A deadlock occurs in a multithreaded program when two or more threads are unable to proceed because each is waiting for the other to release a resource.

public class DeadlockExample {  
  
 public static void main(String[] args) {  
 *// Two resources* Object resource1 = new Object();  
 Object resource2 = new Object();  
  
 *// Thread 1* Thread thread1 = new Thread(() -> {  
 synchronized (resource1) {  
 System.*out*.println("Thread 1: Holding resource 1...");  
  
 try {  
 Thread.*sleep*(100); *// Simulate some work* } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 System.*out*.println("Thread 1: Waiting for resource 2...");  
 synchronized (resource2) {  
 System.*out*.println("Thread 1: Got resource 2!");  
 }  
 }  
 });  
  
 *// Thread 2* Thread thread2 = new Thread(() -> {  
 synchronized (resource2) {  
 System.*out*.println("Thread 2: Holding resource 2...");  
  
 try {  
 Thread.*sleep*(100); *// Simulate some work* } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 System.*out*.println("Thread 2: Waiting for resource 1...");  
 synchronized (resource1) {  
 System.*out*.println("Thread 2: Got resource 1!");  
 }  
 }  
 });  
  
 *// Start the threads* thread1.start();  
 thread2.start();  
  
 *// Wait for both threads to finish* try {  
 thread1.join();  
 thread2.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 System.*out*.println("Both threads have finished.");  
 }  
}

Output :

Thread 1: Holding resource 1...

Thread 2: Holding resource 2...

Thread 2: Waiting for resource 1...

Thread 1: Waiting for resource 2...

When you run this program, you may notice that it hangs and never reaches the "Both threads have finished." message, which is a typical symptom of a deadlock.

The problem of a deadlock can be fixed by ensuring that the threads always acquire resources in the same order. This approach prevents the circular wait condition, which is one of the necessary conditions for a deadlock to occur.

public class DeadlockFixedExample {  
  
 public static void main(String[] args) {  
 *// Two resources* Object resource1 = new Object();  
 Object resource2 = new Object();  
  
 *// Thread 1* Thread thread1 = new Thread(() -> {  
 synchronized (resource1) {  
 System.*out*.println("Thread 1: Holding resource 1...");  
  
 try {  
 Thread.*sleep*(100); *// Simulate some work* } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 System.*out*.println("Thread 1: Waiting for resource 2...");  
 synchronized (resource2) {  
 System.*out*.println("Thread 1: Got resource 2!");  
 }  
 }  
 });  
  
 *// Thread 2* Thread thread2 = new Thread(() -> {  
 synchronized (resource1) { *// Acquire resource1 first* System.*out*.println("Thread 2: Holding resource 1...");  
  
 try {  
 Thread.*sleep*(100); *// Simulate some work* } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 System.*out*.println("Thread 2: Waiting for resource 2...");  
 synchronized (resource2) {  
 System.*out*.println("Thread 2: Got resource 2!");  
 }  
 }  
 });  
  
 *// Start the threads* thread1.start();  
 thread2.start();  
  
 *// Wait for both threads to finish* try {  
 thread1.join();  
 thread2.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 System.*out*.println("Both threads have finished.");  
 }  
}

In this fixed code, both thread1 and thread2 acquire resource1 first and then resource2. This consistent order of acquisition eliminates the possibility of a circular wait and prevents the deadlock from occurring.