1. **wait()** : It tells the calling thread to give up the lock and go to sleep until some other thread enters the same monitor and calls notify(). The wait()method releases the lock prior to waiting and reacquires the lock prior to returning from the wait() method. The wait() method is actually tightly integrated with the synchronization lock, using a feature not available directly from the synchronization mechanism. In other words, it is not possible for us to implement the wait() method purely in Java: **it is a native method**.

General syntax for calling wait() method is like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | synchronized( lockObject )  {      while( ! condition )      {          lockObject.wait();      }        //take the action here;  } |

1. **notify()** : It wakes up the first thread that called wait() on the same object. It should be noted that calling notify() does not actually give up a lock on a resource. It tells a waiting thread that that thread can wake up. However, the lock is not actually given up until the notifier’s synchronized block has completed. So, if a notifier calls notify() on a resource but the notifier still needs to perform 10 seconds of actions on the resource within its synchronized block, the thread that had been waiting will need to wait at least another additional 10 seconds for the notifier to release the lock on the object, even though notify() had been called.

General syntax for calling notify() method is like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | synchronized(lockObject)  {      //establish\_the\_condition;        lockObject.notify();        //any additional code if needed  } |

1. **notifyAll()** : It wakes up all the threads that called wait() on the same object. The highest priority thread will run first in most of the situation, though not guaranteed. Other things are same as notify() method above.

General syntax for calling notify() method is like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | synchronized(lockObject)  {      establish\_the\_condition;        lockObject.notifyAll();  } |

How to Use with wait(), notify() and notifyAll() Methods

In this exercise, we will solve **producer consumer problem** using wait() and notify() methods. To keep program simple and to keep focus on usage ofwait() and notify() methods, we will involve only one producer and one consumer thread.

#### What happens when notify() is called and no thread is waiting?

In general practice, this will not be the case in most scenarios if these methods are used correctly. Though if the notify() method is called when no other thread is waiting, notify() simply returns and the notification is lost.

Since the wait-and-notify mechanism does not know the condition about which it is sending notification, it assumes that a notification goes unheard if no thread is waiting. A thread that later executes the wait() method has to wait for another notification to occur.

#### Can there be a race condition during the period that the wait() method releases OR reacquires the lock?

The wait() method is tightly integrated with the lock mechanism. The object lock is not actually freed until the waiting thread is already in a state in which it can receive notifications. It means only when thread state is changed such that it is able to receive notifications, lock is held. The system prevents any race conditions from occurring in this mechanism.

Similarly, system ensures that lock should be held by object completely before moving the thread out of waiting state.

#### If a thread receives a notification, is it guaranteed that the condition is set correctly?

Simply, no. Prior to calling the wait() method, a thread should always test the condition while holding the synchronization lock. Upon returning from the wait() method, the thread should always retest the condition to determine if it should wait again. This is because another thread can also test the condition and determine that a wait is not necessary — processing the valid data that was set by the notification thread.

This is a common case when multiple threads are involved in the notifications. More particularly, the threads that are processing the data can be thought of as consumers; they consume the data produced by other threads. There is no guarantee that when a consumer receives a notification that it has not been processed by another consumer. As such, when a consumer wakes up, it cannot assume that the state it was waiting for is still valid. It may have been valid in the past, but the state may have been changed after the notify() method was called and before the consumer thread woke up. Waiting threads must provide the option to check the state and to return back to a waiting state in case the notification has already been handled. This is why we always put calls to the wait() method in a loop.

#### What happens when more than one thread is waiting for notification? Which threads actually get the notification when the notify() method is called?

It depends on many factors.Java specification doesn’t define which thread gets notified. In runtime, which thread actually receives the notification varies based on several factors, including the implementation of the Java virtual machine and scheduling and timing issues during the execution of the program. There is no way to determine, even on a single processor platform, which of multiple threads receives the notification.

Just like the notify() method, the notifyAll() method does not allow us to decide which thread gets the notification: they all get notified. When all the threads receive the notification, it is possible to work out a mechanism for the threads to choose among themselves which thread should continue and which thread(s) should call the wait() method again.

#### Does the notifyAll() method really wake up all the threads?

Yes and no. All of the waiting threads wake up, but they still have to reacquire the object lock. So the threads do not run in parallel: they must each wait for the object lock to be freed. Thus, only one thread can run at a time, and only after the thread that called the notifyAll() method releases its lock.

#### Why would you want to wake up all of the threads if only one is going to execute at all?

There are a few reasons. For example, there might be more than one condition to wait for. Since we cannot control which thread gets the notification, it is entirely possible that a notification wakes up a thread that is waiting for an entirely different condition. By waking up all the threads, we can design the program so that the threads decide among themselves which thread should execute next. Another option could be when producers generate data that can satisfy more than one consumer. Since it may be difficult to determine how many consumers can be satisfied with the notification, an option is to notify them all, allowing the consumers to sort it out among themselves.