1. In this problem you are to modify the concurrent Java program on the next page to ensure that its two threads always take turns in a hypothetical game they are playing.

- The two threads playing the game are player1 (a Player1 object) and player2 (a Player2 object).
- The game consists of turns in which each player makes a move (the exact details of the game are not explained). The two players are supposed to alternate moves in the game, and the game should continue forever.
- Each player thread determines its game move based on the other player’s previous move.
- player1 should always make the first move in the game.
- A move in the game is represented using an integer.
- The two threads call methods named decide_next_move to determine their moves. After a thread determines its next move it communicates it to the other thread by calling set_move to store it in a shared Move object in the main function of the Game class. The other thread can examine that value using get_move.
- The function process_move is called by each thread to process the other thread’s most recent move.

Note that the code for the classes Player1 and Player2 are almost identical except the statements in their run methods are in different orders. This is to cause player1 to make the first move in the game, and player2 to initially wait for player1’s first move.

As the program is currently written the two threads do not use any synchronization, so incorrect or inconsistent results may be produced. Add appropriate Java synchronization constructs to the program so it always works properly. The conditions which must be ensured are:

- Every consecutive move value should be stored and printed (as in the output fragment below).
- The threads must strictly alternate their moves.
- Only one thread may be making a move at a time.
- The two threads cannot concurrently modify the value in the shared variable move.
- Neither thread should store a value in move until the other thread has gotten the value which was there and has placed a new value there.
- Neither thread should use the value in move until the other thread has stored a move in it.
As the program is written now, these conditions may not always hold. In particular, nothing forces the threads to strictly alternate their moves, and the other conditions may also be violated.

The rules of the game above must not be modified, so (beyond adding what’s necessary to ensure these conditions) you should not change player1’s or player2’s operations.

Other than that, you may add anything necessary to the program to guarantee these conditions always hold, except that busy-waiting may not be used.

Note that when the program is corrected, its output will always be the same any time it is run, beginning as shown to the right.

Player 1 begins game turn
Player 1 decides move 1
Player 2 processes move 1

Player 2 begins game turn
Player 2 decides move 2
Player 1 processes move 2

Player 1 begins game turn
Player 1 decides move 3
Player 2 processes move 3

Player 2 begins game turn
Player 2 decides move 4
Player 1 processes move 4
import java.io.*;

public class Game {
    public static void main(String args[]) {
        Move move = new Move();
        Player1 player1 = new Player1(move);
        Player2 player2 = new Player2(move);

        player1.start();
        player2.start();
    }
}

class Player1 extends Thread {
    Move move;
    int value;

    Player1(Move m) {
        move = m;
        value = 0;
    }

    int decide_next_move(int prev_move) {
        return prev_move + 1;
    }

    void process_move(int m) {
        // perhaps some computation occurs
        // here which isn't shown
        System.out.println("Player 1 processes move "+m);
    }

    public void run() {
        while (true) {

            System.out.println("\nPlayer 1 begins game turn");

            // compute next move
            value = decide_next_move(value);
            System.out.println("Player 1 decides move "+value);

            // communicate next move to Player2
            move.set_move(value);

            // get Player2's move and process it
            value = move.get_move();
            process_move(value);
        } // while
    }
} // end Player1 class
class Move {
    int move_value;
    void set_move(int new_move_value) {
        move_value = new_move_value;
    }
    int get_move() {
        return move_value;
    }
} // end Move class

class Player2 extends Thread {
    Move move;
    int value;
    Player2(Move m) {
        move = m;
    }
    int decide_next_move(int prev_move) {
        return prev_move + 1;
    }
    void process_move(int m) {
        // perhaps some computation occurs
        System.out.println("Player 2 processes move " + m);
    }
    public void run() {
        while (true) {
            value = move.get_move();
            process_move(value);
            System.out.println("Player 2 begins game turn");
            // compute next move
            value = decide_next_move(value);
            System.out.println("Player 2 decides move " + value);
            // communicate next move to Player1
            move.set_move(value);
        } // while
    } // end Player2 class
2. Consider the Java program below, which does nothing useful but does use concurrency.

```java
import java.io.*;

class ClockRadio {
    synchronized void snooze() {
        try {
            wait();
        } catch (Exception e) {
            // do nothing
        }
    }

    synchronized void alarm() {
        notify();
    }

    synchronized void big_alarm() {
        notifyAll();
    }
}

public class Main extends Thread {
    String name;
    ClockRadio cr;

    Main(String s, ClockRadio c) {
        name= s;
        cr= c;
    }

    public void run() {
        cr.snooze();
        System.out.print(name + " ");
    }

    public static void main(String args[]) {
        ClockRadio cr1= new ClockRadio();
        ClockRadio cr2= new ClockRadio();
        ClockRadio cr3= new ClockRadio();

        Main a= new Main("a", cr1);
        Main b= new Main("b", cr1);
        Main c= new Main("c", cr2);
        Main d= new Main("d", cr2);
        Main e= new Main("e", cr3);
        Main f= new Main("f", cr3);
        Main g= new Main("g", cr3);

        // HERE IS THE COMMENT
    }
```
There is a comment at the end of the main function of the Main class. The following questions ask for all of the possible outputs which could be produced, during different executions of the program, if groups of statements were inserted where the comment appears (and the main function of the Main class was run). Note that all output produced during any execution would appear on a single line, since the only output statement used is a System.out.print().

Any comment reading “do something” indicates some arbitrary computation which isn’t shown, but this arbitrary computation doesn’t use any of Java’s synchronization constructs.

Each part is independent and doesn’t rely on the statements or results of earlier parts. If in any part below the program could produce no output when it’s run, be sure to put “none” as one of your answers. If the same output can be produced in more than one way, just write it once. If in any part the program deadlocks or doesn’t terminate, just give the output (if any) which it would produce before deadlock occurs.

There are probably more answer blanks in each part than are needed.

It may be helpful to draw a diagram of the objects used in the program.

(a) Give all of the outputs which it would be possible to produce different times the program was run, if the following statements were inserted where the comment appears at the end of the main function of the Main class:
a.start();  
// do something
b.start();  
// do something
cr1.alarm();

(b) Give all the outputs which could be produced during different runs if the following statements were inserted where the comment appears:
c.start();  
// do something
d.start();  
// do something
e.start();

(c) Give all the outputs which could be produced during different runs if the following statements were inserted where the comment appears:
a.start();
// do something
cr2.alarm();

d) Give all the outputs which could be produced during different runs if the following statements were inserted where the comment appears:
e.start();  
// do something
f.start();  
// do something
g.start();  
// do something
cr3.alarm();  
// do something
cr3.snooze();  
// do something
cr3.alarm();

e) Give all the outputs which could be produced during different runs if the following statements were inserted where the comment appears:
a.start();  
// do something
cr2.alarm();
3. Consider the concurrent Java program shown below.

```java
import java.io.*;

class ThreadOne extends Thread {
    SharedObject shared_object;

    ThreadOne(SharedObject s) {
        shared_object= s;
    }

    public void run() {
        shared_object.f();
    }
}

class ThreadTwo extends Thread {
    SharedObject shared_object;

    ThreadTwo(SharedObject s) {
        shared_object= s;
    }

    public void run() {
        shared_object.g();
    }
}

class SharedObject {
    int a= 1, b= 2;

    // f() and g() will appear here

    synchronized void print() {
        System.out.print("" + a + " ">
            + b + " ");
    }
}

class Main {

    public static void main(String
        args[]) {

        SharedObject so= new SharedObject();
        ThreadOne t1= new ThreadOne(so);
        ThreadTwo t2= new ThreadTwo(so);

        t1.start();
        t2.start();

        so.print();
    }
}
```
The class **Main** declares one **SharedObject** object and two threads, one of type **ThreadOne** and one of type **ThreadTwo**. The **ThreadOne** object’s **run()** method calls the **SharedObject**’s method **f()**, and the **ThreadTwo** object’s **run()** method calls the **SharedObject**’s method **g()**.

The **SharedObject** class is partly given, but its methods **f()** and **g()** are not shown. The output which the program could produce depends on these methods. Each part below has different versions of these methods (and one part has some additional declarations). For each part, give all the possible outputs which the program could produce during different executions if the versions of the methods in that part (and any declarations) were included in the **SharedObject** class. If any output can be produced in more than one way, write it only **once**. If the program might not produce any output during an execution, write “none” as one of the answers. Note that a call to **SharedObject.print()** prints the values of both **SharedObject** fields **a** and **b**.

Any output which would be produced during one execution would appear on the same output line. Write in each answer blank the entire output produced from one execution. Don’t miss the call to **so.print()** at the end of the **Main** class’ **main()** method.

(a) Give all the outputs which could be produced using these methods in the **SharedObject** class. Each blank below is for all the output for a different execution.

```java
void f() {
    a = b;
}

void g() {
    b = a;
}
```
(b) Give all the outputs which could be produced using these methods in the `SharedObject` class. Each blank below is for all the output for a different execution.

```java
synchronized void f() {
    a = b;
}

synchronized void g() {
    b = a;
}
```

(c) Give all the outputs which could be produced using these methods in the `SharedObject` class. Each blank below is for all the output for a different execution.

```java
synchronized void f() {
    a = 3;
    b = 4;
}

synchronized void g() {
    print();
}
```

(d) Give all the outputs which could be produced using these methods in the `SharedObject` class. Each blank below is for all the output for a different execution.

```java
synchronized void f() {
    a = 3;
}

synchronized void g() {
    b = 4;
}
```

(e) Give all the outputs which could be produced using these methods in the `SharedObject` class. Each blank below is for all the output for a different execution.

```java
Object x = new Object();
Object y = new Object();

void f() {
    synchronized(x) {
        a = 3;
        b = 4;
    }
}

void g() {
    synchronized(y) {
        a = 5;
        b = 6;
    }
}
```

4. A **barrier** is used in a situation in a concurrent program where no thread should proceed past a certain point until all threads reach a that point in execution. All the threads should call the barrier at that point, and all should be delayed until the last thread reaches or calls the barrier, then all of them should be allowed to proceed.

Write a barrier–like Java class, named `Barrier`, that performs barrier synchronization across N Java threads. The constructor for a `Barrier` object takes N as an argument, and is called by only one thread (e.g., the main program thread). `Barrier` should have a synchronized method, called `bar`, that is called by a thread to enter the barrier. The exact syntax of the Java code is not important, but it should be close.