State-dependent classes have operations with state-based preconditions (**FutureTask**, **BlockingQueue**)

Implement new snchronizers using

- intrinsic **condition queue mechanism**
- explicit **Condition** objects
- **AbstractQueuedSynchronizer** framework

Managing state dependence

In a single-threaded program if a precondition is not met it will never become true, therefore, they can be coded to fail if preconditions are not met.
In a concurrent program, such conditions may change because of other thread's actions, and state-dependent methods can wait for preconditions to become true.

The built-in **condition queue mechanism** enables threads

- to block until an object has entered a state that allows progress
- to wake blocked threads when they may be able to make further progress

Alternative (worst) approaches

- throwing exceptions/returning error —- moves burden of retrying on the client, unables ordering
  * sleeping —- can oversleep
  * spin wait —- consumes many CPU cycles
  * **Thread.yield** —- platform-dependent, still possible to oversleep or consume too much CPU
- Crude blocking by polling and sleeping
  * complex implementation
  * have to choose between responsiveness and CPU consumption (timeout)
  * have to deal with **InterruptedException**

A **condition queue** gives a wait set (a group of threads) a way to wait for a specific condition to become true. Elements of **contidion** queues are threads.
Each object can act as a queue

- **wait**, **notify**, **notifyAll** methods
- should be called holding the lock on the object (preserving state consistency: need to wait on the condition (reading) and to change the condition (writing))
- **wait** atomically releases the lock, reacquires upon waking

Advantages:

- CPU efficiency
- context-switch overhead
- responsiveness

Using condition queues

The condition predicate

The condition predicate is the precondition that makes an operation state dependent.

Condition predicate

- involves state variables guarded by a lock
- before testing the condition predicate we must acquire that lock
- that lock object must be the condition queue object

If the precondition is not true **wait** releases the lock, puts thread in a waiting queue, reactuires the lock after some of the following events occured:

- thread is interrupted
- timeout has passed
- thread is notified
- spurious wakeup

Waking up too soon

When a thread is waked up and has reacquired the lock, the condition may not be true:

- other waiting or unrelated threads may have already modified the state
- other threads may wait for another predicate and that notification may not be related to your precondition

The rules

- always check the precondition again
- check the precondition in a loop

Missed signals

Failing to check a precondition before waiting may lead to a “lost signal”. If that happens, your thread may potentially wait indefinitely.

Notifications

Make notification whenever the condition predicate becomes true. Hold the lock associated with the object and invoke notify method

- **notify** —- selects single thread on the condition queue to wake up
- **notifyAll** —- wakes up all the threads waiting on that condition queue

notifyAll vs **notify** problem:

- **notify** may be dangerous. **notifyAll** must be preferred to **notify** in most cases except when both 2 statements are true:

uniform waiters —- only one condition predicate is waited with the condition queue, each thread executes the same logic upon returning from wait (or else **signal hijacking**)
one-in, one-out —- a notification enables at most one thread to proceed (**liveness problem**)

- **notifyAll** causes each thread to wakeup and causes many context switches, O (n^2) in the worst case

Conditional notification optimization —- notifying only when a state transition occured.

Subclass safety issues

A state-dependent class should follow one of the following guidelines

- Effectively prohibit inheritance
 making class final
 hiding the condition queues, locks, state variables from subclasses
 * subclasses may fail using **notify** due to **hijecked signals**!
- Fully expose and document its waiting and notification protocols to subclasses
- Make it possible for subclasses to write code that repairs probable damage
- Encapsulate the condition queue
 otherwise impossible to force the **uniform waiters** requirement, inherited code may mistakenly wait on this queue which may cause **hijacked signals**
 this is inconsistent with using intrinsic locking to guard the state, will stop supporting any kind of client-side locking

Entry and exit protocols

For each state-dependent operation and for each operation that modifies the 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 may have caused some other condition predicates to become true, and if so, notifying on the associated condition queue

AbstractQueuedSynchronizer exploits the concept of exit protocol.

Explicit Condition objects

Condition is a generalization of intrinsic condition queues.

Disadvantage of intrinsic condition queues
Only one queue associated with intrinsic lock:

- typical **notify** vs **notifyAll** problem (see above)
 notifyAll —- performance considerations
 notify —- policy

* uniformed waiters (or else hijacking)
    * one-in, one-out (or else liveness problems)
- the most common pattern for locking involves exposing the condition queue object (owning the reference or inheriting) —- the previous problem with inheritance or using the reference

Condition associated with **Lock** (**Lock.newCondition**) offers a richer feature set than intrinsic locking

- multiple wait sets per lock
- interruptible and uninterruptible condition waits
- deadline-based waiting
- choice of fair and nonfair queueing (inherited from their associated **Lock**)

Condition methods

- await
  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
- signal
    * void signal()
    * void signalAll()

Anatomy of a synchronizer

ReentrantLock, **Semaphore**, **CountDownLatch**, **ReentrantReadWriteLock**, **SynchronousQueue**, **FutureTask** are implemented with **AbstractQueuedSynchronizer**.

Advantages of **AbstractQueuedSynchronizer**:

- have only one point where they might block, reducing context-switch overhead and improving throughput —- designed for scalability
- reduces the implementation effort

AbstractQueuedSynchronizer

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 lock
  Lock, **Semaphore**: **acquire** means **acquire the lock or a permit**
  CountDownLatch: waiting until the latch has reached its terminal state
  FutureTask: wait until the task has completed
- release is not a blocking operation, but it may allow threads blocked in acquire to proceed

AQS takes on the task of managing some of the state for synchronizer class: it manages a single integer of state information that can be manipulated through the protected

- **getState**
- **setState**
- **compareAndSetState**

AQS is used to represent states:

- **ReentrantLock** —- the count of times the owning thread has acquired the lock
- **Semaphore** —- the number of permits remaining
- **FutureTask** —- the state of the task (not yet started, running, completed, cancelled)

Acquisition has two parts

- the synchronizer decides whether the current state permits acquisition
  if so, the thread is allowed to proceed
  if not, the acquire fails or blocks
- possibly updating the synchronizer state —- one thread acquiring the sycnrhonizer affects whether other threads can acquire it

AQS usage

- Use your **Sync** class to implement synchronizer
  acquire, **release** for exclusive access
  acquireShared, **releaseShared** for shared access
- Implement your **Sync** subclassing **AQS**
  acquire, **release** implement exclusive access using methods implemented by a subclass: **tryAcquire**, **tryRelease**, **isHeldExclusively**
  acquireShared, **releaseShared** implement shared access using methods implemented by a subclass: **tryAcquireShared**, **tryReleaseShared**
  use **getState**, **setState** and **compareAndSetState** to examine and update the state according to its acquire and release semantics

tryAcquireShared returns:

- negative value —- acquisition failure
- zero —- exclusive acquisition
- positive value —- non-exclusive acquisition

tryRelease and **tryReleaseShared**:

- true —- the release may have unblocked some threads attempting to acquire the synchronizer
- false otherwise