

23.9 (Advanced) Producer/Consumer Relationship: The Lock and Condition Interfaces

Though the `synchronized` keyword provides for most basic thread-synchronization needs, Java provides other tools to assist in developing concurrent programs. In this section, we discuss the `Lock` and `Condition` interfaces. These interfaces give you more precise control over thread synchronization, but are more complicated to use. *Only the most advanced programmers should use these interfaces.*

Interface `Lock` and Class `ReentrantLock`

Any object can contain a reference to an object that implements the `Lock` interface (of package `java.util.concurrent.locks`). A thread calls the `Lock`'s `lock` method (analogous to entering a `synchronized` block) to acquire the lock. Once a `Lock` has been obtained by one thread, the `Lock` object will not allow another thread to obtain the `Lock` until the first thread releases

the `Lock` (by calling the `Lock`'s `unlock` method— analogous to exiting a `synchronized` block). If several threads are trying to call method `lock` on the same `Lock` object at the same time, only one of these threads can obtain the lock—all the others are placed in the *waiting* state for that lock. When a thread calls method `unlock`, the lock on the object is released and a waiting thread attempting to lock the object proceeds.



Error-Prevention Tip

23.4

Place calls to `Lock` method `unlock` in a `finally` block. If an exception is thrown, `unlock` must still be called or deadlock could occur.

Class `ReentrantLock` (of package `java.util.concurrent.locks`) is a basic implementation of the `Lock` interface. The constructor for a `ReentrantLock` takes a `boolean` argument that specifies whether the lock has a **fairness policy**. If the argument is `true`, the `ReentrantLock`'s fairness policy is “the longest-waiting thread will acquire the lock when it’s available.” Such a fairness policy guarantees that *indefinite postponement* (also called *starvation*) cannot occur. If the fairness policy argument is set to `false`, there’s no guarantee as to which waiting thread will acquire the lock when it’s available.



Software Engineering Observation 23.6

Using a `ReentrantLock` with a fairness policy avoids indefinite postponement.



Performance Tip 23.5

In most cases, a non-fair lock is preferable, because using a fair lock can decrease program performance.

Condition Objects and Interface `Condition`

If a thread that owns a `LOCK` determines that it cannot continue with its task until some condition is satisfied, the thread can wait on a **condition object**. Using `LOCK` objects allows you to explicitly declare the condition objects on which a thread may need to wait. For example, in the producer/consumer relationship, producers can wait on *one* object and consumers can wait on *another*. This is not possible when using the `synchronized` keywords and an object's built-in monitor lock.

Condition objects are associated with a specific `LOCK` and are

created by calling a `Lock`'s `newCondition` method, which returns an object that implements the `Condition` interface (package `java.util.concurrent.locks`). To wait on a `Condition`, the thread can call its `await` method (analogous to `Object` method `wait`). This immediately releases the associated `LOCK` and places the thread in the *waiting* state for that `Condition`. Other threads can then try to obtain the `LOCK`.

When a *runnable* thread completes a task and determines that the *waiting* thread can now continue, the *runnable* thread can call `Condition` method `signal` (analogous to `Object` method `notify`) to allow a thread in that `Condition`'s *waiting* state to return to the *runnable* state. At this point, the thread that transitioned from the *waiting* state to the *runnable* state can attempt to reacquire the `LOCK`. Even if it's able to *reacquire* the `LOCK`, the thread still might not be able to perform its task at this time—in which case the thread can call the `Condition`'s `await` method to *release* the `LOCK` and reenter the *waiting* state.

If multiple threads are in a `Condition`'s *waiting* state when `signal` is called, the default implementation of `Condition` signals the longest-waiting thread to transition to the *runnable* state. If a thread calls `Condition` method `signalAll` (analogous to `Object` method `notifyAll`), then all the threads waiting for that condition transition to the *runnable* state and become eligible to reacquire the `LOCK`. Only one of those threads can obtain the `LOCK` on the object—the others will wait until the `LOCK` becomes available again. If the `LOCK`

has a *fairness policy*, the longest-waiting thread acquires the `Lock`. When a thread is finished with a shared object, it must call method `unlock` to release the `Lock`.



Error-Prevention Tip

23.5

When multiple threads manipulate a shared object using locks, ensure that if one thread calls method `await` to enter the waiting state for a condition object, a separate thread eventually will call `Condition` method `signal` to transition the thread waiting on the condition object back to the runnable state. If multiple threads may be waiting on the condition object, a separate thread can call `Condition` method `signalAll` as a safeguard to ensure that all the waiting threads have another opportunity to perform their tasks. If this is not done, starvation might occur.



Common Programming

Error 23.2

An `IllegalMonitorStateException` occurs if a thread issues an `await`, a `signal`, or a `signalAll` on a `Condition` object that was created from a `ReentrantLock` without having acquired the lock for that

Condition object.

Lock and Condition vs. the synchronized Keyword

In some applications, using `Lock` and `Condition` objects may be preferable to using the `synchronized` keyword. `Locks` allow you to *interrupt* waiting threads or to specify a *timeout* for waiting to acquire a lock, which is not possible using the `synchronized` keyword. Also, a `Lock` is *not* constrained to be acquired and released in the *same* block of code, which is the case with the `synchronized` keyword. `Condition` objects allow you to specify multiple conditions on which threads may *wait*. Thus, it's possible to indicate to waiting threads that a specific condition object is now true by calling `signal` or `signalAll` on that `Condition` object. With `synchronized`, there's no way to explicitly state the condition on which threads are waiting, and thus there's no way to notify threads waiting on one condition that they may proceed without also signaling threads waiting on any other conditions. There are other possible advantages to using `Lock` and `Condition` objects, but generally it's best to use the `synchronized` keyword unless your application requires advanced synchronization capabilities.



Software Engineering Observation 23.7

Think of `Lock` and `Condition` as an advanced version of `synchronized`. `Lock` and `Condition` support timed waits, interruptible waits and multiple `Condition` queues per `Lock`—if you do not need one of these features, you do not need `Lock` and `Condition`.



Error-Prevention Tip 23.6

Using interfaces `Lock` and `Condition` is error prone—`unlock` is not guaranteed to be called, whereas the monitor in a `synchronized` statement will always be released when the statement completes execution. Of course, you can guarantee that `unlock` will be called if it's placed in a `finally` block, as we do in [Fig. 23.20](#).

Using Locks and Conditions to Implement Synchronization

We now implement the producer/consumer relationship using `Lock` and `Condition` objects to coordinate access to a shared single-element buffer (Figs. 23.20 and 23.21). In this case, each produced value is correctly consumed exactly once. Again, 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`

Class `SynchronizedBuffer` (Fig. 23.20) contains five fields. Line 10 initializes `Lock` instance variable `accessLock` with a new `ReentrantLock`. We did not specify a *fairness policy* in this example, because at any time only a single `Producer` or `Consumer` will be waiting to acquire the `Lock`. Lines 13–14 create two `Conditions` using `Lock` method `newCondition`:

- `Condition canWrite` contains a queue for a `Producer` thread waiting while the buffer is *full* (i.e., there's data in the buffer that the `Consumer` has not read yet). If the buffer is *full*, the `Producer` calls method `await` on this `Condition`. When the `Consumer` reads data from a *full* buffer, it calls method `signal` on this `Condition`.
- `Condition canRead` contains a queue for a `Consumer` thread waiting while the buffer is *empty* (i.e., there's no data in the buffer for the `Consumer` to read). If the buffer is *empty*, the `Consumer` calls method `await` on this `Condition`. When the `Producer` writes to the *empty* buffer, it calls method `signal` on this `Condition`.

The `int` variable `buffer` (line 16) holds the shared mutable data. The `boolean` variable `occupied` (line 17) keeps track of whether the buffer currently holds data (that the `Consumer` should read).

```
1  // Fig. 23.20: SynchronizedBuffer.java
2  // Synchronizing access to a shared integer using
3      // interfaces
4  import java.util.concurrent.locks.Lock;
5  import java.util.concurrent.locks.ReentrantLock;
6  import java.util.concurrent.locks.Condition;
7
8  public class SynchronizedBuffer implements Buffer {
9      // Lock to control synchronization with this
10     private final Lock accessLock = new ReentrantLock();
11
12     // conditions to control reading and writing
13     private final Condition canWrite = accessLock.newCondition();
14     private final Condition canRead = accessLock.newCondition();
15
16     private int buffer = -1; // shared by producer and consumer
17     private boolean occupied = false; // whether buffer is occupied
18
19     // place int value into buffer
20     @Override
21     public void blockingPut(int value) throws InterruptedException {
22         accessLock.lock(); // lock this object
23
24         // output thread information and buffer information
25         try {
26             // while buffer is not empty, place thread on canWrite
27             while (occupied) {
28                 System.out.println("Producer tries to put " + value + " but buffer is full");
29                 displayState("Buffer full. Producer blocked");
30                 canWrite.await(); // wait until buffer is empty
31             }
32
33             buffer = value; // set new buffer value
```

```

34
35     // indicate producer cannot store another
36     // until consumer retrieves current buffer
37     occupied = true;
38
39     displayState("Producer writes " + buffer);
40
41     // signal any threads waiting to read from buffer
42     canRead.signalAll();
43 }
44 finally {
45     accessLock.unlock(); // unlock this object
46 }
47 }
48
49 // return value from buffer
50 @Override
51 public int blockingGet() throws InterruptedException
52 {
53     int readValue = 0; // initialize value read
54     accessLock.lock(); // lock this object
55
56     // output thread information and buffer information
57     try {
58         // if there is no data to read, place thread in wait
59         while (!occupied) {
60             System.out.println("Consumer tries to read from buffer");
61             displayState("Buffer empty. Consumer waiting");
62             canRead.await(); // wait until buffer is not empty
63         }
64
65         // indicate that producer can store another buffer
66         // because consumer just retrieved buffer
67         occupied = false;
68
69         readValue = buffer; // retrieve value from buffer
70         displayState("Consumer reads " + readValue);
71
72         // signal any threads waiting for buffer
73         canWrite.signalAll();
74     }

```

```

74         finally {
75             accessLock.unlock(); // unlock this obj
76         }
77
78         return readValue;
79     }
80
81     // display current operation and buffer state
82     private void displayState(String operation) {
83         try {
84             accessLock.lock(); // lock this object
85             System.out.printf("%-40s%d\t\t\tb%n%n",
86                             operation,
87                             readValue);
88             finally {
89                 accessLock.unlock(); // unlock this obj
90             }
91         }
92     }

```

Fig. 23.20

Synchronizing access to a shared integer using the `Lock` and `Condition` interfaces.

Method `blockingPut` calls method `lock` on the `SynchronizedBuffer`'s `accessLock` (line 22). If the lock is *available* (i.e., no other thread has acquired it), this thread now owns the lock and the thread continues. If the lock is *unavailable* (i.e., it's held by another thread), method `lock` waits until the lock is released. After the lock is acquired, lines 25–43 execute. Line 27 determines whether `buffer` is full. If

it is, lines 28–29 display a message indicating that the thread will *wait*. Line 30 calls `Condition` method `await` on the `canWrite` condition object, which temporarily releases the `SynchronizedBuffer`'s `Lock` and *waits* for a signal from the `Consumer` that `buffer` is available for writing. When `buffer` is available, the method proceeds, writing to `buffer` (line 33), setting `occupied` to `true` (line 37) and displaying a message indicating that the producer wrote a value (line 39). Line 42 calls `Condition` method `signal` on the condition object `canRead` to notify the waiting `Consumer` (if there is one) that the buffer has new data. Line 45 calls method `unlock` from a `finally` block to *release* the lock and allow the `Consumer` to proceed.

Line 53 of method `blockingGet` calls method `lock` to *acquire* the `Lock`. This method *waits* until the `Lock` is *available*. Once the `Lock` is *acquired*, line 58 determines whether the buffer is *empty*. If so, line 61 calls method `await` on condition object `canRead`. Recall that method `signal` is called on variable `canRead` in the `blockingPut` method (line 42). When the `Condition` object is *signaled*, the `blockingGet` method continues. Lines 66–69 set `occupied` to `false`, store the value of `buffer` in `readValue` and output the `readValue`. Then line 72 *signals* the condition object `canWrite`. This awakens the `Producer` if it's indeed *waiting* for the buffer to be *emptied*. Line 75 calls method `unlock` from a `finally` block to *release* the lock, and line 78 returns `readValue` to the caller.



Common Programming Error 23.3

Forgetting to signal a waiting thread is a logic error. The thread will remain in the waiting state, preventing it from proceeding. This can lead to indefinite postponement or deadlock.

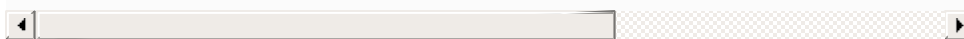
Class SharedBufferTest2

Class `SharedBufferTest2` (Fig. 23.21) is identical to that of Fig. 23.17. Study the outputs in Fig. 23.21. Observe that every integer produced is consumed exactly once—no values are lost, and no values are consumed more than once. The `Lock` and `Condition` objects ensure that the `Producer` and `Consumer` cannot perform their tasks unless it's their turn. The `Producer` *must* go first, the `Consumer` *must* wait if the `Producer` has not produced since the `Consumer` last consumed and the `Producer` *must* wait 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.

```

1  // Fig. 23.21: SharedBufferTest2.java
2  // Two threads manipulating a synchronized buffer
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 new thread pool
10         ExecutorService executorService = Executors
11
12         // create SynchronizedBuffer to store ints
13         Buffer sharedLocation = new Synchroniz
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
-----	-----	-----
Producer writes 1	1	true
Producer tries to write. Buffer full. Producer waits.	1	true
Consumer reads 1	1	false
Producer writes 2	2	true

Producer tries to write. Buffer full. Producer waits.	2	true
--	---	------

Consumer reads 2	2	false
Producer writes 3	3	true
Consumer reads 3	3	false
Producer writes 4	4	true
Consumer reads 4	4	false
Consumer tries to read. Buffer empty. Consumer waits.	4	false
Producer writes 5	5	true
Consumer reads 5	5	false
Consumer tries to read. Buffer empty. Consumer waits.	5	false
Producer writes 6	6	true
Consumer reads 6	6	false
Producer writes 7	7	true
Consumer reads 7	7	false
Producer writes 8	8	true
Consumer reads 8	8	false
Producer writes 9	9	true
Consumer reads 9	9	false
Producer writes 10	10	true
Producer done producing Terminating Producer	10	false

Consumer reads 10

Consumer read values totaling 55

Terminating Consumer

Fig. 23.21

Two threads manipulating a synchronized buffer.