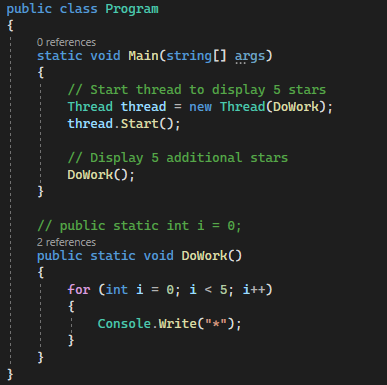
Each running thread gets its own private stack, so all local variables are kept strictly separate. Take a look at the following code:



It has a method called DoWork that outputs 5 stars in a row. I call this method from a separate thread and at the same time from the Main program thread. The loop variable I exists twice, one for each thread so that the threads will not interfere with each other. When I run the program, I get 10 stars as output just as expected.

Now, if you uncomment line of declaring static int and if you modify the loop to use this shared variable instead. So, now we have an interesting situation that both the created thread and the Main thread used the same variable I for their loop statement. What do you think the output is going to be?

The output is only 6 stars, and the reason for this is that both threads are incrementing the same variable. So, instead of counting from 0 to 5, the loops are actually counting from 0 to 5 in steps of two. And the result of this is that only 5 to 6 stars are printed instead of 10. This is a classic example of what is called a race condition, in which threads are fighting for the same variable and as a result, the code starts to behave in unpredictable ways.

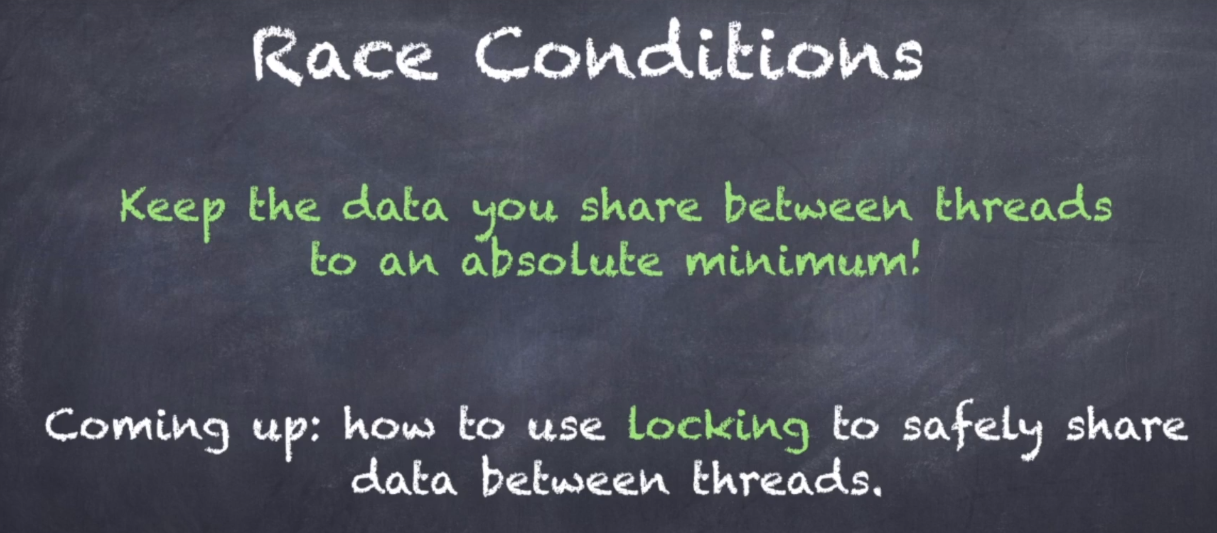
In this case, a simple for loop counting from 0 to 5 suddenly only iterates two or three times before finishing. The solution to race condition is a technique called locking.

What have we learned?



A race condition is when program execution no longer follows the predictable path and as a result code starts behaving in unexpected ways. Variables seemingly randomly change their values, for loops stop prematurely. Simple increment statements do nothing or increments by two instead of one. All of these things can happen.

Nine times out of ten, these race conditions occur between two or more threads are accessing the same variable. So, the main takeaway of this lecture is:



Keep the data you share between threads to an absolute minimum.