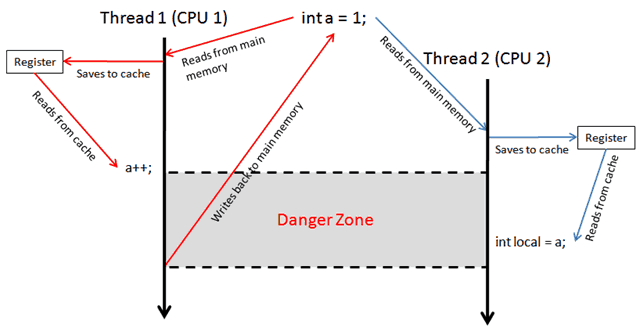
[volatile modifier on MSDN](http://msdn.microsoft.com/en-us/library/x13ttww7.aspx)

* *Fields that are declared volatile are not subject to compiler optimizations that assume access by a single thread. This ensures that the most up-to-date value is present in the field at all times.*
* What the volatile modifier gives you is protection against concurrency issues arising from multiple threads executing on multiple CPUs **caching** data and **re-ordering** instructions.  BUT THIS KEYWORD HELPS ONLY IN PROTECTING IF THE EXECUTION IS DONE IN 1 INSTRUCTION AGAINST THIS VARIABLE. YOU CAN SEE MORE IN THE BELOW EXAMPLE AT THE LAST.
* **Data Caching**
  + A processor might cache a piece of data from the main memory and use the cached data in the execution of a thread, modify it, and only update the main memory at a later time. During which time another thread running concurrently on another CPU might have read the same bit of data from main memory and used the outdated version of the data. ([Jon Skeet’s article on Volatility, Atomicity and Interlocking](http://www.yoda.arachsys.com/csharp/threads/volatility.shtml))
  + Marking a field as volatile would make sure that it is not cached during the execution of a thread.



* **Compiler Optimization (Re-Ordering INSTRUCTIONS)**
  + The .NET compiler is allowed to alter the order of reads and writes in any way which does not change the meaning of the original program.
  + For example, as pointed out in Dr Dobb’s articled (see reference section) it’s legal for the compiler to transform:

*a = 1;  // A // ~~a = 1;~~   – eliminate line A*



*a = 2;  // B a = 2;  // B*

**This is legal because the semantic of the original program is preserved.**

*a = 1;          // C – write to a a = 1;          // C – write to a* 

*local = a;  // D – read from a local = 1; // D’ – apply “constant propagation”*

* it’s easy to see how this can be a problem in a multi-threaded environment where another thread is writing to a concurrently.
* Though the volatile modifier might not be the all-conquering solution to concurrency, but this can be the first step.
* The best way is
* where you’re modifying the value of a field based on what it is right now, which requires you to lock around it. (and everywhere else you’re using the same field!)
* BUT If all you want to do is do a check against a field and then perform an action which does not require using the field again then you can get away with just making it volatile:

private volatile int \_counter = 0;

if (\_counter < 10) // guaranteed latest value of \_counter

{

    // you're safe so long you don't reference \_counter again in here, that

    // goes to any method you call from here too!

    Console.WriteLine("We're still not there yet!");

}