Chapter 14

Building Custom Synchronizers

The class libraries include a number of *state-dependent* classes—those having operations with *state-based preconditions*—such as `FutureTask`, `Semaphore`, and `BlockingQueue`. For example, you cannot remove an item from an empty queue or retrieve the result of a task that has not yet finished; before these operations can proceed, you must wait until the queue enters the "nonempty" state or the task enters the "completed" state.

The easiest way to construct a state-dependent class is usually to build on top of an existing state-dependent library class; we did this in `ValueLatch` on page 187, using a `CountDownLatch` to provide the required blocking behavior. But if the library classes do not provide the functionality you need, you can also build your own synchronizers using the low-level mechanisms provided by the language and libraries, including intrinsic *condition queues*, explicit `Condition` objects, and the `AbstractQueuedSynchronizer` framework. This chapter explores the various options for implementing state dependence and the rules for using the state dependence mechanisms provided by the platform.

14.1 Managing state dependence

In a single-threaded program, if a state-based precondition (like "the connection pool is nonempty") does not hold when a method is called, it will never become true. Therefore, classes in sequential programs can be coded to fail when their preconditions do not hold. But in a concurrent program, state-based conditions can change through the actions of other threads: a pool that was empty a few instructions ago can become nonempty because another thread returned an element. State-dependent methods on concurrent objects can sometimes get away with failing when their preconditions are not met, but there is often a better alternative: wait for the precondition to become true.

State-dependent operations that *block* until the operation can proceed are more convenient and less error-prone than those that simply fail. The built-in condition queue mechanism enables threads to block until an object has entered a state that allows progress and to wake blocked threads when they may be able to make further progress. We cover the details of condition queues in Section 14.2, but to

motivate the value of an efficient condition wait mechanism, we first show how state dependence might be (painfully) tackled using polling and sleeping.

A blocking state-dependent action takes the form shown in Listing 14.1. The pattern of locking is somewhat unusual in that the lock is released and reacquired in the middle of the operation. The state variables that make up the precondition must be guarded by the object's lock, so that they can remain constant while the precondition is tested. But if the precondition does not hold, the lock must be released so another thread can modify the object state—otherwise the precondition will never become true. The lock must then be reacquired before testing the precondition again.

```
acquire lock on object state
while (precondition does not hold) {
    release lock
    wait until precondition might hold
    optionally fail if interrupted or timeout expires
    reacquire lock
}
perform action
release lock
```

Listing 14.1. Structure of blocking state-dependent actions.

Bounded buffers such as `ArrayBlockingQueue` are commonly used in producer-consumer designs. A bounded buffer provides *put* and *take* operations, each of which has preconditions: you cannot take an element from an empty buffer, nor put an element into a full buffer. State dependent operations can deal with precondition failure by throwing an exception or returning an error status (making it the caller's problem), or by blocking until the object transitions to the right state.

We're going to develop several implementations of a bounded buffer that take different approaches to handling precondition failure. Each extends `BaseBoundedBuffer` in Listing 14.2, which implements a classic array-based circular buffer where the buffer state variables (`buf`, `head`, `tail`, and `count`) are guarded by the buffer's intrinsic lock. It provides synchronized `doPut` and `doTake` methods that are used by subclasses to implement the `put` and `take` operations; the underlying state is hidden from the subclasses.

14.1.1 Example: propagating precondition failure to callers

`GrumpyBoundedBuffer` in Listing 14.3 is a crude first attempt at implementing a bounded buffer. The `put` and `take` methods are `synchronized` to ensure exclusive access to the buffer state, since both employ check-then-act logic in accessing the buffer.

While this approach is easy enough to implement, it is annoying to use. Exceptions are supposed to be for exceptional conditions [EJ Item 39]. "Buffer is

```
@ThreadSafe
public abstract class BaseBoundedBuffer<V> {
    @GuardedBy("this") private final V[] buf;
    @GuardedBy("this") private int tail;
    @GuardedBy("this") private int head;
    @GuardedBy("this") private int count;

    protected BaseBoundedBuffer(int capacity) {
        this.buf = (V[]) new Object[capacity];
    }

    protected synchronized final void doPut(V v) {
        buf[tail] = v;
        if (++tail == buf.length)
            tail = 0;
        ++count;
    }

    protected synchronized final V doTake() {
        V v = buf[head];
        buf[head] = null;
        if (++head == buf.length)
            head = 0;
        --count;
        return v;
    }

    public synchronized final boolean isFull() {
        return count == buf.length;
    }

    public synchronized final boolean isEmpty() {
        return count == 0;
    }
}
```

Listing 14.2. Base class for bounded buffer implementations.

```
@ThreadSafe
public class GrumpyBoundedBuffer<V> extends BaseBoundedBuffer<V> {
    public GrumpyBoundedBuffer(int size) { super(size); }

    public synchronized void put(V v) throws BufferFullException {
        if (isFull())
            throw new BufferFullException();
        doPut(v);
    }

    public synchronized V take() throws BufferEmptyException {
        if (isEmpty())
            throw new BufferEmptyException();
        return doTake();
    }
}
```

Listing 14.3. Bounded buffer that balks when preconditions are not met.

full" is not an exceptional condition for a bounded buffer any more than "red" is an exceptional condition for a traffic signal. The simplification in implementing the buffer (forcing the caller to manage the state dependence) is more than made up for by the substantial complication in using it, since now the caller must be prepared to catch exceptions and possibly retry for every buffer operation.[1] A well-structured call to `take` is shown in Listing 14.4—not very pretty, especially if `put` and `take` are called throughout the program.

```
while (true) {
    try {
        V item = buffer.take();
        // use item
        break;
    } catch (BufferEmptyException e) {
        Thread.sleep(SLEEP_GRANULARITY);
    }
}
```

Listing 14.4. Client logic for calling `GrumpyBoundedBuffer`.

A variant of this approach is to return an error value when the buffer is in the wrong state. This is a minor improvement in that it doesn't abuse the exception mechanism by throwing an exception that really means "sorry, try again",

1. Pushing the state dependence back to the caller also makes it nearly impossible to do things like preserve FIFO ordering; by forcing the caller to retry, you lose the information of who arrived first.

but it does not address the fundamental problem: that callers must deal with precondition failures themselves.[2]

The client code in Listing 14.4 is not the only way to implement the retry logic. The caller could retry the `take` immediately, without sleeping—an approach known as *busy waiting* or *spin waiting*. This could consume quite a lot of CPU time if the buffer state does not change for a while. On the other hand, if the caller decides to sleep so as not to consume so much CPU time, it could easily "oversleep" if the buffer state changes shortly after the call to `sleep`. So the client code is left with the choice between the poor CPU usage of spinning and the poor responsiveness of sleeping. (Somewhere between busy waiting and sleeping would be calling `Thread.yield` in each iteration, which is a hint to the scheduler that this would be a reasonable time to let another thread run. If you are waiting for another thread to do something, that something might happen faster if you yield the processor rather than consuming your full scheduling quantum.)

14.1.2 Example: crude blocking by polling and sleeping

`SleepyBoundedBuffer` in Listing 14.5 attempts to spare callers the inconvenience of implementing the retry logic on each call by encapsulating the same crude "poll and sleep" retry mechanism within the `put` and `take` operations. If the buffer is empty, `take` sleeps until another thread puts some data into the buffer; if the buffer is full, `put` sleeps until another thread makes room by removing some data. This approach encapsulates precondition management and simplifies using the buffer—definitely a step in the right direction.

The implementation of `SleepyBoundedBuffer` is more complicated than the previous attempt.[3] The buffer code must test the appropriate state condition with the buffer lock held, because the variables that represent the state condition are guarded by the buffer lock. If the test fails, the executing thread sleeps for a while, first releasing the lock so other threads can access the buffer.[4] Once the thread wakes up, it reacquires the lock and tries again, alternating between sleeping and testing the state condition until the operation can proceed.

From the perspective of the caller, this works nicely—if the operation can proceed immediately, it does, and otherwise it blocks—and the caller need not deal with the mechanics of failure and retry. Choosing the sleep granularity is a tradeoff between responsiveness and CPU usage; the smaller the sleep granularity, the more responsive, but also the more CPU resources consumed. Figure 14.1 shows how sleep granularity can affect responsiveness: there may be a delay between when buffer space becomes available and when the thread wakes up and checks again.

2. `Queue` offers both of these options—`poll` returns `null` if the queue is empty, and `remove` throws an exception—but `Queue` is not intended for use in producer-consumer designs. `BlockingQueue`, whose operations block until the queue is in the right state to proceed, is a better choice when producers and consumers will execute concurrently.

3. We will spare you the details of Snow White's other five bounded buffer implementations, especially `SneezyBoundedBuffer`.

4. It is usually a bad idea for a thread to go to sleep or otherwise block with a lock held, but in this case is even worse because the desired condition (buffer is full/empty) can never become true if the lock is not released!

```
@ThreadSafe
public class SleepyBoundedBuffer<V> extends BaseBoundedBuffer<V> {
    public SleepyBoundedBuffer(int size) { super(size); }

    public void put(V v) throws InterruptedException {
        while (true) {
            synchronized (this) {
                if (!isFull()) {
                    doPut(v);
                    return;
                }
            }
            Thread.sleep(SLEEP_GRANULARITY);
        }
    }

    public V take() throws InterruptedException {
        while (true) {
            synchronized (this) {
                if (!isEmpty())
                    return doTake();
            }
            Thread.sleep(SLEEP_GRANULARITY);
        }
    }
}
```

Listing 14.5. Bounded buffer using crude blocking.

`SleepyBoundedBuffer` also creates another requirement for the caller—dealing with `InterruptedException`. When a method blocks waiting for a condition to become true, the polite thing to do is to provide a cancellation mechanism (see Chapter 7). Like most well-behaved blocking library methods, `SleepyBoundedBuffer` supports cancellation through interruption, returning early and throwing `InterruptedException` if interrupted.

These attempts to synthesize a blocking operation from polling and sleeping were fairly painful. It would be nice to have a way of suspending a thread but ensuring that it is awakened promptly when a certain condition (such as the buffer being no longer full) becomes true. This is exactly what *condition queues* do.

14.1.3 Condition queues to the rescue

Condition queues are like the "toast is ready" bell on your toaster. If you are listening for it, you are notified promptly when your toast is ready and can drop what you are doing (or not, maybe you want to finish the newspaper first) and

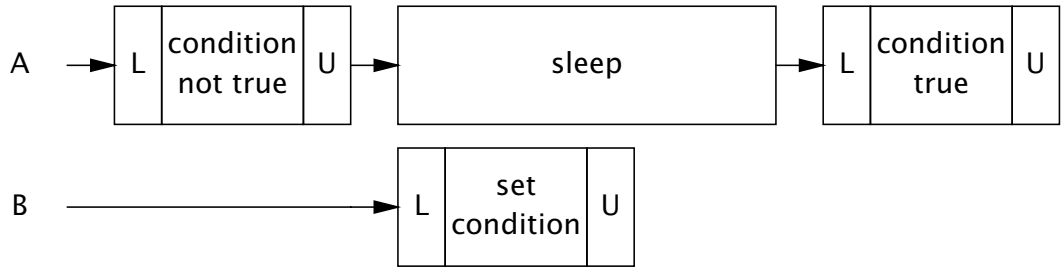

FIGURE 14.1. Thread oversleeping because the condition became true just after it went to sleep.

get your toast. If you are not listening for it (perhaps you went outside to get the newspaper), you could miss the notification, but on return to the kitchen you can observe the state of the toaster and either retrieve the toast if it is finished or start listening for the bell again if it is not.

A *condition queue* gets its name because it gives a group of threads—called the *wait set*—a way to wait for a specific condition to become true. Unlike typical queues in which the elements are data items, the elements of a condition queue are the threads waiting for the condition.

Just as each Java object can act as a lock, each object can also act as a condition queue, and the `wait`, `notify`, and `notifyAll` methods in `Object` constitute the API for intrinsic condition queues. An object's intrinsic lock and its intrinsic condition queue are related: in order to call any of the condition queue methods on object X, you must hold the lock on X. This is because the mechanism for waiting for state-based conditions is necessarily tightly bound to the mechanism for preserving state consistency: you cannot wait for a condition unless you can examine the state, and you cannot release another thread from a condition wait unless you can modify the state.

`Object.wait` atomically releases the lock and asks the OS to suspend the current thread, allowing other threads to acquire the lock and therefore modify the object state. Upon waking, it reacquires the lock before returning. Intuitively, calling `wait` means "I want to go to sleep, but wake me when something interesting happens", and calling the notification methods means "something interesting happened".

`BoundedBuffer` in Listing 14.6 implements a bounded buffer using `wait` and `notifyAll`. This is simpler than the sleeping version, and is both more efficient (waking up less frequently if the buffer state does not change) and more responsive (waking up promptly when an interesting state change happens). This is a big improvement, but note that the introduction of condition queues didn't change the semantics compared to the sleeping version. It is simply an optimization in several dimensions: CPU efficiency, context-switch overhead, and responsiveness. Condition queues don't let you do anything you can't do with sleeping and polling[5], but they make it a lot easier and more efficient to express and manage

5. This is not quite true; a *fair* condition queue can guarantee the relative order in which threads are released from the wait set. Intrinsic condition queues, like intrinsic locks, do not offer fair queueing; explicit `Conditions` offer a choice of fair or nonfair queueing.

state dependence.

```
@ThreadSafe
public class BoundedBuffer<V> extends BaseBoundedBuffer<V> {
    // CONDITION PREDICATE: not-full (!isFull())
    // CONDITION PREDICATE: not-empty (!isEmpty())

    public BoundedBuffer(int size) { super(size); }

    // BLOCKS-UNTIL: not-full
    public synchronized void put(V v) throws InterruptedException {
        while (isFull())
            wait();
        doPut(v);
        notifyAll();
    }

    // BLOCKS-UNTIL: not-empty
    public synchronized V take() throws InterruptedException {
        while (isEmpty())
            wait();
        V v = doTake();
        notifyAll();
        return v;
    }
}
```

Listing 14.6. Bounded buffer using condition queues.

`BoundedBuffer` is finally good enough to use—it is easy to use and manages state dependence sensibly.[6] A production version should also include timed versions of `put` and `take`, so that blocking operations can time out if they cannot complete within a time budget. The timed version of `Object.wait` makes this easy to implement.

14.2 Using condition queues

Condition queues make it easier to build efficient and responsive state-dependent classes, but they are still easy to use incorrectly; there are a lot of rules regarding their proper use that are not enforced by the compiler or platform. (This is one of the reasons to build on top of classes like `LinkedBlockingQueue`, `CountDownLatch`, `Semaphore`, and `FutureTask` when you can; if you can get away with it, it is a lot easier.)

6. `ConditionBoundedBuffer` in Section 14.3 is even better: it is more efficient because it can use single notification instead of `notifyAll`.

14.2.1 The condition predicate

The key to using condition queues correctly is identifying the *condition predicates* that the object may wait for. It is the condition predicate that causes much of the confusion surrounding `wait` and `notify`, because it has no instantiation in the API and nothing in either the language specification or the JVM implementation ensures its correct use. In fact, it is not mentioned directly at all in the language specification or the Javadoc. But without it, condition waits would not work.

The condition predicate is the precondition that makes an operation state-dependent in the first place. In a bounded buffer, `take` can proceed only if the buffer is not empty; otherwise it must wait. For `take`, the condition predicate is "the buffer is not empty", which `take` must test for before proceeding. Similarly, the condition predicate for `put` is "the buffer is not full". Condition predicates are expressions constructed from the state variables of the class; `BaseBoundedBuffer` tests for "buffer not empty" by comparing `count` to zero, and tests for "buffer not full" by comparing `count` to the buffer size.

> Document the condition predicate(s) associated with a condition queue and the operations that wait on them.

There is an important three-way relationship in a condition wait involving locking, the `wait` method, and a condition predicate. The condition predicate involves state variables, and the state variables are guarded by a lock, so before testing the condition predicate, we must hold that lock. The lock object and the condition queue object (the object on which `wait` and `notify` are invoked) must also be the same object.

In `BoundedBuffer`, the buffer state is guarded by the buffer lock and the buffer object is used as the condition queue. The `take` method acquires the buffer lock and then tests the condition predicate (that the buffer is nonempty). If the buffer is indeed nonempty, it removes the first element, which it can do because it still holds the lock guarding the buffer state.

If the condition predicate is not true (the buffer is empty), `take` must wait until another thread puts an object in the buffer. It does this by calling `wait` on the buffer's intrinsic condition queue, which requires holding the lock on the condition queue object. As careful design would have it, `take` already holds that lock, which it needed to test the condition predicate (and if the condition predicate was true, to modify the buffer state in the same atomic operation). The `wait` method releases the lock, blocks the current thread, and waits until the specified timeout expires, the thread is interrupted, or the thread is awakened by a notification. After the thread wakes up, `wait` reacquires the lock before returning. A thread waking up from `wait` gets no special priority in reacquiring the lock; it contends for the lock just like any other thread attempting to enter a `synchronized` block.

Every call to `wait` is implicitly associated with a specific *condition predicate*. When calling `wait` regarding a particular condition predicate, the caller must already hold the lock associated with the condition queue, and that lock must also guard the state variables from which the condition predicate is composed.

14.2.2 Waking up too soon

As if the three-way relationship among the lock, the condition predicate, and the condition queue were not complicated enough, that `wait` returns does not necessarily mean that the condition predicate the thread is waiting for has become true.

A single intrinsic condition queue may be used with more than one condition predicate. When your thread is awakened because someone called `notifyAll`, that doesn't mean that the condition predicate *you* were waiting for is now true. (This is like having your toaster and coffee maker share a single bell; when it rings, you still have to look to see which device raised the signal.)[7] Additionally, `wait` is even allowed to return "spuriously"—not in response to any thread calling `notify`.[8]

When control re-enters the code calling `wait`, it has reacquired the lock associated with the condition queue. Is the condition predicate now true? Maybe. It might have been true at the time the notifying thread called `notifyAll`, but could have become false again by the time *you* reacquire the lock. Other threads may have acquired the lock and changed the object's state between when your thread was awakened and when `wait` reacquired the lock. Or maybe it hasn't been true at all since you called `wait`. You don't know why another thread called `notify` or `notifyAll`; maybe it was because *another* condition predicate associated with the same condition queue became true. Multiple condition predicates per condition queue are quite common—`BoundedBuffer` uses the same condition queue for both the "not full" and "not empty" predicates.[9]

For all these reasons, when you wake up from `wait` you must test the condition predicate *again*, and go back to waiting (or fail) if it is not yet true. Since you can wake up repeatedly without your condition predicate being true, you must therefore always call `wait` from within a loop, testing the condition predicate in each iteration. The canonical form for a condition wait is shown in Listing 14.7.

7. This situation actually describes Tim's kitchen pretty well; so many devices beep that when you hear one, you have to inspect the toaster, the microwave, the coffee maker, and several others to determine the cause of the signal.
8. To push the breakfast analogy way too far, this is like a toaster with a loose connection that makes the bell go off when the toast is ready but also sometimes when it is not ready.
9. It is actually possible for threads to be waiting for both "not full" and "not empty" at the same time! This can happen when the number of producers/consumers exceeds the buffer capacity.

```
void stateDependentMethod() throws InterruptedException {
    // condition predicate must be guarded by lock
    synchronized(lock) {
        while (!conditionPredicate())
            lock.wait();
        // object is now in desired state
    }
}
```

LISTING 14.7. Canonical form for state-dependent methods.

When using condition waits (`Object.wait` or `Condition.await`):

- Always have a condition predicate—some test of object state that must hold before proceeding;
- Always test the condition predicate before calling `wait`, and again after returning from `wait`;
- Always call `wait` in a loop;
- Ensure that the state variables making up the condition predicate are guarded by the lock associated with the condition queue;
- Hold the lock associated with the the condition queue when calling `wait`, `notify`, or `notifyAll`; and
- Do not release the lock after checking the condition predicate but before acting on it.

14.2.3 Missed signals

Chapter 10 discussed liveness failures such as deadlock and livelock. Another form of liveness failure is *missed signals*. A missed signal occurs when a thread must wait for a specific condition that is already true, but fails to check the condition predicate before waiting. Now the thread is waiting to be notified of an event that has already occurred. This is like starting the toast, going out to get the newspaper, having the bell go off while you are outside, and then sitting down at the kitchen table waiting for the toast bell. You could wait a long time—potentially forever.[10] Unlike the marmalade for your toast, notification is not "sticky"—if thread *A* notifies on a condition queue and thread *B* subsequently waits on that same condition queue, *B* does *not* immediately wake up—another notification is required to wake *B*. Missed signals are the result of coding errors like those warned against in the list above, such as failing to test the condition predicate before calling `wait`. If you structure your condition waits as in Listing 14.7, you will not have problems with missed signals.

10. In order to emerge from this wait, someone else would have to make toast, but this will just make matters worse; when the bell rings, you will then have a disagreement about toast ownership.

14.2.4 Notification

So far, we've described half of what goes on in a condition wait: waiting. The other half is notification. In a bounded buffer, `take` blocks if called when the buffer is empty. In order for `take` to *unblock* when the buffer becomes nonempty, we must ensure that *every* code path in which the buffer could become nonempty performs a notification. In `BoundedBuffer`, there is only one such place—after a `put`. So `put` calls `notifyAll` after successfully adding an object to the buffer. Similarly, `take` calls `notifyAll` after removing an element to indicate that the buffer may no longer be full, in case any threads are waiting on the "not full" condition.

> Whenever you wait on a condition, make sure that someone will perform a notification whenever the condition predicate becomes true.

There are two notification methods in the condition queue API—`notify` and `notifyAll`. To call either, you must hold the lock associated with the condition queue object. Calling `notify` causes the JVM to select one thread waiting on that condition queue to wake up; calling `notifyAll` wakes up *all* the threads waiting on that condition queue. Because you must hold the lock on the condition queue object when calling `notify` or `notifyAll`, and waiting threads cannot return from `wait` without reacquiring the lock, the notifying thread should release the lock quickly to ensure that the waiting threads are unblocked as soon as possible.

Because multiple threads could be waiting on the same condition queue for different condition predicates, using `notify` instead of `notifyAll` can be dangerous, primarily because single notification is prone to a problem akin to missed signals.

`BoundedBuffer` provides a good illustration of why `notifyAll` should be preferred to single `notify` in most cases. The condition queue is used for two different condition predicates: "not full" and "not empty". Suppose thread A waits on a condition queue for predicate P_A, while thread B waits on the same condition queue for predicate P_B. Now, suppose P_B becomes true and thread C performs a single `notify`: the JVM will wake up one thread of its own choosing. If A is chosen, it will wake up, see that P_A is not yet true, and go back to waiting. Meanwhile, B, which could now make progress, does not wake up. This is not exactly a missed signal—it's more of a "hijacked signal"—but the problem is the same: a thread is waiting for a signal that has (or should have) already occurred.

Single notify can be used instead of notifyAll only when both of the following conditions hold:

Uniform waiters. Only one condition predicate is associated with the condition queue, and each thread executes the same logic upon returning from wait; and

One-in, one-out. A notification on the condition variable enables at most one thread to proceed.

BoundedBuffer meets the one-in, one-out requirement, but does not meet the uniform waiters requirement because waiting threads might be waiting for either the "not full" and "not empty" condition. A "starting gate" latch like that used in TestHarness on page 96, in which a single event releases a set of threads, does not meet the one-in, one-out requirement because opening the starting gate lets multiple threads proceed.

Most classes don't meet these requirements, so the prevailing wisdom is to use notifyAll in preference to single notify. While this may be inefficient, it is much easier to ensure that your classes behave correctly when using notifyAll instead of notify.

This "prevailing wisdom" makes some people uncomfortable, and for good reason. Using notifyAll when only one thread can make progress is inefficient—sometimes a little, sometimes grossly so. If ten threads are waiting on a condition queue, calling notifyAll causes each of them to wake up and contend for the lock; then most or all of them will go right back to sleep. This means a lot of context switches and a lot of contended lock acquisitions for each event that enables (maybe) a single thread to make progress. (In the worst case, using notifyAll results in $O(n^2)$ wakeups where n would suffice.) This is another situation where performance concerns support one approach and safety concerns support the other.

The notification done by put and take in BoundedBuffer is conservative: a notification is performed every time an object is put into or removed from the buffer. This could be optimized by observing that a thread can be released from a wait only if the buffer goes from empty to not empty or from full to not full, and notifying only if a put or take effected one of these state transitions. This is called *conditional notification.* While conditional notification can improve performance, it is tricky to get right (and also complicates the implementation of subclasses) and so should be used carefully. Listing 14.8 illustrates using conditional notification in BoundedBuffer.put.

Single notification and conditional notification are optimizations. As always, follow the principle "First make it right, and then make it fast—*if* it is not already fast enough" when using these optimizations; it is easy to introduce strange liveness failures by applying them incorrectly.

```
public synchronized void put(V v) throws InterruptedException {
    while (isFull())
        wait();
    boolean wasEmpty = isEmpty();
    doPut(v);
    if (wasEmpty)
        notifyAll();
}
```

LISTING 14.8. Using conditional notification in `BoundedBuffer.put`.

14.2.5 Example: a gate class

The starting gate latch in `TestHarness` on page 96 was constructed with an initial count of one, creating a *binary latch*: one with two states, the initial state and the terminal state. The latch prevents threads from passing the starting gate until it is opened, at which point all the threads can pass through. While this latching mechanism is often exactly what is needed, sometimes it is a drawback that a gate constructed in this manner cannot be reclosed once opened.

It is easy to develop a recloseable `ThreadGate` class using condition waits, as shown in Listing 14.9. `ThreadGate` lets the gate be opened and closed, providing an `await` method that blocks until the gate is opened. The `open` method uses `notifyAll` because the semantics of this class fail the "one-in, one-out" test for single notification.

The condition predicate used by `await` is more complicated than simply testing `isOpen`. This is needed because if N threads are waiting at the gate at the time it is opened, they should all be allowed to proceed. But, if the gate is opened and closed in rapid succession, all threads might not be released if `await` examines only `isOpen`: by the time all the threads receive the notification, reacquire the lock, and emerge from `wait`, the gate may have closed again. So `ThreadGate` uses a somewhat more complicated condition predicate: every time the gate is closed, a "generation" counter is incremented, and a thread may pass `await` if the gate is open now or if the gate has opened since this thread arrived at the gate.

Since `ThreadGate` only supports waiting for the gate to open, it performs notification only in `open`; to support both "wait for open" and "wait for close" operations, it would have to notify in both `open` and `close`. This illustrates why state-dependent classes can be fragile to maintain—the addition of a new state-dependent operation may require modifying many code paths that modify the object state so that the appropriate notifications can be performed.

14.2.6 Subclass safety issues

Using conditional or single notification introduces constraints that can complicate subclassing [CPJ 3.3.3.3]. If you want to support subclassing at all, you must structure your class so subclasses can add the appropriate notification on behalf

```
@ThreadSafe
public class ThreadGate {
    // CONDITION-PREDICATE: opened-since(n) (isOpen || generation>n)
    @GuardedBy("this") private boolean isOpen;
    @GuardedBy("this") private int generation;

    public synchronized void close() {
        isOpen = false;
    }

    public synchronized void open() {
        ++generation;
        isOpen = true;
        notifyAll();
    }

    // BLOCKS-UNTIL: opened-since(generation on entry)
    public synchronized void await() throws InterruptedException {
        int arrivalGeneration = generation;
        while (!isOpen && arrivalGeneration == generation)
            wait();
    }
}
```

LISTING 14.9. Recloseable gate using `wait` and `notifyAll`.

of the base class if it is subclassed in a way that violates one of the requirements for single or conditional notification.

A state-dependent class should either fully expose (and document) its waiting and notification protocols to subclasses, or prevent subclasses from participating in them at all. (This is an extension of "design and document for inheritance, or else prohibit it" [EJ Item 15].) At the very least, designing a state-dependent class for inheritance requires exposing the condition queues and locks and documenting the condition predicates and synchronization policy; it may also require exposing the underlying state variables. (The worst thing a state-dependent class can do is expose its state to subclasses but *not* document its protocols for waiting and notification; this is like a class exposing its state variables but not documenting its invariants.)

One option for doing this is to effectively prohibit subclassing, either by making the class `final` or by hiding the condition queues, locks, and state variables from subclasses. Otherwise, if the subclass does something to undermine the way the base class uses `notify`, it needs to be able to repair the damage. Consider an unbounded blocking stack in which the pop operation blocks if the stack is empty but the push operation can always proceed. This meets the requirements for single notification. If this class uses single notification and a subclass adds a blocking "pop two consecutive elements" method, there are now two classes of

waiters: those waiting to pop one element and those waiting to pop two. But if the base class exposes the condition queue and documents its protocols for using it, the subclass can override the push method to perform a `notifyAll`, restoring safety.

14.2.7 Encapsulating condition queues

It is generally best to encapsulate the condition queue so that it is not accessible outside the class hierarchy in which it is used. Otherwise, callers might be tempted to think they understand your protocols for waiting and notification and use them in a manner inconsistent with your design. (It is impossible to enforce the uniform waiters requirement for single notification unless the condition queue object is inaccessible to code you do not control; if alien code mistakenly waits on your condition queue, this could subvert your notification protocol and cause a hijacked signal.)

Unfortunately, this advice—to encapsulate objects used as condition queues—is not consistent with the most common design pattern for thread-safe classes, in which an object's intrinsic lock is used to guard its state. `BoundedBuffer` illustrates this common idiom, where the buffer object itself is the lock and condition queue. However, `BoundedBuffer` could be easily restructured to use a private lock object and condition queue; the only difference would be that it would no longer support any form of client-side locking.

14.2.8 Entry and exit protocols

Wellings (Wellings, 2004) characterizes the proper use of `wait` and `notify` in terms of *entry* and *exit protocols*. For each state-dependent operation and for each operation that modifies state on which another operation has a state dependency, you should define and document an entry and exit protocol. The entry protocol is the operation's condition predicate; the exit protocol involves examining any state variables that have been changed by the operation to see if they might have caused some other condition predicate to become true, and if so, notifying on the associated condition queue.

`AbstractQueuedSynchronizer`, upon which most of the state-dependent classes in `java.util.concurrent` are built (see Section 14.4), exploits the concept of exit protocol. Rather than letting synchronizer classes perform their own notification, it instead requires synchronizer methods to return a value indicating whether its action might have unblocked one or more waiting threads. This explicit API requirement makes it harder to "forget" to notify on some state transitions.

14.3 Explicit condition objects

As we saw in Chapter 13, explicit `Locks` can be useful in some situations where intrinsic locks are too inflexible. Just as `Lock` is a generalization of intrinsic locks, `Condition` (see Listing 14.10) is a generalization of intrinsic condition queues.

Intrinsic condition queues have several drawbacks. Each intrinsic lock can have only one associated condition queue, which means that in classes like BoundedBuffer multiple threads might wait on the same condition queue for different condition predicates, and the most common pattern for locking involves exposing the condition queue object. Both of these factors make it impossible to enforce the uniform waiter requirement for using notify. If you want to write a concurrent object with multiple condition predicates, or you want to exercise more control over the visibility of the condition queue, the explicit Lock and Condition classes offer a more flexible alternative to intrinsic locks and condition queues.

A Condition is associated with a single Lock, just as a condition queue is associated with a single intrinsic lock; to create a Condition, call Lock.newCondition on the associated lock. And just as Lock offers a richer feature set than intrinsic locking, Condition offers a richer feature set than intrinsic condition queues: multiple wait sets per lock, interruptible and uninterruptible condition waits, deadline-based waiting, and a choice of fair or nonfair queueing.

```
public interface Condition {
    void await() throws InterruptedException;
    boolean await(long time, TimeUnit unit)
            throws InterruptedException;
    long awaitNanos(long nanosTimeout) throws InterruptedException;
    void awaitUninterruptibly();
    boolean awaitUntil(Date deadline) throws InterruptedException;

    void signal();
    void signalAll();
}
```

Listing 14.10. Condition interface.

Unlike intrinsic condition queues, you can have as many Condition objects per Lock as you want. Condition objects inherit the fairness setting of their associated Lock; for fair locks, threads are released from Condition.await in FIFO order.

> Hazard warning: The equivalents of wait, notify, and notifyAll for Condition objects are await, signal, and signalAll. However, Condition extends Object, which means that it also has wait and notify methods. Be sure to use the proper versions—await and signal—instead!

Listing 14.11 shows yet another bounded buffer implementation, this time using two Conditions, notFull and notEmpty, to represent explicitly the "not full" and "not empty" condition predicates. When take blocks because the buffer is empty, it waits on notEmpty, and put unblocks any threads blocked in take by signaling on notEmpty.

The behavior of ConditionBoundedBuffer is the same as BoundedBuffer, but its use of condition queues is more readable—it is easier to analyze a class that uses multiple Conditions than one that uses a single intrinsic condition queue with multiple condition predicates. By separating the two condition predicates into separate wait sets, Condition makes it easier to meet the requirements for single notification. Using the more efficient signal instead of signalAll reduces the number of context switches and lock acquisitions triggered by each buffer operation.

Just as with built-in locks and condition queues, the three-way relationship among the lock, the condition predicate, and the condition variable must also hold when using explicit Locks and Conditions. The variables involved in the condition predicate must be guarded by the Lock, and the Lock must be held when testing the condition predicate and when calling await and signal.[11]

Choose between using explicit Conditions and intrinsic condition queues in the same way as you would choose between ReentrantLock and synchronized: use Condition if you need its advanced features such as fair queueing or multiple wait sets per lock, and otherwise prefer intrinsic condition queues. (If you already use ReentrantLock because you need its advanced features, the choice is already made.)

14.4 Anatomy of a synchronizer

The interfaces of ReentrantLock and Semaphore have a lot in common. Both classes act as a "gate", allowing only a limited number of threads to pass at a time; threads arrive at the gate and are allowed through (lock or acquire returns successfully), are made to wait (lock or acquire blocks), or are turned away (tryLock or tryAcquire returns false, indicating that the lock or permit did not become available in the time allowed). Further, both allow interruptible, uninterruptible, and timed acquisition attempts, and both allow a choice of fair or nonfair queueing of waiting threads.

Given this commonality, you might think that Semaphore was implemented on top of ReentrantLock, or perhaps ReentrantLock was implemented as a Semaphore with one permit. This would be entirely practical; it is a common exercise to prove that a counting semaphore can be implemented using a lock (as in SemaphoreOnLock in Listing 14.12) and that a lock can be implemented using a counting semaphore.

In actuality, they are both implemented using a common base class, AbstractQueuedSynchronizer (AQS)—as are many other synchronizers. AQS is a framework for building locks and synchronizers, and a surprisingly broad range of synchronizers can be built easily and efficiently using it. Not only are ReentrantLock and Semaphore built using AQS, but so are CountDownLatch, ReentrantReadWriteLock, SynchronousQueue,[12] and FutureTask.

11. ReentrantLock requires that the Lock be held when calling signal or signalAll, but Lock implementations are permitted to construct Conditions that do not have this requirement.
12. Java 6 replaces the AQS-based SynchronousQueue with a (more scalable) nonblocking version.

```
@ThreadSafe
public class ConditionBoundedBuffer<T> {
    protected final Lock lock = new ReentrantLock();
    // CONDITION PREDICATE: notFull (count < items.length)
    private final Condition notFull  = lock.newCondition();
    // CONDITION PREDICATE: notEmpty (count > 0)
    private final Condition notEmpty = lock.newCondition();
    @GuardedBy("lock")
    private final T[] items = (T[]) new Object[BUFFER_SIZE];
    @GuardedBy("lock") private int tail, head, count;

    // BLOCKS-UNTIL: notFull
    public void put(T x) throws InterruptedException {
        lock.lock();
        try {
            while (count == items.length)
                notFull.await();
            items[tail] = x;
            if (++tail == items.length)
                tail = 0;
            ++count;
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }

    // BLOCKS-UNTIL: notEmpty
    public T take() throws InterruptedException {
        lock.lock();
        try {
            while (count == 0)
                notEmpty.await();
            T x = items[head];
            items[head] = null;
            if (++head == items.length)
                head = 0;
            --count;
            notFull.signal();
            return x;
        } finally {
            lock.unlock();
        }
    }
}
```

Listing 14.11. Bounded buffer using explicit condition variables.

```
// Not really how java.util.concurrent.Semaphore is implemented
@ThreadSafe
public class SemaphoreOnLock {
    private final Lock lock = new ReentrantLock();
    // CONDITION PREDICATE: permitsAvailable (permits > 0)
    private final Condition permitsAvailable = lock.newCondition();
    @GuardedBy("lock") private int permits;

    SemaphoreOnLock(int initialPermits) {
        lock.lock();
        try {
            permits = initialPermits;
        } finally {
            lock.unlock();
        }
    }

    // BLOCKS-UNTIL: permitsAvailable
    public void acquire() throws InterruptedException {
        lock.lock();
        try {
            while (permits <= 0)
                permitsAvailable.await();
            --permits;
        } finally {
            lock.unlock();
        }
    }

    public void release() {
        lock.lock();
        try {
            ++permits;
            permitsAvailable.signal();
        } finally {
            lock.unlock();
        }
    }
}
```

Listing 14.12. Counting semaphore implemented using Lock.

AQS handles many of the details of implementing a synchronizer, such as FIFO queuing of waiting threads. Individual synchronizers can define flexible criteria for whether a thread should be allowed to pass or be required to wait.

Using AQS to build synchronizers offers several benefits. Not only does it substantially reduce the implementation effort, but you also needn't pay for multiple points of contention, as you would when constructing one synchronizer on top of another. In `SemaphoreOnLock`, acquiring a permit has two places where it might block—once at the lock guarding the semaphore state, and then again if a permit is not available. Synchronizers built with AQS have only one point where they might block, reducing context-switch overhead and improving throughput. AQS was designed for scalability, and all the synchronizers in `java.util.concurrent` that are built with AQS benefit from this.

14.5 `AbstractQueuedSynchronizer`

Most developers will probably never use AQS directly; the standard set of synchronizers covers a fairly wide range of situations. But seeing how the standard synchronizers are implemented can help clarify how they work.

The basic operations that an AQS-based synchronizer performs are some variants of *acquire* and *release*. Acquisition is the state-dependent operation and can always block. With a lock or semaphore, the meaning of acquire is straightforward—acquire the lock or a permit—and the caller may have to wait until the synchronizer is in a state where that can happen. With `CountDownLatch`, acquire means "wait until the latch has reached its terminal state", and with `FutureTask`, it means "wait until the task has completed". Release is not a blocking operation; a release may allow threads blocked in acquire to proceed.

For a class to be state-dependent, it must have some state. AQS takes on the task of managing some of the state for the synchronizer class: it manages a single integer of state information that can be manipulated through the protected `getState`, `setState`, and `compareAndSetState` methods. This can be used to represent arbitrary state; for example, `ReentrantLock` uses it to represent the count of times the owning thread has acquired the lock, `Semaphore` uses it to represent the number of permits remaining, and `FutureTask` uses it to represent the state of the task (not yet started, running, completed, cancelled). Synchronizers can also manage additional state variables themselves; for example, `ReentrantLock` keeps track of the current lock owner so it can distinguish between reentrant and contended lock-acquisition requests.

Acquisition and release in AQS take the forms shown in Listing 14.13. Depending on the synchronizer, acquisition might be *exclusive*, as with `ReentrantLock`, or *nonexclusive*, as with `Semaphore` and `CountDownLatch`. An acquire operation has two parts. First, the synchronizer decides whether the current state permits acquisition; if so, the thread is allowed to proceed, and if not, the acquire blocks or fails. This decision is determined by the synchronizer semantics; for example, acquiring a lock can succeed if the lock is unheld, and acquiring a latch can succeed if the latch is in its terminal state.