Java Memory Model

Order and visibility of operations are influenced by many factors

- compilers may generate instructions in a different order than the “obvious” one suggested by the source code
- generated code may store variables in registers
- CPU caches may vary the order in which writes to variables are committed to main memory
- processors may execute instructions in parallel or out of order
- JVM can permit actions to appear to execute in different order from the perspective of different threads

These factors can

- prevent a thread from seeing the most up-to-date value for a variable
- cause memory actions in other threads to appear to happen out of order

The JMM

- specifies the minimal guarantees the JVM must make about when writes to variables become visible to other threads called **sequential consistency** using **happens-before** semantics
  sequential consistency —- there is a single order in which operations will happen and each read of a variable will see the last write in the execution order to that variable
  happens-before —- A happens before B means B will see the results of action A with its as-if-serial semantics
- abstracts specific CPUs memory models

Correctly synchronized programs exhibit **sequential consistency** using **happens-before** relations.

The rules of **happens-before**:

- **Program order rule** —- **as-if-serial** semantics —- each action in a thread happens-before every action in that thread that comes later in the program order
- **Monitor lock rule** —- an unlock on a monitor lock happens-before every subsequent lock on the same monitor lock (same for **explicit Locks**)
- **Volatile variables rule** —- a write to a volatile field happens-before every subsequent read of that same field (same for **atomic** variables)
- **Thread start rule** —- a call to **Thread.start** on a thread happens-before every action in the started thread
- **Thread termination rule** —- any action in a thread happens-before any other thread detects that thread has terminated, either by successfully return from **Thread.join** or by **Thread.isAlive** returning false

- **Interruption rule** —- a thread calling **interrupt** on another thread happens-before the interrupted thread detects the interrupt (either by having **InterruptedExceotion** thrown, or invoking **isInterrupted** or **interrupted**)
- **Finalizer rule** —- the end of a constructor for an object happens-before the start of the finalizer for that object
- **Transitivity** —- if **A** happens-before **B**, **B** happens-before **C**, then **A** happens-before **C**

Piggybacking on synchronization, other rules of **happens-before**
Piggybacking —- using an existing happens-before ordering that was created for some other reason to order accesses to a variable not otherwise guarded by a lock.

- technique very sensitive to the order in which statements occur and therefore is very fragile
- should be reserved for squeezing the last drop of performance out of the most performance critical classes (like **ReentrantLock**)
- perfectly reasonable when a class commits to a happens-before ordering between methods as part of specification

Other happens-before orderings:

- **placing an item in a thread-safe collection** happens-before **another thread retreives that item from the collection**
- **counting down a CountDownLatch** happens-before **a thread returns from await on that latch**
- **releasing a permit to a Semaphore** happens-before **acquiring a permit from that same Semaphore**
- **actions taken by the task represented by a Future** happen-before **another thread successfully returns from Future.get**
- **submitting a Runnable or Callable to an Executor** happens-before **the task begins execution**
- **a thread arriving at a CyclicBarrier or Exchanger** happens-before **the other threads are released from that same barrier or exchange point**
- if CyclicBarrier uses a barrier action, **arriving at the barrier** happens-before **the barrier action** which in turn happens-before **threads are released from the barrier**

Publication

With the exception of immutable objects, it is not safe to use an object that has been initialized by another thread unless the publication happens-before the consuming thread uses it.

The happens-before guarantee is a stronger promise of visibility and ordering than made by safe publication.

- when X is safely published from A to B, the safe publication guarantees visibility of the state of X but not of the state of other variables A may have touched
- if A putting X on a queue happens-before B fetches X from that queue, not only does B see X in the state that A left it, but B sees everything A did before the handoff

Thinking in terms of handing off object ownership and publication fits better into most program designs than thinking in terms of visibility of individual memory writes.

Safe publication idioms

- use synchronized method to initialize field (lazy)
- use static field with static initialization —- it's guarded by acquiring a lock (eager)
- use inner class with static initialization —- guarded by lock + initialized when accessed (lazy)
- initialization safety: properly constructed objects are always published safely, no matter how they are published
  for final fields initialization safety prohibits reordering any part of construction with initial load of a reference to that object —- guaranteed visibility
  for any variables that can be reached through a final field initialization sefaty prohibits reordering with operations following the post-condition freeze —- guaranteed visibility