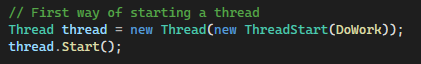
In this lecture, I am going to take a look at how to start a new thread by using the Thread class. A thread is an independent execution path, able to run simultaneously with other threads, a C# program starts in a single thread called automatically by the framework and operating system. This thread is called the Main program thread.



A program is made multithreaded by creating additional threads. The program starts in the static Main method, the program creates a new thread class constructor. The constructor expects a ThreadStart delegates as a parameter, and this is the method that the new thread will run. An important thing to realize is that the thread will only run this single method, when the method is completed, the thread will automatically end. And once ended the thread cannot restart.

In the example, the new thread executes the loop and simply displays the letter B a thousand times. But at the same time, the Main thread is also running, executing the loop and displaying the letter A thousand times. So, what output you expect to see?

Something like ABABAB. When you run the code, you will see that the As and Bs are clumped together in groups, and this is because threads are time sliced. The operating system runs a given thread for a while and then suspends it and runs a different thread. Each run interval is called a time slice, which is the maximum time a thread can run uninterrupted.

So, in the output, each time slice is visible as a group of identical letters. Modern computers have multicore processors that can actually run several threads at once, but at any given time, there are hundreds of active threads in the operating system, many more than the available number of CPU cores. So, there is always a certain amount of time slicing going on.

There are several ways to initialize a thread. You’ve seen the ThreadStart delegate for passing in the startup method into the thread constructor. However, we don’t need to explicitly specify the ThreadStart delegate, the C# compiler is smart enough to infer the delegate from the signature of the startup method itself.



Here, I pass the startup method directly to the thread constructor without specifying that it is a ThreadStart delegate. The compiler figures that out all by itself, which makes the code a lot cleaner. Another simplification is to remove the start of methods and replace it with a lambda expression like:

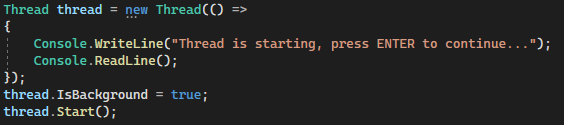


The entire thread startup method is now anonymous delegate. The compiler will figure out that the lambda expression matches the ThreadStart delegate and it will make everything work. Each thread has a name property that you can set. This is especially useful during debugging because the thread name is displayed in the thread window.

You can set a thread name just once. Attempting to change it later will throw an exception. Here is a program that sets up to 10 named threads. You can see the name of the threads in the Thread panel while debugging the program. And, when you double-click a thread, the debugger will show you which code is currently being executed by that particular thread.

Finally, let’s look at foreground and background threads. By default, any thread you create explicitly is a foreground thread. Foreground thread keep the application alive for as long as any one of them is running. Comparing this with background threads, once all foreground threads finish, the application ends and any background threads that are still running at this point will abruptly terminate.

You can query or change the thread’s background status by using the IsBackground property. This program starts a single thread that waits for the user to press enter. IsBackground property is set to true, which means this will be a background thread. While this background thread runs, the main thread continues executing the main method until it ends.



The program terminates and aborts the running background thread. When you will run this program, you will see that it immediately terminated. The background thread waiting for a key press has no effect at all. If you change the IsBackground to false that will create a foreground thread. I still expect the Main thread to end, but at that point, the foreground thread will still be active, waiting for the keypress.

And, because it is now a foreground thread, the program cannot end until I press enter, thereby ending the foreground thread and allowing the entire program to end. Now, when you run the program, you will see that it does not immediately terminate. It continues to run, even though the main thread has already exited the Main method.

And when you press enter the single remaining foreground thread can end and now the program also ends.

What have we learned?

