1. Different solutions are possible and the requirements can be enforced in different ways. The simplest way is probably just to modify the existing methods in the Move class. The following conditions need to be ensured. These are all present in the list of conditions appearing in the middle of the problem, just worded in different forms (the wording in the problem describes what the solution needs to ensure, but doesn’t explain exactly where in the code something could be added to ensure it).

c1: When one player tries to store a value in move they have to wait if the other player hasn’t yet examined the last game move they placed there. In the solution below this is accomplished by the wait() in get_move.

c2: When one player tries to examine the value in move they have to wait if the other player hasn’t yet stored their most recent game move there. In the solution below this is accomplished by the wait() in set_move.

c3: When one player stores a value in move then if the other player is already waiting for their game move they have to be allowed to proceed and examine that value. In the solution below this is accomplished by the notify() in set_move.

c4: When one player examines the value in move then if the other player is already waiting to store their next game move they have to be allowed to proceed and store their value. In the solution below this is accomplished by the notify() in set_move.

c5: If Java synchronization is properly used, then the methods or blocks or objects where threads may wait have to be synchronized with the synchronized keyword.

Here’s the modified version of just the Move class:

```java
class Move {

    int move_value;

    synchronized void set_move(int new_move_value) {
        move_value = new_move_value;
        notify();
        try {
            wait();
        } catch (InterruptedException ie) {
            // we won’t do anything here
        }
    }

    synchronized int get_move() {
        notify();
        try {
            wait();
        } catch (InterruptedException ie) {
            // we won’t do anything here
        }
        return move_value;
    }

} // end Move class
```

Note that two calls to notify() and two calls to wait() are both required. If fewer calls are used (for example, only one call to notify() and one call to wait()), it must be possible for one player to either make more than one move without the other player making a move, or for one player to process another player’s move more than once.
2. (a) • output: a
• output: b
• output: none

(b) • output: c d
• output: d c
• output: c
• output: d
• output: none

(c) • output: e f
• output: e g
• output: f e
• output: g e
• output: e
• output: f
• output: g
• output: none

(d) • output: e
• output: f
• output: g
• output: none

(e) • output: none

3. (a) #1: 1 1
#2: 1 2
#3: 2 1
#4: 2 2

(b) #1: 1 1
#2: 1 2
#3: 3 4
#4: 2 2

(c) #1: 1 2 1 2
#2: 1 2 3 4
#3: 3 4 3 4

(d) #1: 1 2
#2: 1 2
#3: 3 2
#4: 3 4

(e) #1: 1 2
#2: 1 4
#3: 1 6
#4: 3 2
#5: 3 4

4. public class Barrier {
    int num_threads, cur_threads;

    public Barrier(int n) {
        num_threads= n;
        cur_threads= 0;
    }

    public synchronized void bar() {
        cur_threads++;
        if (cur_threads >= num_threads)
            notifyAll();
        else
            while (cur_threads < num_threads)
                try {
                    wait();
                } catch (InterruptedException e) {
                    // do nothing here; watch TV perhaps
                }
    }
}