<http://tutorials.jenkov.com/java-concurrency/deadlock.html>

# Deadlock

* [Thread Deadlock](http://tutorials.jenkov.com/java-concurrency/deadlock.html#thread-deadlock)
* [Deadlock Tutorial Video](http://tutorials.jenkov.com/java-concurrency/deadlock.html#deadlock-tutorial-video)
* [Deadlock Example](http://tutorials.jenkov.com/java-concurrency/deadlock.html#deadlock-example)
* [Database Deadlocks](http://tutorials.jenkov.com/java-concurrency/deadlock.html#database-deadlocks)

## Thread Deadlock

A deadlock is when two or more threads are blocked waiting to obtain locks that some of the other threads in the deadlock are holding. Deadlock can occur when multiple threads need the same locks, at the same time, but obtain them in different order.

## Deadlock Tutorial Video

If you prefer video, I have a video version of this deadlock tutorial here: [**Deadlock in Java**](https://www.youtube.com/watch?v=3cgZbACBpxI&list=PLL8woMHwr36EDxjUoCzboZjedsnhLP1j4&index=14)

**[](https://www.youtube.com/watch?v=3cgZbACBpxI&list=PLL8woMHwr36EDxjUoCzboZjedsnhLP1j4&index=14)**

## Deadlock Example

Below is an example of a deadlock situation:

If thread 1 locks A, and tries to lock B, and thread 2 has already locked B, and tries to lock A, a deadlock arises. Thread 1 can never get B, and thread 2 can never get A. In addition, neither of them will ever know. They will remain blocked on each their object, A and B, forever. This situation is a deadlock.

The situation is illustrated below:

Thread 1 locks A, waits for B

Thread 2 locks B, waits for A

Here is an example of a TreeNode class that call synchronized methods in different instances:

public class TreeNode {

TreeNode parent = null;

List children = new ArrayList();

public synchronized void addChild(TreeNode child){

if(!this.children.contains(child)) {

this.children.add(child);

child.setParentOnly(this);

}

}

public synchronized void addChildOnly(TreeNode child){

if(!this.children.contains(child){

this.children.add(child);

}

}

public synchronized void setParent(TreeNode parent){

this.parent = parent;

parent.addChildOnly(this);

}

public synchronized void setParentOnly(TreeNode parent){

this.parent = parent;

}

}

If a thread (1) calls the parent.addChild(child) method at the same time as another thread (2) calls the child.setParent(parent) method, on the same parent and child instances, a deadlock can occur. Here is some pseudo code that illustrates this:

Thread 1: parent.addChild(child); //locks parent

--> child.setParentOnly(parent);

Thread 2: child.setParent(parent); //locks child

--> parent.addChildOnly()

First thread 1 calls parent.addChild(child). Since addChild() is synchronized thread 1 effectively locks the parent object for access from other treads.

Then thread 2 calls child.setParent(parent). Since setParent() is synchronized thread 2 effectively locks the child object for acces from other threads.

Now both child and parent objects are locked by two different threads. Next thread 1 tries to call child.setParentOnly() method, but the child object is locked by thread 2, so the method call just blocks. Thread 2 also tries to call parent.addChildOnly() but the parent object is locked by thread 1, causing thread 2 to block on that method call. Now both threads are blocked waiting to obtain locks the other thread holds.

Note: The two threads must call parent.addChild(child) and child.setParent(parent) at the same time as described above, and on the same two parent and child instances for a deadlock to occur. The code above may execute fine for a long time until all of a sudden it deadlocks.

The threads really need to take the locks \*at the same time\*. For instance, if thread 1 is a bit ahead of thread2, and thus locks both A and B, then thread 2 will be blocked already when trying to lock B. Then no deadlock occurs. Since thread scheduling often is unpredictable there is no way to predict \*when\* a deadlock occurs. Only that it \*can\* occur.

## More Complicated Deadlocks

Deadlock can also include more than two threads. This makes it harder to detect. Here is an example in which four threads have deadlocked:

Thread 1 locks A, waits for B

Thread 2 locks B, waits for C

Thread 3 locks C, waits for D

Thread 4 locks D, waits for A

Thread 1 waits for thread 2, thread 2 waits for thread 3, thread 3 waits for thread 4, and thread 4 waits for thread 1.

## Database Deadlocks

A more complicated situation in which deadlocks can occur, is a database transaction. A database transaction may consist of many SQL update requests. When a record is updated during a transaction, that record is locked for updates from other transactions, until the first transaction completes. Each update request within the same transaction may therefore lock some records in the database.

If multiple transactions are running at the same time that need to update the same records, there is a risk of them ending up in a deadlock.

For example

Transaction 1, request 1, locks record 1 for update

Transaction 2, request 1, locks record 2 for update

Transaction 1, request 2, tries to lock record 2 for update.

Transaction 2, request 2, tries to lock record 1 for update.

Since the locks are taken in different requests, and not all locks needed for a given transaction are known ahead of time, it is hard to detect or prevent deadlocks in database transactions.

<http://tutorials.jenkov.com/java-concurrency/deadlock-prevention.html>

# Deadlock Prevention

* [Deadlock Prevention Tutorial Video](http://tutorials.jenkov.com/java-concurrency/deadlock-prevention.html#deadlock-prevention-tutorial-video)
* [GitHub Repo](http://tutorials.jenkov.com/java-concurrency/deadlock-prevention.html#github-repo)
* [Lock Ordering](http://tutorials.jenkov.com/java-concurrency/deadlock-prevention.html#ordering)
* [Lock Timeout](http://tutorials.jenkov.com/java-concurrency/deadlock-prevention.html#timeout)
* [Deadlock Detection](http://tutorials.jenkov.com/java-concurrency/deadlock-prevention.html#detection)

In some situations it is possible to prevent deadlocks. I'll describe three techniques in this text:

1. [**Lock Ordering**](http://tutorials.jenkov.com/java-concurrency/deadlock-prevention.html#ordering)
2. [**Lock Timeout**](http://tutorials.jenkov.com/java-concurrency/deadlock-prevention.html#timeout)
3. [**Deadlock Detection**](http://tutorials.jenkov.com/java-concurrency/deadlock-prevention.html#detection)

## Deadlock Prevention Tutorial Video

If you prefer video, I have a video version of this deadlock prevention tutorial here: [**Deadlock Prevention in Java**](https://www.youtube.com/watch?v=6E3aYf3jXdk&list=PLL8woMHwr36EDxjUoCzboZjedsnhLP1j4&index=17).

**[](https://www.youtube.com/watch?v=6E3aYf3jXdk&list=PLL8woMHwr36EDxjUoCzboZjedsnhLP1j4&index=17)**

## GitHub Repo

You can see code examples for the solutions discussed in this tutorial (as discussed in the video at least) in the following GitHub repo:

[**https://github.com/jjenkov/java-examples**](https://github.com/jjenkov/java-examples)

## Lock Ordering

Deadlock occurs when multiple threads need the same locks but obtain them in different order.

If you make sure that all locks are always taken in the same order by any thread, deadlocks cannot occur. Look at this example:

Thread 1:

lock A

lock B

Thread 2:

wait for A

lock C (when A locked)

Thread 3:

wait for A

wait for B

wait for C

If a thread, like Thread 3, needs several locks, it must take them in the decided order. It cannot take a lock later in the sequence until it has obtained the earlier locks.

For instance, neither Thread 2 or Thread 3 can lock C until they have locked A first. Since Thread 1 holds lock A, Thread 2 and 3 must first wait until lock A is unlocked. Then they must succeed in locking A, before they can attempt to lock B or C.

Lock ordering is a simple yet effective deadlock prevention mechanism. However, it can only be used if you know about all locks needed ahead of taking any of the locks. This is not always the case.

## Lock Timeout

Another deadlock prevention mechanism is to put a timeout on lock attempts meaning a thread trying to obtain a lock will only try for so long before giving up. If a thread does not succeed in taking all necessary locks within the given timeout, it will backup, free all locks taken, wait for a random amount of time and then retry. The random amount of time waited serves to give other threads trying to take the same locks a chance to take all locks, and thus let the application continue running without locking.

Here is an example of two threads trying to take the same two locks in different order, where the threads back up and retry:

Thread 1 locks A

Thread 2 locks B

Thread 1 attempts to lock B but is blocked

Thread 2 attempts to lock A but is blocked

Thread 1's lock attempt on B times out

Thread 1 backs up and releases A as well

Thread 1 waits randomly (e.g. 257 millis) before retrying.

Thread 2's lock attempt on A times out

Thread 2 backs up and releases B as well

Thread 2 waits randomly (e.g. 43 millis) before retrying.

In the above example Thread 2 will retry taking the locks about 200 millis before Thread 1 and will therefore likely succeed at taking both locks. Thread 1 will then wait already trying to take lock A. When Thread 2 finishes, Thread 1 will be able to take both locks too (unless Thread 2 or another thread takes the locks in between).

An issue to keep in mind is, that just because a lock times out it does not necessarily mean that the threads had deadlocked. It could also just mean that the thread holding the lock (causing the other thread to time out) takes a long time to complete its task.

Additionally, if enough threads compete for the same resources they still risk trying to take the threads at the same time again and again, even if timing out and backing up. This may not occur with 2 threads each waiting between 0 and 500 millis before retrying, but with 10 or 20 threads the situation is different. Then the likeliness of two threads waiting the same time before retrying (or close enough to cause problems) is a lot higher.

A problem with the lock timeout mechanism is that it is not possible to set a timeout for entering a synchronized block in Java. You will have to create a custom lock class or use one of the Java 5 concurrency constructs in the java.util.concurrency package. Writing custom locks isn't difficult but it is outside the scope of this text. Later texts in the Java concurrency trails will cover custom locks.

## Deadlock Detection

Deadlock detection is a heavier deadlock prevention mechanism aimed at cases in which lock ordering isn't possible, and lock timeout isn't feasible.

Every time a thread **takes** a lock it is noted in a data structure (map, graph etc.) of threads and locks. Additionally, whenever a thread **requests** a lock this is also noted in this data structure.

When a thread requests a lock but the request is denied, the thread can traverse the lock graph to check for deadlocks. For instance, if a Thread A requests lock 7, but lock 7 is held by Thread B, then Thread A can check if Thread B has requested any of the locks Thread A holds (if any). If Thread B has requested so, a deadlock has occurred (Thread A having taken lock 1, requesting lock 7, Thread B having taken lock 7, requesting lock 1).

Of course a deadlock scenario may be a lot more complicated than two threads holding each others locks. Thread A may wait for Thread B, Thread B waits for Thread C, Thread C waits for Thread D, and Thread D waits for Thread A. In order for Thread A to detect a deadlock it must transitively examine all requested locks by Thread B. From Thread B's requested locks Thread A will get to Thread C, and then to Thread D, from which it finds one of the locks Thread A itself is holding. Then it knows a deadlock has occurred.

Below is a graph of locks taken and requested by 4 threads (A, B, C and D). A data structure like this that can be used to detect deadlocks.

|  |
| --- |
| Deadlock Detection Data Structure |

So what do the threads do if a deadlock is detected?

One possible action is to release all locks, backup, wait a random amount of time and then retry. This is similar to the simpler lock timeout mechanism except threads only backup when a deadlock has actually occurred. Not just because their lock requests timed out. However, if a lot of threads are competing for the same locks they may repeatedly end up in a deadlock even if they back up and wait.

A better option is to determine or assign a priority of the threads so that only one (or a few) thread backs up. The rest of the threads continue taking the locks they need as if no deadlock had occurred. If the priority assigned to the threads is fixed, the same threads will always be given higher priority. To avoid this you may assign the priority randomly whenever a deadlock is detected.

# [What is the meaning of “ReentrantLock” in Java?](https://stackoverflow.com/questions/16504231/what-is-the-meaning-of-reentrantlock-in-java)

<https://stackoverflow.com/questions/16504231/what-is-the-meaning-of-reentrantlock-in-java>

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

Since an intrinsic lock is held by a thread, doesn't it mean that a thread run once equals an invocation basis?

Thank you, it seems mean that: in a thread,if I get a lock lockA when process function doA which call function doB, and doB also need a lock lockA,then there wil be a reentrancy. In Java, this phenomenon is acquired per thread, so I needn't consider deadlocks?

A1: Reentrancy means that locks are acquired on a per-thread rather than per-invocation basis.

That is a misleading definition. It is true (sort of), but it misses the real point.

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();

...

synchronized (lock) {

...

doSomething(lock, ...)

...

}

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

synchronized (lock) {

...

}

}

The alternative to reentrant is non-reentrant locking, where it would be an error for a thread to attempt to acquire a lock that it already holds.

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.

=============================================================================

A2: Imagine something like this:

function A():

lock (X)

B()

unlock (X)

function B():

A()

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. So if we wrote the above pseudocode in Java, it would not deadlock.