Volatile Vs Static in java

Is it correct to say that *static* means one copy of the value for all objects and *volatile* means one copy of the value for all threads?

Anyway a *static* variable value is also going to be one value for all threads, then why should we go for *volatile*?

java multithreading concurrency static volatile

edited Oct 30 '11 at 5:36

leppie
70.4k 8 112 209

asked Mar 11 '10 at 9:01

Jothi
2,503 9 35 72

13 please reconsider your accepted answer, since second answer from @stivlo is more correct – saberduck Jun 14 '12 at 11:03

5 Answers

Difference Between Static and Volatile :

Static Variable: If two Threads(suppose `t1` and `t2`) are accessing the same object and updating a variable which is declared as static then it means `t1` and `t2` can make their own local copy of the same object(including static variables) in their respective cache, so update made by `t1` to the static variable in its local cache wont reflect in the static variable for `t2` cache .

Static variables are used in the **Object Context** where update made by one object would reflect in all the other objects of the same class **but not in the Thread context** where update of one thread to the static variable will reflect the changes immediately to all the threads (in their local cache).

Volatile variable: If two Threads(suppose `t1` and `t2`) are accessing the same object and updating a variable which is declared as volatile then it means `t1` and `t2` can make their own local cache of the Object **except the variable which is declared as a volatile** . So the volatile variable will have only one main copy which will be updated by different threads and update made by one thread to the volatile variable will immediately reflect to the other Thread.

So the volatile variable is used in the Thread context .

edited Aug 25 '14 at 16:35

Roman Kagan
4,172 20 52 97

answered Aug 30 '12 at 9:38

Som
1,623 2 8 17

Declaring a **static** variable in Java, means that there will be only one copy, no matter how many objects of the class are created. The variable will be accessible even with no Objects created at all. However, threads may have locally cached values of it.

When a variable is **volatile** and not **static**, there will be one variable for each Object. So, on the surface it seems there is no difference from a normal variable but totally different from *static*. However, even with Object fields, a thread may cache a variable value locally.

This means that if two threads update a variable of the same Object concurrently, and the variable is not declared volatile, there could be a case in which one of the thread has in cache an old value.

Even if you access a *static* value through multiple threads, each thread can have its local cached copy! To avoid this you can declare the variable as **static volatile** and this will force the thread to read each time the global value.

However, *volatile* is not a substitute for proper synchronisation! For instance:

```
private static volatile int counter = 0;

private void concurrentMethodWrong() {
     counter = counter + 5;
     //do something
     counter = counter - 5;
}
```

Executing *concurrentMethodWrong* concurrently many times may lead to a final value of counter different from zero! To solve the problem, you have to implement a lock:

```
private static final Object counterLock = new Object();

private static volatile int counter = 0;

private void concurrentMethodRight() {
     synchronized (counterLock) {
         counter = counter + 5;
     }
     //do something
     synchronized (counterLock) {
         counter = counter - 5;
     }
}
```

Or use the `AtomicInteger` class.

edited Jan 27 '14 at 22:19
Mike Samuel
59.1k 9 100 143

answered Oct 30 '11 at 5:29
stivlo
32.8k 19 77 135

4 The volatile modifier guarantees that any thread that reads a field will see the most recently written value, so it's necessary if the variable is shared among multiple thread and you need this feature, it depends on your use case. – stivlo Nov 11 '12 at 14:41

4 What is the cache when you say "locally cached"? CPU cache, some kind of JVM cache? – mert inan Dec 17 '12 at 21:15

3 @mertinan yes, the variable can be in a cache nearer to the processor or core. See cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html for more details. – stivlo Dec 17 '12 at 22:08

11 'volatile' does ***not*** imply 'one variable per object'. Absence of 'static' does that. -1 for failing to clear up this elementary misconception on the part of the OP. – EJP Aug 25 '13 at 6:19

13 @EJP I thought that the sentence "Declaring a variable as volatile, there will be one variable for each Object. So on the surface it seems there is no difference from a normal variable" was explaining that, I've added *and not static*, feel free to edit the article and improve the wording to make that clearer. – stivlo Aug 25 '13 at 12:47

---

i think static and volatile have no relation at all.i suggest you read java tutorial to understand Atomic Access,and why use atomic access,understand what is interleaved,you will find answer.

edited Sep 17 '13 at 14:24
Sikorski
723 1 7 31

answered Aug 25 '13 at 6:06
wangyiran
520 6 16

Declaring a variable volatile guarantees that on different architectures the jvm is not going to cache that variable thread locally

answered Feb 20 '13 at 21:44
Vlatko Dimov
1

If we declare a variable as static, there will be only one copy of the variable. So, whenever different threads access that variable, there will be only one final value for the variable(since there is only one memory location allocated for the variable).

If a variable is declared as volatile, all threads will have their own copy of the variable but the value is taken from the main memory.So, the value of the variable in all the threads will be the same.

So, in both cases, the main point is that the value of the variable is same across all threads.

answered Apr 20 '11 at 6:04

Jitendra Nalwaya
19 2

8 *If a variable is declared as volatile, all threads will have their own copy of the variable but the value is taken from the main memory.* => **right.** *So, the value of the variable in all the threads will be the same.* => **wrong, each thread will use the same value for the same Object, but each Object will have its own copy.** – stivlo Oct 30 '11 at 5:59