23.6 Producer/Consumer Relationship: `ArrayBlockingQueue`

The best way to synchronize producer and consumer threads is to use classes from Java's `java.util.concurrent` package that *encapsulate the synchronization for you*. Java includes the class `ArrayBlockingQueue`—a fully implemented, *thread-safe buffer class* that implements interface `BlockingQueue`. This interface declares methods `put` and `take`. Method `put` places an element at the end of the `BlockingQueue`, waiting if the queue is full. Method `take` removes an element from the head of the `BlockingQueue`, waiting if the queue is empty. These methods make class `ArrayBlockingQueue` a good choice for implementing a shared buffer. Because method `put` blocks until there's room in the buffer to write data, and method `take` blocks until there's new data to read, the producer must produce a value first, the consumer correctly consumes only after the producer writes a value and the producer correctly produces the next value (after the first) only after the consumer reads the previous (or first) value. `ArrayBlockingQueue` stores the shared mutable data in an array, the size of which is specified as `ArrayBlockingQueue`'s constructor argument. An `ArrayBlockingQueue` is fixed in size and will not expand to accommodate extra elements.

Class `BlockingBuffer`

Figures 23.14–23.15 demonstrate a `Producer` and a `Consumer` accessing an `ArrayBlockingQueue`. Class `BlockingBuffer` (Fig. 23.14) uses an `ArrayBlockingQueue` object that stores an `Integer` (line 6). Line 9 creates the `ArrayBlockingQueue` and passes `1` to the constructor so that the object holds a single value to mimic the `UnsynchronizedBuffer` example in Fig. 23.12. We discuss *multiple-element buffers* in Section 23.8. Because our `BlockingBuffer` class uses the *thread-safe* `ArrayBlockingQueue` class to manage all of its shared state (the shared buffer in this case), `BlockingBuffer` is itself *thread safe*, even though we have not implemented the synchronization ourselves.

```
1  // Fig. 23.14: BlockingBuffer.java
2  // Creating a synchronized buffer using an Array
3  import java.util.concurrent.ArrayBlockingQueue;
4
5  public class BlockingBuffer implements Buffer {
6     private final ArrayBlockingQueue<Integer> buf
7
8     public BlockingBuffer() {
9        buffer = new ArrayBlockingQueue<Integer>(1
10    }
11
12    // place value into buffer
13    @Override
14    public void blockingPut(int value) throws Int
```

```
         buffer.put(value); // place value in buffe
         System.out.printf("%s%2d\t%s%d%n", "Produc
            "Buffer cells occupied: ", buffer.size(
      }

      // return value from buffer
      @Override
      public int blockingGet() throws InterruptedEx
         int readValue = buffer.take(); // remove v
         System.out.printf("%s %2d\t%s%d%n", "Consu
            readValue, "Buffer cells occupied: ", b

         return readValue;
      }
   }
```

Fig. 23.14

Creating a synchronized buffer using an `ArrayBlockingQueue`.

`BlockingBuffer` implements interface `Buffer` (Fig. 23.10 modified to remove line 24) and `Consumer` (Fig. 23.11 modified to remove line 24) from the example in Section 23.5. This approach demonstrates encapsulated synchronization—*the threads accessing the shared object are unaware that their buffer accesses are now synchronized*. The synchronization is handled entirely in the `blockingPut` and `blockingGet` methods of `BlockingBuffer` by calling the synchronized `ArrayBlockingQueue` methods `put` and `take`, respectively. Thus, the `Producer` and `Consumer`

Runnables are properly synchronized simply by calling the shared object’s blockingPut and blockingGet methods.

Line 15 in method blockingPut (Fig. 23.14) calls the ArrayBlockingQueue object’s put method. This method call blocks if necessary until there’s room in the buffer to place the value. Method blockingGet calls the ArrayBlockingQueue object’s take method (line 23). This method call *blocks* if necessary until there’s an element in the buffer to remove. Lines 16–17 and 24–25 use the ArrayBlockingQueue object’s size method to display the total number of elements currently in the ArrayBlockingQueue.

Class BlockingBufferTest

Class BlockingBufferTest (Fig. 23.15) contains the main method that launches the application. Line 11 creates an ExecutorService, and line 14 creates a BlockingBuffer object and assigns its reference to the Buffer variable sharedLocation. Lines 16–17 execute the Producer and Consumer Runnables. Line 19 calls method shutdown to end the application when the threads finish executing the Producer and Consumer tasks, and line 20 waits for the scheduled tasks to complete.

```
1   // Fig. 23.15: BlockingBufferTest.java
```

```java
// Two threads manipulating a blocking buffer tha
// implements the producer/consumer relationship.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class BlockingBufferTest {
    public static void main(String[] args) throws
        // create new thread pool
        ExecutorService executorService = Executor

        // create BlockingBuffer to store ints
        Buffer sharedLocation = new BlockingBuffer

        executorService.execute(new Producer(share
        executorService.execute(new Consumer(share

        executorService.shutdown();
        executorService.awaitTermination(1, TimeUn
    }
}
```

Producer writes	1	Buffer cells occupied: 1
Consumer reads	1	Buffer cells occupied: 0
Producer writes	2	Buffer cells occupied: 1
Consumer reads	2	Buffer cells occupied: 0
Producer writes	3	Buffer cells occupied: 1
Consumer reads	3	Buffer cells occupied: 0
Producer writes	4	Buffer cells occupied: 1
Consumer reads	4	Buffer cells occupied: 0
Producer writes	5	Buffer cells occupied: 1
Consumer reads	5	Buffer cells occupied: 0

Producer writes	6	Buffer cells occupied: 1
Consumer reads	6	Buffer cells occupied: 0
Producer writes	7	Buffer cells occupied: 1
Consumer reads	7	Buffer cells occupied: 0
Producer writes	8	Buffer cells occupied: 1
Consumer reads	8	Buffer cells occupied: 0
Producer writes	9	Buffer cells occupied: 1
Consumer reads	9	Buffer cells occupied: 0
Producer writes	10	Buffer cells occupied: 1

```
Producer done producing
Terminating Producer
```

Consumer reads 10	Buffer cells occupied: 0

```
Consumer read values totaling 55
Terminating Consumer
```

Fig. 23.15

Two threads manipulating a blocking buffer that properly implements the producer/consumer relationship.

While methods `put` and `take` of `ArrayBlockingQueue` are properly synchronized, `BlockingBuffer` methods `blockingPut` and `blockingGet` (Fig. 23.14) are not declared to be synchronized. Thus, the statements performed in method `blockingPut`—the `put` operation (Fig. 23.14, line 15) and the output (lines 16–17)—are *not atomic*; nor are the statements in method `blockingGet`—the `take` operation (line 23) and the output (lines 24–25). So there's no guarantee that each output will occur immediately after the corresponding `put` or `take` operation, and the outputs may appear out of order. Even if they do, the `ArrayBlockingQueue` object is properly synchronizing access to the data, as evidenced by the fact that the sum of values read by the consumer is always correct.