Do you ever use the volatile keyword in Java?

At work today, I came across the `volatile` keyword in Java. Not being very familiar with it, I found this explanation:

> Java theory and practice: Managing volatility

Given the detail in which that article explains the keyword in question, do you ever use it or could you ever see a case in which you could use this keyword in the correct manner?

java multithreading concurrency keyword volatile

edited Feb 7 at 11:54

Daniel Werner
900 12 24

asked Sep 20 '08 at 0:41

Richard
3,601 3 21 29

protected by Aniket Thakur Jun 23 '15 at 8:33

This question is protected to prevent "thanks!", "me too!", or spam answers by new users. To answer it, you must have earned at least 10 reputation on this site (the association bonus does not count).

19 Answers

`volatile` has semantics for memory visibility. Basically, the value of a `volatile` field becomes visible to all readers (other threads in particular) after a write operation completes on it. Without `volatile`, readers could see some non-updated value.

To answer your question: Yes, I use a `volatile` variable to control whether some code continues a loop. The loop tests the `volatile` value and continues if it is `true`. The condition can be set to `false` by calling a "stop" method. The loop sees `false` and terminates when it tests the value after the stop method completes execution.

The book "Java Concurrency in Practice," which I highly recommend, gives a good explanation of `volatile`. This book is written by the same person who wrote the IBM article that is referenced in the question (in fact, he cites his book at the bottom of that article). My use of `volatile` is what his article calls the "pattern 1 status flag."

If you want to learn more about how `volatile` works under the hood, read up on the Java memory model. If you want to go beyond that level, check out a good computer architecture book like Hennessy & Patterson and read about cache coherence and cache consistency.

edited May 11 at 7:44

Basil Bourque
70.4k 17 245 338

answered Sep 20 '08 at 2:09

Greg Mattes
17.7k 13 58 99

3 I concur - I've used volatile for the exact same reason - tracking when to end a loop in a multi-threaded application. – Darren Greaves Jun 3 '09 at 10:13

73 This answer is correct, but incomplete. It omits an important property of `volatile` that came with the new Java Memory Model defined in JSR 133: that when a thread reads a `volatile` variable it sees not only the value last written to it by some other thread, but also all other writes to other variables that were visible in that other thread at the time of the `volatile` write. See this answer and this reference. – Adam Zalcman Jul 18 '13 at 12:32

23 For beginners, I'd request you to demonstrate with some code (please?) – Astrobleme Feb 22 '14 at 6:43

3 The article linked in the question has code examples. – Greg Mattes Feb 22 '14 at 23:37

1 @fefrei: “immediately” is a colloquial term. Of course, that can’t be guaranteed when neither, execution timing nor thread scheduling algorithms, are actually specified. The only way for a program to find out whether a volatile read is subsequent to a particular volatile write, is by checking whether the seen value is the expected written one. – Holger Jul 5 at 8:06

|

“… the volatile modifier guarantees that any thread that reads a field will see the most recently

written value." **- Josh Bloch**

If you are thinking about using `volatile` , read up on the package `java.util.concurrent` which deals with atomic behaviour.

The Wikipedia post on a Singleton Pattern shows volatile in use.

edited May 11 at 7:43

community wiki
8 revs, 2 users 96%
Andrew Turner

1 The quote really brings it to the point. – Zarathustra Mar 28 '14 at 8:19

9 Why is there both `volatile` and `synchronized` keywords? – Patrick May 9 '15 at 2:32

2 The Wikipedia article on a Singleton Pattern has changed a lot since and doesn't feature said `volatile` example any longer. It can be found in an archived version. – bskp Sep 23 '16 at 12:26

Important point about volatile:

1. Synchronization in Java is possible by using Java keywords `synchronized` and `volatile` .
2. In Java, we can not have `synchronized` variable. Using `synchronized` keyword with variable is illegal and will result in compilation error. Instead of `synchronized` variable in Java, you can have java `volatile` variable, which will instruct JVM threads to read value of `volatile` variable from main memory and don't cache it locally.
3. If a variable is not shared between multiple threads no need to use volatile keyword with that variable.

source

Example usage of volatile:

```
public class Singleton {
    private static volatile Singleton _instance; // volatile variable
    public static Singleton getInstance() {
        if (_instance == null) {
            synchronized (Singleton.class) {
                if (_instance == null)
                    _instance = new Singleton();
            }
        }
        return _instance;
    }
}
```

We are creating instance lazily at the time of first request comes.

If we do not make the `_instance` variable `volatile` than the Thread which is creating instance of `Singleton` is not able to communicate other thread, that instance has been created until it comes out of the Singleton block, so if Thread A is creating Singleton instance and just after creation lost the CPU, all other thread will not be able to see value of `_instance` as not null and they will believe its still null.

Why? because reader threads are not doing any locking and until writer thread comes out of synchronized block, memory will not be synchronized and value of `_instance` will not be updated in main memory. With Volatile keyword in Java this is handled by Java himself and such updates will be visible by all reader threads.

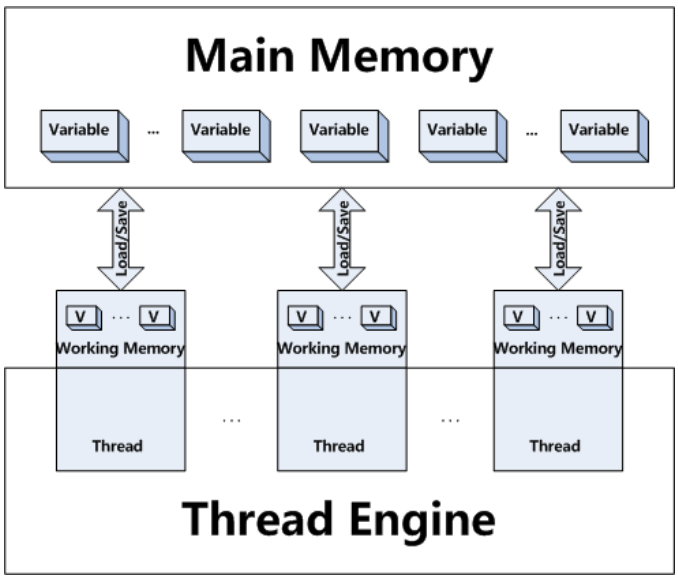
> **Conclusion**: `volatile` keyword is also used to communicate content of memory between threads.

Example usage of without volatile:

```
public class Singleton{
    private static Singleton _instance;   //without volatile variable
    public static Singleton getInstance(){
          if(_instance == null){
              synchronized(Singleton.class){
               if(_instance == null) _instance = new Singleton();
      }
     }
    return _instance;
    }
```

The code above is not thread-safe. Although it checks the value of instance once again within the synchronized block (for performance reasons), the JIT compiler can rearrange the

bytecode in a way that the reference to instance is set before the constructor has finished its execution. This means the method getInstance() returns an object that may not have been initialized completely. To make the code thread-safe, the keyword volatile can be used since Java 5 for the instance variable. Variables that are marked as volatile get only visible to other threads once the constructor of the object has finished its execution completely.
Source



volatile usage in java The fail-fast iterators are *typically* implemented using a `volatile` counter on the list object.

- When the list is updated, the counter is incremented.
- When an `Iterator` is created, the current value of the counter is embedded in the `Iterator` object.
- When an `Iterator` operation is performed, the method compares the two counter values and throws a `ConcurrentModificationException` if they are different.

The implementation of fail-safe iterators is typically light-weight. They typically rely on properties of the specific list implementation's data structures. There is no general pattern.

edited Oct 26 '16 at 16:23

answered Dec 18 '15 at 21:52
Premraj
16.4k 7 99 88

"The fail-fast iterators are typically implemented using a volatile counter" - no longer the case, too costly: bugs.java.com/bugdatabase/view_bug.do?bug_id=6625725 – Vsevolod Golovanov May 24 at 17:15

are the double checking for _instance safe? i thought they are not safe even with volatile – Dexters Nov 25 at 21:18

When is `volatile` enough?

If two threads are both reading and writing to a shared variable, then using the `volatile` keyword for that is not enough. You need to use synchronization in that case to guarantee that the reading and writing of the variable is atomic.

But in case one thread reads and writes the value of a `volatile` variable, and other threads only read the variable, then the reading threads are guaranteed to see the latest value written to the `volatile` variable. Without making the variable `volatile` , this would not be guaranteed.

Performance considerations of using `volatile` :

Reading and writing of `volatile` variables causes the variable to be read or written to main memory. Reading from and writing to main memory is more expensive than accessing the CPU cache. Accessing `volatile` variables also prevent instruction reordering which is a normal performance enhancement technique. Thus, you should only use `volatile` variables when you really need to enforce visibility of variables.

edited Jan 30 '15 at 13:15
kinbiko
964 1 13 29

answered Jan 30 '15 at 11:36
sujith s
953 9 16

thanks @kinbiko to make it more understandable.. – sujith s Apr 6 '15 at 5:49

Every modern CPU has consistent caches, and values are written back to main memory when necessary. – curiousguy Aug 10 '15 at 23:00

"You need to use synchronization" does simply enclosing the scope with `synchronize` keyword would work? – mr5 Nov 11 '15 at 3:48

this answer seem like cite from this link or the other way around? – Jasonw Jul 31 at 7:49

---

volatile is very useful to stop threads.

Not that you should be writing your own threads, Java 1.6 has a lot of nice thread pools. But if you are sure you need a thread, you'll need to know how to stop it.

The pattern I use for threads is:

```
public class Foo extends Thread {
  private volatile boolean close = false;
  public void run() {
    while(!close) {
      // do work
    }
  }
  public void close() {
    close = true;
    // interrupt here if needed
  }
}
```

Notice how there's no need for synchronization

answered Sep 24 '08 at 22:29

Pyrolistical
17.4k 16 69 96

2 I wonder why that is even necessary. Isn't that only necessary if other threads have to react on the status change of this thread in such a way that the threads synchronization is at danger? – Jori Jun 6 '13 at 7:27

16 @Jori, you need volatile because the thread reading close in the while loop is different from the one that calls close(). Without volatile, the thread running the loop may never see the change to close. – Pyrolistical Jun 6 '13 at 18:48

would you say there is an advantage between stopping a thread like that or using Thread#interrupt() and Thread#isInterrupted() methods ? – Ricardo Belchior Sep 3 '15 at 14:33

1 @Pyrolistical - Have you observed the thread *never* seeing the change in practice? Or can you extend the example to reliably trigger that issue? I'm curious because I know I've used (and seen others using) code that's basically identical to the example but without the `volatile` keyword, and it always seems to work fine. – aroth Feb 21 '16 at 13:28

1 @aroth: with today's JVMs, you can observe that in practice, even with the simplest examples, however, you can't *reliably* reproduce this behavior. With more complex applications, you sometimes have other actions with memory visibility guarantees within you code which make it happen to work, which is especially dangerous as you don't know why it works and a simple, apparently unrelated change in your code can break your application… – Holger Oct 18 '16 at 10:39

|

One common example for using `volatile` is to use a `volatile boolean` variable as a flag to terminate a thread. If you've started a thread, and you want to be able to safely interrupt it from a different thread, you can have the thread periodically check a flag. To stop it, set the flag to true. By making the flag `volatile` , you can ensure that the thread that is checking it will see it has been set the next time it checks it without having to even use a `synchronized` block.

answered Sep 20 '08 at 4:00

Dave L.
32k 8 47 55

No one has mentioned the treatment of read and write operation for long and double variable type. Reads and writes are atomic operations for reference variables and for most primitive variables, except for long and double variable types, which must use the volatile keyword to be atomic operations. @link

answered Feb 11 '15 at 14:51

Donatello Boccaforno


111 1 4

To make it even more clearer, there is NO NEED to set a boolean volatile, because the read and write of a boolean IS ALREADY atomic. – Kai Wang Apr 3 at 14:10

@KaiWang you don't need to use volatile on booleans for atomicity purposes. But you certainly might for visibility reasons. Is that what you meant to say? – SusanW Jun 16 at 12:27

Yes, volatile must be used whenever you want a mutable variable to be accessed by multiple threads. It is not very common usecase because typically you need to perform more than a single atomic operation (e.g. check the variable state before modifying it), in which case you would use a synchronized block instead.

answered Sep 20 '08 at 4:26


ykaganovich
11.1k 4 42 85

IMO two important scenarios other than stopping thread in which volatile keyword is used are

1. Double-checked locking mechanism. Used often in Singleton design pattern. In this the `singleton object needs to be declared volatile`.
2. Spurious Wakeups. Thread may sometimes wake up from wait call even if no notify call has been issued. This behavior is called supurious wakeup. This can be countered by using a conditional variable(boolean flag). Put the wait() call in a while loop as long as the flag is true. So if thread wakes up from wait call due to any reasons other than notify/notifyall then it encounters flag is still true and hence calls wait again. Prior to calling notify set this flag to true. In this case the `boolean flag is declared as volatile`.

answered Feb 25 '14 at 9:17


Aniket Thakur
34.5k 19 158 179

`volatile` only guarantees that all threads, even themselves, are incrementing. For example: a counter sees the same face of the variable at the same time. It is not used instead of synchronized or atomic or other stuff, it completely makes the reads synchronized. Please do not compare it with other java keywords. As the example shows below volatile variable operations are also atomic they fail or succeed at once.

```
package io.netty.example.telnet;

import java.util.ArrayList;
import java.util.List;

public class Main {

    public static volatile  int a = 0;
    public static void main(String args[]) throws InterruptedException{

        List<Thread> list = new  ArrayList<Thread>();
        for(int i = 0 ; i<11 ;i++){
            list.add(new Pojo());
        }

        for (Thread thread : list) {
            thread.start();
        }

        Thread.sleep(20000);
        System.out.println(a);
    }
}
class Pojo extends Thread{
    int a = 10001;
    public void run() {
        while(a-->0){
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            Main.a++;
            System.out.println("a = "+Main.a);
        }
    }
}
```

Even you put volatile or not results will always differ. But if you use AtomicInteger as below results will be always same. This is same with synchronized also.

```
package io.netty.example.telnet;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.atomic.AtomicInteger;

public class Main {

    public static volatile  AtomicInteger a = new AtomicInteger(0);
    public static void main(String args[]) throws InterruptedException{

        List<Thread> list = new  ArrayList<Thread>();
        for(int i = 0 ; i<11 ;i++){
            list.add(new Pojo());
        }

        for (Thread thread : list) {
            thread.start();
        }

        Thread.sleep(20000);
        System.out.println(a.get());

    }
}
class Pojo extends Thread{
    int a = 10001;
    public void run() {
        while(a-->0){
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            Main.a.incrementAndGet();
            System.out.println("a = "+Main.a);
        }
    }
}
```

edited Aug 18 '16 at 15:06
bwegs
3,552 2 19 26

answered Mar 2 '13 at 21:07
fatih tekin
591 6 16

You'll need to use 'volatile' keyword, or 'synchronized' and any other concurrency control tools and techniques you might have at your disposal if you are developing a multithreaded application. Example of such application is desktop apps.

If you are developing an application that would be deployed to application server (Tomcat, JBoss AS, Glassfish, etc) you don't have to handle concurrency control yourself as it already addressed by the application server. In fact, if I remembered correctly the Java EE standard prohibit any concurrency control in servlets and EJBs, since it is part of the 'infrastructure' layer which you supposed to be freed from handling it. You only do concurrency control in such app if you're implementing singleton objects. This even already addressed if you knit your components using frameworkd like Spring.

So, in most cases of Java development where the application is a web application and using IoC framework like Spring or EJB, you wouldn't need to use 'volatile'.

edited Jul 10 '12 at 18:02
Arjan Tijms
32.3k 8 86 125

answered Sep 20 '08 at 7:14
Rudi Adianto
222 1 6

Absolutely, yes. (And not just in Java, but also in C#.) There are times when you need to get or set a value that is guaranteed to be an atomic operation on your given platform, an int or boolean, for example, but do not require the overhead of thread locking. The volatile keyword allows you to ensure that when you read the value that you get the *current* value and not a cached value that was just made obsolete by a write on another thread.

edited Jan 6 '12 at 14:28

answered Sep 20 '08 at 1:59
dgvid
19.2k 4 31 52

Corrected. Thanks! – dgvid Jan 6 '12 at 14:29

Yes, I use it quite a lot - it can be very useful for multi-threaded code. The article you pointed to is a good one. Though there are two important things to bear in mind:

1. You should only use volatile if you completely understand what it does and how it differs to synchronized. In many situations volatile appears, on the surface, to be a simpler more performant alternative to synchronized, when often a better understanding of volatile would make clear that synchronized is the only option that would work.
2. volatile doesn't actually work in a lot of older JVMs, although synchronized does. I remember seeing a document that referenced the various levels of support in different JVMs but unfortunately I can't find it now. Definitely look into it if you're using Java pre 1.5 or if you don't have control over the JVMs that your program will be running on.

answered Sep 20 '08 at 11:07

MB.
4,807 5 33 38

Every thread accessing a volatile field will read its current value before continuing, instead of (potentially) using a cached value.

Only member variable can be volatile or transient.

answered Aug 11 '14 at 16:27

tstuber
138 1 9

From oracle documentation page, the need for volatile variable arises to fix memory consistency issues:

> Using volatile variables reduces the risk of memory consistency errors, because any write to a volatile variable establishes a happens-before relationship with subsequent reads of that same variable.

This means that changes to a `volatile` variable are always visible to other threads. It also means that when a thread reads a volatile variable, it sees not just the latest change to the `volatile`, but also the side effects of the code that led up the change.

As explained in `Peter Parker` answer, in absence of `volatile` modifier, each thread's stack may have their own copy of variable. By making the variable as `volatile`, memory consistency issues have been fixed.

Have a look at jenkov tutorial page for better understanding.

Have a look at related SE question for some more details on volatile & use cases to use volatile:

Difference between volatile and synchronized in Java

One practical use case:

You have many threads, which need to print current time in a particular format for example : `java.text.SimpleDateFormat("HH-mm-ss")`. Yon can have one class, which converts current time into `SimpleDateFormat` and updated the variable for every one second. All other threads can simply use this volatile variable to print current time in log files.

edited May 23 at 11:55

Community ♦
1 1

answered May 9 '16 at 13:24

Ravindra babu
22k 5 102 106

The volatile key when used with a variable, will make sure that threads reading this variable will see the same value . Now if you have multiple threads reading and writing to a variable, making the variable volatile will not be enough and data will be corrupted . Image threads have read the same value but each one has done some chages (say incremented a counter) , when writing back to the memory, data integrity is violated . That is why it is necessary to make the varible synchronized (diffrent ways are possible)

If the changes are done by 1 thread and the others need just to read this value, the volatile will be suitable.

answered Dec 24 '15 at 16:41

Java Main
675 5 14

There are two different uses of volatile keyword.

1. Prevents JVM from reading values from register (assume as cache), and forces its value to be read from memory.
2. Reduces the risk of memory in-consistency errors.

> Prevents JVM from reading values in register, and forces its value to be read from memory.

A *busy flag* is used to prevent a thread from continuing while the device is busy and the flag is not protected by a lock:

```
while (busy) {
    /* do something else */
}
```

The testing thread will continue when another thread turns off the *busy flag*:

```
busy = 0;
```

However, since busy is accessed frequently in the testing thread, the JVM may optimize the test by placing the value of busy in a register, then test the contents of the register without reading the value of busy in memory before every test. The testing thread would never see busy change and the other thread would only change the value of busy in memory, resulting in deadlock. Declaring the *busy flag* as volatile forces its value to be read before each test.

> Reduces the risk of memory consistency errors.

Using volatile variables reduces the risk of **memory consistency errors**, because any write to a volatile variable establishes a *"happens-before"* relationship with subsequent reads of that same variable. This means that changes to a volatile variable are always visible to other threads.

The technique of reading, writing without memory consistency errors is called **atomic action**.

An atomic action is one that effectively happens all at once. An atomic action cannot stop in the middle: it either happens completely, or it doesn't happen at all. No side effects of an atomic action are visible until the action is complete.

Below are actions you can specify that are atomic:

- Reads and writes are atomic for reference variables and for most primitive variables (all types except long and double).
- Reads and writes are atomic for all variables declared **volatile** (including long and double variables).

Cheers!

answered Mar 2 at 8:42

dheeran
1,464 13 10

Volatile Variables are light-weight synchronization. When visibility of latest data among all threads is requirement and atomicity can be compromised , in such situations Volatile Variables must be preferred. Read on volatile variables always return most recent write done by any thread since they are neither cached in registers nor in caches where other processors can not see. Volatile is Lock-Free. I use volatile, when scenario meets criteria as mentioned above. For more Lock free and Lock based strategies.

answered Jul 8 '16 at 11:08

Neha Vari
348 1 10

A **Volatile variable** is modified asynchronously by concurrently running threads in a Java application. It is not allowed to have a local copy of a variable that is different from the value currently held in "main" memory. Effectively, a variable declared volatile must have its data synchronized across all threads, so that whenever you access or update the variable in any thread, all other threads immediately see the same value. Of course, it is likely that volatile variables have a higher access and update overhead than "plain" variables, since the reason threads can have their own copy of data is for better efficiency.

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.

for reference, refer this http://techno-terminal.blogspot.in/2015/11/what-are-volatile-variables.html

answered Nov 17 '15 at 9:36

satish
583 4 3