The happens-before Relationship and Model

This lesson continues the in-depth discussion of Java memory model

To understand the *happens-before* relationship we'll need to grasp few related concepts first.

Total order

You are already familiar with total ordering, the sequence of natural numbers i.e. 1, 2, 3 4, … is a total ordering. Each element is either greater or smaller than any other element in the set of natural numbers (**Totality**). If **2 < 4** and **4 < 7** then we know that **2 < 7** necessarily (**Transitivity**). And finally if **3 < 5** then 5 can't be less than 3 (**Asymmetry**).

Partial order

Elements of a set will exhibit *partial ordering* when they possess transitivity and asymmetry but not totality. As an example think about your family tree. Your father is your ancestor, your grandfather is your father's ancestor. By transitivity, your grandfather is also your ancestor. However, your father or grandfather aren't ancestors of your mother and in a sense they are incomparable.

Actions

A program consists of *actions* which can be any of:

1. reads or writes
2. lock and unlocks of a monitor or other locking constructs
3. start of a thread or detecting a thread's termination
4. read or write of volatile variables

The last three actions in the above list are categorized as ***synchronization actions***.

Synchronization order

Each execution of a program has a synchronization order. A synchronization order is a *total order* over all of the synchronization actions of an execution. In terms of synchronization order a data race is defined by JSR-133 as:

> A data race occurs in an execution of a program if there are conflicting actions in that execution that are not ordered by synchronization.

Sequential consistency

Sequential consistency says that all actions within a program are atomic and follow a total order, which matches the program order. A program that has no data races will exhibit sequential consistency across all its runs. However, sequential consistency doesn't imply program correctness. Groups of operations that need to be executed atomically may not execute as such giving rise to errors.

In the sequential memory model a read *R* of a variable *V* sees the value written to the variable *V* by a write *W*, that occurs before *R* in the execution order. Furthermore there's no other write that takes place in-between *W* and *R*.

The sequential memory model is too restrictive and doesn't allow for many of the compiler and hardware optimizations. Quoting JSR-133:

> *A program is correctly synchronized if and only if all sequentially consistent executions are free of data races.*

Happens-before memory model

The happens-before memory model has several requirements and properties. It is mathematically described at great length in JSR-133. For this course, we'll present a brief gist. From the synchronization order of a program a partial order called the *happens-before* order is derived. In a happens-before memory model:

- A read *R* of a variable *V* sees the write *W* to the variable *V*, if *W* occurs before *R* in the *happens-before* partial order of the program execution and there is no other intervening write between *W* and *R*.
- A read *R* of a volatile variable observes the last write *W* of the variable in the synchronization order.

When a program contains two conflicting accesses that are not ordered by a *happens-before*

relationship, it is said to contain a data race. A correctly synchronized program is one that has no data races. Furthermore, if a program is correctly synchronized, then all executions of the program will appear to be sequentially consistent.

The JMM is formally defined in JSR-133 in terms of actions and executions. The JMM isn't purely a happens-before model but adds upon it. The happens-before model is a weak model but in the context of JMM is necessary though not sufficient by itself. We'll skip the weaknesses of the happens-before model, and leave at that. More details can be found in JSR-133, while we'll focus on the practical applications of the JMM next.

Example 1

Informally, we can summarize the above theoretical discussion as actions upon shared data in two threads need to have a happens-before relationship established among them. Without a happens-before relationship the conflicting access to the shared data can result in concurrency bugs. Consider the pseudocode for two methods, each executed by a distinct thread below:

```
void method1() {
    x = 1;
    lock monitor;
    y = 5;
    unlock monitor;
}

void method2(){
    lock monitor;
    r1 = y;
    unlock monitor;
    r2 = x;
}
```

Or we can visualize the code for the two methods as follows:

Thread 1	Thread 2
x = 1	lock monitor
lock monitor	r1 = y
y = 5	unlock monitor
unlock monitor	r2 = x

Without a memory model we can't reason about the ordering of the statements between the two threads. For instance, we can't say if `x = 1` in thread1 is necessarily executed before `r2 = x` in thread2. For statements within the same thread, JMM assures us that any reordering if performed will not alter the outcome of the program. Our concern is then focused on the ordering guarantees of the statements from both the threads. Using synchronization constructs we can **establish the ordering among statements executed by different threads**. For example, the JMM mandates that unlocking of a monitor *happens-before* every subsequent locking of that monitor. Given this, let's consider the following execution trace of our example:

Execution #	Thread 1	Thread 2
1.	x = 1	
2.	lock monitor	
3.	y = 5	
4.	unlock monitor	
5.		lock monitor
6.		r1 = y
7.		unlock monitor
8.		r2 = x

The action of unlocking the monitor at execution#4 by thread1 establishes a happens-before relation with the action of locking the monitor at execution#5 by thread2. The statement `x = 1` happens before the unlocking of the monitor by thread1, since every action in a thread happens-before every subsequent action in the same thread . However, it doesn't mean that `x = 1` can't be reordered within thread1. It can be reordered but from the perspective of thread2 the statement `x = 1` will appear to have been executed in program order (within-thread as-if-serial). By transitivity we can then conclude that the statement `x = 1` and `r2 = x` are ordered by happens-before and `r2` should always be assigned the value 1 when the above execution trace is generated, i.e. when thread1 acquires the lock first.

In contrast, let's examine the scenario when thread2 acquires the lock on the monitor first.

Execution #	Thread 1	Thread 2
1.		lock monitor
2.		r1 = y
3.		unlock monitor
4.		r2 = x
5.	x = 1	
6.	lock monitor	

6.	lock monitor	
7.	y = 5	
8.	unlock monitor	

The first three executions necessarily happen-before the last three executions because the release and acquisition of the same monitor establishes the happens-before relationship. However, there's no happens-before relationship between the statements `r2 = x` and `x = 1`. The following sequence of executions is also valid according to JMM when thread2 acquires the monitor first:

Execution #	Thread 1	Thread 2
1.		lock monitor
2.		r1 = y
3.		unlock monitor
4.	x = 1	
5.		r2 = x
6.	lock monitor	
7.	y = 5	
8.	unlock monitor	

As you can see we have two valid permutations of executions when the monitor is first locked by thread2, that may result in different values for `x`. The reason for this error is that we have failed to establish a happens-before relationship between two conflicting accesses of the same variable in two different threads.

Establishing happens-before relationship

To make sure that the changes done by one thread to shared data are visible immediately to the next thread accessing those same variables, we must establish a *happens-before* relationship between the execution of the two threads. As a developer you can use the following ways to *synchronize* actions with each other. Synchronizing two actions also establish the happens-before relationship between them.

- If an action `A` *happens-before* an action `B` and action `B` *happens-before* action `C` then by transitivity action `A` also *happens-before* action `C`.
- Every action in a thread *happens-before* every subsequent action in that thread. However, for a single-threaded program, instructions can be reordered but the semantics of the program order is still preserved.
- Probably, the most common way to establish *happens-before* relationship is the acquisition and release of a monitor or other locking constructs. An unlock on a monitor *happens-before* or *synchronizes* with every subsequent lock on that same monitor. Note that the `synchronization` block is equivalent of a monitor.
- The read of a volatile variable will fetch the immediately preceding write of the same variable. The write to a volatile variable by any thread synchronizes-with or *happens-before* the reads of the same variable that follow the write in the synchronization order.
- An action that starts a thread *happens-before* or synchronizes-with the first action in the thread it starts. Invoking the `start()` method on a thread object *happens-before* any actions in the started thread.
- The final action in a thread `thread1` synchronizes-with (or *happens-before*) any action in another thread `thread2` that detects that `thread1` has terminated. The methods `thread1.join()` or `thread1.isAlive()` may be used by `thread2` to accomplish this.
- Threads can interrupt other threads by invoking the `interrupt()` method on the thread object. If thread T1 interrupts another thread T2, the interrupt by T1 synchronizes-with or *happens-before* any point where thread T2 or any other thread determines that thread T2 has been interrupted. Thread T2 can determine that it has been interrupted by invoking the method `Thread.interrupted()` or by throwing the `InterruptedException`. Other threads can determine if T2 has been interrupted by invoking the `isInterrupted()` method on T2's thread object.
- Default assignment of values such as `0`, `null` or `false` to variables synchronize with or happens-before the first action in a thread. Consequently the initialization of the object containing these variables will synchronize-with or happens-before the first action in a thread.
- The completion of a constructor for an object *happens-before* the start of the finalizer for that object

Any memory operations which were visible to a thread before exiting a synchronized block are visible to any thread after it enters a synchronized block protected by the same monitor, since all the memory operations happen before the release, and the release happens before the acquire. Exiting a synchronized block causes the cache to be flushed to the main memory so that the writes made by the exiting thread are visible to other threads. Similarly, entering a synchronized block has the effect of invalidating the local processor cache and reloading of variables from the main memory so that the entering thread is able to see the latest values. Additionally, even if the compiler reorders statements that establish a happen-

latest values. Additionally, even if the compiler reorders statements that establish a happen-before relationship, the reordering will still maintain their happens-before relationship.

Example 2

The compiler in the spirit of optimization is free to reorder statements however it must make sure that the outcome of the program is the same as without reordering. The sources of reordering can be numerous. Some examples include:

- If two fields **X** and **Y** are being assigned but don't depend on each other, then the compiler is free to reorder them
- Processors may execute instructions out of order under some circumstances
- Data may be juggled around in the registers, processor cache or the main memory in an order not specified by the program e.g. **Y** can be flushed to main memory before **X**.

Note that all these reorderings may happen behind the scenes in a single-threaded program but the program sees no ill-effects of these reorderings as the JMM guarantees that the outcome of the program would be the same as if these reorderings never happened.

However, when multiple threads are involved then these reorderings take on an altogether different meaning. Without proper synchronization, these same optimizations can wreak havoc and program output would be unpredictable.

The JMM enforces a ***happens-before*** ordering on these actions. When an action A *happens-before* an action B, it implies that A is guaranteed to be ordered before B and visible to B. Consider the below program

```
public class ReorderingExample {

    int x = 3;
    int y = 7;
    int a = 4;
    int b = 9;
    Object lock1 = new Object();
    Object lock2 = new Object();

    public void writerThread() {

        // BLOCK#1
        // The statements in block#1 and block#2 aren't dependent
        // on eachother and the two blocks can be reordered by the
        // compiler
        x = a;

        // BLOCK#2
        // These two writes within block#2 can't be reordered, as
        // they are dependent on eachother. Though this block can
        // be ordered before block#1
        y += y;
        y *= y;

        // BLOCK#3
        // Because this block uses x and y, it can't be placed before
        // the assignments to the two variables, i.e. block#1 and block#2
        synchronized (lock1) {
            x *= x;
            y *= y;
        }

        // BLOCK#4
        // Since this block is also not dependent on block#3, it can be
        // placed before block#3 or block#2. But it can't be placed before
        // block#1, as that would assign a different value to x
        synchronized (lock2) {
            a *= a;
            b *= b;
        }
    }
}
```

Now note that even though all this reordering magic can happen in the background but the notion of *program order* is still maintained i.e. the final outcome is exactly the same as without the ordering. Furthermore, **block#1** will appear to *happen-before* **block#2** even if **block#2** gets executed before. Also note that **block#2** and **block#4** have no ordering dependency on each other.

One can see that there's no partial ordering between **block#1** and **block#2** but there's a partial ordering between **block#1** and **block#3** where **block#3** must come after **block#1**.

The reordering antics are harmless in case of a single threaded program but program outputs quickly go awry when we introduce another thread that shares the data that is being read or written to in the `writerThread` method. Consider the addition of the following method to the previous class.

```
    public void readerThread() {

        a *= 10;

        // BLOCK#4
        // Moved out to readerThread() method from writerThread() method
        synchronized (lock2) {
            a *= a;
            b *= b;
        }
    }
```

Note we moved out **block#4** into the new method `readerThread()`. Say if the `readerThread` runs to completion, it is possible for the `writerThread` to never see the updated value of the

variable **a** as it may never have been flushed out to the main memory, where the `writerThread` would attempt to read from. There's no *happens before* relationship between the two code snippets executed in two different threads!

In our `readerThread()` method if we synchronize on the same lock object as the one we synchronize on in the `writerThread()` method when accessing variable `a` then we would establish a *happens-before* relationship between the statements in the two threads that access variable `a`. Another choice could be to wait for the termination of the `readerThread` in the `writerThread` method using `join()` - a thread termination detection construct that establishes happens-before relationship.

Don't confuse happens-before to mean that one thread executes before the other. All it means is that when `readerThread` releases the monitor, up until that point, whatever shared variables it has manipulated will have their latest values visible to the `writerThread` as soon as the `writerThread` acquires the **same** monitor. If the `writerThread` acquires a different monitor then there's no happens-before relationship and it may or may not see the latest values for the shared variables.

← Back lesson
Reordering Effects

Completed Next →
Blocking Queue | Bounded Buffer | Consumer Producer