CSCD467 – Lab3
Muangkong Yang

1) The provided solution does not work because the thread that accesses the lock cannot be controlled. If thread1 and thread2 alternate in getting the lock, then it will work, but the chance of them alternating accessing the lock is near impossible.

These statements cause the program to hang up:

```
1. synchronized (lock)
2. while( ! isT1turn );
3. while (isT1turn);
```

The program will hang up when a thread, let's say T1, runs and regrabs the lock before the other thread, T2, gets the lock. The boolean check in the while loop will always be true in T1 and thus, never exits the while loop. While T1 is stuck in the while loop, thread T2 be stuck at the synchronized lock line because it does not have the lock of the object. This can happen with vice versa threads.

2) I fixed this problem by creating a shared class (ThreadTurn) to check which thread should be running. The class will incorporate synchronized methods in place of creating an Object lock in the main class. This will use a conditional variable to give information on which thread should run.

```
public class ThreadTurn
{
    private static boolean IsThread1Turn = true;

    public static synchronized void setThread1Turn(boolean value)
    {
        IsThread1Turn = value;
    }

    public static synchronized boolean getThread1Turn()
    {
        return IsThread1Turn;
    }
}
```

The next code I fixed is taking away the while loop and replace it with just an if statement condition. If it's the thread's turn, enter the critical section and print, then set the ThreadTurn condition variable to false in the case of thread 1, (true for thread 2). I also place the index increment after so index i doesn't get incremented if the if loop doesn't get entered. This is the same for thread 2 as well.

Thread1

```
if(ThreadTurn.getThread1Turn())
{
    System.out.println("Message " + i + " from Thread T1.");
    ThreadTurn.setThread1Turn(false);
    i += 2;
    try{Thread.sleep(10);}
    catch(InterruptedException e){}
}
```

CSCD467 – Lab3
Muangkong Yang

Thread2

```
if(!ThreadTurn.getThread1Turn())
{
    System.out.println("Message " + i + " from Thread T2.");
    ThreadTurn.setThread1Turn(true);
    i += 2;
    try{Thread.sleep(10);}
    catch(InterruptedException e){}
}
```

3) This fixes the problem because now because the thread class provides correct mutual exclusion and now the for loops work correctly when it's the thread's turn to enter its' critical section even when it grabs the lock again before the other thread.