Warning: synchronize all access to an object

Suppose you have one statement that synchronizes on an object ob, using

```
synchronized(ob) { … }
```

Then, unless you know absolutely what you are doing, make sure that *every* access to ob is synchronized. We illustrate what may happen if you don't.

Consider the program to the right. Class S declares a static variable c of type Str. Method S.main creates instances of Threads Ta and T1 and starts them both.

Class Str has a field s. Method put(char) appends its parameter to s and prints the name of the current thread and s's value.

Classes Ta and T1 extend Thread. They set the name of the class and then synchronize on static variable S.c. They both append two values to S.c. In addition, Ta sleeps for a relatively long time between the two calls on S.c.put and calls S.c.put('?') again if sleep is interrupted.

Copy these classes into a DrJava or Eclipse project, compile, and execute several times. One of two possible outputs always happens, depending on whether Ta or T1 gets to synchronize first. The one that synchronizes first appends two values to S.c before the other can.

One possibility	**Second possibility**
Ta. s is: a	T1. s is: 1
Ta. s is: ab	T1. s is: 12
T1. s is: ab1	Ta. s is: 12a
T1. s is: ab12	Ta. s is: 12ab

```
public class S {
   public static Str c= new Str();
   public static void main(String... args) {
      Ta ta= new Ta();    T1 t1= new T1();
      ta.start();         t1.start();
   }}

class Str {
   String s= "";

   public void put(char c) {
      s= s + c;
      String n= Thread.currentThread().getName();
      System.out.println(n + ". s is: " + s);
   }}

class Ta extends Thread {
   @Override public void run() {
      setName("Ta");
      synchronized(S.c) {
         S.c.put('a');
         try {sleep(100);}
         catch (InterruptedException e) {S.c.put('?');}
         S.c.put('b');
      }
   }}

class T1 extends Thread {
   @Override public void run() {
      setName("T1");
      synchronized(S.c) {
         S.c.put('1');
         S.c.put('2');
      }
   }}
```

Eliminating one synchronization

Now change class T1 as shown to the right below. The only change is to *not* synchronize on object S.c.

```
class T1 extends Thread {
   @Override public void run() {
      setName("T1");
      S.c.put('1');
      S.c.put('2');
   }
}
```

Now, thread Ta synchronizing on S.c does not prevent thread T1 from accessing S.c! We got these outputs, among others:

One possibility	**Another possibility**
Ta. s is: a1	T1. s is: a
T1. s is: a1	Ta. s is: a1
T1. s is: a12	T1. s is: a12
Ta. s is: a12b	Ta. s is: a12b

Ta owns the object, but T1 is able to access it anyway. Change the sleep time to 1 and you get even more interesting output —run several times and see the differences.

The problem

Suppose you tell some people to use your front door to get into your house, but if it is locked, wait for the person inside to unlock it and come out before going in yourself. And, lock it when you go in. That's what "synchronized" does. But some friends may know that the back door is always open and go in whenever they want. These friends are threads that use the object without worrying about synchronization.

Allowing friends in the back door can increase efficiency. In some cases, accessing an object may safely go on while another thread is changing the object. But you have to know what you are doing to use this correctly.