Synchronization vs. Locking in Java

When to Use Synchronization?

Synchronization ensures thread safety when multiple threads access shared resources. It is useful in the following scenarios:

1. Modifying Shared Data: When multiple threads modify a shared variable or object.
2. Ensuring Atomicity: Operations like check-then-act need to be atomic.
3. Maintaining Consistency: Shared resources should remain in a consistent state.
4. Avoiding Race Conditions: Prevents two threads from executing critical sections simultaneously.
5. Thread Communication: Used with wait/notify mechanisms.

Types of Synchronization

Synchronized Methods

Example:

public synchronized void increment() { counter++; }

Synchronized Blocks

Example:

synchronized(this) { counter++; }

Explicit Locks (ReentrantLock)

Example:

lock.lock(); try { counter++; } finally { lock.unlock(); }

Atomic Variables

Example:

AtomicInteger counter = new AtomicInteger(0); counter.incrementAndGet();

Difference Between Synchronization and Locking

Feature	Synchronization (synchronized)	Explicit Lock (ReentrantLock)
Type	Implicit locking	Explicit locking
Granularity	Method/block level	Fine-grained control
Performance	Can cause thread blocking	Supports tryLock() (non-blocking)
Fairness	No fairness control	Can provide fair locking
Interruptible?	No (waits indefinitely)	Yes (lockInterruptibly())
Condition Variables	No built-in condition support	Supports newCondition()

When to Use What?

- Use **synchronization (synchronized)** for simple thread safety when modifying shared resources.
- Use **explicit locks (ReentrantLock)** for advanced control like fairness, tryLock, or interruptible locks.
- Use **atomic variables (AtomicInteger)** when only counter-like operations are needed.