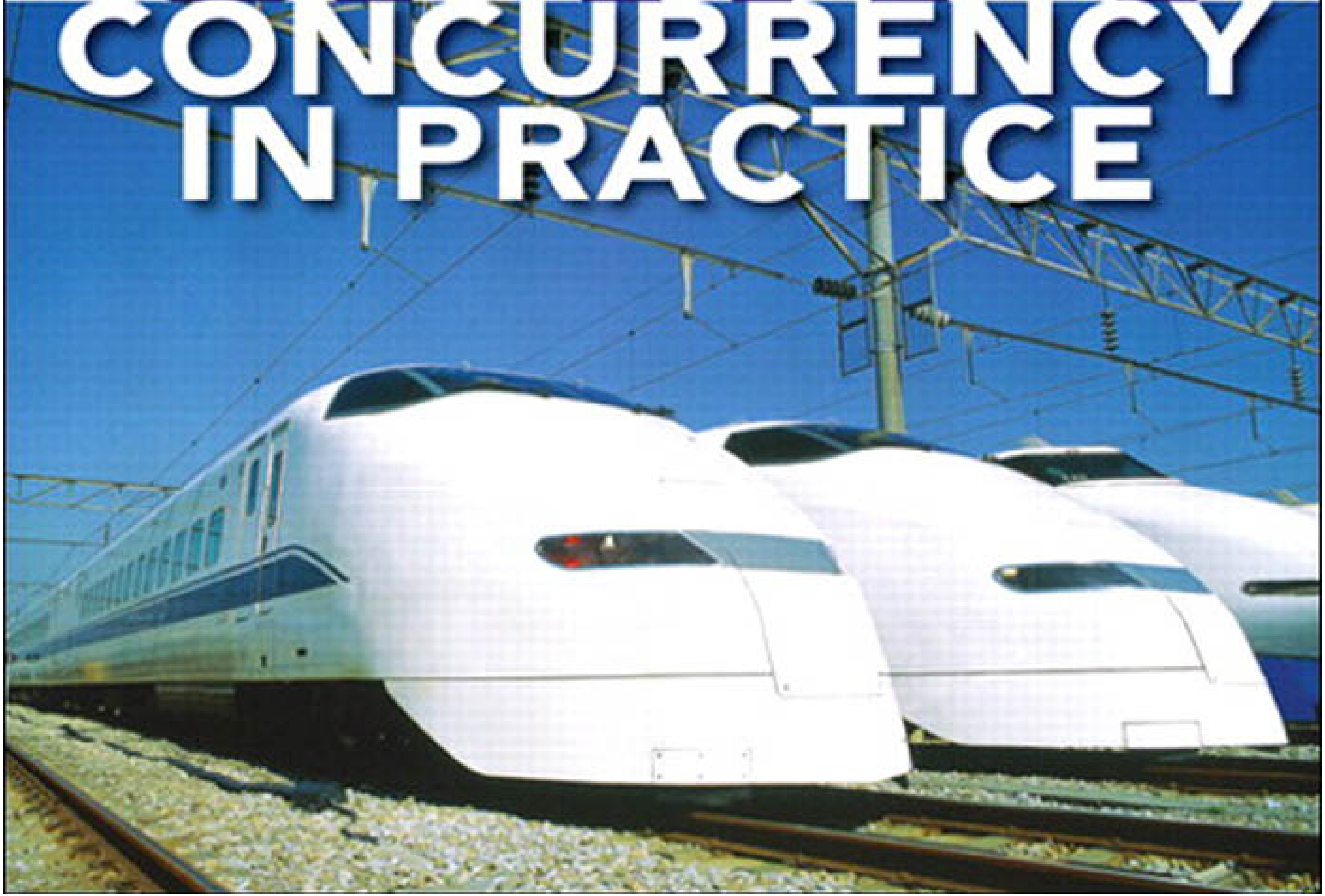
BRIAN GOETZ

WITH TIM PEIERLS, JOSHUA BLOCH,
JOSEPH BOWBEER, DAVID HOLMES,
AND DOUG LEA

JAVA CONCURRENCY IN PRACTICE



Chapter 1

Introduction

Writing correct programs is hard; writing correct concurrent programs is harder. There are simply more things that can go wrong in a concurrent program than in a sequential one. So, why do we bother with concurrency? Threads are an inescapable feature of the Java language, and they can simplify the development of complex systems by turning complicated asynchronous code into simpler straight-line code. In addition, threads are the easiest way to tap the computing power of multiprocessor systems. And, as processor counts increase, exploiting concurrency effectively will only become more important.

1.1 A (very) brief history of concurrency

In the ancient past, computers didn't have operating systems; they executed a single program from beginning to end, and that program had direct access to all the resources of the machine. Not only was it difficult to write programs that ran on the bare metal, but running only a single program at a time was an inefficient use of expensive and scarce computer resources.

Operating systems evolved to allow more than one program to run at once, running individual programs in *processes*: isolated, independently executing programs to which the operating system allocates resources such as memory, file handles, and security credentials. If they needed to, processes could communicate with one another through a variety of coarse-grained communication mechanisms: sockets, signal handlers, shared memory, semaphores, and files.

Several motivating factors led to the development of operating systems that allowed multiple programs to execute simultaneously:

Resource utilization. Programs sometimes have to wait for external operations such as input or output, and while waiting can do no useful work. It is more efficient to use that wait time to let another program run.

Fairness. Multiple users and programs may have equal claims on the machine's resources. It is preferable to let them share the computer via finer-grained time slicing than to let one program run to completion and then start another.

Convenience. It is often easier or more desirable to write several programs that each perform a single task and have them coordinate with each other as necessary than to write a single program that performs all the tasks.

In early timesharing systems, each process was a virtual von Neumann computer; it had a memory space storing both instructions and data, executing instructions sequentially according to the semantics of the machine language, and interacting with the outside world via the operating system through a set of I/O primitives. For each instruction executed there was a clearly defined "next instruction", and control flowed through the program according to the rules of the instruction set. Nearly all widely used programming languages today follow this sequential programming model, where the language specification clearly defines "what comes next" after a given action is executed.

The sequential programming model is intuitive and natural, as it models the way humans work: do one thing at a time, in sequence—mostly. Get out of bed, put on your bathrobe, go downstairs and start the tea. As in programming languages, each of these real-world actions is an abstraction for a sequence of finer-grained actions—open the cupboard, select a flavor of tea, measure some tea into the pot, see if there's enough water in the teakettle, if not put some more water in, set it on the stove, turn the stove on, wait for the water to boil, and so on. This last step—waiting for the water to boil—also involves a degree of *asynchrony*. While the water is heating, you have a choice of what to do—just wait, or do other tasks in that time such as starting the toast (another asynchronous task) or fetching the newspaper, while remaining aware that your attention will soon be needed by the teakettle. The manufacturers of teakettles and toasters know their products are often used in an asynchronous manner, so they raise an audible signal when they complete their task. Finding the right balance of sequentiality and asynchrony is often a characteristic of efficient people—and the same is true of programs.

The same concerns (resource utilization, fairness, and convenience) that motivated the development of processes also motivated the development of *threads*. Threads allow multiple streams of program control flow to coexist within a process. They share process-wide resources such as memory and file handles, but each thread has its own program counter, stack, and local variables. Threads also provide a natural decomposition for exploiting hardware parallelism on multiprocessor systems; multiple threads within the same program can be scheduled simultaneously on multiple CPUs.

Threads are sometimes called *lightweight processes*, and most modern operating systems treat threads, not processes, as the basic units of scheduling. In the absence of explicit coordination, threads execute simultaneously and asynchronously with respect to one another. Since threads share the memory address space of their owning process, all threads within a process have access to the same variables and allocate objects from the same heap, which allows finer-grained data sharing than inter-process mechanisms. But without explicit synchronization to coordinate access to shared data, a thread may modify variables that another thread is in the middle of using, with unpredictable results.

1.2 Benefits of threads

When used properly, threads can reduce development and maintenance costs and improve the performance of complex applications. Threads make it easier to model how humans work and interact, by turning asynchronous workflows into mostly sequential ones. They can also turn otherwise convoluted code into straight-line code that is easier to write, read, and maintain.

Threads are useful in GUI applications for improving the responsiveness of the user interface, and in server applications for improving resource utilization and throughput. They also simplify the implementation of the JVM—the garbage collector usually runs in one or more dedicated threads. Most nontrivial Java applications rely to some degree on threads for their organization.

1.2.1 Exploiting multiple processors

Multiprocessor systems used to be expensive and rare, found only in large data centers and scientific computing facilities. Today they are cheap and plentiful; even low-end server and midrange desktop systems often have multiple processors. This trend will only accelerate; as it gets harder to scale up clock rates, processor manufacturers will instead put more processor cores on a single chip. All the major chip manufacturers have begun this transition, and we are already seeing machines with dramatically higher processor counts.

Since the basic unit of scheduling is the thread, a program with only one thread can run on at most one processor at a time. On a two-processor system, a single-threaded program is giving up access to half the available CPU resources; on a 100-processor system, it is giving up access to 99%. On the other hand, programs with multiple active threads can execute simultaneously on multiple processors. When properly designed, multithreaded programs can improve throughput by utilizing available processor resources more effectively.

Using multiple threads can also help achieve better throughput on single-processor systems. If a program is single-threaded, the processor remains idle while it waits for a synchronous I/O operation to complete. In a multithreaded program, another thread can still run while the first thread is waiting for the I/O to complete, allowing the application to still make progress during the blocking I/O. (This is like reading the newspaper while waiting for the water to boil, rather than waiting for the water to boil before starting to read.)

1.2.2 Simplicity of modeling

It is often easier to manage your time when you have only one type of task to perform (fix these twelve bugs) than when you have several (fix the bugs, interview replacement candidates for the system administrator, complete your team's performance evaluations, and create the slides for your presentation next week). When you have only one type of task to do, you can start at the top of the pile and keep working until the pile is exhausted (or you are); you don't have to spend any mental energy figuring out what to work on next. On the other hand, managing multiple priorities and deadlines and switching from task to task usually carries some overhead.

The same is true for software: a program that processes one type of task sequentially is simpler to write, less error-prone, and easier to test than one managing multiple different types of tasks at once. Assigning a thread to each type of task or to each element in a simulation affords the illusion of sequentiality and insulates domain logic from the details of scheduling, interleaved operations, asynchronous I/O, and resource waits. A complicated, asynchronous workflow can be decomposed into a number of simpler, synchronous workflows each running in a separate thread, interacting only with each other at specific synchronization points.

This benefit is often exploited by frameworks such as servlets or RMI (Remote Method Invocation). The framework handles the details of request management, thread creation, and load balancing, dispatching portions of the request handling to the appropriate application component at the appropriate point in the workflow. Servlet writers do not need to worry about how many other requests are being processed at the same time or whether the socket input and output streams block; when a servlet's `service` method is called in response to a web request, it can process the request synchronously as if it were a single-threaded program. This can simplify component development and reduce the learning curve for using such frameworks.

1.2.3 Simplified handling of asynchronous events

A server application that accepts socket connections from multiple remote clients may be easier to develop when each connection is allocated its own thread and allowed to use synchronous I/O.

If an application goes to read from a socket when no data is available, `read` blocks until some data is available. In a single-threaded application, this means that not only does processing the corresponding request stall, but processing of *all* requests stalls while the single thread is blocked. To avoid this problem, single-threaded server applications are forced to use nonblocking I/O, which is far more complicated and error-prone than synchronous I/O. However, if each request has its own thread, then blocking does not affect the processing of other requests.

Historically, operating systems placed relatively low limits on the number of threads that a process could create, as few as several hundred (or even less). As a result, operating systems developed efficient facilities for multiplexed I/O, such as the Unix `select` and `poll` system calls, and to access these facilities, the Java class libraries acquired a set of packages (`java.nio`) for nonblocking I/O. However, operating system support for larger numbers of threads has improved significantly, making the thread-per-client model practical even for large numbers of clients on some platforms.[1]

[1] The NPTL threads package, now part of most Linux distributions, was designed to support hundreds of thousands of threads. Nonblocking I/O has its own benefits, but better OS support for threads means that there are fewer situations for which it is *essential*.

1.2.4 More responsive user interfaces

GUI applications used to be single-threaded, which meant that you had to either frequently poll throughout the code for input events (which is messy and intrusive) or execute all application code indirectly through a "main event loop". If code called from the main event loop takes too long to execute, the user interface appears to "freeze" until that code finishes, because subsequent user interface events cannot be processed until control is returned to the main event loop.

Modern GUI frameworks, such as the AWT and Swing toolkits, replace the main event loop with an *event dispatch thread* (EDT). When a user interface event such as a button press occurs, application-defined event handlers are called in the event thread. Most GUI frameworks are single-threaded subsystems, so the main event loop is effectively still present, but it runs in its own thread under the control of the GUI toolkit rather than the application.

If only short-lived tasks execute in the event thread, the interface remains responsive since the event thread is always able to process user actions reasonably quickly. However, processing a long-running task in the event thread, such as spell-checking a large document or fetching a resource over the network, impairs responsiveness. If the user performs an action while this task is running, there is a long delay before the event thread can process or even acknowledge it. To add insult to injury, not only does the UI become unresponsive, but it is impossible to cancel the offending task even if the UI provides a cancel button because the event thread is busy and cannot handle the cancel button-press event until the lengthy task completes! If, however, the long-running task is instead executed in a separate thread, the event thread remains free to process UI events, making the UI more responsive.

1.3 Risks of threads

Java's built-in support for threads is a double-edged sword. While it simplifies the development of concurrent applications by providing language and library support and a formal cross-platform memory model (it is this formal cross-platform memory model that makes possible the development of write-once, run-anywhere *concurrent* applications in Java), it also raises the bar for developers because more programs will use threads. When threads were more esoteric, concurrency was an "advanced" topic; now, mainstream developers must be aware of thread-safety issues.

1.3.1 Safety 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. `UnsafeSequence` in Listing 1.1, which is supposed to generate a sequence of unique integer values, offers a simple illustration of how the interleaving of actions in multiple threads can lead to undesirable results. It behaves correctly in a single-threaded environment, but in a multithreaded environment does not.

```
@NotThreadSafe
public class UnsafeSequence {
    private int value;

    /** Returns a unique value. */
    public int getNext() {
        return value++;
    }
}
```

LISTING 1.1. Non-thread-safe sequence generator.

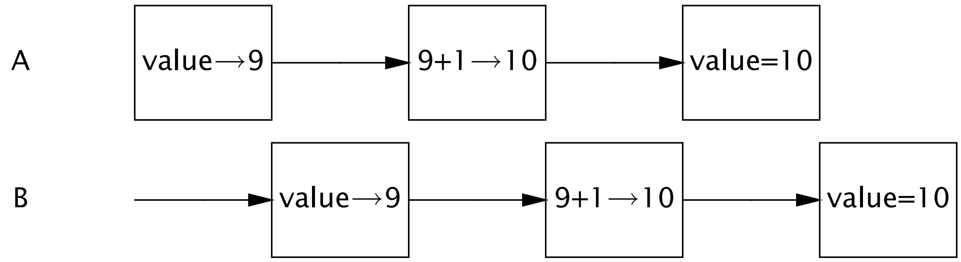


FIGURE 1.1. Unlucky execution of UnsafeSequence.getNext.

The problem with UnsafeSequence is that with some unlucky timing, two threads could call getNext and receive *the same value*. Figure 1.1 shows how this can happen. The increment notation, someVariable++, may *appear* to be a single operation, but is in fact three separate operations: read the value, add one to it, and write out the new value. Since operations in multiple threads may be arbitrarily interleaved by the runtime, it is possible for two threads to read the value at the same time, both see the same value, and then both add one to it. The result is that the same sequence number is returned from multiple calls in different threads.

> Diagrams like Figure 1.1 depict possible interleavings of operations in different threads. In these diagrams, time runs from left to right, and each line represents the activities of a different thread. These interleaving diagrams usually depict the worst case[2] and are intended to show the danger of incorrectly assuming things will happen in a particular order.

UnsafeSequence uses a nonstandard annotation: @NotThreadSafe. This is one of several custom annotations used throughout this book to document concurrency properties of classes and class members. (Other class-level annotations used

[2] Actually, as we'll see in Chapter 3, the worst case can be even worse than these diagrams usually show because of the possibility of reordering.

in this way are @ThreadSafe and @Immutable; see Appendix A for details.) Annotations documenting thread safety are useful to multiple audiences. If a class is annotated with @ThreadSafe, users can use it with confidence in a multithreaded environment, maintainers are put on notice that it makes thread safety guarantees that must be preserved, and software analysis tools can identify possible coding errors.

UnsafeSequence illustrates a common concurrency hazard called a *race condition*. Whether or not getNext returns a unique value when called from multiple threads, as required by its specification, depends on how the runtime interleaves the operations—which is not a desirable state of affairs.

Because threads share the same memory address space and run concurrently, they can access or modify variables that other threads might be using. This is a tremendous convenience, because it makes data sharing much easier than would other inter-thread communications mechanisms. But it is also a significant risk: threads can be confused by having data change unexpectedly. Allowing multiple threads to access and modify the same variables introduces an element of nonsequentiality into an otherwise sequential programming model, which can be confusing and difficult to reason about. For a multithreaded program's behavior to be predictable, access to shared variables must be properly coordinated so that threads do not interfere with one another. Fortunately, Java provides synchronization mechanisms to coordinate such access.

UnsafeSequence can be fixed by making getNext a synchronized method, as shown in Sequence in Listing 1.2,[3] thus preventing the unfortunate interaction in Figure 1.1. (Exactly why this works is the subject of Chapters 2 and 3.)

```
@ThreadSafe
public class Sequence {
    @GuardedBy("this") private int value;

    public synchronized int getNext() {
        return value++;
    }
}
```

Listing 1.2. Thread-safe sequence generator.

In the absence of synchronization, the compiler, hardware, and runtime are allowed to take substantial liberties with the timing and ordering of actions, such as caching variables in registers or processor-local caches where they are temporarily (or even permanently) invisible to other threads. These tricks are in aid of better performance and are generally desirable, but they place a burden on the developer to clearly identify where data is being shared across threads so that these optimizations do not undermine safety. (Chapter 16 gives the gory details on exactly what ordering guarantees the JVM makes and how synchronization

[3] @GuardedBy is described in Section 2.4; it documents the *synchronization policy* for Sequence.

affects those guarantees, but if you follow the rules in Chapters 2 and 3, you can safely avoid these low-level details.)

1.3.2 Liveness hazards

It is critically important to pay attention to thread safety issues when developing concurrent code: safety cannot be compromised. The importance of safety is not unique to multithreaded programs—single-threaded programs also must take care to preserve safety and correctness—but the use of threads introduces additional safety hazards not present in single-threaded programs. 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. The use of threads introduces additional liveness risks. For example, if thread *A* is waiting for a resource that thread *B* holds exclusively, and *B* never releases it, *A* will wait forever. Chapter 10 describes various forms of liveness failures and how to avoid them, including deadlock (Section 10.1), starvation (Section 10.3.1), and livelock (Section 10.3.3). Like most concurrency bugs, bugs that cause liveness failures can be elusive because they depend on the relative timing of events in different threads, and therefore do not always manifest themselves in development or testing.

1.3.3 Performance hazards

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. Just as with safety and liveness, multithreaded programs are subject to all the performance hazards of single-threaded programs, and to others as well that are introduced by the use of threads.

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. When threads share data, they must use synchronization mechanisms that can inhibit compiler optimizations, flush or invalidate memory caches, and create synchronization traffic on the shared memory bus. All these factors introduce additional performance costs; Chapter 11 covers techniques for analyzing and reducing these costs.

1.4 Threads are everywhere

Even if your program never explicitly creates a thread, frameworks may create threads on your behalf, and code called from these threads must be thread-safe. This can place a significant design and implementation burden on developers, since developing thread-safe classes requires more care and analysis than developing non-thread-safe classes.

Every Java application uses threads. When the JVM starts, it creates threads for JVM housekeeping tasks (garbage collection, finalization) and a main thread for running the `main` method. The AWT (Abstract Window Toolkit) and Swing user interface frameworks create threads for managing user interface events. `Timer` creates threads for executing deferred tasks. Component frameworks, such as servlets and RMI create pools of threads and invoke component methods in these threads.

If you use these facilities—as many developers do—you have to be familiar with concurrency and thread safety, because these frameworks create threads and call your components from them. It would be nice to believe that concurrency is an "optional" or "advanced" language feature, but the reality is that nearly all Java applications are multithreaded and these frameworks do not insulate you from the need to properly coordinate access to application state.

When concurrency is introduced into an application by a framework, it is usually impossible to restrict the concurrency-awareness to the framework code, because frameworks by their nature make callbacks to application components that in turn access application state. Similarly, the need for thread safety does not end with the components called by the framework—it extends to all code paths that access the program state accessed by those components. Thus, the need for thread safety is contagious.

> Frameworks introduce concurrency into applications by calling application components from framework threads. Components invariably access application state, thus requiring that *all* code paths accessing that state be thread-safe.

The facilities described below all cause application code to be called from threads not managed by the application. While the need for thread safety may start with these facilities, it rarely ends there; instead, it ripples through the application.

Timer. `Timer` is a convenience mechanism for scheduling tasks to run at a later time, either once or periodically. The introduction of a `Timer` can complicate an otherwise sequential program, because `TimerTask`s are executed in a thread managed by the `Timer`, not the application. If a `TimerTask` accesses data that is also accessed by other application threads, then not only must the `TimerTask` do so in a thread-safe manner, but *so must any other classes that access that data*. Often the easiest way to achieve this is to ensure that objects accessed by the TimerTask are themselves thread-safe, thus encapsulating the thread safety within the shared objects.

Servlets and JavaServer Pages (JSPs). The servlets framework is designed to handle all the infrastructure of deploying a web application and dispatching requests from remote HTTP clients. A request arriving at the server is dispatched, perhaps through a chain of filters, to the appropriate servlet or JSP. Each servlet represents a component of application logic, and in high-volume web sites, multiple clients may require the services of the same servlet at once. The servlets specification requires that a servlet be prepared to be called simultaneously from multiple threads. In other words, servlets need to be thread-safe.

Even if you could guarantee that a servlet was only called from one thread at a time, you would still have to pay attention to thread safety when building a web application. Servlets often access state information shared with other servlets, such as application-scoped objects (those stored in the ServletContext) or session-scoped objects (those stored in the per-client HttpSession). When a servlet accesses objects shared across servlets or requests, it must coordinate access to these objects properly, since multiple requests could be accessing them simultaneously from separate threads. Servlets and JSPs, as well as servlet filters and objects stored in scoped containers like ServletContext and HttpSession, simply have to be thread-safe.

Remote Method Invocation. RMI lets you invoke methods on objects running in another JVM. When you call a remote method with RMI, the method arguments are packaged (marshaled) into a byte stream and shipped over the network to the remote JVM, where they are unpacked (unmarshaled) and passed to the remote method.

When the RMI code calls your remote object, in what thread does that call happen? You don't know, but it's definitely not in a thread you created—your object gets called in a thread managed by RMI. How many threads does RMI create? Could the same remote method on the same remote object be called simultaneously in multiple RMI threads?[4]

A remote object must guard against two thread safety hazards: properly coordinating access to state that may be shared with other objects, and properly coordinating access to the state of the remote object itself (since the same object may be called in multiple threads simultaneously). Like servlets, RMI objects should be prepared for multiple simultaneous calls and must provide their own thread safety.

Swing and AWT. GUI applications are inherently asynchronous. Users may select a menu item or press a button at any time, and they expect that the application will respond promptly even if it is in the middle of doing something else. Swing and AWT address this problem by creating a separate thread for handling user-initiated events and updating the graphical view presented to the user.

[4] Answer: yes, but it's not all that clear from the Javadoc—you have to read the RMI spec.

Swing components, such as `JTable`, are not thread-safe. Instead, Swing programs achieve their thread safety by confining all access to GUI components to the event thread. If an application wants to manipulate the GUI from outside the event thread, it must cause the code that will manipulate the GUI to run in the event thread instead.

When the user performs a UI action, an event handler is called in the event thread to perform whatever operation the user requested. If the handler needs to access application state that is also accessed from other threads (such as a document being edited), then the event handler, along with any other code that accesses that state, must do so in a thread-safe manner.

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 unnacceptably 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 serialized[8] 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

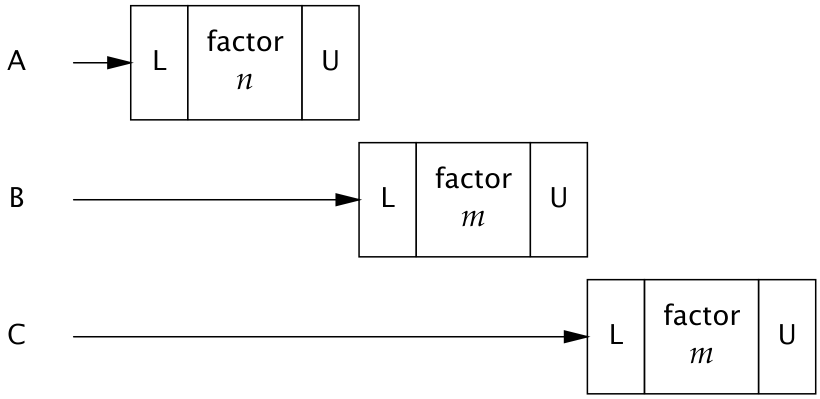


FIGURE 2.1. Poor concurrency of SynchronizedFactorizer.

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 shortest 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.

Chapter 3

Sharing Objects

We stated at the beginning of Chapter 2 that writing correct concurrent programs is primarily about managing access to shared, mutable state. That chapter was about using synchronization to prevent multiple threads from accessing the same data at the same time; this chapter examines techniques for sharing and publishing objects so they can be safely accessed by multiple threads. Together, they lay the foundation for building thread-safe classes and safely structuring concurrent applications using the `java.util.concurrent` library classes.

We have seen how `synchronized` blocks and methods can ensure that operations execute atomically, but it is a common misconception that `synchronized` is *only* about atomicity or demarcating "critical sections". Synchronization also has another significant, and subtle, aspect: *memory visibility*. We want not only to prevent one thread from modifying the state of an object when another is using it, but also to ensure that when a thread modifies the state of an object, other threads can actually *see* the changes that were made. But without synchronization, this may not happen. You can ensure that objects are published safely either by using explicit synchronization or by taking advantage of the synchronization built into library classes.

3.1 Visibility

Visibility is subtle because the things that can go wrong are so counterintuitive. In a single-threaded environment, if you write a value to a variable and later read that variable with no intervening writes, you can expect to get the same value back. This seems only natural. It may be hard to accept at first, but when the reads and writes occur in different threads, *this is simply not the case*. In general, there is *no* guarantee that the reading thread will see a value written by another thread on a timely basis, or even at all. In order to ensure visibility of memory writes across threads, you must use synchronization.

`NoVisibility` in Listing 3.1 illustrates what can go wrong when threads share data without synchronization. Two threads, the main thread and the reader thread, access the shared variables `ready` and `number`. The main thread starts the reader thread and then sets `number` to 42 and `ready` to `true`. The reader thread spins until it sees `ready` is `true`, and then prints out `number`. While it may seem obvious that `NoVisibility` will print 42, it is in fact possible that it will print zero, or never terminate at all! Because it does not use adequate synchronization, there is no guarantee that the values of `ready` and `number` written by the main thread will be visible to the reader thread.

```
public class NoVisibility {
    private static boolean ready;
    private static int number;

    private static class ReaderThread extends Thread {
        public void run() {
            while (!ready)
                Thread.yield();
            System.out.println(number);
        }
    }

    public static void main(String[] args) {
        new ReaderThread().start();
        number = 42;
        ready = true;
    }
}
```

Listing 3.1. Sharing variables without synchronization. *Don't do this.*

`NoVisibility` could loop forever because the value of `ready` might never become visible to the reader thread. Even more strangely, `NoVisibility` could print zero because the write to `ready` might be made visible to the reader thread *before* the write to `number`, a phenomenon known as *reordering*. There is no guarantee that operations in one thread will be performed in the order given by the program, as long as the reordering is not detectable from within *that* thread—*even if the reordering is apparent to other threads.*[1] When the main thread writes first to `number` and then to `ready` without synchronization, the reader thread could see those writes happen in the opposite order—or not at all.

[1] This may seem like a broken design, but it is meant to allow JVMs to take full advantage of the performance of modern multiprocessor hardware. For example, in the absence of synchronization, the Java Memory Model permits the compiler to reorder operations and cache values in registers, and permits CPUs to reorder operations and cache values in processor-specific caches. For more details, see Chapter 16.

> In the absence of synchronization, the compiler, processor, and runtime can do some downright weird things to the order in which operations appear to execute. Attempts to reason about the order in which memory actions "must" happen in insufficiently synchronized multithreaded programs will almost certainly be incorrect.

`NoVisibility` is about as simple as a concurrent program can get—two threads and two shared variables—and yet it is still all too easy to come to the wrong conclusions about what it does or even whether it will terminate. Reasoning about insufficiently synchronized concurrent programs is prohibitively difficult.

This may all sound a little scary, and it should. Fortunately, there's an easy way to avoid these complex issues: *always use the proper synchronization whenever data is shared across threads.*

3.1.1 Stale data

`NoVisibility` demonstrated one of the ways that insufficiently synchronized programs can cause surprising results: *stale data*. When the reader thread examines `ready`, it may see an out-of-date value. Unless synchronization is used *every time a variable is accessed*, it is possible to see a stale value for that variable. Worse, staleness is not all-or-nothing: a thread can see an up-to-date value of one variable but a stale value of another variable that was written first.

When food is stale, it is usually still edible—just less enjoyable. But stale data can be more dangerous. While an out-of-date hit counter in a web application might not be so bad,[2] stale values can cause serious safety or liveness failures. In `NoVisibility`, stale values could cause it to print the wrong value or prevent the program from terminating. Things can get even more complicated with stale values of object references, such as the link pointers in a linked list implementation. *Stale data can cause serious and confusing failures such as unexpected exceptions, corrupted data structures, inaccurate computations, and infinite loops.*

`MutableInteger` in Listing 3.2 is not thread-safe because the `value` field is accessed from both `get` and `set` without synchronization. Among other hazards, it is susceptible to stale values: if one thread calls `set`, other threads calling `get` may or may not see that update.

We can make `MutableInteger` thread safe by synchronizing the getter and setter as shown in `SynchronizedInteger` in Listing 3.3. Synchronizing only the setter would not be sufficient: threads calling `get` would still be able to see stale values.

[2] Reading data without synchronization is analogous to using the `READ_UNCOMMITTED` isolation level in a database, where you are willing to trade accuracy for performance. However, in the case of unsynchronized reads, you are trading away a greater degree of accuracy, since the visible value for a shared variable can be arbitrarily stale.

```
@NotThreadSafe
public class MutableInteger {
    private int value;

    public int  get() { return value; }
    public void set(int value) { this.value = value; }
}
```

Listing 3.2. Non-thread-safe mutable integer holder.

```
@ThreadSafe
public class SynchronizedInteger {
    @GuardedBy("this") private int value;

    public synchronized int get() { return value; }
    public synchronized void set(int value) { this.value = value; }
}
```

Listing 3.3. Thread-safe mutable integer holder.

3.1.2 Nonatomic 64-bit operations

When a thread reads a variable without synchronization, it may see a stale value, but at least it sees a value that was actually placed there by some thread rather than some random value. This safety guarantee is called *out-of-thin-air safety*.

Out-of-thin-air safety applies to all variables, with one exception: 64-bit numeric variables (`double` and `long`) that are not declared `volatile` (see Section 3.1.4). The Java Memory Model requires fetch and store operations to be atomic, but for nonvolatile `long` and `double` variables, the JVM is permitted to treat a 64-bit read or write as two separate 32-bit operations. If the reads and writes occur in different threads, it is therefore possible to read a nonvolatile `long` and get back the high 32 bits of one value and the low 32 bits of another.[3] Thus, even if you don't care about stale values, it is not safe to use shared mutable `long` and `double` variables in multithreaded programs unless they are declared `volatile` or guarded by a lock.

3.1.3 Locking and visibility

Intrinsic locking can be used to guarantee that one thread sees the effects of another in a predictable manner, as illustrated by Figure 3.1. When thread *A* executes a `synchronized` block, and subsequently thread *B* enters a `synchronized` block guarded by the same lock, the values of variables that were visible to *A* prior to releasing the lock are guaranteed to be visible to *B* upon acquiring the

[3] When the Java Virtual Machine Specification was written, many widely used processor architectures could not efficiently provide atomic 64-bit arithmetic operations.

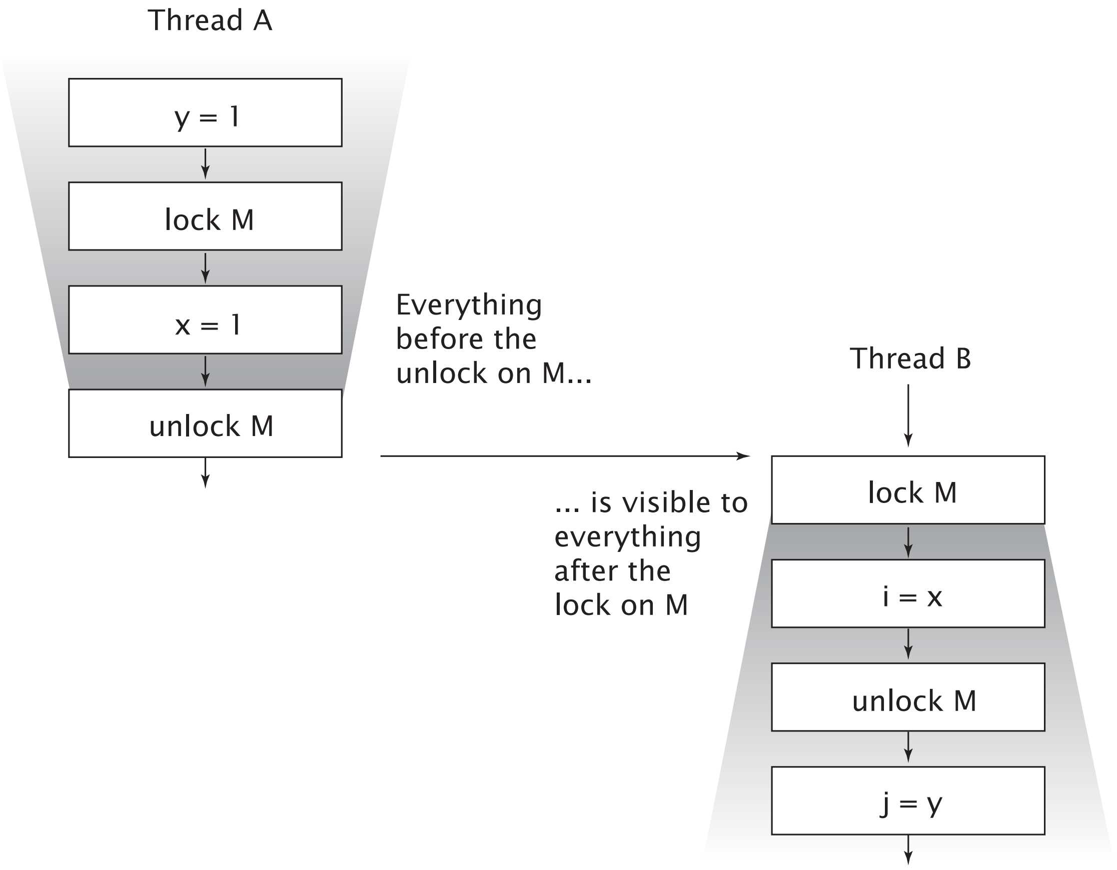


FIGURE 3.1. Visibility guarantees for synchronization.

lock. In other words, everything *A* did in or prior to a `synchronized` block is visible to *B* when it executes a `synchronized` block guarded by the same lock. *Without synchronization, there is no such guarantee.*

We can now give the other reason for the rule requiring all threads to synchronize on the *same* lock when accessing a shared mutable variable—to guarantee that values written by one thread are made visible to other threads. Otherwise, if a thread reads a variable without holding the appropriate lock, it might see a stale value.

> Locking is not just about mutual exclusion; it is also about memory visibility. To ensure that all threads see the most up-to-date values of shared mutable variables, the reading and writing threads must synchronize on a common lock.

3.1.4 Volatile variables

The Java language also provides an alternative, weaker form of synchronization, *volatile variables*, to ensure that updates to a variable are propagated predictably to other threads. When a field is declared volatile, the compiler and runtime are put on notice that this variable is shared and that operations on it should not be reordered with other memory operations. Volatile variables are not cached in registers or in caches where they are hidden from other processors, so a read of a volatile variable always returns the most recent write by any thread.

A good way to think about volatile variables is to imagine that they behave roughly like the SynchronizedInteger class in Listing 3.3, replacing reads and writes of the volatile variable with calls to get and set.[4] Yet accessing a volatile variable performs no locking and so cannot cause the executing thread to block, making volatile variables a lighter-weight synchronization mechanism than synchronized.[5]

The visibility effects of volatile variables extend beyond the value of the volatile variable itself. When thread *A* writes to a volatile variable and subsequently thread *B* reads that same variable, the values of *all* variables that were visible to *A* prior to writing to the volatile variable become visible to *B* after reading the volatile variable. So from a memory visibility perspective, writing a volatile variable is like exiting a synchronized block and reading a volatile variable is like entering a synchronized block. However, we do not recommend relying too heavily on volatile variables for visibility; code that relies on volatile variables for visibility of arbitrary state is more fragile and harder to understand than code that uses locking.

> Use volatile variables only when they simplify implementing and verifying your synchronization policy; avoid using volatile variables when veryfing correctness would require subtle reasoning about visibility. Good uses of volatile variables include ensuring the visibility of their own state, that of the object they refer to, or indicating that an important lifecycle event (such as initialization or shutdown) has occurred.

Listing 3.4 illustrates a typical use of volatile variables: checking a status flag to determine when to exit a loop. In this example, our anthropomorphized thread is trying to get to sleep by the time-honored method of counting sheep. For this example to work, the asleep flag must be volatile. Otherwise, the thread might not notice when asleep has been set by another thread.[6] We could instead have

[4] This analogy is not exact; the memory visibility effects of SynchronizedInteger are actually slightly stronger than those of volatile variables. See Chapter 16.

[5] Volatile reads are only slightly more expensive than nonvolatile reads on most current processor architectures.

[6] Debugging tip: For server applications, be sure to always specify the -server JVM command line switch when invoking the JVM, even for development and testing. The server JVM performs more optimization than the client JVM, such as hoisting variables out of a loop that are not modified in the loop; code that might appear to work in the development environment (client JVM) can break in the deployment environment (server JVM). For example, had we "forgotten" to declare the variable asleep as volatile in Listing 3.4, the server JVM could hoist the test out of the loop (turning it into an infinite loop), but the client JVM would not. An infinite loop that shows up in development is far less costly than one that only shows up in production.

used locking to ensure visibility of changes to asleep, but that would have made the code more cumbersome.

```
volatile boolean asleep;
...
    while (!asleep)
        countSomeSheep();
```

Listing 3.4. Counting sheep.

Volatile variables are convenient, but they have limitations. The most common use for volatile variables is as a completion, interruption, or status flag, such as the asleep flag in Listing 3.4. Volatile variables can be used for other kinds of state information, but more care is required when attempting this. For example, the semantics of volatile are not strong enough to make the increment operation (count++) atomic, unless you can guarantee that the variable is written only from a single thread. (Atomic variables do provide atomic read-modify-write support and can often be used as "better volatile variables"; see Chapter 15.)

> Locking can guarantee both visibility and atomicity; volatile variables can only guarantee visibility.

You can use volatile variables only when all the following criteria are met:

- Writes to the variable do not depend on its current value, or you can ensure that only a single thread ever updates the value;
- The variable does not participate in invariants with other state variables; and
- Locking is not required for any other reason while the variable is being accessed.

3.2 Publication and escape

Publishing an object means making it available to code outside of its current scope, such as by storing a reference to it where other code can find it, returning it from a nonprivate method, or passing it to a method in another class. In many situations, we want to ensure that objects and their internals are *not* published. In other situations, we do want to publish an object for general use, but doing so in a thread-safe manner may require synchronization. Publishing internal state variables can compromise encapsulation and make it more difficult to preserve invariants; publishing objects before they are fully constructed can compromise thread safety. An object that is published when it should not have been is said to have *escaped*. Section 3.5 covers idioms for safe publication; right now, we look at how an object can escape.

The most blatant form of publication is to store a reference in a public static field, where any class and thread could see it, as in Listing 3.5. The `initialize` method instantiates a new `HashSet` and publishes it by storing a reference to it into `knownSecrets`.

```
public static Set<Secret> knownSecrets;

public void initialize() {
    knownSecrets = new HashSet<Secret>();
}
```

LISTING 3.5. Publishing an object.

Publishing one object may indirectly publish others. If you add a `Secret` to the published `knownSecrets` set, you've also published that `Secret`, because any code can iterate the `Set` and obtain a reference to the new `Secret`. Similarly, returning a reference from a nonprivate method also publishes the returned object. `UnsafeStates` in Listing 3.6 publishes the supposedly private array of state abbreviations.

```
class UnsafeStates {
    private String[] states = new String[] {
        "AK", "AL" ...
    };
    public String[] getStates() { return states; }
}
```

LISTING 3.6. Allowing internal mutable state to escape. *Don't do this.*

Publishing `states` in this way is problematic because any caller can modify its contents. In this case, the `states` array has escaped its intended scope, because what was supposed to be private state has been effectively made public.

Publishing an object also publishes any objects referred to by its nonprivate fields. More generally, any object that is *reachable* from a published object by following some chain of nonprivate field references and method calls has also been published.

From the perspective of a class *C*, an *alien* method is one whose behavior is not fully specified by *C*. This includes methods in other classes as well as overrideable methods (neither `private` nor `final`) in *C* itself. Passing an object to an alien method must also be considered publishing that object. Since you can't know what code will actually be invoked, you don't know that the alien method won't publish the object or retain a reference to it that might later be used from another thread.

Whether another thread actually does something with a published reference doesn't really matter, because the risk of misuse is still present.[7] Once an ob-

[7] If someone steals your password and posts it on the `alt.free-passwords` newsgroup, that infor-

ject escapes, you have to assume that another class or thread may, maliciously or carelessly, misuse it. This is a compelling reason to use encapsulation: it makes it practical to analyze programs for correctness and harder to violate design constraints accidentally.

A final mechanism by which an object or its internal state can be published is to publish an inner class instance, as shown in `ThisEscape` in Listing 3.7. When `ThisEscape` publishes the `EventListener`, it implicitly publishes the enclosing `ThisEscape` instance as well, because inner class instances contain a hidden reference to the enclosing instance.

```
public class ThisEscape {
    public ThisEscape(EventSource source) {
        source.registerListener(
            new EventListener() {
                public void onEvent(Event e) {
                    doSomething(e);
                }
            });
    }
}
```

Listing 3.7. Implicitly allowing the `this` reference to escape. *Don't do this.*

3.2.1 Safe construction practices

`ThisEscape` illustrates an important special case of escape—when the `this` references escapes during construction. When the inner `EventListener` instance is published, so is the enclosing `ThisEscape` instance. But an object is in a predictable, consistent state only after its constructor returns, so publishing an object from within its constructor can publish an incompletely constructed object. This is true *even if the publication is the last statement in the constructor.* If the `this` reference escapes during construction, the object is considered *not properly constructed*.[8]

Do not allow the `this` reference to escape during construction.

A common mistake that can let the `this` reference escape during construction is to start a thread from a constructor. When an object creates a thread from its constructor, it almost always shares its `this` reference with the new thread, either explicitly (by passing it to the constructor) or implicitly (because the `Thread` or

mation has escaped: whether or not someone has (yet) used those credentials to create mischief, your account has still been compromised. Publishing a reference poses the same sort of risk.

[8] More specifically, the `this` reference should not escape from the *thread* until after the constructor returns. The `this` reference can be stored somewhere by the constructor so long as it is not *used* by another thread until after construction. `SafeListener` in Listing 3.8 uses this technique.

Runnable is an inner class of the owning object). The new thread might then be able to see the owning object before it is fully constructed. There's nothing wrong with *creating* a thread in a constructor, but it is best not to *start* the thread immediately. Instead, expose a start or initialize method that starts the owned thread. (See Chapter 7 for more on service lifecycle issues.) Calling an overrideable instance method (one that is neither private nor final) from the constructor can also allow the this reference to escape.

If you are tempted to register an event listener or start a thread from a constructor, you can avoid the improper construction by using a private constructor and a public factory method, as shown in SafeListener in Listing 3.8.

```
public class SafeListener {
    private final EventListener listener;

    private SafeListener() {
        listener = new EventListener() {
            public void onEvent(Event e) {
                doSomething(e);
            }
        };
    }

    public static SafeListener newInstance(EventSource source) {
        SafeListener safe = new SafeListener();
        source.registerListener(safe.listener);
        return safe;
    }
}
```

Listing 3.8. Using a factory method to prevent the this reference from escaping during construction.

3.3 Thread confinement

Accessing shared, mutable data requires using synchronization; one way to avoid this requirement is to *not share*. If data is only accessed from a single thread, no synchronization is needed. This technique, *thread confinement*, is one of the simplest ways to achieve thread safety. When an object is confined to a thread, such usage is automatically thread-safe even if the confined object itself is not [CPJ 2.3.2].

Swing uses thread confinement extensively. The Swing visual components and data model objects are not thread safe; instead, safety is achieved by confining them to the Swing event dispatch thread. To use Swing properly, code running in threads other than the event thread should not access these objects. (To make this easier, Swing provides the invokeLater mechanism to schedule a Runnable for execution in the event thread.) Many concurrency errors in Swing applications stem from improper use of these confined objects from another thread.

Another common application of thread confinement is the use of pooled JDBC (Java Database Connectivity) `Connection` objects. The JDBC specification does not require that `Connection` objects be thread-safe.[9] In typical server applications, a thread acquires a connection from the pool, uses it for processing a single request, and returns it. Since most requests, such as servlet requests or EJB (Enterprise JavaBeans) calls, are processed synchronously by a single thread, and the pool will not dispense the same connection to another thread until it has been returned, this pattern of connection management implicitly confines the `Connection` to that thread for the duration of the request.

Just as the language has no mechanism for enforcing that a variable is guarded by a lock, it has no means of confining an object to a thread. Thread confinement is an element of your program's design that must be enforced by its implementation. The language and core libraries provide mechanisms that can help in maintaining thread confinement—local variables and the `ThreadLocal` class—but even with these, it is still the programmer's responsibility to ensure that thread-confined objects do not escape from their intended thread.

3.3.1 Ad-hoc thread confinement

Ad-hoc thread confinement describes when the responsibility for maintaining thread confinement falls entirely on the implementation. Ad-hoc thread confinement can be fragile because none of the language features, such as visibility modifiers or local variables, helps confine the object to the target thread. In fact, references to thread-confined objects such as visual components or data models in GUI applications are often held in public fields.

The decision to use thread confinement is often a consequence of the decision to implement a particular subsystem, such as the GUI, as a single-threaded subsystem. Single-threaded subsystems can sometimes offer a simplicity benefit that outweighs the fragility of ad-hoc thread confinement.[10]

A special case of thread confinement applies to volatile variables. It is safe to perform read-modify-write operations on shared volatile variables as long as you ensure that the volatile variable is only written from a single thread. In this case, you are confining the *modification* to a single thread to prevent race conditions, and the visibility guarantees for volatile variables ensure that other threads see the most up-to-date value.

Because of its fragility, ad-hoc thread confinement should be used sparingly; if possible, use one of the stronger forms of thread confinment (stack confinement or `ThreadLocal`) instead.

[9] The connection *pool* implementations provided by application servers are thread-safe; connection pools are necessarily accessed from multiple threads, so a non-thread-safe implementation would not make sense.

[10] Another reason to make a subsystem single-threaded is deadlock avoidance; this is one of the primary reasons most GUI frameworks are single-threaded. Single-threaded subsystems are covered in Chapter 9.

3.3.2 Stack confinement

Stack confinement is a special case of thread confinement in which an object can only be reached through local variables. Just as encapsulation can make it easier to preserve invariants, local variables can make it easier to confine objects to a thread. Local variables are intrinsically confined to the executing thread; they exist on the executing thread's stack, which is not accessible to other threads. Stack confinement (also called *within-thread* or *thread-local* usage, but not to be confused with the `ThreadLocal` library class) is simpler to maintain and less fragile than ad-hoc thread confinement.

For primitively typed local variables, such as `numPairs` in `loadTheArk` in Listing 3.9, you cannot violate stack confinement even if you tried. There is no way to obtain a reference to a primitive variable, so the language semantics ensure that primitive local variables are always stack confined.

```
public int loadTheArk(Collection<Animal> candidates) {
    SortedSet<Animal> animals;
    int numPairs = 0;
    Animal candidate = null;

    // animals confined to method, don't let them escape!
    animals = new TreeSet<Animal>(new SpeciesGenderComparator());
    animals.addAll(candidates);
    for (Animal a : animals) {
        if (candidate == null || !candidate.isPotentialMate(a))
            candidate = a;
        else {
            ark.load(new AnimalPair(candidate, a));
            ++numPairs;
            candidate = null;
        }
    }
    return numPairs;
}
```

Listing 3.9. Thread confinement of local primitive and reference variables.

Maintaining stack confinement for object references requires a little more assistance from the programmer to ensure that the referent does not escape. In `loadTheArk`, we instantiate a `TreeSet` and store a reference to it in `animals`. At this point, there is exactly one reference to the `Set`, held in a local variable and therefore confined to the executing thread. However, if we were to publish a reference to the `Set` (or any of its internals), the confinement would be violated and the animals would escape.

Using a non-thread-safe object in a within-thread context is still thread-safe. However, be careful: the design requirement that the object be confined to the executing thread, or the awareness that the confined object is not thread-safe, often exists only in the head of the developer when the code is written. If the assumption of within-thread usage is not clearly documented, future maintainers might mistakenly allow the object to escape.

3.3.3 ThreadLocal

A more formal means of maintaining thread confinement is `ThreadLocal`, which allows you to associate a per-thread value with a value-holding object. `ThreadLocal` provides `get` and `set` accessor methods that maintain a separate copy of the value for each thread that uses it, so a `get` returns the most recent value passed to `set` *from the currently executing thread*.

Thread-local variables are often used to prevent sharing in designs based on mutable Singletons or global variables. For example, a single-threaded application might maintain a global database connection that is initialized at startup to avoid having to pass a `Connection` to every method. Since JDBC connections may not be thread-safe, a multithreaded application that uses a global connection without additional coordination is not thread-safe either. By using a `ThreadLocal` to store the JDBC connection, as in `ConnectionHolder` in Listing 3.10, each thread will have its own connection.

```
private static ThreadLocal<Connection> connectionHolder
    = new ThreadLocal<Connection>() {
        public Connection initialValue() {
            return DriverManager.getConnection(DB_URL);
        }
    };

public static Connection getConnection() {
    return connectionHolder.get();
}
```

Listing 3.10. Using `ThreadLocal` to ensure thread confinement.

This technique can also be used when a frequently used operation requires a temporary object such as a buffer and wants to avoid reallocating the temporary object on each invocation. For example, before Java 5.0, `Integer.toString` used a `ThreadLocal` to store the 12-byte buffer used for formatting its result, rather than using a shared static buffer (which would require locking) or allocating a new buffer for each invocation.[11]

When a thread calls `ThreadLocal.get` for the first time, `initialValue` is consulted to provide the initial value for that thread. Conceptually, you can think of a `ThreadLocal<T>` as holding a `Map<Thread,T>` that stores the thread-specific

[11] This technique is unlikely to be a performance win unless the operation is performed very frequently or the allocation is unusually expensive. In Java 5.0, it was replaced with the more straightforward approach of allocating a new buffer for every invocation, suggesting that for something as mundane as a temporary buffer, it is not a performance win.

values, though this is not how it is actually implemented. The thread-specific values are stored in the Thread object itself; when the thread terminates, the thread-specific values can be garbage collected.

If you are porting a single-threaded application to a multithreaded environment, you can preserve thread safety by converting shared global variables into ThreadLocals, if the semantics of the shared globals permits this; an application-wide cache would not be as useful if it were turned into a number of thread-local caches.

ThreadLocal is widely used in implementing application frameworks. For example, J2EE containers associate a transaction context with an executing thread for the duration of an EJB call. This is easily implemented using a static ThreadLocal holding the transaction context: when framework code needs to determine what transaction is currently running, it fetches the transaction context from this ThreadLocal. This is convenient in that it reduces the need to pass execution context information into every method, but couples any code that uses this mechanism to the framework.

It is easy to abuse ThreadLocal by treating its thread confinement property as a license to use global variables or as a means of creating "hidden" method arguments. Like global variables, thread-local variables can detract from reusability and introduce hidden couplings among classes, and should therefore be used with care.

3.4 Immutability

The other end-run around the need to synchronize is to use *immutable* objects [EJ Item 13]. Nearly all the atomicity and visibility hazards we've described so far, such as seeing stale values, losing updates, or observing an object to be in an inconsistent state, have to do with the vagaries of multiple threads trying to access the same mutable state at the same time. If an object's state cannot be modified, these risks and complexities simply go away.

An immutable object is one whose state cannot be changed after construction. Immutable objects are inherently thread-safe; their invariants are established by the constructor, and if their state cannot be changed, these invariants always hold.

Immutable objects are always thread-safe.

Immutable objects are *simple*. They can only be in one state, which is carefully controlled by the constructor. One of the most difficult elements of program design is reasoning about the possible states of complex objects. Reasoning about the state of immutable objects, on the other hand, is trivial.

Immutable objects are also *safer*. Passing a mutable object to untrusted code, or otherwise publishing it where untrusted code could find it, is dangerous—the untrusted code might modify its state, or, worse, retain a reference to it and modify its state later from another thread. On the other hand, immutable objects cannot be subverted in this manner by malicious or buggy code, so they are safe to share and publish freely without the need to make defensive copies [EJ Item 24].

Neither the Java Language Specification nor the Java Memory Model formally defines immutability, but immutability is *not* equivalent to simply declaring all fields of an object `final`. An object whose fields are all final may still be mutable, since final fields can hold references to mutable objects.

An object is *immutable* if:

- Its state cannot be modified after construction;
- All its fields are `final`;[12] and
- It is *properly constructed* (the `this` reference does not escape during construction).

Immutable objects can still use mutable objects internally to manage their state, as illustrated by `ThreeStooges` in Listing 3.11. While the `Set` that stores the names is mutable, the design of `ThreeStooges` makes it impossible to modify that `Set` after construction. The `stooges` reference is `final`, so all object state is reached through a `final` field. The last requirement, proper construction, is easily met since the constructor does nothing that would cause the `this` reference to become accessible to code other than the constructor and its caller.

```
@Immutable
public final class ThreeStooges {
    private final Set<String> stooges = new HashSet<String>();

    public ThreeStooges() {
        stooges.add("Moe");
        stooges.add("Larry");
        stooges.add("Curly");
    }

    public boolean isStooge(String name) {
        return stooges.contains(name);
    }
}
```

Listing 3.11. Immutable class built out of mutable underlying objects.

Because program state changes all the time, you might be tempted to think that immutable objects are of limited use, but this is not the case. There is a dif-

[12] It is technically possible to have an immutable object without all fields being final—`String` is such a class—but this relies on delicate reasoning about benign data races that requires a deep understanding of the Java Memory Model. (For the curious: `String` lazily computes the hash code the first time `hashCode` is called and caches it in a nonfinal field, but this works only because that field can take on only one nondefault value that is the same every time it is computed because it is derived deterministically from immutable state. Don’t try this at home.)

ference between an *object* being immutable and the *reference* to it being immutable. Program state stored in immutable objects can still be updated by "replacing" immutable objects with a new instance holding new state; the next section offers an example of this technique.[13]

3.4.1 Final fields

The `final` keyword, a more limited version of the `const` mechanism from C++, supports the construction of immutable objects. Final fields can't be modified (although the objects they refer to can be modified if they are mutable), but they also have special semantics under the Java Memory Model. It is the use of final fields that makes possible the guarantee of *initialization safety* (see Section 3.5.2) that lets immutable objects be freely accessed and shared without synchronization.

Even if an object is mutable, making some fields `final` can still simplify reasoning about its state, since limiting the mutability of an object restricts its set of possible states. An object that is "mostly immutable" but has one or two mutable state variables is still simpler than one that has many mutable variables. Declaring fields `final` also documents to maintainers that these fields are not expected to change.

Just as it is a good practice to make all fields `private` unless they need greater visibility [EJ Item 12], it is a good practice to make all fields `final` unless they need to be mutable.

3.4.2 Example: Using `volatile` to publish immutable objects

In `UnsafeCachingFactorizer` on page 24, we tried to use two `AtomicReferences` to store the last number and last factors, but this was not thread-safe because we could not fetch or update the two related values atomically. Using volatile variables for these values would not be thread-safe for the same reason. However, immutable objects can sometimes provide a weak form of atomicity.

The factoring servlet performs two operations that must be atomic: updating the cached result and conditionally fetching the cached factors if the cached number matches the requested number. Whenever a group of related data items must be acted on atomically, consider creating an immutable holder class for them, such as `OneValueCache`[14] in Listing 3.12.

Race conditions in accessing or updating multiple related variables can be eliminated by using an immutable object to hold all the variables. With a mutable

[13] Many developers fear that this approach will create performance problems, but these fears are usually unwarranted. Allocation is cheaper than you might think, and immutable objects offer additional performance advantages such as reduced need for locking or defensive copies and reduced impact on generational garbage collection.

[14] `OneValueCache` wouldn't be immutable without the `copyOf` calls in the constructor and getter. `Arrays.copyOf` was added as a convenience in Java 6; `clone` would also work.

```
@Immutable
class OneValueCache {
    private final BigInteger lastNumber;
    private final BigInteger[] lastFactors;

    public OneValueCache(BigInteger i,
                         BigInteger[] factors) {
        lastNumber = i;
        lastFactors = Arrays.copyOf(factors, factors.length);
    }

    public BigInteger[] getFactors(BigInteger i) {
        if (lastNumber == null || !lastNumber.equals(i))
            return null;
        else
            return Arrays.copyOf(lastFactors, lastFactors.length);
    }
}
```

Listing 3.12. Immutable holder for caching a number and its factors.

holder object, you would have to use locking to ensure atomicity; with an immutable one, once a thread acquires a reference to it, it need never worry about another thread modifying its state. If the variables are to be updated, a new holder object is created, but any threads working with the previous holder still see it in a consistent state.

VolatileCachedFactorizer in Listing 3.13 uses a OneValueCache to store the cached number and factors. When a thread sets the volatile cache field to reference a new OneValueCache, the new cached data becomes immediately visible to other threads.

The cache-related operations cannot interfere with each other because OneValueCache is immutable and the cache field is accessed only once in each of the relevant code paths. This combination of an immutable holder object for multiple state variables related by an invariant, and a volatile reference used to ensure its timely visibility, allows VolatileCachedFactorizer to be thread-safe even though it does no explicit locking.

3.5 Safe publication

So far we have focused on ensuring that an object *not* be published, such as when it is supposed to be confined to a thread or within another object. Of course, sometimes we *do* want to share objects across threads, and in this case we must do so safely. Unfortunately, simply storing a reference to an object into a public field, as in Listing 3.14, is *not* enough to publish that object safely.

```
@ThreadSafe
public class VolatileCachedFactorizer implements Servlet {
    private volatile OneValueCache cache =
        new OneValueCache(null, null);

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = cache.getFactors(i);
        if (factors == null) {
            factors = factor(i);
            cache = new OneValueCache(i, factors);
        }
        encodeIntoResponse(resp, factors);
    }
}
```

Listing 3.13. Caching the last result using a volatile reference to an immutable holder object.

```
// Unsafe publication
public Holder holder;

public void initialize() {
    holder = new Holder(42);
}
```

Listing 3.14. Publishing an object without adequate synchronization. *Don't do this.*

You may be surprised at how badly this harmless-looking example could fail. Because of visibility problems, the `Holder` could appear to another thread to be in an inconsistent state, even though its invariants were properly established by its constructor! This improper publication could allow another thread to observe a *partially constructed object*.

3.5.1 Improper publication: when good objects go bad

You cannot rely on the integrity of partially constructed objects. An observing thread could see the object in an inconsistent state, and then later see its state suddenly change, even though it has not been modified since publication. In fact, if the `Holder` in Listing 3.15 is published using the unsafe publication idiom in Listing 3.14, and a thread other than the publishing thread were to call `assertSanity`, it could throw `AssertionError`![15]

[15] The problem here is not the `Holder` class itself, but that the `Holder` is not properly published. However, `Holder` can be made immune to improper publication by declaring the `n` field to be `final`,

```
public class Holder {
    private int n;

    public Holder(int n) { this.n = n; }

    public void assertSanity() {
        if (n != n)
            throw new AssertionError("This statement is false.");
    }
}
```

Listing 3.15. Class at risk of failure if not properly published.

Because synchronization was not used to make the `Holder` visible to other threads, we say the `Holder` was *not properly published*. Two things can go wrong with improperly published objects. Other threads could see a stale value for the `holder` field, and thus see a `null` reference or other older value even though a value has been placed in `holder`. But far worse, other threads could see an up-to-date value for the `holder` reference, but stale values for the *state* of the `Holder`.[16] To make things even less predictable, a thread may see a stale value the first time it reads a field and then a more up-to-date value the next time, which is why `assertSanity` can throw `AssertionError`.

At the risk of repeating ourselves, some very strange things can happen when data is shared across threads without sufficient synchronization.

3.5.2 Immutable objects and initialization safety

Because immutable objects are so important, the Java Memory Model offers a special guarantee of *initialization safety* for sharing immutable objects. As we've seen, that an object reference becomes visible to another thread does not necessarily mean that the state of that object is visible to the consuming thread. In order to guarantee a consistent view of the object's state, synchronization is needed.

Immutable objects, on the other hand, can be safely accessed *even when synchronization is not used to publish the object reference*. For this guarantee of initialization safety to hold, all of the requirements for immutability must be met: unmodifiable state, all fields are `final`, and proper construction. (If `Holder` in Listing 3.15 were immutable, `assertSanity` could not throw `AssertionError`, even if the `Holder` was not properly published.)

which would make `Holder` immutable; see Section 3.5.2.

[16] While it may seem that field values set in a constructor are the first values written to those fields and therefore that there are no "older" values to see as stale values, the `Object` constructor first writes the default values to all fields before subclass constructors run. It is therefore possible to see the default value for a field as a stale value.

Immutable objects can be used safely by any thread without additional synchronization, even when synchronization is not used to publish them.

This guarantee extends to the values of all final fields of properly constructed objects; final fields can be safely accessed without additional synchronization. However, if final fields refer to mutable objects, synchronization is still required to access the state of the objects they refer to.

3.5.3 Safe publication idioms

Objects that are not immutable must be *safely published*, which usually entails synchronization by both the publishing and the consuming thread. For the moment, let's focus on ensuring that the consuming thread can see the object in its as-published state; we'll deal with visibility of modifications made after publication soon.

To publish an object safely, both the reference to the object and the object's state must be made visible to other threads at the same time. A properly constructed object can be safely published by:

- Initializing an object reference from a static initializer;
- Storing a reference to it into a `volatile` field or `AtomicReference`;
- Storing a reference to it into a `final` field of a properly constructed object; or
- Storing a reference to it into a field that is properly guarded by a lock.

The internal synchronization in thread-safe collections means that placing an object in a thread-safe collection, such as a `Vector` or `synchronizedList`, fulfills the last of these requirements. If thread *A* places object *X* in a thread-safe collection and thread *B* subsequently retrieves it, *B* is guaranteed to see the state of *X* as *A* left it, even though the application code that hands *X* off in this manner has no *explicit* synchronization. The thread-safe library collections offer the following safe publication guarantees, even if the Javadoc is less than clear on the subject:

- Placing a key or value in a `Hashtable`, `synchronizedMap`, or `ConcurrentMap` safely publishes it to any thread that retrieves it from the `Map` (whether directly or via an iterator);

- Placing an element in a `Vector`, `CopyOnWriteArrayList`, `CopyOnWriteArraySet`, `synchronizedList`, or `synchronizedSet` safely publishes it to any thread that retrieves it from the collection;

- Placing an element on a `BlockingQueue` or a `ConcurrentLinkedQueue` safely publishes it to any thread that retrieves it from the queue.

Other handoff mechanisms in the class library (such as `Future` and `Exchanger`) also constitute safe publication; we will identify these as providing safe publication as they are introduced.

Using a static initializer is often the easiest and safest way to publish objects that can be statically constructed:

```
public static Holder holder = new Holder(42);
```

Static initializers are executed by the JVM at class initialization time; because of internal synchronization in the JVM, this mechanism is guaranteed to safely publish any objects initialized in this way [JLS 12.4.2].

3.5.4 Effectively immutable objects

Safe publication is sufficient for other threads to safely access objects that are not going to be modified after publication without additional synchronization. The safe publication mechanisms all guarantee that the as-published state of an object is visible to all accessing threads as soon as the reference to it is visible, and if that state is not going to be changed again, this is sufficient to ensure that any access is safe.

Objects that are not technically immutable, but whose state will not be modified after publication, are called *effectively immutable*. They do not need to meet the strict definition of immutability in Section 3.4; they merely need to be treated by the program as if they were immutable after they are published. Using effectively immutable objects can simplify development and improve performance by reducing the need for synchronization.

Safely published *effectively immutable* objects can be used safely by any thread without additional synchronization.

For example, `Date` is mutable,[17] but if you use it as if it were immutable, you may be able to eliminate the locking that would otherwise be required when sharing a `Date` across threads. Suppose you want to maintain a `Map` storing the last login time of each user:

```
public Map<String, Date> lastLogin =
    Collections.synchronizedMap(new HashMap<String, Date>());
```

If the `Date` values are not modified after they are placed in the `Map`, then the synchronization in the `synchronizedMap` implementation is sufficient to publish the `Date` values safely, and no additional synchronization is needed when accessing them.

[17] This was probably a mistake in the class library design.

3.5.5 Mutable objects

If an object may be modified after construction, safe publication ensures only the visibility of the as-published state. Synchronization must be used not only to publish a mutable object, but also every time the object is accessed to ensure visibility of subsequent modifications. To share mutable objects safely, they must be safely published *and* be either thread-safe or guarded by a lock.

The publication requirements for an object depend on its mutability:

- *Immutable objects* can be published through any mechanism;
- *Effectively immutable objects* must be safely published;
- *Mutable objects* must be safely published, and must be either thread-safe or guarded by a lock.

3.5.6 Sharing objects safely

Whenever you acquire a reference to an object, you should know what you are allowed to do with it. Do you need to acquire a lock before using it? Are you allowed to modify its state, or only to read it? Many concurrency errors stem from failing to understand these "rules of engagement" for a shared object. When you publish an object, you should document how the object can be accessed.

The most useful policies for using and sharing objects in a concurrent program are:

Thread-confined. A thread-confined object is owned exclusively by and confined to one thread, and can be modified by its owning thread.

Shared read-only. A shared read-only object can be accessed concurrently by multiple threads without additional synchronization, but cannot be modified by any thread. Shared read-only objects include immutable and effectively immutable objects.

Shared thread-safe. A thread-safe object performs synchronization internally, so multiple threads can freely access it through its public interface without further synchronization.

Guarded. A guarded object can be accessed only with a specific lock held. Guarded objects include those that are encapsulated within other thread-safe objects and published objects that are known to be guarded by a specific lock.

Chapter 4

Composing Objects

So far, we've covered the low-level basics of thread safety and synchronization. But we don't want to have to analyze each memory access to ensure that our program is thread-safe; we want to be able to take thread-safe components and safely compose them into larger components or programs. This chapter covers patterns for structuring classes that can make it easier to make them thread-safe and to maintain them without accidentally undermining their safety guarantees.

4.1 Designing a thread-safe class

While it is possible to write a thread-safe program that stores all its state in public static fields, it is a lot harder to verify its thread safety or to modify it so that it remains thread-safe than one that uses encapsulation appropriately. Encapsulation makes it possible to determine that a class is thread-safe without having to examine the entire program.

The design process for a thread-safe class should include these three basic elements:

- Identify the variables that form the object's state;
- Identify the invariants that constrain the state variables;
- Establish a policy for managing concurrent access to the object's state.

An object's state starts with its fields. If they are all of primitive type, the fields comprise the entire state. `Counter` in Listing 4.1 has only one field, so the `value` field comprises its entire state. The state of an object with n primitive fields is just the n-tuple of its field values; the state of a 2D `Point` is its (x, y) value. If the object has fields that are references to other objects, its state will encompass fields from the referenced objects as well. For example, the state of a `LinkedList` includes the state of all the link node objects belonging to the list.

The *synchronization policy* defines how an object coordinates access to its state without violating its invariants or postconditions. It specifies what combination of

```
@ThreadSafe
public final class Counter {
    @GuardedBy("this") private long value = 0;

    public synchronized long getValue() {
        return value;
    }
    public synchronized long increment() {
        if (value == Long.MAX_VALUE)
            throw new IllegalStateException("counter overflow");
        return ++value;
    }
}
```

Listing 4.1. Simple thread-safe counter using the Java monitor pattern.

immutability, thread confinement, and locking is used to maintain thread safety, and which variables are guarded by which locks. To ensure that the class can be analyzed and maintained, document the synchronization policy.

4.1.1 Gathering synchronization requirements

Making a class thread-safe means ensuring that its invariants hold under concurrent access; this requires reasoning about its state. Objects and variables have a *state space*: the range of possible states they can take on. The smaller this state space, the easier it is to reason about. By using final fields wherever practical, you make it simpler to analyze the possible states an object can be in. (In the extreme case, immutable objects can only be in a single state.)

Many classes have invariants that identify certain states as *valid* or *invalid*. The `value` field in `Counter` is a `long`. The state space of a `long` ranges from `Long.MIN_VALUE` to `Long.MAX_VALUE`, but `Counter` places constraints on `value`; negative values are not allowed.

Similarly, operations may have postconditions that identify certain *state transitions* as invalid. If the current state of a `Counter` is 17, the *only* valid next state is 18. When the next state is derived from the current state, the operation is necessarily a compound action. Not all operations impose state transition constraints; when updating a variable that holds the current temperature, its previous state does not affect the computation.

Constraints placed on states or state transitions by invariants and postconditions create additional synchronization or encapsulation requirements. If certain states are invalid, then the underlying state variables must be encapsulated, otherwise client code could put the object into an invalid state. If an operation has invalid state transitions, it must be made atomic. On the other hand, if the class does not impose any such constraints, we may be able to relax encapsulation or serialization requirements to obtain greater flexibility or better performance.

A class can also have invariants that constrain multiple state variables. A number range class, like `NumberRange` in Listing 4.10, typically maintains state variables for the lower and upper bounds of the range. These variables must obey the constraint that the lower bound be less than or equal to the upper bound. Multivariable invariants like this one create atomicity requirements: related variables must be fetched or updated in a single atomic operation. You cannot update one, release and reacquire the lock, and then update the others, since this could involve leaving the object in an invalid state when the lock was released. When multiple variables participate in an invariant, the lock that guards them must be held for the duration of any operation that accesses the related variables.

> You cannot ensure thread safety without understanding an object's invariants and postconditions. Constraints on the valid values or state transitions for state variables can create atomicity and encapsulation requirements.

4.1.2 State-dependent operations

Class invariants and method postconditions constrain the valid states and state transitions for an object. Some objects also have methods with state-based *preconditions*. For example, you cannot remove an item from an empty queue; a queue must be in the "nonempty" state before you can remove an element. Operations with state-based preconditions are called *state-dependent* [CPJ 3].

In a single-threaded program, if a precondition does not hold, the operation has no choice but to fail. But in a concurrent program, the precondition may become true later due to the action of another thread. Concurrent programs add the possibility of waiting until the precondition becomes true, and then proceeding with the operation.

The built-in mechanisms for efficiently waiting for a condition to become true—`wait` and `notify`—are tightly bound to intrinsic locking, and can be difficult to use correctly. To create operations that wait for a precondition to become true before proceeding, it is often easier to use existing library classes, such as blocking queues or semaphores, to provide the desired state-dependent behavior. Blocking library classes such as `BlockingQueue`, `Semaphore`, and other *synchronizers* are covered in Chapter 5; creating state-dependent classes using the low-level mechanisms provided by the platform and class library is covered in Chapter 14.

4.1.3 State ownership

We implied in Section 4.1 that an object's state could be a subset of the fields in the object graph rooted at that object. Why might it be a subset? Under what conditions are fields reachable from a given object *not* part of that object's state?

When defining which variables form an object's state, we want to consider only the data that object *owns*. Ownership is not embodied explicitly in the language, but is instead an element of class design. If you allocate and populate a HashMap, you are creating multiple objects: the HashMap object, a number of Map.Entry objects used by the implementation of HashMap, and perhaps other internal objects as well. The logical state of a HashMap includes the state of all its Map.Entry and internal objects, even though they are implemented as separate objects.

For better or worse, garbage collection lets us avoid thinking carefully about ownership. When passing an object to a method in C++, you have to think fairly carefully about whether you are transferring ownership, engaging in a short-term loan, or envisioning long-term joint ownership. In Java, all these same ownership models are possible, but the garbage collector reduces the cost of many of the common errors in reference sharing, enabling less-than-precise thinking about ownership.

In many cases, ownership and encapsulation go together—the object encapsulates the state it owns and owns the state it encapsulates. It is the owner of a given state variable that gets to decide on the locking protocol used to maintain the integrity of that variable's state. Ownership implies control, but once you publish a reference to a mutable object, you no longer have exclusive control; at best, you might have "shared ownership". A class usually does not own the objects passed to its methods or constructors, unless the method is designed to explicitly transfer ownership of objects passed in (such as the synchronized collection wrapper factory methods).

Collection classes often exhibit a form of "split ownership", in which the collection owns the state of the collection infrastructure, but client code owns the objects stored in the collection. An example is ServletContext from the servlet framework. ServletContext provides a Map-like object container service to servlets where they can register and retrieve application objects by name with setAttribute and getAttribute. The ServletContext object implemented by the servlet container must be thread-safe, because it will necessarily be accessed by multiple threads. Servlets need not use synchronization when calling setAttribute and getAttribute, but they may have to use synchronization when *using* the objects stored in the ServletContext. These objects are owned by the application; they are being stored for safekeeping by the servlet container on the application's behalf. Like all shared objects, they must be shared safely; in order to prevent interference from multiple threads accessing the same object concurrently, they should either be thread-safe, effectively immutable, or explicitly guarded by a lock.[1]

4.2 Instance confinement

If an object is not thread-safe, several techniques can still let it be used safely in a multithreaded program. You can ensure that it is only accessed from a single thread (thread confinement), or that all access to it is properly guarded by a lock.

[1] Interestingly, the HttpSession object, which performs a similar function in the servlet framework, may have stricter requirements. Because the servlet container may access the objects in the HttpSession so they can be serialized for replication or passivation, they must be thread-safe because the container will be accessing them as well as the web application. (We say "may have" since replication and passivation is outside of the servlet specification but is a common feature of servlet containers.)

Encapsulation simplifies making classes thread-safe by promoting *instance confinement*, often just called *confinement* [CPJ 2.3.3]. When an object is encapsulated within another object, all code paths that have access to the encapsulated object are known and can be therefore be analyzed more easily than if that object were accessible to the entire program. Combining confinement with an appropriate locking discipline can ensure that otherwise non-thread-safe objects are used in a thread-safe manner.

> Encapsulating data within an object confines access to the data to the object's methods, making it easier to ensure that the data is always accessed with the appropriate lock held.

Confined objects must not escape their intended scope. An object may be confined to a class instance (such as a private class member), a lexical scope (such as a local variable), or a thread (such as an object that is passed from method to method within a thread, but not supposed to be shared across threads). Objects don't escape on their own, of course—they need help from the developer, who assists by publishing the object beyond its intended scope.

`PersonSet` in Listing 4.2 illustrates how confinement and locking can work together to make a class thread-safe even when its component state variables are not. The state of `PersonSet` is managed by a `HashSet`, which is not thread-safe. But because `mySet` is private and not allowed to escape, the `HashSet` is confined to the `PersonSet`. The only code paths that can access `mySet` are `addPerson` and `containsPerson`, and each of these acquires the lock on the `PersonSet`. All its state is guarded by its intrinsic lock, making `PersonSet` thread-safe.

```java
@ThreadSafe
public class PersonSet {
    @GuardedBy("this")
    private final Set<Person> mySet = new HashSet<Person>();

    public synchronized void addPerson(Person p) {
        mySet.add(p);
    }

    public synchronized boolean containsPerson(Person p) {
        return mySet.contains(p);
    }
}
```

LISTING 4.2. Using confinement to ensure thread safety.

This example makes no assumptions about the thread-safety of `Person`, but if it is mutable, additional synchronization will be needed when accessing a `Person` retrieved from a `PersonSet`. The most reliable way to do this would be to make

Person thread-safe; less reliable would be to guard the Person objects with a lock and ensure that all clients follow the protocol of acquiring the appropriate lock before accessing the Person.

Instance confinement is one of the easiest ways to build thread-safe classes. It also allows flexibility in the choice of locking strategy; PersonSet happened to use its own intrinsic lock to guard its state, but any lock, consistently used, would do just as well. Instance confinement also allows different state variables to be guarded by different locks. (For an example of a class that uses multiple lock objects to guard its state, see ServerStatus on page 236.)

There are many examples of confinement in the platform class libraries, including some classes that exist solely to turn non-thread-safe classes into thread-safe ones. The basic collection classes such as ArrayList and HashMap are not thread-safe, but the class library provides wrapper factory methods (Collections.synchronizedList and friends) so they can be used safely in multithreaded environments. These factories use the Decorator pattern (Gamma et al., 1995) to wrap the collection with a synchronized wrapper object; the wrapper implements each method of the appropriate interface as a synchronized method that forwards the request to the underlying collection object. So long as the wrapper object holds the only reachable reference to the underlying collection (i.e., the underlying collection is confined to the wrapper), the wrapper object is then thread-safe. The Javadoc for these methods warns that all access to the underlying collection must be made through the wrapper.

Of course, it is still possible to violate confinement by publishing a supposedly confined object; if an object is intended to be confined to a specific scope, then letting it escape from that scope is a bug. Confined objects can also escape by publishing other objects such as iterators or inner class instances that may indirectly publish the confined objects.

Confinement makes it easier to build thread-safe classes because a class that confines its state can be analyzed for thread safety without having to examine the whole program.

4.2.1 The Java monitor pattern

Following the principle of instance confinement to its logical conclusion leads you to the *Java monitor pattern*.[2] An object following the Java monitor pattern encapsulates all its mutable state and guards it with the object's own intrinsic lock.

Counter in Listing 4.1 shows a typical example of this pattern. It encapsulates one state variable, value, and all access to that state variable is through the methods of Counter, which are all synchronized.

[2] The Java monitor pattern is inspired by Hoare's work on *monitors* (Hoare, 1974), though there are significant differences between this pattern and a true monitor. The bytecode instructions for entering and exiting a synchronized block are even called monitorenter and monitorexit, and Java's built-in (intrinsic) locks are sometimes called *monitor locks* or *monitors*.

The Java monitor pattern is used by many library classes, such as Vector and Hashtable. Sometimes a more sophisticated synchronization policy is needed; Chapter 11 shows how to improve scalability through finer-grained locking strategies. The primary advantage of the Java monitor pattern is its simplicity.

The Java monitor pattern is merely a convention; any lock object could be used to guard an object's state so long as it is used consistently. Listing 4.3 illustrates a class that uses a private lock to guard its state.

```
public class PrivateLock {
    private final Object myLock = new Object();
    @GuardedBy("myLock") Widget widget;

    void someMethod() {
        synchronized(myLock) {
            // Access or modify the state of widget
        }
    }
}
```

Listing 4.3. Guarding state with a private lock.

There are advantages to using a private lock object instead of an object's intrinsic lock (or any other publicly accessible lock). Making the lock object private encapsulates the lock so that client code cannot acquire it, whereas a publicly accessible lock allows client code to participate in its synchronization policy—correctly or incorrectly. Clients that improperly acquire another object's lock could cause liveness problems, and verifying that a publicly accessible lock is properly used requires examining the entire program rather than a single class.

4.2.2 Example: tracking fleet vehicles

Counter in Listing 4.1 is a concise, but trivial, example of the Java monitor pattern. Let's build a slightly less trivial example: a "vehicle tracker" for dispatching fleet vehicles such as taxicabs, police cars, or delivery trucks. We'll build it first using the monitor pattern, and then see how to relax some of the encapsulation requirements while retaining thread safety.

Each vehicle is identified by a String and has a location represented by (x, y) coordinates. The VehicleTracker classes encapsulate the identity and locations of the known vehicles, making them well-suited as a data model in a model-view-controller GUI application where it might be shared by a view thread and multiple updater threads. The view thread would fetch the names and locations of the vehicles and render them on a display:

```
Map<String, Point> locations = vehicles.getLocations();
for (String key : locations.keySet())
    renderVehicle(key, locations.get(key));
```

Similarly, the updater threads would modify vehicle locations with data received from GPS devices or entered manually by a dispatcher through a GUI interface:

```
void vehicleMoved(VehicleMovedEvent evt) {
    Point loc = evt.getNewLocation();
    vehicles.setLocation(evt.getVehicleId(), loc.x, loc.y);
}
```

Since the view thread and the updater threads will access the data model concurrently, it must be thread-safe. Listing 4.4 shows an implementation of the vehicle tracker using the Java monitor pattern that uses `MutablePoint` in Listing 4.5 for representing the vehicle locations.

Even though `MutablePoint` is not thread-safe, the tracker class is. Neither the map nor any of the mutable points it contains is ever published. When we need to return vehicle locations to callers, the appropriate values are copied using either the `MutablePoint` copy constructor or `deepCopy`, which creates a new `Map` whose values are copies of the keys and values from the old `Map`.[3]

This implementation maintains thread safety in part by copying mutable data before returning it to the client. This is usually not a performance issue, but could become one *if* the set of vehicles is very large.[4] Another consequence of copying the data on each call to `getLocation` is that the contents of the returned collection do not change even if the underlying locations change. Whether this is good or bad depends on your requirements. It could be a benefit if there are internal consistency requirements on the location set, in which case returning a consistent snapshot is critical, or a drawback if callers require up-to-date information for each vehicle and therefore need to refresh their snapshot more often.

4.3 Delegating thread safety

All but the most trivial objects are composite objects. The Java monitor pattern is useful when building classes from scratch or composing classes out of objects that are not thread-safe. But what if the components of our class are already thread-safe? Do we need to add an additional layer of thread safety? The answer is ... "it depends". In some cases a composite made of thread-safe components is thread-safe (Listings 4.7 and 4.9), and in others it is merely a good start (Listing 4.10).

In `CountingFactorizer` on page 23, we added an `AtomicLong` to an otherwise stateless object, and the resulting composite object was still thread-safe. Since the state of `CountingFactorizer` *is* the state of the thread-safe `AtomicLong`, and since `CountingFactorizer` imposes no additional validity constraints on the state of the

[3] Note that `deepCopy` can't just wrap the `Map` with an `unmodifiableMap`, because that protects only the *collection* from modification; it does not prevent callers from modifying the mutable objects stored in it. For the same reason, populating the `HashMap` in `deepCopy` via a copy constructor wouldn't work either, because only the *references* to the points would be copied, not the point objects themselves.

[4] Because `deepCopy` is called from a `synchronized` method, the tracker's intrinsic lock is held for the duration of what might be a long-running copy operation, and this could degrade the responsiveness of the user interface when many vehicles are being tracked.

```
@ThreadSafe
public class MonitorVehicleTracker {
    @GuardedBy("this")
    private final Map<String, MutablePoint> locations;

    public MonitorVehicleTracker(
            Map<String, MutablePoint> locations) {
        this.locations = deepCopy(locations);
    }

    public synchronized Map<String, MutablePoint> getLocations() {
        return deepCopy(locations);
    }

    public synchronized MutablePoint getLocation(String id) {
        MutablePoint loc = locations.get(id);
        return loc == null ? null : new MutablePoint(loc);
    }

    public synchronized void setLocation(String id, int x, int y) {
        MutablePoint loc = locations.get(id);
        if (loc == null)
            throw new IllegalArgumentException("No such ID: " + id);
        loc.x = x;
        loc.y = y;
    }

    private static Map<String, MutablePoint> deepCopy(
            Map<String, MutablePoint> m) {
        Map<String, MutablePoint> result =
                new HashMap<String, MutablePoint>();
        for (String id : m.keySet())
            result.put(id, new MutablePoint(m.get(id)));
        return Collections.unmodifiableMap(result);
    }
}

public class MutablePoint { /* Listing 4.5 */ }
```

Listing 4.4. Monitor-based vehicle tracker implementation.

```
@NotThreadSafe
public class MutablePoint {
    public int x, y;

    public MutablePoint() { x = 0; y = 0; }
    public MutablePoint(MutablePoint p) {
        this.x = p.x;
        this.y = p.y;
    }
}
```

Listing 4.5. Mutable point class similar to `java.awt.Point`.

counter, it is easy to see that `CountingFactorizer` is thread-safe. We could say that `CountingFactorizer` *delegates* its thread safety responsibilities to the `AtomicLong`: `CountingFactorizer` is thread-safe because `AtomicLong` is.[5]

4.3.1 Example: vehicle tracker using delegation

As a more substantial example of delegation, let's construct a version of the vehicle tracker that delegates to a thread-safe class. We store the locations in a `Map`, so we start with a thread-safe `Map` implementation, `ConcurrentHashMap`. We also store the location using an immutable `Point` class instead of `MutablePoint`, shown in Listing 4.6.

```
@Immutable
public class Point {
    public final int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Listing 4.6. Immutable `Point` class used by `DelegatingVehicleTracker`.

`Point` is thread-safe because it is immutable. Immutable values can be freely shared and published, so we no longer need to copy the locations when returning them.

[5] If `count` were not `final`, the thread safety analysis of `CountingFactorizer` would be more complicated. If `CountingFactorizer` could modify `count` to reference a different `AtomicLong`, we would then have to ensure that this update was visible to all threads that might access the count, and that there were no race conditions regarding the value of the `count` reference. This is another good reason to use `final` fields wherever practical.

DelegatingVehicleTracker in Listing 4.7 does not use any explicit synchronization; all access to state is managed by ConcurrentHashMap, and all the keys and values of the Map are immutable.

```
@ThreadSafe
public class DelegatingVehicleTracker {
    private final ConcurrentMap<String, Point> locations;
    private final Map<String, Point> unmodifiableMap;

    public DelegatingVehicleTracker(Map<String, Point> points) {
        locations = new ConcurrentHashMap<String, Point>(points);
        unmodifiableMap = Collections.unmodifiableMap(locations);
    }

    public Map<String, Point> getLocations() {
        return unmodifiableMap;
    }

    public Point getLocation(String id) {
        return locations.get(id);
    }

    public void setLocation(String id, int x, int y) {
        if (locations.replace(id, new Point(x, y)) == null)
            throw new IllegalArgumentException(
                "invalid vehicle name: " + id);
    }
}
```

Listing 4.7. Delegating thread safety to a ConcurrentHashMap.

If we had used the original MutablePoint class instead of Point, we would be breaking encapsulation by letting getLocations publish a reference to mutable state that is not thread-safe. Notice that we've changed the behavior of the vehicle tracker class slightly; while the monitor version returned a snapshot of the locations, the delegating version returns an unmodifiable but "live" view of the vehicle locations. This means that if thread *A* calls getLocations and thread *B* later modifies the location of some of the points, those changes are reflected in the Map returned to thread *A*. As we remarked earlier, this can be a benefit (more up-to-date data) or a liability (potentially inconsistent view of the fleet), depending on your requirements.

If an unchanging view of the fleet is required, getLocations could instead return a shallow copy of the locations map. Since the contents of the Map are immutable, only the structure of the Map, not the contents, must be copied, as shown in Listing 4.8 (which returns a plain HashMap, since getLocations did not promise to return a thread-safe Map).

```
public Map<String, Point> getLocations() {
    return Collections.unmodifiableMap(
            new HashMap<String, Point>(locations));
}
```

Listing 4.8. Returning a static copy of the location set instead of a "live" one.

4.3.2 Independent state variables

The delegation examples so far delegate to a single, thread-safe state variable. We can also delegate thread safety to more than one underlying state variable as long as those underlying state variables are *independent*, meaning that the composite class does not impose any invariants involving the multiple state variables.

`VisualComponent` in Listing 4.9 is a graphical component that allows clients to register listeners for mouse and keystroke events. It maintains a list of registered listeners of each type, so that when an event occurs the appropriate listeners can be invoked. But there is no relationship between the set of mouse listeners and key listeners; the two are independent, and therefore `VisualComponent` can delegate its thread safety obligations to two underlying thread-safe lists.

```
public class VisualComponent {
    private final List<KeyListener> keyListeners
        = new CopyOnWriteArrayList<KeyListener>();
    private final List<MouseListener> mouseListeners
        = new CopyOnWriteArrayList<MouseListener>();

    public void addKeyListener(KeyListener listener) {
        keyListeners.add(listener);
    }

    public void addMouseListener(MouseListener listener) {
        mouseListeners.add(listener);
    }

    public void removeKeyListener(KeyListener listener) {
        keyListeners.remove(listener);
    }

    public void removeMouseListener(MouseListener listener) {
        mouseListeners.remove(listener);
    }
}
```

Listing 4.9. Delegating thread safety to multiple underlying state variables.

`VisualComponent` uses a `CopyOnWriteArrayList` to store each listener list; this is a thread-safe `List` implementation particularly suited for managing listener lists (see Section 5.2.3). Each `List` is thread-safe, and because there are no constraints coupling the state of one to the state of the other, `VisualComponent` can delegate its thread safety responsibilities to the underlying `mouseListeners` and `keyListeners` objects.

4.3.3 When delegation fails

Most composite classes are not as simple as `VisualComponent`: they have invariants that relate their component state variables. `NumberRange` in Listing 4.10 uses two `AtomicIntegers` to manage its state, but imposes an additional constraint—that the first number be less than or equal to the second.

```
public class NumberRange {
    // INVARIANT: lower <= upper
    private final AtomicInteger lower = new AtomicInteger(0);
    private final AtomicInteger upper = new AtomicInteger(0);

    public void setLower(int i) {
        // Warning -- unsafe check-then-act
        if (i > upper.get())
            throw new IllegalArgumentException(
                    "can’t set lower to " + i + " > upper");
        lower.set(i);
    }

    public void setUpper(int i) {
        // Warning -- unsafe check-then-act
        if (i < lower.get())
            throw new IllegalArgumentException(
                    "can’t set upper to " + i + " < lower");
        upper.set(i);
    }

    public boolean isInRange(int i) {
        return (i >= lower.get() && i <= upper.get());
    }
}
```

Listing 4.10. Number range class that does not sufficiently protect its invariants. *Don’t do this.*

`NumberRange` is not thread-safe; it does not preserve the invariant that constrains `lower` and `upper`. The `setLower` and `setUpper` methods *attempt* to respect this invariant, but do so poorly. Both `setLower` and `setUpper` are check-then-act sequences, but they do not use sufficient locking to make them atomic. If the number range holds $(0, 10)$, and one thread calls `setLower(5)` while another thread calls `setUpper(4)`, with some unlucky timing both will pass the checks in the setters and both modifications will be applied. The result is that the range now holds (5, 4)—an invalid state. So while the underlying `AtomicInteger`s are thread-safe, the composite class is not. Because the underlying state variables `lower` and `upper` are not independent, `NumberRange` cannot simply delegate thread safety to its thread-safe state variables.

`NumberRange` could be made thread-safe by using locking to maintain its invariants, such as guarding `lower` and `upper` with a common lock. It must also avoid publishing `lower` and `upper` to prevent clients from subverting its invariants.

If a class has compound actions, as `NumberRange` does, delegation alone is again not a suitable approach for thread safety. In these cases, the class must provide its own locking to ensure that compound actions are atomic, unless the entire compound action can also be delegated to the underlying state variables.

> If a class is composed of multiple *independent* thread-safe state variables and has no operations that have any invalid state transitions, then it can delegate thread safety to the underlying state variables.

The problem that prevented `NumberRange` from being thread-safe even though its state components were thread-safe is very similar to one of the rules about volatile variables described in Section 3.1.4: a variable is suitable for being declared `volatile` only if it does not participate in invariants involving other state variables.

4.3.4 Publishing underlying state variables

When you delegate thread safety to an object's underlying state variables, under what conditions can you publish those variables so that other classes can modify them as well? Again, the answer depends on what invariants your class imposes on those variables. While the underlying `value` field in `Counter` could take on any integer value, `Counter` constrains it to take on only positive values, and the increment operation constrains the set of valid next states given any current state. If you were to make the `value` field public, clients could change it to an invalid value, so publishing it would render the class incorrect. On the other hand, if a variable represents the current temperature or the ID of the last user to log on, then having another class modify this value at any time probably would not violate any invariants, so publishing this variable might be acceptable. (It still may not be a good idea, since publishing mutable variables constrains future development and opportunities for subclassing, but it would not *necessarily* render the class not thread-safe.)

> If a state variable is thread-safe, does not participate in any invariants that constrain its value, and has no prohibited state transitions for any of its operations, then it can safely be published.

For example, it would be safe to publish `mouseListeners` or `keyListeners` in `VisualComponent`. Because `VisualComponent` does not impose any constraints on the valid states of its listener lists, these fields could be made public or otherwise published without compromising thread safety.

4.3.5 Example: vehicle tracker that publishes its state

Let's construct another version of the vehicle tracker that publishes its underlying mutable state. Again, we need to modify the interface a little bit to accommodate this change, this time using mutable but thread-safe points.

```
@ThreadSafe
public class SafePoint {
    @GuardedBy("this") private int x, y;

    private SafePoint(int[] a) { this(a[0], a[1]); }

    public SafePoint(SafePoint p) { this(p.get()); }

    public SafePoint(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public synchronized int[] get() {
        return new int[] { x, y };
    }

    public synchronized void set(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Listing 4.11. Thread-safe mutable point class.

`SafePoint` in Listing 4.11 provides a getter that retrieves both the x and y values at once by returning a two-element array.[6] If we provided separate getters

[6] The private constructor exists to avoid the race condition that would occur if the copy constructor were implemented as `this(p.x, p.y)`; this is an example of the *private constructor capture idiom* (Bloch and Gafter, 2005).

for x and y, then the values could change between the time one coordinate is retrieved and the other, resulting in a caller seeing an inconsistent value: an (x, y) location where the vehicle never was. Using `SafePoint`, we can construct a vehicle tracker that publishes the underlying mutable state without undermining thread safety, as shown in the `PublishingVehicleTracker` class in Listing 4.12.

```
@ThreadSafe
public class PublishingVehicleTracker {
    private final Map<String, SafePoint> locations;
    private final Map<String, SafePoint> unmodifiableMap;

    public PublishingVehicleTracker(
                            Map<String, SafePoint> locations) {
        this.locations
            = new ConcurrentHashMap<String, SafePoint>(locations);
        this.unmodifiableMap
            = Collections.unmodifiableMap(this.locations);
    }

    public Map<String, SafePoint> getLocations() {
        return unmodifiableMap;
    }

    public SafePoint getLocation(String id) {
        return locations.get(id);
    }

    public void setLocation(String id, int x, int y) {
        if (!locations.containsKey(id))
            throw new IllegalArgumentException(
                "invalid vehicle name: " + id);
        locations.get(id).set(x, y);
      }
}
```

Listing 4.12. Vehicle tracker that safely publishes underlying state.

`PublishingVehicleTracker` derives its thread safety from delegation to an underlying `ConcurrentHashMap`, but this time the contents of the `Map` are thread-safe mutable points rather than immutable ones. The `getLocation` method returns an unmodifiable copy of the underlying `Map`. Callers cannot add or remove vehicles, but could change the location of one of the vehicles by mutating the `SafePoint` values in the returned `Map`. Again, the "live" nature of the `Map` may be a benefit or a drawback, depending on the requirements. `PublishingVehicleTracker` is thread-safe, but would not be so if it imposed any additional constraints on the valid values for vehicle locations. If it needed to be able to "veto" changes to vehicle locations or to take action when a location changes, the approach taken by `PublishingVehicleTracker` would not be appropriate.

4.4 Adding functionality to existing thread-safe classes

The Java class library contains many useful "building block" classes. Reusing existing classes is often preferable to creating new ones: reuse can reduce development effort, development risk (because the existing components are already tested), and maintenance cost. Sometimes a thread-safe class that supports all of the operations we want already exists, but often the best we can find is a class that supports *almost* all the operations we want, and then we need to add a new operation to it without undermining its thread safety.

As an example, let's say we need a thread-safe `List` with an atomic put-if-absent operation. The synchronized `List` implementations nearly do the job, since they provide the `contains` and `add` methods from which we can construct a put-if-absent operation.

The concept of put-if-absent is straightforward enough—check to see if an element is in the collection before adding it, and do not add it if it is already there. (Your "check-then-act" warning bells should be going off now.) The requirement that the class be thread-safe implicitly adds another requirement—that operations like put-if-absent be *atomic*. Any reasonable interpretation suggests that, if you take a `List` that does not contain object *X*, and add *X* twice with put-if-absent, the resulting collection contains only one copy of *X*. But, if put-if-absent were not atomic, with some unlucky timing two threads could both see that *X* was not present and both add *X*, resulting in two copies of *X*.

The safest way to add a new atomic operation is to modify the original class to support the desired operation, but this is not always possible because you may not have access to the source code or may not be free to modify it. If you can modify the original class, you need to understand the implementation's synchronization policy so that you can enhance it in a manner consistent with its original design. Adding the new method directly to the class means that all the code that implements the synchronization policy for that class is still contained in one source file, facilitating easier comprehension and maintenance.

Another approach is to extend the class, assuming it was designed for extension. `BetterVector` in Listing 4.13 extends `Vector` to add a `putIfAbsent` method. Extending `Vector` is straightforward enough, but not all classes expose enough of their state to subclasses to admit this approach.

Extension is more fragile than adding code directly to a class, because the implementation of the synchronization policy is now distributed over multiple, separately maintained source files. If the underlying class were to change its synchronization policy by choosing a different lock to guard its state variables, the subclass would subtly and silently break, because it no longer used the right lock to control concurrent access to the base class's state. (The synchronization policy of `Vector` is fixed by its specification, so `BetterVector` would not suffer from this problem.)

```
@ThreadSafe
public class BetterVector<E> extends Vector<E> {
    public synchronized boolean putIfAbsent(E x) {
        boolean absent = !contains(x);
        if (absent)
            add(x);
        return absent;
    }
}
```

Listing 4.13. Extending `Vector` to have a put-if-absent method.

4.4.1 Client-side locking

For an `ArrayList` wrapped with a `Collections.synchronizedList` wrapper, neither of these approaches—adding a method to the original class or extending the class—works because the client code does not even know the class of the `List` object returned from the synchronized wrapper factories. A third strategy is to extend the functionality of the class without extending the class itself by placing extension code in a "helper" class.

Listing 4.14 shows a failed attempt to create a helper class with an atomic put-if-absent operation for operating on a thread-safe `List`.

```
@NotThreadSafe
public class ListHelper<E> {
    public List<E> list =
        Collections.synchronizedList(new ArrayList<E>());
    ...
    public synchronized boolean putIfAbsent(E x) {
        boolean absent = !list.contains(x);
        if (absent)
            list.add(x);
        return absent;
    }
}
```

Listing 4.14. Non-thread-safe attempt to implement put-if-absent. *Don't do this.*

Why wouldn't this work? After all, `putIfAbsent` is `synchronized`, right? The problem is that it synchronizes on the *wrong lock*. Whatever lock the `List` uses to guard its state, it sure isn't the lock on the `ListHelper`. `ListHelper` provides only the *illusion of synchronization*; the various list operations, while all `synchronized`, use different locks, which means that `putIfAbsent` is *not* atomic relative to other operations on the `List`. So there is no guarantee that another thread won't modify the list while `putIfAbsent` is executing.

To make this approach work, we have to use the *same* lock that the `List` uses by using *client-side locking* or *external locking*. Client-side locking entails guarding client code that uses some object *X* with the lock *X* uses to guard its own state. In order to use client-side locking, you must know what lock *X* uses.

The documentation for `Vector` and the synchronized wrapper classes states, albeit obliquely, that they support client-side locking, by using the intrinsic lock for the `Vector` or the wrapper collection (not the wrapped collection). Listing 4.15 shows a `putIfAbsent` operation on a thread-safe `List` that correctly uses client-side locking.

```
@ThreadSafe
public class ListHelper<E> {
    public List<E> list =
        Collections.synchronizedList(new ArrayList<E>());
    ...
    public boolean putIfAbsent(E x) {
        synchronized (list) {
            boolean absent = !list.contains(x);
            if (absent)
                list.add(x);
            return absent;
        }
    }
}
```

Listing 4.15. Implementing put-if-absent with client-side locking.

If extending a class to add another atomic operation is fragile because it distributes the locking code for a class over multiple classes in an object hierarchy, client-side locking is even more fragile because it entails putting locking code for class *C* into classes that are totally unrelated to *C*. Exercise care when using client-side locking on classes that do not commit to their locking strategy.

Client-side locking has a lot in common with class extension—they both couple the behavior of the derived class to the implementation of the base class. Just as extension violates encapsulation of implementation [EJ Item 14], client-side locking violates encapsulation of synchronization policy.

4.4.2 Composition

There is a less fragile alternative for adding an atomic operation to an existing class: *composition*. `ImprovedList` in Listing 4.16 implements the `List` operations by delegating them to an underlying `List` instance, and adds an atomic `putIfAbsent` method. (Like `Collections.synchronizedList` and other collections wrappers, `ImprovedList` assumes that once a list is passed to its constructor, the client will not use the underlying list directly again, accessing it only through the `ImprovedList`.)

```
@ThreadSafe
public class ImprovedList<T> implements List<T> {
    private final List<T> list;

    public ImprovedList(List<T> list) { this.list = list; }

    public synchronized boolean putIfAbsent(T x) {
        boolean contains = list.contains(x);
        if (contains)
            list.add(x);
        return !contains;
    }

    public synchronized void clear() { list.clear(); }
    // ... similarly delegate other List methods
}
```

Listing 4.16. Implementing put-if-absent using composition.

`ImprovedList` adds an additional level of locking using its own intrinsic lock. It does not care whether the underlying `List` is thread-safe, because it provides its own consistent locking that provides thread safety even if the `List` is not thread-safe or changes its locking implementation. While the extra layer of synchronization may add some small performance penalty,[7] the implementation in `ImprovedList` is less fragile than attempting to mimic the locking strategy of another object. In effect, we've used the Java monitor pattern to encapsulate an existing `List`, and this is guaranteed to provide thread safety so long as our class holds the only outstanding reference to the underlying `List`.

4.5 Documenting synchronization policies

Documentation is one of the most powerful (and, sadly, most underutilized) tools for managing thread safety. Users look to the documentation to find out if a class is thread-safe, and maintainers look to the documentation to understand the implementation strategy so they can maintain it without inadvertently compromising safety. Unfortunately, both of these constituencies usually find less information in the documentation than they'd like.

Document a class's thread safety guarantees for its clients; document its synchronization policy for its maintainers.

[7] The penalty will be small because the synchronization on the underlying `List` is guaranteed to be uncontended and therefore fast; see Chapter 11.

Each use of synchronized, volatile, or any thread-safe class reflects a *synchronization policy* defining a strategy for ensuring the integrity of data in the face of concurrent access. That policy is an element of your program's design, and should be documented. Of course, the best time to document design decisions is at design time. Weeks or months later, the details may be a blur—so write it down before you forget.

Crafting a synchronization policy requires a number of decisions: which variables to make volatile, which variables to guard with locks, which lock(s) guard which variables, which variables to make immutable or confine to a thread, which operations must be atomic, etc. Some of these are strictly implementation details and should be documented for the sake of future maintainers, but some affect the publicly observable locking behavior of your class and should be documented as part of its specification.

At the very least, document the thread safety guarantees made by a class. Is it thread-safe? Does it make callbacks with a lock held? Are there any specific locks that affect its behavior? Don't force clients to make risky guesses. If you don't want to commit to supporting client-side locking, that's fine, but say so. If you want clients to be able to create new atomic operations on your class, as we did in Section 4.4, you need to document which locks they should acquire to do so safely. If you use locks to guard state, document this for future maintainers, because it's so easy—the @GuardedBy annotation will do the trick. If you use more subtle means to maintain thread safety, document them because they may not be obvious to maintainers.

The current state of affairs in thread safety documentation, even in the platform library classes, is not encouraging. How many times have you looked at the Javadoc for a class and wondered whether it was thread-safe?[8] Most classes don't offer any clue either way. Many official Java technology specifications, such as servlets and JDBC, woefully underdocument their thread safety promises and requirements.

While prudence suggests that we not assume behaviors that aren't part of the specification, we have work to get done, and we are often faced with a choice of bad assumptions. Should we assume an object is thread-safe because it seems that it ought to be? Should we assume that access to an object can be made thread-safe by acquiring its lock first? (This risky technique works only if we control *all* the code that accesses that object; otherwise, it provides only the illusion of thread safety.) Neither choice is very satisfying.

To make matters worse, our intuition may often be wrong on which classes are "probably thread-safe" and which are not. As an example, java.text.SimpleDateFormat isn't thread-safe, but the Javadoc neglected to mention this until JDK 1.4. That this particular class isn't thread-safe comes as a surprise to many developers. How many programs mistakenly create a shared instance of a non-thread-safe object and used it from multiple threads, unaware that this might cause erroneous results under heavy load?

The problem with SimpleDateFormat could be avoided by not assuming a class is thread-safe if it doesn't say so. On the other hand, it is impossible to

[8] If you've never wondered this, we admire your optimism.

develop a servlet-based application without making some pretty questionable assumptions about the thread safety of container-provided objects like HttpSession. Don't make your customers or colleagues have to make guesses like this.

4.5.1 Interpreting vague documentation

Many Java technology specifications are silent, or at least unforthcoming, about thread safety guarantees and requirements for interfaces such as ServletContext, HttpSession, or DataSource.[9] Since these interfaces are implemented by your container or database vendor, you often can't look at the code to see what it does. Besides, you don't want to rely on the implementation details of one particular JDBC driver—you want to be compliant with the standard so your code works properly with any JDBC driver. But the words "thread" and "concurrent" do not appear at all in the JDBC specification, and appear frustratingly rarely in the servlet specification. So what do you do?

You are going to have to guess. One way to improve the quality of your guess is to interpret the specification from the perspective of someone who will *implement* it (such as a container or database vendor), as opposed to someone who will merely use it. Servlets are always called from a container-managed thread, and it is safe to assume that if there is more than one such thread, the container knows this. The servlet container makes available certain objects that provide service to multiple servlets, such as HttpSession or ServletContext. So the servlet container should expect to have these objects accessed concurrently, since it has created multiple threads and called methods like Servlet.service from them that could reasonably be expected to access the ServletContext.

Since it is impossible to imagine a single-threaded context in which these objects would be useful, one has to assume that they have been made thread-safe, even though the specification does not explicitly require this. Besides, if they required client-side locking, on what lock should the client code synchronize? The documentation doesn't say, and it seems absurd to guess. This "reasonable assumption" is further bolstered by the examples in the specification and official tutorials that show how to access ServletContext or HttpSession and do not use any client-side synchronization.

On the other hand, the objects placed in the ServletContext or HttpSession with setAttribute are owned by the web application, not the servlet container. The servlet specification does not suggest any mechanism for coordinating concurrent access to shared attributes. So attributes stored by the container on behalf of the web application should be thread-safe or effectively immutable. If all the container did was store these attributes on behalf of the web application, another option would be to ensure that they are consistently guarded by a lock when accessed from servlet application code. But because the container may want to serialize objects in the HttpSession for replication or passivation purposes, and the servlet container can't possibly know your locking protocol, you should make them thread-safe.

[9] We find it particularly frustrating that these omissions persist despite multiple major revisions of the specifications.

One can make a similar inference about the JDBC DataSource interface, which represents a pool of reusable database connections. A DataSource provides service to an application, and it doesn't make much sense in the context of a single-threaded application. It is hard to imagine a use case that doesn't involve calling getConnection from multiple threads. And, as with servlets, the examples in the JDBC specification do not suggest the need for any client-side locking in the many code examples using DataSource. So, even though the specification doesn't promise that DataSource is thread-safe or require container vendors to provide a thread-safe implementation, by the same "it would be absurd if it weren't" argument, we have no choice but to assume that DataSource.getConnection does not require additional client-side locking.

On the other hand, we would not make the same argument about the JDBC Connection objects dispensed by the DataSource, since these are not necessarily intended to be shared by other activities until they are returned to the pool. So if an activity that obtains a JDBC Connection spans multiple threads, it must take responsibility for ensuring that access to the Connection is properly guarded by synchronization. (In most applications, activities that use a JDBC Connection are implemented so as to confine the Connection to a specific thread anyway.)