**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. **DEADLOCK** is a condition where two or more threads are blocked forever.

**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 between one another** 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:**

1. A criminal wants money from police to release hostage and police want criminal to release hostage.
2. Two friends try to call each other over phone.
3. Two cars want to pass the bridge. Car1 allows first Car2 and Car2 wants to allow Car1 first.

**public class** Person\_1 {  
  
 **private boolean** isBusy;  
  
 **public boolean** isBusy() {  
 **return** isBusy;  
 }  
  
 **public void** sleep(**long** time) {  
 **try** {  
 TimeUnit.***SECONDS***.sleep(time);  
 } **catch** (InterruptedException e) {  
 **throw new** RuntimeException(e);  
 }  
 }  
  
 **public void** makeCall(Person\_2 person\_2) {  
 **while** (!person\_2.isBusy()) {  
 System.***out***.println("Person 1 is calling Person 2 ...");  
 sleep(1);  
 }  
 // person 1 is free now  
 isBusy = **true**; // 🡺you can make false also.  
 }  
}

**public class** Person\_2 {  
 **private boolean** isBusy;  
  
 **public boolean** isBusy() {  
 **return** isBusy;  
 }  
  
 **public void** sleep(**long** time) {  
 **try** {  
 TimeUnit.***SECONDS***.sleep(time);  
 } **catch** (InterruptedException e) {  
 **throw new** RuntimeException(e);  
 }  
 }  
  
 **public void** makeCall(Person\_1 person\_1) {  
 **while** (!person\_1.isBusy()) {  
 System.***out***.println("Person 2 is calling Person 1 ...");  
 sleep(1);  
 }  
 // person 2 is free now  
 isBusy = **true**; // 🡺you can make false also.  
 }  
}

**public static void** main(String[] args) {  
 Person\_2 person2 = **new** Person\_2();  
 Person\_1 person1 = **new** Person\_1();  
 Runnable r1 = () -> person1.makeCall(person2);  
 Runnable r2 = () -> person2.makeCall(person1);  
 Thread t1 = **new** Thread(r1);  
 Thread t2 = **new** Thread(r2);  
 t1.start();  
 t2.start();  
  
}

Better way to write the above to make more meaningful.

public class Person\_1 {  
  
 private boolean free = true;  
  
 public boolean isFree() {  
 return free;  
 }  
  
 public void sleep(long time) {  
 try {  
 TimeUnit.*SECONDS*.sleep(time);  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 }  
  
 public void makeCall(Person\_2 person\_2) {  
 while (person\_2.isFree()) {  
 System.*out*.println("Person 1 is calling Person 2 ...");  
 sleep(1);  
 }  
 free = false; *// ⎝ Since he is making call* }  
}

public class Person\_2 {  
 private boolean free = true;  
  
 public boolean isFree() {  
 return free;  
 }  
  
 public void sleep(long time) {  
 try {  
 TimeUnit.*SECONDS*.sleep(time);  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 }  
  
 public void makeCall(Person\_1 person\_1) {  
 while (person\_1.isFree()) {  
 System.*out*.println("Person 2 is calling Person 1 ...");  
 sleep(1);  
 }  
 free = false; *// ⎝ Since he is making call* }  
}

**public static void** main(String[] args) {  
 Person\_2 person2 = **new** Person\_2();  
 Person\_1 person1 = **new** Person\_1();  
 Runnable r1 = () -> person1.makeCall(person2);  
 Runnable r2 = () -> person2.makeCall(person1);  
 Thread t1 = **new** Thread(r1);  
 Thread t2 = **new** Thread(r2);  
 t1.start();  
 t2.start();  
  
}

Another Example

**public class** Car1 {  
 **private boolean** flag = **true**;  
  
 **public void** passBridge(Car2 car2) {  
 **while** (car2.hasPassedBridge()) {  
 System.***out***.println("Car1 waiting to pass the bridge");  
 **try** {  
 Thread.*sleep*(1000);  
 } **catch** (InterruptedException ex) {  
 ex.printStackTrace();  
 }  
 }  
 System.***out***.println("Passed bridge");  
 **this**.flag = **false**;  
 }  
  
 **public boolean** hasPassedBridge() {  
 **return this**.flag;  
 }  
}

**public class** Car2 {  
 **private boolean** flag = **true**;  
  
 **public void** passBridge(Car1 car1) {  
 **while** (car1.hasPassedBridge()) {  
 System.***out***.println("Car 2 is waiting to pass the bridge!");  
 **try** {  
 Thread.*sleep*(1000);  
 } **catch** (InterruptedException ex) {  
 ex.printStackTrace();  
 }  
 }  
 System.***out***.println("Car 2 has passed the bridge!");  
 **this**.flag = **false**;  
 }  
  
 **public boolean** hasPassedBridge() {  
 **return this**.flag;  
 }  
}

**public class** BridgeCheck {  
 **static final** Car2 ***car2*** = **new** Car2();  
 **static final** Car1 ***car1*** = **new** Car1();  
 **public static void** main(String[] args) {  
 Thread t1 = **new** Thread(**new** Runnable() {  
 **public void** run() {  
 ***car2***.passBridge(***car1***);  
 }  
 });  
 t1.start();  
   
 Thread t2 = **new** Thread(**new** Runnable() {  
 **public void** run() {  
 ***car1***.passBridge(***car2***);  
 }  
 });  
 t2.start();  
 }  
}

As per the above two examples, you can modify Criminal and Police livelock example.

**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(); }

}