CMPE 1666- Intermediate Programming

Lecture 10
C# Multi-Threading

Acknowledgements JD Silver

Slow Operations – A Problem

- Traditional applications follow a sequential flow of control except when they encounter branching or looping.
- Consider the following application that runs 3 methods.

```
static void Main(string[] args)
        {
            FindSine();
            FindCos();
            displayHello();
        }
```

- Running the application in a traditional way means that each method runs to completion before the next one starts

Slow Operations – A Problem

- Some methods take a long time to execute, sometimes because of heavy computations, at other times because they block waiting for events (I/O, network operations)
- In a purely sequential execution, a slow method will delay the execution of all methods executing after it.
- Additionally, purely sequential applications cannot take advantage of multi-core (or multiple-processor) systems.

Slow Operations – A Problem

- Form-based programs appear to become unresponsive to the user when a slow operation is taking place.
- Professionally designed software will typically place slow code, or code which blocks, into a background thread.
- The form uses a thread and remains responsive.
- The feature of using multiple threads in an application is known as Multithreading.

Demos & Exercises

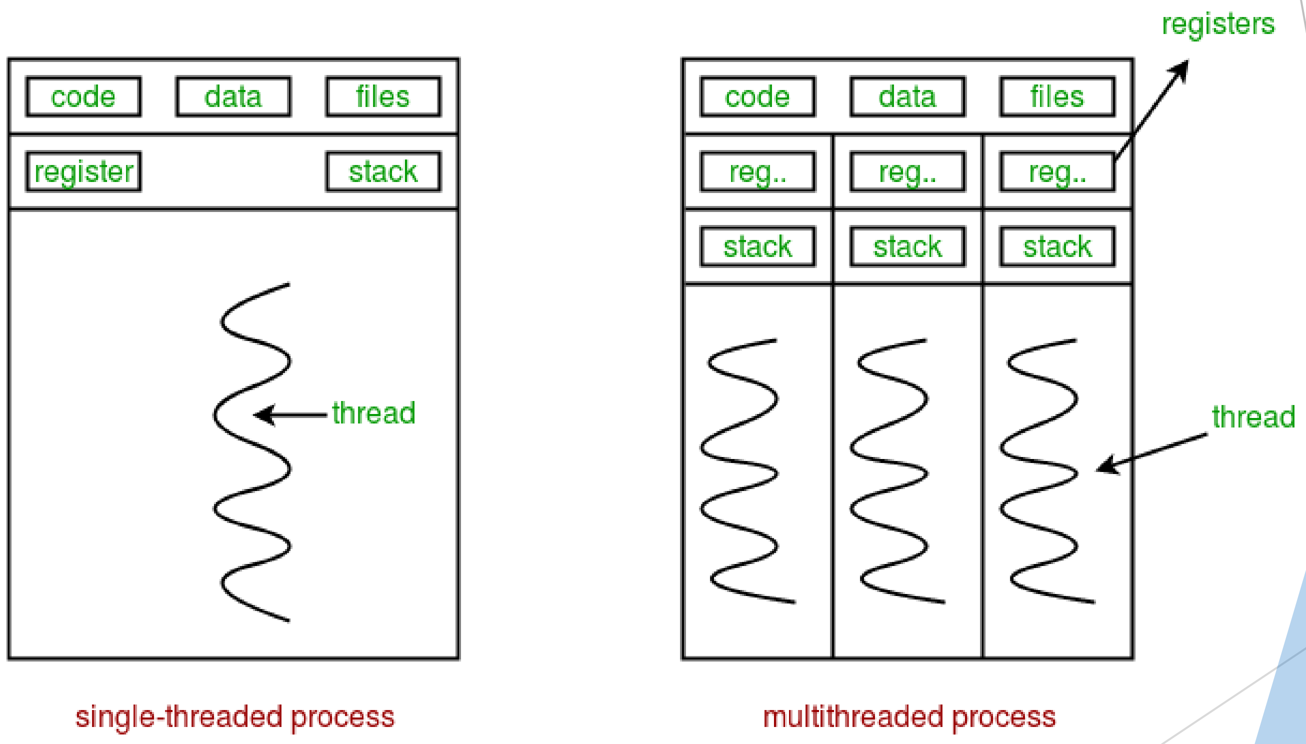
- To illustrate how Multithreading improves application performance, we are going to look at the following demos:
- Demo1 will run a purely sequential console application.
- Demo2 will convert the application to a multithreaded one.
- Demo3 will run a form-based application without multi-threading
- Exercise1 will add muti-threading to the application in Demo3

Lecture10-demo1

- You have been provided with a zip file (ThreadDemo1.zip). Unzip it and run the application.
- The given application is a console application executes 3 methods sequentially.
  - The first 2 methods have some heavy computation
  - All 3 methods have some blocking between subsequent iterations

Introduction To Multithreading

- Multithreading allows an application to have several threads of execution running concurrently or in parallel.



Source:https://cdncontribute.geeksforgeeks.org/wp-content/uploads/multithreading-python-21.png

Advantages of Multithreading

- On a single-core system, when a method blocks waiting for an event, another method, running on a different thread, can make use of the CPU
- On a multi-core system, multiple threads running different methods (or multiple instances of the same method) can be running on different processors at the same time.

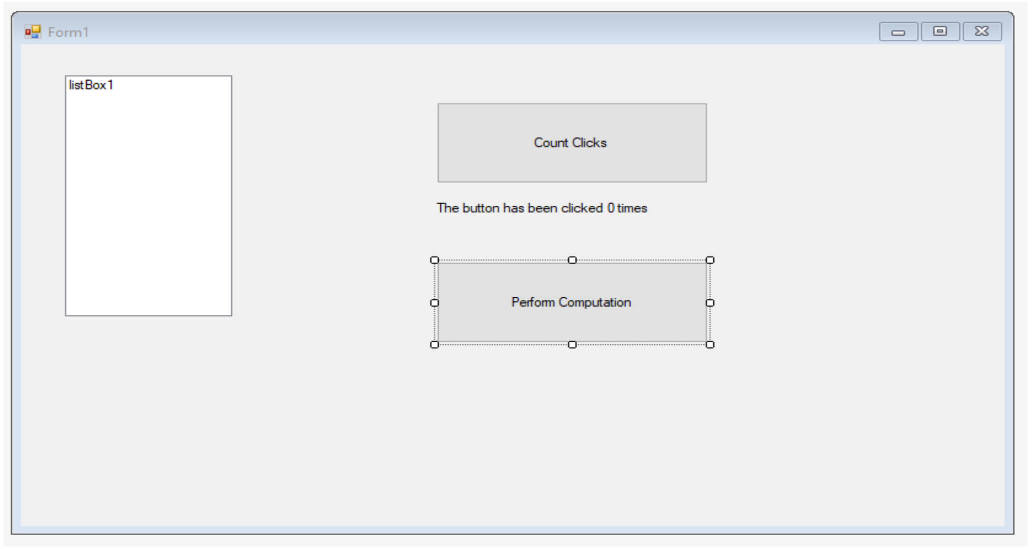
Lecture10-demo2

- Create a new console application called Lecture10-demo2
- Copy the code from program.cs in Lecture10-demo1 and paste it onto the same file in Lecture10-demo2
- Make the following modifications to the Program.cs file to make the application multithreaded:

1. Add the declaration of 3 Thread objects, initializing them to null, as follows:
   ```
   Thread Thread1=null, Thread2=null, Thread3=null;
   ```
2. Replace the call to the 3 methods by:
   ```
     Thread1= new Thread(FindSine);
     Thread2 = new Thread(FindCos);
     Thread3 = new Thread(displayHello);
   ```
3. Run the 3 threads through:
   ```
         Thread1.Start();
         Thread2.Start();
         Thread3.Start();
   ```

Lecture 10 –Demo3

- Create a form-based application with the controls shown below.



- Names the buttons **UI_Count_Btn** and **UI_Compute_Btn** respectively
- Leave the name of the list box unchanged
- Name the label UI_**Count_Lbl**
- Make a copy of the FindSine() method in the previous demos and paste it in the Form1.cs file.

(contd on next slide)

Lecture 10-demo3 (contd)

- Place the following member variable declaration in the Form1 class:

```
private int count=0;
```

- Add code to the event handlers as follows

```
private void UI_Count_btn_Click(object sender, EventArgs e)
 {
 Count_lbl.Text= $"Button has been clicked {++count} times ";
   }

private void UI_Compute_btn_Click(object sender, EventArgs e)
   {
      FindSine();
   }
```

```
Modify the FindSine() method so that the angles and their sine
values are added to the listbox instead of displaying on the
console
(contd on next slide)
```

Lecture 10-demo3 (contd)

- Run the application
- Click on the “Count Clicks” button a few times.
- Now click on the “Perform Computation” Button
- Try to Click on the “Count Clicks” Button and observe what happens
- You’ll note that when FindSine() is running, the form doesn’t respond.

Lecture 10- Exercise 1

- Create a similar application to Demo3. You can copy the code where appropriate.
- In the Form1 class declare a Thread variable and assign it to null.
- Modify the event handler for the "Perform Computation" Button, so that instead of calling the FindSine() method, we create a new Thread object to run it and assign it to the Thread variable. The event handler should then run the thread.
- Run the program using "**Start Without Debugging**"
- Now click on the "Count Clicks" button and observe the change.

Using Delegates to access Form controls

- Try to run the program using "**Start Debugging**"
- We note that trying to access the listBox from the newly created thread causes an exception.
- To avoid that, we make use of delegates.
- The Form class has an **Invoke()** method that can be used to invoke delegates.

Using Delegates to access Form controls

- The steps to use the delegate are:
  1. We need to create a method to access the control (such as adding items to the list box).

```
private void AddToListBox(string str){
    UI_Values_Lbx.Items.Add(str);
  }
```

  2. In the thread method (FineSine() in exercise1), we can create an action delegate.

```
Action<string> delWriteSine= AddToListBox;
```

  3. use the Invoke method. We pass a delegate object, as well as the parameters to the method as arguments to Invoke.

```
 Invoke(delWriteSine, str);
```

Using Delegates to access Form controls

```
public void FindSine()
 { Action<string> delWriteSine= AddToListBox;
  for (double x = 0; x < 90; x = x + 0.1)
   {
     double rad = Math.PI * x / 180;
     double sineValue = Math.Sin(rad);
     string str=$"{x:F2} degrees = {rad:F4} radians. Sine={sineValue:F4}";
     Invoke(delWriteSine, str);
    Thread.Sleep(50);
           }
        }
```

Exception Handling

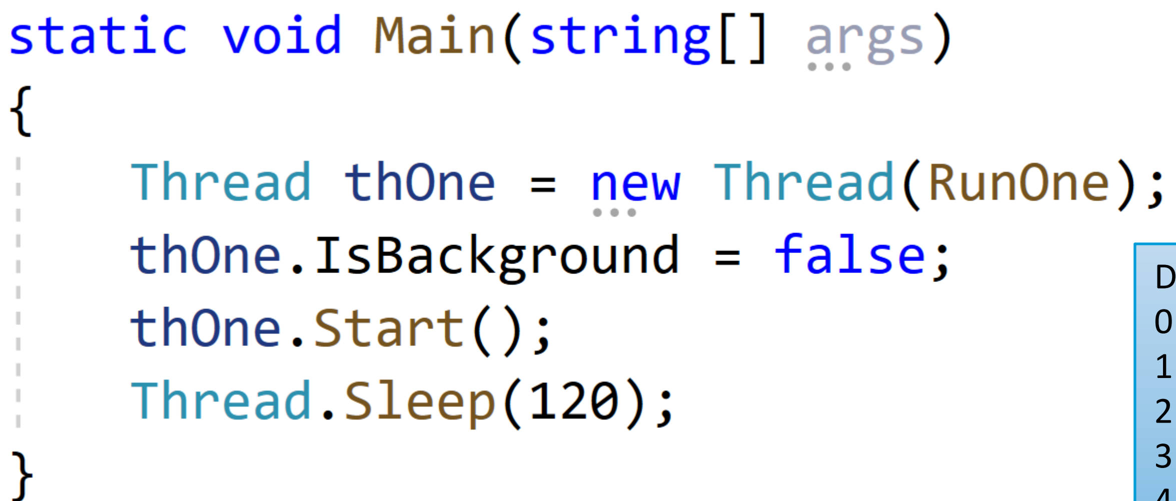
- To ensure that our program doesn't crash in case something goes wrong with our thread, we need to place the Invoke() in a **try……..catch** block

```
try{
        Invoke(delWriteSine,str);
  }
catch(exception e)
 {
          Console.WriteLine("Delegate could not be invoked");
}
```

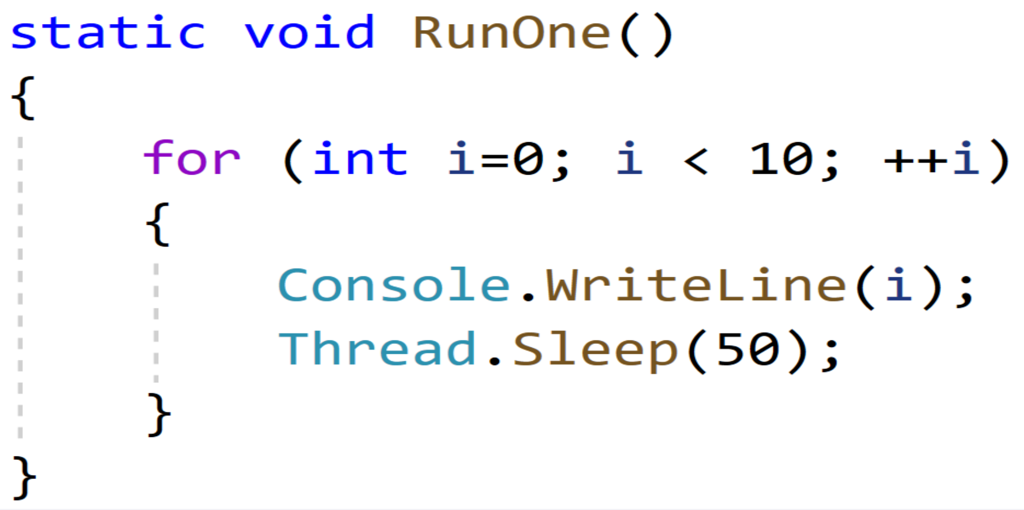
Background and Foreground Threads

- When a thread is created, it is always a **foreground** thread.
- The CLR will wait for all foreground threads to complete before unloading the AppDomain (the program).
- If the thread's **IsBackground** property is true, then the thread is a background thread and will be terminated when the program is done.

Foreground Thread



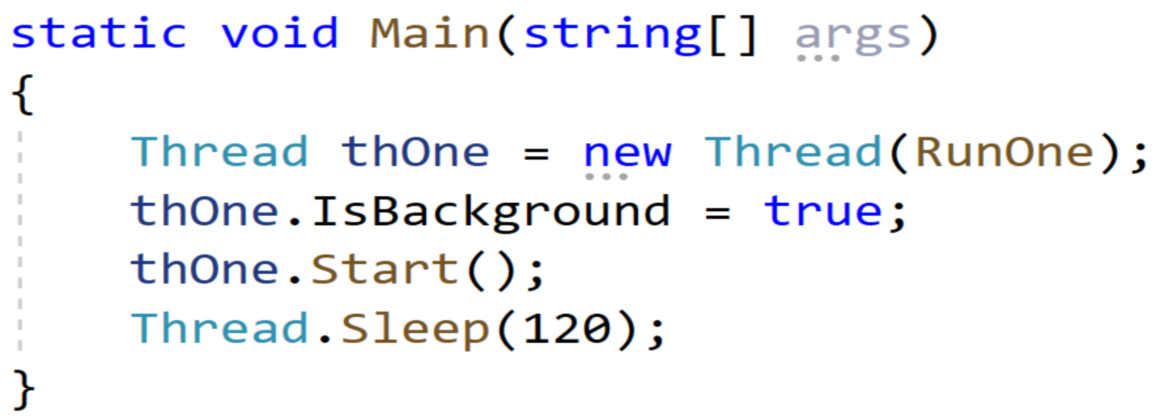
```
static void Main(string[] args)
{
    Thread thOne = new Thread(RunOne);
    thOne.IsBackground = false;
    thOne.Start();
    Thread.Sleep(120);
}
```



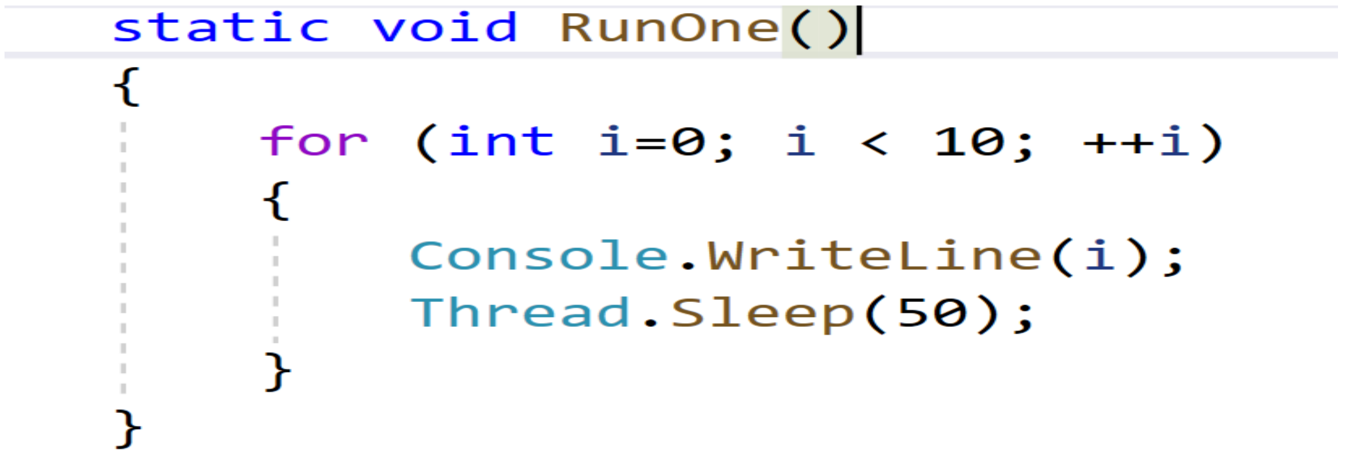
```
static void RunOne()
{
    for (int i=0; i < 10; ++i)
    {
        Console.WriteLine(i);
        Thread.Sleep(50);
    }
}
```

```
Displays…
0
1
2
3
4
5
6
7
8
9
Press any key to continue . . .
```

Background Thread



```
static void Main(string[] args)
{
    Thread thOne = new Thread(RunOne);
    thOne.IsBackground = true;
    thOne.Start();
    Thread.Sleep(120);
}
```



```
static void RunOne()
{
    for (int i=0; i < 10; ++i)
    {
        Console.WriteLine(i);
        Thread.Sleep(50);
    }
}
```

Terminating a Thread

- Some Times if a thread is taking too long, we may want to have a way to terminate it, even if the main thread continues to run.
- The easiest way to do this is to have a Boolean class variable that we initially set to true. We cause our thread method to execute only as long as the value of the variable is true
- We can then have another event, such as a button click that changes the variable to false.

Terminating a Thread

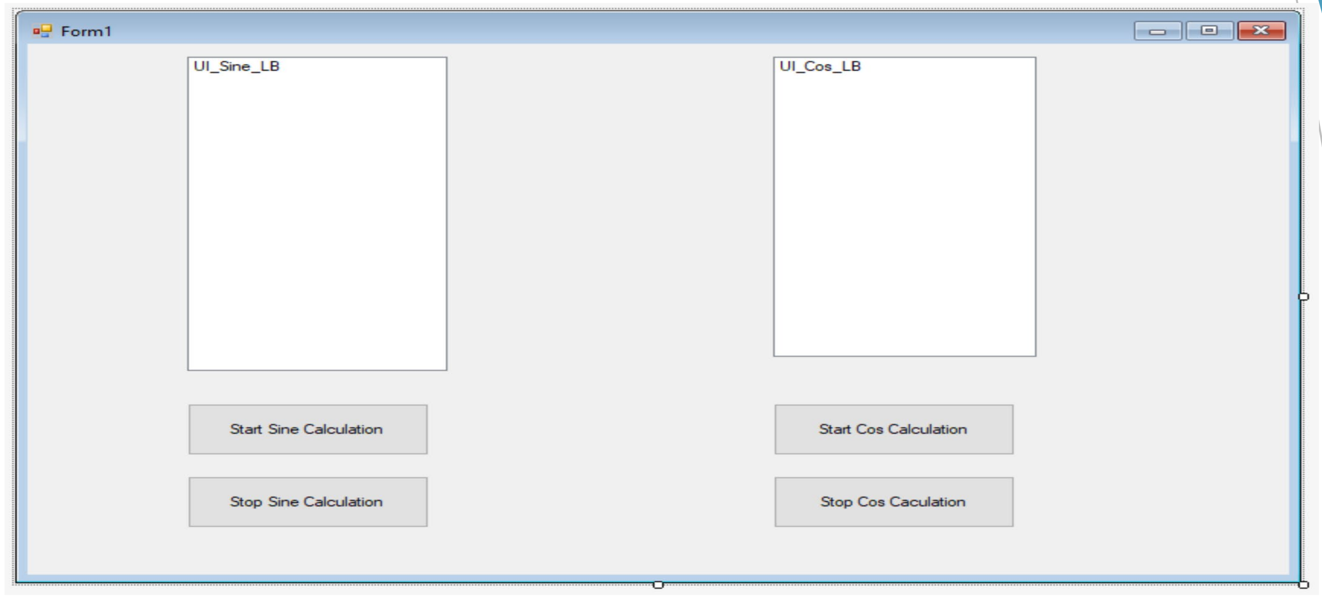
```
bool runThread = true; //Defined globally in the class

public void FindSine()
 { delVoidString delWriteSine= addToListBox;

 double x=0;
while ((x<90) && (runThread))
  {
   if (runThread)
    double rad = Math.PI * x / 180;
    double sineValue = Math.Sin(rad);
    string str=$"{x:F2} degrees = {rad:F4} radians. Sine={sineValue:F4}";
    try{
    Invoke(delWriteSine, str);
    }
   catch(Exception e){
          Console.WriteLine(“An Error Occurred”);
     }
   Thread.Sleep(50);
      X +=0.1;
         }
      }
```

Lecture 10- Exercise2

- You need to create a Form-based application with the controls shown below.

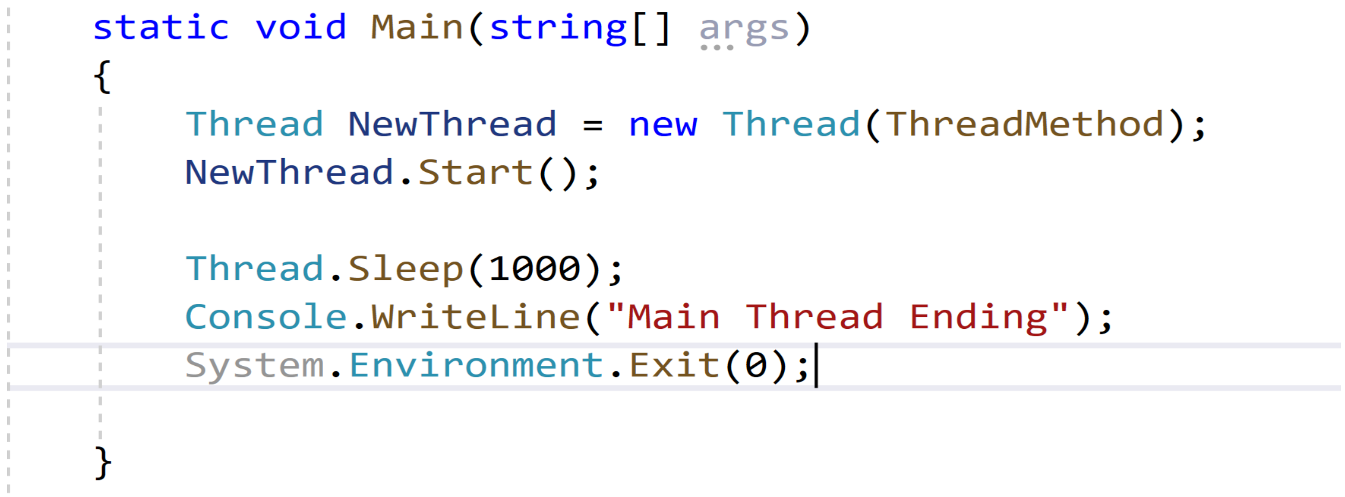


- You need to copy the FindSine() and FindCos() methods from demo1. Modify them to use a while loop instead of the for loop. Also include a class-level Boolean variable for each method, so that you can use it to stop the thread.
- The "Start Sine Calculation" and "Start Cos Calculation" buttons must each start a thread that respectively execute the FindSine() and FindCos() methods and add the values in the Sine and Cos list boxes respectively.
- Each of the stop buttons is to stop the corresponding Thread.

System.Environment.Exit

- When the Main program ends, using return or an implied return, the threads continue to run.
- If you wish to have the threads end with the Main program, use System.Environment.Exit().
- The argument is usually System.Environment.ExitCode or an integer.

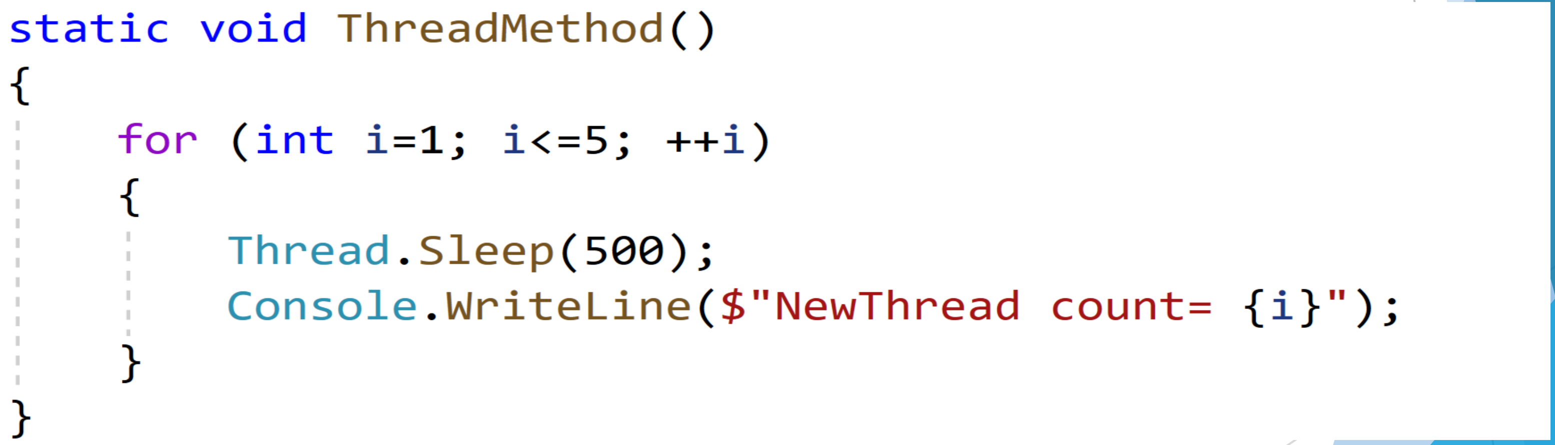
Lecture10-demo3



```
static void Main(string[] args)
{
    Thread NewThread = new Thread(ThreadMethod);
    NewThread.Start();

    Thread.Sleep(1000);
    Console.WriteLine("Main Thread Ending");
    System.Environment.Exit(0);

}
```



```
static void ThreadMethod()
{
    for (int i=1; i<=5; ++i)
    {
        Thread.Sleep(500);
        Console.WriteLine($"NewThread count= {i}");
    }
}
```

System.Environment.Exit

- When the Main program ended, the thread NewThread immediately terminated, without completing the for loop.

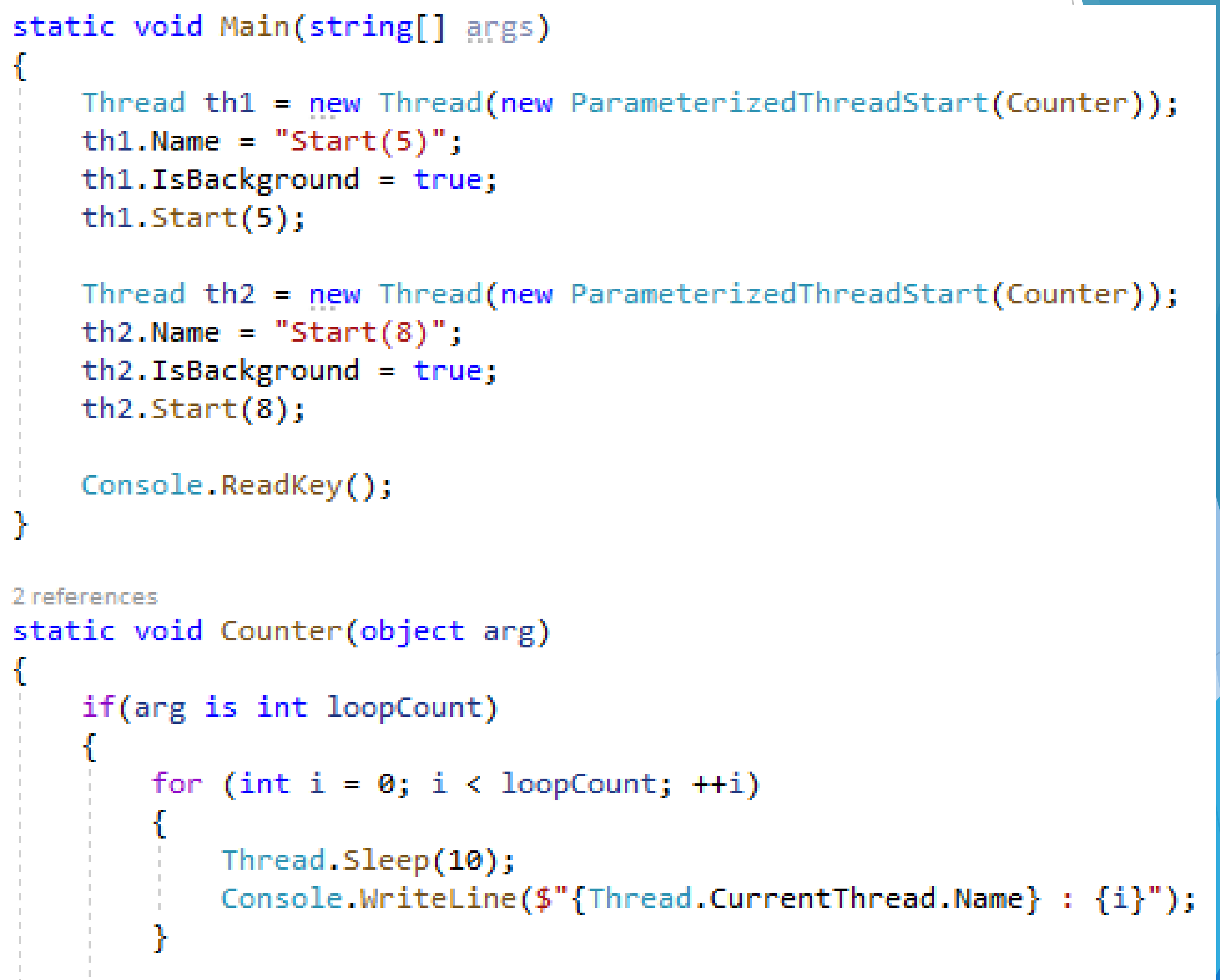
Using ThreadStart delegate

- The way we have created our Threads so far, we have implicitly made use of the ThreadStart delegate.
- **Eg. Thread thread1=new Thread(method1);**
- We can also explicitly use the delegate
- **Eg. Thread thread1=new Thread(new ThreadStart(method1));**

ParameterizedThreadStart

- Threads created by making (implicit or explicit) use of the **ThreadStart** delegate cannot have parameters.
- So we cannot pass data to the thread.
- Data can be passed to a thread as type **object** by using the **ParameterizedThreadStart** delegate.
- The data to be passed to the thread is provided as an argument to the **Start()** method.
- The data will be **boxed** to an **object** type.
- The data will be **unboxed** in the thread's method.
- If you wish to pass more than one value to a thread, create a **struct** or class to **hold** the data

Lecture10-demo4



```
static void Main(string[] args)
{
    Thread th1 = new Thread(new ParameterizedThreadStart(Counter));
    th1.Name = "Start(5)";
    th1.IsBackground = true;
    th1.Start(5);

    Thread th2 = new Thread(new ParameterizedThreadStart(Counter));
    th2.Name = "Start(8)";
    th2.IsBackground = true;
    th2.Start(8);

    Console.ReadKey();
}

2 references
static void Counter(object arg)
{
    if(arg is int loopCount)
    {
        for (int i = 0; i < loopCount; ++i)
        {
            Thread.Sleep(10);
            Console.WriteLine($"{Thread.CurrentThread.Name} : {i}");
        }
```

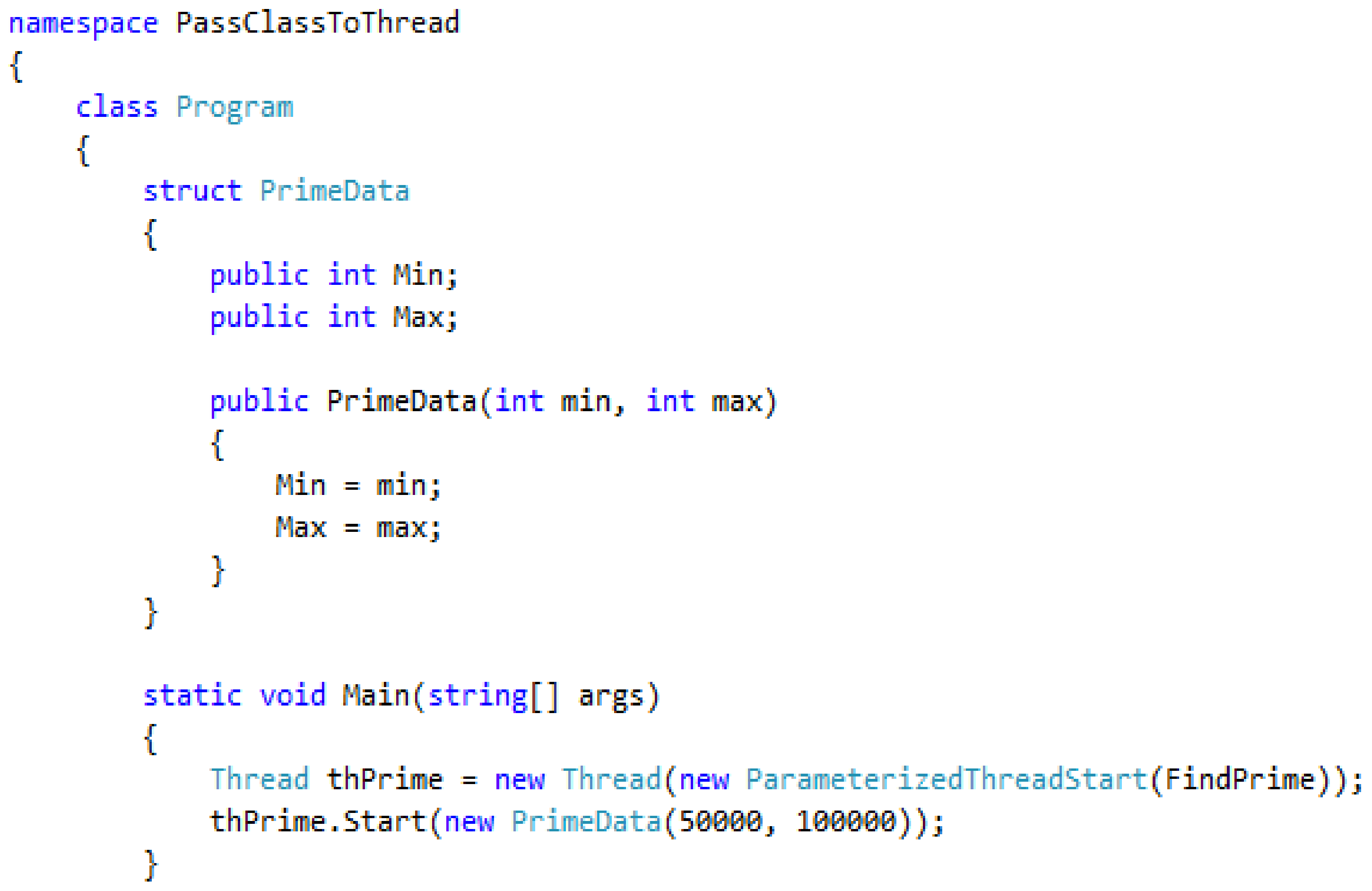
ParameterizedThreadStart

- In the previous example, an integer is passed to each thread as type **object**.
- The parameter received in the **Counter** method is checked using the **is** operator to ensure that it is of the expected data type.
- If the data type is incorrect, the thread does not run the loop, but exits.

ParameterizedThreadStart

- The next example uses a struct to pass two values to the thread.
- The thread will determine the prime numbers found between (inclusive) the two values passed to it.
- The algorithm used is (deliberately) very inefficient.

ParameterizedThreadStart

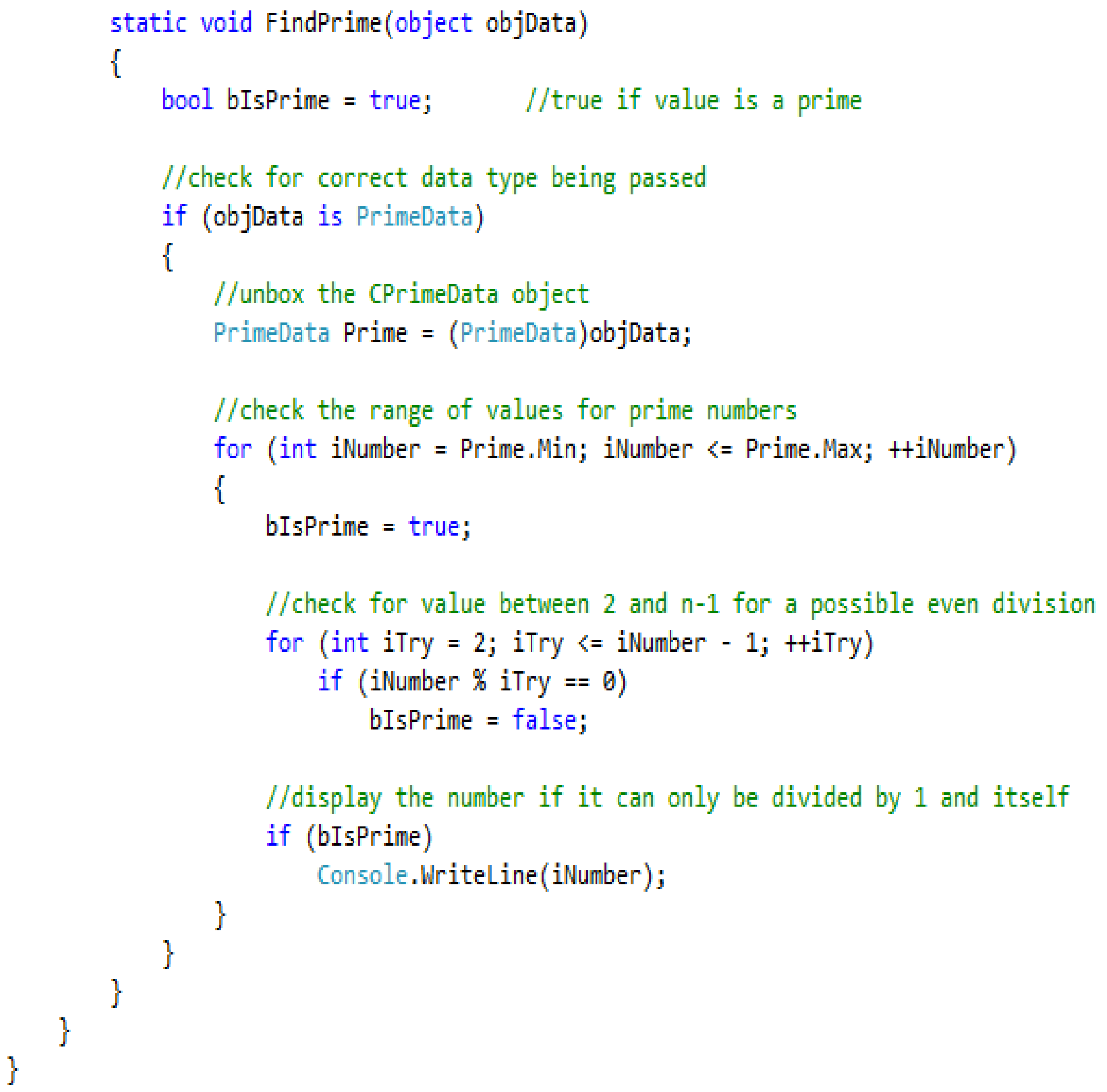


```
namespace PassClassToThread
{
    class Program
    {
        struct PrimeData
        {
            public int Min;
            public int Max;

            public PrimeData(int min, int max)
            {
                Min = min;
                Max = max;
            }
        }

        static void Main(string[] args)
        {
            Thread thPrime = new Thread(new ParameterizedThreadStart(FindPrime));
            thPrime.Start(new PrimeData(50000, 100000));
        }
```

ParameterizedThreadStart



```
        static void FindPrime(object objData)
        {
            bool bIsPrime = true;       //true if value is a prime

            //check for correct data type being passed
            if (objData is PrimeData)
            {
                //unbox the CPrimeData object
                PrimeData Prime = (PrimeData)objData;

                //check the range of values for prime numbers
                for (int iNumber = Prime.Min; iNumber <= Prime.Max; ++iNumber)
                {
                    bIsPrime = true;

                    //check for value between 2 and n-1 for a possible even division
                    for (int iTry = 2; iTry <= iNumber - 1; ++iTry)
                        if (iNumber % iTry == 0)
                            bIsPrime = false;

                    //display the number if it can only be divided by 1 and itself
                    if (bIsPrime)
                        Console.WriteLine(iNumber);
                }
            }
        }
    }
}
```

ParameterizedThreadStart

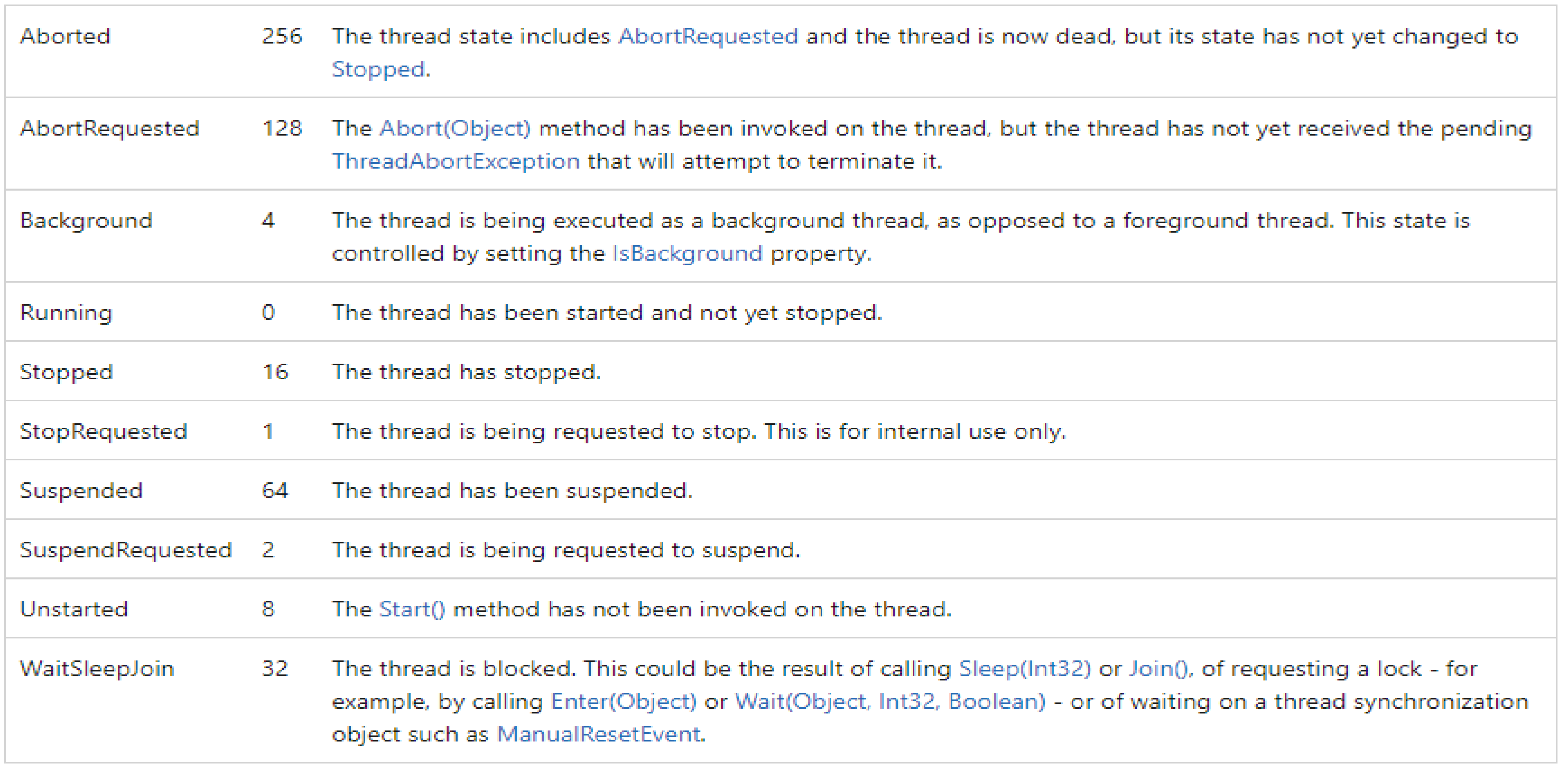
- The **PrimeData** struct is constructed within the **Start()** method call.
- The **FindPrime()** method receives the **PrimeData** struct as type **object**, which is checked then **unboxed** back to **PrimeData**.
- The prime number calculation takes place using a foreground thread.

ParameterizedThreadStart

- It is to be noted that when we use **new Thread(method)**, the system can infer from the method signature whether to use the **ThreadStart** or **ParameterizedThreadStart** delegate.
- So, in the previous cases, we can still use **new Thread(Counter)** and **new Thread(FindPrime)**. Then, we pass the required parameter to **Thread.Start()**

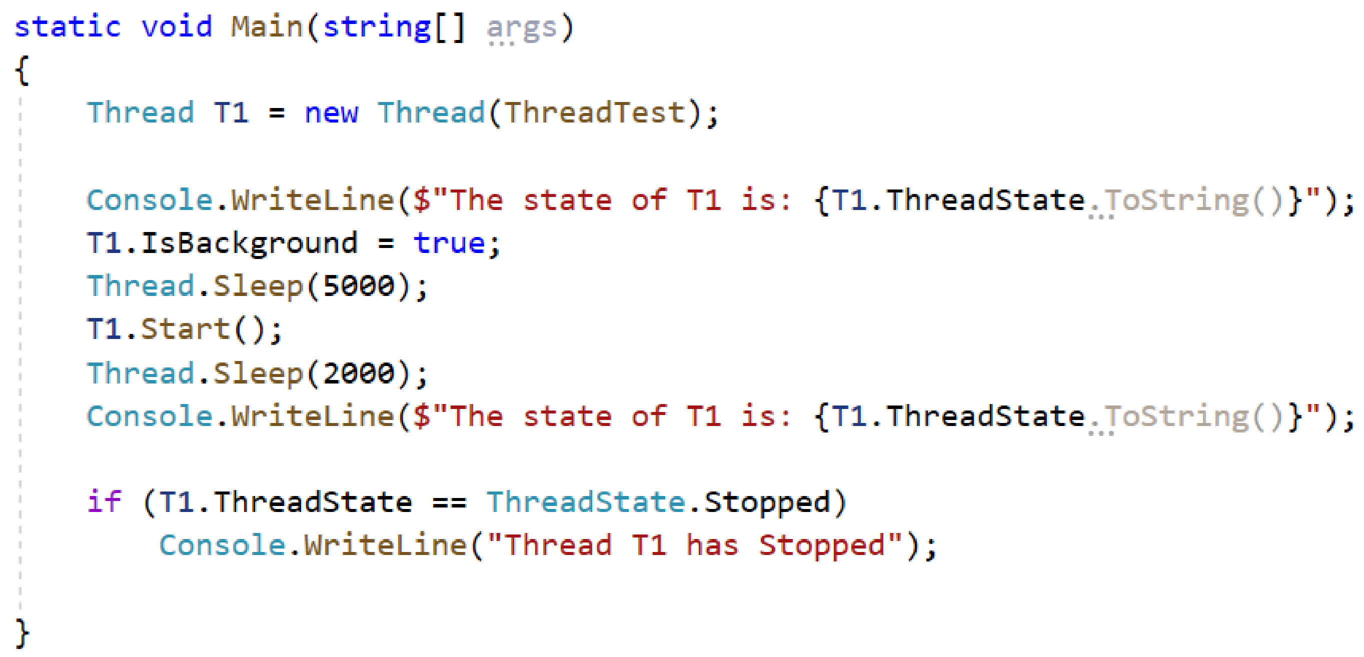
Thread States

- A thread can be in one of different states such as it may be running, not yet started or terminated ..etc.
- The ThreadState enumeration type allow us to keep track or to check the state of a thread. The possible values and their meanings are as listed below:



Aborted	256	The thread state includes AbortRequested and the thread is now dead, but its state has not yet changed to Stopped.
AbortRequested	128	The Abort(Object) method has been invoked on the thread, but the thread has not yet received the pending ThreadAbortException that will attempt to terminate it.
Background	4	The thread is being executed as a background thread, as opposed to a foreground thread. This state is controlled by setting the IsBackground property.
Running	0	The thread has been started and not yet stopped.
Stopped	16	The thread has stopped.
StopRequested	1	The thread is being requested to stop. This is for internal use only.
Suspended	64	The thread has been suspended.
SuspendRequested	2	The thread is being requested to suspend.
Unstarted	8	The Start() method has not been invoked on the thread.
WaitSleepJoin	32	The thread is blocked. This could be the result of calling Sleep(Int32) or Join(), of requesting a lock - for example, by calling Enter(Object) or Wait(Object, Int32, Boolean) - or of waiting on a thread synchronization object such as ManualResetEvent.

Displaying and Verifying Thread States

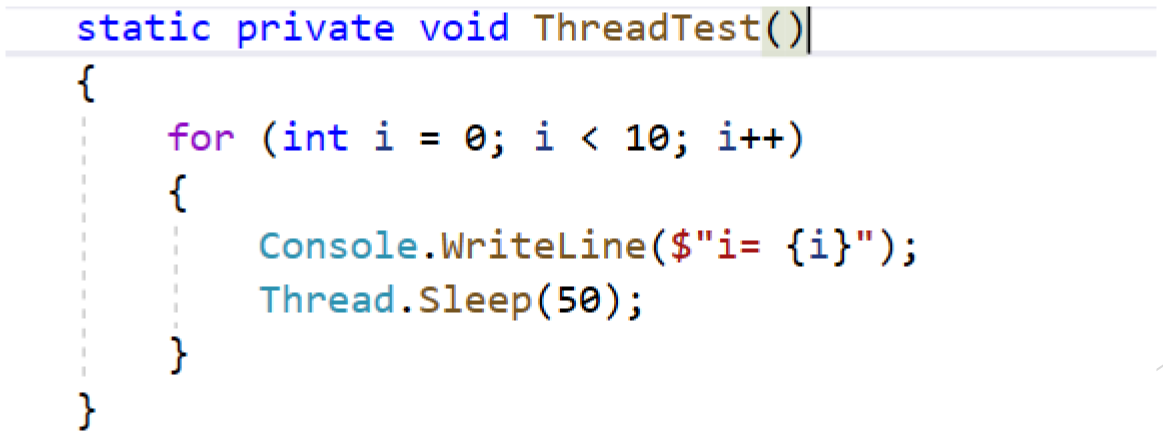


```
static void Main(string[] args)
{
    Thread T1 = new Thread(ThreadTest);

    Console.WriteLine($"The state of T1 is: {T1.ThreadState.ToString()}");
    T1.IsBackground = true;
    Thread.Sleep(5000);
    T1.Start();
    Thread.Sleep(2000);
    Console.WriteLine($"The state of T1 is: {T1.ThreadState.ToString()}");

    if (T1.ThreadState == ThreadState.Stopped)
        Console.WriteLine("Thread T1 has Stopped");

}
```



```
    static private void ThreadTest()
    {
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine($"i= {i}");
            Thread.Sleep(50);
        }
    }
```

Lecture 10- Exercise3

- Make a copy of Exercise2 and call it Lecture10Exercise3
- Modify both the methods **FindSine()** and **FindCos()** so as to use a class-level variable instead of the local variable x. You will need a class-level variable for each method. This will allow the threads to continue from the last value reached.
- Ensure that you don't clear the listboxes before display.
- Declare the variables runSine and runCos to be volatile.

Lecture 10- Exercise4

- Make a copy of exercise3 and modify it such that the stopping angle for both the **FindSine()** and the **FindCos()** methods are passed as parameters to the created threads.

Lecture 10- Exercise5

- **Make a copy of exercise4 and modify it such that the both the starting and stopping angles for both the FindSine() and the FindCos() methods are passed as parameters to the created threads.**