Introduction to *synchronized*

You know that an int variable x shared by two threads may be involved in a race condition because even the assignment statement

```
x= x + 1;
```

is not an atomic, indivisible, action —it is executed as a sequence of three steps: (1) load x into a register, (2) add 1 to the register; (3) store the register into x.

A *critical section* is a code segment that has to be executed as an atomic action. We now show how to make a critical section into an atomic action.

The synchronized statement

Each object in Java has a "lock", which a thread can obtain in order to keep other threads from using the object. Execution of a statement

```
synchronized(object) { … }
```

waits until no other thread has the lock on *object*; it then obtains the lock, executes the code { … }, and finally relinquishes the lock. That's all there is to it!

Synchronizing on a primitive value is not possible; it has to be an object. (i.e. a pointer to an object)

Example

First, we write class X, which contains variable x, as shown to the right. Variable x is to be shared by two threads. It is wrapped to make this possible.

```
public class X {
   public int x= 0;
}
```

Class FirstSync declares static variable var of type X. Object var contains the variable x to be shared. Method main starts threads T1 and T2 running.

Look at T1. Its method main contains the assignment that increments shared variable FirstSynx.var.x, but that assignment is synchronized on FirstSync.var; it looks like this:

```
synchronized(FirstSync.var) {
   FirstSync.var.x= FirstSync.var.x + 1;
}
```

When this statement is executed, T1 obtains the lock on FirstSync.var. Therefore the assignment is performed as an atomic action in that any other thread that attempts to execute a statement synchronized on FirstSync.var must wait until this one is finished.

Now look at thread T2. Its method run also has a statement that synchronizes on shared object FirstSync.var. This synchronized statement doubles shared variable x.

```
public class FirstSync {
   public static X var= new X();

   public static void main(String[] arg) {
      new T1().start();
      new T2().start();
   }
}

public class T1 extends Thread {
   public void run() {
      …
      synchronized(FirstSync.var) {
         FirstSync.var.x= FirstSync.var.x + 1;
      }
      …
   }}

public class T2 extends Thread {
   public void run() {
      …
      synchronized(FirstSync.var) {
         FirstSync.var.x= 2*FirstSync.var.x;
      }
      …
}}
```

Better organization

The organization of this program is not good in that in writing each thread, the programmer has to worry about whether statements should be synchronized or not. It would be better if the shared object itself could do all the synchronizing. This means that the threads would call methods in the shared object.

We show this on the next page.

To the right, we show the reorganized class X. We have removed comments to save space —the methods are simple enough to understand without them.

Field x is now private, providing more security. Function getX returns its value.

Function incr contains a synchronized statement, synchronized on this object itself. Remember, "this" evaluates to the pointer to the object in which it appears.

Look at method doubble —we call it *doubble* instead of *double* because *double* is a Java keyword. Method doubble shows a new feature: a *synchronized method*. It is simply syntactic sugar; the following two are equivalent:

```
public void m(…) {
   synchronized(this) {statements}
}

public synchronized void m(…) {statements}
```

The synchronized statement is more flexible, in that the whole method need not be synchronized. But if the whole method *will* be synchronized, use a synchronized method.

To the right, we show the modified classes T1 and T2. Instead of having synchronized statements, they call methods in the shared object.

```
public class X {
   private int x= 0;

   public int getX() {return x;}

   public void incr() {
      synchronized(this) {x= x + 1;}
   }

   public synchronized void doubble() {
      x= 2*x;
   }
}
```

```
public class Sync {
   public static X var= new X();

   public static void main(String[] arg) {
      new T1().start();
      new T2().start();
   }
}

class T1 extends Thread {
   public void run() {
      …
      FirstSync.var.incr();
      …
   }}

class T2 extends Thread {
   public void run() {
      …
      FirstSync.var.doubble();
      …
   }}
```