**Why Use synchronized (this)?**

* When you use synchronized (this), you're locking the current instance (the object on which the method/block is called).
* This ensures that **only one thread at a time** can execute the synchronized block/method **for that particular object**.

**What If Synchronization Was Not Used?**

Without synchronization, **multiple threads could execute the critical section simultaneously**, leading to **race conditions**. For example, in a banking application, if two threads try to modify the same account balance at the same time, the result could be inconsistent.

**Locks**

The synchronized keyword in Java provides basic thread-safety but has limitations: it locks the entire method or block, leading to potential performance issues. It lacks a try-lock mechanism, causing threads to block indefinitely, increasing the risk of deadlocks. Additionally, synchronized doesn't support multiple condition variables, offering only a single monitor per object with basic wait/notify mechanisms. In contrast, explicit locks (Lock interface) offer finer-grained control, try-lock capabilities to avoid blocking, and more sophisticated thread coordination through multiple condition variables, making them more flexible and powerful for complex concurrency scenarios.

**Why synchronized Doesn't Support Multiple Condition Variables?**

In Java, the synchronized keyword relies on the **intrinsic monitor lock** associated with an object. Each monitor supports only a **single condition queue**, and its condition management is very basic:

1. **Single Monitor/Condition Queue:** When a thread calls wait(), it releases the lock and waits on this single monitor queue until another thread calls notify() or notifyAll().
2. **No Fine-Grained Control:** The wait/notify mechanism does not allow multiple independent condition variables. All waiting threads are notified collectively or selectively (but only via a single condition queue).
3. **Limited Features:** There’s no support for features like **try-lock** (attempting to acquire a lock without blocking) or **timeout-based locking**.

**Explicit Locks and Condition Variables**

Java’s **Lock interface** (provided in the java.util.concurrent.locks package) offers a much more flexible locking mechanism compared to synchronized. The Lock interface supports:

1. **Multiple Condition Variables:**
   * With a single Lock, you can create **multiple independent condition queues** using the newCondition() method.
   * Each condition queue can signal its waiting threads separately, providing finer-grained control for more complex scenarios.
2. **Better Thread Coordination:**
   * Unlike synchronized (which uses wait()/notify()), explicit locks use Condition.await() and Condition.signal() to manage thread communication, offering clearer semantics.
3. **tryLock() Capability:**
   * tryLock() allows a thread to attempt to acquire the lock **without blocking**. If the lock is unavailable, it can fail immediately or after a specified timeout.
   * This avoids potential deadlocks where threads could block indefinitely.
4. **Interruptible Locks:**
   * Explicit locks allow threads to interrupt waiting threads via lockInterruptibly().

**Key Difference: Multiple Condition Variables Example**

Here’s an example to demonstrate how synchronized and the Lock interface handle thread coordination.

**Explanation**

1. **Multiple Condition Variables:**
   * The Lock interface allows you to create multiple Condition objects (condition1, condition2) with lock.newCondition().
   * Threads can wait on specific conditions and receive signals only relevant to their condition.
2. **Finer-Grained Control:**
   * signal() or signalAll() can target specific threads waiting on a particular condition, avoiding unnecessary wake-ups.
3. **Advanced Features:**
   * tryLock() and lockInterruptibly() provide timeout-based and interruptible locking mechanisms that are not possible with synchronized.

