Lock

-synchronized:

[Multi-threaded](https://www.geeksforgeeks.org/multithreading-in-java/)programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.

So it needs to be made sure by some synchronization method that only one thread can access the resource at a given point in time. Java provides a way of creating threads and synchronizing their tasks using synchronized blocks. Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block in Java is synchronized on some object. All synchronized blocks synchronize on the same object can only have one thread executing inside them at a time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

-lock:

Lock implementations provide more extensive locking operations than can be obtained using synchronized methods and statements. They allow more flexible structuring, may have quite different properties, and may support multiple associated [Condition](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Condition.html) objects.

A lock is a tool for controlling access to a shared resource by multiple threads. Commonly, a lock provides exclusive access to a shared resource: only one thread at a time can acquire the lock and all access to the shared resource requires that the lock be acquired first. However, some locks may allow concurrent access to a shared resource, such as the read lock of a [ReadWriteLock](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReadWriteLock.html" \o "interface in java.util.concurrent.locks).

The use of synchronized methods or statements provides access to the implicit monitor lock associated with every object, but forces all lock acquisition and release to occur in a block-structured way: when multiple locks are acquired they must be released in the opposite order, and all locks must be released in the same lexical scope in which they were acquired.

While the scoping mechanism for synchronized methods and statements makes it much easier to program with monitor locks, and helps avoid many common programming errors involving locks, there are occasions where you need to work with locks in a more flexible way. For example, some algorithms for traversing concurrently accessed data structures require the use of "hand-over-hand" or "chain locking": you acquire the lock of node A, then node B, then release A and acquire C, then release B and acquire D and so on. Implementations of the Lock interface enable the use of such techniques by allowing a lock to be acquired and released in different scopes, and allowing multiple locks to be acquired and released in any order.

With this increased flexibility comes additional responsibility. The absence of block-structured locking removes the automatic release of locks that occurs with synchronized methods and statements. In most cases, the following idiom should be used:

Lock l = ...;

l.lock();

try {

// access the resource protected by this lock

} finally {

l.unlock();

}

Lock interface

* lock(). unlock(), newCondition(). tryLock(), lockInterruptibly();

[https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html](Lock%20interface%20lock().%20unlock(),%20newCondition().%20tryLock(),%20lockInterruptibly()%3B)

* ReentrantLock class

<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html>

[https://www.geeksforgeeks.org/reentrant-lock-java/https://www.geeksforgeeks.org/reentrant-lock-java/](https://www.geeksforgeeks.org/reentrant-lock-java/)

ReadWriteLock interface

* method
  + Lock readLock()
  + Lock writeLock()

<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReadWriteLock.html>

<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.ReadLock.html>

Future vs CompletableFuturex

<https://stackoverflow.com/a/35347215/12007527>