**Homework Assignment 2: Design Document**

Q1) Consider the following partial code snippet below. You can assume the curr is an object of type Element that has two fields’ p and n. Given the code below should the fields’ p and n be marked as volatile? Should the variable temp be marked as volatile? You must justify your answers and your reasoning. You should create a text file with the answers to these questions and submit it along with your code for part 1. ***Please note that the solution to assignment one, which will be posted will not have the volatile keyword as to not give away the answer and a final solution will be posted after this assignment.***

temp = curr.p;

synchronized (temp){

synchronized (curr){

if(temp == curr.p){

synchronized (curr.n){

…

}

}

}

}

Should p be marked volatile in the class definition for Element - 10 points

Should n be marked volatile in the class definition for Element - 10 points

Should the temporary variable temp be marked volatile - 10 points

**Answer**: Let us first answer the simpler question which is when we do require a volatile variable? Well, we know that volatile keyword is a field modifier, which actually means that it changes the way a variable is accessed and the magic of making a variable as volatile can easily be seen in multithreaded environments, where there are multiple threads sharing the same program resources and we as programmers want to ensure the fact that most operations such as reads and writes to these shared variables happen atomically i.e in one instruction and also that the most updated value of the variable is communicated to threads in the program that have access to that variable. So some salient uses of volatiles are as follows -:

* If we declare a variable as volatile then that means the value of the variable will always be fetched from the main memory and not from the thread’s local cache.
* If we declare a variable as volatile then that’s an indication to the compiler and the JVM underneath to not re-order the statements that involve the volatile variables as that may disturb the total program order.
* If we declare a variable as volatile then that induces a happens-before relationship among the threads in the program, which essentially means that all writes to a volatile variable happen first and only after these writes have been performed can any subsequent reads happen, this help the Java Memory Model to maintain memory consistency.
* If we declare primitive longs and doubles as volatiles then that helps us perform reads and writes of these primitive variables atomically as they are not atomic by default.
* Volatiles help us instill some kind synchronization but it’s a synchronization that addresses the visibility issues of a variable across threads, because one doesn’t need to hold any kind of locks and block other threads from gaining access to a variable, so it’s kind of a non-blocking synchronization we have with volatiles.

*“Should p be marked volatile in the class definition for Element?”*

* The answer to this question would be yes, to give an explanation for the answer, let us consider the Element class definition:

*public class Element {*

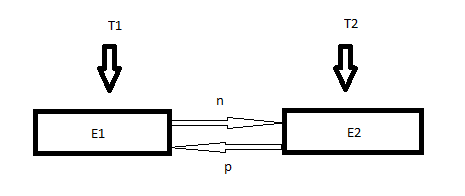
*int val;*

*Element p;*

*Element n;*

*}*

And let us say that initially there are 2 nodes in the linked list of type Element, and also imagine that there are 2 threads T1 and T2 out of which T1 is trying to insert a node after the element E1 and the thread T2 is trying to read the value of the previous pointer of the Element E2. The thing to notice in Java is that all writes and reads of the reference variables are atomic in Java. Even though the reads and writes of the object reference variables in Java are atomic it doesn’t guarantee us that we are reading the correct value or writing the correct value, to ensure such consistency we must have a mechanism for communicating this most updated information to all the threads in the system. To instill this in the system we use volatiles. So to answer the question, yes we will make the “p” field in the Element class as volatile, because the value of p field will be written by T1 if it is trying to insert a node after E1 because in that process it will be updating the previous field of the E2 and if in that period T2 tries to read the value of E2’s p field it will be the old cached value of its p field that it will read which is something that we don’t want, this happens because the reader thread has no way to see the most updated value of the p field, its only when the writer thread moves out of the synchronized block that the value of the field is reflected across threads, so to provide immediate updated lookups yes we need to make the p field of the Element class as volatile and also to ensure the fact that all writes to the variable happen before any subsequent reads, Java handles all this internally if we declare the variables as volatiles also there are no compiler re-orderings that are performed when there are volatile variables in the program.



*“Should n be marked volatile in the class definition for Element?”*

* The answer to this question would be yes as it is based on a similar sort of reasoning as the previous answer but demands a slightly different context in which it is preferred to make the n variable as volatile. In this we imagine if the thread T2 inserts before the element E2 and the thread T1 wants to read the next variable of element E1 then if the next variable of the element class is not volatile the thread T1 won’t be able to read the most updated value of n as T1 has no way to know if the thread T2 updated the value of its next pointer, or not and it still believes that its next pointer keeps pointing to E2 instead of the newly inserted element, hence to ensure that we have a happens-before relationship between the reads and writes of the field variable n and that all writes to the variable must occur before any subsequent reads, we should make n as volatile.

*“Should the temporary variable temp be marked volatile?”*

* The answer to this question is “yes we can make the temp variable as volatile but there is **no reason** to do so” this is because temp is a temporary variable that’s stored in the thread’s local cache and it just stores the value of element’s previous pointer, hence it’s the element’s previous field that requires special care and not the temporary variable temp. The modifications if any done using the temp variable eventually happen on either the current variable or current’s next field or current’s previous field, so temp is there just for reference manipulation purposes and nothing else and as it is all local variables are stored inside the thread’s local stack.

Q2) Please describe the difference between synchronized blocks and methods. In your discussion show how you can encode the behavior of synchronized methods through the use of synchronized blocks.

Answer: The synchronized keyword in java is the mechanism that Java provides programmers to support the concept of locking, which in turn is used to protect shared resources from illegal and inconsistent access and helps restore memory consistency inside the multithreaded program. Let us first take a look at what are synchronized methods in effect. Well there are different flavors of synchronized methods in java, lets discuss them one by one:

1. **Synchronized Instance Methods** : A synchronized instance method in java is declared using the following declaration pattern –

*public* ***synchronized*** *void doSomething(){}*

This is the same old method declaration in java but with the synchronized keyword in its declaration. Whenever doSomething() method is called on an object this method firstly acquires the lock on the object is has been called on, this is called a monitor lock, so if for instance there are 2 or more threads sharing the same object and calling the doSomething() method simultaneously then there will be only a single thread that will get access to this method which will be the one that has acquired the lock first. So, the synchronized method in this case implicitly locks the object it was called on.

1. **Synchronized Class Methods** : A synchronized class method in java is one which acquires the lock on the entire class and this is a static method of the class which is declared as follows:

*public static synchronized void doSomething(){}*

This method when called from outside the class first will acquire the implicit class lock which is **“Class\_Name.class”** and then will start executing the statements inside the method. So if 2 or more threads have to execute the doSomething() method they will first have to acquire the class lock and proceed with the execution.

The above use of synchronized methods is one way we can achieve synchronization in a program but the downside with above approach is that it hampers the performance of the system as there can only be just one thread in the system that can hold the lock exclusively, and other threads are not allowed even to enter the methods. An alternative mechanism that java provides us to inculcate synchronization within the program is using synchronized blocks, and there are 2 flavors of synchronized blocks too, well synchronized blocks are preferred over synchronized methods because they can be used to lock specific portions of the code rather than blocking access to entire methods, let us look at a situation in which we can use synchronized blocks of code fruitfully -:

*“Let us say that we have a printing machine available with us that performs printing of paper but besides printing paper there are other functions in the machine that can be handled independently of the printing procedure for instance loading the paper tray, loading the printer ink and so on.”*

Looking at the above situation it makes no sense to lock up an entire machine to perform a single task. The same is true with programming in java. Let us see how we can map the above situation in the programming world using lock objects:

public class Printer{

Object printLockObject = new Object();

Object paperLoadLockObject = new Object();

Object inkLoadObject = new Object();

public void print(){

synchronized(printLockObject){ // do something }

}

public void loadPaper(){

synchronized(paperLoadObject){// do something }

}

public void loadInk(inkLoadObject){

synchronized(inkLoadObject){// do something }

}

}

The above is the power of synchronized blocks of code that have been provided to us by Java, with the help of these we can lock specific portions of code rather than locking up an entire object which actually is very restrictive in some sense and also hampers the performance of the system.

Encoding the behavior of synchronized methods using synchronized blocks of code, this can be done in the following way:

For every inclusion of synchronized keyword in the method declaration enclose all the statements in the method inside the following synchronized block which actually acquires a lock on the object instance it was called on:

public synchronized void doSomething(){

// do something

}

Is equivalent to:

public void doSomething(){

synchronized(this){

// do something

}

}