Explicit Locks

Why Explicit Locks?

Before Java 5

- Only **synchronized** and **volatile** available.
- Limitations:
  - Cannot interrupt threads waiting on intrinsic locks
  - Cannot try to acquire lock without blocking forever
  - No non-block-structured locking

Java 5+

- Introduced **ReentrantLock** with:
  - Interruptible locking
  - Timed + polled attempts
  - Fair vs non-fair acquisition
  - Flexible locking patterns

The Lock Interface

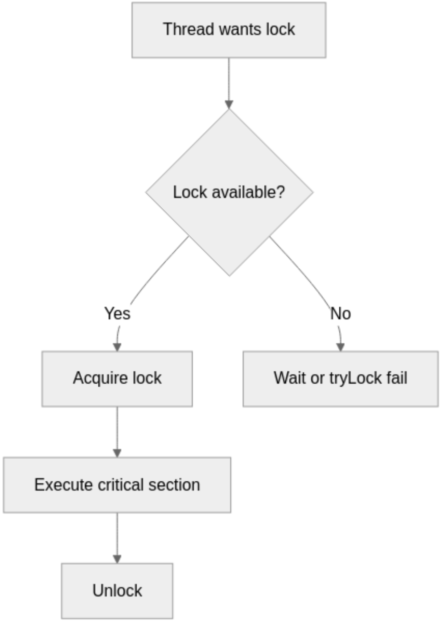
```java
public interface Lock {
    void lock();
    void lockInterruptibly() throws InterruptedException;
    boolean tryLock();
    boolean tryLock(long timeout, TimeUnit unit)
        throws InterruptedException;
    void unlock();
    Condition newCondition();
}
```

Canonical Lock Usage

```java
Lock lock = new ReentrantLock();
lock.lock();
try {
    // update object state
} finally {
    lock.unlock();
}
```

- Must release in `finally`
- Forgetting to unlock can be dangerous

Basic Locking



Timed & Polled Lock Acquisition

- Helps avoid deadlock
- Enables probabilistic deadlock avoidance
- Supports time-budgeted tasks
- Example:

```java
if (lock.tryLock(10, TimeUnit.MILLISECONDS)) {
    try { ... }
    finally { lock.unlock(); }
} else {
    // alternate path
}
```

Example: Deadlock Avoidance

```
if (from.lock.tryLock()) {
    try {
        if (to.lock.tryLock()) {
            try {
                // transfer
            } finally { to.lock.unlock(); }
        }
    } finally { from.lock.unlock(); }
}
```

Interruptible Lock Acquisition

```
lock.lockInterruptibly();
try {
    return send(message);
} finally {
    lock.unlock();
}
```

- Allows cancellation-friendly locking

Non-block-structured Locking

- Intrinsic locks always release on block exit
- ReentrantLock allows flexible patterns
- Useful in hand-over-hand locking

Performance Considerations

- Java 5: ReentrantLock faster
- Java 6+: intrinsic locks improved
- Performance evolves; no guarantee

Fairness Options

- Fair locks: FIFO, prevent barging
- Non-fair locks: better throughput
- Fair locks may be $100\times$ slower

Synchronized vs ReentrantLock

Feature	synchronized	ReentrantLock
Compact syntax	yes	no
Auto release	yes	no
Interruptible	no	yes
Timed lock try	no	yes
Fairness	no	yes
JVM debugging	yes	Improved

- Note: Use ReentrantLock only when needed.

Read-Write Locks

```
public interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}
```

- Allows multiple readers, single writer

ReadWriteLock Behavior

- Reader barging options
- Writer preference options
- Reentrancy
- Downgrading allowed; upgrading dangerous

Example: ReadWriteMap

```java
public class ReadWriteMap<K,V> {
    private final Map<K,V> map;
    private final ReadWriteLock lock =
        new ReentrantReadWriteLock();
    private final Lock r = lock.readLock();
    private final Lock w = lock.writeLock();
    public V put(K key, V value) {
        w.lock();
        try { return map.put(key, value); }
        finally { w.unlock(); }
    }
    public V get(Object key) {
        r.lock();
        try { return map.get(key); }
        finally { r.unlock(); }
    }
}
```

Summary

- ReentrantLock adds advanced features
- Not a replacement for synchronized
- Read-write locks boost read-heavy performance
- Performance varies by JVM version