**Starvation, LiveLock and** **DeadLock in Java**

**Deadlock: "Me first, Me first"  
Livelock: " You first, You first"  
Starvation: "Some first, Others never"**

**DeadLock**

**DEADLOCK** Deadlock is a condition in which **a task waits indefinitely for conditions that can never be satisfied** - task holds resources while waiting for other resources to be released - tasks cannot be forced to relinquish resources **- a circular waiting condition** exists.

**public class** TestDeadLock1 {  
  
 **private** Object **lock1** = **new** Object();  
 **private** Object **lock2** = **new** Object();

private void sleep(long sleepTime) {  
 try {  
 TimeUnit.*SECONDS*.sleep(sleepTime);  
 } catch (InterruptedException ie) {  
 }  
}

**public void** foo() {  
 **synchronized** (**lock1**) {  
 **sleep(3);** 🡸 Without using sleep, it will hang

**synchronized** (**lock2**) {  
 System.***out***.println(**"Inside foo method ..."**);  
 }  
 }  
 }  
  
 **public void** bar() {  
 **synchronized** (**lock2**) {  
 **sleep(2);** 🡸 Without using sleep, it will hang  
 **synchronized** (**lock1**) {  
 System.***out***.println(**"Inside bar method ..."**);  
 }  
 }  
 }  
  
 **public void** test() {  
 Runnable r1 = () -> foo();  
 Runnable r2 = () -> bar();  
  
 **new** Thread(r1).start();  
 **new** Thread(r2).start();  
 }  
  
 **public static void** main(String[] args) {  
 **new** TestDeadLock1().test(so);  
 }  
}

**Using ReentrantLock**

**public class** TestDeadLock1 {  
  
 **private** Lock **lock1** = **new** ReentrantLock(**true**);  
 **private** Lock **lock2** = **new** ReentrantLock(**true**);

private void sleep(long sleepTime) {  
 try {  
 TimeUnit.*SECONDS*.sleep(sleepTime);  
 } catch (InterruptedException ie) {}  
   
 }  
  
 **public void** foo() {  
 **lock1**.lock();  
 **sleep(2);**

**lock2**.lock();  
 System.***out***.println(**"Inside foo method ..."**);  
  
 **lock2**.unlock();  
 **lock1**.unlock();  
 }  
  
 **public void** bar() {  
 **lock2**.lock();  
 **sleep(3);**  
 **lock1**.lock();  
 System.***out***.println(**"Inside bar method ..."**);  
 **lock1**.unlock();  
 **lock2**.unlock();  
 }  
  
 **public void** test() {  
 Runnable r1 = () -> foo();  
 Runnable r2 = () -> bar();  
  
 **new** Thread(r1).start();  
 **new** Thread(r2).start();  
 }  
  
 **public static void** main(String[] args) {  
 **new** TestDeadLock1().test();  
 }  
}

**Livelock**

Livelock is another concurrency problem and is similar to deadlock. In livelock, **two or more threads keep on transferring states betwe en one another** instead of waiting infinitely. **One example will be both wife and husband want eat soup using one spoon, they say, you first and you first**. **Pehele Aapp and Pehele Aapp. *Livelock describes situation where two threads are busy responding to each other’s action*. Livelock is a situation where two threads are blocked while responding to each other’s actions.**

**A Livelock Example:**

Let’s see an example: a criminal kidnaps a hostage and he asks for ransom in order to release the hostage. A police agrees to give the criminal the money he wants once the hostage is released. The criminal releases the hostage only when he gets the money. Both are waiting for each other to act first, hence livelock.

## **Criminal.java**

public class Criminal {  
 private boolean hostageReleased = false;  
  
 private void sleep(long time) {  
 try {  
 TimeUnit.*SECONDS*.sleep(time);  
 } catch (InterruptedException ex) {  
 ex.printStackTrace();  
 }  
 }  
  
 public void toReleaseHostage(Police police) {  
 while (!police.isMoneySent()) {  
 System.*out*.println("Criminal: waiting police to give ransom");  
*// sleep(1);* }  
 System.*out*.println("Criminal: released hostage");  
 this.hostageReleased = true;  
 }  
  
 public boolean isHostageReleased() {  
 return this.hostageReleased;  
 }  
}

## **Police.java**

public class Police {  
 private boolean moneySent = false;  
  
 public void toPayRansom(Criminal criminal) {  
 while (!criminal.isHostageReleased()) {  
 System.*out*.println("Police: waiting criminal to release hostage");  
*// sleep(1);* }  
 System.*out*.println("Police: sent money");  
 this.moneySent = true;  
 }  
  
 public boolean isMoneySent() {  
 return this.moneySent;  
 }  
}

## **Test.java**

**public class** Test {  
  
 **public static void** main(String[] args) {  
 Police police = **new** Police();  
 Criminal criminal = **new** Criminal();  
  
 Runnable policeRun = () -> police.toPayRansom(criminal);  
 Runnable criminalRun = () -> criminal.toReleaseHostage(police);

**new** Thread(policeRun).start();  
 **new** Thread(criminalRun).start();  
 }  
}

**Starvation**

***Starvation* describes a situation where a thread is unable to gain regular access to shared resources** and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads. For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked. ***Starvation describes a situation where a greedy thread holds a resource for a long time so other threads are blocked forever***.

**public class** TestStarvation {  
  
 **public synchronized void** m1() {  
 System.***out***.println(**"Executing ..."**);  
 **while**(**true**) {  
 // Do something  
 }  
 }  
  
 **public void** check() {  
 **for**(**int** i = 0; i < 3; i++) {  
 Runnable runnable = () -> m1();  
 **new** Thread(runnable).start();  
 }  
 }  
  
 **public static void** main(String[] args) {  
 **new** TestStarvation().check();  
 }  
}

A solution to solve this starvation problem is to make the current thread waits for a specified amount of time so other threads have chance to acquire the lock on the Worker object:

while (true) {

    System.out.println(name + " is working");

    try {

**wait(1000);**

    } catch (InterruptedException ex) { ex.printStackTrace(); }

}