Quiz 6

Questions regarding memory visibility in multithreaded scenarios

Question # 1

Consider the below class:

```
public class MemoryVisibility {

    int myvalue = 2;
    boolean done = false;

    void thread1() {

        while (!done);
        System.out.println(myvalue);

    }

    void thread2() {

        myvalue = 5;
        done = true;
    }
}
```

We create an object of the above class and have two threads run each of the two methods like so:

```
        MemoryVisibility mv = new MemoryVisibility();

        Thread thread1 = new Thread(() -> {
            mv.thread1();
        });

        Thread thread2 = new Thread(() -> {
            mv.thread2();
```

```
        });

        thread1.start();
        thread2.start();

        thread1.join();
        thread2.join();
```

What will be the output by thread1?

Q

A) loops forever and doesn't print anything

B) prints 5

C) prints 2

D) May loop forever or print 2 or print 5

CHECK ANSWERS

Show Explanation

Question # 2

Will the following change guarantee that thead1 sees the changes made to shared variables by thead2?

```
public class MemoryVisibility {
```

```
    int myvalue = 2;
    boolean done = false;

    void thread1() {

        synchronized (this) {
            while (!done);
            System.out.println(myvalue);
        }
    }

    void thread2() {

        myvalue = 5;
        done = true;
    }
}
```

Q

- A) Yes
- B) No

CHECK ANSWERS

Show Explanation

Question # 3

Does `synchronized` ensure memory visibility? How can we fix the above code using synchronization?

We have already seen that synchronization ensures ***atomicity*** i.e. operations within a synchronized code block all execute together without interruption. You can imagine these operations to be executed like a transaction, where either all of them execute or none execute.

From a memory visibility perspective, say two threads A and B are synchronized on the same object. Once thread A exits the synchronized block (releases the lock), all the variable values that were visible to thread A prior to leaving the synchronized block (releasing the lock) will become visible to thread B as soon as thread B enters the synchronized block (acquires the lock).

The memory visibility class can be fixed with synchronization as follows:

```
public class MemoryVisibility {

    int myvalue = 2;
    boolean done = false;

    void thread1() throws InterruptedException {

        synchronized (this) {
            while (!done)
                this.wait();
            System.out.println(myvalue);
        }
    }

    void thread2() {

        synchronized (this) {
            myvalue = 5;
            done = true;
            this.notify();
        }
    }
}
```

Question # 4

Describe `volatile`? Can it help us with the `MemoryVisibility` class.

When a field is declared `volatile`, it is an indication to the compiler and the runtime that the field is shared and operations on it shouldn't be reordered. Volatile variables aren't cached in registers or caches where they are hidden from other processors. Note that variables declared `volatile` when read always return the most recent write by any thread.

Furthermore volatile variables only guarantee memory visibility but not atomicity.

In the fixed `MemoryVisibility` class using synchronization may seem an overkill as acquiring and releasing locks is never cheap. Volatile provides a weaker form of synchronization and can alleviate the situation in the `MemoryVisibility` class if we declare both the shared variables `volatile`.

Question # 5

Will it be enough to declare the `done` flag `volatile` or do we need to declare `myvalue` `volatile` too?

```
public class MemoryVisibility {

    int myvalue = 2;
    volatile boolean done = false;

    void thread1() {

        while (!done);
        System.out.println(myvalue);

    }

    void thread2() {

        myvalue = 5;
```

```
            done = true;
            this.notify();

        }
    }
```

It is intuitive to think that if we declare just the boolean flag `volatile`, it'll prevent from infinite looping but the latest value for the variable `myvalue` may not get printed, since it is not declared `myvalue`. However, that is not true and we can get away by only declaring the boolean flag as `volatile`. Though note that declaring both the shared variables `volatile` is acceptable too.

Writing to a `volatile` variable is akin to exiting a synchronized block and reading a volatile variable is akin to entering a synchronized variable. Similar to the visibility guarantees for a synchronized block, after a reader-thread reads a volatile variable, it sees the same values of all the variables as seen by a writer-thread just before the writer-thread wrote to the same volatile variable.

Question # 6

If we introduced a third thread that could also mutate the value of `myvalue` variable in the fixed `MemoryVisibility` class that uses `volatile`, how can that affect the value printed by thread1?

Consider the below sequence of thread scheduling

- Thread 2 changes the value of `myvalue` to 5 and sets the volatile flag `done` to true
- Thread 3 mutates the value of `myvalue` to say 16 that gets stored in the register

- Thread 1 when scheduled will be guaranteed to see all the values of variables when `done` was updated to true by thead 2. At that time `myvalue` was set to 5 and even though thread 3 changed it to 16, there's no guarantee that thread 2 sees it because it happened after the write to the volatile variable. Therefore at this point it may print 5 or 16.

Question # 7

When is `volatile` most commonly used?

Common situation where `volatile` can be used are:

- Most common use of volatile variables is as a interruption, completion or status flag
- When writes to a variables don't depend on its current value e.g. a counter is not suitable to be declared volatile as its next value depends on its current value.
- When a single thread ever writes to the variable. Imagine a scenario where only a single thread writes or modifies a shared volatile variable but the variable is read by several other threads. In this situation, race conditions are prevented because only one thread is allowed to write to the shared variable and visibility guarantees of volatile ensure other threads see the most up to date value.
- When locking isn't required for reading the variable or that the variable doesn't participate in maintaining a variant with other state variables