Return to Module Overview Page (https://onlinelearning.berkeley.edu/courses/665040/pages/module-seven)

7.7: Condition Variables

Java provides a special mechanism to transparently deal with the fact that sometimes a thread will acquire the monitor for an object that's not quite ready for processing. Imagine a program in which several threads were creating string objects (input from several networked users, perhaps) and putting them into a well-known variable inside a specific object. Other threads in the same program are removing the strings and processing them, setting the variable to null as soon as a thread accepts a string. In this program, it is clearly important that the input and output threads alternate. But in a real program, the input will arrive at random times, and the output may take varying lengths of time to process. How can we arrange things so the threads alternate, yet still allow input to arrive in random order and deal with the varying times output processing may take?

Every Java object inherits the methods wait( ) and notify( ) from java.lang.Object. These methods communicate directly with the underlying thread mechanism, and perform a bit of magic that let an object with synchronized methods manage exactly this sort of situation. An example will help:

```
class DataHolder
{
    String data = null;
    synchronized void insert(String s)
    {
      try {
       while (data != null)
       {
          // data already holds a string
          // wait for some other Thread
          //to remove it!
          // awaken one other waiting Thread
          notify();
          // release the monitor
          wait();
       }
      } catch (InterruptedException e) {}
      // data is now null
      data = s;
      notify();
   }

  synchronized String extract()
  {
    try {
     while (data == null)
     {
       // no data to extract
       // wait for some to arrive
       // awaken one other waiting Thread
       notify();
       // release the monitor and go to sleep
```

```
        wait();
      }
      } catch (InterruptedException e) {}

      // data is now not full
      String temp = data;
      data = null;
      notify();
      return temp;
      }
      // more methods
}
```

In this example, some threads are calling insert( ), and some extract( ). It will quite frequently happen that a thread enters insert( ) to find that the variable data is already holding a string. Instead of returning with an error, and forcing the caller of insert( ) to try again later, Java provides an automated mechanism to do effectively the same thing. By calling wait( ), a thread announces that it wants to release the monitor for the object, but have a chance to try again at some later time. Java releases the monitor and puts the thread to sleep, *atomically*. (This means the two actions are guaranteed to happen without being interrupted by another thread.) wait( )ing threads are placed in a separate queue, in which they will stay forever until awoken by a call to notify( ) on the same object. notify( ) wakes up one thread that is wait( )ing on that object (the one that has been wait( )ing the longest). Note that once a thread is awoken, it still is in a queue; it does not get the monitor right away.

The following is an example of the execution flow of our DataHolder class, with one thread attempting to insert (Thread A) and another attempting to extract (Thread B):

Table: Example of Execution Flow

Time	Thread A	Thread B	Data	Wait Que
		started	null	empty
		extract ( )	null	empty
		check if data is still null	null	empty
		since data is null, put in wait state	null	empty
		notify ( )	null	empty
		wait ( )	null	empty
	started	waiting	null	Thread B
	insert ( )		null	
	check if data is null, set it to the give value (hello)		null	
	notify ( )	removed from wait queue by notify ( ) call from thread A	hello	empty

	done	set a temp variable to data	hello	empty
		set data to null	hello	empty
		notify ( )	null	empty
		return hello	null	empty
		done	null	empty

Sometimes you need a reference to the currently running Thread object. You can get it using the static method Thread.currentThread( ). If you've subclassed Thread, you can just use this, of course.