Synchronization in Java

Java is a multi-threaded language where multiple threads run in parallel to complete program execution. In such environment, synchronization of **Java objects** and **Java classes** becomes extremely important.

**Synchronized keyword in java provides mutual exclusive access of critical section, of shared resource among multiple thread in java. Synchronization assures that only one thread at a time can access the shared resource.**

Achieved by two keyword- "**synchronized**" and "**volatile**”

**“Concurrent access of shared objects**” introduces to kind of errors –

1. Thread interference (corruption of state)
2. Memory consistence.

In order to avoid these errors we have to properly synchronize object/classes to allow **mutual exclusive access of critical section**.

1. Synchronization provides "locking" which ensures "mutual exclusive access” of shared resource and prevent data race.
2. Synchronized keyword also prevents reordering of code statement by compiler.

Any- thread 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 before it releases the lock, it flushes write operation into main memory which eliminates memory inconsistency errors.

Note:-

1. Read only and[Immutable object](http://javarevisited.blogspot.com/2013/03/how-to-create-immutable-class-object-java-example-tutorial.html), DO NOT need synchronization.
2. Code inside synchronized block or enclosed inside synchronized method will be “mutually exclusive”, and can only be executed by one thread at a time.
3. You can have both [static synchronized method and non static synchronized method](http://javarevisited.blogspot.sg/2012/03/mixing-static-and-non-static.html) also synchronized blocks in Java but CAN NOT have synchronized variable(compilation error).
4. Instead of synchronized variable in Java, you can make use of java volatile variable, which will instruct JVM threads to read value of volatile variable from main memory and DO NOT cache it locally.
5. Block synchronization is preferred over method synchronization because synchronization comes with cost of performance; we need to synchronize only part of code which absolutely needs to be synchronized.
6. static synchronized method locked on class object lock and non static synchronized method locks on current object (this), So it is possible that both static and non static java synchronized method running in parallel.

**Example**: -

**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;

}

1. [Java synchronized keyword is re-entrant in nature it means if a java synchronized method calls another synchronized method which requires same lock thencurrent thread](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html) which is holding lock can enter into that method without acquiring lock.
2. Java synchronized feature come with the cost – “The performance” 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.
3. Volatile variable is more efficient instead of accessing these variables via synchronized block/method. But it requires more care and attention to avoid memory consistency errors.
4. synchronized keyword - synchronizes the whole of thread memory with main memory.
5. Important method related to synchronization in Java are wait(), notify() and notifyAll() which is defined in Object class, not in java.lang.Thread class, why ?
6. Do 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);

}

1. It is 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) {

//........

}

}

1. From Java library **Calendar** and **SimpleDateFormat** classes are not thread-safe and require external synchronization.

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

Disadvantages:-

1. Java Synchronization will throw NullPointerException if object used in java synchronized block is null e.g. synchronized (myInstance)will throws java.lang.NullPointerException if myInstance is null.
2. Disadvantage of Java synchronized keyword is that it doesn't allow concurrent read. (java.util.concurrent.locks.ReentrantReadWriteLock provides ready made implementation of ReadWriteLock in Java.)
3. If Synchronization is not implemented properly – I can cause “DeadLock”
4. Synchronized keyword cannot be used with Constructor it will throw “Compilation Error”.
5. Synchronized keyword doesn't allow separate locks for reading and writing. As we know that multiple thread can read without affecting thread, synchronized keyword suffer performance due to this in case of multiple reader and one or few writer.  
    If one thread is waiting for lock then there is no way to time out or interrupt the thread, thread can wait indefinitely for lock. All these limitation of synchronized keyword is addressed and resolved by using **ReadWriteLock** and **ReentrantLock** in Java 5.

Additional Info:-

1. synchronized keyword in internally implemented using two byte code instructions MonitorEnter and MonitorExit, this is generated by compiler. Compiler also ensures that there must be a MonitorExit for every MonitorEnter in different code path e.g. normal execution and abrupt execution, because of Exception.  
     
   java.util.concurrent package different locking mechanism than provided by synchronized keyword, they mostly used ReentrantLock, which internally use CAS operations, volatile variables and atomic variables to get better performance.
2. With synchronized keyword, you have to leave the lock, once you exist a synchronized method or block, there is no way you can take the lock to other method. java.util.concurrent.locks.ReentrantLock solves this problem by providing control for acquiring and releasing lock, which means you can acquire lock in method A and can release in method B, if they both needs to be locked in same object lock. Though this could be risky as compiler will neither check nor warn you about any accidental leak of locks. Which means, this can potentially block other threads, which are waiting for same lock.  
     
   Prefer ReentrantLock over synchronized keyword, it provides more control on lock acquisition, lock release and better performance compared to synchronized keyword.
3. Any thread trying to acquire lock using synchronized method will block indefinitely, until lock is available. Instead this, tryLock() method of java.util.concurrent.locks.ReentrantLock will not block if lock is not available.