

53% completed

Search Course

Java Concurrency Reference

- Setting-up Threads
- Basic Thread Handling
- Executor Framework
- Executor Implementations
- Thread Pools
- Types of Thread Pools
- An Example: Timer vs ScheduledThreadPool
- ThreadPoolExecutor
- Callable Interface
- Future Interface
- CompletionService Interface
- ThreadLocal
- ThreadLocalRandom
- CountDownLatch
- CyclicBarrier
- Concurrent Collections
- ConcurrentHashMap
- ConcurrentModificationException

Practice Mock Interview

Java Multithreading for Senior Engineering Interviews / ... / Lock Interface

# Lock Interface

The Lock interface explained with examples.

If you are interviewing, consider buying our number#1 course for [Java Multithreading Interviews](#).

## Explanation

The `Lock` interface provides a tool for implementing mutual exclusion that is more flexible and capable than `synchronized` methods and statements. A single thread is allowed to acquire the lock and gain access to a shared resource, however, some implementing classes such as the `ReentrantReadWriteLock` allow multiple threads concurrent access to shared resource. The use of `synchronized` methods or statements provides access to the implicit monitor lock associated with every object, but requires all lock acquisitions and releases to proceed in a block-structured way. Locks acquired in a nested fashion must be released in the exact opposite order, and all locks must be released in the same lexical scope in which they were acquired. These requirements restrict how `synchronized` methods and statements can be used and `Lock` implementations can be used for more complicated use-cases.

The `Lock` interface has the following classes implementing it:

- `ReentrantLock`
- `ReentrantReadWriteLock.ReadLock`
- `ReentrantReadWriteLock.WriteLock`

## Difference between Lock and Synchronized

If you have worked with `synchronized` you may be wondering why we need the `Lock` interface and its implementing classes. The answer - locks offer additional functionality and far more flexibility in usage than synchronized methods and statements. For instance:

- A `Lock` can be tested for acquisition in a non-blocking fashion using the `tryLock()` method
- A `Lock` can be waited upon for acquisition with a specified timeout using the `tryLock(timeout)` method. After the timeout the thread abandons its attempt to acquire the lock and moves-on.
- A `Lock` can be waited upon for acquisition with the option to interrupt the acquiring thread using the `lockInterruptibly` method.
- Some `Lock` implementations also provide monitoring and deadlock detection. Additionally, `Lock` implementation can provide fair-use mode for locks, guaranteed ordering and non-reentrant use.

The flexibility and functionality of `Lock` implementations come at the cost of higher chance of human error since the locks are not automatically released as is the case with `synchronized` blocks and statements. The developer must remember to unlock the `Lock` in a `finally` block and as many times as the lock has been acquired for in case reentrancy is supported. The idiomatic use of any `Lock` implementation follows the below pattern:

```
Lock ourLock = // ... instantiate a Lock

ourLock.lock();
try {
    // ... Perform operations
} finally {
    ourLock.unlock();
}
```

## Using Lock and synchronized

Finally, since `Lock` implementation is also an object it can be used as the argument to `synchronized` statement but such use is discouraged other than for internal implementation of the class. For example, the following is a bad practice:

```
Lock ourLock = // ... instantiate a Lock

synchronized(ourLock){
    // ... Not a good idea
}
```

Back lesson

Mark As Completed

Next

ConcurrentModificationException

LockSupport