The JMM defines a ***partial ordering on all actions within a program***. This might sound like a loaded statement so bear with me as I explain below.

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.

Java Memory Model Details

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 is defined in terms of *actions* which can be any of the following

- read and writes of variables
- locks and unlocks of monitors
- starting and joining of threads

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 tricks are harmless in case of a single threaded program but all hell will break loose 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 here 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!

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. A happens before relationship can be established in the following ways.

- Each action in a thread *happens-before* every action in that thread that comes later in the program's order. However, for a single-threaded program, instructions can be reordered but the semantics of the program order is still preserved.
- An unlock on a monitor *happens-before* every subsequent lock on that same monitor. The `synchronization` block is equivalent of a monitor.
- A write to a volatile field *happens-before* every subsequent read of that same volatile.
- A call to start() on a thread *happens-before* any actions in the started thread.
- All actions in a thread *happen-before* any other thread successfully returns from a join() on that thread.
- The constructor for an object *happens-before* the start of the finalizer for that object
- A thread interrupting another thread *happens-before* the interrupted thread detects it has been interrupted.

This implies that 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.

In our `readerThread` if we synchronize on the same lock object as the one we synchronize on in the `writerThread` then we would establish a *happens-before* relationship between the two threads. Don't confuse it to mean that one thread executes before the other. All it means is that when `readerThread` releases the monitor, up till that point, whatever shared variables it has manipulated will have their latest values visible to the `writerThread` as soon as it acquires the **same** monitor. If it 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