**Atomic Access & Volatile**

In programming, 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.

We have already seen that an increment expression, such as c++, does not describe an atomic action. Even very simple expressions can define complex actions that can decompose into other actions. However, there 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).

Atomic actions cannot be interleaved, so they can be used without fear of thread interference. However, this does not eliminate all need to synchronize atomic actions, because memory consistency errors are still possible.

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. What's more, 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.

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.

Some of the classes in the [java.util.concurrent](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html" \t "_blank) package provide atomic methods that do not rely on synchronization. We'll discuss them in the section on [High Level Concurrency Objects](http://docs.oracle.com/javase/tutorial/essential/concurrency/highlevel.html).

**Volatile**

**Volatile keyword in Java** is used as an indicator to Java compiler and  [Thread](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html)that do not cache value of this variable and always read it from [main memory](http://javarevisited.blogspot.sg/2011/05/java-heap-space-memory-size-jvm.html). So if you want to share any variable in which read and writes operation is atomic by implementation e.g. read and write in int or boolean variable you can declare them as volatile variable.

[**Java volatile keyword**](http://javarevisited.blogspot.com/2011/06/volatile-keyword-java-example-tutorial.html)cannot be used with method or class and it can only be used with variable. Java volatile keyword also guarantees **visibility** and **ordering** , after Java 5 write to any volatile variable happens before any read into volatile variable. By the way use of volatile keyword also prevents compiler or JVM from reordering of code or moving away them from[synchronization barrier](http://javarevisited.blogspot.sg/2011/04/synchronization-in-java-synchronized.html).  
  
**Example of volatile keyword in Java:**

To Understand example of volatile keyword in java let’s go back to [Singleton pattern in Java](http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html) and see [double checked locking in Singleton](http://javarevisited.blogspot.gr/2012/07/why-enum-singleton-are-better-in-java.html) with Volatile and without volatile keyword in java.

/\*\*  
 \* Java program to demonstrate **where to use Volatile keyword in Java**.  
 \* In this example Singleton Instance is declared as volatile variable to ensure  
 \* every thread see updated value for \_instance.  
 \*   
 \* @author Javin Paul  
 \*/  
**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;  
  
}

If you look at the code carefully you will be able to figure out:

1) We are only creating instance one time

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

If we do not make \_instance variable volatile then Thread which is creating [instance of Singleton](http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html) 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](http://javarevisited.blogspot.sg/2012/06/common-cause-of-javalangnullpointerexce.html).

Why because reader threads are not doing any locking and until writer thread comes out of [synchronized block](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html), 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.

So in Summary apart from [synchronized keyword in java](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html), volatile keyword is also used to communicate content of memory between threads.

**Let’s see another example of volatile keyword in Java:**

[Volatile variable example in Java](http://4.bp.blogspot.com/-v_PbvhXcJ7I/UBJ7S-JdzAI/AAAAAAAAAZ4/Z7XHJpHQ_f4/s1600/scratch_001.gif) Most of the time while writing game we use a variable bExist to check whether user has pressed exit button or not, value of this variable is updated in [event thread](http://javarevisited.blogspot.sg/2011/09/invokeandwait-invokelater-swing-example.html) and checked in game thread , So if we don't  use volatile keyword with this variable , Game Thread might miss update from event handler thread if its not [synchronized in java](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html) already. volatile keyword in java guarantees that value of volatile variable will always be read from main memory  and  "**happens-before**" relationship in **Java Memory model** will ensure that content of memory will be communicated to different threads.

 private **boolean** bExit;  
  
 while(!bExit) {  
    checkUserPosition();  
    updateUserPosition();  
 }

In this code example One Thread (Game Thread) can cache the value of "bExit" instead of getting it from [main memory](http://javarevisited.blogspot.sg/2011/05/java-heap-space-memory-size-jvm.html) every time and if in between any other thread (Event handler Thread) changes the value; it would not be visible to this thread. Making boolean variable "bExit" as **volatile in java** ensures this will not happen.

## When to use Volatile variable in Java

One of the most important thing in learning of volatile keyword is understanding *when to use volatile variable in Java*. Many [programmer](http://javarevisited.blogspot.sg/2011/06/top-programming-interview-questions.html) knows what is volatile variable and How does it work but they never really used volatile for any practical purpose. Here is couple of example to demonstrate when to use Volatile keyword in Java:

1) You can use Volatile variable if you want to read and write long and [double](http://javarevisited.blogspot.sg/2011/10/convert-double-to-string-example.html) variable atomically. **Long** and **Double** both are [64 bit](http://javarevisited.blogspot.sg/2012/01/find-jvm-is-32-or-64-bit-java-program.html) data type and by default writing of long and double is not atomic and platform dependence. Many  platform perform write in long and double variable 2 step, writing 32 bit in each step, due to this it’s possible for a Thread to see 32 bit from two different write. You can avoid this issue by making long and double variable volatile in Java.

2) Volatile variable can be used as an alternative way of achieving [synchronization in Java](http://javarevisited.blogspot.sg/2011/04/synchronization-in-java-synchronized.html) in some cases, like Visibility. With volatile variable it’s guaranteed that all reader thread will see updated value of volatile variable once write operation completed, without volatile keyword different reader thread may see different values.

3) Volatile variable can be used to inform compiler that a particular field is subject to be accessed by multiple threads, which will prevent compiler from doing any reordering or any kind of optimization which is not desirable in multi-threaded environment. Without volatile variable compiler can re-order code, free to cache value of volatile variable instead of always reading from [main memory](http://javarevisited.blogspot.sg/2011/08/increase-heap-size-maven-ant.html). Like following example without volatile variable may result in [infinite loop](http://javarevisited.blogspot.sg/2011/12/how-to-traverse-or-loop-hashmap-in-java.html)

**private** **boolean** isActive = thread;  
**public** **void** printMessage(){  
  while(isActive){  
     **System**.out.println("Thread is Active");  
  }  
}

without volatile modifier its not guaranteed that one [Thread](http://javarevisited.blogspot.sg/2012/01/difference-thread-vs-runnable-interface.html) see the updated value of isActive from other thread. compiler is also free to cache value of isActive instead of reading it from main memory in every iteration. By making isActive a volatile variable you avoid these issue.

4) Another place where volatile variable can be used is to fixing double checked locking in Singleton pattern. As we discussed in [Why should you use Enum as Singleton](http://javarevisited.blogspot.gr/2012/07/why-enum-singleton-are-better-in-java.html) that double checked locking was broken in Java 1.4 environment.

**Important points on Volatile keyword in Java**

1. *Volatile keyword in Java is only application to variable* and using volatile keyword with class and method is illegal.

2. Volatile keyword in Java guarantees that value of **volatile variable** will always be read from main memory and not from Thread's local cache.

3. In Java reads and writes are [atomic](http://javarevisited.blogspot.sg/2012/02/what-is-race-condition-in.html) for all variables declared using **Java volatile keyword** (including long and double variables).

4. Using Volatile keyword in Java on variables reduces the risk of **memory** **consistency errors**, because any write to a volatile variable in Java establishes a happens-before relationship with subsequent reads of that same variable.

5. From Java 5 changes to a volatile variable are always visible to other threads. What’s more it also means that when a thread reads a volatile variable in java, it sees not just the latest change to the volatile variable but also the side effects of the code that led up the change.

6. Reads and writes are atomic for reference variables are for most primitive variables (all types except long and double) even without use of volatile keyword in Java.

7. An access to a volatile variable in Java never has chance to block, since we are only doing a simple read or write, so unlike a synchronized block we will never hold on to any lock or wait for any [lock](http://javarevisited.blogspot.sg/2010/10/what-is-deadlock-in-java-how-to-fix-it.html).

8. Java volatile variable that is an object reference may be null.

9. Java volatile keyword doesn't means atomic, its common misconception that after declaring volatile ++ will be atomic, to make the operation atomic you still need to ensure exclusive access using [synchronized method or block in Java](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html).

10. If a variable is not shared between[multiple threads](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html) no need to use volatile keyword with that variable.

**Difference between synchronized and volatile keyword in Java**

Difference between volatile and synchronized is another popular core Java question asked in multi-threading and concurrency interviews. Remember volatile is not a replacement of synchronized keyword but can be used as an alternative in certain cases. Here are few differences between volatile and synchronized keyword in Java.

1. Volatile keyword in java is a field modifier, while [synchronized modifies code blocks and methods](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html).

2. Synchronized obtains and releases lock on monitor’s java volatile keyword doesn't require that.

3. Threads in Java can be blocked for waiting any monitor in case of synchronized, that is not the case with volatile keyword in Java.

4. [Synchronized method affects performance](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html) more than volatile keyword in Java.

5. Since volatile keyword in Java only synchronizes the value of one variable between Thread memory  and "main" memory  while synchronized synchronizes the value of all variable between thread memory and "main" memory and locks and releases a monitor to boot. Due to this reason [synchronized keyword in Java](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html) is likely to have more overhead than volatile.

6. You can’t synchronize on null object but your volatile variable in java could be null.

7. From Java 5 Writing into a volatile field has the same memory effect as a monitor release, and reading from a volatile field has the same memory effect as a monitor acquire

In Summary ***volatile keyword in Java*** is not a replacement of synchronized block or method but in some situation is very handy and can save performance overhead which comes with [use of synchronization in Java](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html)

**Synchronization**

## Example of Synchronization in Java using synchronized method and block

Synchronization in Java is an important concept since Java is a multi-threaded language where multiple threads run in parallel to complete program execution. In multi-threaded environment *synchronization of java object or synchronization of java class becomes extremely important*. Synchronization in Java is possible by using***java keyword "synchronized"*** and ***"volatile”***. Concurrent access of shared objects in Java introduces to kind of errors: thread interference and memory consistency errors and to avoid these errors you need to properly synchronize your java object to allow mutual exclusive access of critical section to two threads.

This **Java Synchronization tutorial** is in continuation of my article [**How HashMap works in Java**](http://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html) and[**difference between HashMap and Hashtable in Java**](http://javarevisited.blogspot.com/2010/10/difference-between-hashmap-and.html)  if you haven’t read already you may find some useful information based on my experience in Java Collections.

### Why do we need Synchronization in Java?

If your code is executing in multi-threaded environment you need **synchronization for objects** which are shared among multiple threads to avoid any corruption of state or any kind of unexpected behavior. Synchronization in Java will only be needed if shared object is mutable. if your shared object is read only or immutable object you don't need synchronization despite running multiple threads. Same is true with what threads are doing with object if all the threads are only reading value then you don't require **synchronization in java**. JVM guarantees that ***Java synchronized code will only be executed by one thread at a time***.  
  
In Summary Java Synchronized Keyword provides following functionality essential for concurrent programming :  
  
1) synchronized keyword in java provides locking which ensures mutual exclusive access of shared resource and prevent data race.  
2) synchronized keyword also prevent reordering of code statement by compiler which can cause subtle concurrent issue if we don't use synchronized or [volatile keyword](http://javarevisited.blogspot.com/2011/06/volatile-keyword-java-example-tutorial.html).  
3) synchronized keyword involve locking and unlocking. before entering into synchronized method or block thread needs to acquire the lock at this point it reads data from main memory than cache and when it release the lock it flushes write operation into main memory which eliminates memory inconsistency errors.

### Synchronized keyword in Java

Prior to Java5 synchronized keyword in java was only way to provide synchronization of shared object. Any code written in **synchronized block in java** will be mutual exclusive and can only be executed by one thread at a time. You can have both *static synchronized method and non static synchronized method*and synchronized blocks in java but we **can not** have **synchronized variable in java**.

Using synchronized keyword with variable is illegal and will result in compilation error. Instead of java synchronized variable 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.

***Block synchronization in java is preferred over method synchronization in java*** because by using block synchronization you only need to lock the critical section of code instead of whole method. Since java synchronization comes with cost of performance we need to synchronize only part of code which absolutely needs to be synchronized.

### Example of synchronized method in Java

Using **synchronized keyword** along with method is easy just apply synchronized keyword in front of method. What we need to take care is that static synchronized method locked on class object lock and non static synchronized method locks on current object (this). So it’s possible that both static and non static java synchronized method running in parallel.  This is the common mistake a naive developer do while **writing java synchronized code**.

**public class Counter**{

private static int count = 0;

public **static synchronized** int getCount(){

  return count;

}

public **synchoronized**setCount(int count){

   this.count = count;

}

}

In this example of *java synchronization code is not properly synchronized* because both getCount() and setCount() are not getting locked on same object and can run in parallel which results in getting incorrect count. Here getCount() will lock in Counter.class object while setCount() will lock on current object (this). To make this code properly synchronized in java *you need to either make both method static or non static or use java synchronized block instead of java synchronized method*.

### Example of synchronized block in Java

Using **synchronized block in java** is also similar to using **synchronized keyword in methods**. Only important thing to note here is that if object used to lock synchronized block of code, Singleton.class in below example is null then java synchronized block will throw a NullPointerException.

**public class Singleton{**

private static volatile Singleton \_instance;

public static Singleton getInstance(){

   if(\_instance == null){

**synchronized(Singleton.class){**

              if(\_instance == null)

              \_instance = new Singleton();

            }

   }

   return \_instance;

}

This is a classic example of double checked locking in Singleton. In this **example of java synchronized code**we have made only critical section (part of code which is creating instance of singleton) synchronized and saved some performance because if you make whole method synchronized every call of this method will be blocked while you only need to create instance on first call. To read more about [Singleton in Java](http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html) see here.

### Important points of synchronized keyword in Java

[synchronized keywrod java example , synchronization in java tutorial](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html)1. **Synchronized keyword in Java** is used to provide mutual exclusive access of a shared resource with multiple threads in Java. Synchronization in java guarantees that no two threads can execute a synchronized method which requires same lock simultaneously or concurrently.

2. You can use java synchronized keyword only on synchronized method or synchronized block.

3. Whenever a thread enters into java synchronized method or block it **acquires a lock** and whenever it leaves java synchronized method or block it releases the lock. Lock is released even if thread leaves synchronized method after completion or due to any Error or Exception.

4. Java Thread acquires an **object level lock** when it enters into an instance synchronized java method and acquires a class level lock when it enters into static synchronized java method.

5.**java synchronized keyword is re-entrant in nature** it means if a java synchronized method calls another synchronized method which requires same lock then current thread which is holding lock can enter into that method without acquiring lock.

6. **Java Synchronization** will throw **NullPointerException**if object used in **java synchronized block is null**e.g. synchronized (myInstance) will throws NullPointerException if myInstance is null.

7. One Major **disadvantage of java synchronized keyword** is that it doesn't allow concurrent read which you can implement using java.util.concurrent.locks.ReentrantLock.

8. One **limitation of java synchronized keyword** is that it can only be used to control access of shared object within the same JVM. If you have more than one JVM and need to synchronized access to a shared file system or database, the java synchronized keyword is not at all sufficient. You need to implement a kind of global lock for that.

9. **Java synchronized keyword incurs performance cost.** Synchronized method in Java is very slow and can degrade performance. So use synchronization in java when it absolutely requires and consider using java synchronized block for synchronizing critical section only.

10. **Java synchronized block is better than java synchronized method** in java because by using synchronized block you can only lock critical section of code and avoid locking whole method which can possibly degrade performance. A good example of java synchronization around this concept is[getInstance() method Singleton class](http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html). See here.

11. Its possible that **both static synchronized and non static synchronized method can run simultaneously** or concurrently because they lock on different object.

12. From java 5 after change in Java memory model **reads and writes are atomic for all variables declared using volatile keyword** (including long and double variables) and simple atomic variable access is more efficient instead of accessing these variables via synchronized java code. But it requires more care and attention from the programmer to avoid memory consistency errors.

13. **Java synchronized code could result in deadlock or starvation** while accessing by multiple thread if synchronization is not implemented correctly. To know [how to avoid deadlock in java](http://javarevisited.blogspot.com/2010/10/what-is-deadlock-in-java-how-to-fix-it.html) see here.

14. According to the Java language specification **you can not use java synchronized keyword with constructor** it’s illegal and result in compilation error. So you can’t synchronized constructor in Java which seems logical because other threads cannot see the object being created until the thread creating it has finished it.

15. **You cannot apply java synchronized keyword with variables**and can not use java volatile keyword with method.

16. Java.util.concurrent.locks extends capability provided by java synchronized keyword for writing more sophisticated programs since they offer more capabilities e.g. **Reentrancy**and **interruptible locks**.

17. **java synchronized keyword also synchronizes memory**. In fact java synchronized synchronizes the whole of thread memory with main memory.

18. Important method related to **synchronization in Java are wait(), notify() and notifyAll()**which is defined in Object class.  
19. D**o not synchronize on non final field on synchronized block in Java**. because reference of non final field may change any time and then different thread might synchronizing on different objects i.e. no synchronization at all. example of synchronizing on non final field :

private String lock = new String("lock");  
**synchronized(lock){**  
System.out.println("locking on :"  + lock);

}  
any if you write synchronized code like above in java you may get warning **"Synchronization on non-final field"**  in IDE like Netbeans and InteliJ  
  
20. Its **not recommended to use String object as lock in java synchronized block** because [string is immutable object](http://javarevisited.blogspot.com/2010/10/why-string-is-immutable-in-java.html) and literal string and interned string gets stored in String pool. so by any chance if any other part of code or any third party library used same String as there lock then they both will be locked on same object despite being completely unrelated which could result in unexpected behavior and bad performance. Instead of String object its advised to use new Object() for **Synchronization in Java on synchronized block**.  
  
**private static final String LOCK = "lock";   //not recommended**  
private static final Object OBJ\_LOCK = new Object(); //better  
  
public void process() {  
  **synchronized(LOCK) {**  
      ........  
   }  
}  
  
21. From Java library Calendar and SimpleDateFormat classes are not thread-safe and requires**external synchronization in Java** to be used in multi-threaded environment.  

Probably most important point about synchronization in Java is that in the absence of synchronized keyword or construct compiler, JVM and hardware are free to make optimization, assumption, reordering or caching of code and variable which can cause subtle concurrency bugs in code. By introducing synchronization may be either using volatile or synchronized keyword we instruct compiler or JVM to not to do that

If you like to read UNIX command tips you may find  [10 tips of using find command in Linux](http://javarevisited.blogspot.com/2011/03/10-find-command-in-unix-examples-basic.html),  [10 tips to increase speed on Unix command](http://javarevisited.blogspot.com/2011/03/unix-command-tutorial-working-fast-in.html)and  [10 basic networking Commands in Unix](http://javarevisited.blogspot.com/2010/10/basic-networking-commands-in-linuxunix.html) useful.  
  
Update: Recently I have been reading several **java synchronization and concurrency** articles in internet and I come across jeremymanson's blog which works in google and has worked on JSR 133 **Java Memory Mode**l, I would recommend some of this blog post for every java developer, he has covered certain details about concurrent programming , synchronization and volatility in simple and easy to understand language, here is the link [atomicity, visibility and ordering](http://jeremymanson.blogspot.com/2007/08/atomicity-visibility-and-ordering.html)**.**[**http://jeremymanson.blogspot.com/2007/08/atomicity-visibility-and-ordering.html**](http://jeremymanson.blogspot.com/2007/08/atomicity-visibility-and-ordering.html)