

## Chapter 16. The Java Memory Model

Throughout this book, we've mostly avoided the low-level details of the Java Memory Model (JMM) and instead focused on higher-level design issues such as safe publication, specification of, and adherence to synchronization policies. These derive their safety from the JMM, and you may find it easier to use these mechanisms effectively when you understand why they work. This chapter pulls back the curtain to reveal the low-level requirements and guarantees of the Java Memory Model and the reasoning behind some of the higher-level design rules offered in this book.

### 16.1. What is a Memory Model, and Why would I Want One?

Suppose one thread assigns a value to `avariable`:

```
avariable = 3;
```

A memory model addresses the question "Under what conditions does a thread that reads `avariable` see the value 3?" This may sound like a dumb question, but in the absence of synchronization, there are a number of reasons a thread might not immediately - or ever - see the results of an operation in another thread. Compilers may generate instructions in a different order than the "obvious" one suggested by the source code, or store variables in registers instead of in memory; processors may execute instructions in parallel or out of order; caches may vary the order in which writes to variables are committed to main memory; and values stored in processor-local caches may not be visible to other processors. These factors can prevent a thread from seeing the most up-to-date value for a variable and can cause memory actions in other threads to appear to happen out of order - if you don't use adequate synchronization.

In a single-threaded environment, all these tricks played on our program by the environment are hidden from us and have no effect other than to speed up execution. The Java Language Specification requires the JVM to maintain within thread as-if-serial semantics: as long as the program has the same result as if it were executed in program order in a strictly sequential environment, all these games are permissible. And that's a good thing, too, because these rearrangements are responsible for much of the improvement in computing performance in recent years. Certainly higher clock rates have contributed to improved performance, but so has increased parallelism - pipelined superscalar execution units, dynamic instruction scheduling, speculative execution, and sophisticated multilevel memory caches. As processors have become more sophisticated, so too have compilers, rearranging instructions to facilitate optimal execution and using sophisticated global register-allocation algorithms. And as processor manufacturers transition to multicore processors, largely because clock rates are getting harder to increase economically, hardware parallelism will only increase.

In a multithreaded environment, the illusion of sequentiality cannot be maintained without significant performance cost. Since most of the time threads within a concurrent application are each "doing their own thing", excessive inter-thread coordination would only slow down the application to no real benefit. It is only when multiple threads share data that it is necessary to coordinate their activities, and the JVM relies on the program to identify when this is happening by using synchronization.

The JMM specifies the minimal guarantees the JVM must make about when writes to variables become visible to other threads. It was designed to balance the need for predictability and ease of program development with the realities of implementing high-performance JVMs on a wide range of popular processor architectures. Some aspects of the JMM may be disturbing at first if you are not familiar with the tricks used by modern processors and compilers to squeeze extra performance out of your program.

#### 16.1.1. Platform Memory Models

In a shared-memory multiprocessor architecture, each processor has its own cache that is periodically reconciled with main memory. Processor architectures provide varying degrees of cache coherence; some provide minimal guarantees that allow different processors to see different values for the same memory location at virtually any time. The operating system, compiler, and runtime (and sometimes, the program, too) must make up the difference between what the hardware provides and what thread safety requires.

Ensuring that every processor knows what every other processor is doing at all times is expensive. Most of the time this information is not needed, so processors relax their memory-coherency guarantees to improve performance. An architecture's memory model tells programs what guarantees they can expect from the memory system, and specifies the special instructions required (called memory barriers or fences) to get the additional memory coordination guarantees required when sharing data. In order to shield the Java developer from the differences between memory

models across architectures, Java provides its own memory model, and the JVM deals with the differences between the JMM and the underlying platform's memory model by inserting memory barriers at the appropriate places.

One convenient mental model for program execution is to imagine that there is a single order in which the operations happen in a program, regardless of what processor they execute on, and that each read of a variable will see the last write in the execution order to that variable by any processor. This happy, if unrealistic, model is called sequential consistency. Software developers often mistakenly assume sequential consistency, but no modern multiprocessor offers sequential consistency and the JMM does not either. The classic sequential computing model, the von Neumann model, is only a vague approximation of how modern multiprocessors behave.

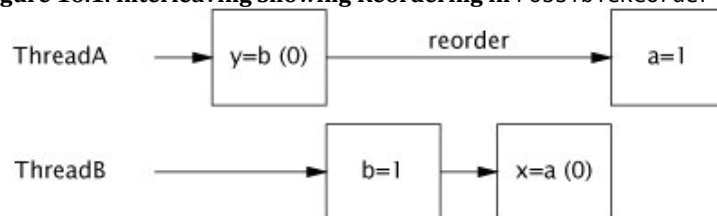
The bottom line is that modern shared-memory multiprocessors (and compilers) can do some surprising things when data is shared across threads, unless you've told them not to through the use of memory barriers. Fortunately, Java programs need not specify the placement of memory barriers; they need only identify when shared state is being accessed, through the proper use of synchronization.

### 16.1.2. Reordering

In describing race conditions and atomicity failures in Chapter 2, we used interaction diagrams depicting "unlucky timing" where the scheduler interleaved operations so as to cause incorrect results in insufficiently synchronized programs. To make matters worse, the JMM can permit actions to appear to execute in different orders from the perspective of different threads, making reasoning about ordering in the absence of synchronization even more complicated. The various reasons why operations might be delayed or appear to execute out of order can all be grouped into the general category of reordering.

PossibleReordering in Listing 16.1 illustrates how difficult it is to reason about the behavior of even the simplest concurrent programs unless they are correctly synchronized. It is fairly easy to imagine how PossibleReordering could print (1, 0), or (0, 1), or (1, 1): thread A could run to completion before B starts, B could run to completion before A starts, or their actions could be interleaved. But, strangely, PossibleReordering can also print (0, 0)! The actions in each thread have no dataflow dependence on each other, and accordingly can be executed out of order. (Even if they are executed in order, the timing by which caches are flushed to main memory can make it appear, from the perspective of B, that the assignments in A occurred in the opposite order.) Figure 16.1 shows a possible interleaving with reordering that results in printing (0, 0).

Figure 16.1. Interleaving Showing Reordering in PossibleReordering.



PossibleReordering is a trivial program, and it is still surprisingly tricky to enumerate its possible results. Reordering at the memory level can make programs behave unexpectedly. It is prohibitively difficult to reason about ordering in the absence of synchronization; it is much easier to ensure that your program uses synchronization appropriately. Synchronization inhibits the compiler, runtime, and hardware from reordering memory operations in ways that would violate the visibility guarantees provided by the JMM.<sup>[1]</sup>

[1] On most popular processor architectures, the memory model is strong enough that the performance cost of a volatile read is in line with that of a nonvolatile read.

### 16.1.3. The Java Memory Model in 500 Words or Less

The Java Memory Model is specified in terms of actions, which include reads and writes to variables, locks and unlocks of monitors, and starting and joining with threads. The JMM defines a partial ordering<sup>[2]</sup> called happens-before on all actions within the program. To guarantee that the thread executing action B can see the results of action A (whether or not A and B occur in different threads), there must be a happens-before relationship between A and B. In the absence of a happens-before ordering between two operations, the JVM is free to reorder them as it pleases.

[2] A partial ordering  $\prec$  is a relation on a set that is antisymmetric, reflexive, and transitive, but for any two elements  $x$  and  $y$ , it need not be the case that  $x \prec y$  or  $y \prec x$ . We use partial orderings every day to express preferences; we may prefer sushi to cheeseburgers and Mozart to Mahler, but we don't necessarily have a clear preference between cheeseburgers and Mozart.

**Listing 16.1. Insufficiently Synchronized Program that can have Surprising Results. *Don't Do this.***

```

public class PossibleReordering {
    static int x = 0, y = 0;
    static int a = 0, b = 0;

    public static void main(String[] args)
        throws InterruptedException {
        Thread one = new Thread(new Runnable() {
            public void run() {
                a = 1;
                x = b;
            }
        });
        Thread other = new Thread(new Runnable() {
            public void run() {
                b = 1;
                y = a;
            }
        });
        one.start(); other.start();
        one.join();  other.join();
        System.out.println("(" + x + ", " + y + ")");
    }
}

```

A data race occurs when a variable is read by more than one thread, and written by at least one thread, but the reads and writes are not ordered by happens-before. A correctly synchronized program is one with no data races; correctly synchronized programs exhibit sequential consistency, meaning that all actions within the program appear to happen in a fixed, global order.

The rules for happens-before are:

- Program order rule. Each action in a thread happens-before every action in that thread that comes later in the program order.
- Monitor lock rule. An unlock on a monitor lock happens-before every subsequent lock on that same monitor lock.<sup>[3]</sup>
- Volatile variable rule. A write to a volatile field happens-before every subsequent read of that same field.<sup>[4]</sup>
- Thread start rule. A call to `Thread.start` on a thread happens-before every action in the started thread.
- Thread termination rule. Any action in a thread happens-before any other thread detects that thread has terminated, either by successfully return from `Thread.join` or by `Thread.isAlive` returning false.
- Interruption rule. A thread calling `interrupt` on another thread happens-before the interrupted thread detects the interrupt (either by having `InterruptedException` thrown, or invoking `isInterrupted` or `interrupted`).
- Finalizer rule. The end of a constructor for an object happens-before the start of the finalizer for that object.
- Transitivity. If A happens-before B, and B happens-before C, then A happens-before C.

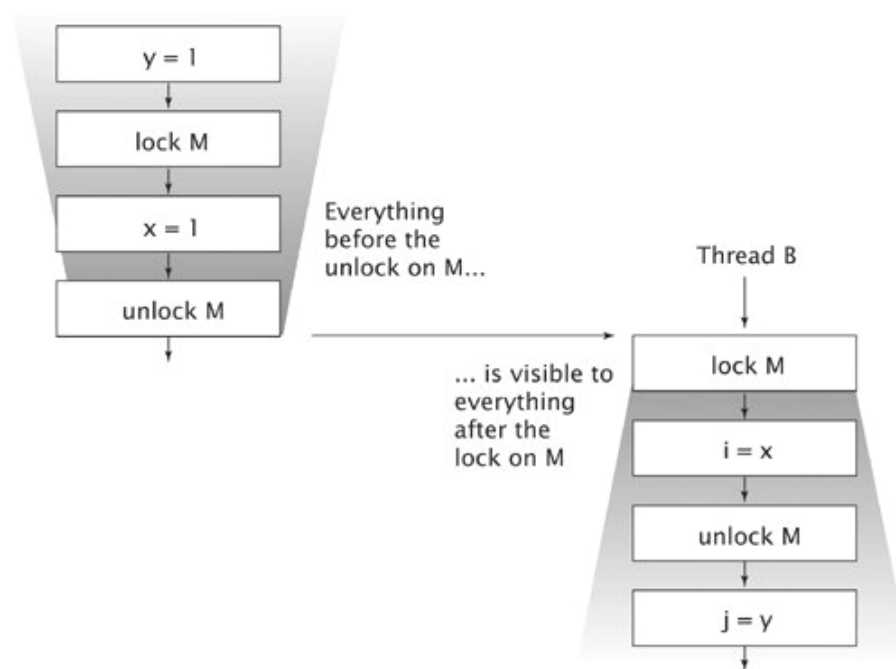
[3] Locks and unlocks on explicit `LOCK` objects have the same memory semantics as intrinsic locks.

[4] Reads and writes of atomic variables have the same memory semantics as volatile variables.

Even though actions are only partially ordered, synchronization actions - lock acquisition and release, and reads and writes of `volatile` variables - are totally ordered. This makes it sensible to describe happens-before in terms of "subsequent" lock acquisitions and reads of `volatile` variables.

Figure 16.2 illustrates the happens-before relation when two threads synchronize using a common lock. All the actions within thread A are ordered by the program order rule, as are the actions within thread B. Because A releases lock M and B subsequently acquires M, all the actions in A before releasing the lock are therefore ordered before the actions in B after acquiring the lock. When two threads synchronize on different locks, we can't say anything about the ordering of actions between them - there is no happens-before relation between the actions in the two threads.

**Figure 16.2. Illustration of Happens-before in the Java Memory Model.**



#### 16.1.4. Piggybacking on Synchronization

Because of the strength of the happens-before ordering, you can sometimes piggyback on the visibility properties of an existing synchronization. This entails combining the program order rule for happens-before with one of the other ordering rules (usually the monitor lock or volatile variable rule) to order accesses to a variable not otherwise guarded by a lock. This technique is very sensitive to the order in which statements occur and is therefore quite fragile; it is an advanced technique that should be reserved for squeezing the last drop of performance out of the most performance-critical classes like `ReentrantLock`.

The implementation of the protected `AbstractQueuedSynchronizer` methods in `FutureTask` illustrates piggybacking. AQS maintains an integer of synchronizer state that `FutureTask` uses to store the task state: running, completed, or cancelled. But `FutureTask` also maintains additional variables, such as the result of the computation. When one thread calls `set` to save the result and another thread calls `get` to retrieve it, the two had better be ordered by happens-before. This could be done by making the reference to the result `volatile`, but it is possible to exploit existing synchronization to achieve the same result at lower cost.

`FutureTask` is carefully crafted to ensure that a successful call to `tryReleaseShared` always happens-before a subsequent call to `tryAcquireShared`; `tryReleaseShared` always writes to a volatile variable that is read by `tryAcquireShared`. Listing 16.2 shows the `innerSet` and `innerGet` methods that are called when the result is saved or retrieved; since `innerSet` writes `result` before calling `releaseShared` (which calls `tryReleaseShared`) and `innerGet` reads `result` after calling `acquireShared` (which calls `tryAcquireShared`), the program order rule combines with the volatile variable rule to ensure that the write of `result` in `innerSet` happens-before the read of `result` in `innerGet`.

**Listing 16.2. Inner Class of FutureTask Illustrating Synchronization Piggybacking.**

```
// Inner class of FutureTask
private final class Sync extends AbstractQueuedSynchronizer {
    private static final int RUNNING = 1, RAN = 2, CANCELLED = 4;
    private V result;
    private Exception exception;

    void innerSet(V v) {
        while (true) {
            int s = getState();
            if (ranOrCancelled(s))
                return;
            if (compareAndSetState(s, RAN))
                break;
        }
        result = v;
        releaseShared(0);
        done();
    }

    V innerGet() throws InterruptedException, ExecutionException {
        acquireSharedInterruptibly(0);
        if (getState() == CANCELLED)
            throw new CancellationException();
        if (exception != null)
            throw new ExecutionException(exception);
        return result;
    }
}
```

We call this technique "piggybacking" because it uses an existing happens-before ordering that was created for some other reason to ensure the visibility of object X, rather than creating a happens-before ordering specifically for publishing X.

Piggybacking of the sort employed by FutureTask is quite fragile and should not be undertaken casually. However, in some cases piggybacking is perfectly reasonable, such as when a class commits to a happens-before ordering between methods as part of its specification. For example, safe publication using a BlockingQueue is a form of piggybacking. One thread putting an object on a queue and another thread subsequently retrieving it constitutes safe publication because there is guaranteed to be sufficient internal synchronization in a BlockingQueue implementation to ensure that the enqueue happens-before the dequeue.

Other happens-before orderings guaranteed by the class library include:

- Placing an item in a thread-safe collection happens-before another thread retrieves that item from the collection;
- Counting down on a CountDownLatch happens-before a thread returns from await on that latch;
- Releasing a permit to a Semaphore happens-before acquiring a permit from that same Semaphore;
- Actions taken by the task represented by a Future happens-before another thread successfully returns from Future.get;
- Submitting a Runnable or Callable to an Executor happens-before the task begins execution; and
- A thread arriving at a CyclicBarrier or Exchanger happens-before the other threads are released from that same barrier or exchange point. If CyclicBarrier uses a barrier action, arriving at the barrier happens-before the barrier action, which in turn happens-before threads are released from the barrier.

## 16.2. Publication

Chapter 3 explored how an object could be safely or improperly published. The safe publication techniques described there derive their safety from guarantees provided by the JMM; the risks of improper publication are consequences of the absence of a happens-before ordering between publishing a shared object and accessing it from another thread.

### 16.2.1. Unsafe Publication

The possibility of reordering in the absence of a happens-before relationship explains why publishing an object without adequate synchronization can allow another thread to see a partially constructed object (see Section 3.5). Initializing a new object involves writing to variables - the new object's fields. Similarly, publishing a reference involves writing to another variable - the reference to the new object. If you do not ensure that publishing the shared reference happens-

before another thread loads that shared reference, then the write of the reference to the new object can be reordered (from the perspective of the thread consuming the object) with the writes to its fields. In that case, another thread could see an up-to-date value for the object reference but out-of-date values for some or all of that object's state a partially constructed object.

Unsafe publication can happen as a result of an incorrect lazy initialization, as shown in Figure 16.3. At first glance, the only problem here seems to be the race condition described in Section 2.2.2. Under certain circumstances, such as when all instances of the `Resource` are identical, you might be willing to overlook these (along with the inefficiency of possibly creating the `Resource` more than once). Unfortunately, even if these defects are overlooked, `UnsafeLazyInitialization` is still not safe, because another thread could observe a reference to a partially constructed `Resource`.

**Listing 16.3. Unsafe Lazy Initialization. Don't Do this.**



```
@NotThreadSafe
public class UnsafeLazyInitialization {
    private static Resource resource;

    public static Resource getInstance() {
        if (resource == null)
            resource = new Resource(); // unsafe publication
        return resource;
    }
}
```

Suppose thread A is the first to invoke `getInstance`. It sees that `resource` is `null`, instantiates a new `Resource`, and sets `resource` to reference it. When thread B later calls `getInstance`, it might see that `resource` already has a non-null value and just use the already constructed `Resource`. This might look harmless at first, but there is no happens-before ordering between the writing of `resource` in A and the reading of `resource` in B. A data race has been used to publish the object, and therefore B is not guaranteed to see the correct state of the `Resource`.

The `Resource` constructor changes the fields of the freshly allocated `Resource` from their default values (written by the object constructor) to their initial values. Since neither thread used synchronization, B could possibly see A's actions in a different order than A performed them. So even though A initialized the `Resource` before setting `resource` to reference it, B could see the write to `resource` as occurring before the writes to the fields of the `Resource`. B could thus see a partially constructed `Resource` that may well be in an invalid state and whose state may unexpectedly change later.

With the exception of immutable objects, it is not safe to use an object that has been initialized by another thread unless the publication happens-before the consuming thread uses it.

### 16.2.2. Safe Publication

The safe-publication idioms described in Chapter 3 ensure that the published object is visible to other threads because they ensure the publication happens-before the consuming thread loads a reference to the published object. If thread A places X on a `BlockingQueue` (and no thread subsequently modifies it) and thread B retrieves it from the queue, B is guaranteed to see X as A left it. This is because the `BlockingQueue` implementations have sufficient internal synchronization to ensure that the put happens-before the take. Similarly, using a shared variable guarded by a lock or a shared volatile variable ensures that reads and writes of that variable are ordered by happens-before.

This happens-before guarantee is actually a stronger promise of visibility and ordering than made by safe publication. When X is safely published from A to B, the safe publication guarantees visibility of the state of X, but not of the state of other variables A may have touched. But if A putting X on a queue happens-before B fetches X from that queue, not only does B see X in the state that A left it (assuming that X has not been subsequently modified by A or anyone else), but B sees everything A did before the handoff (again, subject to the same caveat).<sup>[5]</sup>

[5] The JMM guarantees that B sees a value at least as up-to-date as the value that A wrote; subsequent writes may or may not be visible.

Why did we focus so heavily on `@GuardedBy` and safe publication, when the JMM already provides us with the more powerful happens-before? Thinking in terms of handing off object ownership and publication fits better into most program designs than thinking in terms of visibility of individual memory writes. The happens-before ordering operates

at the level of individual memory accesses - it is a sort of "concurrency assembly language". Safe publication operates at a level closer to that of your program's design.

### 16.2.3. Safe Initialization Idioms

It sometimes makes sense to defer initialization of objects that are expensive to initialize until they are actually needed, but we have seen how the misuse of lazy initialization can lead to trouble. `unsafeLazyInitialization` can be fixed by making the `getResource` method synchronized, as shown in Listing 16.4. Because the code path through `getInstance` is fairly short (a test and a predicted branch), if `getInstance` is not called frequently by many threads, there is little enough contention for the `SafeLazyInitialization` lock that this approach offers adequate performance.

The treatment of static fields with initializers (or fields whose value is initialized in a static initialization block [JPL 2.2.1 and 2.5.3]) is somewhat special and offers additional thread-safety guarantees. Static initializers are run by the JVM at class initialization time, after class loading but before the class is used by any thread. Because the JVM acquires a lock during initialization [JLS 12.4.2] and this lock is acquired by each thread at least once to ensure that the class has been loaded, memory writes made during static initialization are automatically visible to all threads. Thus statically initialized objects require no explicit synchronization either during construction or when being referenced. However, this applies only to the as-constructed state - if the object is mutable, synchronization is still required by both readers and writers to make subsequent modifications visible and to avoid data corruption.

#### Listing 16.4. Thread-safe Lazy Initialization.

```
@ThreadSafe
public class SafeLazyInitialization {
    private static Resource resource;

    public synchronized static Resource getInstance() {
        if (resource == null)
            resource = new Resource();
        return resource;
    }
}
```

#### Listing 16.5. Eager Initialization.

```
@ThreadSafe
public class EagerInitialization {
    private static Resource resource = new Resource();

    public static Resource getResource() { return resource; }
}
```

Using eager initialization, shown in Listing 16.5, eliminates the synchronization cost incurred on each call to `getInstance` in `SafeLazyInitialization`. This technique can be combined with the JVM's lazy class loading to create a lazy initialization technique that does not require synchronization on the common code path. The lazy initialization holder class idiom [EJ Item 48] in Listing 16.6 uses a class whose only purpose is to initialize the `Resource`. The JVM defers initializing the `ResourceHolder` class until it is actually used [JLS 12.4.1], and because the `Resource` is initialized with a static initializer, no additional synchronization is needed. The first call to `getResource` by any thread causes `ResourceHolder` to be loaded and initialized, at which time the initialization of the `Resource` happens through the static initializer.

#### Listing 16.6. Lazy Initialization Holder Class Idiom.

```
@ThreadSafe
public class ResourceFactory {
    private static class ResourceHolder {
        public static Resource resource = new Resource();
    }

    public static Resource getResource() {
        return ResourceHolder.resource;
    }
}
```

### 16.2.4. Double-checked Locking

No book on concurrency would be complete without a discussion of the infamous double-checked locking (DCL) anti-pattern, shown in Listing 16.7. In very early JVMs, synchronization, even uncontended synchronization, had a significant performance cost. As a result, many clever (or at least clever-looking) tricks were invented to reduce the impact of synchronization - some good, some bad, and some ugly. DCL falls into the "ugly" category.

Again, because the performance of early JVMs left something to be desired, lazy initialization was often used to avoid potentially unnecessary expensive operations or reduce application startup time. A properly written lazy initialization



method requires synchronization. But at the time, synchronization was slow and, more importantly, not completely understood: the exclusion aspects were well enough understood, but the visibility aspects were not.

DCL purported to offer the best of both worlds - lazy initialization without paying the synchronization penalty on the common code path. The way it worked was first to check whether initialization was needed without synchronizing, and if the resource reference was not `null`, use it. Otherwise, synchronize and check again if the resource is initialized, ensuring that only one thread actually initializes the shared resource. The common code path - fetching a reference to an already constructed resource - doesn't use synchronization. And that's where the problem is: as described in Section 16.2.1, it is possible for a thread to see a partially constructed resource.

The real problem with DCL is the assumption that the worst thing that can happen when reading a shared object reference without synchronization is to erroneously see a stale value (in this case, `null`); in that case the DCL idiom compensates for this risk by trying again with the lock held. But the worst case is actually considerably worse - it is possible to see a current value of the reference but stale values for the object's state, meaning that the object could be seen to be in an invalid or incorrect state.

Subsequent changes in the JMM (Java 5.0 and later) have enabled DCL to work if resource is made `volatile`, and the performance impact of this is small since `volatile` reads are usually only slightly more expensive than nonvolatile reads. However, this is an idiom whose utility has largely passed - the forces that motivated it (slow uncontended synchronization, slow JVM startup) are no longer in play, making it less effective as an optimization. The lazy initialization holder idiom offers the same benefits and is easier to understand.

**Listing 16.7. Double-checked-locking Anti-pattern. *Don't Do this.***



```
@NotThreadSafe
public class DoubleCheckedLocking {
    private static Resource resource;

    public static Resource getInstance() {
        if (resource == null) {
            synchronized (DoubleCheckedLocking.class) {
                if (resource == null)
                    resource = new Resource();
            }
        }
        return resource;
    }
}
```

### 16.3. Initialization Safety

The guarantee of initialization safety allows properly constructed immutable objects to be safely shared across threads without synchronization, regardless of how they are published even if published using a data race. (This means that `UnsafeLazyInitialization` is actually safe if `Resource` is immutable.)

Without initialization safety, supposedly immutable objects like `String` can appear to change their value if synchronization is not used by both the publishing and consuming threads. The security architecture relies on the immutability of `String`; the lack of initialization safety could create security vulnerabilities that allow malicious code to bypass security checks.

Initialization safety guarantees that for properly constructed objects, all threads will see the correct values of final fields that were set by the constructor, regardless of how the object is published. Further, any variables that can be reached through a final field of a properly constructed object (such as the elements of a final array or the contents of a `HashMap` referenced by a final field) are also guaranteed to be visible to other threads. <sup>[6]</sup>

[6] This applies only to objects that are reachable only through `final` fields of the object under construction.

For objects with final fields, initialization safety prohibits reordering any part of construction with the initial load of a reference to that object. All writes to final fields made by the constructor, as well as to any variables reachable through those fields, become "frozen" when the constructor completes, and any thread that obtains a reference to that object is guaranteed to see a value that is at least as up to date as the frozen value. Writes that initialize variables reachable through final fields are not reordered with operations following the post-construction freeze.



Initialization safety means that `SafeStates` in Listing 16.8 could be safely published even through unsafe lazy initialization or stashing a reference to a `SafeStates` in a public static field with no synchronization, even though it uses no synchronization and relies on the non-thread-safe `HashSet`.

**Listing 16.8. Initialization Safety for Immutable Objects.**

```
@ThreadSafe
public class SafeStates {
    private final Map<String, String> states;

    public SafeStates() {
        states = new HashMap<String, String>();
        states.put("alaska", "AK");
        states.put("alabama", "AL");
        ...
        states.put("wyoming", "WY");
    }

    public String getAbbreviation(String s) {
        return states.get(s);
    }
}
```

However, a number of small changes to `SafeStates` would take away its thread safety. If `states` were not final, or if any method other than the constructor modified its contents, initialization safety would not be strong enough to safely access `SafeStates` without synchronization. If `SafeStates` had other non-final fields, other threads might still see incorrect values of those fields. And allowing the object to escape during construction invalidates the initialization-safety guarantee.

Initialization safety makes visibility guarantees only for the values that are reachable through final fields as of the time the constructor finishes. For values reachable through non-final fields, or values that may change after construction, you must use synchronization to ensure visibility.

## Summary

The Java Memory Model specifies when the actions of one thread on memory are guaranteed to be visible to another. The specifics involve ensuring that operations are ordered by a partial ordering called happens-before, which is specified at the level of individual memory and synchronization operations. In the absence of sufficient synchronization, some very strange things can happen when threads access shared data. However, the higher-level rules offered in Chapters 2 and 3, such as `@GuardedBy` and safe publication, can be used to ensure thread safety without resorting to the low-level details of happens-before.