Parallel and Concurrent Programming with Java

## Chapter 4. Locks

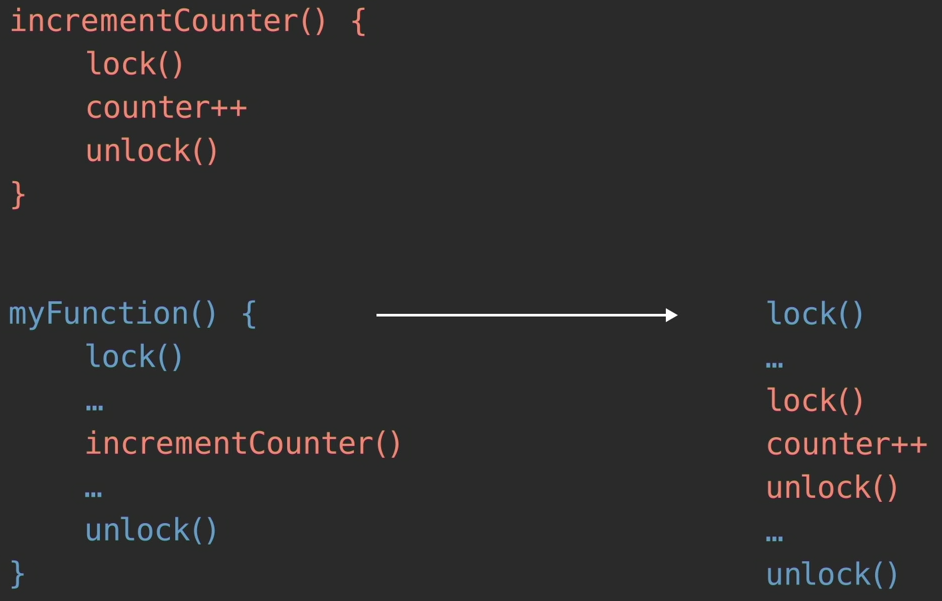
# Reentrant Locks

A reentrant lock allows a thread to re-enter into same lock multiple times without a deadlock.

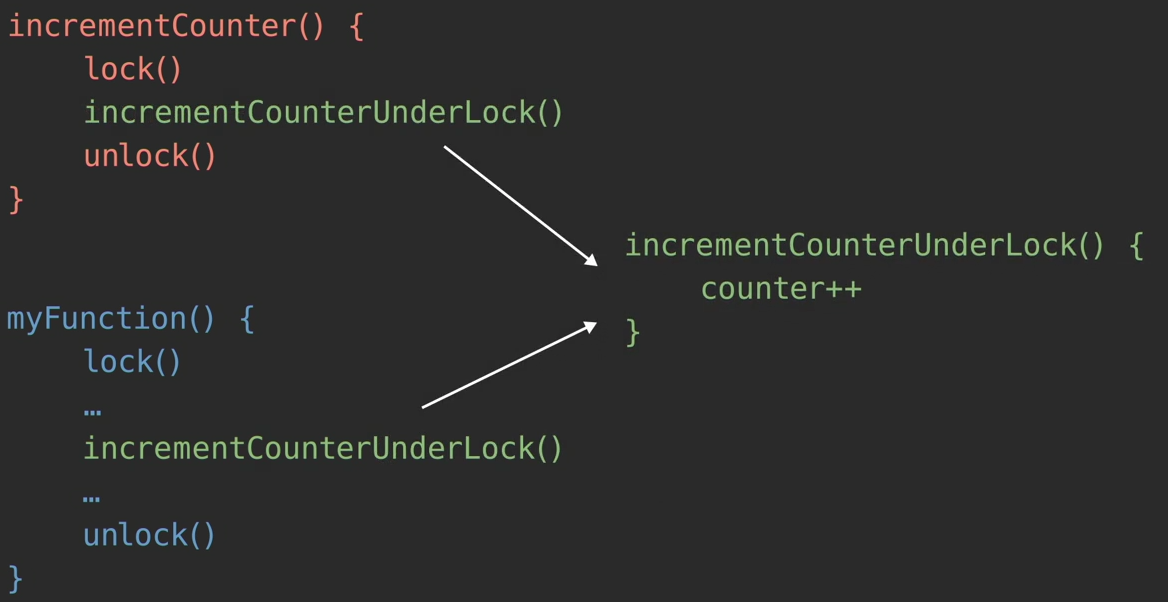
* Deadlock
* when a thread that has acquired a mutex lock waits to acquire the same lock again.
  + other active threads cannot acquire lock too.
  + All threads blocked, cannot continue executing.
* Behavior similar to implicit monitor lock accessed using synchronized methods and statements, i.e.
  + only 1 thread can acquire lock
  + only 1 thread can access shared resources
  + mutually exclusive to all other threads
* But with more features, e.g. a waiting queue, optional fairness parameter so that no treads are starved of resources for long time.
* must be unlocked as many times as it was locked
  + reentrant lock keeps track of number of times (count) current thread has hold on lock
  + if active thread finished executing critical section or blocked, and reentrant lock not unlocked correct number of times, all threads remain deadlocked.

Helpful for **nested** codes.

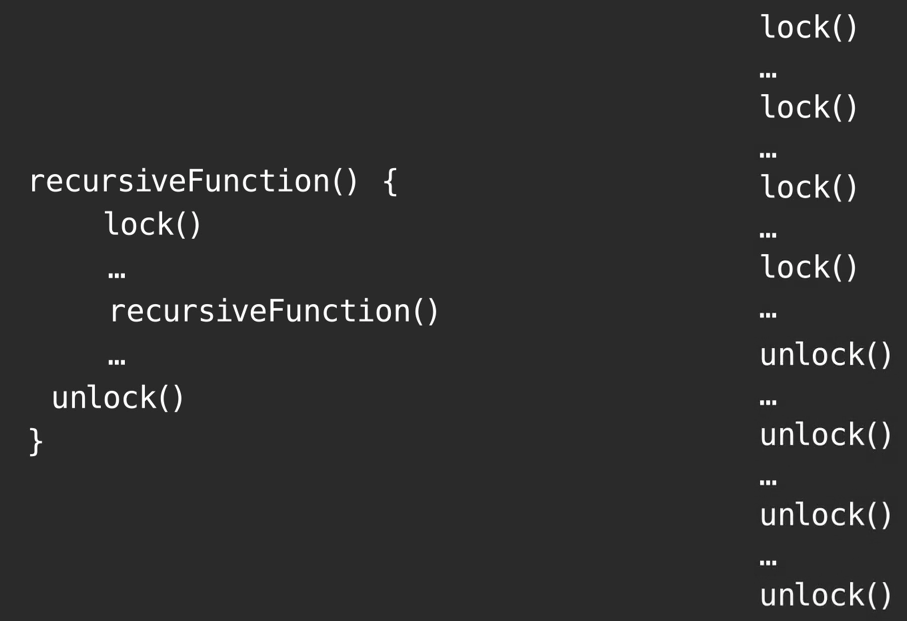
* + IncrementCount() -> lock() -> nested myFunction() -> lock() again.
  + If it is a normal lock, thread will be deadlocked/blocked for 2nd lock() attempt.



* + Avoid nested locks by having counter within function incrementCounterUnderLock(). Function itself is only called within critical section.



Needed for **recursive** functions.



Can configure for fairness policy.

* Constructor for class has optional fair ordering parameter.
  + When true, lock favors granting access to longest waiting tread.
* Program with many threads and fair locks
  + may exhibit lower throughput, i.e. run slower.
  + Less variation in access times to locks
  + guarantee no thread starvation
* does not guarantee fairness in thread scheduling by OS
  + a thread may acquire fair lock multiple times in succession while other active threads are not progressing
* untimed tryLock method ignore fairness setting

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/locks/ReentrantLock.html

Why / When use reentrant locks?

* When need one of the features that cannot be done with synchrnized methods/statements.
* Example:
  + non-block code structure
  + fair locks
  + tryLock within specified waiting time
  + poll/monitor for lock, queued threads, fairness status
  + allows for current thread interruption
  + multiple condition variables

Similar terminology

* Reentrant mutex
* Reentrant lock
* Recursive mutex
* Recursive lock

*src*/ReentrantLockDemo.java

# tryLock

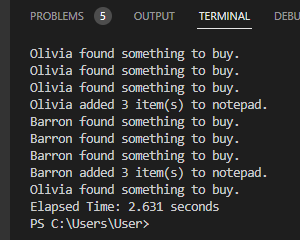
In a program with multiple threads and multiple tasks, not efficient to have threads attempt to acquire a lock and be blocked (in waiting state) frequently.

Non-blocking lock.

3 possible outcomes for tryLock():

* Lock available. Acquires the lock, return immediately with value TRUE. Set lock count to 1.
* Lock held by current thread. Immediately returns TRUE. Increment lock count by 1.
* Lock held by another thread. Immediately returns FALSE.
  + Can code If (lock.tryLock == TRUE) {... do this} else {... do something else ...}

*/src*/TryLockDemo.java

With tryLock, can code for thread to run “else” block when reentrant lock is unavailable (being locked by another thread). Better performance.

Read-write lock

Recap:

Mutex lock

* protect critical section of code
* prevent data races
  + when multiple threads concurrently accessing same shared memory area
  + at least 1 thread writing to it.
* Only **ONE** thread has exclusive access, regardless whether is reading, writing, waiting/sleeping.

Class x {

private static Lock pencil = new Lock();

// ....

public void run() {

pencil.lock();

try {

// method body

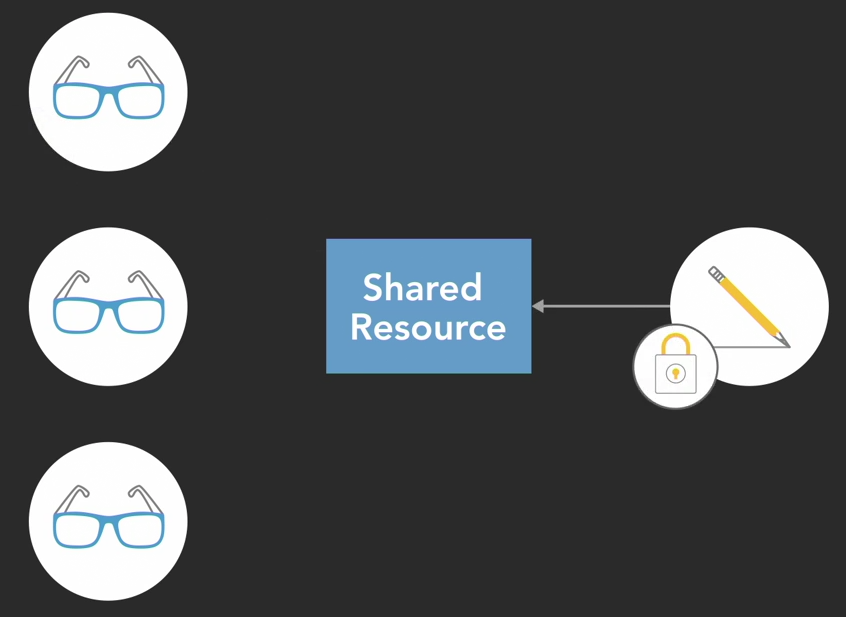
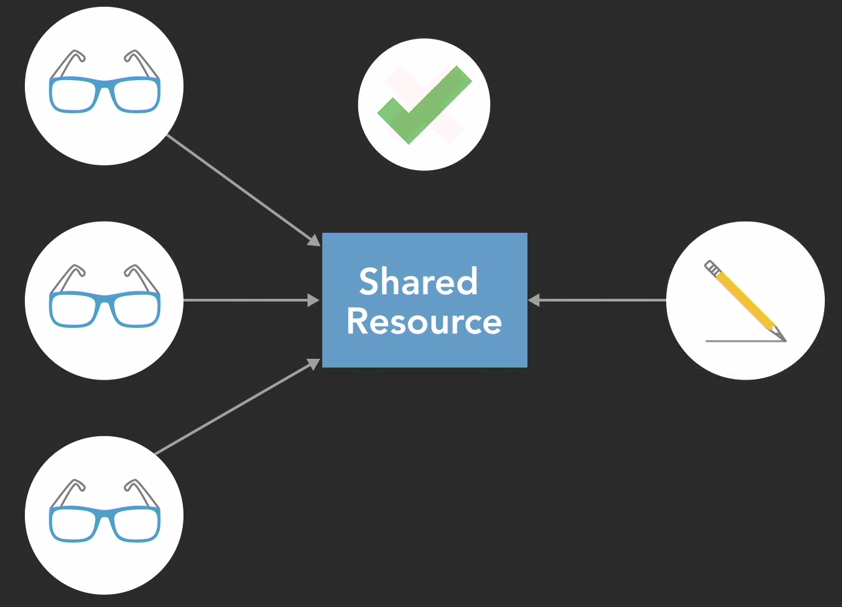
} finally {

pencil.unlock();

}

Access to share resources involve 3 operations: read, modify, write.

* OK for multiple threads to read shared memory location. See same data.
* If a thread is modifying / writing, must limit access to this single thread.



A read-write lock has 2 modes

* Shared Read mode – multiple threads can read at same time. Read locks can be held simultaneously by multiple reading threads. No threads can modify/write to shared resource, as long as a single thread is still holding a read lock.
* Exclusive Write mode – only 1 thread has exclusive access/hold and is writing. No other threads can have a read lock or read the data being modified. No other threads can write either.

Implementation – when to use read-write lock?

=> A data collection that will be read often and modified infrequently.

* e.g. dictionary, catalog, pre-populated database.
* Read-write lock can improve performance over basic mutex lock.
* Conditions needed:
  + # read threads > # write threads.
    - concurrency of read threads
    - when data modified, it is exclusively locked by 1 thread only.
  + Duration of read operations > Duration of write operations.
    - Read-write lock more complex than basic mutex lock.
    - Overhead of read-write lock implementation higher than for basic mutex lock. More resources used to keep track of multiple readers.
    - If read durations short, significant % of time and resources spent implementing locks -> high execution cost.

Different programming languages may have different policies.

* Example: if read threads and write threads are waiting, to prefer read or write threads?
  + Writes preferred as they are expected to be short-lived and infrequent. Subsequent reads will be of updated data.
  + If reads preferred, write thread may be blocked for long time because reads are long and frequent.
* Affect performance.

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/locks/ReadWriteLock.html

Read-write lock: Java Demo

*/src*/ReadWriteLockDemo.java

10 read threads vs 2 write threads.