**Java Concurrency in Depth**

Java comes with strong support for multi-threading and concurrency, which makes it easy to write concurrent applications. But usually, multi-threaded applications are tricky to debug, troubleshoot, and sometimes to scale. From my experience with concurrent applications, most of the issues are found when they run at scale, which means when they go live in many cases. In order to make this easier, it is better to understand how things work under the hood and the pros and cons of every choice.

This article is the first in a series of articles discussing the internals of Java concurrency.

Let's start with this example:

public class Foo {

private int x;

public int getX() {

return x;

}

public void setX(int x) {

this.x = x;

}

}

This code is obviously not thread-safe. One way to make it thread safe is to make setX() and getX() both [synchronized](https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html).

## How Synchronization Works

When a thread calls a synchronized method or block, it tries to acquire an intrinsic lock (monitor). Once a thread acquires the lock, other threads block until the lock is released.

This looks okay! But there are some drawbacks for synchronization:

1. Starvation: Synchronization doesn't guarantee fairness. This means that if there are many threads competing to acquire the lock, then there is a possibility that some threads don't get a chance to continue, which means starvation.
2. Deadlock: Calling synchronized code from other synchronized code can cause deadlocks.
3. Less throughput: Using synchronization means only one thread is executing on a particular object. In many cases, this is not necessary because it is enough to lock access to the variable only on write, and there no need to lock the variable if all the threads at the moment are reading (concurrent reads).

Synchronization is good for thread safety but not optimal for concurrency.

Check out this Javadoc about [liveness](https://docs.oracle.com/javase/tutorial/essential/concurrency/liveness.html) problems.

## Volatile

Another solution is using volatile.

public class Foo {

private volatile int x;

...

}

### **How Volatile Works**

Volatile is said to guarantee:

1. **Visibility**: If one thread changes a value of a variable, the change will be visible immediately to other threads reading the variable. This is guaranteed by not allowing the compiler or the JVM to allocate those variables in the CPU registers. Any write to a volatile variable is flushed immediately to main memory and any read of it is fetched from main memory. That means there is a little bit of performance penalty, but that's far better from a concurrency point of view.
2. **Ordering**: Sometimes for performance optimization, the JVM reorders instructions. This is not allowed when accessing volatile variables. Access to volatile variables is not reordered with access to other volatile variables, nor with access to other normal fields around them. This makes writes to non-volatile fields around them visible immediately to other threads.

Let's look at an example to clarify this:

public class Foo {

private int x = -1;

private volatile boolean v = false;

public void setX(int x) {

this.x = x;

v = true;

}

public int getX() {

if (v == true) {

return x;

}

return 0;

}

}

Because of the first rule, if thread A calls setX(), and thread B calls getX(), then the change to v will be visible immediately to thread B. And because of the second rule, the change to x will be visible to thread B immediately as well.

However, volatile is not suitable for some operations, like ++, --, etc. This is because these operations translate into multiple read and write instructions. For example:

public int increment() {

//x++

int tmp = x;

tmp = tmp + 1;

x = tmp;

return x;

}

In a multi-threaded program, such operations should be [atomic](https://docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html), which volatile doesn't guarantee. Java SE comes with a set of atomic classes like AtomicInteger, AtomicLong, and AtomicBoolean, which can be used to solve this problem.

## **How Atomic Classes Work**

Java relies on machine instructions/algorithms to achieve atomicity. Prior to Java 8, Atomic classes used [Compare-and-Swap](https://en.wikipedia.org/wiki/Compare-and-swap). Starting in Java 8, some methods of atomic classes began using [Fetch-and-Add](https://en.wikipedia.org/wiki/Fetch-and-add).

Let's have a look at this implementation of AtomicInteger.getAndIncrement() in Java 7:

public final int getAndIncrement() {

for (;;) {

int current = get();

int next = current + 1;

if (compareAndSet(current, next))

return current;

}

}

In Java 8, that implementation has changed to:

public final int getAndIncrement() {

return unsafe.getAndAddInt(this, valueOffset, 1);

}

In the first implementation, compareAndSet returns true only if the actual value equals the current one, so the loop goes indefinitely until this condition is met.

This will be completely fine in an environment with few threads, but let's think: What if we have 100 threads calling this function? Due to the high contention, race conditions are worse — so the loop might keep going on for a long time. That could lead to a [livelock](https://docs.oracle.com/javase/tutorial/essential/concurrency/starvelive.html) situation. In such cases, solutions have to be designed carefully. One idea could be using something like a map-reduce solution, where you divide the threads into sets (mappers) and each set shares an atomic instance and a reducer thread collects values from the shared atomic instances.

Is this problem solved in Java 8?

1. Keep in mind there are still some methods using the first approach, like getAndUpdate(IntUnaryOperator).
2. Performance under contention still goes down, but it remains much better in Java 8. Check out this [blog post](http://ashkrit.blogspot.nl/2014/02/atomicinteger-java-7-vs-java-8.html) where Ashkrit has plotted graphs comparing the performance of both.

**In the next part, I will discuss different types of locks...**

In this article, I will discuss other high-level locks that are built on top of volatile, atomic classes and Compare-And-Swap.

The primary reason for writing a multi-threaded application is improving performance. In this case, we are trading performance gain with more complexity in code, debugging, and monitoring. When having a shared object, locking becomes inevitable and this shared object becomes the bottleneck where we see performance issues, among others. So, choosing the right locking mechanism can have a profound impact on performance and operations.

But why is synchronized not enough?

1. Synchronized is an exclusive locking mechanism that cannot be tailored to different use cases.
2. No way to instruct the JVM to use fair policy.
3. When deadlock or starvation occur, the only way to resolve the problem is killing the process because threads block indefinitely and there is no way to interrupt a blocked thread in this case.

## Lock

Having an interface like [java.util.concurrent.locks.Lock](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html) provides great flexibility to have different implementations tailored to different use cases, as well as overcoming the drawbacks of synchronized. The interface defines the following methods:

* **lock()**: The thread blocks indefinitely until it acquires the lock. No way to interrupt the blocked thread.
* **lockInterruptibly()**: The thread blocks indefinitely until it acquires the lock. The blocked thread can be interrupted
* **tryLock()**: Non-blocking call. Returns immediately with true if the lock is acquired and false otherwise.
* **tryLock(long, TimeUnit)**: Blocking call until the lock is acquired or the specified timeout has elapsed. Returns with true if the lock is acquired and false otherwise. It is also possible to interrupt a blocked thread.
* **unlock()**: Releases the lock. It is important to ALWAYS have it in a finally block.
* **newCondition()**: This is one of the most useful features. In some use cases, the thread needs to wait until a certain condition is satisfied, and during this waiting time, it is useful to release the lock for other threads to continue execution. This is similar to Object.wait and Object.notify/notifyAll, which work with synchronized.

## ReentrantLock

[ReentrantLock](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html) implements Lock interface. It has two constructors to choose from depending on whether you need a fair lock or a non-fair one.

It is important to know that both the fair and non-fair variants of ReentrantLock use a waiting queue to park blocked threads. This means:

* Thread priority (can be set using Thread.setPriority()) has no effect. So, don't rely on it in when using reentrant locks.
* Fairness here is very simple — it is about getting a chance to execute. It's nothing like complex schedulers, which means it is still possible to cause starvation if a thread doesn't release the lock or holds it for a long time.

Fair and non-fair variants use [AbstractQueuedSynchronizer](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/AbstractQueuedSynchronizer.html), which implements the waiting queue. It is a variant of the [CLH (Craig, Landin, and Hagersten) lock queue](http://www.programering.com/a/MjM5gTNwATE.html).

The logic behind granting locks:

* Fair lock: grant the lock only if the call is recursive (i.e. the thread is already holding the lock) or there are no other waiting threads, or the thread is the first in the queue.
* Non-fair lock: tries to acquire the lock and, if it can't, then the thread is queued.

### Important Usage Guidelines

* tryLock(), unlike the rest of the methods, acts in a **non-fair way** with both fair and non-fair implementations. So, pay attention when using it with a fair lock, as you might be expecting different behavior.
* When using ReentrantLock, try to avoid using lock(). Use any of the other methods and, depending on the use case, you might add an exponential backoff or maximum retries or both methods combined. Even if the use case requires blocking, for instance until data is available, use the lockInterruptibly() or tryLock(long, Timeout) methods.

boolean acquired = false;

long wait = 100;

int retries = 0;

int maxRetries = 10;

try {

while (!acquired && retries < maxRetries) {

acquired = lock.tryLock(wait, TimeUnit.MILLISECONDS);

wait \*= 2;

++retries;

}

if (!acquired) {

// log error or throw exception

}

} catch (InterruptedException e) {

// log error or throw exception

} finally {

lock.unlock();

}

* One bad practice, in general, is ignoring interrupts. Interrupts should be handled properly to avoid any problems with application or thread pool termination. Even when you're sure that you need to ignore them, then log the exception. But don't just swallow it.
* The class provides many methods that, as per the Javadoc, should be used only for debugging and instrumentation and not for synchronization purposes. My advice is keeping programming to the interface because you don't need to pollute your code with such granular debug information that you can actually get using a decent profiler.

## ReentrantReadWriteLock

[ReentrantReadWriteLock](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html) actually consists of two reentrant locks: a read lock and a write lock. The read lock is shared (which means multiple threads can acquire it as long as the write lock is not acquired), whereas the exclusive write lock can be acquired by one thread only as long as the read lock is not acquired by any other thread. This is very useful in many use cases and helps increase concurrency.

This also has both fair and non-fair policies.

## Other Useful Synchronization Tools

* [CountDownLatch](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CountDownLatch.html): This is useful when one or more threads need to wait until a set of operations completes. A count is passed to the constructor and it can be decremented by calling countDown(). When the count goes to zero, waiting threads are notified.
* [CyclicBarrier](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CyclicBarrier.html): This is useful when a set of threads are required to wait until they reach a common point. The same behavior could be achieved using CoutDownLatch, but CyclicBarrier has some additional options:
  + The count in CyclicBarrier can be reset.
  + CyclicBarrier accepts an optional Runnable implementation that will execute when the barrier is tripped.
  + A barrier is considered broken if any of the threads leaves the barrier because of interruption, failure, or timeout. When this happens, all other threads waiting will leave the barrier by throwing BarrierBrokenException. The state of the barrier can be tested using isBroker(). This state is kept until reset() is called.
* [StampedLock](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/StampedLock.html): Yet another implementation of a read-write lock. It doesn't implement the Lock interface, and it is more complicated to use that ReentrantReadWriteLock. It has a very interesting feature — Optimistic Reading — but it is quite tricky and fragile to use. I strongly recomment using ReentrantReadWriteLock instead of StampedLock.

**compare-and-swap: -** compare-and-swap (CAS) is an atomic instruction used in multithreading to achieve synchronization. It compares the contents of a memory location with a given value and, only if they are the same, modifies the contents of that memory location to a new given value. This is done as a single atomic operation. The atomicity guarantees that the new value is calculated based on up-to-date information; if the value had been updated by another thread in the meantime, the write would fail. The result of the operation must indicate whether it performed the substitution; this can be done either with a simple boolean response (this variant is often called compare-and-set), or by returning the value read from the memory location (not the value written to it).

**Fetch-and-add: -** fetch-and-add CPU instruction (FAA) atomically increments the contents of a memory location by a specified value. That is, fetch-and-add performs the operation increment the value at address x by a, where x is a memory location and a is some value, and return the original value at x in such a way that if this operation is executed by one process in a concurrent system, no other process will ever see an intermediate result. Fetch-and-add can be used to implement concurrency control structures such as mutex locks and semaphores.