Complete Guide to Java Locks and Synchronization

Table of Contents

---

1. Introduction

Java provides multiple mechanisms to achieve **thread safety** and prevent **data races**.
While the `synchronized` keyword is simple, it lacks advanced features like **timeouts**, **fairness policies**, or **multiple conditions**.
The `java.util.concurrent.locks` package introduces a **flexible Lock API** that gives developers fine-grained control over synchronization.

---

2. The Need for Locks

⚡ The Race Condition Problem

When two or more threads **read and write** shared data without coordination, inconsistent results occur.

Example Scenario
Two threads withdraw from the same account:

```
Thread 1: Read balance = 100 → Calculate (100 - 50) → Write (50)
Thread 2: Read balance = 100 → Calculate (100 - 50) → Write (50)
Result: Balance = 50 (expected 0)
```

Traditional Synchronization (`synchronized`)

Java's `synchronized` keyword provides **mutual exclusion**, ensuring only one thread enters a critical section at a time.

Limitations

Feature	synchronized	ReentrantLock
Automatic release	✅ Yes	❌ Must unlock manually
Timeout support	❌ No	✅ `tryLock`
Interruptible	❌ No	✅ `lockInterruptibly`
Fair/Unfair policy	❌ Always JVM decided	✅ Configurable
Condition objects	Limited (wait/notify)	✅ Multiple conditions

3. Lock Interface and `ReentrantLock`

The `Lock` interface defines the basic contract:

java — Copy code

```java
public interface Lock {
    void lock();                    // Acquire lock (blocks if necessary)
    void unlock();                  // Release lock
    boolean tryLock();              // Attempt without waiting
    boolean tryLock(long time, TimeUnit unit); // Try with timeout
    void lockInterruptibly() throws InterruptedException; // Respond to interruption
    Condition newCondition();       // Create condition for advanced signaling
}
```

`ReentrantLock`

A concrete implementation that allows:

- **Reentrancy**: Same thread can acquire the lock multiple times.

- **Fairness Policy**: Threads acquire locks in FIFO order if requested.
- **Timeouts** and **Interruptible Locking**.

---

4. Key Features of `ReentrantLock`

4.1 Reentrancy

A thread can **lock the same lock repeatedly** without deadlocking itself.

This is tracked using a **hold count**.

java Copy code

```java
class ReentrantExample {
    private final ReentrantLock lock = new ReentrantLock();
    public void outerMethod() {
        lock.lock(); // hold count: 1
        try {
            innerMethod();
        } finally {
            lock.unlock(); // hold count: 0 (fully released)
        }
    }
    private void innerMethod() {
        lock.lock(); // hold count: 2
        try {
            System.out.println("Inner method");
        } finally {
            lock.unlock(); // hold count: 1
        }
    }
}
```

Key Point: Each call to `lock()` increments the hold count; each `unlock()` decrements it.

4.2 Timeout Support

The `tryLock()` method lets a thread attempt to acquire a lock for a **limited time**, avoiding indefinite waiting.

java Copy code

```java
if (lock.tryLock(1, TimeUnit.SECONDS)) {
    try {
        // Critical section
    } finally {
        lock.unlock();
    }
} else {
    System.out.println("Could not acquire lock");
}
```

4.3 Interruptible Lock Acquisition

Using `lockInterruptibly()`, a waiting thread can be interrupted:

java Copy code

```java
lock.lockInterruptibly();
```

5. `ReadWriteLock`

Allows **multiple concurrent readers** but ensures **exclusive writers**.

java Copy code

```java
class ReadWriteCounter {
    private final ReentrantReadWriteLock rwLock = new ReentrantReadWriteLock();
    private int count = 0;

    public int getCount() {
        rwLock.readLock().lock();
        try { return count; }
        finally { rwLock.readLock().unlock(); }
    }

    public void increment() {
        rwLock.writeLock().lock();
        try { count++; }
        finally { rwLock.writeLock().unlock(); }
    }
}
```

Performance Tip:
Use when **reads >> writes** (e.g., caching systems, analytics dashboards).

---

6. Fair vs Unfair Locks

- **Fair Lock (`new ReentrantLock(true)`)**: Threads acquire locks in **FIFO order**. Prevents starvation but slightly slower.
- **Unfair Lock (default)**: Threads may **jump the queue** for better throughput.

```java
ReentrantLock fairLock = new ReentrantLock(true);
```

7. Advanced Lock Types

`StampedLock`

Introduced in Java 8, provides:

- **Optimistic Reads**: Read without blocking writers (verify if write occurred).
- **Better performance** for read-heavy workloads.

Example:

```java
long stamp = lock.tryOptimisticRead();
int current = value;
if (!lock.validate(stamp)) {  // Writer modified
    stamp = lock.readLock();
    try { current = value; }
    finally { lock.unlockRead(stamp); }
}
```

8. Best Practices

✅ Always unlock in a `finally` block:

```java
lock.lock();
try {
    // work
} finally {
    lock.unlock();
}
```

✅ Prefer `tryLock()` for deadlock prevention.

✅ Use `ReadWriteLock` for **read-dominant** workloads.

✅ Acquire locks in a **consistent order** to avoid deadlocks.

9. Code Walkthroughs

9.1 BankAccount – Safe Withdrawals

java Copy code

```java
class BankAccount {
    private int balance = 100;
    private final Lock lock = new ReentrantLock();

    public void withdraw(int amount) {
        try {
            if (lock.tryLock(1, TimeUnit.SECONDS)) {
                try {
                    if (balance >= amount) {
                        Thread.sleep(100);
                        balance -= amount;
                        System.out.println(Thread.currentThread().getName() +
                            " withdrew " + amount + ", balance: " + balance);
                    }
                } finally {
                    lock.unlock();
                }
            } else {
                System.out.println(Thread.currentThread().getName() +
                    " could not acquire lock");
            }
        } catch (InterruptedException e) { e.printStackTrace(); }
    }
}
```

9.2 Deadlock Demonstration

java Copy code

```java
Lock lock1 = new ReentrantLock();
Lock lock2 = new ReentrantLock();

Runnable task1 = () -> {
    lock1.lock();
    try {
        Thread.sleep(50);
        lock2.lock(); // Potential deadlock!
        try { /* work */ }
        finally { lock2.unlock(); }
    } catch (InterruptedException e) {}
    finally { lock1.unlock(); }
};
```

💡 Fix: Always acquire locks in the **same order**.

💡 Fix: Always acquire locks in the same order.

10. Performance Considerations

Scenario	Best Choice	Reason
Simple low-contention lock	`synchronized`	JVM optimizations (biased locking)
High contention	`ReentrantLock`	Timeout & fairness options
Read-heavy workload	`ReadWriteLock` / `StampedLock`	Multiple readers

11. Common Pitfalls

❌ Forgetting to unlock

❌ Mixing readLock for write operations

❌ Using multiple locks with inconsistent ordering

12. Conclusion

The Java Lock framework provides powerful, fine-grained control for multithreaded programming:

Use ReentrantLock when you need timeouts, interruptible acquisition, or fairness policies.

Use ReadWriteLock for read-heavy workloads.

Always follow best practices: try-finally unlocking, consistent lock ordering, and timeout strategies.

Proper use of these mechanisms ensures safe, efficient, and deadlock-free concurrent applications.