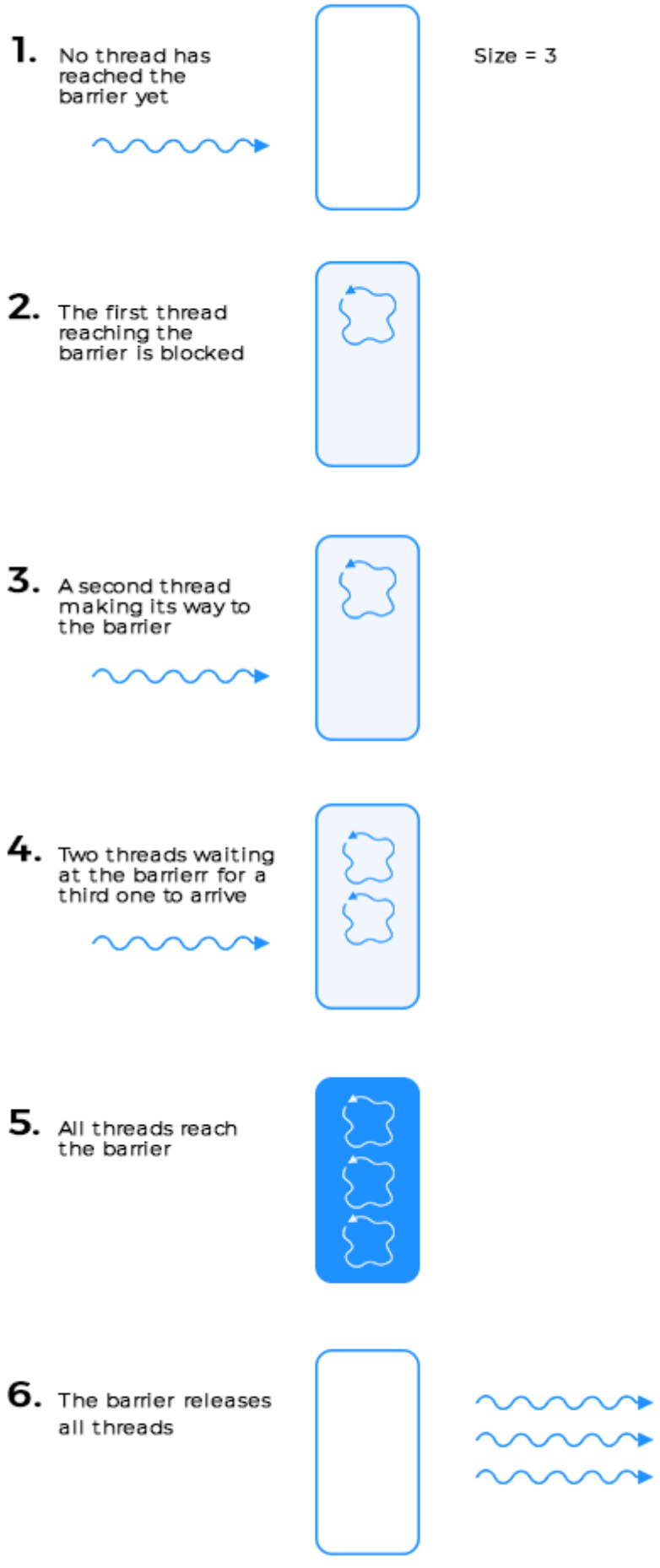
Implementing a Barrier

This lesson discusses how a barrier can be implemented in Java.

Problem

A barrier can be thought of as a point in the program code, which all or some of the threads need to reach at before any one of them is allowed to proceed further.

Working of a Barrier



Solution

A barrier allows multiple threads to congregate at a point in code before any one of the threads is allowed to move forward. Java and most other languages provide libraries which make barrier construct available for developer use. Even though we are re-inventing the wheel but this makes for a good interview question.

We can immediately realize that our solution will need a count variable to track the number of threads that have arrived at the barrier. If we have ***n*** threads, then ***n-1*** threads must wait for the **nth** thread to arrive. This suggests we have the **n-1** threads execute the wait method and the **nth** thread wakes up all the asleep **n-1** threads.

Below is the code:

```
1. public class Barrier {
2.
3.  int count = 0;
4.  int totalThreads;
5.
6.  public Barrier(int totalThreads) {
7.      this.totalThreads = totalThreads;
8.  }
9.
10. public synchronized void await() throws InterruptedException {
11.     // increment the counter whenever a thread arrives at the
12.     // barrier.
13.     count++;
14.
15.     if (count == totalThreads) {
16.         // wake up all the threads.
17.         notifyAll();
18.         // remember to reset count so that barrier can be reused
19.         count = 0;
20.     } else {
21.         // wait if you aren't the nth thread
22.         wait();
23.     }
24. }
```

```
25.}
```

Notice how we are resetting the count to zero in **line 19**. This is done so that we are able to re-use the barrier.

Below is the working code, alongwith a test case. The test-case creates three threads and has them synchronize on a barrier three times. We introduce sleeps accordingly so that, thread 1 reaches the barrier first, then thread 2 and finally thread 3. None of the thread is able to move forward until all the threads reach the barrier. This is verified by the order in which each thread prints itself in the output.

First Cut

Here's the first stab at the problem:

```
class Demonstration {
    public static void main( String args[] ) throws Exception{
        Barrier.runTest();
    }
}

class Barrier {

    int count = 0;
    int totalThreads;

    public Barrier(int totalThreads) {
        this.totalThreads = totalThreads;
    }

    public synchronized void await() throws InterruptedException {
        count++;

        if (count == totalThreads) {
            notifyAll();
            count = 0;
        } else {
            wait();
        }
    }

    public static void runTest() throws InterruptedException {
        final Barrier barrier = new Barrier(3);
```

```
        Thread p1 = new Thread(new Runnable() {
            public void run() {
                try {
                    System.out.println("Thread 1");
                    barrier.await();
                    System.out.println("Thread 1");
                    barrier.await();
                    System.out.println("Thread 1");
                    barrier.await();
                } catch (InterruptedException ie) {
                }
            }
        });

        Thread p2 = new Thread(new Runnable() {
            public void run() {
                try {
                    Thread.sleep(500);
                    System.out.println("Thread 2");
                    barrier.await();
                    Thread.sleep(500);
                    System.out.println("Thread 2");
                    barrier.await();
                    Thread.sleep(500);
                    System.out.println("Thread 2");
                    barrier.await();
                } catch (InterruptedException ie) {
                }
            }
        });

        Thread p3 = new Thread(new Runnable() {
            public void run() {
                try {
                    Thread.sleep(1500);
                    System.out.println("Thread 3");
                    barrier.await();
                    Thread.sleep(1500);
                    System.out.println("Thread 3");
                    barrier.await();
                    Thread.sleep(1500);
                    System.out.println("Thread 3");
                    barrier.await();
                } catch (InterruptedException ie) {
                }
            }
        });

        p1.start();
        p2.start();
        p3.start();

        p1.join();
        p2.join();
        p3.join();
    }
}
```

When you run the above code, you'll see that the threads print themselves in order i.e. first thread 1 then thread 2 and finally thread 3 prints. Thread 1 after reaching the barrier waits for the other two threads to reach the barrier before moving forward.

The above code has a subtle but very crucial bug! Can you spot the bug and try to fix it before reading on?

Second Cut

The previous code would have been hunky dory if we were guaranteed that no spurious wake-ups could ever occur. The `wait()` method invocation without the while loop is an error. We discussed in previous sections that `wait()` should always be used with a while loop that checks for a condition and if found false should make the thread wait again.

The condition the while loop can check for is simply how many threads have incremented the `count` variable so far. A thread that wakes up spuriously should go back to sleep if the `count` is less than the total number of threads. We can check for this condition as follows:

```
            while (count < totalThreads)
                wait();
```

The while loop introduces another problem. When the last thread does a `notifyAll()` it also resets the `count` to 0, which means the threads that are legitimately woken up will always be stuck in the while loop because `count` is immediately set to zero. What we really want is not to reset the `count` variable to zero until all the threads escape the while condition when `count` becomes `totalThreads`. Below is the improved version:

```
1. public class Barrier {
2.   int released = 0;
3.   int count = 0;
4.   int totalThreads;
5.
```

```
6.  public Barrier(int totalThreads) {
7.      this.totalThreads = totalThreads;
8.  }
9.
10. public synchronized void await() throws InterruptedException {
11.     // increment the counter whenever a thread arrives at the
12.     // barrier.
13.     count++;
14.
15.     if (count == totalThreads) {
16.         // wake up all the threds.
17.         notifyAll();
18.         // remember to reset count so that barrier can be reused
19.         released = totalThreads;
20.     } else {
21.         // wait till all threads reach barrier
22.         while (count < totalThreads)
23.              wait();
24.     }
25.
26.     released--;
27.     if (released == 0) count = 0;
28. }
29.}
```

The above code introduces a new variable `released` that keeps tracks of how many threads exit the barrier and when the last thread exits the barrier it resets `count` to zero, so that the barrier object can be reused in the future.

There is still a bug in the above code! Can you guess what it is?

Final Cut

To understand why the above code is broken, consider three threads **t1, t2, and t3** trying to `await()` on a barrier object in an infinite loop. Note the following sequence of events

1. Threads t1 and t2 invoke `await()` and end up waiting at **line#23**. The count variable is set to 2 and any spurious wakeups will cause t1 and

t2 to go back to waiting.

2. Threads t3 comes along, executes the if block on **line#15** and finds `count == totalThreads`. Thread t3 doesn't wait, notifies threads t1 and t2 to wakeup and exits.

3. **If thread t3 attempts to invoke `await()` immediately after exiting it and is also granted the monitor before threads t1 or t2 get a chance to acquire the monitor then the count variable will be incremented to 4.**

4. With `count` equal to 4, t3 will not block at the barrier and exit which breaks the contract for the barrier.

5. The invocation order of the `await()` method was t1,t2,t3, and t3 again. The right behavior would have been to release t1,t2, or t3 in any order and then block t3 on its second invocation of the `await()` method.

6. Another flaw with the above code is, it can cause a deadlock. Suppose we wanted the three threads t1, t2, and t3 to congregate at a barrier twice. The first invocation was in the order [t1, t2, t3] and the second was in the order [t3, t2, t1]. If t3 immediately invoked await after the first barrier, it would go past the second barrier without stopping while t2 and t1 would become stranded at the second barrier, since `count` would never equal `totalThreads`.

The fix requires us to block any new threads from proceeding until all the threads that have reached the previous barrier are released. The code with the fix appears below:

```
1. public class Barrier {
2.  int released = 0;
3.  int count = 0;
4.  int totalThreads;
5.
6.  public Barrier(int totalThreads) {
7.      this.totalThreads = totalThreads;
8.  }
9.
10. public synchronized void await() throws InterruptedException {
```

```
11.
12.     // block any new threads from proceeding till,
13.     // all threads from previous barrier are released
14.     while (count == totalThreads) wait();
15.
16.     // increment the counter whenever a thread arrives at the
17.     // barrier.
18.     count++;
19.
20.     if (count == totalThreads) {
21.         // wake up all the threds.
22.         notifyAll();
23.         // remember to set released to totalThreads
24.         released = totalThreads;
25.     } else {
26.         // wait till all threads reach barrier
27.         while (count < totalThreads)
28.              wait();
29.     }
30.
31.     released--;
32.     if (released == 0) {
33.         count = 0;
34.         // remember to wakeup any threads
35.         // waiting on line#14
36.         notifyAll();
37.     }
38. }
39.}
```

```
class Demonstration {
    public static void main( String args[] ) throws Exception{
        Barrier.runTest();
    }
}

class Barrier {

    int count = 0;
    int released = 0;
    int totalThreads;

    public Barrier(int totalThreads) {
        this.totalThreads = totalThreads;
    }

    public static void runTest() throws InterruptedException {
        final Barrier barrier = new Barrier(3);
```

```
    Thread p1 = new Thread(new Runnable() {
        public void run() {

            try {
                System.out.println("Thread 1");
                barrier.await();
                System.out.println("Thread 1");
                barrier.await();
                System.out.println("Thread 1");
                barrier.await();
            } catch (InterruptedException ie) {
            }
        }
    });

    Thread p2 = new Thread(new Runnable() {
        public void run() {
            try {
                Thread.sleep(500);
                System.out.println("Thread 2");
                barrier.await();
                Thread.sleep(500);
                System.out.println("Thread 2");
                barrier.await();
                Thread.sleep(500);
                System.out.println("Thread 2");
                barrier.await();
            } catch (InterruptedException ie) {
            }
        }
    });

    Thread p3 = new Thread(new Runnable() {
        public void run() {
            try {
                Thread.sleep(1500);
                System.out.println("Thread 3");
                barrier.await();
                Thread.sleep(1500);
                System.out.println("Thread 3");
                barrier.await();
                Thread.sleep(1500);
                System.out.println("Thread 3");
                barrier.await();
            } catch (InterruptedException ie) {
            }
        }
    });

    p1.start();
    p2.start();
    p3.start();

    p1.join();
    p2.join();
    p3.join();
}


public synchronized void await() throws InterruptedException {

    while (count == totalThreads)
```

```
            wait();

        count++;

        if (count == totalThreads) {
            notifyAll();
            released = totalThreads;
        } else {

            while (count < totalThreads)
                wait();
        }

        released--;
        if (released == 0) {
            count = 0;
            // remember to wakeup any threads
            // waiting on line#81
            notifyAll();
        }
    }
}
```