CLASSTIME | Page No.
Date / /

VOLATILE

Local variables in the thread.

If you are working with the multithreaded programming, the volatile keyword will be more useful. When multiple threads using the same variable, each thread will have its own copy of the local ~~variable~~ cache for that variable. So, when it is updating the value, it is actually updating in the local cache not in the main variable memory. The other thread which is using the same variable doesn't know anything about the values changed by the another thread.

To avoid this problem, if you declare a variable as volatile, then it ~~doesn't~~ will not be stored in local cache. Whenever thread are updating the values it is updated to the main memory. So, other threads can access the updated value.

- The Java 'volatile' keyword is used to mark a Java variable as "being stored in memory". More precisely that ~~means~~ means, that every read of the

volatile keyword variable will be read from the computers main memory, and not from the CPU cache, and that every write to a volatile variable will be written to main memory, not just the CPU cache.

Problem

CLASSTIME | Page No.
Date / /

Atomic

Program

```
public class IcwAtomics
{
    public static void main() throws InterruptedException
    {
        ProcessingThread pt = new ProcessingThread();
        Thread t1 = new Thread(pt, "t1");
        t1.start();
        Thread t2 = new Thread(pt, "t2");
        t2.start();
        t1.join();
        t2.join();
```

```
Sop ("Processing count=" + pt.getCount());
}

class ProcessingThread implements Runnable {
    private volatile int count;
    // int count;

    @Override
    public void run() {
        for (int i=1; i<5; i++) {
            Sop("hello");
            processSomething(i);
            count++;
        }
    }

    public int getCount() {
        return this.count();
    }

    public void processSomething(int i) {
        // processing some job
        try {
            Thread.sleep(1000);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

ATOMIC

- To solve this issue, we will have to make sure that increment operation on count is atomic, we can do that using Synchronization but Java 5 java.util.concurrent.atomic provides wrapper classes for int & long that can be used to achive this atomically without usage of Synchronization
- Here is the updated program that will always o/p count value as 8 bcoz AtomicInteger incrementAndGet() atomically increments current value by one..

Program

```
import java.util.concurrent.atomic.AtomicInteger;

public class JavaAtomic{

psvm() throws InterruptedExcepn {

ProcessingThread pt = new ProcessingThread();

Thread t1 = new Thread(pt, "t1");
t1.start();
```

```
Thread t2 = new Thread(pt, "t2");
    t2.start();
    t1.join()
    t2.join();
    Sop("Processing Count = " + pt.getCount());
}
}

Class ProcessingThread implements Runnable
{
private AtomicInteger count = new
            AtomicInteger();

public void run(){
for(int i = 1; i<5; i++){
    processSomething(i);
count.incrementAndGet();
}
}

public int getCount(){
    return this.count.get();
}

private void processSomething(int i){
    try{
        Thread.sleep(1000);
```

CLASSTIME Page No.
Date / /

```
}
catch(InterruptedException e)
{ e.printStackTrace(); }
}
}
```