volatile Keyword in Java

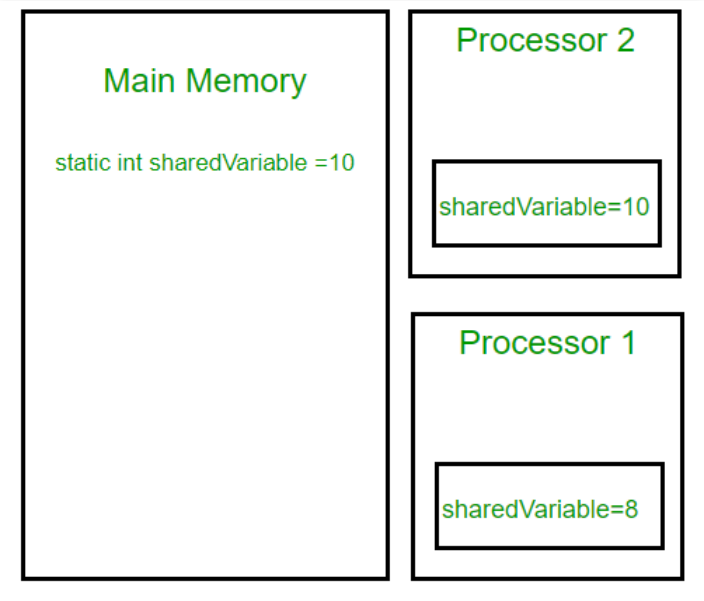
Difficulty Level : Medium Last Updated : 23 Jan, 2022

Using volatile is yet another way (like synchronized, atomic wrapper) of making class thread-safe. Thread-safe means that a method or class instance can be used by multiple threads at the same time without any problem. Consider the below example.

```
class SharedObj
{
   // Changes made to sharedVar in one thread
   // may not immediately reflect in other thread
   static int sharedVar = 6;
}
```

Suppose that two threads are working on **SharedObj**. If two threads run on different processors each thread may have its own local copy of **sharedVariable**. If one thread modifies its value the change might not reflect in the original one in the main memory instantly. This depends on the write policy of cache. Now the other thread is not aware of the modified value which leads to data inconsistency.

The below diagram shows that if two threads are run on different processors, then the value of **sharedVariable** may be different in different threads.



Note that writing of normal variables without any synchronization actions, might not be visible to any reading thread (this behavior is called sequential consistency). Although most modern hardware provides good cache coherence, therefore, most probably the

changes in one cache are reflected in another but it's not a good practice to rely on hardware to 'fix' a faulty application.

```
class SharedObj
{
   // volatile keyword here makes sure that
   // the changes made in one thread are
   // immediately reflect in other thread
   static volatile int sharedVar = 6;
}
```

Note that volatile should not be confused with the static modifier. static variables are class members that are shared among all objects. There is only one copy of them in the main memory.

volatile vs synchronized: Before we move on let's take a look at two important features of locks and synchronization.

1. **Mutual Exclusion:** It means that only one thread or process can execute a block of code (critical section) at a time.
2. **Visibility**: It means that changes made by one thread to shared data are visible to other threads.

Java's synchronized keyword guarantees both mutual exclusion and visibility. If we make the blocks of threads that modify the value of the shared variable synchronized only one thread can enter the block and changes made by it will be reflected in the main memory. All other threads trying to enter the block at the same time will be blocked and put to sleep.

In some cases, we may only desire visibility and not atomicity. The use of synchronized in such a situation is overkill and may cause scalability problems. Here volatile comes to the rescue. Volatile variables have the visibility features of synchronized but not the atomicity features. The values of the volatile variable will never be cached and all writes and reads will be done to and from the main memory. However, the use of volatile is limited to a very restricted set of cases as most of the times atomicity is desired. For example, a simple increment statement such as x = x + 1; or x++ seems to be a single operation but is really a compound read-modify-write sequence of operations that must execute atomically.

```
// Java Program to demonstrate the
// use of Volatile Keyword in Java

public class VolatileTest {
    private static final Logger LOGGER
        = MyLoggerFactory.getSimplestLogger();
    private static volatile int MY_INT = 0;

    public static void main(String[] args)
    {
        new ChangeListener().start();
        new ChangeMaker().start();
    }

    static class ChangeListener extends Thread {
        @Override public void run()
        {
            int local_value = MY_INT;
            while (local_value < 5) {
                if (local_value != MY_INT) {
                    LOGGER.log(
                        Level.INFO,
                        "Got Change for MY_INT : {0}",
                        MY_INT);
                    local_value = MY_INT;
                }
            }
        }
    }

    static class ChangeMaker extends Thread {
        @Override public void run()
        {
            int local_value = MY_INT;
            while (MY_INT < 5) {
                LOGGER.log(Level.INFO,
                           "Incrementing MY_INT to {0}",
                           local_value + 1);
                MY_INT = ++local_value;
                try {
                    Thread.sleep(500);
                }
                catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Output (With the Volatile Keyword):

```
Incrementing MY_INT to 1
Got Change for MY_INT : 1
Incrementing MY_INT to 2
Got Change for MY_INT : 2
Incrementing MY_INT to 3
Got Change for MY_INT : 3
Incrementing MY_INT to 4
Got Change for MY_INT : 4
Incrementing MY_INT to 5
Got Change for MY_INT : 5
```

Output (Without the Volatile Keyword)

```
Incrementing MY_INT to 1
Incrementing MY_INT to 2
Incrementing MY_INT to 3
Incrementing MY_INT to 4
Incrementing MY_INT to 5
```

volatile in Java vs C/C++:
Volatile in Java is different from the "volatile" qualifier in C/C++. For Java, "volatile" tells the compiler that the value of a variable must never be cached as its value may change outside of the scope of the program itself. In C/C++, "volatile" is needed when developing embedded systems or device drivers, where you need to read or write a memory-mapped hardware device. The contents of a particular device register could change at any time, so you need the "volatile" keyword to ensure that such accesses aren't optimized away by the compiler.

This article is contributed by **Sulabh Kumar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above