**Exclusion 🡪 Synchronization 🡪 Java Memory Model**

***2.2.7.1 Atomicity***

Accesses and updates to the memory cells corresponding to fields of any type except long or

double are guaranteed to be atomic. This includes fields serving as references to other objects.

Additionally, atomicity extends to volatile long and double. (Even though non-volatile

longs and doubles are not guaranteed atomic, they are of course allowed to be.)

Atomicity guarantees ensure that when a non-long/double field is used in an expression, you

will obtain either its initial value or some value that was written by some thread, but not some jumble

of bits resulting from two or more threads both trying to write values at the same time. However, as

seen below, atomicity alone does not guarantee that you will get the value most recently written by

any thread. For this reason, atomicity guarantees *per se* normally have little impact on concurrent

program design.

***2.2.7.2 Visibility***

Changes to fields made by one thread are *guaranteed* to be visible to other threads only under the

following conditions:

1. A writing thread releases a synchronization lock and a reading thread subsequently acquires

that same synchronization lock.

In essence, releasing a lock forces a flush of all writes from working memory employed by

the thread, and acquiring a lock forces a (re)load of the values of accessible fields. While lock

actions provide exclusion only for the operations performed within a synchronized

method or block, these memory effects are defined to cover all fields used by the thread

performing the action.

Note the double meaning of synchronized: it deals with locks that permit higher-level

synchronization protocols, while at the same time dealing with the memory system

(sometimes via low-level *memory barrier* machine instructions) to keep value representations

in synch across threads. This reflects one way in which concurrent programming bears more

similarity to distributed programming than to sequential programming. The latter sense of

synchronized may be viewed as a mechanism by which a method running in one thread

indicates that it is willing to send and/or receive changes to variables to and from methods

running in other threads. From this point of view, using locks and passing messages might be

seen merely as syntactic variants of each other.

2. If a field is declared as volatile, any value written to it is flushed and made visible by

the writer thread before the writer thread performs any further memory operation (i.e., for the

purposes at hand it is flushed immediately). Reader threads must reload the values of

volatile fields upon each access.

3. The first time a thread accesses a field of an object, it sees either the initial value[4] of the field

or a value since written by some other thread.

[4] As of this writing, the *JLS* does not yet clearly state that the visible initial value read for an

initialized final field is the value assigned in its initializer or constructor. However, this

anticipated clarification is assumed throughout this book. The visible initial default values of nonfinal

fields are zero for scalars and null for references.

Among other consequences, it is bad practice to make available the reference to an

incompletely constructed object (see § 2.1.2). It can also be risky to start new threads inside a

constructor, especially in a class that may be subclassed. Thread.start has the same

memory effects as a lock release by the thread calling start, followed by a lock acquire by

the started thread. If a Runnable superclass invokes new Thread(this).start()

before subclass constructors execute, then the object might not be fully initialized when the

run method executes. Similarly, if you create and start a new thread T and then create an

object X used by thread T, you cannot be sure that the fields of X will be visible to T unless

you employ synchronization surrounding all references to object X. Or, when applicable, you

can create X before starting T.

4. As a thread terminates, all written variables are flushed to main memory.

For example, if one thread synchronizes on the termination of another thread using

Thread.join, then it is guaranteed to see the effects made by that thread (see § 4.3.2).

Note that visibility problems *never* arise when passing references to objects across methods in the

*same* thread.

The memory model guarantees that, given the eventual occurrence of the above operations, a

particular update to a particular field made by one thread will eventually be visible to another. But

*eventually* can be an arbitrarily long time. Long stretches of code in threads that use no

synchronization can be hopelessly out of synch with other threads with respect to values of fields. In

particular, it is always wrong to write loops waiting for values written by other threads unless the

fields are volatile or accessed via synchronization (see § 3.2.6).

The model also allows inconsistent visibility in the absence of synchronization. For example, it is

possible to obtain a fresh value for one field of an object, but a stale value for another. Similarly, it is

possible to read a fresh, updated value of a reference variable, but a stale value of one of the fields of

the object now being referenced.

However, the rules do not require visibility failures across threads, they merely allow these failures to

occur. This is one aspect of the fact that not using synchronization in multithreaded code doesn't

guarantee safety violations, it just allows them. On most current JVM implementations and platforms,

even those employing multiple processors, detectable visibility failures rarely occur. The use of

common caches across threads sharing a CPU, the lack of aggressive compiler-based optimizations,

and the presence of strong cache consistency hardware often cause values to act as if they propagate

immediately among threads. This makes testing for freedom from visibility-based errors impractical,

since such errors might occur extremely rarely, or only on platforms you do not have access to, or

only on those that have not even been built yet. These same comments apply to multithreaded safety

failures more generally. Concurrent programs that do not use synchronization fail for many reasons,

including memory consistency problems.

***2.2.7.3 Ordering***

Ordering rules fall under two cases, within-thread and between-thread:

• From the point of view of the thread performing the actions in a method, instructions proceed

in the normal as-if-serial manner that applies in sequential programming languages.

• From the point of view of other threads that might be "spying" on this thread by concurrently

running unsynchronized methods, almost anything can happen. The only useful constraint is

that the relative orderings of synchronized methods and blocks, as well as operations on

volatile fields, are always preserved.

Again, these are only the minimal guaranteed properties. In any given program or platform, you may

find stricter orderings. But you cannot rely on them, and you may find it difficult to test for code that

would fail on JVM implementations that have different properties but still conform to the rules.

Note that the within-thread point of view is implicitly adopted in all other discussions of semantics in

*JLS*. For example, arithmetic expression evaluation is performed in left-to-right order (*JLS* section

15.6) as viewed by the thread performing the operations, but not necessarily as viewed by other

threads.

The within-thread as-if-serial property is helpful only when only one thread at a time is manipulating

variables, due to synchronization, structural exclusion, or pure chance. When multiple threads are all

running unsynchronized code that reads and writes common fields, then arbitrary interleavings,

atomicity failures, race conditions, and visibility failures may result in execution patterns that make

the notion of as-if-serial just about meaningless with respect to any given thread.

Even though *JLS* addresses some particular legal and illegal reorderings that can occur, interactions

with these other issues reduce practical guarantees to saying that the results may reflect just about any

possible interleaving of just about any possible reordering. So there is no point in trying to reason

about the ordering properties of such code.

***2.2.7.4 Volatile***

In terms of atomicity, visibility, and ordering, declaring a field as volatile is nearly identical in

effect to using a little fully synchronized class protecting only that field via get/set methods, as in:

final class VFloat {

private float value;

final synchronized void set(float f) { value = f; }

final synchronized float get() { return value; }

}

Declaring a field as volatile differs only in that no locking is involved. In particular, composite

read/write operations such as the "++'' operation on volatile variables are *not* performed

atomically.

Also, ordering and visibility effects surround only the single access or update to the volatile field

itself. Declaring a reference field as volatile does not ensure visibility of *non*-volatile fields

that are accessed via this reference. Similarly, declaring an array field as volatile does not ensure

visibility of its elements. Volatility cannot be manually propagated for arrays because array elements

themselves cannot be declared as volatile.

Because no locking is involved, declaring fields as volatile is likely to be cheaper than using

synchronization, or at least no more expensive. However, if volatile fields are accessed

frequently inside methods, their use is likely to lead to slower performance than would locking the

entire methods.

Declaring fields as volatile can be useful when you do not need locking for any other reason, yet

values must be accurately accessible across multiple threads. This may occur when:

• The field need not obey any invariants with respect to others.

• Writes to the field do not depend on its current value.

• No thread ever writes an illegal value with respect to intended semantics.

• The actions of readers do not depend on values of other non-volatile fields.

Using volatile fields can make sense when it is somehow known that only one thread can change

a field, but many other threads are allowed to read it at any time. For example, a Thermometer

class might declare its temperature field as volatile. As discussed in § 3.4.2, a

volatile can be useful as a completion flag. Additional examples are illustrated in § 4.4, where

the use of lightweight executable frameworks automates some aspects of synchronization, but

volatile declarations are needed to ensure that result field values are visible across tasks.