



DZone > Java Zone > Java Concurrency: Understanding the 'Volatile' Keyword

Java Concurrency: Understanding the 'Volatile' Keyword



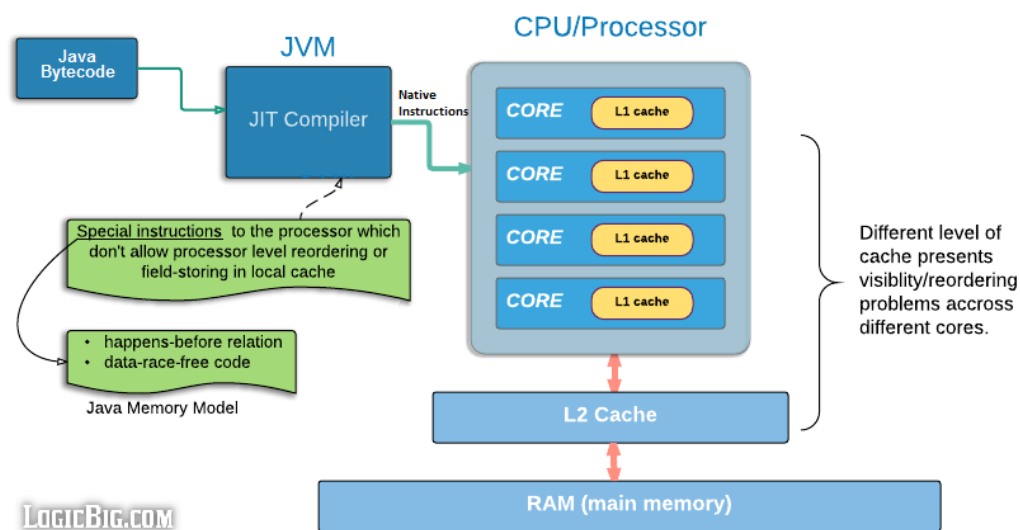
by Bruno Masci · Mar. 08, 21 · Java Zone · Tutorial

Like (15) Comment Save Tweet



MariaDB SkySQL Migration Guide

Learn how to seamlessly migrate your current MariaDB cloud or on-prem deployment to SkySQL. [Learn more](#) ▶



In this article, I would like to share a different approach to this subject that maybe helpful for you, and also to talk about some particularities and misconceptions about `volatile`. There's very good material on this topic out there (like "Java Concurrency in Practice" [JCIP]). I encourage you to read it!

We will be talking about the semantics of `volatile` as defined from Java 5 onwards (in previous Java versions, `volatile` had no memory-consistency/state-visibility semantic).

Java Concurrency: What Is Volatile?

Short answer: `volatile` is a keyword we can apply to a **field** to ensure that when one thread writes a value to that field, the value written is "immediately available" to any thread that subsequently reads it (visibility feature).

Context

When multiple threads need to interact with some shared data, there are three aspects to consider:

- **Visibility:** The effects of an action on the shared data by a thread should be observable by other threads

- **Atomicity:** No thread should interfere while another thread is executing some actions on the shared data.

In the absence of necessary synchronizations, the compiler, Runtime, or processors may apply all sorts of optimizations, like **code reordering** and **caching**. Those optimizations can interact with incorrectly synchronized code in ways that are not immediately obvious when the source code is examined. Even if statements execute in the order of their appearance in a thread, caching can prevent the latest values from being reflected in the main memory.

In order to prevent incorrect or unpredictable outcomes, we can use (some form of) **synchronization**.

It is worth mentioning that synchronization doesn't imply the use of the `synchronized` keyword (which is based on implicit/monitor/intrinsic `locks` internally); for example, `lock objects` and the `volatile` keyword are also synchronization mechanisms.

In a multi-threaded context, we usually must resort to explicit synchronization. But, in particular scenarios, there are some alternatives we could use that do not rely on synchronization, such as `atomic methods` and `immutable objects`.

To illustrate the risks of an improperly synchronized program, let's consider the following example borrowed from JCIIP book. This program could simply print 42, or even print 0, or even hang forever!

```

Java
1 @NotThreadSafe
2 public class NoVisibility {
3     private static boolean ready;
4     private static int number;
5
6     private static class ReaderThread extends Thread {
7         public void run() {
8             while (!ready)
9                 Thread.yield();
10            System.out.println(number);
11        }
12    }
13
14    public static void main(String[] args) {
15        new ReaderThread().start();
16        number = 42;
17        ready = true;
18    }
19 }

```

Why could this happen? In Java, by default, a piece of code is assumed to be executed by only one thread. As we saw, the compiler, Runtime, or processor could optimize things away, as long as we get the same result as with the original code. In a multi-threaded context, it's a developer's responsibility to use the appropriate mechanisms to correctly synchronize accesses to a shared state.

In this particular example, the compiler could see that 'ready' is false (default value) and it never changes, so we would end up with an infinite loop. On the other hand, if the updated value for 'ready' is visible to `ReaderThread` before 'number' is (code reordering/caching), `ReaderThread` would see that 'number' is 0 (default value).

The Happens-Before Relationship

From [here](#) (section 17.4.5):

Two actions can be ordered by a happens-before relationship. If one action happens before another, then the first is visible to and ordered before the second (for example, the write of a default value to every field of an object constructed by a thread need not happen before the beginning of that thread, as long as no read ever

This relationship is established by either:

- The `synchronized` construct (this also provides atomicity).
- The `volatile` construct.
- `Thread.start()` and `Thread.join()` methods.
- The methods of all classes in `java.util.concurrent` and its subpackages.
- etc.

To understand the importance of this relationship, let's review the concept of data race. "A data race occurs when a variable is written to by at least one thread and read by at least another thread, and the reads and writes are not ordered by a *happens-before* relationship. A correctly synchronized program is one with no data races" (from [here](#)).

More specifically, a correctly synchronized program is one whose sequentially consistent executions do not have any data races.

Volatile

`volatile` is a lightweight form of synchronization that tackles the visibility and ordering aspects. `volatile` is used as a **field** modifier. The purpose of `volatile` is to ensure that when one thread writes a value to a field, the value written is "immediately available" to any thread that subsequently reads it.

There's a common misconception about the relationship between `volatile`, on the one hand, and references and objects, on the other hand. In Java, there are *objects* and *references* to those objects. We can think of references as "pointers" to objects. All variables (except for primitives like boolean or long) contain references (object **references**).

`volatile` acts either on a primitive variable or on a reference variable. There's no relation between `volatile` and the object referred to by the (reference) variable.

Declaring a shared variable as `volatile` **ensures visibility**.

`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. "The visibility effects of `volatile` variables extend beyond the value of the 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" (from JCIP book).

In order to guarantee that the results of one action are observable to a second action, then the first must **happen before** the second.

`volatile` also **limits reordering** of accesses (accesses to the reference) by preventing the compiler and Runtime from reordering of code. The ability to perceive ordering constraints among actions is only guaranteed to actions **that share a happens-before relationship with them**.

Under the hood, `volatile` causes reads and writes to be accompanied by a special CPU instruction known as a **memory barrier**. For more low-level details, you can check [here](#) and [here](#).

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. But note that `volatile` doesn't block as `synchronized` does.

side effects of an atomic action are visible until the action is complete.

There's a correlation between `volatile` and atomic actions: for non-`volatile` 64-bit numeric (long and double) primitive variables, the JVM is free to treat a 64-bit read or write as two separate 32-bit operations. Using `volatile` on those types ensures the read and write operations are atomic (accessing the rest of the **primitive** variables and **reference** variables is atomic by default).

`volatile` accesses do not guarantee the atomicity of composite operations such as incrementing a counter. Compound operations on shared variables must be performed atomically to prevent data races and race conditions.

When to Use Volatile

We can use `volatile` variables only when all the following criteria are met (from JCIP book):

- 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
- Locking is not required for any other reason while the variable is being accessed.

Some suitable scenarios are:

- When we have a simple flag (like a completion, interruption, or status flag) or other primitive item of data that is accessed (read) by multiple threads.
- When we have a more complex immutable object that is initialized by one thread and then accessed by others: here we want to update the reference to the immutable data structure and allow other threads to reference it.

Considerations Before Using Volatile

- Code that relies on `volatile` variables for visibility of arbitrary state is more fragile and harder to understand than code that uses locking.
- The semantics of `volatile` is not strong enough to guarantee atomicity (atomic variables do provide atomic read-modify-write support and can often be used as “better `volatile` variables”).
- Synchronization can introduce thread contention, which occurs when two or more threads try to access the same resource simultaneously.

Examples

Several of the following examples were taken from [here](#). You can find a lot more information about the examples and their internal working there.

1) Incrementing a counter

```
Java
1 @NotThreadSafe
2 public class UnsafeCounter {
3     private volatile Integer counter;
4
5     public void increment() {
6         counter++;
7     }
8 }
```

Even though it appears as a single operation, the “`counter++;`” statement is not atomic. It is actually a combination of three different operations: read, update, write. So, even making counter `volatile` is not enough to avoid data races in a multi-threaded context: we may end up with **lost updates**.

```

Java
1  @ThreadSafe
2  public class Singleton {
3      private static volatile Singleton INSTANCE;
4
5      private Singleton() {
6      }
7
8      public static Singleton getInstance() {
9          if (INSTANCE != null)
10             return singleton;
11         synchronized (Singleton.class) {
12             if (INSTANCE == null)
13                 INSTANCE = new Singleton();
14             return INSTANCE;
15         }
16     }
17 }

```

In this case, not making `INSTANCE` `volatile` could lead `getInstance()` to return a non-fully initialized object, breaking the Singleton pattern. Why? The assignment of the reference variable `INSTANCE` could happen **while** the object is being initialized, exposing to other threads a (wrong) state that will end up changing, even if the object is immutable! Using `volatile`, we can ensure a **one-time safe publication** (see Pattern #2 [here](#)) of the object.

3) Visibility

```

Java
1  @NotThreadSafe
2  public class NoVisibility {
3      private static boolean ready;
4      private static int number;
5
6      private static class ReaderThread extends Thread {
7          public void run() {
8              while (!ready)
9                  Thread.yield();
10             System.out.println(number);
11         }
12     }
13
14     public static void main(String[] args) {
15         new ReaderThread().start();
16         number = 42;
17         ready = true;
18     }
19 }

```

In this **particular** case, making 'ready' `volatile` is enough to make that class thread-safe because after `main()` updates 'ready', `ReaderThread` will see the new value for both 'ready' and 'number' (*happens-before* relationship). But these kinds of constructs are fragile and we should avoid using them: let's consider for example the case someone flips the two assignment statements.

Some *thread-safe* alternatives: `synchronized`, `Lock`.

4) Immutable collaborator

```

Java
1  // Immutable Helper
2  public final class Helper {
3      private final int n;
4
5      public Helper(int n) {
6          this.n = n;
7      }
8      // ...
9  }
10
11 // Mutable Foo
12 @NotThreadSafe

```

```

18 }
19
20 public void setHelper(int num) {
21     helper = new Helper(num);
22 }
23 }

```

Even though (mutable) *Foo* contains fields referring to only immutable objects (*Helper*), *Foo* is **not** thread-safe. Mutable objects may not be fully constructed when their references are made visible. The reason is that, while the shared object (*Helper*) is immutable, the reference used to access it is itself shared and mutable. In the example, that means a separate thread could observe a stale reference in the *helper* field of the *Foo* object. One solution to that issue could be making the *helper* field of *Foo* `volatile`; `volatile` guarantees an object is constructed properly before its reference is made visible.

Some *thread-safe* alternatives: `synchronized`, [AtomicReference](#).

5) Compound operation

```

Java
1 @NotThreadSafe
2 final class Flag {
3     private volatile boolean flag = true;
4
5     public void toggle() { // Unsafe
6         flag ^= true;
7     }
8
9     public boolean getFlag() { // Safe
10        return flag;
11    }
12 }

```

As in the first example we saw, here we have a composite operation.

`volatile` is not enough in this case.

Some *thread-safe* alternatives: `synchronized`, `volatile` + `synchronized`, [ReadWriteLock](#), [AtomicBoolean](#).

6) Arrays

```

Java
1 @NotThreadSafe
2 final class Foo {
3     private volatile int[] arr = new int[20];
4
5     public int getFirst() {
6         return arr[0];
7     }
8
9     public void setFirst(int n) {
10        arr[0] = n;
11    }
12    // ...
13 }

```

Here it is the array **reference** which is `volatile`, not the array itself.

Some *thread-safe* alternatives: `synchronized`, [AtomicIntegerArray](#).

Conclusion

`volatile` variables are a weaker form of synchronization than locking, which in some cases are a good alternative to locking. If we follow the conditions for using `volatile` we discussed before, maybe we can achieve thread-safety and better performance than with locking. However, code using `volatile` is often more fragile than code using locking.

As `volatile` doesn't block (unlike `synchronized`), there is no possibility for deadlocks to occur. But, just as other synchronization mechanisms, the use of `volatile` implies some performance penalties.

On highly-contended contexts, using `volatile` could be detrimental for performance.

Note that `volatile` only applies to **fields**. It would not make sense to apply it to method parameters or local variables given all they are local/private to the executing thread.

`volatile` is related to object references, and not to operations on objects themselves.

Using simple atomic variable access is more efficient than accessing these variables through `synchronized` code, but requires more care by the programmer to avoid memory consistency errors. Whether the extra effort is worthwhile depends on the size and complexity of the application.

Volatile FAQs

-Can we use `volatile` without using `synchronized`?

Yes.

-Can we use `volatile` together with `synchronized`?

Yes.

-Should we use `volatile` together with `synchronized`?

It depends.

-Does `volatile` apply to an object?

No, it applies to an object *reference* or to a primitive type.

-Does `volatile` tackle the atomicity aspect?

No, but there is a special case for 64-bit primitives where it does.

-Which is the difference between `volatile` and `synchronized`?

Besides state visibility, `synchronized` keyword provides atomicity (through mutual exclusion) over a block of code. But the changes inside that block are visible to other threads only after the exit of that `synchronized` block.

-Does `volatile` imply thread-safety?

No at all: `volatile`, `synchronized`, etc. are just synchronization mechanisms. As developers we must use them as appropriate, and those mechanisms depend on the case at hand.

-Does `volatile` improve thread performance?

The opposite. The use of `volatile` implies some performance penalties.

Bibliography

- <https://jcip.net/>
- <https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.3.1.4>
- <https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.3.1.4>
- <https://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html#jls-17.4.2>

- https://www.javamex.com/tutorials/synchronization_volatile.shtml
- <https://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html>
- <https://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html>
- <https://www.ibm.com/developerworks/java/library/j-jtp06197/index.html>
- <https://dzone.com/articles/demystifying-volatile-and-synchronized-again>
- <https://www.logicbig.com/tutorials/core-java-tutorial/java-multi-threading/java-memory-model.html>
- <https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html>
- <https://docs.oracle.com/javase/tutorial/essential/concurrency/locksinc.html>
- <https://docs.oracle.com/javase/tutorial/essential/concurrency/newlocks.html>
- <https://docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html>
- <https://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html>
- <https://docs.oracle.com/javase/tutorial/essential/concurrency/locksinc.html>
- <https://docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html>
- <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html#MemoryVisibility>
- <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html>
- https://en.wikipedia.org/wiki/Memory_barrier
- <https://devepaper.com/memory-barrier-and-its-application-in-jvm/>
- <https://devepaper.com/memory-barrier-and-its-application-in-jvm-2/>
- https://www.infoq.com/articles/memory_barriers_jvm_concurrency/
- <https://dzone.com/articles/memory-barriersfences>

Topics: CONCURRENCY AND LOCKING, VOLATILE FIELD, PROGRAMMING & DESIGN, SYNCHRONIZATION, SYNCHRONIZED, JAVA

Opinions expressed by DZone contributors are their own.

Popular on DZone

- [UnifiedFlow - Git Branching Strategy](#)
- [9 Best Free and Open Source Tools for Reporting](#)
- [How To Use DORA Engineering Metrics To Improve Your Dev Team](#)
- [Lessons From The Trenches: Cloud Modern Engineering](#)

Java Partner Resources

Careers

Sitemap

ADVERTISE

Advertise with DZone

+1 (919) 238-7100

CONTRIBUTE ON DZONE

Article Submission Guidelines

MVB Program

Become a Contributor

Visit the Writers' Zone

LEGAL

Terms of Service

Privacy Policy

CONTACT US

600 Park Offices Drive


Suite 300

Durham, NC 27709

support@dzone.com

+1 (919) 678-0300



DZone.com is powered by  AnswerHub logo