**Deadlock :** In Java, a [deadlock](https://en.wikipedia.org/wiki/Deadlock" \t "https://howtodoinjava.com/java/multi-threading/writing-a-deadlock-and-resolving-in-java/_blank) is a situation where minimum two threads are holding the lock on some different resource, and both are waiting for other’s resource to complete its task.

Deadlock can occur when multiple threads need the same locks, at the same time, but obtain them in different order.

Thread-1 has resource A but need B to complete processing and similarly Thread-2 has resource B but need A first.

Example :

**package** com.gopal.multithreading;

**public** **class** DeadLock {

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

DeadLock deadLock = **new** DeadLock();

**final** A a = deadLock.**new** A();

**final** B b = deadLock.**new** B();

// Thread-1

Runnable block1 = **new** Runnable() {

**public** **void** run() {

**synchronized** (a) {

**try** {

// Adding delay so that both threads can start trying to lock resources

Thread.*sleep*(100);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

// Thread-1 have A but need B also

**synchronized** (b) {

System.***out***.println("In block 1");

}

}

}

};

// Thread-2

Runnable block2 = **new** Runnable() {

**public** **void** run() {

**synchronized** (b) {

// Thread-2 have B but need A also

**synchronized** (a) {

System.***out***.println("In block 2");

}

}

}

};

**new** Thread(block1).start();

**new** Thread(block2).start();

}

// Resource A

**private** **class** A {

**private** **int** i = 10;

}

// Resource B

**private** **class** B {

**private** **int** i = 20;

}

}

**Example :** 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.

Thread 1 locks A, waits for B

Thread 2 locks B, waits for A

**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.

**How to avoid deadlock :**

we will simply re-order the statements where the code is accessing shared resources.

**DeadLock Prevention :**

In Java, a deadlock is a programming situation where two or more threads are blocked forever. A deadlock condition will occur with at least two threads and two or more resources.

**How To Avoid Deadlock**

* **Avoid Nested Locks:** A deadlock mainly happens when we give locks to multiple threads. Avoid giving a lock to multiple threads if we already have given to one.
* **Avoid Unnecessary Locks:** We can have a lock only those members which are required. Having a lock unnecessarily can lead to a deadlock.
* **Using Thread.join():** A deadlock condition appears when one thread is waiting other to finish. If this condition occurs we can use Thread.join() with the maximum time the execution will take.

**Object level lock vs Class level lock in Java :**

In Java, a ****synchronized**** block of code can only be executed by one thread at a time. Also, java supports multiple threads to be executed concurrently.

we can use **synchronized keyword** in the class on defined methods or blocks. synchronized keyword can not be used with variables or attributes in class definition.

**Object level lock in Java : **Object level lock**** is mechanism when we want to synchronize a ****non-static method**** or ****non-static code block**** such that only one thread will be able to execute the code block on given instance of the class. This should always be done ****to make instance level data thread safe****.

**public** **class** DemoClass{

**public** **synchronized** **void** demoMethod(){}

}

*or*

**public** **class** DemoClass{

**public** **void** demoMethod(){

**synchronized** (**this**) {

//other thread safe code

}

}

}

*or*

**public** **class** DemoClass{

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

**public** **void** demoMethod(){

**synchronized** (lock) {

//other thread safe code

}

}}

**Class level lock in Java :** Class level lock prevents multiple threads to enter in synchronized block in any of all available instances of the class on runtime. This means if in runtime there are 100 instances of DemoClass, then only one thread will be able to execute demoMethod() in any one of instance at a time, and all other instances will be locked for other threads.

Class level locking should always be done to make static data thread safe. As we know that [static](https://howtodoinjava.com/java/keywords/java-static-keyword/) keyword associate data of methods to class level, so use locking at static fields or methods to make it on class level.

**public** **class** DemoClass{

//Method is static

**public** **synchronized** **static** **void** demoMethod(){

}

}

or

**public** **class** DemoClass{

**public** **void** demoMethod() {

//Acquire lock on .class reference

**synchronized** (DemoClass.**class**) {

//other thread safe code

}

}

}

or

**public** **class** DemoClass{

**private** **final** **static** Object lock = **new** Object();

**public** **void** demoMethod() {

//Lock object is static

**synchronized** (lock) {

//other thread safe code

}

}

}

* Synchronization in Java guarantees that no two threads can execute a synchronized method, which requires same lock, simultaneously or concurrently.
* synchronized keyword can be used only with methods and code blocks. These methods or blocks can be static or non-static both.
* When ever a thread enters into Java synchronized method or block it acquires a lock and whenever it leaves synchronized method or block it releases the lock. Lock is released even if thread leaves synchronized method after completion or due to any Error or Exception.
* Java synchronized keyword is re-entrant in nature it means if a synchronized method calls another synchronized method which requires same lock then current thread which is holding lock can enter into that method without acquiring lock.
* It’s possible that both static synchronized and non static synchronized method can run simultaneously or concurrently because they lock on different object.
* According to the Java language specification you can not use synchronized keyword with constructor. It is illegal and result in compilation error.