Example of the use of the back door in a synchronized object

Class `C`, to the right, illustrates how a thread can use the back door to a synchronized object and change that object even when another thread is synchronized on it.

Look first at method `m`. Its body is synchronized on static variable `y`. Here is what `m` does.

1. Print a message.
2. Sleep for 3 seconds. This gives another thread time to get in the back door.
3. Print a message that it woke up.
4. Add 1 to `y`.
5. Print a message that it added 1 to `y`.

Now look at method `run`, a call of which takes place in a different thread from method `m`. Here is what `run` does —notice the two commented out lines around the assignment to `y`; we'll discuss them later.

1. Print a message.
2. Sleep for 1 second. This is to ensure that the call on method `m` will be executed before this method changes `y`.
3. Print a message that it woke up.
4. Set `y` to -100.
5. Print a message that it set `y` to -100.

Method `main` is called to start the program. It creates an instance of the class and stores it in `r`, creates an instance of `Thread` with `r` as argument, and call its method start. This results in a call on method `run`. Then method `m` is called. Methods `run` and `m`, are running simultaneously in two different threads.

Below to the left, we show what the program prints. Method `run` changed `y` even though method `m` had synchronized on `y` —`run` used the back door.

Now, look at the two commented lines surrounding the assignment `y = -100;` in method `run`. Uncomment those lines and the assignment to `y` becomes synchronized on `y`. Run the program again, and it prints the output shown in the box to the right below. You can see that method `run` had to wait until after method `m` completed its synchronized block before changing `y`.

```
/** This class illustrates that a thread can use the back
    door and change a variable even though another
    thread has synchronized on that variable. */
public class C implements Runnable {
    static Integer y= 2;

    public static void main(String args[]) {
        Runnable r= new C();
        new Thread(r).start();
        m();
        System.out.println("Ending. y is: " + y);
    }

    /** Add 1 to y, synchronizing on y to do it.
        Print a message when starting, after waking up,
        and after changing y. */
    public static void m() {
        synchronized (y) {
            System.out.println("m starting");
            try {
                Thread.sleep(3000);
            } catch (InterruptedException e) { }
            System.out.println("m woke up");
            y= y + 1;
            System.out.println("m added 1 to y.");
        }
    }

    /** Sleep for 1 second, then set y to -100.
        Print a message when starting, after waking up,
        and after changing y. */
    public @Override void run() {
        System.out.println("run starting");
        try {
            Thread.currentThread().sleep(1000);
        } catch (InterruptedException e) { }
        System.out.println("run woke up");
        // synchronized (y) {
        y= -100;
        // }
        System.out.println("run: y set to -100");
    }
}
```

We urge you to download this code from JavaHyperText and run this program yourself; it will help your understanding to do this.

Method run uses the back door

```
m starting; it's synchronized on y
run starting
run woke up
run: y set to -100
m woke up; it's synchronized on y
m added 1 to y. m releasing the lock
Ending. y is: -99
```

Method run synchronizes on y

```
m starting; it's synchronized on y
run starting
run woke up
m woke up; it's synchronized on y
m added 1 to y. m releasing the lock
run: y set to -100
Ending. y is: -100
```