### **Runnable vs Thread**

If your class provides more functionality rather than just running as Thread, you should implement Runnable interface to provide a way to run it as Thread. If your class only goal is to run as Thread, you can extend Thread class.  
Implementing Runnable is preferred because java supports implementing multiple interfaces. If you extend Thread class, you can’t extend any other classes.

Volatile Keyword

volatile keyword guarantees visibility of changes to variables across threads. variable value will never be cached thread-locally: all reads, and writes will go straight to "main memory".

class Processor extends Thread {

private volatile boolean running = true;

public void run() {

while(running) {

System.out.println("Running");

try {

Thread.sleep(50);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public void shutdown() {

running = false;

}

}

public class App {

public static void main(String[] args) {

Processor pro = new Processor();

pro.start();

// Wait for the enter key

new Scanner(System.in).nextLine();

pro.shutdown();

}

}

**Why volatile is Not Always Enough**

Even if the volatile keyword guarantees that all reads of a volatile variable are read directly from main memory, and all writes to a volatile variable are written directly to main memory, there are still situations where it is not enough to declare a variable volatile.

Consider two threads A and B having shared volatile variable (no synchronization ) 'counter', Thread A is used to incrementing the counter variable by 1 and Thread B is used to print the shared variable.

Output of code should be “1,2,3,4,5,6,....” , means Thread A is incrementing by 1 and then Thread B is printing counter value and so on.

As there is no Synchronization, Thread A and Thread B may execute like →

Thread A ,Thread A ,Thread B ,Thread A ,Thread B, Thread B ,... which may give output like → “2,3,3,...”

**Why synchronization is Not Always Enough**

Even if the synchronized keyword guarantees that all reads and write operation will happen sequentially, there are still situations where it is not enough to declare a variable volatile.

Example – Consider two threads A and B having shared synchronized variable (but no volatile) 'counter', Thread A is used to incrementing the counter variable by 1 and Thread B is used to print the shared variable.

Output of code should be “1,2,3,4,5,6,....” , means Thread A is incrementing by 1 and then Thread B is printing counter value and so on.

As counter is not volatile, Thread A and Thread B will execute sequentially →

Thread A ,Thread B ,Thread A ,Thread B, Thread A , Thread B ,... but it might possible read threads will read or write value from their local cache like   
Thread A   
which may give output like →

“2,3,3,...”

**Example with Volatile and Synchronization**

First have a look at Singleton Code.

public class Singleton {

private static Singleton \_instance;

public static Singleton getInstance() {

if (\_instance == null)

\_instance = new Singleton();

return \_instance;

}

}

Now if multiple threads will be making the object of this singleton class then this code will not work.

public class Singleton {

private static Singleton \_instance;

public static Singleton synchronized getInstance() {

if (\_instance == null) {

\_instance = new Singleton();

}

return \_instance;

}

}

Above Code will work fine but we have to bear the cost of synchronization in every call of getInstance() . Here Double checked locking pattern will come in picture. so, we can optimize this.

public class Singleton {

private static Singleton \_instance;

public static Singleton getInstance() {

if (\_instance == null) {

synchronized(Singleton.class) {

if (\_instance == null)

\_instance = new Singleton();

}

}

return \_instance;

}

}

From Execution control perspective, this code will work fine but threads can locally cache the singleton object so we need to declare the “\_instance” as volatile object.

Here double-checked locking Synchronization will guarantee all read and write operation will happen in sequentially manner and volatile keyword will guarantee all changes will reflect to all threads.

public class Singleton {

private volatile static Singleton \_instance;

public static Singleton getInstance() {

if (\_instance == null) {

synchronized(Singleton.class) {

if (\_instance == null)

\_instance = new Singleton();

}

}

return \_instance;

}

}

**Synchronized Keyword and Volatile Keyword.**

1. Volatile keyword in java is a field modifier, while synchronized modifies code blocks and methods.
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.