

CHAPTER 2

Thread Safety

Perhaps surprisingly, concurrent programming isn't so much about threads or locks, any more than civil engineering is about rivets and I-beams. Of course, building bridges that don't fall down requires the correct use of a lot of rivets and I-beams, just as 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.

By *shared*, we mean that a variable could be accessed by multiple threads; by *mutable*, we mean that its value could change during its lifetime. We may talk about thread safety as if it were about *code*, but what we are really trying to do is protect *data* from uncontrolled concurrent access.

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.

You should avoid the temptation to think that there are “special” situations in which this rule does not apply. A program that omits needed synchronization might appear to work, passing its tests and performing well for years, but it is still broken and may fail at any moment.

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.*

In a large program, identifying whether multiple threads might access a given variable can be complicated. Fortunately, 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 Java language doesn't force you to encapsulate state—it is perfectly allowable to store state in public fields (even public static fields) or publish a reference to an otherwise internal object—but the better encapsulated your program state, the easier it is to make your program thread-safe and to help maintainers keep it that way.

When designing thread-safe classes, good object-oriented techniques—encapsulation, immutability, and clear specification of invariants—are your best friends.

There will be times when good object-oriented design techniques are at odds with real-world requirements; it may be necessary in these cases to compromise the rules of good design for the sake of performance or for the sake of backward compatibility with legacy code. Sometimes abstraction and encapsulation are at odds with performance—although not nearly as often as many developers believe—but it is always a good practice first to make your code right, and *then* make it fast. Even then, pursue optimization only if your performance measurements and requirements tell you that you must, and if those same measurements tell you that your optimizations actually made a difference under realistic conditions.¹

If you decide that you simply must break encapsulation, all is not lost. It is still possible to make your program thread-safe, it is just a lot harder. Moreover, the

1. In concurrent code, this practice should be adhered to even more than usual. Because concurrency bugs are so difficult to reproduce and debug, the benefit of a small performance gain on some infrequently used code path may well be dwarfed by the risk that the program will fail in the field.

thread safety of your program will be more fragile, increasing not only development cost and risk but maintenance cost and risk as well. Chapter 4 characterizes the conditions under which it is safe to relax encapsulation of state variables.

We've used the terms "thread-safe class" and "thread-safe program" nearly interchangeably thus far. Is a thread-safe program one that is constructed entirely of thread-safe classes? Not necessarily—a program that consists entirely of thread-safe classes may not be thread-safe, and a thread-safe program may contain classes that are not thread-safe. The issues surrounding the composition of thread-safe classes are also taken up in Chapter 4. In any case, the concept of a thread-safe class makes sense only if the class encapsulates its own state. 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.

2.1 What is thread safety?

Defining thread safety is surprisingly tricky. The more formal attempts are so complicated as to offer little practical guidance or intuitive understanding, and the rest are informal descriptions that can seem downright circular. A quick Google search turns up numerous "definitions" like these:

...can be called from multiple program threads without unwanted interactions between the threads.

...may be called by more than one thread at a time without requiring any other action on the caller's part.

Given definitions like these, it's no wonder we find thread safety confusing! They sound suspiciously like "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.

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. Since we often don't write adequate specifications for our classes, how can we possibly know they are correct? We can't, but that doesn't stop us from using them anyway once we've convinced ourselves that "the code works". This "code confidence" is about as close as many of us get to correctness, so let's just assume that single-threaded correctness is something that "we know it when we see it". Having optimistically defined "correctness" as something that can be recognized, we can now define thread safety in a somewhat less circular way: a class is thread-safe when it continues to behave correctly when accessed from multiple threads.

A class is *thread-safe* if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code.

Since any single-threaded program is also a valid multithreaded program, it cannot be thread-safe if it is not even correct in a single-threaded environment.² 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.*

Thread-safe classes encapsulate any needed synchronization so that clients need not provide their own.

2.1.1 Example: a stateless servlet

In Chapter 1, we listed 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 shows our simple factorization servlet. It unpacks the number to be factored from the servlet request, factors it, and packages the results into the servlet response.

```
@ThreadSafe
public class StatelessFactorizer implements Servlet {
    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        encodeIntoResponse(resp, factors);
    }
}
```

LISTING 2.1. A stateless servlet.

2. If the loose use of “correctness” here bothers you, you may prefer to think of a thread-safe class as one that is no more broken in a concurrent environment than in a single-threaded environment.

`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.

The fact that most servlets can be implemented with no state greatly reduces the burden of making servlets thread-safe. It is only when servlets want to remember things from one request to another that the thread safety requirement becomes an issue.


2.2 Atomicity

What happens when we add one element of state to what was a stateless object? Suppose we want to add a “hit counter” that measures the number of requests processed. The obvious approach is to add a `long` field to the servlet and increment it on each request, as shown in `UnsafeCountingFactorizer` in Listing 2.2.

```
@NotThreadSafe
public class UnsafeCountingFactorizer implements Servlet {
    private long count = 0;

    public long getCount() { return count; }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        ++count;
        encodeIntoResponse(resp, factors);
    }
}
```



LISTING 2.2. Servlet that counts requests without the necessary synchronization. *Don't do this.*

Unfortunately, `UnsafeCountingFactorizer` is not thread-safe, even though it would work just fine in a single-threaded environment. Just like `UnsafeSequence` on page 6, it is susceptible to *lost updates*. 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.

Figure 1.1 on page 6 shows what can happen if two threads try to increment a counter simultaneously without synchronization. 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.

You might think that having a slightly inaccurate count of hits in a web-based service is an acceptable loss of accuracy, and sometimes it is. But if the counter is being used to generate sequences or unique object identifiers, returning the same value from multiple invocations could cause serious data integrity problems.³ 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*.

2.2.1 Race conditions

`UnsafeCountingFactorizer` has several *race conditions* that make its results unreliable. 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.⁴ 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.

We often encounter race conditions in real life. Let's say you planned to meet a friend at noon at the Starbucks on University Avenue. But when you get there, you realize there are *two* Starbucks on University Avenue, and you're not sure which one you agreed to meet at. At 12:10, you don't see your friend at Starbucks *A*, so you walk over to Starbucks *B* to see if he's there, but he isn't there either. There are a few possibilities: your friend is late and not at either Starbucks; your friend arrived at Starbucks *A* after you left; or your friend *was* at Starbucks *B*, but went to look for you, and is now en route to Starbucks *A*. Let's assume the worst and say it was the last possibility. Now it's 12:15, you've both been to both Starbucks, and you're both wondering if you've been stood up. What do you do now? Go back to the other Starbucks? How many times are you going to go back

3. The approach taken by `UnsafeSequence` and `UnsafeCountingFactorizer` has other serious problems, including the possibility of stale data (Section 3.1.1).

4. 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 nonfinal field. You risk a data race whenever a thread writes a variable that might next be read by another thread or reads a variable that might have last been written by another thread if both threads do not use synchronization; code with data races has no useful defined semantics under the Java Memory Model. Not all race conditions are data races, and not all data races are race conditions, but they both can cause concurrent programs to fail in unpredictable ways. `UnsafeCountingFactorizer` has both race conditions and data races. See Chapter 16 for more on data races.

and forth? Unless you have agreed on a protocol, you could both spend the day walking up and down University Avenue, frustrated and undercaffeinated.

The problem with the “I’ll just nip up the street and see if he’s at the other one” approach is that while you’re walking up the street, your friend might have moved. You look around Starbucks *A*, observe “he’s not here”, and go looking for him. And you can do the same for Starbucks *B*, but *not at the same time*. It takes a few minutes to walk up the street, and during those few minutes, *the state of the system may have changed*.

The Starbucks example illustrates a race condition because reaching the desired outcome (meeting your friend) depends on the relative timing of events (when each of you arrives at one Starbucks or the other, how long you wait there before switching, etc). The observation that he is not at Starbucks *A* becomes potentially invalid as soon as you walk out the front door; he could have come in through the back door and you wouldn’t know. It is this invalidation of observations that characterizes most race conditions—using a potentially stale observation to make a decision or perform a computation. This type of race condition is called *check-then-act*: you observe something to be true (file *X* doesn’t exist) and then take action based on that observation (create *X*); but in fact the observation could have become invalid between the time you observed it and the time you acted on it (someone else created *X* in the meantime), causing a problem (unexpected exception, overwritten data, file corruption).

2.2.2 Example: race conditions in lazy initialization

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 it is initialized only once. `LazyInitRace` in Listing 2.3 illustrates the lazy initialization idiom. The `getInstance` method first checks whether the `ExpensiveObject` has already been initialized, in which case it returns the existing instance; otherwise it creates a new instance and returns it after retaining a reference to it so that future invocations can avoid the more expensive code path.

```
@NotThreadSafe
public class LazyInitRace {
    private ExpensiveObject instance = null;

    public ExpensiveObject getInstance() {
        if (instance == null)
            instance = new ExpensiveObject();
        return instance;
    }
}
```



LISTING 2.3. Race condition in lazy initialization. *Don’t do this.*

`LazyInitRace` has race conditions that can undermine its correctness. Say that threads *A* and *B* execute `getInstance` at the same time. *A* sees that `instance` is `null`, and instantiates a new `ExpensiveObject`. *B* also checks if `instance` is `null`. Whether `instance` is `null` at this point depends unpredictably on timing, including the vagaries of scheduling and how long *A* takes to instantiate the `ExpensiveObject` and set the `instance` field. If `instance` is `null` when *B* examines it, the two callers to `getInstance` may receive two different results, even though `getInstance` is always supposed to return the same instance.

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.

Like most concurrency errors, race conditions don't *always* result in failure: some unlucky timing is also required. But race conditions can cause serious problems. If `LazyInitRace` is used to instantiate an application-wide registry, having it return different instances from multiple invocations could cause registrations to be lost or multiple activities to have inconsistent views of the set of registered objects. If `UnsafeSequence` is used to generate entity identifiers in a persistence framework, two distinct objects could end up with the same ID, violating identity integrity constraints.

2.2.3 Compound actions

Both `LazyInitRace` and `UnsafeCountingFactorizer` contained a sequence of operations that needed to be *atomic*, or indivisible, relative to other operations on the same state. To avoid race conditions, there must be a way to prevent other threads from using a variable while we're in the middle of modifying it, so we can ensure that other threads can observe or modify the state only before we start or after we finish, but not in the middle.

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.

If the increment operation in `UnsafeSequence` were atomic, the race condition illustrated in Figure 1.1 on page 6 could not occur, and each execution of the increment operation would have the desired effect of incrementing the counter by exactly one. 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. In the next section, we'll consider *locking*, Java's built-in mechanism for ensuring atomicity. For now, we're going to fix the problem

another way, by using an existing thread-safe class, as shown in `CountingFactorizer` in Listing 2.4.

```
@ThreadSafe
public class CountingFactorizer implements Servlet {
    private final AtomicLong count = new AtomicLong(0);

    public long getCount() { return count.get(); }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        count.incrementAndGet();
        encodeIntoResponse(resp, factors);
    }
}
```

LISTING 2.4. Servlet that counts requests using `AtomicLong`.

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.⁵ Because the state of the servlet *is* the state of the counter and the counter is thread-safe, our servlet is once again thread-safe.

We were able to add a counter to our factoring servlet and maintain thread safety by using an existing thread-safe class to manage the counter state, `AtomicLong`. When a *single* element of state is added to a stateless class, the resulting class will be thread-safe if the state is entirely managed by a thread-safe object. But, as we'll see in the next section, going from one state variable to more than one is not necessarily as simple as going from zero to one.

Where practical, use existing thread-safe objects, like `AtomicLong`, to manage your class's state. It is simpler to reason about the possible states and state transitions for existing thread-safe objects than it is for arbitrary state variables, and this makes it easier to maintain and verify thread safety.

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

5. `CountingFactorizer` calls `incrementAndGet` to increment the counter, which also returns the incremented value; in this case the return value is ignored.

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. (This is unlikely to be an effective caching strategy; we offer a better one in Section 5.6.) To implement this strategy, we need to remember two things: the last number factored, and its factors.

We used `AtomicLong` to manage the counter state in a thread-safe manner; could we perhaps use its cousin, `AtomicReference`,⁶ to manage the last number and its factors? An attempt at this is shown in `UnsafeCachingFactorizer` in Listing 2.5.

```
@NotThreadSafe
public class UnsafeCachingFactorizer implements Servlet {
    private final AtomicReference<BigInteger> lastNumber
        = new AtomicReference<BigInteger>();
    private final AtomicReference<BigInteger[]> lastFactors
        = new AtomicReference<BigInteger[]>();

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        if (i.equals(lastNumber.get()))
            encodeIntoResponse(resp, lastFactors.get());
        else {
            BigInteger[] factors = factor(i);
            lastNumber.set(i);
            lastFactors.set(factors);
            encodeIntoResponse(resp, factors);
        }
    }
}
```



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

6. Just as `AtomicLong` is a thread-safe holder class for a `long` integer, `AtomicReference` is a thread-safe holder class for an object reference. Atomic variables and their benefits are covered in Chapter 15.

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.

2.3.1 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.) 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.)

```
synchronized (lock) {  
    // Access or modify shared state guarded by lock  
}
```

Every Java object can implicitly act as a lock for purposes of synchronization; these built-in locks are called *intrinsic locks* or *monitor locks*. The lock is automatically acquired by the executing thread before entering a synchronized block and automatically released when control exits the synchronized block, whether by the normal control path or by throwing an exception out of the block. The only way to acquire an intrinsic lock is to enter a synchronized block or method guarded by that lock.

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.

Since only one thread at a time can execute a block of code guarded by a given lock, the synchronized blocks guarded by the same lock execute atomically with respect to one another. In the context of concurrency, atomicity means the same thing as it does in transactional applications—that a group of statements appear to execute as a single, indivisible unit. No thread executing a synchronized block

can observe another thread to be in the middle of a synchronized block guarded by the same lock.

The machinery of synchronization makes it easy to restore thread safety to the factoring servlet. Listing 2.6 makes the service method synchronized, so only one thread may enter service at a time. SynchronizedFactorizer is now thread-safe; however, this approach is fairly extreme, since it inhibits multiple clients from using the factoring servlet simultaneously at all—resulting in unacceptably poor responsiveness. This problem—which is a performance problem, not a thread safety problem—is addressed in Section 2.5.

@ThreadSafe

```
public class SynchronizedFactorizer implements Servlet {
    @GuardedBy("this") private BigInteger lastNumber;
    @GuardedBy("this") private BigInteger[] lastFactors;

    public synchronized void service(ServletRequest req,
                                    ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        if (i.equals(lastNumber))
            encodeIntoResponse(resp, lastFactors);
        else {
            BigInteger[] factors = factor(i);
            lastNumber = i;
            lastFactors = factors;
            encodeIntoResponse(resp, factors);
        }
    }
}
```



LISTING 2.6. Servlet that caches last result, but with unacceptably poor concurrency. *Don't do this.*

2.3.2 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.⁷ 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

7. This differs from the default locking behavior for pthreads (POSIX threads) mutexes, which are granted on a per-invocation basis.

is incremented, and when the owning thread exits the synchronized block, the count is decremented. When the count reaches zero, the lock is released.

Reentrancy facilitates encapsulation of locking behavior, and thus simplifies the development of object-oriented concurrent code. Without reentrant locks, the very natural-looking code in Listing 2.7, in which a subclass overrides a synchronized method and then calls the superclass method, would deadlock. Because the `doSomething` methods in `Widget` and `LoggingWidget` are both synchronized, each tries to acquire the lock on the `Widget` before proceeding. But if intrinsic locks were not reentrant, the call to `super.doSomething` would never be able to acquire the lock because it would be considered already held, and the thread would permanently stall waiting for a lock it can never acquire. Reentrancy saves us from deadlock in situations like this.

```
public class Widget {
    public synchronized void doSomething() {
        ...
    }
}

public class LoggingWidget extends Widget {
    public synchronized void doSomething() {
        System.out.println(toString() + ": calling doSomething");
        super.doSomething();
    }
}
```

LISTING 2.7. Code that would deadlock if intrinsic locks were not reentrant.

2.4 Guarding state with locks

Because locks enable serialized⁸ 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.

8. Serializing access to an object has nothing to do with object serialization (turning an object into a byte stream); serializing access means that threads take turns accessing the object exclusively, rather than doing so concurrently.

It is a common mistake to assume that synchronization needs to be used only when *writing* to shared variables; *this is simply not true*. (The reasons for this will become clearer in Section 3.1.)

For each mutable state variable that may be accessed by more than one thread, *all* accesses to that variable must be performed with the *same* lock held. In this case, we say that the variable is *guarded by* that lock.

In `SynchronizedFactorizer` in Listing 2.6, `lastNumber` and `lastFactors` are guarded by the servlet object's intrinsic lock; this is documented by the `@GuardedBy` annotation.

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.

Every shared, mutable variable should be guarded by exactly one lock. Make it clear to maintainers which lock that is.

A common locking convention is to encapsulate all mutable state within an object and to protect it from concurrent access by synchronizing any code path that accesses mutable state using the object's intrinsic lock. This pattern is used by many thread-safe classes, such as `Vector` and other synchronized collection classes. In such cases, all the variables in an object's state are guarded by the object's intrinsic lock. However, there is nothing special about this pattern, and neither the compiler nor the runtime enforces this (or any other) pattern of locking.¹⁰ It is also easy to subvert this locking protocol accidentally by adding a new method or code path and forgetting to use synchronization.

Not all data needs to be guarded by locks—only mutable data that will be accessed from multiple threads. In Chapter 1, we described how adding a simple asynchronous event such as a `TimerTask` can create thread safety requirements that ripple throughout your program, especially if your program state is poorly encapsulated. Consider a single-threaded program that processes a large amount of data. Single-threaded programs require no synchronization, because no data is shared across threads. Now imagine you want to add a feature to create periodic

9. In retrospect, this design decision was probably a bad one: not only can it be confusing, but it forces JVM implementors to make tradeoffs between object size and locking performance.

10. Code auditing tools like FindBugs can identify when a variable is frequently but not always accessed with a lock held, which may indicate a bug.

snapshots of its progress, so that it does not have to start again from the beginning if it crashes or must be stopped. You might choose to do this with a `TimerTask` that goes off every ten minutes, saving the program state to a file.

Since the `TimerTask` will be called from another thread (one managed by `Timer`), any data involved in the snapshot is now accessed by two threads: the main program thread and the `Timer` thread. This means that not only must the `TimerTask` code use synchronization when accessing the program state, but so must any code path in the rest of the program that touches that same data. What used to require no synchronization now requires synchronization throughout the program.

When a variable is guarded by a lock—meaning that *every* access to that variable is performed with that lock held—you’ve ensured that only one thread at a time can access that variable. When a class has invariants that involve more than one state variable, there is an additional requirement: each variable participating in the invariant must be guarded by the *same* lock. This allows you to access or update them in a single atomic operation, preserving the invariant. `SynchronizedFactorizer` demonstrates this rule: both the cached number and the cached factors are guarded by the servlet object’s intrinsic lock.

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:

```
if (!vector.contains(element))  
    vector.add(element);
```

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. (See Section 4.4 for some techniques for safely adding additional atomic operations to thread-safe objects.) At the same time, synchronizing every method can lead to liveness or performance problems, as we saw in `SynchronizedFactorizer`.

2.5 Liveness and performance

In `UnsafeCachingFactorizer`, we introduced some caching into our factoring servlet in the hope of improving performance. Caching required some shared state, which in turn required synchronization to maintain the integrity of that state. But the way we used synchronization in `SynchronizedFactorizer` makes it perform badly. The synchronization policy for `SynchronizedFactorizer` is to

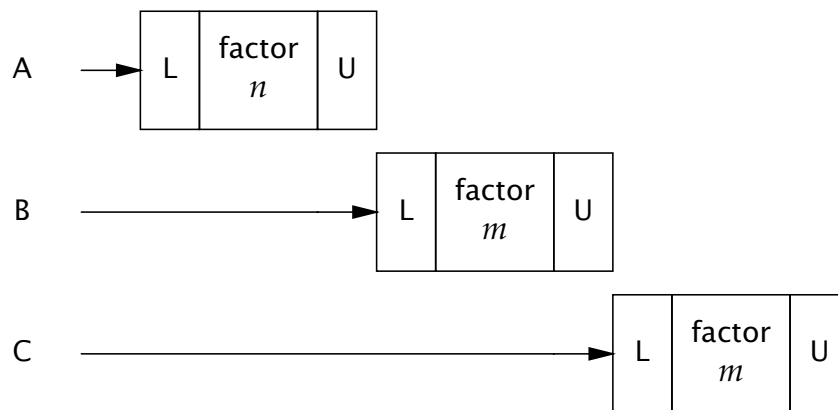


FIGURE 2.1. Poor concurrency of SynchronizedFactorizer.

guard each state variable with the servlet object’s intrinsic lock, and that policy was implemented by synchronizing the entirety of the service method. This simple, coarse-grained approach restored safety, but at a high price.

Because service is synchronized, only one thread may execute it at once. This subverts the intended use of the servlet framework—that servlets be able to handle multiple requests simultaneously—and can result in frustrated users if the load is high enough. If the servlet is busy factoring a large number, other clients have to wait until the current request is complete before the servlet can start on the new number. If the system has multiple CPUs, processors may remain idle even if the load is high. In any case, even short-running requests, such as those for which the value is cached, may take an unexpectedly long time because they must wait for previous long-running requests to complete.

Figure 2.1 shows what happens when multiple requests arrive for the synchronized factoring servlet: they queue up and are handled sequentially. We would describe this web application as exhibiting *poor concurrency*: the number of simultaneous invocations is limited not by the availability of processing resources, but by the structure of the application itself. Fortunately, it is easy to improve the concurrency of the servlet while maintaining thread safety by narrowing the scope of the synchronized block. You should be careful not to make the scope of the synchronized block *too* small; you would not want to divide an operation that should be atomic into more than one synchronized block. But it is reasonable to try to exclude from synchronized blocks long-running operations that do not affect shared state, so that other threads are not prevented from accessing the shared state while the long-running operation is in progress.

CachedFactorizer in Listing 2.8 restructures the servlet to use two separate synchronized blocks, each limited to a short section of code. One guards the check-then-act sequence that tests whether we can just return the cached result, and the other guards updating both the cached number and the cached factors. As a bonus, we’ve reintroduced the hit counter and added a “cache hit” counter as well, updating them within the initial synchronized block. Because these counters constitute shared mutable state as well, we must use synchronization everywhere they are accessed. The portions of code that are outside the synchronized blocks operate exclusively on local (stack-based) variables, which are not

shared across threads and therefore do not require synchronization.

```

@ThreadSafe
public class CachedFactorizer implements Servlet {
    @GuardedBy("this") private BigInteger lastNumber;
    @GuardedBy("this") private BigInteger[] lastFactors;
    @GuardedBy("this") private long hits;
    @GuardedBy("this") private long cacheHits;

    public synchronized long getHits() { return hits; }
    public synchronized double getCacheHitRatio() {
        return (double) cacheHits / (double) hits;
    }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = null;
        synchronized (this) {
            ++hits;
            if (i.equals(lastNumber)) {
                ++cacheHits;
                factors = lastFactors.clone();
            }
        }
        if (factors == null) {
            factors = factor(i);
            synchronized (this) {
                lastNumber = i;
                lastFactors = factors.clone();
            }
        }
        encodeIntoResponse(resp, factors);
    }
}

```

LISTING 2.8. Servlet that caches its last request and result.

`CachedFactorizer` no longer uses `AtomicLong` for the hit counter, instead reverting to using a `long` field. It would be safe to use `AtomicLong` here, but there is less benefit than there was in `CountingFactorizer`. Atomic variables are useful for effecting atomic operations on a single variable, but since we are already using `synchronized` blocks to construct atomic operations, using two different synchronization mechanisms would be confusing and would offer no performance or safety benefit.

The restructuring of `CachedFactorizer` provides a balance between simplicity (synchronizing the entire method) and concurrency (synchronizing the short-

est possible code paths). Acquiring and releasing a lock has some overhead, so it is undesirable to break down synchronized blocks *too* far (such as factoring `++hits` into its own synchronized block), even if this would not compromise atomicity. `CachedFactorizer` holds the lock when accessing state variables and for the duration of compound actions, but releases it before executing the potentially long-running factorization operation. This preserves thread safety without unduly affecting concurrency; the code paths in each of the synchronized blocks are “short enough”.

Deciding how big or small to make synchronized blocks may require tradeoffs among competing design forces, including safety (which must not be compromised), simplicity, and performance. Sometimes simplicity and performance are at odds with each other, although as `CachedFactorizer` illustrates, a reasonable balance can usually be found.

There is frequently a tension between simplicity and performance. When implementing a synchronization policy, resist the temptation to prematurely sacrifice simplicity (potentially compromising safety) for the sake of performance.

Whenever you use locking, you should be aware of what the code in the block is doing and how likely it is to take a long time to execute. Holding a lock for a long time, either because you are doing something compute-intensive or because you execute a potentially blocking operation, introduces the risk of liveness or performance problems.

Avoid holding locks during lengthy computations or operations at risk of not completing quickly such as network or console I/O.