Reentrant Lock

* A flexible custom lock that allows only one thread to access the critical section, regardless of the object.
* Locking is done using lock.lock() and released using lock.unlock().
* Example: If two threads use different objects but the same lock, they won’t be allowed into the critical section simultaneously.
* [Example](https://github.com/anubhavgit786/Java-Backend/tree/main/3.%20Advance%20Locking/2.%20Re-entrant%20Lock/1.%20Code%20Example)

Fair Lock

By default, locks are not fair. When a fair lock is enabled, the threads acquire the lock in the order they requested it. This is often referred to as first-come, first-served (FCFS). A fair lock ensures that if one thread has been waiting longer than others, it will be granted access to the lock before newer threads. This can help avoid starvation, where a thread may wait indefinitely if new threads keep arriving and acquiring the lock before it. In the context of Re-entrant Lock, fairness refers to how the lock manages the order in which threads are granted access to a shared resource.

Re-entrant Lock

A Re-entrant Lock is a type of lock mechanism provided in multithreaded programming that allows the same thread to acquire the lock multiple times without causing a deadlock. This lock ensures mutual exclusion, meaning only one thread can access the critical section at a time, but it also allows a thread to reenter a critical section it has already locked. In Java, the ReentrantLock class in the java.util.concurrent.locks package is used for this purpose. Here's how it works:

Key Features:

* Re-entrancy: If a thread acquires the lock, it can lock it again without blocking itself. The lock must be released the same number of times as it was acquired.
* Explicit Locking: Unlike synchronized blocks, you have to explicitly lock and unlock using lock() and unlock() methods.
* Fairness: The lock can be configured for fairness. If fairness is set, the longest-waiting thread will get the lock first.
* Condition Support: It provides better thread coordination through Condition objects, offering wait/notify-like mechanisms with finer control.
* Use Cases:

1. Thread-safe shared resources: When multiple threads need controlled access to a shared resource.
2. Re-entering critical sections: If a method calls another method that also requires locking, Re-entrant Lock prevents deadlocks.

* [Example](https://github.com/anubhavgit786/Java-Backend/tree/main/3.%20Advance%20Locking/2.%20Re-entrant%20Lock/2.%20Code%20Example)

Producer Consumer Problem using Reentrant Lock

[Example🤚🏻](https://github.com/anubhavgit786/Java-Backend/tree/main/3.%20Advance%20Locking/2.%20Re-entrant%20Lock/3.%20Code%20Example)

This code demonstrates a classic Producer-Consumer problem using thread-safe mechanisms provided by the ReentrantLock and Condition for managing the synchronization between threads.