# Java Concurrency Practice

# 2.Thread Safety

* Don’t share the state variable across threads.
* make state variable immutable or
* use synchronization whenever accessing the state variable.

## 2.3 locking

* To preserve state consistency, update related state variables in a single atomic operation.

### Intrinsic lock or monitor lock act as mutexs

* lock are acquired by per thread rather than per invocation( reentrancy)

## 2.4 Guarding state with locks

* For each mutable state variable that may be accessed by more than one thread, all access that variable must be performed with **same** lock held, in this case we sat that the variable is guarded by that lock.
* Every shared, mutable variable should be guarded by exactly one lock.
* For every invariant that involves more than one variable, all the variables involved in that invariant must be guarded by same lock.

# 3.Sharing object

## 3.1 Visibility

* Lock can guarantee both visibility and atomicity; violate variable can only guarantee visibility

## 3.2 Publication and escape

* publishing an object means make it available to code outside of its current scope.
* an object that is published when it should not been seen is said to have escaped
* don’t allow the this reference escape during construction
* run a thread in constructor
* call no final/private function within constructor

## 3.3 Thread confinement

* stack confinement
* thread-local confinement

## 3.4 Immutability

* Immutable object are always thread safe
* Immutable object:
  + object state can not be modified after constructed
  + all field are final
  + it is property constructed
* It is a good practice that declare all fields private and final
* final field has special meaning of JMM

## 3.5 safe publication

* immutable object(final fields) can be used safety by any thread without additional synchronization even when synchronization is not used for published.
* safe publication:
  + static initialization
  + use violate and AtomicReference
  + use final field
  + use lock
* safe published effective immutable object can be used safely without additional synchronization
* share objects in concurrency environment
  + thread confine
  + shared read-only: immutable and effective immutable
  + shared thread-safe

# 4. Composing Object

* Confinement makes it easier to build thread-safe class because a class that confines its state can analyzed for thread safety without having examine the whole program.
* check-then-act sequences need to atomic
* if a class is composed of multiple independent thread-safe state variables and has no operations that have any invalid state transitions, then it can delegate thread safety to underlying state variables.
* using ctor(p.get()) instead of ctor(x, y), to prevent from race condition at ctor.

# 5. Building Blocks

* synchronized collections may throw ConcurrentModificationException
* Concurrent collections.
* Bounded queues are a powerful resource management tool for building reliable applications. Comparing others like map, set, it also can block thread run in some situation
* Synchronizers: latches and barriers, latches wait for an event, barriers wail for all threads.
* Using concurrency collection and FutureTask for high performance cache

# Summary of Part I:

* It’s the mutable state, stupid.
* Make fields final unless they need to be mutable
* Immutable object are automatically thread-safe
* Encapsulation makes it practical to manage the complexity
* Guard each mutable variable with a lock.
* Guard all variables in an invariant with the same lock
* Hold locks for the duration of compound actions
* A program that access a mutable variable from multiple threads without synchronization is a broken program.

# 6 Task Execution

* Whenever you see

new Thread(runnable).start()

please consider replacing it with Executor

# 7 Task cancellation

* Calling interrupt does not necessarily stop the target thread from doing what it is doing; it merely delivers the message that interruption has been requested.
* interruption is usually the most sensible way to implement cancellation
* don’t swallow interruption requests.
* Daemon threads
* poison pill
* Avoid finalizers
* java don’t provide a preemptive mechanism for cancelling activities or terminating threads

# 10 Avoiding Liveness Hazards

* dead lock
* using thread dump to identify/analysis deadlock
* Avoid the temptation to use thread priorities. Most concurrent applications can use the default priority for all threads
* Lock striping

# 13 Explicit Locks

* lock and ReentrantLock

lock.lock();

try {

} finally {

lock.unlock();

}

* notify and notifyAll

# 14 Building Custom Synchronizers

* Object.wait automically releases the lock and suspend the current trhead

acquire lock on object state while (precondition does not hold) {

release lock wait until precondition might hold optionally fail if interrupted or timeout expires reacquire lock

} perform action release lock

void stateDependentMethod() throws InterruptedException {

// condition predicate must be guarded by lock

synchronized(lock) { while (!conditionPredicate())

} }

lock.wait();

// object is now in desired state

* Use notifyAll() instead of notify();

##### public synchronized void open() {

##### …

##### notifyAll();

##### }

* Condition has await, signal, signalAll

# 15 Atomic Variables and Nonblocking

* Compare and swap(CAS)
* Use CAS instead of lock, which can improve performance
* Semaphores
  + Maintain count of the number of threads allowed to pass
* Latches
  + Boolean conditions that are set once, ever
* Barriers
  + Counters that cause all threads to wait until all have finished
* Reentrant Locks
  + Java-style locks allowing multiple acquisition by same thread, but that may be acquired and released as need
* Mutexes
  + Non-reentrant locks
* Read/Write Locks
  + Paris of conditions in which the readLock may be shared, but the writeLock is exclusive