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

7.6: Synchronization

Alas, nothing so apparently simple is ever without a few complications. The most important complication here is keeping multiple running threads from interrupting each other in the middle of something important.

Imagine that you're at a fine restaurant with three of your dearest friends, enjoying an exquisite meal. The lobby of the restaurant is filled with prospective diners waiting to be seated. At one point, one of your party feels the need to make a phone call and excuses herself from the table. How would you feel if an overeager lobby denizen seated himself in your friend's place and began gobbling up the remainder of her dinner? You'd be quite upset. (At least, I would be.)

Fortunately, in real life this never happens, because we humans have an unwritten understanding: as soon as the first guest is seated at a restaurant table, until the dirty dishes have been cleared away the entire table is reserved for the exclusive use of the seated party. Java has a special keyword that lets you express exactly this notion: the synchronized keyword.

```
public class RestaurantTable
{
   public synchronized void eating()
    {
       // Implementation details omitted...
    }
}
```

Marking a method "synchronized" tells the Java system that only one thread should be allowed to enter this method at a time. If a thread is in the middle of eating( ), and another thread calls eating( ), the second thread blocks (it effectively goes to sleep). When the first thread is through with eating( ), then and only then is the second thread awakened and allowed to begin eating( ).

Eating is not a common programming task, but many real functions would malfunction just as badly if interrupted. Examples include inserting elements into a queue (this might involve three or four statements, during which the queue is in an undefined state); growing or shrinking an array by allocating a new one and copying the old one; and saving a file safely by creating a new file, renaming the old one, renaming the new one, and then deleting the old one. Any multistep process whose earlier steps are assumed to affect its later steps is a candidate for synchronization.

Actually, the rule is that only one thread can be inside of any synchronized method of a given object at a time. Each object that is an instance of a class with synchronized methods has a data structure called a *monitor* associated with it. A monitor is just like the baton in a relay race: only the thread holding the monitor is allowed to run in a synchronized method, and only one thread can hold the monitor for a given object at a time.

This system is rather elegant. From the point of view of an individual thread interested in eating( ), all it has do is call the appropriate method. Acquiring the monitor, or being suspended and waiting for it, is completely transparent; the thread's code does not need to worry about the potential problem. This notion is important to understand as we head into the final section of this commentary. The client code (code that calls the synchronized method) can be very simple because it does not have to deal with synchronization issues. Threads that attempt to call a synchronized method while the object is busy are placed in a queue, where they lie dormant until their turn arrives.