Making a Variable Thread-Safe in Java

1 Why Thread Safety Matters

A *thread-safe* variable guarantees correct, predictable results even when multiple threads read or modify it concurrently. Race conditions, lost updates, and visibility issues disappear when the variable is properly protected.

2 Techniques to Make a Variable Thread-Safe

2.1 `synchronized`

```java
private int count = 0;

public synchronized void increment() {
    count++;             // atomic within the intrinsic lock of this object
}
```

Locks on either the instance (`this`) or a custom monitor object, ensuring **mutual exclusion**.

🔒What is Mutual Exclusion?

Mutual exclusion means that **only one thread at a time can access a critical section of code** (like modifying a shared variable). It prevents race conditions by blocking other threads until the first thread finishes its task.

Example:

```java
private int balance = 1000;

public synchronized void withdraw(int amount) {
    if (balance >= amount) {
        balance -= amount; // Only one thread can do this at a time
    }
}
```

In this example, the `synchronized` keyword ensures that no two threads can execute the `withdraw()` method simultaneously.

2.2 `volatile`

🔍What is Atomicity?

Atomicity means that an operation is performed as a single, indivisible step. Either it completes fully, or it doesn't happen at all — there is no visible intermediate state.

In multithreaded programming, atomicity ensures that no other thread can observe the operation in a half-done state.

Example (non-atomic increment):

```
private int counter = 0;

public void increment() {
    counter++; // NOT atomic: it's actually 3 steps (read, modify, write)
}
```

Even if `counter` is declared `volatile`, the above operation is not atomic.

To make it atomic, use:

```
AtomicInteger counter = new AtomicInteger();
counter.incrementAndGet(); // Atomic and thread-safe
```

```
private volatile boolean flag = false;
```

```
private volatile boolean flag = false;
```

Guarantees **visibility** of the latest value across threads, but *not* atomicity—use only for simple read-write flags.

2.3 Atomic Classes

```
AtomicInteger counter = new AtomicInteger();
int newVal = counter.incrementAndGet(); // atomic increment
```

High-performance, lock-free operations for numbers, booleans, references, and arrays.

2.4 Thread-Safe Collections

```
Map<String, String> map = new ConcurrentHashMap<>();
List<String> list = new CopyOnWriteArrayList<>();
```

Implementations in `java.util.concurrent` scale better than `Collections.synchronizedXxx()` wrappers.

2.5 Explicit Locks (`ReentrantLock`, `ReadWriteLock`, etc.)

Provide greater flexibility—try-lock, timed lock, interruptible lock, and condition variables—than the intrinsic monitor used by `synchronized`.

---

3 What Is a *Reentrant* Lock?

A **reentrant lock** allows **the same thread** to acquire the lock repeatedly without deadlocking. Each `lock()` increments an internal hold count; each `unlock()` decrements it.

3.1 Why Use `try` / `finally` with `ReentrantLock`

Ensures that the lock is **always released**, even if an exception exits the critical section prematurely.

```
private final ReentrantLock lock = new ReentrantLock();

public void update() {
    lock.lock();
    try {
        // critical section—safe access/modification
    } finally {
        lock.unlock();
    }
}
```

---

4 Fair vs Unfair Locks

Aspect	**Fair Lock** (`new ReentrantLock(true)`)	**Unfair Lock** (`new ReentrantLock()` or `false`)
Acquisition order	Strict FIFO: longest-waiting thread wins	“Barging” allowed: current requests may skip queue

Aspect	**Fair Lock** (`new ReentrantLock(true)`)	**Unfair Lock** (`new ReentrantLock()` or `false`)
Starvation risk	Very low	Possible for long-waiting threads
Throughput	Slightly lower (more context switches)	Higher overall
Latency jitter	Predictable & uniform	Variable

4.1 Code Examples

```
// Fair lock—threads granted access in arrival order
ReentrantLock fairLock = new ReentrantLock(true);

// Unfair lock (default)—higher throughput, possible starvation
ReentrantLock unfairLock = new ReentrantLock(); // or new ReentrantLock(false)
```

---

5 Quick Reference Cheat Sheet

- **Small flag?** → `volatile` (visibility only)
- **Numeric counter?** → `AtomicInteger`, `LongAdder` (high contention)
- **Multiple statements / complex updates?** → `synchronized` *or* `ReentrantLock`
- **High-read concurrent map?** → `ConcurrentHashMap`
- **Fairness required?** → `new ReentrantLock(true)`

Further Reading

- *Java Concurrency in Practice* — Brian Goetz et al.
- *The Art of Multiprocessor Programming* — Herlihy & Shavit

© 2025