**JAVA : Concurrency in Practice by Brian Goetz Notes**

**Fundamentals**

**Risk of Threads**

1. Safety Hazards
2. Liveness Hazards
3. Performance Hazards

Thread safety can be unexpectedly subtle because, in the absence of sufficient synchronization, the ordering of operations in multiple threads is unpredictable and sometimes surprising. Similarly, the use of threads introduces additional forms of *liveness failure* that do not occur in single-threaded programs.

While *safety* means “nothing bad ever happens”, liveness concerns the complementary goal that “something good eventually happens”. A liveness failure occurs when an activity gets into a state such that it is permanently unable to make forward progress. One form of liveness failure that can occur in sequential programs is an inadvertent infinite loop, where the code that follows the loop never gets executed. Various forms of liveness failures are deadlock, starvation, and livelock.

Related to liveness is *performance*. While liveness means that something good *eventually* happens, eventually may not be good enough—we often want good things to happen quickly. Performance issues subsume a broad range of problems, including poor service time, responsiveness, throughput, resource consumption, or scalability. In well designed concurrent applications the use of threads is a net performance gain, but threads nevertheless carry some degree of runtime overhead. *Context switches*—when the scheduler suspends the active thread temporarily so another thread can run—are more frequent in applications with many threads, and have significant costs: saving and restoring execution context, loss of locality, and CPU time spent scheduling threads instead of running them

**Thread Safety**

**2.1. WHAT IS THREAD SAFETY?**

Building concurrent programs require the correct use of threads and locks. But these are just *mechanisms*—means to an end. Writing thread-safe code is, at its core, about managing access to *state*, and in particular to *shared, mutable state*.

Informally, an object's *state* is its data, stored in *state variables* such as instance or static fields. An object's state may include fields from other, dependent objects; a HashMap's state is partially stored in the HashMap object itself, but also in many Map.Entry objects. An object's state encompasses any data that can affect its externally visible behavior.

Whether an object needs to be thread-safe depends on whether it will be accessed from multiple threads. This is a property of how the object is *used* in a program, not what it *does*. Making an object thread-safe requires using synchronization to coordinate access to its mutable state; failing to do so could result in data corruption and other undesirable consequences

*Whenever more than one thread accesses a given state variable, and one of them might write to it, they all must coordinate their access to it using synchronization.* The primary mechanism for synchronization in Java is the “synchronized” keyword, which provides exclusive locking, but the term “synchronization” also includes the use of volatile variables, explicit locks, and atomic variables.

If multiple threads access the same mutable state variable without appropriate synchronization, *your program is broken*. There are three ways to fix it:

* *Don't share* the state variable across threads;
* Make the state variable *immutable*; or
* Use *synchronization* whenever accessing the state variable.

If you haven't considered concurrent access in your class design, some of these approaches can require significant design modifications, so fixing the problem might not be as trivial as this advice makes it sound. *It is far easier to design a class to be thread-safe than to retrofit it for thread safety later.*

the same object-oriented techniques that help you write well-organized, maintainable classes—such as encapsulation and data hiding—can also help you create thread-safe classes. The less code that has access to a particular variable, the easier it is to ensure that all of it uses the proper synchronization, and the easier it is to reason about the conditions under which a given variable might be accessed. The better encapsulated your program state, the easier it is to make your program thread-safe and to help maintainers keep it that way.

Thread safety may be a term that is applied to *code*, but it is about *state*, and it can only be applied to the entire body of code that encapsulates its state, which may be an object or an entire program.

A vague definition of thread safety is “a class is thread-safe if it can be used safely from multiple threads.” You can't really argue with such a statement, but it doesn't offer much practical help either. How do we tell a thread-safe class from an unsafe one? What do we even mean by “safe”?

At the heart of any reasonable definition of thread safety is the concept of *correctness*. If our definition of thread safety is fuzzy, it is because we lack a clear definition of correctness.

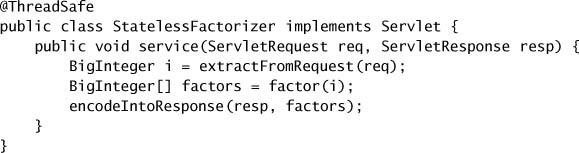
A class is thread-safe when it continues to behave correctly when accessed from multiple threads. Correctness means that a class *conforms to its specification*. A good specification defines *invariants* constraining an object's state and *postconditions* describing the effects of its operations.

If an object is correctly implemented, no sequence of operations—calls to public methods and reads or writes of public fields—should be able to violate any of its invariants or postconditions. *No set of operations performed sequentially or concurrently on instances of a thread-safe class can cause an instance to be in an invalid state.*

**Example: A Stateless Servlet**

A number of frameworks that create threads and call your components from those threads, leaving you with the responsibility of making your components thread-safe. Very often, thread-safety requirements stem not from a decision to use threads directly but from a decision to use a facility like the Servlets framework. We're going to develop a simple example—a servlet-based factorization service—and slowly extend it to add features while preserving its thread safety.

**Listing 2.1. A Stateless Servlet.**



StatelessFactorizer is, like most servlets, stateless: it has no fields and references no fields from other classes. The transient state for a particular computation exists solely in local variables that are stored on the thread's stack and are accessible only to the executing thread. One thread accessing a StatelessFactorizer cannot influence the result of another thread accessing the same StatelessFactorizer; because the two threads do not share state, it is as if they were accessing different instances. Since the actions of a thread accessing a stateless object cannot affect the correctness of operations in other threads, stateless objects are thread-safe.

Stateless objects are always thread-safe.

**2.2. ATOMICITY**

Suppose we want to add a “hit counter” that measures the number of requests processed to above servlet code i.e. add ++count. While the increment operation, ++count, may look like a single action because of its compact syntax, it is not *atomic*, which means that it does not execute as a single, indivisible operation. Instead, it is a shorthand for a sequence of three discrete operations: fetch the current value, add one to it, and write the new value back. This is an example of a *read-modify-write* operation, in which the resulting state is derived from the previous state.

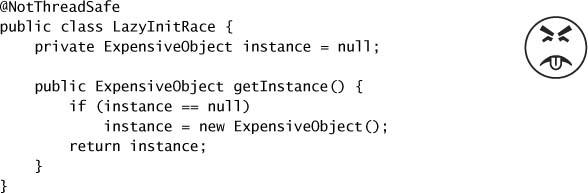
If the counter is initially 9, with some unlucky timing each thread could read the value, see that it is 9, add one to it, and each set the counter to 10. This is clearly not what is supposed to happen; an increment got lost along the way, and the hit counter is now permanently off by one.

The possibility of incorrect results in the presence of unlucky timing is so important in concurrent programming that it has a name: a *race condition*. A race condition occurs when the correctness of a computation depends on the relative timing or interleaving of multiple threads by the runtime; in other words, when getting the right answer relies on lucky timing. The most common type of race condition is *check-then-act*, where a potentially stale observation is used to make a decision on what to do next.

The term *race condition* is often confused with the related term *data race*, which arises when synchronization is not used to coordinate all access to a shared non-final field.

A common idiom that uses check-then-act is *lazy initialization*. The goal of lazy initialization is to defer initializing an object until it is actually needed while at the same time ensuring that object is initialized only once

**Listing 2.3. Race Condition in Lazy Initialization. *Don't do this.***



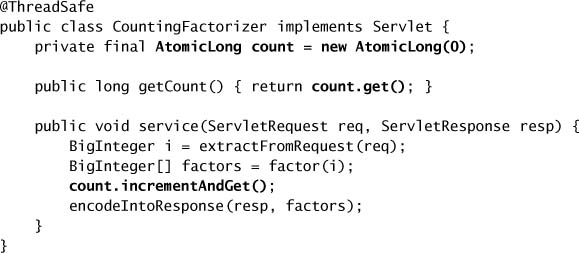
The hit-counting operation in UnsafeCountingFactorizer has another sort of race condition. Read-modify-write operations, like incrementing a counter, define a transformation of an object's state in terms of its previous state. To increment a counter, you have to know its previous value *and* make sure no one else changes or uses that value while you are in mid-update.

Both LazyInitRace and UnsafeCountingFactorizer contained a sequence of operations that needed to be *atomic*, or indivisible, relative to other operations on the same state. Operations *A* and *B* are *atomic* with respect to each other if, from the perspective of a thread executing *A*, when another thread executes *B*, either all of *B* has executed or none of it has. An *atomic operation* is one that is atomic with respect to all operations, including itself, that operate on the same state.

To ensure thread safety, check-then-act operations (like lazy initialization) and read-modify-write operations (like increment) must always be atomic. We refer collectively to check-then-act and read-modify-write sequences as *compound actions*: sequences of operations that must be executed atomically in order to remain thread-safe.

The java.util.concurrent.atomic package contains *atomic variable* classes for effecting atomic state transitions on numbers and object references. By replacing the long counter with an AtomicLong, we ensure that all actions that access the counter state are atomic. Because the state of the servlet *is* the state of the counter and the counter is thread-safe in the code below, our servlet is once again thread-safe.

**Listing 2.4. Servlet that Counts Requests Using** AtomicLong**.**

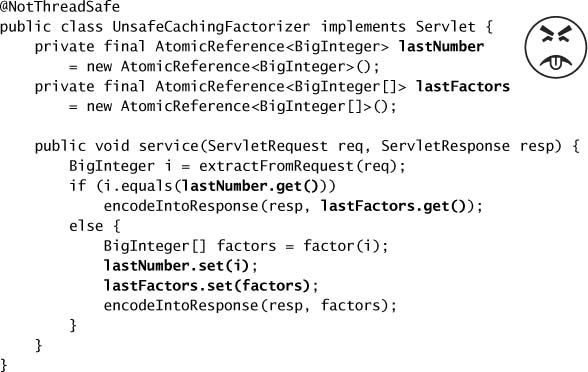


**2.3. LOCKING**

We were able to add one state variable to our servlet while maintaining thread safety by using a thread-safe object to manage the entire state of the servlet. But if we want to add more state to our servlet, can we just add more thread-safe state variables?

Imagine that we want to improve the performance of our servlet by caching the most recently computed result, just in case two consecutive clients request factorization of the same number. To implement this strategy, we need to remember two things: the last number factored, and its factors. It can be done using atomic variables as shown below.

**Listing 2.5. Servlet that Attempts to Cache its Last Result without Adequate Atomicity. *Don't do this.***



Unfortunately, this approach does not work. Even though the atomic references are individually thread-safe, UnsafeCachingFactorizer has race conditions that could make it produce the wrong answer.

The definition of thread safety requires that invariants be preserved regardless of timing or interleaving of operations in multiple threads. One invariant of UnsafeCachingFactorizer is that the product of the factors cached in lastFactors equal the value cached in lastNumber; our servlet is correct only if this invariant always holds. When multiple variables participate in an invariant, they are not *independent*: the value of one constrains the allowed value(s) of the others. Thus when updating one, you must update the others *in the same atomic operation*.

With some unlucky timing, UnsafeCachingFactorizer can violate this invariant. Using atomic references, we cannot update both lastNumber and lastFactors simultaneously, even though each call to set is atomic; there is still a window of vulnerability when one has been modified and the other has not, and during that time other threads could see that the invariant does not hold. Similarly, the two values cannot be fetched simultaneously: between the time when thread *A* fetches the two values, thread *B* could have changed them, and again *A* may observe that the invariant does not hold.

To preserve state consistency, update related state variables in a single atomic operation.

**Intrinsic Locks**

Java provides a built-in locking mechanism for enforcing atomicity: the synchronized block. (There is also another critical aspect to locking and other synchronization mechanisms—visibility—which is covered in [Chapter 3](https://learning.oreilly.com/library/view/java-concurrency-in/0321349601/ch03.html#ch03).) A synchronized block has two parts: a reference to an object that will serve as the *lock*, and a block of code to be guarded by that lock. A synchronized method is a shorthand for a synchronized block that spans an entire method body, and whose lock is the object on which the method is being invoked. (Static synchronized methods use the Class object for the lock.)

c0025-01.jpg

Intrinsic locks in Java act as *mutexes* (or *mutual exclusion locks*), which means that at most one thread may own the lock. When thread *A* attempts to acquire a lock held by thread *B*, *A* must wait, or *block*, until *B* releases it. If *B* never releases the lock, *A* waits forever.

**Reentrancy**

When a thread requests a lock that is already held by another thread, the requesting thread blocks. But because intrinsic locks are *reentrant*, if a thread tries to acquire a lock that *it* already holds, the request succeeds. Reentrancy means that locks are acquired on a per-thread rather than per-invocation basis. Reentrancy is implemented by associating with each lock an acquisition count and an owning thread. When the count is zero, the lock is considered unheld. When a thread acquires a previously unheld lock, the JVM records the owner and sets the acquisition count to one. If that same thread acquires the lock again, the count is incremented, and when the owning thread exits the synchronized block, the count is decremented. When the count reaches zero, the lock is released.

**2.4. GUARDING STATE WITH LOCKS**

Because locks enable serialized access to the code paths they guard, we can use them to construct protocols for guaranteeing exclusive access to shared state. Following these protocols consistently can ensure state consistency.

Compound actions on shared state, such as incrementing a hit counter (read-modify-write) or lazy initialization (check-then-act), must be made atomic to avoid race conditions. Holding a lock for the *entire duration* of a compound action can make that compound action atomic. However, just wrapping the compound action with a synchronized block is not sufficient; if synchronization is used to coordinate access to a variable, it is needed *everywhere that variable is accessed*. Further, when using locks to coordinate access to a variable, the *same* lock must be used wherever that variable is accessed.

There is no inherent relationship between an object's intrinsic lock and its state; an object's fields need not be guarded by its intrinsic lock, though this is a perfectly valid locking convention that is used by many classes. Acquiring the lock associated with an object does *not* prevent other threads from accessing that object—the only thing that acquiring a lock prevents any other thread from doing is acquiring that same lock. The fact that every object has a built-in lock is just a convenience so that you needn't explicitly create lock objects. [9] It is up to you to construct *locking protocols* or *synchronization policies* that let you access shared state safely, and to use them consistently throughout your program.

For every invariant that involves more than one variable, *all* the variables involved in that invariant must be guarded by the *same* lock.

If synchronization is the cure for race conditions, why not just declare every method synchronized? It turns out that such indiscriminate application of synchronized might be either too much or too little synchronization. Merely synchronizing every method, as Vector does, is not enough to render compound actions on a Vector atomic:

c0029-01.jpg

This attempt at a put-if-absent operation has a race condition, even though both contains and add are atomic. While synchronized methods can make individual operations atomic, additional locking is required when multiple operations are combined into a compound action.

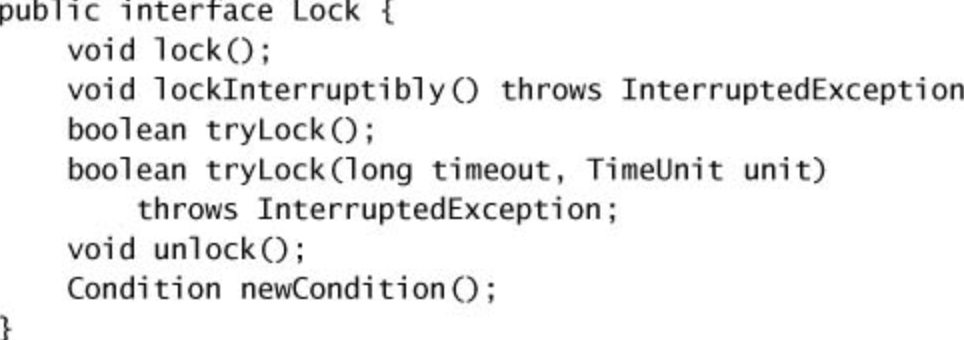
**Explicit Locks**

Before Java 5.0, the only mechanisms for coordinating access to shared data were

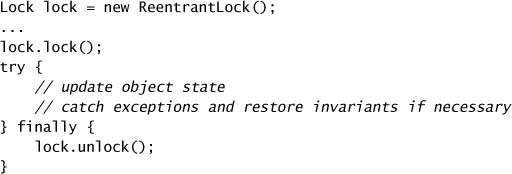
* synchronized
* volatile.

Java 5.0 adds another option: ReentrantLock. Contrary to what some have written, ReentrantLock is not a replacement for intrinsic locking, but rather an *alternative with advanced features for when intrinsic locking proves too limited*.

Unlike intrinsic locking, Lock offers a choice of unconditional, polled, timed, and interruptible lock acquisition, and all lock and unlock operations are explicit. Lock implementations must provide the same memory-visibility semantics as intrinsic locks, but can differ in their locking semantics, scheduling algorithms, ordering guarantees, and performance characteristics.



ReentrantLock implements Lock, providing the same mutual exclusion and memory-visibility guarantees as synchronized. Acquiring a ReentrantLock has the same memory semantics as entering a synchronized block, and releasing a ReentrantLock has the same memory semantics as exiting a synchronized block. Like synchronized, ReentrantLock offers reentrant locking semantics. ReentrantLock supports all of the lock-acquisition modes defined by Lock, providing more flexibility for dealing with lock unavailability than does synchronized.



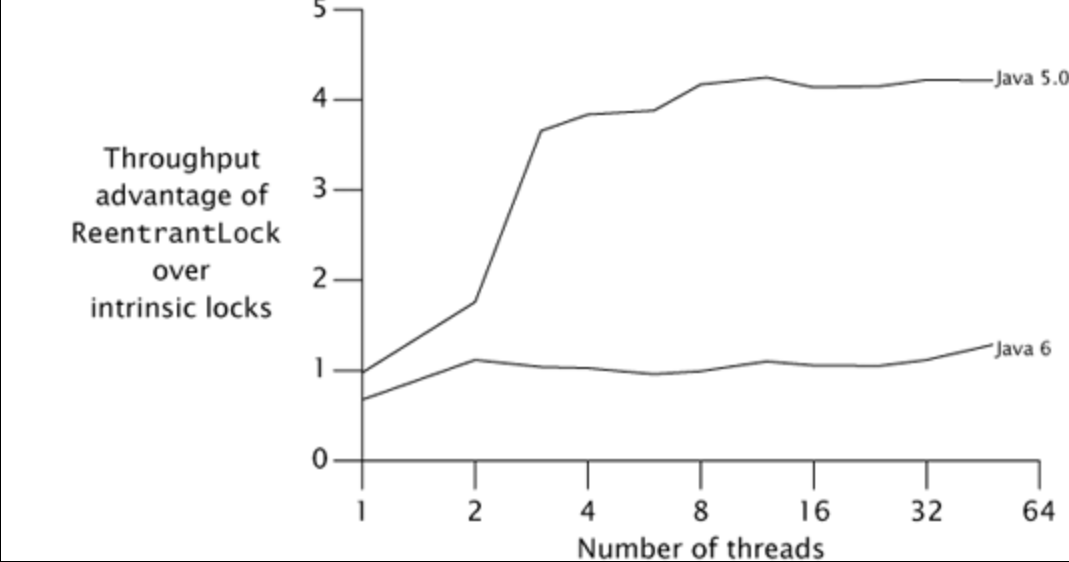
Failing to release an explicit Lock (with or without finally block) is a ticking time bomb. When it goes off, you will have a hard time tracking down its origin as there will be no record of where or when the Lock should have been released. This is one reason not to use ReentrantLock as a blanket substitute for synchronized: it is more “dangerous” because it doesn't automatically clean up the lock when control leaves the guarded block. While remembering to release the lock from a finally block is not all that difficult, it is also not impossible to forget.

To recap, some of the features of explicit locks:

* Polled and Timed Lock Acquisition
* Interruptible Lock Acquisition
* Non-Block Structured Locking : Automatic lock release as is done in Synchronized block simplifies analysis and prevents potential coding errors, but sometimes a more flexible locking discipline is needed. Reducing lock granularity can enhance scalability. Lock striping allows different hash chains in a hash-based collection to use different locks. We can apply a similar principle to reduce locking granularity in a linked list by using a separate lock for *each link node*, allowing different threads to operate independently on different portions of the list.

**Performance Considerations (Intrinsic locking vs explicit locking)**

Though this particular graph doesn't show it, the scalability difference between Java 5.0 and Java 6 really does come from improvement in intrinsic locking, rather than from regression in Reentrant-Lock.



Performance is a moving target; yesterday's benchmark showing that *X* is faster than *Y* may already be out of date today.

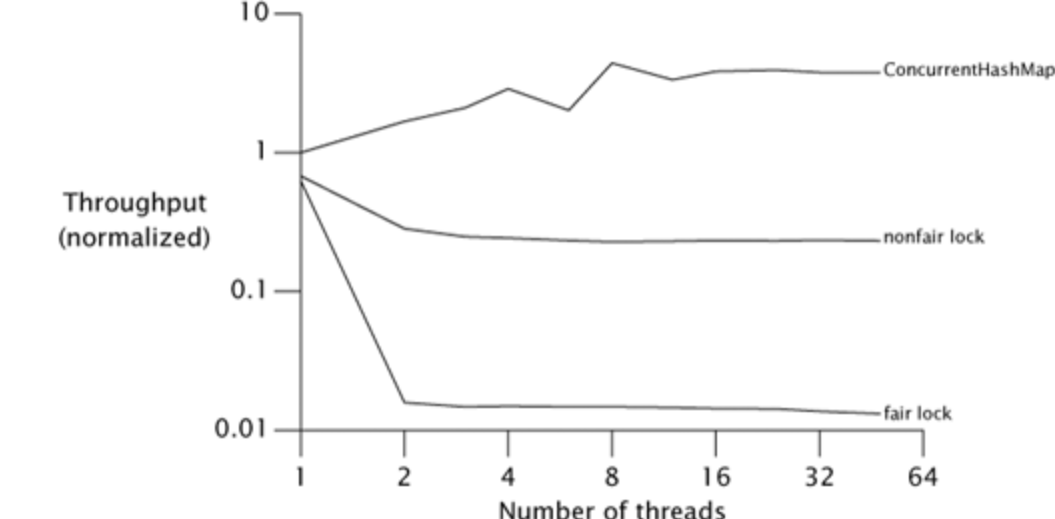
**Fairness Guarantees**

The ReentrantLock constructor offers a choice of two *fairness* options: create a *nonfair (i.e. barging)* lock (the default) or a *fair (no barging)* lock. Threads acquire a fair lock in the order in which they requested it, whereas a nonfair lock permits *barging*: threads requesting a lock can jump ahead of the queue of waiting threads if the lock happens to be available when it is requested.

When it comes to locking, though, fairness has a significant performance cost because of the overhead of suspending and resuming threads. In practice, a statistical fairness guarantee—promising that a blocked thread will *eventually* acquire the lock—is often good enough, and is far less expensive to deliver. Some algorithms rely on fair queueing to ensure their correctness, but these are unusual. In most cases, the performance benefits of nonfair locks outweigh the benefits of fair queueing.

*Don't pay for fairness (or no barging) if you don't need it.*

One reason barging locks perform so much better than fair locks under heavy contention is that there can be a significant delay between when a suspended thread is resumed and when it actually runs. Let's say thread *A* holds a lock and thread *B* asks for that lock. Since the lock is busy, *B* is suspended. When *A* releases the lock, *B* is resumed so it can try again. In the meantime, though, if thread *C* requests the lock, there is a good chance that *C* can acquire the lock, use it, and release it before *B* even finishes waking up. In this case, everyone wins: *B* gets the lock no later than it otherwise would have, *C* gets it much earlier, and throughput is improved.



Like the ***default*** ReentrantLock, intrinsic locking offers no deterministic fairness guarantees, but the statistical fairness guarantees of most locking implementations are good enough for almost all situations. The language specification does not require the JVM to implement intrinsic locks fairly, and no production JVMs do. ReentrantLock does not depress lock fairness to new lows—it only makes explicit something that was present all along.

**Choosing Between Intrinsic & Reentrant Lock**

ReentrantLock provides the same locking and memory semantics as intrinsic locking, as well as additional features such as:

* timed lock waits
* interruptible lock waits
* polled lock acquisition
* fairness
* ability to implement non-block-structured locking.

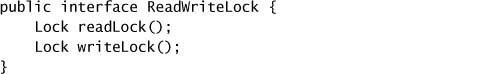
Despite all additional features as above, Intrinsic locks still have significant advantages over explicit locks. The notation is familiar and compact, and many existing programs already use intrinsic locking—and mixing the two could be confusing and error-prone. Reentrant-Lock is definitely a more dangerous tool than synchronization; if you forget to wrap the unlock call in a finally block, your code will probably appear to run properly, but you've created a time bomb that may well hurt innocent bystanders. Save ReentrantLock for situations in which you need something ReentrantLock provides that intrinsic locking doesn't.

ReentrantLock is an advanced tool for situations where intrinsic locking is not practical. Use it if you need its advanced features: timed, polled, or interruptible lock acquisition, fair queueing, or non-block-structured locking. Otherwise, prefer synchronized.

**Read-Write Locks**

ReentrantLock implements a standard mutual-exclusion lock: at most one thread at a time can hold a ReentrantLock. But mutual exclusion is frequently a stronger locking discipline than needed to preserve data integrity, and thus limits concurrency more than necessary. Mutual exclusion is *a conservative locking strategy that prevents writer/writer and writer/reader overlap, but also prevents reader/reader overlap*. In many cases, data structures are “read-mostly”—they are mutable and are sometimes modified, but most accesses involve only reading. In these cases, it would be nice to relax the locking requirements to allow multiple readers to access the data structure at once. As long as each thread is guaranteed an up-to-date view of the data and no other thread modifies the data while the readers are viewing it, there will be no problems. This is what read-write locks allow: a resource can be accessed by multiple readers or a single writer at a time, but not both.

ReadWriteLock interface shown below exposes two Lock objects—one for reading and one for writing. To read data guarded by a ReadWriteLock you must first acquire the read lock, and to modify data guarded by a ReadWriteLock you must first acquire the write lock. While there may appear to be two separate locks, the read lock and write lock are simply different views of an integrated read-write lock object.

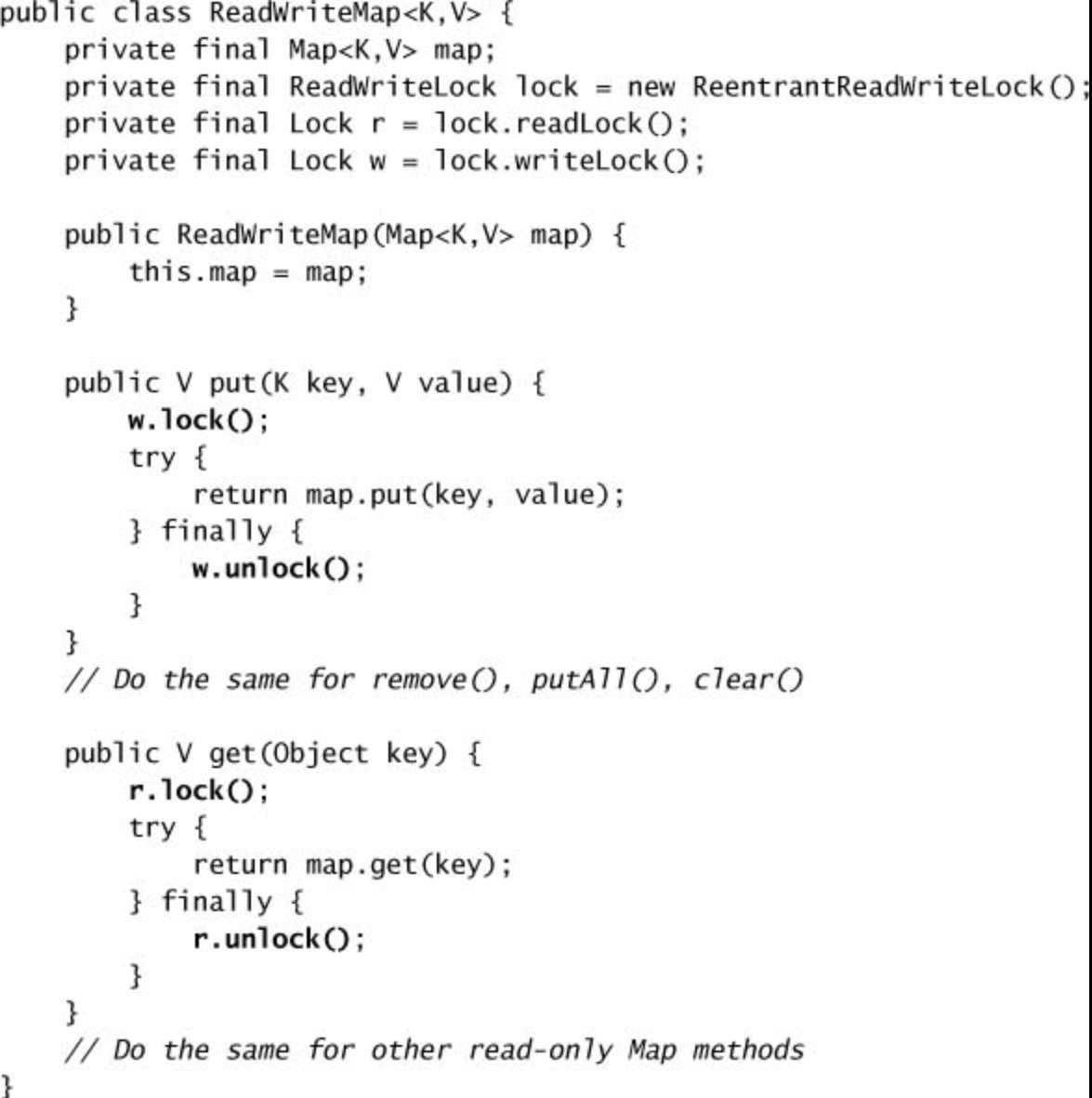


The locking strategy implemented by read-write locks allows multiple simultaneous readers but only a single writer. Like Lock, ReadWriteLock admits multiple implementations that can vary in performance, scheduling guarantees, acquisition preference, fairness, or locking semantics.

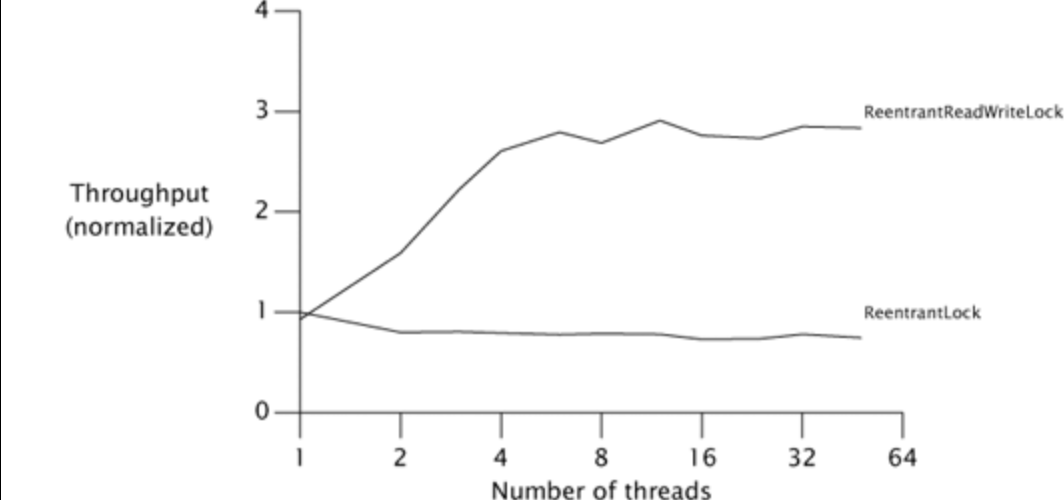
ReentrantReadWriteLock implements the ReadWriteLock interface. Like ReentrantLock, the writer lock in ReentrantReadWriteLock has a unique owner and can be released only by the thread that acquired it. In Java 5.0, the read lock behaves more like a Semaphore than a lock, maintaining only the count of active readers, not their identities. This behavior was changed in Java 6 to keep track also of which threads have been granted the read lock

Read-write locks can improve concurrency when locks are typically held for a moderately long time and most operations do not modify the guarded resources. ReadWriteMap in following listing uses a ReentrantReadWriteLock to wrap a Map so that it can be shared safely by multiple readers and still prevent reader-writer or writer-writer conflicts. In reality, ConcurrentHashMap's (another concurrent data structure, not one below) performance is so good that you would probably use it rather than this approach if all you needed was a concurrent hash-based map, *but this technique would be useful if you want to provide more concurrent access to an alternate Map implementation such as LinkedHashMap.*

ReadWriteMap does not implement Map because implementing the view methods such as entrySet and values would be difficult and the “easy” methods are usually sufficient.



Following graph shows a throughput comparison between an ArrayList wrapped with a ReentrantLock and with a ReantrantReadWriteLock on a four-way Opteron system running Solaris. In the test program used here, each operation randomly selects a value and searches for it in the collection, and a small percentange of operations modify the contents of the collection.



**Non-Blocking Algorithms & Atomic Variables**

Nonblocking algorithms maintain thread safety by using low-level concurrency primitives such as compare-and-swap instead of locks. These low-level primitives are exposed through the atomic variable classes, which can also be used as “better volatile variables” providing atomic update operations for integers and object references.

The trick to building nonblocking algorithms is to limit the scope of atomic changes to a single variable.

**CAS**

Compare And Swap (CASwap):

As the name suggests, the functionality is expected to swap and return the value of the variable with which it is trying a swap with! So no matter what happens with assignment to the value of that variable with new value the CAS is providing with, CAS functionality will always return that existing or old value of the variable.

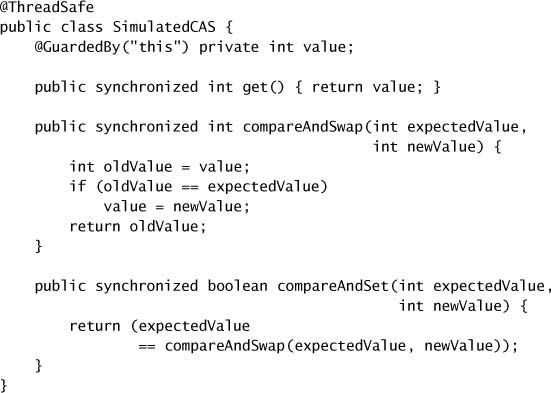
Compare And Set (CASet):

CASet works on top of CASwap. It gets the result of CASwap and compares with the expected value. If both match that means the swap has successfully taken place as the if condition in above CASet will swap only when the expected value will match with existing or old value in the variable.

There are three values of a variable considered below.

1. Existing or old value the variable currently has
2. New value that CASwap is trying to assign
3. Expected value by CASwap for a successful swapping. Expected value according to CASwap is the last value it assigned to the variable. It expects the variable has the same value unless some other thread has changed it!

Take a look at below code to understand how CASwap and CASet actually work at the processor level.



CAS at processor level addresses the problem of implementing atomic read-modify-write sequences without locking, because it can detect interference from other threads. It is the optimistic way of achieving atomicity of transaction rather than pessimistic way of locking.

**Atomic Variables**

There are twelve atomic variable classes, divided into four groups: scalars, field updaters, arrays, and compound variables.

**Scalars**

The most commonly used atomic variables are the scalars: AtomicInteger, AtomicLong, AtomicBoolean, and AtomicReference.

AtomicInteger:

The atomic variable classes provide a generalization of volatile variables to support atomic conditional read-modify-write operations. AtomicInteger represents an int value, and provides get and set methods with the same memory semantics as reads and writes to a volatile int.

It also provides an atomic compareAndSet method (which if successful has the memory effects of both reading and writing a volatile variable) and, for convenience, atomic add, increment, and decrement methods. AtomicInteger bears a superficial resemblance to an extended Counter class, but offers far greater scalability under contention because it can directly exploit underlying hardware support for concurrency.

While the atomic scalar classes extend Number, they do not extend the primitive wrapper classes such as Integer or Long. In fact, they cannot: the primitive wrapper classes are immutable whereas the atomic variable classes are mutable. The atomic variable classes also do not redefine hashCode or equals; each instance is distinct. Like most mutable objects, they are not good candidates for keys in hash-based collections.

**Arrays**

The atomic array classes (available in Integer, Long, and Reference versions) are arrays whose elements can be updated atomically. The atomic array classes provide volatile access semantics to the elements of the array, a feature not available for ordinary arrays—a volatile array has volatile semantics only for the array reference, not for its elements.