Mutual Exclusion with "synchronized"[i]

It's pretty easy to program several threads to carry out completely independent tasks. The real difficulty arises when threads have to interact in some way. One way that threads interact is by sharing resources. When two threads need access to the same resource, such as a variable or a window on the screen, some care must be taken that they don'ttry to use the same resource at the same time. Otherwise, the situation could be something like this: Imagine several cooks sharing the use of just one measuring cup, and imagine that Cook A fills the measuring cup with milk, only to have Cook B grab the cup before Cook A has a chance to empty the milk into his bowl. There has to be some way for Cook A to claim exclusive rights to the cup while he performs the two operations: Add-Milk-To-Cup and Empty-Cup-Into-Bowl.

Something similar happens with threads, even with something as simple as adding one to a counter. The statement

```
count = count + 1;
```

is actually a sequence of three operations:

```
Step 1. Get the value of count
Step 2. Add 1 to the value.
Step 3. Store the new value in count
```

Suppose that several threads perform these three steps. Remember that it's possible for two threads to run at the same time, and even if there is only one processor, it's possible for that processor to switch from one thread to another at any point. Suppose that while one thread is between Step 2 and Step 3, another thread starts executing the same sequence of steps. Since the first thread has not yet stored the new value in count, the second thread reads the old value of count and adds one to that old value. Both threads have computed the same new value for count, and both threads then go on to store that value back into count by executing Step 3. After both threads have done so, the value of count has gone up only by 1 instead of by 2! This type of problem is called a race condition. This occurs when one thread is in the middle of a multi-step operation, and another thread can change some value or condition that the first thread is depending upon. (The first thread is "in a race" to complete all the steps before it is interrupted by another thread.)

Another example of a race condition can occur in an if statement. Consider the following statement, which is meant to avoid a division-by-zero error:

```
if ( A != 0 )
      B = C / A;
```

Suppose that this statement is executed by some thread. If the variable A is shared by one or more other threads, and if nothing is done to guard against the race condition, then it is possible that one of those other threads will change the value of A to zero between the time that the first thread checks the condition A != 0and the time that it does the division. This means that the thread can end up dividing by zero, even though it just checked that A was not zero!

To fix the problem of race conditions, there has to be some way for a thread to get exclusive Access to a shared resource. This is not a trivial thing to implement, but Java provides a high-level and relatively easy-to-use approach to exclusive access. It's done with synchronized methods and with the synchronized statement. These are used to protect shared resources by making sure that only one thread at a time will try to access the resource. Synchronization in Java actually provides only mutual exclusion, which means that exclusive access to a resource is only guaranteed if every thread that needs access to that resource uses synchronization. Synchronization is like a cook leaving a note that says, "I'm using the measuring cup." This will get the cook exclusive access to the cup—but only if all the cooks agree to check the note before trying to grab the cup.

---

[i] Adapted from "*Introduction to Programming Using Java*", David J. Eck, Hobart and William Smith Colleges