Reentrancy and ReentrantLock in Java – 2021-2022

**What is Reentrancy in Java**

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

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

Reentrancy means (in general CS / IT terminology) that you do something, and while you are still doing it, you do it again. In the case of locks it means you do something like this on a single thread:

1. Acquire a lock on "foo".
2. Do something
3. Acquire a lock on "foo". Note that we haven't released the lock that we previously acquired.
4. ...
5. Release lock on "foo"
6. ...
7. Release lock on "foo"

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. The lock will only be released when the current holder of the lock has released it twice.

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

Object lock = new Object();

**function A():**

**lock (X)**

**B()**

**unlock (X)**

...

**function B():**

**A()**

**synchronized (lock) {**

**...**

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

**...**

**}**

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

**synchronized (lock) {**

**...**

**}**

**}**

Now we call A. The following happens:

* We enter A, locking X
* We enter B
* We enter A again, locking X again

Since we never exited the first invocation of A, X is still locked. This is called re-entrance - while function A has not yet returned, function A is called again. If A relies on some global, static state, this can cause a 're-entrance bug', where before the static state is cleaned up from the function's exit, the function is run again, and the half computed values collide with the start of the second call.

In this case, we run into a lock we are already holding. If the lock is re-entrance aware, it will realize we are the same thread holding the lock already and let us through. Otherwise, it will deadlock forever - it will be waiting for a lock it already holds. In java, lock and synchronized are re-entrance aware - if a lock is held by a thread, and the thread tries to re-acquire the same lock, it is allowed.

The advantage of using reentrant locks is that you don't have to worry about the possibility of failing due to accidentally acquiring a lock that you already hold. The downside is that you can't assume that nothing you call will change the state of the variables that the lock is designed to protect. However, that's not usually a problem. Locks are generally used to protect against concurrent state changes made by *other* threads.

So I needn't consider deadlocks? **Yes you do.**

**A thread won't deadlock against itself (if the lock is reentrant).** However, you could get a deadlock if there are other threads that might have a lock on the object you are trying to lock.

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

Let me explain what it exactly means. First of all Intrinsic locks are reentrant by nature. The way reentrancy is achieved is by maintaining a counter for number of locks acquired and owner of the lock. If the count is 0 and no owner is associated to it, means lock is not held by any thread. When a thread acquires the lock, JVM records the owner and sets the counter to 1. If same thread tries to acquire the lock again the counter is incremented, and when the owning thread exist synchronized block counter is decremented. When count reaches 0 again lock is released.

<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.**

A cleaner approach is implemented by ReentrantLock with the use of **tryLock()** method. This approach is called the "timed and polled lock-acquisition." It lets you regain control if you cannot acquire all the required locks, release the ones you have acquired and retry. So, using tryLock, we will attempt to acquire both locks. If we cannot attain both, we will release if one of these has been acquired, then retry.

**Interruptible Lock Acquisition**

Interruptible lock acquisition allows locking to be used within cancellable activities.

The *lockInterruptibly* method allows us to try and acquire a lock while being available for interruption. So, basically it allows the thread to immediately react to the interrupt signal sent to it from another thread.

**Fairness**

The ReentrantLock constructor offers a choice of two fairness options: create a non-fair lock or a fair lock. **With fair locking, threads can acquire locks only in the order in which they were requested**, whereas an unfair lock allows a lock to acquire it out of its turn. This is called **barging** (breaking the queue and acquiring the lock when it became available).

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

**Lock Reentrance**

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. Here is an example:

**public class Reentrant{**

**public synchronized outer(){**

**inner();**

**}**

**public synchronized inner(){**

**//do something**

**}**

**}**

Notice how both outer() and inner() are declared synchronized, which in Java is equivalent to a synchronized(this) block. If a thread calls outer() there is no problem calling inner() from inside outer(), since both methods (or blocks) are synchronized on the same monitor object ("this"). If a thread already holds the lock on a monitor object, it has access to all blocks synchronized on the same monitor object. This is called reentrance. The thread can reenter any block of code for which it already holds the lock.

**Reentrant Lock in Java**

The traditional way to achieve thread synchronization in Java is by the use of [synchronized](https://www.geeksforgeeks.org/synchronized-in-java/) keyword. While it provides a certain basic synchronization, the synchronized keyword is quite rigid in its use. For example, a thread can take a lock only once. **Synchronized blocks don’t offer any mechanism of a waiting queue** and after the exit of one thread, any thread can take the lock. **This could lead to starvation of resources for some other thread for a very long period of time.**

**What are Reentrant Locks?**

The ReentrantLock class implements the Lock interface and provides synchronization to methods while accessing shared resources. The code which manipulates the shared resource is surrounded by calls to lock and unlock method. This gives a lock to the current working thread and blocks all other threads which are trying to take a lock on the shared resource.

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** ATM { 🡸ATM Machine code is given below.  
 **public synchronized int** getCash(**int** amount) {  
 **int** withdrawnAmt = 0;  
 **try** {  
 System.***out***.println(Thread.*currentThread*().getName() + **" trying to withdraw cash "** + amount);  
 TimeUnit.***SECONDS***.sleep(5);  
 withdrawnAmt = amount;  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 **return** withdrawnAmt;  
 }  
}

There are 5 persons, each person is considered as thread. The code is given below.

**public class PersonThread** **extends** Thread {  
 **private** ATM **atm**;  
 **private int amount**;  
 **private** String **name**;  
  
 **public** PersonThread(ATM atm, **int** amount, String name) {  
 **super**(name);  
 **this**.**atm** = atm;  
 **this**.**amount** = amount;  
 **this**.**name** = name;  
 }  
  
 @Override  
 **public void** run() {  
 **int** value = **atm**.getCash(**amount**);  
 System.***out***.println(  
 Thread.*currentThread*().getName() + **" got cash "** + value + **" using synchronization"**);  
 }  
}

Test Code

OUTPUT

John trying to withdraw cash 3000

Kent trying to withdraw cash 5000

John got cash 3000 using synchronization

Kent got cash 5000 using synchronization

Nalini trying to withdraw cash 9000

Nalini got cash 9000 using synchronization

Ramesh trying to withdraw cash 7000

Ramesh got cash 7000 using synchronization

Vidya trying to withdraw cash 1000

Vidya got cash 1000 using synchronization

**public class SynchronizationTest1** {  
 **public static void** main(String[] args) {  
 ATM atm = **new** ATM();  
 Thread t1 = **new** PersonThread(atm, 3000, **"John"**);  
 Thread t2 = **new** PersonThread(atm, 1000, **"Vidya"**);  
 Thread t3 = **new** PersonThread(atm, 7000, **"Ramesh"**);  
 Thread t4 = **new** PersonThread(atm, 9000, **"Nalini"**);  
 Thread t5 = **new** PersonThread(atm, 5000, **"Kent"**);  
 t1.start();  
 t2.start();  
 t3.start();  
 t4.start();  
 t5.start();  
 }  
}

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

ATM Machine Code using ReentrantLock is given below

**import** java.util.concurrent.TimeUnit;  
**import** java.util.concurrent.locks.Lock;  
**import** java.util.concurrent.locks.ReentrantLock;  
  
**public class ATM** {  
 **private** Lock **lock** = **new** ReentrantLock();  
  
 **public int** withdrawAmount(**int** amount) {  
 **int** withdrawnAmt = 0;  
 **lock.lock();**  
 **try** {  
 System.***out***.println(Thread.*currentThread*().getName() + **" trying to withdraw amount "** + amount);  
 TimeUnit.***SECONDS***.sleep(5);  
 withdrawnAmt = amount;  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 } **finally** {  
 **lock.unlock();** }  
 **return** withdrawnAmt;  
 }  
 }

OUTPUT

John trying to withdraw amount 3000

John got money 3000

Vidya trying to withdraw amount 1000

Vidya got money 1000

Nana trying to withdraw amount 2000

Nana got money 2000

Kent trying to withdraw amount 53000

Kent got money 53000

Douglas trying to withdraw amount 7000

Douglas got money 7000

**PersonThread.java**  
**public class PersonThread** **extends** Thread {  
 **private** ATM **atm**;  
 **private int amount**;  
 **private** String **name**;  
  
 **public** PersonThread(ATM atm, **int** amount, String name) {  
 **super**(name);  
 **this**.**atm** = atm;  
 **this**.**amount** = amount;  
 **this**.**name** = name;  
 }  
 @Override  
 **public void** run() {  
 **int** value = **atm**.withdrawAmount(**amount**);  
 System.***out***.println(Thread.*currentThread*().getName() + **" got money "** + value);  
 }  
}

**public class TestLockLock** { 🡸Test Class  
 **public static void** main(String[] args) {  
 ATM atm = **new** ATM();  
 Thread t1 = **new** PersonThread(atm, 3000, **"John"**);  
 Thread t2 = **new** PersonThread(atm, 1000, **"Vidya"**);  
 Thread t3 = **new** PersonThread(atm, 2000, **"Nana"**);  
 Thread t4 = **new** PersonThread(atm, 53000, **"Kent"**);  
 Thread t5 = **new** PersonThread(atm, 7000, **"Douglas"**);  
  
 t1.start();  
 t2.start();  
 t3.start();  
 t4.start();  
 t5.start();  
 }  
}

**ReentrantLock usage**

<https://stackoverflow.com/questions/11821801/why-use-a-reentrantlock-if-one-can-use-synchronizedthis>

A [ReentrantLock](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantLock.html) is *unstructured*, unlike synchronized constructs -- i.e. you don't need to use a block structure for locking and can even hold a lock across methods. An example:

private ReentrantLock lock;

public void foo() {

...

lock.lock();

...

}

public void bar() {

...

lock.unlock();

...

}

Such flow is impossible to represent via a single monitor in a synchronized construct.

Aside from that, ReentrantLock supports [lock polling](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html#tryLock()) and [interruptible lock waits that support time-out](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html#tryLock(long,%20java.util.concurrent.TimeUnit)). ReentrantLock also has support for [configurable *fairness* policy](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html#ReentrantLock(boolean)), allowing more flexible thread scheduling.

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. Programs using fair locks accessed by many threads may display lower overall throughput (i.e., are slower; often much slower) than those using the default setting, but have smaller variances in times to obtain locks and guarantee lack of starvation. **Note however, that fairness of locks does not guarantee fairness of thread scheduling.** Thus, one of many threads using a fair lock may obtain it multiple times in succession while other active threads are not progressing and not currently holding the lock. Also note that the untimed tryLock method does not honor the fairness setting. It will succeed if the lock is available even if other threads are waiting.

ReentrantLock **may** also be [**more scalable**](http://lycog.com/concurency/performance-reentrantlock-synchronized/), performing much better under higher contention.

The answer is pretty simple -- use it when you actually need something it provides that synchronized doesn't, like timed lock waits, interruptible lock waits, non-block-structured locks, multiple condition variables, or lock polling. ReentrantLock also has scalability benefits, and you should use it if you actually have a situation that exhibits high contention, but remember that the vast majority of synchronized blocks hardly ever exhibit any contention, let alone high contention. I would advise developing with synchronization until synchronization has proven to be inadequate, rather than simply assuming "the performance will be better" if you use ReentrantLock. Remember, these are advanced tools for advanced users.

A reentrant mutual exclusion Lock with the same basic behaviour and semantics as the implicit monitor lock accessed using synchronized methods and statements, but with extended capabilities.

1. A **ReentrantLock** is owned by the thread last successfully locking, but not yet unlocking it. A thread invoking lock will return, successfully acquiring the lock, when the lock is not owned by another thread. The method will return immediately if the current thread already owns the lock.
2. 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.

*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.**

Acquires the lock if it is not held by another thread and returns immediately with the value true, setting the lock hold count to one. Even when this lock has been set to use a fair ordering policy, a call to tryLock() *will* immediately acquire the lock if it is available, whether or not other threads are currently waiting for the lock. This "barging" behavior can be useful in certain circumstances, even though it breaks fairness. If you want to honor the fairness setting for this lock, then use [tryLock(0, TimeUnit.SECONDS)](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html" \l "tryLock(long,%20java.util.concurrent.TimeUnit))which is almost equivalent (it also detects interruption). If the current thread already holds this lock then the hold count is incremented by one and the method returns true.

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()**.

Code is given below.

**import** java.util.concurrent.TimeUnit;  
**import** java.util.concurrent.locks.Lock;  
**import** java.util.concurrent.locks.ReentrantLock;  
  
**public class ATM** {  
 **private** Lock **lock** = **new** ReentrantLock();  
  
 **public int** withdrawCash(**int** amount) {  
 **int** withdrawnAmt = 0;  
 **boolean tryLockFlag = lock.tryLock();** **if** (tryLockFlag) {  
 **try** {  
 System.***out***.println(Thread.*currentThread*().getName() + **" trying to withdraw cash "** + amount);  
 TimeUnit.***SECONDS***.sleep(5);  
 withdrawnAmt = amount;  
 } **catch** (Exception e) {e.printStackTrace();}

OUTPUT

John trying to withdraw cash 3000

Vidya got money 0

Ramesh got money 0

Nalini got money 0

Kent got money 0

John got money 3000

**finally** {  
 **lock.unlock();** }  
 }  
 **return** withdrawnAmt;  
 }  
 }

**public class ThreadPerson** **extends** Thread {  
 **private** ATM **atm**;  
 **private int amount**;  
 **private** String **name**;  
  
 **public** ThreadPerson(ATM atm, **int** amount, String name) {  
 **super**(name);  
 **this**.**atm** = atm;  
 **this**.**amount** = amount;  
 **this**.**name** = name;  
 }  
  
 @Override  
 **public void** run() {  
 **int** value = **atm**.withdrawCash(**amount**);  
 System.***out***.println(Thread.*currentThread*().getName() + **" got money "** + value);  
 }  
}

**public class TestTryLock1** { 🡸 Test Class is given below  
 **public static void** main(String[] args) {  
 ATM atm = **new** ATM();  
 Thread t1 = **new** ThreadPerson(atm, 3000, **"John"**);  
 Thread t2 = **new** ThreadPerson(atm, 1000, **"Vidya"**);  
 Thread t3 = **new** ThreadPerson(atm, 7000, **"Ramesh"**);  
 Thread t4 = **new** ThreadPerson(atm, 9000, **"Nalini"**);  
 Thread t5 = **new** ThreadPerson(atm, 5000, **"Kent"**);  
  
 t1.start();  
 t2.start();  
 t3.start();  
 t4.start();  
 t5.start();  
 }  
}

**Here, you can see only John got money and others did not get**.

In case of financial transactions or ATM type system, this is not at all recommended approach.

**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 the current thread already holds this lock then the hold count is incremented by one and the method returns true.

If the lock is held by another thread then the current thread becomes disabled for thread scheduling purposes and lies dormant until one of three things happens:

* The lock is acquired by the current thread; or
* Some other thread [interrupts](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#interrupt()) the current thread; or
* The specified waiting time elapses

If the lock is acquired then the value true is returned and the lock hold count is set to one.

If the current thread:

* has its interrupted status set on entry to this method; or
* is [interrupted](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#interrupt()) while acquiring the lock,

then [InterruptedException](https://docs.oracle.com/javase/7/docs/api/java/lang/InterruptedException.html" \o "class in java.lang) is thrown and the current thread's interrupted status is cleared.

If the specified waiting time elapses then the value false is returned. If the time is less than or equal to zero, the method will not wait at all.

In this implementation, as this method is an explicit interruption point, preference is given to responding to the interrupt over normal or reentrant acquisition of the lock, and over reporting the elapse of the waiting time.

Code is given below.

**import** java.util.concurrent.TimeUnit;  
**import** java.util.concurrent.locks.Lock;  
**import** java.util.concurrent.locks.ReentrantLock;  
**public class ATM** {  
 **private** Lock **lock** = **new** ReentrantLock();  
 **public int** withdrawMoney(**int** amount) {  
 **int** withdrawnAmt = 0;  
 **try** {

// **If you increase time to 5, probably 2 persons may get money**  
 **boolean flag = lock.tryLock( 2, TimeUnit.*SECONDS*);**

System.***out***.println(**"What is the value of flag ? "** + flag);  
 **if** (flag) {  
 **try** {  
 System.***out***.println(Thread.*currentThread*().getName() + **" trying to withdraw "** + amount);  
 TimeUnit.***SECONDS***.sleep(5);  
 withdrawnAmt = amount;  
 } **catch** (Exception e) {e.printStackTrace(); }  
 **finally** {  
 **lock.unlock();** }  
 }  
 } **catch** (Exception e) { e.printStackTrace(); }  
 **return** withdrawnAmt;  
 }  
}

OUTPUT

What is the value of flag ? true

Ramesh trying to withdraw 7000

What is the value of flag ? false

Nalini got money 0

What is the value of flag ? false

Kent got money 0

What is the value of flag ? false

Vidya got money 0

What is the value of flag ? false

John got money 0

Ramesh got money 7000

**public class RunnablePerson** **implements** Runnable {  
 **private** ATM **atm**;  
 **private int amount**;  
  
 **public** RunnablePerson(ATM atm, **int** amount) {  
 **this**.**atm** = atm;  
 **this**.**amount** = amount;  
 }  
  
 @Override  
 **public void** run() {  
 **int** value = **atm**.withdrawMoney(**amount**);  
 System.***out***.println(Thread.*currentThread*().getName() + **" got money "** + value);  
 }  
}

**public class TestTryLockParams** { 🡸 Test Class is given below  
 **public static void** main(String[] args) {  
 ATM atm = **new** ATM();  
 Thread t1 = **new** Thread(**new** RunnablePerson(atm, 3000), **"John"**);  
 Thread t2 = **new** Thread(**new** RunnablePerson(atm, 1000), **"Vidya"**);  
 Thread t3 = **new** Thread(**new** RunnablePerson(atm, 7000), **"Ramesh"**);  
 Thread t4 = **new** Thread(**new** RunnablePerson(atm, 9000), **"Nalini"**);  
 Thread t5 = **new** Thread(**new** RunnablePerson(atm, 5000), **"Kent"**);  
 t1.start();  
 t2.start();  
 t3.start();  
 t4.start();  
 t5.start();  
 }  
} **In this case, only Ramesh got money**, if you increase the time for lock, it may be possible that others may get withdrawal amount.

**lockInterruptibly()**

The logic is the same as for all interruptible blocking methods: it allows the thread to immediately react to the interrupt signal sent to it from another thread. How this particular feature is used is up to the application design. For example, it can be used to kill a contingent of threads in a pool which are all waiting to aquire a lock. lockInterruptibly() may block if the lock is already held by another thread and will wait until the lock is aquired. This is the same as with regular lock(). But if another thread interrupts the waiting thread lockInterruptibly() will throw InterruptedException.

**lockInterruptibly() Example**

When the current thread calls lockInterruptibly() and the resource is free, this thread acquires lock and the hold count is incremented by one and returns immediately. If the resource is already held by any other thread, then it will wait until it gets lock or any other thread interrupts it. It means if current thread is waiting for lock but mean while any other thread reaches to acquire lock, then the previous one will be interrupted and returns immediately without acquiring lock. Now find the example of lockInterruptibly().

import java.util.concurrent.locks.ReentrantLock;

public class LockInterruptiblyDemo implements Task{

final ReentrantLock reentrantLock = new ReentrantLock();

@Override

public void performTask() {

try {

reentrantLock.lockInterruptibly();

//if it is not able to acquire lock because of other threads interrupts,

//it will throw InterruptedException and control will go to catch block.

try {

System.out.println(Thread.currentThread().getName() +": Lock acquired.");

System.out.println("Work on progress...");

Thread.sleep(2000);

} finally {

System.out.println(Thread.currentThread().getName() +": Lock released.");

reentrantLock.unlock();

}

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

**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**. Let use the code below.

import java.util.concurrent.TimeUnit;  
public class **CommonTask** {  
  
 public synchronized void perform() {  
 String currentThreadName = Thread.*currentThread*().getName();  
 System.*out*.println(currentThreadName + " started ...");  
 try {  
 System.*out*.println(  
 currentThreadName + " is interrupted " + Thread.currentThread().isInterrupted());  
 TimeUnit.*SECONDS*.sleep(10);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.*out*.println(currentThreadName + " completed ...");  
 }  
}

**public class TaskThread** **implements** Runnable {  
 **private** CommonTask **commonTask**;  
  
 **public** TaskThread(CommonTask commonTask) {  
 **this**.**commonTask** = commonTask;  
 }  
  
 @Override  
 **public void** run() {  
 **commonTask**.perform();  
 }  
}

**import** java.util.concurrent.TimeUnit;  
**public class** Test1 {  
 **public static void** main(String[] args) {  
 CommonTask commonTask = **new** CommonTask();  
 Thread t1 = **new** Thread(**new** TaskThread(commonTask), **"Thread-1"**);  
 Thread t2 = **new** Thread(**new** TaskThread(commonTask), **"Thread-2"**);  
 Thread t3 = **new** Thread(**new** TaskThread(commonTask), **"Thread-3"**);  
  
 t1.start();  
 t2.start();  
 t3.start();  
  
 **boolean** flag = **true**;  
 **while** (flag) {  
 **try** {  
 System.***out***.println(**"Running ..."**);  
 TimeUnit.***SECONDS***.sleep(5);  
 **if (t3.isAlive()) t3.interrupt();**  
 flag = **false**;  
 } **catch** (InterruptedException e) { e.printStackTrace(); }  
 }  
 }  
}

OUTPUT

Running ...

Thread-1 started ...

Thread-1 is interrupted false

Thread-1 completed ...

Thread-3 started ...

Thread-3 is interrupted true

java.lang.InterruptedException: sleep interrupted

at java.lang.Thread.sleep(Native Method)

at java.lang.Thread.sleep(Thread.java:340)

at java.util.concurrent.TimeUnit.sleep(TimeUnit.java:386)

at com.ddlab.rnd.interrupt.CommonTask.perform(CommonTask.java:13)

at com.ddlab.rnd.interrupt.TaskThread.run(TaskThread.java:12)

at java.lang.Thread.run(Thread.java:748)

Thread-3 completed ...

Thread-2 started ...

Thread-2 is interrupted false

Thread-2 completed ...

As you can see that you cannot interrupt a thread waiting to acquire the lock using synchronized keyword.

Now let us try using ReentrantLock. Code is given below. This is possible using **lock.lockInterruptibly()**.

**import** java.util.concurrent.TimeUnit;  
**import** java.util.concurrent.locks.Lock;  
**import** java.util.concurrent.locks.ReentrantLock;  
  
**public class CommonTask1** {  
 **private ReentrantLock lock = new ReentrantLock();**  
  
 **public void** perform() {  
 String currentThreadName = Thread.*currentThread*().getName();  
 System.***out***.println(currentThreadName + **" started ..."**);  
 **try** {  
 **lock.lockInterruptibly();** System.out.println(currentThreadName + " holds lock " + lock.isHeldByCurrentThread());  
 System.***out***.println(  
 "Is " + currentThreadName + " interrupted ? " + Thread.currentThread().isInterrupted());  
 TimeUnit.***SECONDS***.sleep(10);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 } **finally** {  
 **if (lock.isHeldByCurrentThread()) {  
 lock.unlock();  
 }** }  
 System.***out***.println(Thread.*currentThread*().getName() + **" completed ..."**);  
 }  
}

**public class TaskThread1** **implements** Runnable {  
 **private** CommonTask1 **commonTask1**;  
  
 **public** TaskThread1(CommonTask1 commonTask1) {  
 **this**.**commonTask1** = commonTask1;  
 }  
  
 @Override  
 **public void** run() {  
 **commonTask1**.perform();  
 }  
}

**import** java.util.concurrent.TimeUnit;  
**public class** Test2 {  
 **public static void** main(String[] args) {  
 CommonTask1 commonTask1 = **new** CommonTask1();  
 Thread t1 = **new** Thread(**new** TaskThread1(commonTask1), **"Thread-1"**);  
 Thread t2 = **new** Thread(**new** TaskThread1(commonTask1), **"Thread-2"**);  
 Thread t3 = **new** Thread(**new** TaskThread1(commonTask1), **"Thread-3"**);  
  
 t1.start();  
 t2.start();  
 t3.start();  
  
 **boolean** flag = **true**;  
 **while** (flag) {  
 **try** {  
 System.***out***.println(**"Running ..."**);  
 TimeUnit.***SECONDS***.sleep(5);  
 **if** (t3.isAlive()) t3.interrupt();  
 flag = **false**;  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

OUTPUT

Running ...

Thread-2 started ...

Thread-1 started ...

Thread-1 holds lock true

Is Thread-1 interrupted ? false

Thread-3 started ...

java.lang.InterruptedException

at java.util.concurrent.locks.AbstractQueuedSynchronizer.doAcquireInterruptibly(AbstractQueuedSynchronizer.java:898)

at java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireInterruptibly(AbstractQueuedSynchronizer.java:1222)

at java.util.concurrent.locks.ReentrantLock.lockInterruptibly(ReentrantLock.java:335)

at com.ddlab.rnd.interrupt.CommonTask1.perform(CommonTask1.java:14)

at com.ddlab.rnd.interrupt.TaskThread1.run(TaskThread1.java:12)

at java.lang.Thread.run(Thread.java:748)

Thread-3 completed ...

Thread-2 holds lock true

Is Thread-2 interrupted ? false

Thread-1 completed ...

Thread-2 completed ...

Another 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**.

Using synchronized approach

**public class** ATM {  
  
 **public static volatile boolean** *isCashOver* = **true**;  
 **private** Lock **lock** = **new** ReentrantLock();  
  
 **public synchronized void** withdrawCash(**int** amount) {  
**try** {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) {  
 e.printStackTrace();  
 }  
}  
}

**public class** Person **extends** Thread {  
 **private** ATM **atm**;  
 **private int amount**;  
  
 **public** Person(ATM atm, String name, **int** amount) {  
 **this**.**atm** = atm;  
 **this**.**amount** = amount;  
 **super**.setName(name);  
 }  
  
 **public void** run() {  
 **atm**.withdrawCash(**this**.**amount**);  
 }  
}

**public class** TestLockInterruptibly {  
 **public static void** main(String[] args) {  
 ATM atm = **new** ATM();  
 Person p1 = **new** Person(atm, **"John"**, 3000);  
 Person p2 = **new** Person(atm, **"Vidya"**, 2000);  
 Person p3 = **new** Person(atm, **"Hati"**, 1000);  
  
 p1.start();  
 p2.start();  
 p3.start();  
  
 **boolean** flag = **true**;  
 **while**(flag) {  
 flag = ATM.*isCashOver*;  
 **if**(flag == **false**) {  
 p2.interrupt();  
 p3.interrupt();  
 }  
 }  
 System.***out***.println(**"Completed ..."**);  
  
 }  
}

**OUTPUT**

John trying to withdraw cash 3000

John got 3000

Completed ...

Hati trying to withdraw cash 1000

Vidya trying to withdraw cash 2000

java.lang.InterruptedException: sleep interrupted

at java.base/java.lang.Thread.sleep(Native Method)

The ATM class can be rewritten as

**public class** ATM {  
  
 **public static volatile boolean** *isCashOver* = **true**;  
 **private** Lock **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) {  
 e.printStackTrace();  
 }  
 **finally** {  
 **lock**.unlock();  
 }  
 }  
}

**OUTPUT**

Vidya trying to withdraw cash 2000

Vidya got 2000

Completed ...

John trying to withdraw cash 3000

java.lang.InterruptedException: sleep interrupted

at java.base/java.lang.Thread.sleep(Native Method)

**lockInterruptibly**

public void lockInterruptibly() throws [InterruptedException](https://docs.oracle.com/javase/7/docs/api/java/lang/InterruptedException.html)

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 purposes and lies dormant until one of two things happens:

* The lock is acquired by the current thread; or
* Some other thread [interrupts](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#interrupt()) the current thread.

If the lock is acquired by the current thread then the lock hold count is set to one.

If the current thread:

* has its interrupted status set on entry to this method; or
* is [interrupted](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#interrupt()) while acquiring the lock,

then [InterruptedException](https://docs.oracle.com/javase/7/docs/api/java/lang/InterruptedException.html" \o "class in java.lang) is thrown and the current thread's interrupted status is cleared.

In this implementation, as this method is an explicit interruption point, preference is given to responding to the interrupt over normal or reentrant acquisition of the lock.

A blocking operation can be interrupted only if it is declared to throw InterruptedException. Clearly, a synchronized block does not declare it, therefore it is impossible to interrupt a thread while it is waiting to acquire a lock. Alternatively you can use an explicit lock and call Lock.lockInterruptibly().

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

Let us see the code below.

**import** java.util.concurrent.TimeUnit;  
**public class Bank** {  
 **public synchronized void depositMoney**(**int** amount) {  
 **try** {  
 System.***out***.println(Thread.*currentThread*().getName() + **" depositing money of Rs "** + amount);  
 TimeUnit.***SECONDS***.sleep(10);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(Thread.*currentThread*().getName() + **" deposited money ..."**);  
 }  
  
 **public synchronized void withdrawCash**(**int** amount) {  
 **try** {  
 System.***out***.println(Thread.*currentThread*().getName() + **" withdrawing money of Rs "** + amount);  
 TimeUnit.***SECONDS***.sleep(5);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(Thread.*currentThread*().getName() + **" withdrew money ..."**);  
 }  
}

**public class Depositor** **implements** Runnable {  
 **private** Bank **bank**;  
 **private int amount**;  
  
 **public** Depositor(Bank bank, **int** amount) {  
 **this**.**bank** = bank;  
 **this**.**amount** = amount;  
 }  
  
 @Override  
 **public void** run() {  
 **bank**.depositMoney(**amount**);  
 }  
}

**public class Customer** **implements** Runnable {  
 **private** Bank **bank**;  
 **private int amount**;  
  
 **public** Customer(Bank bank, **int** amount) {  
 **this**.**bank** = bank;  
 **this**.**amount** = amount;  
 }  
  
 @Override  
 **public void** run() {  
 **bank**.withdrawCash(**amount**);  
 }  
}

**public class TestBank** { 🡸 Test Program is given below  
 **public static void** main(String[] args) {  
 Bank bank = **new** Bank();  
 Runnable target;  
 Thread t1 = **new** Thread(**new** Depositor(bank, 3000), **"John"**);  
 Thread t2 = **new** Thread(**new** Customer(bank, 3000), **"Ram"**);  
 t1.start();  
 t2.start();  
  
 }  
}

OUTPUT

John depositing money of Rs 3000

John deposited money ...

Ram withdrawing money of Rs 3000

Ram withdrew money ...

In this case, only one thread can access either deposit(0 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.

**import** java.util.concurrent.TimeUnit;  
**import** java.util.concurrent.locks.Lock;  
**import** java.util.concurrent.locks.ReentrantLock;  
  
**public class** Bank {  
 **private** Lock **lock** = **new** ReentrantLock(**true**);  
  
 **public void** depositMoney(**int** amount) {  
 **try** {  
 System.***out***.println(Thread.*currentThread*().getName() + **" depositing money of Rs "** + amount);  
 **lock**.lock();  
 TimeUnit.***SECONDS***.sleep(10);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 } **finally** {  
 **lock**.unlock();  
 }  
 System.***out***.println(Thread.*currentThread*().getName() + **" deposited money ..."**);  
 }  
  
 **public void** withdrawCash(**int** amount) {  
 **try** {  
 System.***out***.println(Thread.*currentThread*().getName() + **" withdrawing money of Rs "** + amount);  
 **lock.unlock();** TimeUnit.***SECONDS***.sleep(5);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 } **finally** {  
 **lock.unlock();** }  
 System.***out***.println(Thread.*currentThread*().getName() + **" withdrew money ..."**);  
 }  
}

OUTPUT

John depositing money of Rs 3000

Ram withdrawing money of Rs 3000

java.lang.IllegalMonitorStateException

at java.util.concurrent.locks.ReentrantLock$Sync.tryRelease(ReentrantLock.java:151)

at java.util.concurrent.locks.AbstractQueuedSynchronizer.release(AbstractQueuedSynchronizer.java:1261)

at java.util.concurrent.locks.ReentrantLock.unlock(ReentrantLock.java:457)

at com.ddlab.rnd.reentrant.Bank.withdrawCash(Bank.java:26)

So, it means that you can not solve this problem simply by replacing synchronized keyword with ReentrantLock. Let us rewrite the Bank class and let us see what happens.

**import** java.util.concurrent.TimeUnit;  
**import** java.util.concurrent.locks.Lock;  
**import** java.util.concurrent.locks.ReadWriteLock;  
**import** java.util.concurrent.locks.ReentrantReadWriteLock;  
  
**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() + **" withdrew money ..."**);  
 }  
}

OUTPUT

John depositing money of Rs 3000

🡸It waits for the thread John to complete operation.

John deposited money ...

Ram withdrawing money of Rs 3000

Ram withdrew money ...

So it is clear that even if you use ReentrantLock or ReadWriteLock, both the methods can not be accessed at the same time. But if there is another thread to draw money, the withdraw method will work.

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

**import** java.util.concurrent.TimeUnit;  
**import** java.util.concurrent.locks.ReadWriteLock;  
**import** java.util.concurrent.locks.ReentrantReadWriteLock;  
  
**public class** Bank {  
 **private** ReadWriteLock **readWriteLock** = **new** ReentrantReadWriteLock();  
  
 **public void** depositMoney(**int** amount) {  
 **try** {  
 **readWriteLock**.writeLock().lock();  
 System.***out***.println(Thread.*currentThread*().getName() + **" depositing money of Rs "** + amount);  
 TimeUnit.***SECONDS***.sleep(10);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 } **finally** {  
 **readWriteLock**.writeLock().unlock();  
 }  
 System.***out***.println(Thread.*currentThread*().getName() + **" deposited money ..."**);  
 }  
  
 **public void** withdrawCash(**int** amount) {  
 **try** {  
 **readWriteLock**.readLock().lock();  
 System.***out***.println(Thread.*currentThread*().getName() + **" withdrawing money of Rs "** + amount);  
 TimeUnit.***SECONDS***.sleep(5);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 } **finally** {  
 **readWriteLock**.readLock().unlock();  
 }  
 System.***out***.println(Thread.*currentThread*().getName() + **" got money ..."**);  
 }  
}

You can have the following structure.

**public class** Test11 {  
 **public static void** main(String[] args) {  
 Bank bank = **new** Bank();  
 Runnable target;  
 Thread t1 = **new** Thread(**new** Depositor(bank, 3000), **"John"**);  
 Thread t2 = **new** Thread(**new** Customer(bank, 3000), **"Ram"**);  
 Thread t3 = **new** Thread(**new** Customer(bank, 5000), **"Shyam"**);  
 t1.start();  
 t2.start();  
 t3.start();  
  
 }  
}

# What is ReentrantLock in Java

ReentrantLock is mutual exclusive lock, similar to implicit locking provided by synchronized keyword in Java, with extended feature like **fairness, which can be used to provide lock to longest waiting thread**. Lock is acquired by **lock()** method and held by Thread until a call to unlock() method. **Fairness parameter is provided while creating instance of ReentrantLock in constructor**. ReentrantLock provides same visibility and ordering guarantee, provided by implicitly locking, which means, unlock() happens before another thread get lock().

# Difference between ReentrantLock and synchronized keyword in Java

Though ReentrantLock provides same visibility and orderings guaranteed as implicit lock, acquired by synchronized keyword in Java, it provides more functionality and differs in certain aspect. As stated earlier, main difference between synchronized and ReentrantLock is ability to trying for lock interruptibly, and with timeout. Thread doesn’t need to block infinitely, which was the case with synchronized. Let’s see few more differences between synchronized and Lock in Java.

1) Another significant difference between **ReentrantLock and synchronized keyword is fairness**. **synchronized keyword doesn't support fairness**. Any thread can acquire lock once released, no preference can be specified, on the other hand you can make ReentrantLock fair by specifying fairness property, while creating instance of ReentrantLock. Fairness property provides lock to longest waiting thread, in case of contention.

2) Second difference between synchronized and Reentrant lock is tryLock() method. ReentrantLock provides convenient tryLock() method, which acquires lock only if its available or not held by any other thread. This reduces blocking of thread waiting for lock in Java application.

3) One more worth noting difference between ReentrantLock and synchronized keyword in Java is, ability to interrupt Thread while waiting for Lock. In case of synchronized keyword, a thread can be blocked waiting for lock, for an indefinite period of time and there was no way to control that. ReentrantLock provides a method called **lockInterruptibly()**, which can be used to interrupt thread when it is waiting for lock. Similarly tryLock() with timeout can be used to timeout if lock is not available in certain time period.

4) ReentrantLock also provides convenient method to get List of all threads waiting for lock.

So, you can see, lot of significant differences between synchronized keyword and ReentrantLock in Java. In short, Lock interface adds lot of power and flexibility and allows some control over lock acquisition process, which can be leveraged to write highly scalable systems in Java.

# Benefits of ReentrantLock in Java

Most of the benefits derive from the differences covered between synchronized vs ReentrantLock in last section. Here is summary of benefits offered by ReentrantLock over synchronized in Java:

**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.**

# Disadvantages of ReentrantLock in Java

Major drawback of using ReentrantLock in Java is wrapping method body inside try-finally block, which makes code unreadable and hides business logic. It’s really cluttered and I hate it most, though IDE like Eclipse and Netbeans can add those try catch block for you. Another disadvantage is that, now programmer is responsible for acquiring and releasing lock, which is a power but also opens gate for new subtle bugs, when programmer forget to release the lock in finally block.

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

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

**Reentrant Locks**

**Java Reentrant Locks**

Java has a few lock implementations in the java.util.concurrent.locks package.

The general classes of locks are nicely laid out as interfaces:

* **Lock** – the simplest case of a lock which can be acquired and released
* **ReadWriteLock** – a lock implementation that has both read and write lock types – multiple read locks can be held at a time unless the exclusive write lock is held

Java provides two implementations of these locks that we care about – both of which are reentrant (this just means a thread can reacquire the same lock multiple times without any issue).

* **ReentrantLock** – as you’d expect, a reentrant Lock implementation
* **ReentrantReadWriteLock** – a reentrant ReadWriteLock implementation

Now, let’s see some examples.

**A Read/Write Lock Example**

So how does one use a lock? It’s pretty simple: just acquire and release (and never forget to release – finally is your friend!).

Imagine we have a very simple case where we need to synchronize access to a pair of variables. One is a simple value and another is derived based on some lengthy calculation. First, this is how we would perform that with the synchronized keyword.

public class Calculator {

private int calculatedValue;

private int value;

public synchronized void calculate(int value) {

this.value = value;

this.calculatedValue = doMySlowCalculation(value);

}

public synchronized int getCalculatedValue() {

return calculatedValue;

}

public synchronized int getValue() {

return value;

}

}

Simple, but if we have a lot of contention or if we perform a lot of reads and few writes, synchronization could hurt performance. Since frequently reads occur a lot more often than writes, Using a **ReadWriteLock** helps us minimize the issue:

public class Calculator {

private int calculatedValue;

private int value;

private ReadWriteLock lock = new ReentrantReadWriteLock();

public void calculate(int value) {

lock.writeLock().lock();

try {

this.value = value;

this.calculatedValue = doMySlowCalculation(value);

} finally {

lock.writeLock().unlock();

}

}

public int getCalculatedValue() {

lock.readLock().lock();

try {

return calculatedValue;

} finally {

lock.readLock().unlock();

}

}

public int getValue() {

lock.readLock().lock();

try {

return value;

} finally {

lock.readLock().unlock();

}

}

}

This example actually shows one big advantage using synchronized has: it is concise and more foolproof than using explicit locks. But locks give use flexibility we wouldn’t otherwise have.

In the example above, we can have hundreds of threads reading the same value at once with no issue, and we only block readers when we acquire the write lock. Remember that: many readers can acquire the read lock at the same time, but there are no readers OR writers allowed when acquiring the write lock.