Reentrant lock

In the realm of concurrent programming, where multiple threads execute simultaneously, the need to synchronize access to shared resources is crucial for ensuring data integrity and preventing race conditions.

Java provides powerful tools for this purpose, such as **ReentrantLock** and **ReentrantReadWriteLock**.

Main concept

In concurrent programming, ReentrantLock serves as a powerful tool for synchronization.

Going beyond the capabilities of the traditional synchronized keyword, ReentrantLock offers a robust mechanism for controlling access to shared resources.

Central to its utility is the feature of "reentrancy," which allows a thread to acquire the same lock multiple times.

This feature enables safe nested synchronization and helps circumvent deadlocks, thereby enhancing code flexibility.

Explicit lock acquisition and release provide fine-grained synchronization, allowing threads to operate concurrently on different sections of the shared resource while maintaining order and integrity.

```
ReentrantLock reentrantLock = new ReentrantLock();
```

While ReentrantLock excels in providing exclusive access to critical sections, ReentrantReadWriteLock takes synchronization a step further.

It recognizes that not all thread interactions require the same level of exclusivity.

By implementing the ReadWriteLock interface, ReentrantReadWriteLock allows multiple threads to read a shared resource concurrently while permitting only one thread to write to the resource at any given time.

This lock is particularly useful for improving concurrency and performance in situations where there are more reader threads than writer threads. Additionally, you have the option of using ReentrantLock for other synchronization needs.

```
ReentrantReadWriteLock reentrantReadWriteLock = new ReentrantReadWriteLock();
```

Dividing locks into read and write locks is an important concept in multithreaded programming, especially when working with data that is frequently read but rarely updated.

Consider a situation where you have a data structure that serves as a cache. If you use full synchronization through the synchronized keyword, only one thread can read or write the data at any given moment.

This can lead to significant delays, especially when you have many threads that frequently read the data.

Now let's consider the same situation, but using ReentrantReadWriteLock. In this case, if a thread wants to write data, it acquires the write lock.

During this time, no other thread can read or write data. However, if a thread wants to read data, it acquires the read lock.

If another thread also wants to read data, it can likewise acquire the read lock, even if it's already held by another thread.

This allows multiple threads to read data simultaneously, greatly improving performance in scenarios where read operations are much more frequent than write operations.

Fair or non-fair threads

The ReentrantLock class provides a constructor with a single boolean parameter, commonly referred to as the "fair flag." This flag determines whether the lock should operate in a fair or non-fair mode.

- **fair=true**: When you instantiate a ReentrantLock constructor with this flag set to true, the lock operates in fair mode. In this mode, when multiple threads are waiting to acquire the lock, they are granted access in the order in which they requested it. Fair mode can be useful in scenarios where you want to ensure that threads do not experience indefinite waits and are given an opportunity to acquire the lock in a first-come, first-served manner.
- **fair=false(default)**: When you create a ReentrantLock with the fair flag set to false, the lock operates in non-fair mode. In this mode, there is no guaranteed order for thread access; threads are allowed to acquire the lock in a more arbitrary manner. This could potentially lead to some threads experiencing longer waits or being disproportionately favored. Non-fair mode can offer better performance and throughput compared to fair mode, but it may result in less predictable and

equitable thread scheduling.

```
import java.util.concurrent.locks.ReentrantLock;

public class FairnessExample {
    public static void main(String[] args) {
        ReentrantLock fairLock = new ReentrantLock(true); // Creating a fair lock
        ReentrantLock nonFairLock = new ReentrantLock(); // Creating a non-fair
lock

        ReentrantReadWriteLock reentrantReadWriteLock = new
ReentrantReadWriteLock(true);
        ReentrantReadWriteLock reentrantReadWriteLock = new
ReentrantReadWriteLock();
    }
}
```

Methods of class

The ReentrantLock class provides methods for acquiring the lock and attempting to acquire the lock without blocking.

- **lock()**: Acquires the lock, blocking the current thread if the lock is not available. This method should be used within a try block to ensure proper unlocking in a finally block.
- **tryLock()**: Attempts to acquire the lock without blocking.
  Returns true if the lock was successfully acquired, and false otherwise.
  This is useful when you wish to try acquiring a lock and proceed with an alternative action if the lock isn't immediately available.
- **unlock()**: Releases the lock that was previously acquired by a thread using the lock() method.
- **getHoldCount()**: Queries the number of holds on this lock by the current thread. A thread has a hold on a lock for
  each lock() or lockInterruptibly() action that is not balanced by an unlock() action
- **getQueueLength()** : Returns the number of threads waiting to acquire the lock.

```
ReentrantLock lock = new ReentrantLock();
lock.lock();  // Acquire the lock (within a try block)
try {
    // Critical section
} finally {
    lock.unlock();  // Release the lock (within a finally block)
```

```
}
```
You can also use tryLock() to attempt to acquire a lock without blocking the current thread:
```
if (lock.tryLock()) {
    try {
        // Critical section
    } finally {
        lock.unlock();
    }
} else {
    // Handle alternative action if lock is not available
}
```

ReentrantReadWriteLock offers the same methods as ReentrantLock, but also includes additional methods:

- **readLock()**: Acts as a special pass allowing multiple threads to read a shared resource simultaneously. When a thread holds the readLock, it indicates that it's only reading and not modifying the resource.
- **writeLock()**: Is reserved for exclusive purposes. It's akin to being granted the role of the author, who can add, modify, or rewrite the content of a book. Only one thread at a time is allowed to hold the writeLock.
- **getReadLockCount()**: Returns the number of read locks held. This includes not just the count of threads that have acquired the read lock, but also the total count of these acquisitions. A single thread can acquire the read lock multiple times, and each acquisition is counted.
- **getWriteHoldCount()**: Queries the number of reentrant write holds on this lock by the current thread. A writer thread has a hold on a lock for each lock action that is not matched by an unlock action.

```
ReentrantReadWriteLock rwLock = new ReentrantReadWriteLock();
ReentrantLock readLock = rwLock.readLock();
ReentrantLock writeLock = rwLock.writeLock();
```

If a thread has acquired the writeLock, no other thread can acquire either it or the readLock until the writeLock is released. This is because only one thread at a time is permitted to hold the writeLock.
On the other hand, if a thread has acquired the readLock, other threads can still acquire the readLock, even while it is held by the first thread. However, no thread can acquire the writeLock until all instances of the readLock have been released. This enables multiple threads to read data simultaneously while ensuring that writes are exclusive.

In this example, the recursiveMethod acquires the lock before performing its

operations and then releases the lock in a finally block. The method calls itself recursively while holding the lock.
Next, we'll show an example involving ReentrantReadWriteLock:

```
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class Cache {
    private Map<String, String> cache = new HashMap<>();
    private ReentrantReadWriteLock lock = new ReentrantReadWriteLock();

    public String readFromCache(String key) {
        lock.readLock().lock();
        try {
            return cache.get(key);
        } finally {
            lock.readLock().unlock();
        }
    }

    public void writeToCache(String key, String value) {
        lock.writeLock().lock();
        try {
            cache.put(key, value);
        } finally {
            lock.writeLock().unlock();
        }
    }
}
```

In this example, the readFromCache method acquires the read lock before accessing the cache. This allows multiple threads to read from the cache simultaneously. The writeToCache method acquires the write lock before modifying the cache. This ensures that no other threads can read or write while the cache is being updated.

Conclusion

Both ReentrantLock and ReentrantReadWriteLock offer the valuable feature of "fairness," allowing threads to acquire locks in the order in which they were requested. These locks provide essential methods such as lock, unlock, and tryLock, to ensure precise control over concurrent access to shared resources. Moreover, ReentrantReadWriteLock extends this functionality by introducing the readLock() and writeLock() methods, adeptly catering to scenarios where a balance between reading and writing operations is crucial. As a best practice for maintaining synchronization integrity, the tried-and-true approach of using try-finally blocks comes highly recommended, to ensure that locks are consistently and responsibly released.