#### 8.3.1.4 volatile Fields

As described in §17, the Java language allows threads that access shared variables to keep private working copies of the variables; this allows a more efficient implementation of multiple threads. These working copies need be reconciled with the master copies in the shared main memory only at prescribed synchronization points, namely when objects are locked or unlocked. As a rule, to ensure that shared variables are consistently and reliably updated, a thread should ensure that it has exclusive use of such variables by obtaining a lock that, conventionally, enforces mutual exclusion for those shared variables.

Java provides a second mechanism that is more convenient for some purposes: a field may be declared `volatile`, in which case a thread must reconcile its working copy of the field with the master copy every time it accesses the variable. Moreover, operations on the master copies of one or more volatile variables on behalf of a thread are performed by the main memory in exactly the order that the thread requested.

If, in the following example, one thread repeatedly calls the method `one` (but no more than `Integer.MAX_VALUE` (§20.7.2) times in all), and another thread repeatedly calls the method `two`:

```
class Test {

     static int i = 0, j = 0;


     static void one() { i++; j++; }

     static void two() {
          System.out.println("i=" + i + " j=" + j);
     }

}
```

then method `two` could occasionally print a value for `j` that is greater than the value of `i`, because the example includes no synchronization and, under the rules explained in §17, the shared values of `i` and `j` might be updated out of order.

One way to prevent this out-or-order behavior would be to declare methods `one` and `two` to be `synchronized` (§8.4.3.5):

```
class Test {

     static int i = 0, j = 0;


     static synchronized void one() { i++; j++; }

     static synchronized void two() {
          System.out.println("i=" + i + " j=" + j);
     }

}
```

This prevents method one and method two from being executed concurrently, and furthermore guarantees that the shared values of i and j are both updated before method one returns. Therefore method two never observes a value for j greater than that for i; indeed, it always observes the same value for i and j.

Another approach would be to declare i and j to be volatile:

```
class Test {

     static volatile int i = 0, j = 0;


     static void one() { i++; j++; }

     static void two() {
          System.out.println("i=" + i + " j=" + j);
     }

}
```

This allows method one and method two to be executed concurrently, but guarantees that accesses to the shared values for i and j occur exactly as many times, and in exactly the same order, as they appear to occur during execution of the program text by each thread. Therefore, method two never observes a value for j greater than that for i, because each update to i must be reflected in the shared value for i before the update to j occurs. It is possible, however, that any given invocation of method two might observe a value for j that is much greater than the value observed for i, because method one might be executed many times between the moment when method two fetches the value of i and the moment when method two fetches the value of j.

See §17 for more discussion and examples.

A compile-time error occurs if a final variable is also declared volatile.