Reentrancy and ReentrantLock in Java – 2024

**What is Reentrancy in Java: A single thread is allowed to lock the same object multiple times.**

**Why called ReentrantLock?**

**Its name comes from its design:** When a thread tries to lock an object which is already locked by another thread, the requesting thread will be blocked. But when a thread tries to lock object which is held by itself, the request will succeed and the lock's counter will be incremented, which is called reentrancy**.**

**Here is an example of using ReentrantLock to implement reentrancy:**

**public class ReentrantExample {**

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

**public void inner() {**

**lock.lock();**

**try {**

**SOP("inner lock acquired");**

**} finally {**

**lock.unlock();**

**SOP("inner lock released");**

**}**

**}**

**public void outer() {**

**lock.lock();**

**try {**

**SOP("outer lock acquired");**

**inner();**

**} finally {**

**lock.unlock();**

**SOP("outer lock released");**

**}**

**}**

**public static void main(String[] args) {**

**ReentrantExample example = new ReentrantExample();**

**example.outer();**

**}**

**}**

**In this example, both the outer() and inner() methods attempt to acquire the same lock. When we call the outer() method in the main() method, the lock is first acquired, and then the inner() method is called within outer(). Since the lock is reentrant, the inner() method can successfully acquire the lock without being blocked.**

<http://stackoverflow.com/questions/16504231/reentrancy-in-java>

**Reentrancy means that locks are acquired on a per-thread rather than per-invocation basis.**

With a reentrant lock / locking mechanism, the attempt to acquire the same lock will succeed, and will increment an internal counter belonging to the lock.

Here's a example in Java using primitive object locks / monitors ... which are reentrant:

Object lock = new Object();

...

**synchronized (lock) {**

**...**

**doSomething(lock, ...)**

**...**

**}**

**public void doSomething(Object lock, ...) {**

**synchronized (lock) {**

**...**

**}**

**}**

**A thread won't deadlock against itself (if the lock is reentrant).**

**Java concurrency in practice book states - Reentrancy means that locks are acquired on a per-thread rather than per-invocation basis.**

**First of all Intrinsic locks are reentrant by nature.**

<https://dzone.com/articles/what-are-reentrant-locks>

An intrinsic locking mechanism can have some functional limitations, such as:

1.) **It is not possible to interrupt a thread waiting to acquire a lock (lock Interruptibly).**

2.) It is not possible to attempt to acquire a lock without being willing to wait for it forever (try lock).

3.) Cannot implement non-block-structured locking disciplines, **as intrinsic locks must be released in the same block in which they are acquired. With intrinsic locks, acquire-release pairs are block-structured and a lock is always released in the same basic block in which it was acquired.**

**Fairness**

The ReentrantLock constructor offers a choice of two fairness options: create a non-fair lock or a fair lock.

**In non-fair mode**

* It selects thread without any criteria.
* In this mode, the thread that requests the lock, tries to get it immediately without getting enqueued. If that succeeds, It is not put into any queue, since it could get the lock. It does not care if there are already waiting threads in the queue, thus it is "unfair". If it could not acquire the lock (meaning someone else has it), it is put on the queue.
* Unfair lock ignore the waiting queue, try to acquire the lock directly.

**In fair mode**

* In fair mode, it selects thread that has waiting for the most time.
* In fair mode, the thread that calls lock, first has to check if there are already threads waiting for this lock. If there are no such threads and it can take the lock: no enqueue and take the lock. If it can't - it is enqueued.
* fair lock uses a fair strategy (ordering and waiting according to the waiting queue) every time it tries to acquire a lock.

The process of acquiring locks is the same for unfair locks and fair locks. The difference between them is different mechanisms for trying to obtain locks.

**If a thread is not granted CPU time because other threads grab it all, it is called "starvation** **The solution to starvation is called "fairness"** - that all threads are fairly granted a chance to execute.

**Synchronized blocks in Java are reentrant**. This means, that if a Java thread enters a synchronized block of code, and thereby take the lock on the monitor object the block is synchronized on, the thread can enter other Java code blocks synchronized on the same monitor object.

**Reentrant Lock in Java**

As the name says, **ReentrantLock allow threads to enter into lock on a resource more than once. When the thread first enters into lock, a hold count is set to one. Before unlocking the thread can re-enter into lock again and every time hold count is incremented by one. For every unlock request, hold count is decremented by one and when hold count is 0, the resource is unlocked.**

Before we move into ReentrantLock, let us see a use case using typical **synchronized** keyword.

Problem: There is an ATM machine, there are 5 persons who want to withdraw cash.

**public class** TestATM {  
  
 **public void** sleep(**int** time) {  
 **try** {  
 TimeUnit.***SECONDS***.sleep(time);  
 } **catch** (InterruptedException e) {  
 **throw new** RuntimeException(e);  
 }  
 }  
  
 **public synchronized int** getCash(String name, **int** amount) {  
 System.***out***.println(name + " trying to withdraw cash " + amount);  
 sleep(2);  
 System.***out***.println(name + " got cash " + amount);  
 **return** amount;  
 }  
  
 **public void** check() {  
 **for** (**int** i = 0; i < 5; i++) {  
 String name = "T-" + i;  
 **new** Thread(() -> getCash(name, 2000)).start();  
 }  
 }  
  
 **public static void** main(String[] args) {  
 **new** TestATM().check();  
 }  
}

**OUTPUT**

T-0 trying to withdraw cash 2000

T-0 got cash 2000

T-4 trying to withdraw cash 2000

T-4 got cash 2000

T-2 trying to withdraw cash 2000

T-2 got cash 2000

T-3 trying to withdraw cash 2000

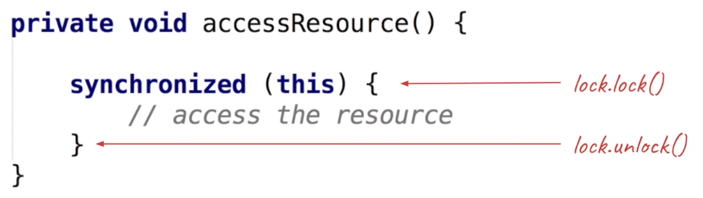
T-3 got cash 2000

T-1 trying to withdraw cash 2000

T-1 got cash 2000

It means all the persons got the money. This is how synchronization keyword works.

With synchronized keyword, we can achieve thread synchronization like this.



ATM Machine Code using ReentrantLock is given below

**public class** TestATM {  
  
 **private** Lock lock = **new** ReentrantLock();  
  
 **public void** sleep(**int** time) {  
 **try** {  
 TimeUnit.***SECONDS***.sleep(time);  
 } **catch** (InterruptedException e) {  
 **throw new** RuntimeException(e);  
 }  
 }  
  
 **public int** getCash(String name, **int** amount) {  
 lock.lock();  
 **try** {  
 System.***out***.println(name + " trying to withdraw cash " + amount);  
 sleep(2);  
 System.***out***.println(name + " got cash " + amount);  
 } **finally** {  
 lock.unlock();  
 }  
 **return** amount;  
 }  
  
 **public void** check() {  
 **for** (**int** i = 0; i < 5; i++) {  
 String name = "T-" + i;  
 **new** Thread(() -> getCash(name, 2000)).start();  
 }  
 }  
  
 **public static void** main(String[] args) {  
 **new** TestATM().check();  
 }  
  
}

**OUTPUT**

T-0 trying to withdraw cash 2000

T-0 got cash 2000

T-2 trying to withdraw cash 2000

T-2 got cash 2000

T-1 trying to withdraw cash 2000

T-1 got cash 2000

T-3 trying to withdraw cash 2000

T-3 got cash 2000

T-4 trying to withdraw cash 2000

T-4 got cash 2000

The constructor for this class accepts an optional *fairness* parameter. When set true, under contention, locks favor granting access to the longest-waiting thread. Otherwise **this lock does not guarantee any particular access order**. **Note however, that fairness of locks does not guarantee fairness of thread scheduling.**

**Why ReentrantLock is Better Over Intrinsic Lock (synchronized)**

* **A *synchronizedblock* is fully contained within a method**. In case of synchronized block structure, lock will be acquired and released in the same block. We can have lock() and unlock() operations in separate method.
* **A s*ynchronized block* doesn’t support the fairness**. Any thread can acquire the lock once the is released, and no preference can be specified. We can achieve fairness within the *Lock* APIs by specifying the ***fairness* property**. It makes sure that the longest waiting thread is given access to the lock.
* A thread gets blocked if it can’t get an access to the synchronized *block*. The *Lock* API provides ***tryLock()***method. The thread acquires lock only if it’s available and not held by any other thread. This reduces blocking time of thread waiting for the lock.
* A thread that is in “**waiting**” state to acquire the access to *synchronized block* can’t be interrupted. The *Lock* API provides a method ***lockInterruptibly()*** that can be used to interrupt the thread when it’s waiting for the lock.

*ReentrantLock* key features as per this [article](http://javarevisited.blogspot.com/2013/03/reentrantlock-example-in-java-synchronized-difference-vs-lock.html#ixzz3uIbhr77e)

1. Ability to lock interruptibly.
2. Ability to timeout while waiting for lock.
3. Power to create fair lock.
4. API to get list of waiting thread for lock.
5. Flexibility to try for lock without blocking.

You can use *ReentrantReadWriteLock.ReadLock, ReentrantReadWriteLock.WriteLock* to further acquire control on granular locking on read and write operations.

**tryLock()**: When the thread calls tryLock() on the resource then if the resource is available, thread acquires the lock and tryLock() returns **true** and hold count is incremented by 1. **TryLock() acquires the lock only if it is free at the**[**time**](https://crunchify.com/java-timer-and-timertask-reminder-class-tutorials-example/)**of invocation.**

If the lock is held by another thread then this method will return immediately with the value false.

**The example will be, if only one film ticket is available, only one person will get it but 3 persons are trying to get the ticket. In this situation, we have to use tryLock()**.

**public class** MovieTicket {  
 **private Lock lock = new ReentrantLock();**  
  
 **public void** sleep(**int** time) {  
 **try** {  
 TimeUnit.***SECONDS***.sleep(time);  
 } **catch** (InterruptedException e) {  
 **throw new** RuntimeException(e);  
 }  
 }  
  
 **public void** getTicket(String name) {  
 **boolean tryLockFlag = lock.tryLock();**  
 **try** {  
 **if** (tryLockFlag) {  
 System.***out***.println(name + " trying to get ticket ");  
 sleep(2);  
 System.***out***.println(name + " got ticket ");  
 }  
 } **catch**(Exception ex) { // don't do anything }  
 **finally** {  
 **try** {  
 **lock.unlock();**  
 } **catch** (Exception e) {  
 // Don't do anything  
 }  
 }  
 }  
  
 **public void** check() {  
 **for** (**int** i = 0; i < 5; i++) {  
 String name = "T-" + i;  
 **new** Thread(() -> getTicket(name)).start();  
 }  
 }  
  
 **public static void** main(String[] args) {  
 **new** MovieTicket().check();  
 }  
}

**OUTPUT**

T-0 trying to get ticket

T-0 got ticket 🡺 **Here, you can see only T-0** **got ticket and others did not get**.

The above method can also be written like this.

**public void** getTicket(String name) {  
 **try** {  
 boolean tryLockFlag = lock.tryLock(100, TimeUnit.*MILLISECONDS*);  
 **if** (tryLockFlag) {  
 System.***out***.println(name + " trying to get ticket ");  
 sleep(2);  
 System.***out***.println(name + " got ticket ");  
 }  
 } **catch** (Exception ex) {  
 // don't do anything  
 } **finally** {

**lock.unlock();**  
 }  
}

**tryLock(long timeout, TimeUnit unit):** We can give a waiting time to tryLock() so that it must wait to acquire lock for the given time if lock is not available. Suppose resource is locked by any thread and current thread calls tryLock(100, TimeUnit.MILLISECONDS) then current thread will wait max for 100 milliseconds to acquire lock and once other thread which already has lock, unlocks it within 100 milliseconds, the current thread will acquire lock and hold count will be incremented by one, even if other threads are waiting for lock. If for the specified time, current thread is not able to acquire lock, this method will return **false**.

If you want a timed tryLock that does permit barging on a fair lock then combine the timed and un-timed forms together:

***if (lock.tryLock() || lock.tryLock(timeout, unit) ) { ... }***

**If you increase time to 3 seconds, probably 2 persons may get money**

**public class** MovieTicket {  
  
 **private** Lock lock = **new** ReentrantLock();  
  
 **public void** sleep(**int** time) {  
 **try** {  
 TimeUnit.***SECONDS***.sleep(time);  
 } **catch** (InterruptedException e) {  
 **throw new** RuntimeException(e);  
 }  
 }  
  
 **public void** getTicket(String name) {  
 **try** {  
 **boolean** tryLockFlag = lock.tryLock(3, TimeUnit.***SECONDS***);  
 **if** (tryLockFlag) {  
 System.***out***.println(name + " trying to get ticket ");  
 sleep(2);  
 System.***out***.println(name + " got ticket ");  
 }  
 } **catch** (Exception ex) {  
 // don't do anything  
 } **finally** {  
 **try** {  
 lock.unlock();  
 } **catch** (Exception e) {  
 // Don't do anything  
 }  
 }  
 }  
  
 **public void** check() {  
 **for** (**int** i = 0; i < 5; i++) {  
 String name = "T-" + i;  
 **new** Thread(() -> getTicket(name)).start();  
 }  
 }  
  
 **public static void** main(String[] args) {  
 **new** MovieTicket().check();  
 }  
  
}

OUTPUT

T-0 trying to get ticket

T-0 got ticket

T-1 trying to get ticket

T-1 got ticket

}

**In this case, two threads got the ticket after increasing the time.**

**lockInterruptibly()**

* Acquires the lock unless the current thread is [interrupted](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#interrupt()).
* Acquires the lock if it is not held by another thread and returns immediately, setting the lock hold count to one.
* If the current thread already holds this lock then the hold count is incremented by one and the method returns immediately.
* If the lock is held by another thread then the current thread becomes disabled for thread scheduling.

**Problem: Is it possible to interrupt a thread waiting to acquire the lock ?**

**By the way it is not possible in case of synchronized keyword in java**. This is possible using **lock.lockInterruptibly()**.

A good example is that: **There 3 persons who are trying to withdraw cash from ATM machine, if the cash is over, they should be informed so that they can come out. Technically, if the cash is over in ATM, other threads should be interrupted**.

**public class BankATM {  
  
 public static volatile boolean *isCashOver* = true;  
 private ReentrantLock lock = new ReentrantLock();  
  
 public void withdrawCash(int amount) {  
 try {  
 lock.lockInterruptibly();  
 System.*out*.println(Thread.*currentThread*().getName() + " trying to withdraw cash " + amount);  
 TimeUnit.*SECONDS*.sleep(5);  
 System.*out*.println(Thread.*currentThread*().getName() + " got " + amount);  
 *isCashOver* = false;  
 TimeUnit.*SECONDS*.sleep(5);  
 } catch (Exception e) {  
 System.*out*.println("Which Thread Interrupted ? "+Thread.*currentThread*().getName());  
 } finally {  
 if(lock.isHeldByCurrentThread())  
 lock.unlock();  
 }  
 }  
  
 public void check() {  
 Thread t1 = new Thread( () -> withdrawCash(2000));  
 Thread t2 = new Thread( () -> withdrawCash(2000));  
 Thread t3 = new Thread( () -> withdrawCash(2000));  
 t1.start();  
 t2.start();  
 t3.start();  
 boolean flag = true;  
 while(flag) {  
 flag = BankATM.*isCashOver*;  
 if(flag == false) {  
 t2.interrupt();  
 t3.interrupt();  
 }  
 }  
 System.*out*.println("Completed ...");  
 }  
  
 public static void main(String[] args) {  
 new BankATM().check();  
 }  
}**

**OUTPUT**

Thread-0 trying to withdraw cash 2000

Thread-0 got 2000

Completed ...

Which Thread Interrupted ? Thread-1

Which Thread Interrupted ? Thread-2

**Problem: What is the real benefit of ReentrantLock over synchronized method ?**

Let us see the code below.

**public class** Bank {  
  
 **public synchronized void** depositMoney(**int** amount) {  
 **try** {  
 System.***out***.println(Thread.*currentThread*().getName() + " trying to deposit money of Rs " + amount);  
 TimeUnit.***SECONDS***.sleep(10);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(Thread.*currentThread*().getName() + " finally deposited money ...");  
 }  
  
 **public synchronized void** withdrawCash(**int** amount) {  
 **try** {  
 System.***out***.println(Thread.*currentThread*().getName() + " trying to withdraw money of Rs " + amount);  
 TimeUnit.***SECONDS***.sleep(5);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(Thread.*currentThread*().getName() + " finally got money ...");  
 }  
  
  
 **public void** check() {  
 Thread t1 = **new** Thread(() -> depositMoney(2000));  
 Thread t2 = **new** Thread(() -> withdrawCash(2000));  
  
 t1.start();  
 t2.start();  
  
 System.***out***.println("Completed ...");  
 }  
  
 **public static void** main(String[] args) {  
 **new** Bank().check();  
 }  
  
}

**OUTPUT**

Completed ...

Thread-0 trying to deposit money of Rs 2000

Thread-0 finally deposited money ...

Thread-1 trying to withdraw money of Rs 2000

Thread-1 finally got money ...

In this case, only one thread can access either deposit( or withdraw() method at a time.

You may be thinking that instead of using synchronized, we can use ReentrantLock,

let us see what happens if we change the Bank class.

**public class** Bank {  
  
 **private** Lock lock = **new** ReentrantLock(**true**);  
  
 **public void** depositMoney(**int** amount) {  
 **try** {  
 System.***out***.println(Thread.*currentThread*().getName() + " trying to deposit money of Rs " + amount);  
 lock.lock();  
 TimeUnit.***SECONDS***.sleep(10);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 } **finally** {  
 lock.unlock();  
 }  
 System.***out***.println(Thread.*currentThread*().getName() + " finally deposited money ...");  
 }  
  
 **public void** withdrawCash(**int** amount) {  
 **try** {  
 System.***out***.println(Thread.*currentThread*().getName() + " trying to withdraw money of Rs " + amount);  
 lock.lock();  
 TimeUnit.***SECONDS***.sleep(5);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 } **finally** {  
 lock.unlock();  
 }  
 System.***out***.println(Thread.*currentThread*().getName() + " finally got money ...");  
 }  
  
  
 **public void** check() {  
 Thread t1 = **new** Thread(() -> depositMoney(2000));  
 Thread t2 = **new** Thread(() -> withdrawCash(2000));  
  
 t1.start();  
 t2.start();  
  
 System.***out***.println("Completed ...");  
 }  
  
 **public static void** main(String[] args) {  
 **new** Bank().check();  
 }  
  
}

**OUTPUT**

Thread-0 trying to deposit money of Rs 2000

Thread-1 trying to withdraw money of Rs 2000

Thread-0 finally deposited money ...

Thread-1 finally got money ...

To some extent, the problem can be solved. But it can be written in more better way.

**public class** Bank {  
  
 **private** ReadWriteLock readWriteLock = **new** ReentrantReadWriteLock();  
 **private** Lock depositLock = readWriteLock.writeLock();  
 **private** Lock withdrawLock = readWriteLock.readLock();  
  
 **public void** depositMoney(**int** amount) {  
 **try** {  
 depositLock.lock();  
 System.***out***.println(Thread.*currentThread*().getName() + " depositing money of Rs " + amount);  
 TimeUnit.***SECONDS***.sleep(10);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 } **finally** {  
 depositLock.unlock();  
 }  
 System.***out***.println(Thread.*currentThread*().getName() + " deposited money ...");  
 }  
  
 **public void** withdrawCash(**int** amount) {  
 **try** {  
 withdrawLock.lock();  
 System.***out***.println(Thread.*currentThread*().getName() + " withdrawing money of Rs " + amount);  
 TimeUnit.***SECONDS***.sleep(5);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 } **finally** {  
 withdrawLock.unlock();  
 }  
 System.***out***.println(Thread.*currentThread*().getName() + " got money ...");  
 }  
  
 **public void** check() {  
 Thread t2 = **new** Thread(() -> withdrawCash(2000));  
 t2.setName("Vidya");  
 Thread t4 = **new** Thread(() -> withdrawCash(4000));  
 t4.setName("Samir");  
 Thread t1 = **new** Thread(() -> depositMoney(2000));  
 t1.setName("John");  
 Thread t3 = **new** Thread(() -> depositMoney(3000));  
 t3.setName("Swati");  
  
 t2.start();  
 t4.start();  
 t1.start();  
 t3.start();  
  
 }  
  
 **public static void** main(String[] args) {  
 **new** Bank().check();  
 }  
  
}

The main concept here is that it allows multiple threads to read the data concurrently and one thread to update the data exclusively.

Note: **The fairness parameter is used to provide lock to longest waiting thread. For example it will be like**

**Lock atmMachine = new ReentrantLock(true);**

Code provided by Microsoft Copilot and modified – an example on read write lock

**public class ReadWriteLockExample {  
 private final ReadWriteLock readWriteLock = new ReentrantReadWriteLock();  
 private final Lock readLock = readWriteLock.readLock();  
 private final Lock writeLock = readWriteLock.writeLock();  
 private final List<String> list = new ArrayList<>();  
  
 public void addElement(String element) {  
 writeLock.lock();  
 try {  
 list.add(element);  
 System.*out*.println(Thread.*currentThread*().getName() + " added: " + element);  
 } finally {  
 writeLock.unlock();  
 }  
 }  
  
 public void sleep(long time) {  
 try {  
 TimeUnit.*SECONDS*.sleep(time);  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 }  
  
 public void write() {  
 for (int i = 0; i < 5; i++) {  
 addElement("Element " + i);  
 sleep(1);  
 }  
 }  
  
 public void readData() {  
 for (int i = 0; i < 5; i++) {  
 readLock.lock();  
 try {  
 String threadName = Thread.*currentThread*().getName();  
 System.*out*.println("Who is reading ?"+threadName+" and Data: "+list);  
 } finally {  
 readLock.unlock();  
 }  
 sleep(1);  
 }  
 }  
  
 public static void main(String[] args) {  
 ReadWriteLockExample example = new ReadWriteLockExample();  
  
 Runnable writer = () -> example.write();  
 Runnable reader = () -> example.readData();  
  
 Thread writerThread = new Thread(writer, "WriterThread");  
 Thread readerThread1 = new Thread(reader, "ReaderThread-1");  
 Thread readerThread2 = new Thread(reader, "ReaderThread-2");  
  
 writerThread.start();  
 readerThread1.start();  
 readerThread2.start();  
 }  
}**