23.7 (Advanced) Producer/Consumer Relationship with `synchronized`, `wait`, `notify` and `notifyAll`

[*Note:* This section is intended for *advanced* programmers who want to control synchronization.2] The previous example showed how multiple threads can share a single-element buffer in a thread-safe manner by using the `ArrayBlockingQueue` class that encapsulates the synchronization necessary to protect the shared mutable data. For educational purposes, we now explain how you can implement a shared buffer yourself using the `synchronized` keyword and methods of class `Object`. *Using an* `ArrayBlockingQueue` *generally results in more-maintainable, better-performing code.*

2. For detailed information on `wait`, `notify` and `notifyAll`, see Chapter 14 of *Java Concurrency in Practice* by Brian Goetz, et al., Addison-Wesley Professional, 2006.

After identifying the shared mutable data and the *synchronization policy* (i.e., associating the data with a lock that guards it), the next step in synchronizing access to the buffer is to implement methods `blockingGet` and `blockingPut` as `synchronized` methods. This requires

that a thread obtain the *monitor lock* on the `Buffer` object before attempting to access the buffer data, but it does not automatically ensure that threads proceed with an operation only if the buffer is in the proper state. We need a way to allow our threads to *wait*, depending on whether certain conditions are true. In the case of placing a new item in the buffer, the condition that allows the operation to proceed is that the *buffer is not full*. In the case of fetching an item from the buffer, the condition that allows the operation to proceed is that the *buffer is not empty*. If the condition in question is true, the operation may proceed; if it's false, the thread must *wait* until it becomes true. When a thread is waiting on a condition, it's removed from contention for the processor and placed into the *waiting* state and the lock it holds is released.

Methods `wait`, `notify` and `notifyAll`

`Object` methods `wait`, `notify` and `notifyAll` can be used with conditions to make threads *wait* when they cannot perform their tasks. If a thread obtains the *monitor lock* on an object, then determines that it cannot continue with its task on that object until some condition is satisfied, the thread can call `Object` method `wait` on the `synchronized` object; this *releases the monitor lock* on the object, and the thread waits in the *waiting* state while the other threads try to enter the object's `synchronized` statement(s) or method(s). When a thread executing a `synchronized` statement (or method)

completes or satisfies the condition on which another thread may be waiting, it can call `Object` method `notify` on the `synchronized` object to allow a waiting thread to transition to the *runnable* state again. At this point, the thread that was transitioned from the *waiting* state to the *runnable* state can attempt to *reacquire the monitor lock* on the object. Even if the thread is able to reacquire the monitor lock, it still might not be able to perform its task at this time—in which case the thread will reenter the *waiting* state and implicitly *release the monitor lock*. If a thread calls `notifyAll` on the `synchronized` object, then *all* the threads waiting for the monitor lock become eligible to *reacquire the lock* (that is, they all transition to the *runnable* state).

Remember that only *one* thread at a time can obtain the monitor lock on the object—other threads that attempt to acquire the same monitor lock will be *blocked* until the monitor lock becomes available again (i.e., until no other thread is executing in a `synchronized` statement on that object).

Common Programming Error 23.1

It's an error if a thread issues a `wait`*, a* `notify` *or a* `notifyAll` *on an object without having acquired a lock for it. This causes an* `IllegalMonitorStateException`.

Error-Prevention Tip 23.2

It's a good practice to use `notifyAll` *to notify* waiting *threads to become* runnable. *Doing so avoids the possibility that your program would forget about waiting threads, which would otherwise starve.*

Figures 23.16 and 23.17 demonstrate a `Producer` and a `Consumer` accessing a shared buffer with synchronization. In this case, the `Producer` always produces a value *first*, the `Consumer` correctly consumes only *after* the `Producer` produces a value and the `Producer` correctly produces the next value only after the `Consumer` consumes the previous (or first) value. We reuse interface `Buffer` and classes `Producer` and `Consumer` from the example in Section 23.5, except that line 24 is removed from class `Producer` and class `Consumer`.

Class `SynchronizedBuffer`

The synchronization is handled in class `SynchronizedBuffer`'s `blockingPut` and `blockingGet` methods (Fig. 23.16). Thus, the `Producer`'s and `Consumer`'s `run` methods simply call the shared object's `synchronized blockingPut` and `blockingGet`

methods. Again, we output messages from the `synchronized` methods for demonstration purposes only—I/O *should not* be performed in `synchronized` blocks, because it's important to minimize the amount of time that an object is "locked."

```
   1  // Fig. 23.16: SynchronizedBuffer.java
2  // Synchronizing access to shared mutable data u
   3  // methods wait and notifyAll.
4  public class SynchronizedBuffer implements Buffe
5     private int buffer = -1; // shared by produce
6     private boolean occupied = false;
                  7
8     // place value into buffer
9     @Override
10    public synchronized void blockingPut(int valu
11       throws InterruptedException {
12       // while there are no empty locations, pla
13       while (occupied) {
14          // output thread information and buffer
15          System.out.println("Producer tries to w
16          displayState("Buffer full. Producer wai
17          wait();
18       }
                  19
20       buffer = value; // set new buffer value
                  21
22       // indicate producer cannot store another
23       // until consumer retrieves current buffer
24       occupied = true;
                  25
26       displayState("Producer writes " + buffer);
                  27
28       notifyAll(); // tell waiting thread(s) to
29    } // end method blockingPut; releases lock on
                  30
31    // return value from buffer
32    @Override
```

```
33      public synchronized int blockingGet() throws
34         // while no data to read, place thread in
          35         while (!occupied) {
36            // output thread information and buffer
37            System.out.println("Consumer tries to r
38            displayState("Buffer empty. Consumer wa
               39            wait();
                    40         }
                         41
42         // indicate that producer can store anothe
43         // because consumer just retrieved buffer
           44         occupied = false;
                         45
46         displayState("Consumer reads " + buffer);
                         47
48         notifyAll(); // tell waiting thread(s) to
                         49
            50         return buffer;
51      } // end method blockingGet; releases lock on
                         52
53      // display current operation and buffer state
54      private synchronized void displayState(String
55         System.out.printf("%-40s%d\t\t%b%n%n", ope
                  56      }
                    57   }
```

Fig. 23.16

Synchronizing access to shared mutable data using `Object` methods `wait` and `notifyAll`.

Fields and Methods of Class

SynchronizedBuffer

Class SynchronizedBuffer contains fields buffer (line 5) and occupied (line 6)—you must synchronize access to *both* fields to ensure that class SynchronizedBuffer is thread safe. Methods blockingPut (lines 9–29) and blockingGet (lines 32–51) are declared as synchronized—only *one* thread can call either of these methods at a time on a particular SynchronizedBuffer object. Field occupied is used to determine whether it's the Producer's or the Consumer's turn to perform a task. This field is used in conditional expressions in both the blockingPut and blockingGet methods. If occupied is false, then buffer is empty, so the Consumer cannot read the value of buffer, but the Producer can place a value into buffer. If occupied is true, the Consumer can read a value from buffer, but the Producer cannot place a value into buffer.

Method blockingPut and the Producer Thread

When the Producer thread's run method invokes synchronized method blockingPut, the thread attempts to acquire the SynchronizedBuffer object's monitor lock. If the monitor lock is available, the Producer thread *implicitly* acquires the lock. Then the loop at lines 13–18 first determines whether occupied is true. If so,

`buffer` is *full* and we want to wait until the buffer is empty, so line 15 outputs a message indicating that the `Producer` thread is trying to write a value, and line 16 invokes method `displayState` (lines 54–56) to output another message indicating that `buffer` is *full* and that the `Producer` thread is *waiting* until there's space. Line 17 invokes method `wait` (inherited from `Object` by `SynchronizedBuffer`) to place the thread that called method `blockingPut` (i.e., the `Producer` thread) in the *waiting* state for the `SynchronizedBuffer` object. The call to `wait` causes the calling thread to *implicitly* release the lock on the `SynchronizedBuffer` object. This is important because the thread cannot currently perform its task and because other threads (in this case, the `Consumer`) should be allowed to access the object to allow the condition (`occupied`) to change. Now another thread can attempt to acquire the `SynchronizedBuffer` object's lock and invoke the object's `blockingPut` or `blockingGet` method.

The `Producer` thread remains in the *waiting* state until another thread *notifies* the `Producer` that it may proceed—at which point the `Producer` returns to the *runnable* state and attempts to implicitly reacquire the lock on the `SynchronizedBuffer` object. If the lock is available, the `Producer` thread reacquires it, and method `blockingPut` continues executing with the next statement after the `wait` call. Because `wait` is called in a loop, the loop-continuation condition is tested again to determine whether the thread can proceed. If not, then `wait` is invoked again—otherwise, method `blockingPut` continues with the next statement

after the loop.

Line 20 in method `blockingPut` assigns the `value` to the `buffer`. Line 24 sets `occupied` to `true` to indicate that the `buffer` now contains a value (i.e., a consumer can read the value, but a `Producer` cannot yet put another value there). Line 26 invokes method `displayState` to output a message indicating that the `Producer` is writing a new value into the `buffer`. Line 28 invokes method `notifyAll` (inherited from `Object`). If any threads are *waiting* on the `SynchronizedBuffer` object's monitor lock, those threads enter the *runnable* state and can now attempt to *reacquire the lock*. Method `notifyAll` returns immediately, and method `blockingPut` then returns to the caller (i.e., the `Producer`'s `run` method). When method `blockingPut` returns, it *implicitly releases the monitor lock* on the `SynchronizedBuffer` object.

Method `blockingGet` and the `Consumer` Thread

Methods `blockingGet` and `blockingPut` are implemented similarly. When the `Consumer` thread's `run` method invokes `synchronized` method `blockingGet`, the thread attempts to *acquire the monitor lock* on the `SynchronizedBuffer` object. If the lock is available, the `Consumer` thread acquires it. Then the `while` loop at lines 35–40 determines whether `occupied` is `false`. If so, the

buffer is empty, so line 37 outputs a message indicating that the `Consumer` thread is trying to read a value, and line 38 invokes method `displayState` to output a message indicating that the buffer is *empty* and that the `Consumer` thread is *waiting*. Line 39 invokes method `wait` to place the thread that called method `blockingGet` (i.e., the `Consumer`) in the *waiting* state for the `SynchronizedBuffer` object. Again, the call to `wait` causes the calling thread to *implicitly release the lock* on the `SynchronizedBuffer` object, so another thread can attempt to acquire the `SynchronizedBuffer` object's lock and invoke the object's `blockingPut` or `blockingGet` method. If the lock on the `SynchronizedBuffer` is not available (e.g., if the `Producer` has not yet returned from method `blockingPut`), the `Consumer` is *blocked* until the lock becomes available.

The `Consumer` thread remains in the *waiting* state until it's *notified* by another thread that it may proceed—at which point the `Consumer` thread returns to the *runnable* state and attempts to *implicitly reacquire the lock* on the `SynchronizedBuffer` object. If the lock is available, the `Consumer` reacquires it, and method `blockingGet` continues executing with the next statement after `wait`. Because `wait` is called in a loop, the loop-continuation condition is tested again to determine whether the thread can proceed with its execution. If not, `wait` is invoked again—otherwise, method `blockingGet` continues with the next statement after the loop. Line 44 sets `occupied` to `false` to indicate that `buffer` is now empty (i.e., a `Consumer` cannot

read the value, but a `Producer` can place another value in `buffer`), line 46 calls method `displayState` to indicate that the consumer is reading and line 48 invokes method `notifyAll`. If any threads are in the *waiting* state for the lock on this `SynchronizedBuffer` object, they enter the *runnable* state and can now attempt to *reacquire the lock*. Method `notifyAll` returns immediately, then method `blockingGet` returns the value of `buffer` to its caller. When method `blockingGet` returns (line 50), the lock on the `SynchronizedBuffer` object is *implicitly released*.

 Error-Prevention Tip 23.3

Always invoke method `wait` *in a loop that tests the condition the task is waiting on. It's possible that a thread will reenter the runnable state (via a timed wait or another thread calling* `notifyAll`*) before the condition is satisfied. Testing the condition again ensures that the thread will not erroneously execute if it was notified early.*

Method `displayState` Is Also `synchronized`

Notice that method `displayState` is a `synchronized`

method. This is important because it, too, reads the `SynchronizedBuffer`'s shared mutable data. Though only one thread at a time may acquire a given object's lock, one thread may acquire the same object's lock *multiple* times—this is known as a **reentrant lock** and enables one `synchronized` method to invoke another on the same object.

Testing Class `SynchronizedBuffer`

Class `SharedBufferTest2` (Fig. 23.17) is similar to class `SharedBufferTest` (Fig. 23.13). Line 10 creates an `ExecutorService` to run the `Producer` and `Consumer` tasks. Line 13 creates a `SynchronizedBuffer` object and assigns its reference to `Buffer` variable `sharedLocation`. This object stores the data that will be shared between the `Producer` and `Consumer`. Lines 15–16 display the column heads for the output. Lines 19–20 execute a `Producer` and a `Consumer`. Finally, line 22 calls method `shutdown` to end the application when the `Producer` and `Consumer` complete their tasks, and line 23 waits for the scheduled tasks to complete. When method `main` ends, the main thread of execution terminates.

```
1  // Fig. 23.17: SharedBufferTest2.java
2  // Two threads correctly manipulating a synchroni
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
```

```
5  import java.util.concurrent.TimeUnit;
6
7  public class SharedBufferTest2 {
8     public static void main(String[] args) throws
9        // create a newCachedThreadPool
10       ExecutorService executorService = Executor
11
12       // create SynchronizedBuffer to store ints
13       Buffer sharedLocation = new SynchronizedBu
14
15       System.out.printf("%-40s%s\t\t%s%n%-40s%s%
16          "Buffer", "Occupied", "---------", "---
17
18       // execute the Producer and Consumer tasks
19       executorService.execute(new Producer(share
20       executorService.execute(new Consumer(share
21
22       executorService.shutdown();
23       executorService.awaitTermination(1, TimeUn
24    }
25 }
```

Operation	Buffer	Occupied
---	---	---
Consumer tries to read.		
Buffer empty. Consumer waits.	-1	false
Producer writes 1	1	true
Consumer reads 1	1	false
Consumer tries to read.		
Buffer empty. Consumer waits.	1	false
Producer writes 2	2	true
Consumer reads 2	2	false

Producer writes 3	3	true
Consumer reads 3	3	false
Producer writes 4	4	true
Producer tries to write.		
Buffer empty. Consumer waits.	4	true
Consumer reads 4	4	false
Producer writes 5	5	true
Consumer reads 5	5	false
Producer writes 6	6	true

Producer tries to write.		
Buffer empty. Consumer waits.	6	true
Consumer reads 6	6	false
Producer writes 7	7	true
Producer tries to write.		
Buffer empty. Consumer waits.	7	true
Consumer reads 7	7	false
Producer writes 8	8	true
Consumer reads 8	8	false
Consumer tries to read.		
Buffer full. Producer waits.	8	false
Producer writes 9	9	true
Consumer reads 9	9	false

Consumer tries to read.		
Buffer empty. Consumer waits.	9	false
Producer writes 10	10	true
Consumer reads 10	10	false

```
Producer done producing
Terminating Producer

Consumer read values totaling 55
Terminating Consumer
```

Fig. 23.17

Two threads correctly manipulating a synchronized buffer.

Study the outputs in Fig. 23.17. Observe that *every integer produced is consumed exactly once—no values are lost, and no values are consumed more than once*. The synchronization ensures that the `Producer` produces a value only when the buffer is *empty* and the `Consumer` consumes only when the buffer is *full*. The `Producer` always goes first, the `Consumer` *waits* if the `Producer` has not produced since the `Consumer` last consumed, and the `Producer` waits if the `Consumer` has not yet consumed the value that the `Producer` most recently produced. Execute this program several times to confirm that every integer produced is consumed exactly *once*. In the sample output, note the highlighted lines indicating when the `Producer` and

`Consumer` must *wait* to perform their respective tasks.