock) now also takes the thread that locked the Lock instance into consideration. If either the lock is unlocked (isLocked = false) or the calling thread is the thread that locked the Lock instance, the while loop will not execute, and the thread calling lock() will be allowed to exit the method.

Additionally, we need to count the number of times the lock has been locked by the same thread. Otherwise, a single call to unlock() will unlock the lock, even if the lock has been locked multiple times. We don't want the lock to be unloced until the thread that locked it, has executed the same amount of unlock() calls as lock() calls.

The Lock class is now reentrant.

## Lock Fairness

Java's synchronized blocks makes no guarantees about the sequence in which threads trying to enter them are granted access. Therefore, if many threads are constantly competing for access to the same synchronized block, there is a risk that one or more of the threads are never granted access - that access is always granted to other threads. This is called starvation. To avoid this a Lock should be fair. Since the Lock implementations shown in this text uses synchronized blocks internally, they do not guarantee fairness. Starvation and fairness are discussed in more detail in the text [Starvation and Fairness](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html).

## Calling unlock() From a finally-clause

When guarding a critical section with a Lock, and the critical section may throw exceptions, it is important to call theunlock() method from inside a finally-clause. Doing so makes sure that the Lock is unlocked so other threads can lock it. Here is an example:

lock.lock();

try{

//do critical section code, which may throw exception

} finally {

lock.unlock();

}

This little construct makes sure that the Lock is unlocked in case an exception is thrown from the code in the critical section. If unlock() was not called from inside a finally-clause, and an exception was thrown from the critical section, the Lock would remain locked forever, causing all threads calling lock() on that Lock instance to halt indefinately.

A read / write lock is more sophisticated lock than the Lock implementations shown in the text [Locks in Java](http://tutorials.jenkov.com/java-concurrency/locks.html). Imagine you have an application that reads and writes some resource, but writing it is not done as much as reading it is. Two threads reading the same resource does not cause problems for each other, so multiple threads that want to read the resource are granted access at the same time, overlapping. But, if a single thread wants to write to the resource, no other reads nor writes must be in progress at the same time. To solve this problem of allowing multiple readers but only one writer, you will need a read / write lock.

Java 5 comes with read / write lock implementations in the java.util.concurrent package. Even so, it may still be useful to know the theory behind their implementation.

## Read / Write Lock Java Implementation

First let's summarize the conditions for getting read and write access to the resource:

|  |  |
| --- | --- |
| **Read Access** | If no threads are writing, and no threads have requested write access. |
| **Write Access** | If no threads are reading or writing. |

If a thread wants to read the resource, it is okay as long as no threads are writing to it, and no threads have requested write access to the resource. By up-prioritizing write-access requests we assume that write requests are more important than read-requests. Besides, if reads are what happens most often, and we did not up-prioritize writes,[starvation](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html) could occur. Threads requesting write access would be blocked until all readers had unlocked theReadWriteLock. If new threads were constantly granted read access the thread waiting for write access would remain blocked indefinately, resulting in [starvation](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html). Therefore a thread can only be granted read access if no thread has currently locked the ReadWriteLock for writing, or requested it locked for writing.

A thread that wants write access to the resource can be granted so when no threads are reading nor writing to the resource. It doesn't matter how many threads have requested write access or in what sequence, unless you want to guarantee fairness between threads requesting write access.

With these simple rules in mind we can implement a ReadWriteLock as shown below:

public class ReadWriteLock{

private int readers = 0;

private int writers = 0;

private int writeRequests = 0;

public synchronized void lockRead() throws InterruptedException{

while(writers > 0 || writeRequests > 0){

wait();

}

readers++;

}

public synchronized void unlockRead(){

readers--;

notifyAll();

}

public synchronized void lockWrite() throws InterruptedException{

writeRequests++;

while(readers > 0 || writers > 0){

wait();

}

writeRequests--;

writers++;

}

public synchronized void unlockWrite() throws InterruptedException{

writers--;

notifyAll();

}

}

The ReadWriteLock has two lock methods and two unlock methods. One lock and unlock method for read access and one lock and unlock for write access.

The rules for read access are implemented in the lockRead() method. All threads get read access unless there is a thread with write access, or one or more threads have requested write access.

The rules for write access are implemented in the lockWrite() method. A thread that wants write access starts out by requesting write access (writeRequests++). Then it will check if it can actually get write access. A thread can get write access if there are no threads with read access to the resource, and no threads with write access to the resource. How many threads have requested write access doesn't matter.

It is worth noting that both unlockRead() and unlockWrite() calls notifyAll() rather than notify(). To explain why that is, imagine the following situation:

Inside the ReadWriteLock there are threads waiting for read access, and threads waiting for write access. If a thread awakened by notify() was a read access thread, it would be put back to waiting because there are threads waiting for write access. However, none of the threads awaiting write access are awakened, so nothing more happens. No threads gain neither read nor write access. By calling noftifyAll() all waiting threads are awakened and check if they can get the desired access.

Calling notifyAll() also has another advantage. If multiple threads are waiting for read access and none for write access, and unlockWrite() is called, all threads waiting for read access are granted read access at once - not one by one.

## Read / Write Lock Reentrance

The ReadWriteLock class shown earlier is not [reentrant](http://tutorials.jenkov.com/java-concurrency/locks.html#reentrance). If a thread that has write access requests it again, it will block because there is already one writer - itself. Furthermore, consider this case:

1. Thread 1 gets read access.
2. Thread 2 requests write access but is blocked because there is one reader.
3. Thread 1 re-requests read access (re-enters the lock), but is blocked because there is a write request

In this situation the previous ReadWriteLock would lock up - a situation similar to deadlock. No threads requesting neither read nor write access would be granted so.

To make the ReadWriteLock reentrant it is necessary to make a few changes. Reentrance for readers and writers will be dealt with separately.

## Read Reentrance

To make the ReadWriteLock reentrant for readers we will first establish the rules for read reentrance:

* A thread is granted read reentrance if it can get read access (no writers or write requests), or if it already has read access (regardless of write requests).

To determine if a thread has read access already a reference to each thread granted read access is kept in a Map along with how many times it has acquired read lock. When determing if read access can be granted this Map will be checked for a reference to the calling thread. Here is how the lockRead() and unlockRead() methods looks after that change:

public class ReadWriteLock{

private Map<Thread, Integer> readingThreads =

new HashMap<Thread, Integer>();

private int writers = 0;

private int writeRequests = 0;

public synchronized void lockRead() throws InterruptedException{

Thread callingThread = Thread.currentThread();

while(! canGrantReadAccess(callingThread)){

wait();

}

readingThreads.put(callingThread,

(getAccessCount(callingThread) + 1));

}

public synchronized void unlockRead(){

Thread callingThread = Thread.currentThread();

int accessCount = getAccessCount(callingThread);

if(accessCount == 1){ readingThreads.remove(callingThread); }

else { readingThreads.put(callingThread, (accessCount -1)); }

notifyAll();

}

private boolean canGrantReadAccess(Thread callingThread){

if(writers > 0) return false;

if(isReader(callingThread) return true;

if(writeRequests > 0) return false;

return true;

}

private int getReadAccessCount(Thread callingThread){

Integer accessCount = readingThreads.get(callingThread);

if(accessCount == null) return 0;

return accessCount.intValue();

}

private boolean isReader(Thread callingThread){

return readingThreads.get(callingThread) != null;

}

}

As you can see read reentrance is only granted if no threads are currently writing to the resource. Additionally, if the calling thread already has read access this takes precedence over any writeRequests.

## Write Reentrance

Write reentrance is granted only if the thread has already write access. Here is how the lockWrite() andunlockWrite() methods look after that change:

public class ReadWriteLock{

private Map<Thread, Integer> readingThreads =

new HashMap<Thread, Integer>();

private int writeAccesses = 0;

private int writeRequests = 0;

private Thread writingThread = null;

public synchronized void lockWrite() throws InterruptedException{

writeRequests++;

Thread callingThread = Thread.currentThread();

while(! canGrantWriteAccess(callingThread)){

wait();

}

writeRequests--;

writeAccesses++;

writingThread = callingThread;

}

public synchronized void unlockWrite() throws InterruptedException{

writeAccesses--;

if(writeAccesses == 0){

writingThread = null;

}

notifyAll();

}

private boolean canGrantWriteAccess(Thread callingThread){

if(hasReaders()) return false;

if(writingThread == null) return true;

if(!isWriter(callingThread)) return false;

return true;

}

private boolean hasReaders(){

return readingThreads.size() > 0;

}

private boolean isWriter(Thread callingThread){

return writingThread == callingThread;

}

}

Notice how the thread currently holding the write lock is now taken into account when determining if the calling thread can get write access.

## Read to Write Reentrance

Sometimes it is necessary for a thread that have read access to also obtain write access. For this to be allowed the thread must be the only reader. To achieve this the writeLock() method should be changed a bit. Here is what it would look like:

public class ReadWriteLock{

private Map<Thread, Integer> readingThreads =

new HashMap<Thread, Integer>();

private int writeAccesses = 0;

private int writeRequests = 0;

private Thread writingThread = null;

public synchronized void lockWrite() throws InterruptedException{

writeRequests++;

Thread callingThread = Thread.currentThread();

while(! canGrantWriteAccess(callingThread)){

wait();

}

writeRequests--;

writeAccesses++;

writingThread = callingThread;

}

public synchronized void unlockWrite() throws InterruptedException{

writeAccesses--;

if(writeAccesses == 0){

writingThread = null;

}

notifyAll();

}

private boolean canGrantWriteAccess(Thread callingThread){

**if(isOnlyReader(callingThread)) return true;**

if(hasReaders()) return false;

if(writingThread == null) return true;

if(!isWriter(callingThread)) return false;

return true;

}

private boolean hasReaders(){

return readingThreads.size() > 0;

}

private boolean isWriter(Thread callingThread){

return writingThread == callingThread;

}

**private boolean isOnlyReader(Thread thread){**

**return readers == 1 && readingThreads.get(callingThread) != null;**

**}**

}

Now the ReadWriteLock class is read-to-write access reentrant.

## Write to Read Reentrance

Sometimes a thread that has write access needs read access too. A writer should always be granted read access if requested. If a thread has write access no other threads can have read nor write access, so it is not dangerous. Here is how the canGrantReadAccess() method will look with that change:

public class ReadWriteLock{

private boolean canGrantReadAccess(Thread callingThread){

**if(isWriter(callingThread)) return true;**

if(writingThread != null) return false;

if(isReader(callingThread) return true;

if(writeRequests > 0) return false;

return true;

}

}

## Fully Reentrant ReadWriteLock

Below is the fully reentran ReadWriteLock implementation. I have made a few refactorings to the access conditions to make them easier to read, and thereby easier to convince yourself that they are correct.

public class ReadWriteLock{

private Map<Thread, Integer> readingThreads =

new HashMap<Thread, Integer>();

private int writeAccesses = 0;

private int writeRequests = 0;

private Thread writingThread = null;

public synchronized void lockRead() throws InterruptedException{

Thread callingThread = Thread.currentThread();

while(! canGrantReadAccess(callingThread)){

wait();

}

readingThreads.put(callingThread,

(getReadAccessCount(callingThread) + 1));

}

private boolean canGrantReadAccess(Thread callingThread){

if( isWriter(callingThread) ) return true;

if( hasWriter() ) return false;

if( isReader(callingThread) ) return true;

if( hasWriteRequests() ) return false;

return true;

}

public synchronized void unlockRead(){

Thread callingThread = Thread.currentThread();

if(!isReader(callingThread)){

throw new IllegalMonitorStateException("Calling Thread does not" +

" hold a read lock on this ReadWriteLock");

}

int accessCount = getReadAccessCount(callingThread);

if(accessCount == 1){ readingThreads.remove(callingThread); }

else { readingThreads.put(callingThread, (accessCount -1)); }

notifyAll();

}

public synchronized void lockWrite() throws InterruptedException{

writeRequests++;

Thread callingThread = Thread.currentThread();

while(! canGrantWriteAccess(callingThread)){

wait();

}

writeRequests--;

writeAccesses++;

writingThread = callingThread;

}

public synchronized void unlockWrite() throws InterruptedException{

if(!isWriter(Thread.currentThread()){

throw new IllegalMonitorStateException("Calling Thread does not" +

" hold the write lock on this ReadWriteLock");

}

writeAccesses--;

if(writeAccesses == 0){

writingThread = null;

}

notifyAll();

}

private boolean canGrantWriteAccess(Thread callingThread){

if(isOnlyReader(callingThread)) return true;

if(hasReaders()) return false;

if(writingThread == null) return true;

if(!isWriter(callingThread)) return false;

return true;

}

private int getReadAccessCount(Thread callingThread){

Integer accessCount = readingThreads.get(callingThread);

if(accessCount == null) return 0;

return accessCount.intValue();

}

private boolean hasReaders(){

return readingThreads.size() > 0;

}

private boolean isReader(Thread callingThread){

return readingThreads.get(callingThread) != null;

}

private boolean isOnlyReader(Thread callingThread){

return readingThreads.size() == 1 &&

readingThreads.get(callingThread) != null;

}

private boolean hasWriter(){

return writingThread != null;

}

private boolean isWriter(Thread callingThread){

return writingThread == callingThread;

}

private boolean hasWriteRequests(){

return this.writeRequests > 0;

}

}

## Calling unlock() From a finally-clause

When guarding a critical section with a ReadWriteLock, and the critical section may throw exceptions, it is important to call the readUnlock() and writeUnlock() methods from inside a finally-clause. Doing so makes sure that the ReadWriteLock is unlocked so other threads can lock it. Here is an example:

lock.lockWrite();

try{

//do critical section code, which may throw exception

} finally {

lock.unlockWrite();

}

This little construct makes sure that the ReadWriteLock is unlocked in case an exception is thrown from the code in the critical section. If unlockWrite() was not called from inside a finally-clause, and an exception was thrown from the critical section, the ReadWriteLock would remain write locked forever, causing all threads callinglockRead() or lockWrite() on that ReadWriteLock instance to halt indefinately. The only thing that could unlock the ReadWriteLockagain would be if the ReadWriteLock is reentrant, and the thread that had it locked when the exception was thrown, later succeeds in locking it, executing the critical section and callingunlockWrite() again afterwards. That would unlock the ReadWriteLock again. But why wait for that to happen,**if** it happens? Calling unlockWrite() from a finally-clause is a much more robust solution.

# Reentrance Lockout

Reentrance lockout is a situation similar to [deadlock](http://tutorials.jenkov.com/java-concurrency/deadlock.html) and [nested monitor lockout](http://tutorials.jenkov.com/java-concurrency/nested-monitor-lockout.html). Reentrance lockout is also covered in part in the texts on [Locks](http://tutorials.jenkov.com/java-concurrency/locks.html) and [Read / Write Locks](http://tutorials.jenkov.com/java-concurrency/read-write-locks.html).

Reentrance lockout may occur if a thread reenters a [Lock](http://tutorials.jenkov.com/java-concurrency/locks.html), [ReadWriteLock](http://tutorials.jenkov.com/java-concurrency/read-write-locks.html) or some other synchronizer that is not reentrant. Reentrant means that a thread that already holds a lock can retake it. Java's synchronized blocks are reentrant. Therefore the following code will work without problems:

public class Reentrant{

public synchronized outer(){

inner();

}

public synchronized inner(){

//do something

}

}

Notice how both outer() and inner() are declared synchronized, which in Java is equivalent to asynchronized(this) block. If a thread calls outer() there is no problem calling inner() from inside outer(), since both methods (or blocks) are synchronized on the same monitor object ("this"). If a thread already holds the lock on a monitor object, it has access to all blocks synchronized on the same monitor object. This is called reentrance. The thread can reenter any block of code for which it already holds the lock.

The following Lock implementation is not reentrant:

public class Lock{

private boolean isLocked = false;

public synchronized void lock()

throws InterruptedException{

while(isLocked){

wait();

}

isLocked = true;

}

public synchronized void unlock(){

isLocked = false;

notify();

}

}

If a thread calls lock() twice without calling unlock() in between, the second call to lock() will block. A reentrance lockout has occurred.

To avoid reentrance lockouts you have two options:

1. Avoid writing code that reenters locks
2. Use reentrant locks

Which of these options suit your project best depends on your concrete situation. Reentrant locks often don't perform as well as non-reentrant locks, and they are harder to implement, but this may not necessary be a problem in your case. Whether or not your code is easier to implement with or without lock reentrance must be determined case by case.

# Sempahores

A Semaphore is a thread synchronization construct that can be used either to send signals between threads to avoid[missed signals](http://tutorials.jenkov.com/java-concurrency/thread-signaling.html#missedsignals), or to guard a [critical section](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html) like you would with a [lock](http://tutorials.jenkov.com/java-concurrency/locks.html). Java 5 comes with semaphore implementations in the java.util.concurrent package so you don't have to implement your own semaphores. Still, it can be useful to know the theory behind their implementation and use.

Java 5 comes with a built-in Semaphore so you don't have to implement your own. You can read more about it in the[java.util.concurrent.Semaphore](http://tutorials.jenkov.com/java-util-concurrent/semaphore.html) text, in my java.util.concurrent tutorial.

## Simple Semaphore

Here is a simple Semaphore implementation:

public class Semaphore {

private boolean signal = false;

public synchronized void take() {

this.signal = true;

this.notify();

}

public synchronized void release() throws InterruptedException{

while(!this.signal) wait();

this.signal = false;

}

}

The take() method sends a signal which is stored internally in the Semaphore. The release() method waits for a signal. When received the signal flag is cleared again, and the release() method exited.

Using a semaphore like this you can avoid missed signals. You will call take() instead of notify() and release()instead of wait(). If the call to take() happens before the call to release() the thread calling release() will still know that take() was called, because the signal is stored internally in the signal variable. This is not the case with wait() and notify().

The names take() and release() may seem a bit odd when using a semaphore for signaling. The names origin from the use of semaphores as locks, as explained later in this text. In that case the names make more sense.

## Using Semaphores for Signaling

Here is a simplified example of two threads signaling each other using a Semaphore:

Semaphore semaphore = new Semaphore();

SendingThread sender = new SendingThread(semaphore);

ReceivingThread receiver = new ReceivingThread(semaphore);

receiver.start();

sender.start();

public class SendingThread {

Semaphore semaphore = null;

public SendingThread(Semaphore semaphore){

this.semaphore = semaphore;

}

public void run(){

while(true){

//do something, then signal

this.semaphore.take();

}

}

}

public class RecevingThread {

Semaphore semaphore = null;

public ReceivingThread(Semaphore semaphore){

this.semaphore = semaphore;

}

public void run(){

while(true){

this.semaphore.release();

//receive signal, then do something...

}

}

}

## Counting Semaphore

The Semaphore implementation in the previous section does not count the number of signals sent to it by take()method calls. We can change the Semaphore to do so. This is called a counting semaphore. Here is a simple implementation of a counting semaphore:

public class CountingSemaphore {

private int signals = 0;

public synchronized void take() {

this.signals++;

this.notify();

}

public synchronized void release() throws InterruptedException{

while(this.signals == 0) wait();

this.signals--;

}

}

## Bounded Semaphore

The CoutingSemaphore has no upper bound on how many signals it can store. We can change the semaphore implementation to have an upper bound, like this:

public class BoundedSemaphore {

private int signals = 0;

private int bound = 0;

public BoundedSemaphore(int upperBound){

this.bound = upperBound;

}

public synchronized void take() throws InterruptedException{

while(this.signals == bound) wait();

this.signals++;

this.notify();

}

public synchronized void release() throws InterruptedException{

while(this.signals == 0) wait();

this.signals--;

this.notify();

}

}

Notice how the take() method now blocks if the number of signals is equal to the upper bound. Not until a thread has called receive will the thread calling take() be allowed to deliver its signal, if the BoundedSemaphore has reached its upper signal limit.

## Using Semaphores as Locks

It is possible to use a bounded semaphore as a lock. To do so, set the upper bound to 1, and have the call to take()and release() guard the critical section. Here is an example:

BoundedSemaphore semaphore = new BoundedSemaphore(1);

...

semaphore.take();

try{

//critical section

} finally {

semaphore.release();

}

In contrast to the signaling use case the methods take() and release() are now called by the same thread. Since only one thread is allowed to take the semaphore, all other threads calling take() will be blocked until release() is called. The call to release() will never block since there has always been a call to take() first.

You can also use a bounded semaphore to limit the number of threads allowed into a section of code. For instance, in the example above, what would happen if you set the limit of the BoundedSemaphore to 5? 5 threads would be allowed to enter the critical section at a time. You would have to make sure though, that the thread operations do not conflict for these 5 threads, or you application will fail.

The relase() method is called from inside a finally-block to make sure it is called even if an exception is thrown from the critical section.

# Blocking Queues

A blocking queue is a queue that blocks when you try to dequeue from it and the queue is empty, or if you try to enqueue items to it and the queue is already full. A thread trying to dequeue from an empty queue is blocked until some other thread inserts an item into the queue. A thread trying to enqueue an item in a full queue is blocked until some other thread makes space in the queue, either by dequeuing one or more items or clearing the queue completely.

Here is a diagram showing two threads cooperating via a blocking queue:

|  |
| --- |
| A BlockingQueue with one thread putting into it, and another thread taking from it. |
| **A BlockingQueue with one thread putting into it, and another thread taking from it.** |

Java 5 comes with blocking queue implementations in the java.util.concurrent package. You can read about that class in my [java.util.concurrent.BlockingQueue](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html) tutorial. Even if Java 5 comes with a blocking queue implementation, it can be useful to know the theory behind their implementation.

## Blocking Queue Implementation

The implementation of a blocking queue looks similar to a [Bounded Semaphore](http://tutorials.jenkov.com/java-concurrency/semaphores.html#bounded). Here is a simple implementation of a blocking queue:

public class BlockingQueue {

private List queue = new LinkedList();

private int limit = 10;

public BlockingQueue(int limit){

this.limit = limit;

}

public synchronized void enqueue(Object item)

throws InterruptedException {

while(this.queue.size() == this.limit) {

wait();

}

if(this.queue.size() == 0) {

notifyAll();

}

this.queue.add(item);

}

public synchronized Object dequeue()

throws InterruptedException{

while(this.queue.size() == 0){

wait();

}

if(this.queue.size() == this.limit){

notifyAll();

}

return this.queue.remove(0);

}

}

Notice how notifyAll() is only called from enqueue() and dequeue() if the queue size is equal to the size bounds (0 or limit). If the queue size is not equal to either bound when enqueue() or dequeue() is called, there can be no threads waiting to either enqueue or dequeue items.

# Thread Pools

Thread Pools are useful when you need to limit the number of threads running in your application at the same time. There is a performance overhead associated with starting a new thread, and each thread is also allocated some memory for its stack etc.

Instead of starting a new thread for every task to execute concurrently, the task can be passed to a thread pool. As soon as the pool has any idle threads the task is assigned to one of them and executed. Internally the tasks are inserted into a [Blocking Queue](http://tutorials.jenkov.com/java-concurrency/blocking-queues.html) which the threads in the pool are dequeuing from. When a new task is inserted into the queue one of the idle threads will dequeue it successfully and execute it. The rest of the idle threads in the pool will be blocked waiting to dequeue tasks.

Thread pools are often used in multi threaded servers. Each connection arriving at the server via the network is wrapped as a task and passed on to a thread pool. The threads in the thread pool will process the requests on the connections concurrently. A later trail will get into detail about implementing multithreaded servers in Java.

Java 5 comes with built in thread pools in the java.util.concurrent package, so you don't have to implement your own thread pool. You can read more about it in my text on the [java.util.concurrent.ExecutorService](http://tutorials.jenkov.com/java-util-concurrent/executorservice.html). Still it can be useful to know a bit about the implementation of a thread pool anyways.

Here is a simple thread pool implementation:

public class ThreadPool {

private BlockingQueue taskQueue = null;

private List<PoolThread> threads = new ArrayList<PoolThread>();

private boolean isStopped = false;

public ThreadPool(int noOfThreads, int maxNoOfTasks){

taskQueue = new BlockingQueue(maxNoOfTasks);

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

threads.add(new PoolThread(taskQueue));

}

for(PoolThread thread : threads){

thread.start();

}

}

public void synchronized execute(Runnable task){

if(this.isStopped) throw

new IllegalStateException("ThreadPool is stopped");

this.taskQueue.enqueue(task);

}

public synchronized void stop(){

this.isStopped = true;

for(PoolThread thread : threads){

thread.stop();

}

}

}

public class PoolThread extends Thread {

private BlockingQueue taskQueue = null;

private boolean isStopped = false;

public PoolThread(BlockingQueue queue){

taskQueue = queue;

}

public void run(){

while(!isStopped()){

try{

Runnable runnable = (Runnable) taskQueue.dequeue();

runnable.run();

} catch(Exception e){

//log or otherwise report exception,

//but keep pool thread alive.

}

}

}

public synchronized void stop(){

isStopped = true;

this.interrupt(); //break pool thread out of dequeue() call.

}

public synchronized void isStopped(){

return isStopped;

}

}

The thread pool implementation consists of two parts. A ThreadPool class which is the public interface to the thread pool, and a PoolThread class which implements the threads that execute the tasks.

To execute a task the method ThreadPool.execute(Runnable r) is called with a Runnable implementation as parameter. The Runnable is enqueued in the [blocking queue](http://tutorials.jenkov.com/java-concurrency/blocking-queues.html) internally, waiting to be dequeued.

The Runnable will be dequeued by an idle PoolThread and executed. You can see this in thePoolThread.run() method. After execution the PoolThread loops and tries to dequeue a task again, until stopped.

To stop the ThreadPool the method ThreadPool.stop() is called. The stop called is noted internally in theisStopped member. Then each thread in the pool is stopped by calling PoolThread.stop(). Notice how theexecute() method will throw an IllegalStateException if execute() is called after stop() has been called.

The threads will stop after finishing any task they are currently executing. Notice the this.interrupt() call inPoolThread.stop(). This makes sure that a thread blocked in a wait() call inside the taskQueue.dequeue()call breaks out of the wait() call, and leaves the dequeue() method call with an InterruptedExceptionthrown. This exception is caught in the PoolThread.run() method, reported, and then the isStopped variable is checked. Since isStopped is now true, the PoolThread.run() will exit and the thread dies.

# Anatomy of a Synchronizer

Even if many synchronizers (locks, semaphores, blocking queue etc.) are different in function, they are often not that different in their internal design. In other words, they consist of the same (or similar) basic parts internally. Knowing these basic parts can be a great help when designing synchronizers. It is these parts this text looks closer at.

**Note:** The content of this text is a part result of a M.Sc. student project at the IT University of Copenhagen in the spring 2004 by Jakob Jenkov, Toke Johansen and Lars Bjørn. During this project we asked Doug Lea if he knew of similar work. Interestingly he had come up with similar conclusions independently of this project during the development of the Java 5 concurrency utilities. Doug Lea's work, I believe, is described in the book ["Java Concurrency in Practice"](http://www.amazon.com/Java-Concurrency-Practice-Brian-Goetz/dp/0321349601/ref=pd_bbs_sr_1?ie=UTF8&s=books&qid=1215418711&sr=8-1). This book also contains a chapter with the title "Anatomy of a Synchronizer" with content similar to this text, though not exactly the same.

The purpose of most (if not all) synchronizers is to guard some area of the code (critical section) from concurrent access by threads. To do this the following parts are often needed in a synchronizer:

1. [State](http://tutorials.jenkov.com/java-concurrency/anatomy-of-a-synchronizer.html#state)
2. [Access Condition](http://tutorials.jenkov.com/java-concurrency/anatomy-of-a-synchronizer.html#accesscondition)
3. [State Changes](http://tutorials.jenkov.com/java-concurrency/anatomy-of-a-synchronizer.html#statechanges)
4. [Notification Strategy](http://tutorials.jenkov.com/java-concurrency/anatomy-of-a-synchronizer.html#notificationstrategy)
5. [Test and Set Method](http://tutorials.jenkov.com/java-concurrency/anatomy-of-a-synchronizer.html#testandset)
6. [Set Method](http://tutorials.jenkov.com/java-concurrency/anatomy-of-a-synchronizer.html#set)

Not all synchronizers have all of these parts, and those that have may not have them exactly as they are described here. Usually you can find one or more of these parts, though.

## State

The state of a synchronizer is used by the access condition to determine if a thread can be granted access. In a [Lock](http://tutorials.jenkov.com/java-concurrency/locks.html)the state is kept in a boolean saying whether the Lock is locked or not. In a [Bounded Semaphore](http://tutorials.jenkov.com/java-concurrency/semaphores.html#bounded) the internal state is kept in a counter (int) and an upper bound (int) which state the current number of "takes" and the maximum number of "takes". In a [Blocking Queue](http://tutorials.jenkov.com/java-concurrency/blocking-queues.html) the state is kept in the List of elements in the queue and the maximum queue size (int) member (if any).

Here are two code snippets from both Lock and a BoundedSemaphore. The state code is marked in bold.

public class Lock{

**//state is kept here**

**private boolean isLocked = false;**

public synchronized void lock()

throws InterruptedException{

while(isLocked){

wait();

}

isLocked = true;

}

...

}

public class BoundedSemaphore {

**//state is kept here**

**private int signals = 0;**

**private int bound = 0;**

public BoundedSemaphore(int upperBound){

this.bound = upperBound;

}

public synchronized void take() throws InterruptedException{

while(this.signals == bound) wait();

this.signal++;

this.notify();

}

...

}

## Access Condition

The access conditions is what determines if a thread calling a test-and-set-state method can be allowed to set the state or not. The access condition is typically based on the [state](http://tutorials.jenkov.com/java-concurrency/anatomy-of-a-synchronizer.html#state) of the synchronizer. The access condition is typically checked in a while loop to guard against [Spurious Wakeups](http://tutorials.jenkov.com/java-concurrency/thread-signaling.html#spuriouswakeups). When the access condition is evaluated it is either true or false.

In a [Lock](http://tutorials.jenkov.com/java-concurrency/locks.html) the access condition simply checks the value of the isLocked member variable. In a [Bounded Semaphore](http://tutorials.jenkov.com/java-concurrency/semaphores.html#bounded)there are actually two access conditions depending on whether you are trying to "take" or "release" the semaphore. If a thread tries to take the semaphore the signals variable is checked against the upper bound. If a thread tries to release the semaphore the signals variable is checked against 0.

Here are two code snippets of a Lock and a BoundedSemaphore with the access condition marked in bold. Notice how the conditions is always checked inside a while loop.

public class Lock{

private boolean isLocked = false;

public synchronized void lock()

throws InterruptedException{

**//access condition**

while(**isLocked**){

wait();

}

isLocked = true;

}

...

}

public class BoundedSemaphore {

private int signals = 0;

private int bound = 0;

public BoundedSemaphore(int upperBound){

this.bound = upperBound;

}

public synchronized void take() throws InterruptedException{

**//access condition**

while(**this.signals == bound**) wait();

this.signals++;

this.notify();

}

public synchronized void release() throws InterruptedException{

**//access condition**

while(**this.signals == 0**) wait();

this.signals--;

this.notify();

}

}

## State Changes

Once a thread gains access to the critical section it has to change the state of the synchronizer to (possibly) block other threads from entering it. In other words, the state needs to reflect the fact that a thread is now executing inside the critical section. This should affect the access conditions of other threads attempting to gain access.

In a [Lock](http://tutorials.jenkov.com/java-concurrency/locks.html) the state change is the code setting isLocked = true. In a semaphore it is either the code signals--or signals++;

Here are two code snippets with the state change code marked in bold:

public class Lock{

private boolean isLocked = false;

public synchronized void lock()

throws InterruptedException{

while(isLocked){

wait();

}

**//state change**

**isLocked = true;**

}

public synchronized void unlock(){

**//state change**

**isLocked = false;**

notify();

}

}

public class BoundedSemaphore {

private int signals = 0;

private int bound = 0;

public BoundedSemaphore(int upperBound){

this.bound = upperBound;

}

public synchronized void take() throws InterruptedException{

while(this.signals == bound) wait();

**//state change**

**this.signals++;**

this.notify();

}

public synchronized void release() throws InterruptedException{

while(this.signals == 0) wait();

**//state change**

**this.signals--;**

this.notify();

}

}

## Notification Strategy

Once a thread has changed the state of a synchronizer it may sometimes need to notify other waiting threads about the state change. Perhaps this state change might turn the access condition true for other threads.

Notification Strategies typically fall into three categories.

1. Notify all waiting threads.
2. Notify 1 random of N waiting threads.
3. Notify 1 specific of N waiting thread.

Notifying all waiting threads is pretty easy. All waiting threads call wait() on the same object. Once a thread want to notify the waiting threads it calls notifyAll() on the object the waiting threads called wait() on.

Notifying a single random waiting thread is also pretty easy. Just have the notifying thread call notify() on the object the waiting threads have called wait() on. Calling notify makes no guarantee about which of the waiting threads will be notified. Hence the term "random waiting thread".

Sometimes you may need to notify a specific rather than a random waiting thread. For instance if you need to guarantee that waiting threads are notified in a specific order, be it the order they called the synchronizer in, or some prioritized order. To achive this each waiting thread must call wait() on its own, separate object. When the notifying thread wants to notify a specific waiting thread it will call notify() on the object this specific thread has calledwait() on. An example of this can be found in the text [Starvation and Fairness](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html).

Below is a code snippet with the notification strategy (notify 1 random waiting thread) marked in bold:

public class Lock{

private boolean isLocked = false;

public synchronized void lock()

throws InterruptedException{

while(isLocked){

**//wait strategy - related to notification strategy**

**wait();**

}

isLocked = true;

}

public synchronized void unlock(){

isLocked = false;

**notify(); //notification strategy**

}

}

## Test and Set Method

Synchronizer most often have two types of methods of which test-and-set is the first type ([set](http://tutorials.jenkov.com/java-concurrency/anatomy-of-a-synchronizer.html#set) is the other). Test-and-set means that the thread calling this method **tests** the internal state of the synchronizer against the access condition. If the condition is met the thread **sets** the internal state of the synchronizer to reflect that the thread has gained access.

The state transition usually results in the access condition turning false for other threads trying to gain access, but may not always do so. For instance, in a [Read - Write Lock](http://tutorials.jenkov.com/java-concurrency/read-write-locks.html) a thread gaining read access will update the state of the read-write lock to reflect this, but other threads requesting read access will also be granted access as long as no threads has requested write access.

It is imperative that the test-and-set operations are executed atomically meaning no other threads are allowed to execute in the test-and-set method in between the test and the setting of the state.

The program flow of a test-and-set method is usually something along the lines of:

1. Set state before test if necessary
2. Test state against access condition
3. If access condition is not met, wait
4. If access condition is met, set state, and notify waiting threads if necessary

The lockWrite() method of a [ReadWriteLock](http://tutorials.jenkov.com/java-concurrency/read-write-locks.html) class shown below is an example of a test-and-set method. Threads calling lockWrite() first sets the state before the test (writeRequests++). Then it tests the internal state against the access condition in the canGrantWriteAccess() method. If the test succeeds the internal state is set again before the method is exited. Notice that this method does not notify waiting threads.

public class ReadWriteLock{

private Map<Thread, Integer> readingThreads =

new HashMap<Thread, Integer>();

private int writeAccesses = 0;

private int writeRequests = 0;

private Thread writingThread = null;

...

**public synchronized void lockWrite() throws InterruptedException{**

**writeRequests++;**

**Thread callingThread = Thread.currentThread();**

**while(! canGrantWriteAccess(callingThread)){**

**wait();**

**}**

**writeRequests--;**

**writeAccesses++;**

**writingThread = callingThread;**

**}**

...

}

The BoundedSemaphore class shown below has two test-and-set methods: take() and   
release(). Both methods test and sets the internal state.

public class BoundedSemaphore {

private int signals = 0;

private int bound = 0;

public BoundedSemaphore(int upperBound){

this.bound = upperBound;

}

**public synchronized void take() throws InterruptedException{**

**while(this.signals == bound) wait();**

**this.signals++;**

**this.notify();**

**}**

**public synchronized void release() throws InterruptedException{**

**while(this.signals == 0) wait();**

**this.signals--;**

**this.notify();**

**}**

}

## Set Method

The set method is the second type of method that synchronizers often contain. The set method just sets the internal state of the synchronizer without testing it first. A typical example of a set method is the unlock() method of aLock class. A thread holding the lock can always unlock it without having to test if the Lock is unlocked.

The program flow of a set method is usually along the lines of:

1. Set internal state
2. Notify waiting threads

Here is an example unlock() method:

public class Lock{

private boolean isLocked = false;

**public synchronized void unlock(){**

**isLocked = false;**

**notify();**

**}**

}

# java.util.concurrent - Java Concurrency Utilities

# BlockingQueue

The Java BlockingQueue interface in the java.util.concurrent class represents a queue which is thread safe to put into, and take instances from. In this text I will show you how to use this BlockingQueue.

This text will not discuss how to implement a BlockingQueue in Java yourself. If you are interested in that, I have a text on [Blocking Queues](http://tutorials.jenkov.com/java-concurrency/blocking-queues.html) in my more theoretical [Java Concurrency Tutorial](http://tutorials.jenkov.com/java-concurrency/index.html).

## BlockingQueue Usage

A BlockingQueue is typically used to have on thread produce objects, which another thread consumes. Here is a diagram that illustrates this principle:

|  |
| --- |
| A BlockingQueue with one thread putting into it, and another thread taking from it. |
| **A BlockingQueue with one thread putting into it, and another thread taking from it.** |

The producing thread will keep producing new objects and insert them into the queue, until the queue reaches some upper bound on what it can contain. It's limit, in other words. If the blocking queue reaches its upper limit, the producing thread is blocked while trying to insert the new object. It remains blocked until a consuming thread takes an object out of the queue.

The consuming thread keeps taking objects out of the blocking queue, and processes them. If the consuming thread tries to take an object out of an empty queue, the consuming thread is blocked until a producing thread puts an object into the queue.

### BlockingQueue Methods

A BlockingQueue has 4 different sets of methods for inserting, removing and examining the elements in the queue. Each set of methods behaves differently in case the requested operation cannot be carried out immediately. Here is a table of the methods:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Throws Exception** | **Special Value** | **Blocks** | **Times Out** |
| **Insert** | add(o) | offer(o) | put(o) | offer(o, timeout, timeunit) |
| **Remove** | remove(o) | poll(o) | take(o) | poll(timeout, timeunit) |
| **Examine** | element(o) | peek(o) |  |  |

The 4 different sets of behaviour means this:

1. **Throws Exception**:   
   If the attempted operation is not possible immediately, an exception is thrown.
2. **Special Value**:   
   If the attempted operation is not possible immediately, a special value is returned (often true / false).
3. **Blocks**:   
   If the attempted operation is not possible immedidately, the method call blocks until it is.
4. **Times Out**:   
   If the attempted operation is not possible immedidately, the method call blocks until it is, but waits no longer than the given timeout. Returns a special value telling whether the operation succeeded or not (typically true / false).

It is not possible to insert null into a BlockingQueue. If you try to insert null, the BlockingQueue will throw aNullPointerException.

It is also possible to access all the elements inside a BlockingQueue, and not just the elements at the start and end. For instance, say you have queued an object for processing, but your application decides to cancel it. You can then call e.g. remove(o) to remove a specific object in the queue. However, this is not done very efficiently, so you should not use these Collection methods unless you really have to.

## BlockingQueue Implementations

Since BlockingQueue is an interface, you need to use one of its implementations to use it. Thejava.util.concurrent package has the following implementations of the BlockingQueue interface (in Java 6):

* [ArrayBlockingQueue](http://tutorials.jenkov.com/java-util-concurrent/arrayblockingqueue.html)
* [DelayQueue](http://tutorials.jenkov.com/java-util-concurrent/delayqueue.html)
* [LinkedBlockingQueue](http://tutorials.jenkov.com/java-util-concurrent/linkedblockingqueue.html)
* [PriorityBlockingQueue](http://tutorials.jenkov.com/java-util-concurrent/priorityblockingqueue.html)
* [SynchronousQueue](http://tutorials.jenkov.com/java-util-concurrent/synchronousqueue.html)

Click the links in the list to read more about each implementation. If a link cannot be clicked, that implementation has not yet been described. Check back again in the future, or check out the JavaDoc's for more detail.

## Java BlockingQueue Example

Here is a Java BlockingQueue example. The example uses the ArrayBlockingQueue implementation of theBlockingQueue interface.

First, the BlockingQueueExample class which starts a Producer and a Consumer in separate threads. TheProducer inserts strings into a shared BlockingQueue, and the Consumer takes them out.

public class BlockingQueueExample {

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

BlockingQueue queue = new ArrayBlockingQueue(1024);

Producer producer = new Producer(queue);

Consumer consumer = new Consumer(queue);

new Thread(producer).start();

new Thread(consumer).start();

Thread.sleep(4000);

}

}

Here is the Producer class. Notice how it sleeps a second between each put() call. This will cause the Consumer to block, while waiting for objects in the queue.

public class Producer implements Runnable{

protected BlockingQueue queue = null;

public Producer(BlockingQueue queue) {

this.queue = queue;

}

public void run() {

try {

queue.put("1");

Thread.sleep(1000);

queue.put("2");

Thread.sleep(1000);

queue.put("3");

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

Here is the Consumer class. It just takes out the objects from the queue, and prints them to System.out.

public class Consumer implements Runnable{

protected BlockingQueue queue = null;

public Consumer(BlockingQueue queue) {

this.queue = queue;

}

public void run() {

try {

System.out.println(queue.take());

System.out.println(queue.take());

System.out.println(queue.take());

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

# ArrayBlockingQueue

The ArrayBlockingQueue class implements the [BlockingQueue](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html) interface. Read the [BlockingQueue](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html) text for more information about the interface.

ArrayBlockingQueue is a bounded, blocking queue that stores the elements internally in an array. That it is bounded means that it cannot store unlimited amounts of elements. There is an upper bound on the number of elements it can store at the same time. You set the upper bound at instantiation time, and after that it cannot be changed.

The ArrayBlockingQueue stores the elements internally in FIFO (First In, First Out) order. The head of the queue is the element which has been in queue the longest time, and the tail of the queue is the element which has been in the queue the shortest time.

Here is how to instantiate and use an ArrayBlockingQueue:

BlockingQueue queue = new ArrayBlockingQueue(1024);

queue.put("1");

Object object = queue.take();

Here is a BlockingQueue example that uses Java Generics. Notice how you can put and take String's instead of :

BlockingQueue<String> queue = new ArrayBlockingQueue<String>(1024);

queue.put("1");

String string = queue.take();

# DelayQueue

DelayQueue class implements the [BlockingQueue](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html) interface. Read the [BlockingQueue](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html) text for more information about the interface.

The DelayQueue keeps the elements internally until a certain delay has expired. The elements must implement the interface java.util.concurrent.Delayed. Here is how the interface looks:

public interface Delayed extends Comparable<Delayed< {

public long getDelay(TimeUnit timeUnit);

}

The value returned by the getDelay() method should be the delay remaining before this element can be released. If 0 or a negative value is returned, the delay will be considered expired, and the element released at the nexttake() etc. call on the DelayQueue.

The TimeUnit instance passed to the getDelay() method is an Enum that tells which time unit the delay should be returned in. The TimeUnit enum can take these values:

DAYS

HOURS

MINUTES

SECONDS

MILLISECONDS

MICROSECONDS

NANOSECONDS

The Delayed interface also extends the java.lang.Comparable interface, as you can see, which means thatDelayed objects can be compared to each other. This is probably used internally in the DelayQueue to order the elements in the queue, so they are released ordered by their expiration time.

Here is an example of how to use the DelayQueue:

public class DelayQueueExample {

public static void main(String[] args) {

DelayQueue queue = new DelayQueue();

Delayed element1 = new DelayedElement();

queue.put(element1);

Delayed element2 = queue.take();

}

}

The DelayedElement is an implementation of the Delayed interface that I have created. It is not part of thejava.util.concurrent package. You will have to create your own implementation of the Delayed interface to use the DelayQueue class.

# LinkedBlockingQueue

The LinkedBlockingQueue class implements the [BlockingQueue](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html) interface. Read the [BlockingQueue](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html) text for more information about the interface.

The LinkedBlockingQueue keeps the elements internally in a linked structure (linked nodes). This linked structure can optionally have an upper bound if desired. If no upper bound is specified, Integer.MAX\_VALUE is used as the upper bound.

The LinkedBlockingQueue stores the elements internally in FIFO (First In, First Out) order. The head of the queue is the element which has been in queue the longest time, and the tail of the queue is the element which has been in the queue the shortest time.

Here is how to instantiate and use a LinkedBlockingQueue:

BlockingQueue<String> unbounded = new LinkedBlockingQueue<String>();

BlockingQueue<String> bounded = new LinkedBlockingQueue<String>(1024);

bounded.put("Value");

String value = bounded.take();

# PriorityBlockingQueue

The PriorityBlockingQueue class implements the [BlockingQueue](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html) interface. Read the [BlockingQueue](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html) text for more information about the interface.

The PriorityBlockingQueue is an unbounded concurrent queue. It uses the same ordering rules as thejava.util.PriorityQueue class. You cannot insert null into this queue.

All elements inserted into the PriorityBlockingQueue must implement the java.lang.Comparableinterface. The elements thus order themselves according to whatever priority you decide in your Comparableimplementation.

Notice that the PriorityBlockingQueue does not enforce any specific behaviour for elements that have equal priority (compare() == 0).

Also notice, that in case you obtain an Iterator from a PriorityBlockingQueue, the Iterator does not guarantee to iterate the elements in priority order.

Here is an example of how to use the PriorityBlockingQueue:

BlockingQueue queue = new PriorityBlockingQueue();

//String implements java.lang.Comparable

queue.put("Value");

String value = queue.take();

# SynchronousQueue

The SynchronousQueue class implements the [BlockingQueue](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html) interface. Read the [BlockingQueue](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html) text for more information about the interface.

The SynchronousQueue is a queue that can only contain a single element internally. A thread inseting an element into the queue is blocked until another thread takes that element from the queue. Likewise, if a thread tries to take an element and no element is currently present, that thread is blocked until a thread insert an element into the queue.

Calling this class a queue is a bit of an overstatement. It's more of a rendesvouz point.

# BlockingDeque

The BlockingDeque interface in the java.util.concurrent class represents a deque which is thread safe to put into, and take instances from. In this text I will show you how to use this BlockingDeque.

The BlockingDeque class is a Deque which blocks threads tring to insert or remove elements from the deque, in case it is either not possible to insert or remove elements from the deque.

A deque is short for "Double Ended Queue". Thus, a deque is a queue which you can insert and take elements from, from both ends.

## BlockingDeque Usage

A BlockingDeque could be used if threads are both producing and consuming elements of the same queue. It could also just be used if the producting thread needs to insert at both ends of the queue, and the consuming thread needs to remove from both ends of the queue. Here is an illustration of that:

|  |
| --- |
| A BlockingDeque - threads can put and take from both ends of the deque. |
| **A BlockingDeque - threads can put and take from both ends of the deque.** |

A thread will produce elements and insert them into either end of the queue. If the deque is currently full, the inserting thread will be blocked until a removing thread takes an element out of the deque. If the deque is currently empty, a removing thread will be blocked until an inserting thread inserts an element into the deque.

### BlockingDeque methods

A BlockingDeque has 4 different sets of methods for inserting, removing and examining the elements in the deque. Each set of methods behaves differently in case the requested operation cannot be carried out immediately. Here is a table of the methods:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Throws Exception** | **Special Value** | **Blocks** | **Times Out** |
| **Insert** | addFirst(o) | offerFirst(o) | putFirst(o) | offerFirst(o, timeout, timeunit) |
| **Remove** | removeFirst(o) | pollFirst(o) | takeFirst(o) | pollFirst(timeout, timeunit) |
| **Examine** | getFirst(o) | peekFirst(o) |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Throws Exception** | **Special Value** | **Blocks** | **Times Out** |
| **Insert** | addLast(o) | offerLast(o) | putLast(o) | offerLast(o, timeout, timeunit) |
| **Remove** | removeLast(o) | pollLast(o) | takeLast(o) | pollLast(timeout, timeunit) |
| **Examine** | getLast(o) | peekLast(o) |  |  |

The 4 different sets of behaviour means this:

1. **Throws Exception**:   
   If the attempted operation is not possible immediately, an exception is thrown.
2. **Special Value**:   
   If the attempted operation is not possible immediately, a special value is returned (often true / false).
3. **Blocks**:   
   If the attempted operation is not possible immedidately, the method call blocks until it is.
4. **Times Out**:   
   If the attempted operation is not possible immedidately, the method call blocks until it is, but waits no longer than the given timeout. Returns a special value telling whether the operation succeeded or not (typically true / false).

## BlockingDeque Extends BlockingQueue

The BlockingDeque interface extends the BlockingQueue interface. That means that you can use aBlockingDeque as a BlockingQueue. If you do so, the various inserting methods will add the elements to the end of the deque, and the removing methods will remove the elements from the beginning of the deque. The inserting and removing methods of the BlockingQueue interface, that is.

Here is a table of what the methods of the BlockingQueue does in a BlockingDeque implementation:

|  |  |
| --- | --- |
| **BlockingQueue** | **BlockingDeque** |
| add() | addLast() |
| offer() x 2 | offerLast() x 2 |
| put() | putLast() |
|  |  |
| remove() | removeFirst() |
| poll() x 2 | pollFirst() |
| take() | takeFirst() |
|  |  |
| element() | getFirst() |
| peek() | peekFirst() |

## BlockingDeque Implementations

Since BlockingDeque is an interface, you need to use one of its many implementations to use it. Thejava.util.concurrent package has the following implementations of the BlockingDeque interface:

* [LinkedBlockingDeque](http://tutorials.jenkov.com/java-util-concurrent/linkedblockingdeque.html)

## BlockingDeque Code Example

Here is a small code example of how to use the BlockingDeque methods:

BlockingDeque<String> deque = new LinkedBlockingDeque<String>();

deque.addFirst("1");

deque.addLast("2");

String two = deque.takeLast();

String one = deque.takeFirst();

# LinkedBlockingDeque

The LinkedBlockingDeque class implements the [BlockingDeque](http://tutorials.jenkov.com/java-util-concurrent/blockingdeque.html) interface. Read the [BlockingDeque](http://tutorials.jenkov.com/java-util-concurrent/blockingdeque.html) text for more information about the interface.

The word Deque comes from the term "Double Ended Queue". A Deque is thus a queue where you can insert and remove elements from both ends of the queue.

The LinkedBlockingDeque is a Deque which will block if a thread attempts to take elements out of it while it is empty, regardless of what end the thread is attempting to take elements from.

Here is how to instantiate and use a LinkedBlockingDeque:

BlockingDeque<String> deque = new LinkedBlockingDeque<String>();

deque.addFirst("1");

deque.addLast("2");

String two = deque.takeLast();

String one = deque.takeFirst();

# ConcurrentMap

# java.util.concurrent.ConcurrentMap

The java.util.concurrent.ConcurrentMap interface represents a [Map](http://tutorials.jenkov.com/java-collections/map.html) which is capable of handling concurrent access (puts and gets) to it.

The ConcurrentMap has a few extra atomic methods in addition to the methods it inherits from its superinterface,[java.util.Map](http://tutorials.jenkov.com/java-collections/map.html).

## ConcurrentMap Implementations

Since ConcurrentMap is an interface, you need to use one of its implementations in order to use it. Thejava.util.concurrent package contains the following implementations of the ConcurrentMap interface:

* ConcurrentHashMap

### ConcurrentHashMap

The ConcurrentHashMap is very similar to the java.util.HashTable class, except that ConcurrentHashMapoffers better concurrency than HashTable does. ConcurrentHashMap does not lock the Map while you are reading from it. Additionally, ConcurrentHashMap does not lock the entire Map when writing to it. It only locks the part of the Map that is being written to, internally.

Another difference is that ConcurrentHashMap does not throw ConcurrentModificationException if theConcurrentHashMap is changed while being iterated. The Iterator is not designed to be used by more than one thread though.

Checkout the official JavaDoc for more details about ConcurrentMap and ConcurrentHashMap.

## ConcurrentMap Example

Here is an example of how to use the ConcurrentMap interface. The example uses a ConcurrentHashMapimplementation:

ConcurrentMap concurrentMap = new ConcurrentHashMap();

concurrentMap.put("key", "value");

Object value = concurrentMap.get("key");

# CountDownLatch

A java.util.concurrent.CountDownLatch is a concurrency construct that allows one or more threads to wait for a given set of operations to complete.

A CountDownLatch is initialized with a given count. This count is decremented by calls to the countDown()method. Threads waiting for this count to reach zero can call one of the await() methods. Calling await() blocks the thread until the count reaches zero.

Below is a simple example. After the Decrementer has called countDown() 3 times on the CountDownLatch, the waiting Waiter is released from the await() call.

CountDownLatch latch = new CountDownLatch(3);

Waiter waiter = new Waiter(latch);

Decrementer decrementer = new Decrementer(latch);

new Thread(waiter) .start();

new Thread(decrementer).start();

Thread.sleep(4000);

public class Waiter implements Runnable{

CountDownLatch latch = null;

public Waiter(CountDownLatch latch) {

this.latch = latch;

}

public void run() {

try {

latch.await();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Waiter Released");

}

}

public class Decrementer implements Runnable {

CountDownLatch latch = null;

public Decrementer(CountDownLatch latch) {

this.latch = latch;

}

public void run() {

try {

Thread.sleep(1000);

this.latch.countDown();

Thread.sleep(1000);

this.latch.countDown();

Thread.sleep(1000);

this.latch.countDown();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

# CyclicBarrier

The java.util.concurrent.CyclicBarrier class is a synchronization mechanism that can synchronize threads progressing through some algorithm. In other words, it is a barrier that all threads must wait at, until all threads reach it, before any of the threads can continue. Here is a diagram illustrating that:

|  |
| --- |
| Two threads waiting for each other at CyclicBarriers. |
| **Two threads waiting for each other at CyclicBarriers.** |

The threads wait for each other by calling the await() method on the CyclicBarrier. Once N threads are waiting at the CyclicBarrier, all threads are released and can continue running.

## Creating a CyclicBarrier

When you create a CyclicBarrier you specify how many threads are to wait at it, before releasing them. Here is how you create a CyclicBarrier:

CyclicBarrier barrier = new CyclicBarrier(2);

## Waiting at a CyclicBarrier

Here is how a thread waits at a CyclicBarrier:

barrier.await();

You can also specify a timeout for the waiting thread. When the timeout has passed the thread is also released, even if not all N threads are waiting at the CyclicBarrier. Here is how you specify a timeout:

barrier.await(10, TimeUnit.SECONDS);

The waiting threads waits at the CyclicBarrier until either:

* The last thread arrives (calls await() )
* The thread is interrupted by another thread (another thread calls its interrupt() method)
* Another waiting thread is interrupted
* Another waiting thread times out while waiting at the CyclicBarrier
* The CyclicBarrier.reset() method is called by some external thread.

## CyclicBarrier Action

The CyclicBarrier supports a barrier action, which is a Runnable that is executed once the last thread arrives. You pass the Runnable barrier action to the CyclicBarrier in its constructor, like this:

Runnable barrierAction = ... ;

CyclicBarrier barrier = new CyclicBarrier(2, barrierAction);

## CyclicBarrier Example

Here is a code example that shows you how to use a CyclicBarrier:

Runnable barrier1Action = new Runnable() {

public void run() {

System.out.println("BarrierAction 1 executed ");

}

};

Runnable barrier2Action = new Runnable() {

public void run() {

System.out.println("BarrierAction 2 executed ");

}

};

CyclicBarrier barrier1 = new CyclicBarrier(2, barrier1Action);

CyclicBarrier barrier2 = new CyclicBarrier(2, barrier2Action);

CyclicBarrierRunnable barrierRunnable1 =

new CyclicBarrierRunnable(barrier1, barrier2);

CyclicBarrierRunnable barrierRunnable2 =

new CyclicBarrierRunnable(barrier1, barrier2);

new Thread(barrierRunnable1).start();

new Thread(barrierRunnable2).start();

Here is the CyclicBarrierRunnable class:

public class CyclicBarrierRunnable implements Runnable{

CyclicBarrier barrier1 = null;

CyclicBarrier barrier2 = null;

public CyclicBarrierRunnable(

CyclicBarrier barrier1,

CyclicBarrier barrier2) {

this.barrier1 = barrier1;

this.barrier2 = barrier2;

}

public void run() {

try {

Thread.sleep(1000);

System.out.println(Thread.currentThread().getName() +

" waiting at barrier 1");

this.barrier1.await();

Thread.sleep(1000);

System.out.println(Thread.currentThread().getName() +

" waiting at barrier 2");

this.barrier2.await();

System.out.println(Thread.currentThread().getName() +

" done!");

} catch (InterruptedException e) {

e.printStackTrace();

} catch (BrokenBarrierException e) {

e.printStackTrace();

}

}

}

Here is the console output for an execution of the above code. Note that the sequence in which the threads gets to write to the console may vary from execution to execution. Sometimes Thread-0 prints first, sometimes Thread-1prints first etc.

Thread-0 waiting at barrier 1

Thread-1 waiting at barrier 1

BarrierAction 1 executed

Thread-1 waiting at barrier 2

Thread-0 waiting at barrier 2

BarrierAction 2 executed

Thread-0 done!

Thread-1 done!

**Exchanger**

The java.util.concurrent.Exchanger class represents a kind of rendezvous point where two threads can exchange objects. Here is an illustration of this mechanism:

|  |
| --- |
| Two threads exchanging objects via an Exchanger. |
| **Two threads exchanging objects via an Exchanger.** |

Exchanging objects is done via one of the two exchange() methods. Here is an example:

Exchanger exchanger = new Exchanger();

ExchangerRunnable exchangerRunnable1 =

new ExchangerRunnable(exchanger, "A");

ExchangerRunnable exchangerRunnable2 =

new ExchangerRunnable(exchanger, "B");

new Thread(exchangerRunnable1).start();

new Thread(exchangerRunnable2).start();

Here is the ExchangerRunnable code:

public class ExchangerRunnable implements Runnable{

Exchanger exchanger = null;

Object object = null;

public ExchangerRunnable(Exchanger exchanger, Object object) {

this.exchanger = exchanger;

this.object = object;

}

public void run() {

try {

Object previous = this.object;

this.object = this.exchanger.exchange(this.object);

System.out.println(

Thread.currentThread().getName() +

" exchanged " + previous + " for " + this.object

);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

This example prints out this:

Thread-0 exchanged A for B

Thread-1 exchanged B for A

# Semaphore

The java.util.concurrent.Semaphore class is a [counting semaphore](http://tutorials.jenkov.com/java-concurrency/semaphores.html#counting). That means that it has two main methods:

* acquire()
* release()

The counting semaphore is initialized with a given number of "permits". For each call to acquire() a permit is taken by the calling thread. For each call to release() a permit is returned to the semaphore. Thus, at most N threads can pass the acquire() method without any release() calls, where N is the number of permits the semaphore was initialized with. The permits are just a simple counter. Nothing fancy here.

## Semaphore Usage

As semaphore typically has two uses:

1. To guard a critical section against entry by more than N threads at a time.
2. To send signals between two threads.

### Guarding Critical Sections

If you use a semaphore to guard a critical section, the thread trying to enter the critical section will typically first try to acquire a permit, enter the critical section, and then release the permit again after. Like this:

Semaphore semaphore = new Semaphore(1);

//critical section

semaphore.acquire();

...

semaphore.release();

### Sending Signals Between Threads

If you use a semaphore to send signals between threads, then you would typically have one thread call theacquire() method, and the other thread to call the release() method.

If no permits are available, the acquire() call will block until a permit is released by another thread. Similarly, arelease() calls is blocked if no more permits can be released into this semaphore.

Thus it is possible to coordinate threads. For instance, if acquire was called after Thread 1 had inserted an object in a shared list, and Thread 2 had called release() just before taking an object from that list, you had essentially created a blocking queue. The number of permits available in the semaphore would correspond to the maximum number of elements the blocking queue could hold.

## Fairness

No guarantees are made about [fairness](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html) of the threads acquiring permits from the Semaphore. That is, there is no guarantee that the first thread to call acquire() is also the first thread to obtain a permit. If the first thread is blocked waiting for a permit, then a second thread checking for a permit just as a permit is released, may actually obtain the permit ahead of thread 1.

If you want to enforce fairness, the Semaphore class has a constructor that takes a boolean telling if the semaphore should enforce fairness. Enforcing fairness comes at a performance / concurrency penalty, so don't enable it unless you need it.

Here is how to create a Semaphore in fair mode:

Semaphore semaphore = new Semaphore(1, true);

## More Methods

The java.util.concurrent.Semaphore class has lots more methods. For instance:

* availablePermits()
* acquireUninterruptibly()
* drainPermits()
* hasQueuedThreads()
* getQueuedThreads()
* tryAcquire()
* etc.

Check out the JavaDoc for more details on these methods.

# ExecutorService

The java.util.concurrent.ExecutorService interface represents an asynchronous execution mechanism which is capable of executing tasks in the background. An ExecutorService is thus very similar to a [thread pool](http://tutorials.jenkov.com/java-concurrency/thread-pools.html). In fact, the implementation of ExecutorService present in the java.util.concurrent package is a thread pool implementation.

## ExecutorService Example

Here is a simple Java ExectorService example:

ExecutorService executorService = Executors.newFixedThreadPool(10);

executorService.execute(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

}

});

executorService.shutdown();

First an ExecutorService is created using the newFixedThreadPool() factory method. This creates a thread pool with 10 threads executing tasks.

Second, an anonymous implementation of the Runnable interface is passed to the execute() method. This causes the Runnable to be executed by one of the threads in the ExecutorService.

## Task Delegation

Here is a diagram illustrating a thread delegating a task to an ExecutorService for asynchronous execution:

|  |
| --- |
| A thread delegating a task to an ExecutorService for asynchronous execution. |
| **A thread delegating a task to an ExecutorService for asynchronous execution.** |

Once the thread has delegated the task to the ExecutorService, the thread continues its own execution independent of the execution of that task.

## ExecutorService Implementations

Since ExecutorService is an interface, you need to its implementations in order to make any use of it. TheExecutorService has the following implementation in the java.util.concurrent package:

* [ThreadPoolExecutor](http://tutorials.jenkov.com/java-util-concurrent/threadpoolexecutor.html)
* [ScheduledThreadPoolExecutor](http://tutorials.jenkov.com/java-util-concurrent/scheduledexecutorservice.html)

## Creating an ExecutorService

How you create an ExecutorService depends on the implementation you use. However, you can use theExecutors factory class to create ExecutorService instances too. Here are a few examples of creating anExecutorService:

ExecutorService executorService1 = Executors.newSingleThreadExecutor();

ExecutorService executorService2 = Executors.newFixedThreadPool(10);

ExecutorService executorService3 = Executors.newScheduledThreadPool(10);

## ExecutorService Usage

There are a few different ways to delegate tasks for execution to an ExecutorService:

* execute(Runnable)
* submit(Runnable)
* submit(Callable)
* invokeAny(...)
* invokeAll(...)

I will take a look at each of these methods in the following sections.

### execute(Runnable)

The execute(Runnable) method takes a java.lang.Runnable object, and executes it asynchronously. Here is an example of executing a Runnable with an ExecutorService:

ExecutorService executorService = Executors.newSingleThreadExecutor();

executorService.execute(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

}

});

executorService.shutdown();

There is no way of obtaining the result of the executed Runnable, if necessary. You will have to use a Callable for that (explained in the following sections).

### submit(Runnable)

The submit(Runnable) method also takes a Runnable implementation, but returns a Future object. ThisFuture object can be used to check if the Runnable as finished executing.

Here is a ExecutorService submit() example:

Future future = executorService.submit(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

}

});

future.get(); //returns null if the task has finished correctly.

### submit(Callable)

The submit(Callable) method is similar to the submit(Runnable) method except for the type of parameter it takes. The Callable instance is very similar to a Runnable except that its call() method can return a result. TheRunnable.run() method cannot return a result.

The Callable's result can be obtained via the Future object returned by the submit(Callable) method. Here is an ExecutorService Callable example:

Future future = executorService.submit(new Callable(){

public Object call() throws Exception {

System.out.println("Asynchronous Callable");

return "Callable Result";

}

});

System.out.println("future.get() = " + future.get());

The above code example will output this:

Asynchronous Callable

future.get() = Callable Result

### invokeAny()

The invokeAny() method takes a collection of Callable objects, or subinterfaces of Callable. Invoking this method does not return a Future, but returns the result of one of the Callable objects. You have no guarantee about which of the Callable's results you get. Just one of the ones that finish.

If one of the tasks complete (or throws an exception), the rest of the Callable's are cancelled.

Here is a code example:

ExecutorService executorService = Executors.newSingleThreadExecutor();

Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 1";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 2";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 3";

}

});

String result = executorService.invokeAny(callables);

System.out.println("result = " + result);

executorService.shutdown();

This code example will print out the object returned by one of the Callable's in the given collection. I have tried running it a few times, and the result changes. Sometimes it is "Task 1", sometimes "Task 2" etc.

### invokeAll()

The invokeAll() method invokes all of the Callable objects you pass to it in the collection passed as parameter. The invokeAll() returns a list of Future objects via which you can obtain the results of the executions of eachCallable.

Keep in mind that a task might finish due to an exception, so it may not have "succeeded". There is no way on aFuture to tell the difference.

Here is a code example:

ExecutorService executorService = Executors.newSingleThreadExecutor();

Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 1";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 2";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 3";

}

});

List<Future<String>> futures = executorService.invokeAll(callables);

for(Future<String> future : futures){

System.out.println("future.get = " + future.get());

}

executorService.shutdown();

## ExecutorService Shutdown

When you are done using the ExecutorService you should shut it down, so the threads do not keep running.

For instance, if your application is started via a main() method and your main thread exits your application, the application will keep running if you have an active ExexutorService in your application. The active threads inside this ExecutorService prevents the JVM from shutting down.

To terminate the threads inside the ExecutorService you call its shutdown() method. The ExecutorServicewill not shut down immediately, but it will no longer accept new tasks, and once all threads have finished current tasks, the ExecutorService shuts down. All tasks submitted to the ExecutorService before shutdown() is called, are executed.

If you want to shut down the ExecutorService immediately, you can call the shutdownNow() method. This will attempt to stop all executing tasks right away, and skips all submitted but non-processed tasks. There are no guarantees given about the executing tasks. Perhaps they stop, perhaps the execute until the end. It is a best effort attempt.

# ThreadPoolExecutor

The java.util.concurrent.ThreadPoolExecutor is an implementation of the [ExecutorService](http://tutorials.jenkov.com/java-util-concurrent/executorservice.html)interface. The ThreadPoolExecutor executes the given task (Callable or Runnable) using one of its internally pooled threads.

The thread pool contained inside the ThreadPoolExecutor can contain a varying amount of threads. The number of threads in the pool is determined by these variables:

* corePoolSize
* maximumPoolSize

If less than corePoolSize threads are created in the the thread pool when a task is delegated to the thread pool, then a new thread is created, even if idle threads exist in the pool.

If the internal queue of tasks is full, and corePoolSize threads or more are running, but less thanmaximumPoolSize threads are running, then a new thread is created to execute the task.

Here is a diagram illustrating the ThreadPoolExecutor principles:

|  |
| --- |
| A ThreadPoolExecutor. |
| **A ThreadPoolExecutor** |

## Creating a ThreadPoolExecutor

The ThreadPoolExecutor has several constructors available. For instance:

int corePoolSize = 5;

int maxPoolSize = 10;

long keepAliveTime = 5000;

ExecutorService threadPoolExecutor =

new ThreadPoolExecutor(

corePoolSize,

maxPoolSize,

keepAliveTime,

TimeUnit.MILLISECONDS,

new LinkedBlockingQueue<Runnable>()

);

However, unless you need to specify all these parameters explicitly for your ThreadPoolExecutor, it is often easier to use one of the factory methods in the java.util.concurrent.Executors class, as shown in the[ExecutorService](http://tutorials.jenkov.com/java-util-concurrent/executorservice.html) text.

# ScheduledExecutorService

The java.util.concurrent.ScheduledExecutorService is an [ExecutorService](http://tutorials.jenkov.com/java-util-concurrent/executorservice.html) which can schedule tasks to run after a delay, or to execute repeatedly with a fixed interval of time in between each execution. Tasks are executed asynchronously by a worker thread, and not by the thread handing the task to theScheduledExecutorService.

## ScheduledExecutorService Example

Here is a simple ScheduledExecutorService example:

ScheduledExecutorService scheduledExecutorService =

Executors.newScheduledThreadPool(5);

ScheduledFuture scheduledFuture =

scheduledExecutorService.schedule(new Callable() {

public Object call() throws Exception {

System.out.println("Executed!");

return "Called!";

}

},

5,

TimeUnit.SECONDS);

First a ScheduledExecutorService is created with 5 threads in. Then an anonymous implementation of theCallable interface is created and passed to the schedule() method. The two last parameters specify that theCallable should be executed after 5 seconds.

## ScheduledExecutorService Implementations

Since ScheduledExecutorService is an interface, you will have to use its implementation in thejava.util.concurrent package, in order to use it. ScheduledExecutorService as the following implementation:

* ScheduledThreadPoolExecutor

## Creating a ScheduledExecutorService

How you create an ScheduledExecutorService depends on the implementation you use. However, you can use the Executors factory class to create ScheduledExecutorService instances too. Here is an example:

ScheduledExecutorService scheduledExecutorService =

Executors.newScheduledThreadPool(5);

## ScheduledExecutorService Usage

Once you have created a ScheduledExecutorService you use it by calling one of its methods:

* schedule (Callable task, long delay, TimeUnit timeunit)
* schedule (Runnable task, long delay, TimeUnit timeunit)
* scheduleAtFixedRate (Runnable, long initialDelay, long period, TimeUnit timeunit)
* scheduleWithFixedDelay (Runnable, long initialDelay, long period, TimeUnit timeunit)

I will briefly cover each of these methods below.

### schedule (Callable task, long delay, TimeUnit timeunit)

This method schedules the given Callable for execution after the given delay.

The method returns a ScheduledFuture which you can use to either cancel the task before it has started executing, or obtain the result once it is executed.

Here is an example:

ScheduledExecutorService scheduledExecutorService =

Executors.newScheduledThreadPool(5);

ScheduledFuture scheduledFuture =

scheduledExecutorService.schedule(new Callable() {

public Object call() throws Exception {

System.out.println("Executed!");

return "Called!";

}

},

5,

TimeUnit.SECONDS);

System.out.println("result = " + scheduledFuture.get());

scheduledExecutorService.shutdown();

This example outputs:

Executed!

result = Called!

### schedule (Runnable task, long delay, TimeUnit timeunit)

This method works like the method version taking a Callable as parameter, except a Runnable cannot return a value, so the ScheduledFuture.get() method returns null when the task is finished.

### scheduleAtFixedRate (Runnable, long initialDelay, long period, TimeUnit timeunit)

This method schedules a task to be executed periodically. The task is executed the first time after theinitialDelay, and then recurringly every time the period expires.

If any execution of the given task throws an exception, the task is no longer executed. If no exceptions are thrown, the task will continue to be executed until the ScheduledExecutorService is shut down.

If a task takes longer to execute than the period between its scheduled executions, the next execution will start after the current execution finishes. The scheduled task will not be executed by more than one thread at a time.

### scheduleWithFixedDelay (Runnable, long initialDelay, long period, TimeUnit timeunit)

This method works very much like scheduleAtFixedRate() except that the period is interpreted differently.

In the scheduleAtFixedRate() method the period is interpreted as a delay between the start of the previous execution, until the start of the next execution.

In this method, however, the period is interpreted as the delay between the **end** of the previous execution, until the start of the next. The delay is thus between finished executions, not between the beginning of executions.

## ScheduledExecutorService Shutdown

Just like an ExecutorService, the ScheduledExecutorService needs to be shut down when you are finished using it. If not, it will keep the JVM running, even when all other threads have been shut down.

You shut down a ScheduledExecutorService using the shutdown() or shutdownNow() methods which are inherited from the ExecutorService interface. See the [ExecutorService Shutdown](http://tutorials.jenkov.com/java-util-concurrent/executorservice.html#executorservice-shutdown) section for more information.

# Lock

A java.util.concurrent.locks.Lock is a thread synchronization mechanism just like synchronized blocks. A Lock is, however, more flexible and more sophisticated than a synchronized block.

By the way, in my [Java Concurrency tutorial](http://tutorials.jenkov.com/java-concurrency/index.html) I have described how to implement your own locks, in case you are interested (or need it). See my text on [Locks](http://tutorials.jenkov.com/java-concurrency/locks.html) for more details.

## Java Lock Example

Since Lock is an interface, you need to use one of its implementations to use a Lock in your applications. Here is a simple usage example:

Lock lock = new ReentrantLock();

lock.lock();

//critical section

lock.unlock();

First a Lock is created. Then it's lock() method is called. Now the Lock instance is locked. Any other thread callinglock() will be blocked until the thread that locked the lock calls unlock(). Finally unlock() is called, and theLock is now unlocked so other threads can lock it.

## Java Lock Implementations

The java.util.concurrent.locks package has the following implementations of the Lock interface:

* ReentrantLock

## Main Differences Between Locks and Synchronized Blocks

The main differences between a Lock and a synchronized block are:

* A synchronized block makes no guarantees about the sequence in which threads waiting to entering it are granted access.
* You cannot pass any parameters to the entry of a synchronized block. Thus, having a timeout trying to get access to a synchronized block is not possible.
* The synchronized block must be fully contained within a single method. A Lock can have it's calls to lock() andunlock() in separate methods.

## Lock Methods

The Lock interface has the following primary methods:

* lock()
* lockInterruptibly()
* tryLock()
* tryLock(long timeout, TimeUnit timeUnit)
* unlock()

The lock() method locks the Lock instance if possible. If the Lock instance is already locked, the thread callinglock() is blocked until the Lock is unlocked.

The lockInterruptibly() method locks the Lock unless the thread calling the method has been interrupted. Additionally, if a thread is blocked waiting to lock the Lock via this method, and it is interrupted, it exits this method calls.

The tryLock() method attempts to lock the Lock instance immediately. It returns true if the locking succeeds, false if Lock is already locked. This method never blocks.

The tryLock(long timeout, TimeUnit timeUnit) works like the tryLock() method, except it waits up the given timeout before giving up trying to lock the Lock.

The unlock() method unlocks the Lock instance. Typically, a Lock implementation will only allow the thread that has locked the Lock to call this method. Other threads calling this method may result in an unchecked exception (RuntimeException).

# ReadWriteLock

A java.util.concurrent.locks.ReadWriteLock is an advanced thread lock mechanism. It allows multiple threads to read a certain resource, but only one to write it, at a time.

The idea is, that multiple threads can read from a shared resource without causing concurrency errors. The concurrency errors first occur when reads and writes to a shared resource occur concurrently, or if multiple writes take place concurrently.

In this text I only cover Java's built-in ReadWriteLock. If you want to read more about the theory behind the implemenation of a ReadWriteLock, you can read it in my text on [Read Write Locks](http://tutorials.jenkov.com/java-concurrency/read-write-locks.html) in my Java Concurrency tutorial.

## ReadWriteLock Locking Rules

The rules by which a thread is allowed to lock the ReadWriteLock either for reading or writing the guarded resource, are as follows:

|  |  |
| --- | --- |
| **Read Lock** | If no threads have locked the ReadWriteLock for writing,  and no thread have requested a write lock (but not yet obtained it).  Thus, multiple threads can lock the lock for reading. |
| **Write Lock** | If no threads are reading or writing.  Thus, only one thread at a time can lock the lock for writing. |

## ReadWriteLock Implementations

ReadWriteLock is an interface. Thus, to use a ReadWriteLock

The java.util.concurrent.locks package contains the following ReadWriteLock implementation:

* ReentrantReadWriteLock

## ReadWriteLock Code Example

Here is a simple code example that shows how to create a ReadWriteLock and how to lock it for reading and writing:

ReadWriteLock readWriteLock = new ReentrantReadWriteLock();

readWriteLock.readLock().lock();

// multiple readers can enter this section

// if not locked for writing, and not writers waiting

// to lock for writing.

readWriteLock.readLock().unlock();

readWriteLock.writeLock().lock();

// only one writer can enter this section,

// and only if no threads are currently reading.

readWriteLock.writeLock().unlock();

Notice how the ReadWriteLock actually internally keeps two Lock instances. One guarding read access, and one guarding write access.