**Threads and Locks**

The Java programming language provides multiple mechanisms for communicating between threads.

1. “synchronized” block or method
2. “volatile” variable
3. “java.util.concurrent” package classes

**Synchronization**

* Synchronization
* Process of preventing *Thread Interference* and *Memory Consistency* *Errors* may arise when threads communicate with each other by sharing access to fields and object references to fields refer to.
* *Thread Interference* describes how errors are introduced when multiple threads access shared data.
* *Memory Consistency* *Errors* describes errors that result from inconsistent views of shared memory.
* Synchronization (synchronized block/method, volatile variable, Thread.start(), and Thread.join()) establishes happens-before relationship on memory operations such as reads and writes of shared variables.
* Process
* Processes are often seen as synonymous with programs or applications.
* A process has self-contained execution environment, private set of run-time resources and own memory space.
* Thread
* Threads are called lightweight processes; share the run-time resources of memory and hardware resources of a process.
* Execution of program unit is called thread.
* Threads exist within a process; every process has at least one thread.
* Semaphore
* A signaling mechanism (I’m done; you can carry on kind of signal) interrupts current execution and transfers control to another thread.
* Mutex/Lock
* A locking mechanism used to synchronize access to a resource. Simply, mutex is a lock.
* Locks offer two primary features: *mutual exclusion* and *visibility*.
* Mutual exclusion
* Only one thread at a time can hold a given lock/mutex.
* Coordinates access to shared data such that only one thread at a time will be using the shared data.
* Visibility
* Ensures that changes made to shared data prior to releasing a lock are made available to another thread that subsequently acquires that lock.
* Monitor
* Each object in Java is associated with a monitor, which a thread can lock/unlock.
* Only one thread at a time may hold a lock on monitor.
* Wait set
* Set of threads waiting for an object after wait/join action.
* When an object is first created, a wait set is empty.
* Atomicity
* An operation appears to occur at a single instant between its invocation and response.
* Happens-before relationship

1. **Synchronized Block or Method**
2. **Overview**

* Data synchronization normally happens at the start and the end of synchronization block or method.

1. **Detail**
2. **Reads and writes of volatile variable**
3. **Overview**

* Volatile variables share the *visibility* features of *synchronized*, but none of the atomicity features.
* Threads will always see the most up-to-date value for the volatile variables.
* Prefer to use volatile variables instead of locks for two reasons: simplicity and scalability.
* Unlike locks, volatile variable does not cause a thread to block.
* Volatile can be applied to variables (with null or non-null reference).

1. **Conditions for correct use of volatile**

We can use volatile instead of locks only when both of the following conditions satisfy:

1. Writes to the variable do not depend on current value.
2. The variable doe not participate in invariants with other variables.

* Use volatile if a variable is truly independent of both other variables and its own prior values.
* Apply volatile correctly for: Status flag, one-time safe publication, independent observations, and cheap read-write lock.

1. **Use of classes in the java.util.concurrent package**
2. **Brief Description**
3. **Detail**
4. **Summary**