C# Threading and Types of Threading Step by Step : Learn ...

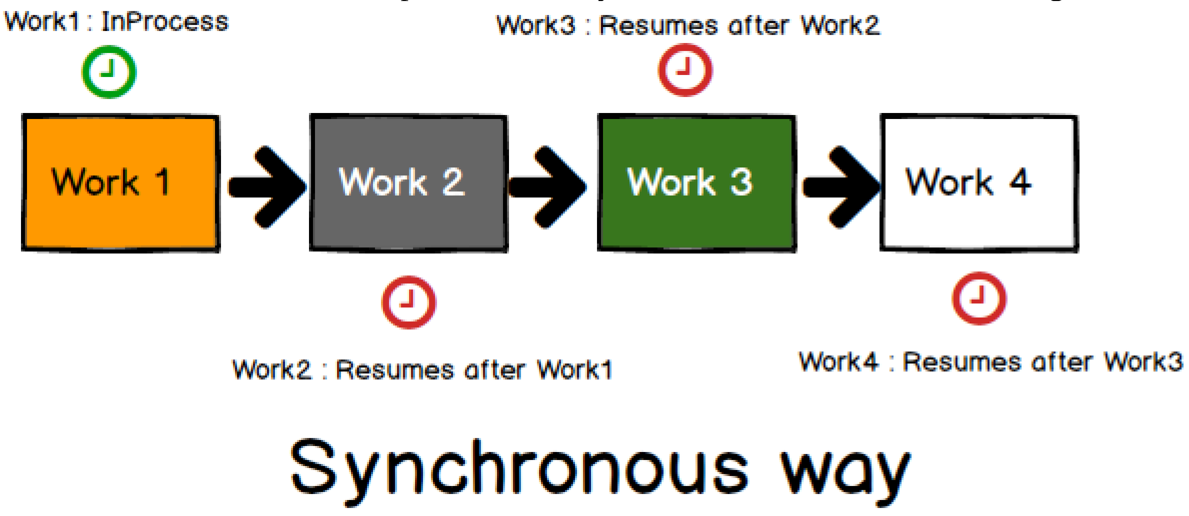
Introduction

C# Threading

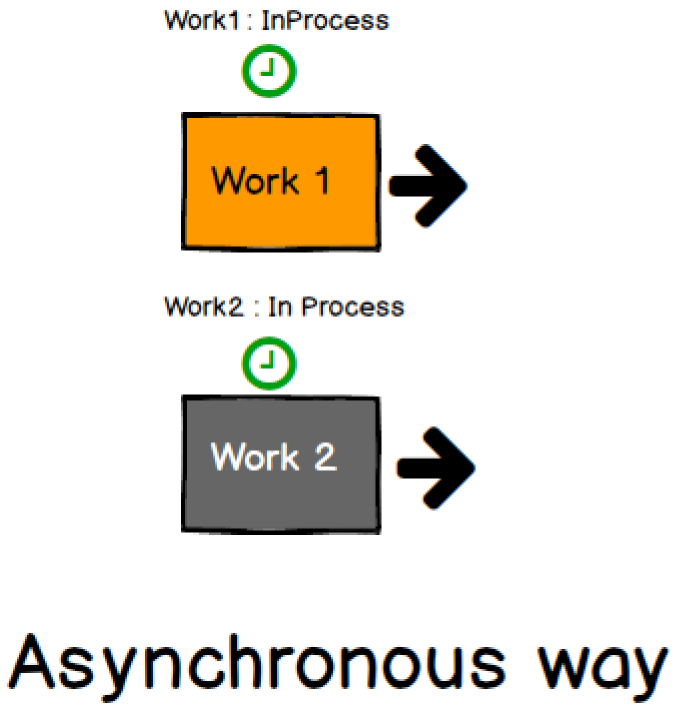
To define Threading in a one line, means parallel work or code execution. To perform any multiple task simultaneously means Threading.

For example executing Microsoft PPTX and Excel simultaneously in a desktop, laptop or any device is known as Threading.
Work or a code task is always been executed in a two ways i.e. Synchronous or Asynchronous way.

Synchronous way means where work multiple jobs are executed one after the other. Here Work 2 have to wait till Work 1 is completed same way the others as shown in below image.


Work1 : InProcess
Work3 : Resumes after Work2
Work 1
Work 2
Work 3
Work 4
Work2 : Resumes after Work1
Work4 : Resumes after Work3
Synchronous way

Asynchronous means multiple work has been executed simultaneously like doing multitask at a same time.



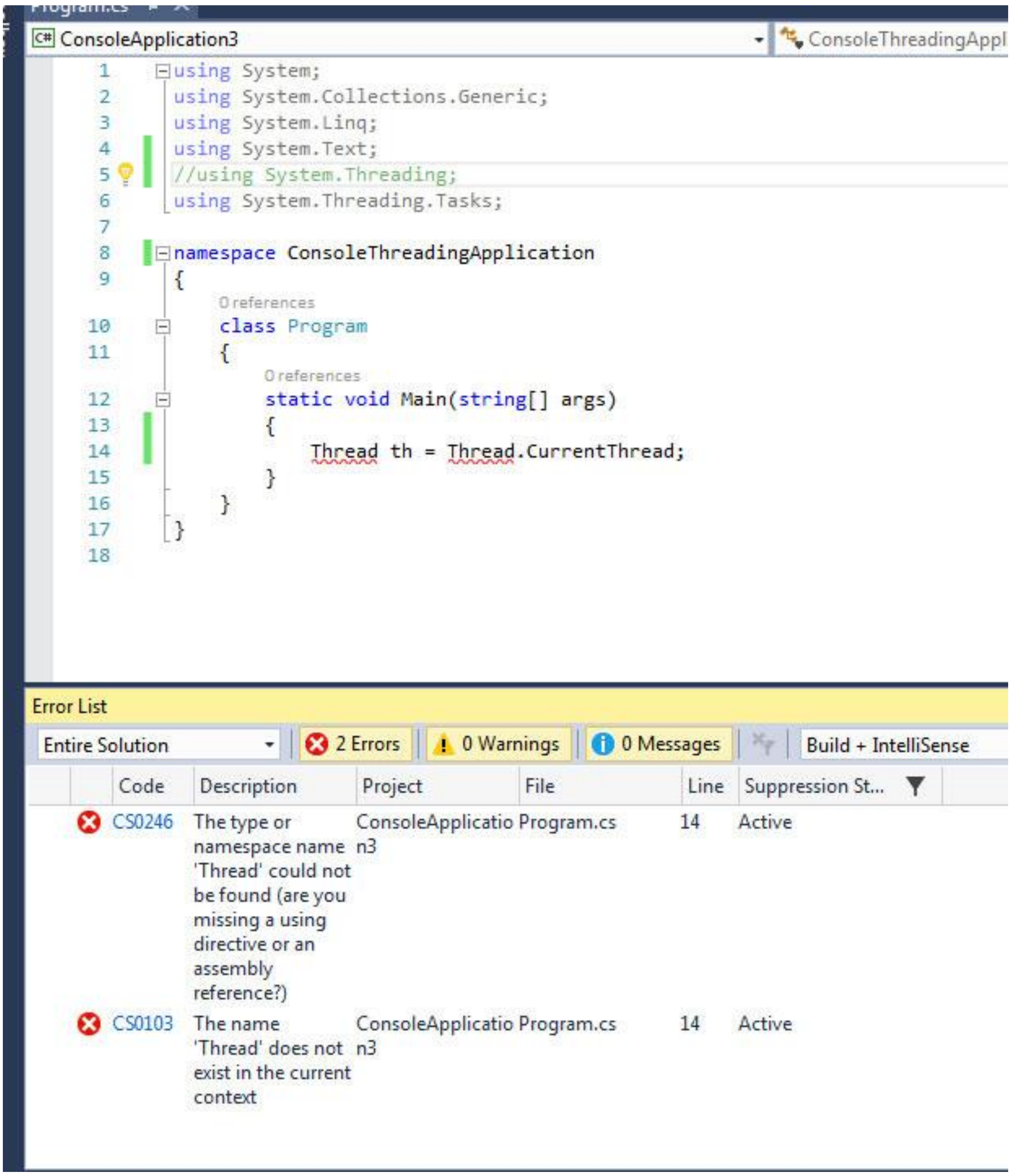
Execution of Microsoft Powerpoint and Excel asynchronously on our personal laptop is most common example for threading. System generates new thread for each application been launched because every thread has an independent execution path to run these applications asynchronously.

C# widely supports threading in Console Application, Windows Form, WPF so when we create a new thread in a console app it means we are making it multithread environment.

CSharp Threading step by step using example

Here we will demonstrate threading using an example step by step, We will do this example in a visual studio 2015.
In order to create threading in C# VS 2015 we need to import threading namespace i.e. System.Threading using this namespace we can create thread object which can help us to create thread applications.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleThreadingApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            Thread th = Thread.CurrentThread;
        }
    }
}
```


Program.cs

ConsoleApplication3 | ConsoleThreadingAppl

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
//using System.Threading;
using System.Threading.Tasks;

namespace ConsoleThreadingApplication
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Thread th = Thread.CurrentThread;
        }
    }
}
```

Error List

Entire Solution | 2 Errors | 0 Warnings | 0 Messages | Build + IntelliSense

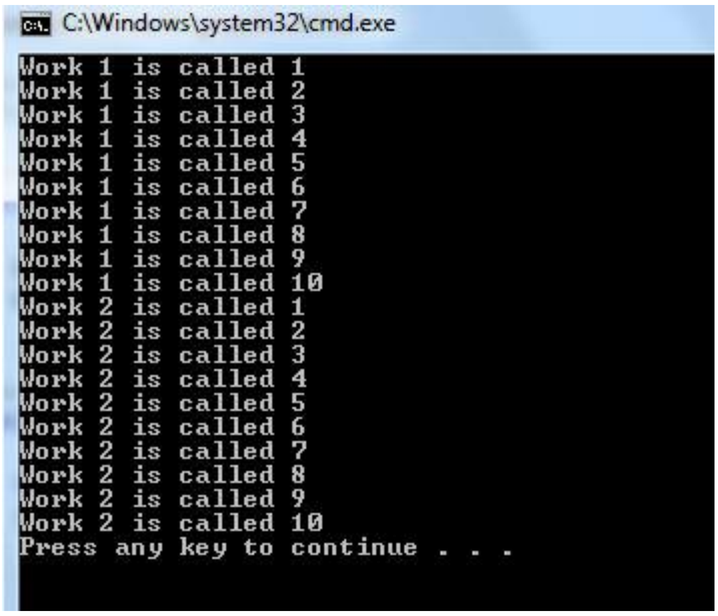
	Code	Description	Project	File	Line	Suppression St...
⊗	CS0246	The type or namespace name 'Thread' could not be found (are you missing a using directive or an assembly reference?)	ConsoleApplication3	Program.cs	14	Active
⊗	CS0103	The name 'Thread' does not exist in the current context	ConsoleApplication3	Program.cs	14	Active

As you saw in an above image that without Threading Namespace we can't able to use Thread classes and objects. So it is must to import threading namespace in-order to create Threading application.

Now to make you understand more better we will create two methods i.e. Work1 and Work2 respectively and inside that we will make a FOR loop. Here we want to test how both functions are executes.

```
class Program
    {
        static void Main(string[] args)
        {
            Work1();
            Work2();
        }
        static void Work1()
        {
            for(int i = 1; i <=10; i++)
            {

                Console.WriteLine("Work 1 is called " + i.ToString());

            }

        }

        static void Work2()

        {
            for (int i = 1; i <= 10; i++)
            {

                Console.WriteLine("Work 2 is called " + i.ToString());

            }

        }
    }
```

As you see in above code snippet we have created both normal functions and that we will execute, So let's run it and see the output.



```
C:\Windows\system32\cmd.exe
Work 1 is called 1
Work 1 is called 2
Work 1 is called 3
Work 1 is called 4
Work 1 is called 5
Work 1 is called 6
Work 1 is called 7
Work 1 is called 8
Work 1 is called 9
Work 1 is called 10
Work 2 is called 1
Work 2 is called 2
Work 2 is called 3
Work 2 is called 4
Work 2 is called 5
Work 2 is called 6
Work 2 is called 7
Work 2 is called 8
Work 2 is called 9
Work 2 is called 10
Press any key to continue . . .
```

If you see both functions / methods ran synchronously i.e one after the other. Here Work 2 have to wait till Work 1 has finished his loop.

But in this fast moving world we have a habbit of doing multi-tasking so here also in a same way we need some kind of mechanism which helps us to run both methods asynchronously i.e simultaneously. So to make that c# has a mechanism called Threading.
So we understood the problem and now we will see how threading helps to fix this problem i.e. Running both methods simultaneously.

Step 1

First and foremost step to import Threading namespace.

```
using System;
using System.Threading;
```

Step 2

Here in this step we will create thread objects in our Main method.

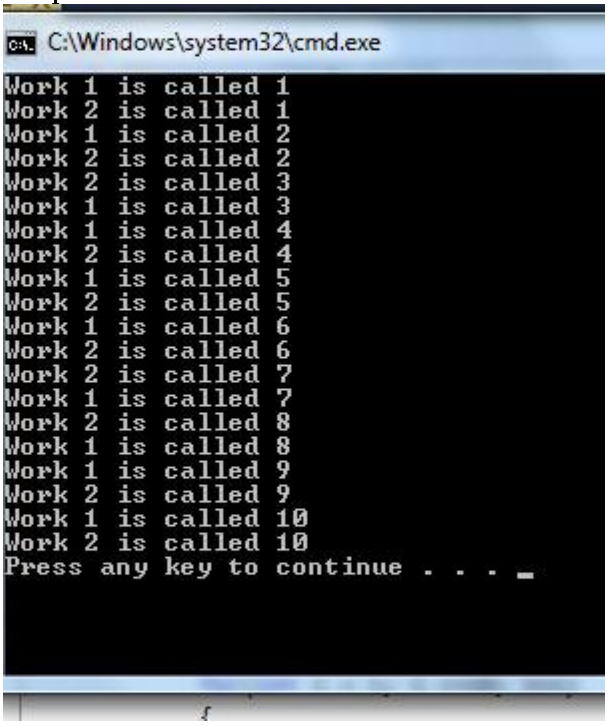
```
 class Program
    {
        static void Main(string[] args)
        {
            Thread oThreadone = new Thread(Work1);
            Thread oThreadtwo = new Thread(Work2);


        }
  }
```

Step 3

In this step we will invoke our thread objects

```
 class Program
    {
        static void Main(string[] args)
        {
            Thread oThreadone = new Thread(Work1);
            Thread oThreadtwo = new Thread(Work2);

           oThreadone.Start();
            oThreadtwo.Start();


        }
  }
```

As you see we have invoked Thread objects successfully. Now let's run this program to see the Output.



```
Work 1 is called 1
Work 2 is called 1
Work 1 is called 2
Work 2 is called 2
Work 2 is called 3
Work 1 is called 3
Work 1 is called 4
Work 2 is called 4
Work 1 is called 5
Work 2 is called 5
Work 1 is called 6
Work 2 is called 6
Work 2 is called 7
Work 1 is called 7
Work 2 is called 8
Work 1 is called 8
Work 1 is called 9
Work 2 is called 9
Work 1 is called 10
Work 2 is called 10
Press any key to continue . . . _
```

As you see in output that Work2 method is also simultaneously executing with Work1 method it means both methods are working asynchronously.

So as per above examples it is been concluded that using threading we can execute mutiple work in a asynchronously.

For creating threading application there some important methods which used regularly for implementing threading.

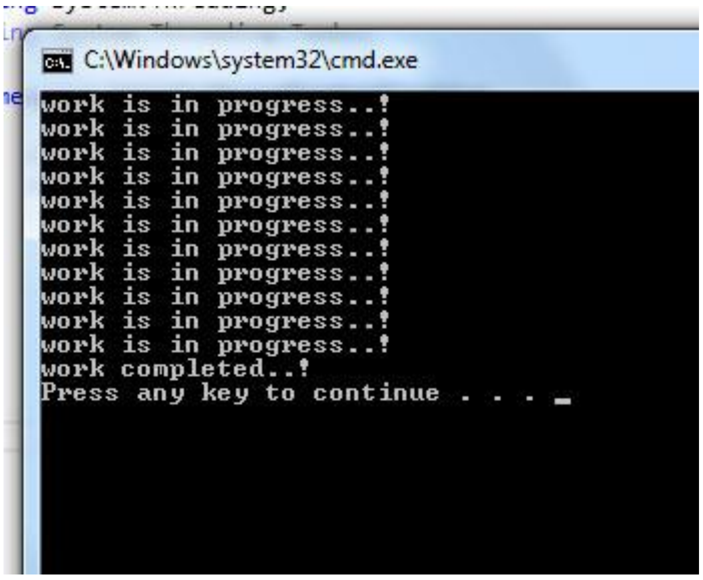
1. 1 : Thread Join
2. 2 : Thread Sleep
3. 3 : Thread Abort

Use of Thread Join

Thread.Join() make thread to finish its work or makes other thread to halt until it finishes work. Join method when attached to any thread, it makes that thread to execute first and halts other threads. Now on the same we will see a simple example where we apply Join method to thread.

```
class Program
    {

        static void Main(string[] args)
        {

            Thread oThread = new Thread(MethodJoin);
            oThread.Start();
            oThread.Join();
            Console.WriteLine("work completed..!");

        }

        static void MethodJoin()
        {
            for (int i = 0; i <= 10; i++)
            {
                Console.WriteLine("work is in progress..!");

            }

        }


    }
```

So hey friends as you see we have created a method called "MethodJoin()" and attached it with a Thread.Join method so as per our theory discussion above "MethodJoin()" will execute first and then main method let's see the output.



```
work is in progress..!
work is in progress..!
work is in progress..!
work is in progress..!
work is in progress..!
work is in progress..!
work is in progress..!
work is in progress..!
work is in progress..!
work is in progress..!
work is in progress..!
work completed..!
Press any key to continue . . . _
```

As you see the output "MethodJoin" is been executed first i.e. output as "work is in progress..!" and then Main method of console application executed i.e. output as "work completed..!"

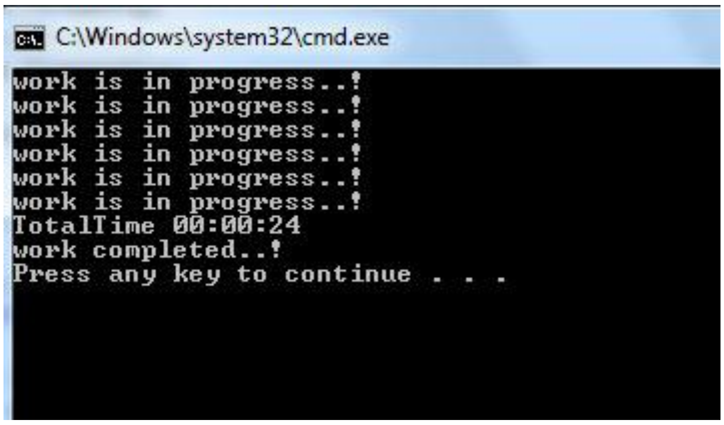
Use of Thread Sleep

Thread.Sleep a method used to suspend current thread for a specific interval of time. Time can be specified in milliseconds or Timespan. While in a Sleep mode a method does not consumes any CPU resources so indirectly it save memory for other thread processes.

On the same we will create a simple example where in the for loop while printing output we will make a thread to sleep for 4000 milliseconds i.e. 4 secs for per print and total we are going to print 6 outputs. To count it properly we have used .NET Diagnostics namespace which allows us to use Stopwatch and TimeSpan to count elapsedTime. Complete example is shown below

```
using System.Threading;
using System.Diagnostics;

    class Program
    {

        static void Main(string[] args)
        {
            Stopwatch stWatch = new Stopwatch();
            stWatch.Start();

            Thread oThread = new Thread(ProcessSleep);
            oThread.Start();
            oThread.Join();

            stWatch.Stop();
            TimeSpan ts = stWatch.Elapsed;

            string elapsedTime = String.Format("{0:00}:{1:00}:{2:00}",ts.Hours,
ts.Minutes, ts.Seconds);
            Console.WriteLine("TotalTime " + elapsedTime);

            Console.WriteLine("work completed..!");

        }

        static void ProcessSleep()
        {
            for (int i = 0; i <= 5; i++)
            {
                Console.WriteLine("work is in progress..!");
                Thread.Sleep(4000); //Sleep for 4 seconds

            }

        }


    }
```

As you see from above example we have implemented Sleep method in a ProcessSleep method and made thread to sleep for 4 secs for each print. For only to show you total time consumed to print all output we have used Diagnostics.Stopwatch and Diagnostics.Timespan. Output is shown below.



```
work is in progress..!
work is in progress..!
work is in progress..!
work is in progress..!
work is in progress..!
work is in progress..!
TotalTime 00:00:24
work completed..!
Press any key to continue . . .
```

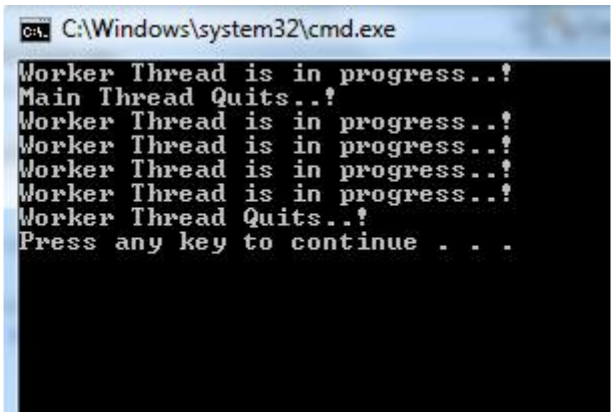
Use of Thread Abort
As name implies "Abort" so same way Thread.Abort helps to end or abort any thread to process it further. It raises ThreadAbortException in the thread for process of termination.

```
Thread objThread = new Thread(ProcessJoin);
objThread.Start();
objThread.Join();

objThread.Abort();
```

Types of Threads in C#
There are two types of Thread in Csharp i.e. Foreground Thread and Background Thread. Further let's see the uses of these threads.

Foreground Thread
As we know that Main method is also runs on single thread, So in a Main method when we attach any other method to a thread it means we are making a multithread application. Foreground threads are those threads which keeps running until it finishes his work even if the Main method thread quits its process. To make you understand more better let me show you. Lifespan of foreground threads does not depends on main thread.

```
        class Program
    {

         static void Main(string[] args)
         {
             Thread oThread = new Thread(WorkThread);
             oThread.Start();
             Console.WriteLine("Main Thread Quits..!");
         }

         static void WorkThread()
         {
             for (int i = 0; i <= 4; i++)
             {
                 Console.WriteLine("Worker Thread is in progress..!");
                 Thread.Sleep(2000); //Sleep for 2 seconds

             }
             Console.WriteLine("Worker Thread Quits..!");

         }


    }
```


C:\Windows\system32\cmd.exe

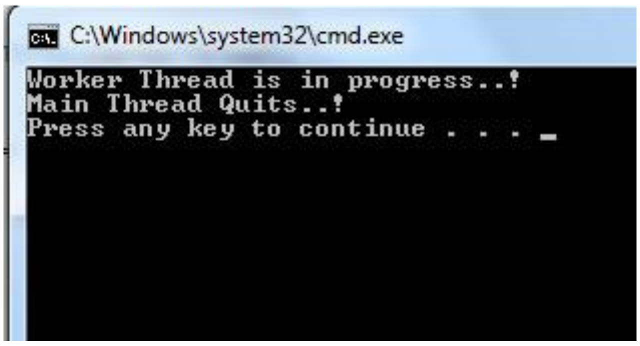
```
Worker Thread is in progress..!
Main Thread Quits..!
Worker Thread is in progress..!
Worker Thread is in progress..!
Worker Thread is in progress..!
Worker Thread is in progress..!
Worker Thread Quits..!
Press any key to continue . . .
```

As you see output even if the Main thread quits then also WorkThread() continue to execute and complete its work. This was an ideal example to understand foreground thread. If you guys know any better example feel free to post under comment section your ideas will help us to write this article in more better and help to share knowledge.

Background Thread

Background thread is just opposite of foreground thread here background thread quits its job when main thread quits. Here lifespan of background threads depends on main thread. In order to implement background thread in a program we need to set property called IsBackground to true.

```
Thread oThread = new Thread(WorkThread);
oThread.IsBackground = true;

class Program
   {

       static void Main(string[] args)
       {

           Thread oThread = new Thread(WorkThread);
           oThread.Start();

           oThread.IsBackground = true;

           Console.WriteLine("Main Thread Quits..!");


       }

       static void WorkThread()
       {


           for (int i = 0; i <= 4; i++)
           {
               Console.WriteLine("Worker Thread is in progress..!");
               Thread.Sleep(2000); //Sleep for 2 seconds

           }

           Console.WriteLine("Worker Thread Quits..!");

       }


   }
```



C:\Windows\system32\cmd.exe

```
Worker Thread is in progress..!
Main Thread Quits..!
Press any key to continue . . .
```

C# - Multithreading

A **thread** is defined as the execution path of a program. Each thread defines a unique flow of control. If your application involves complicated and time consuming operations, then it is often helpful to set different execution paths or threads, with each thread performing a particular job.
Threads are **lightweight processes**. One common example of use of thread is implementation of concurrent programming by modern operating systems. Use of threads saves wastage of CPU cycle and increase efficiency of an application.
So far we wrote the programs where a single thread runs as a single process which is the running instance of the application. However, this way the application can perform one job at a time. To make it execute more than one task at a time, it could be divided into smaller threads.

Thread Life Cycle

The life cycle of a thread starts when an object of the System.Threading.Thread class is created and ends when the thread is terminated or completes execution.
Following are the various states in the life cycle of a thread –

- **The Unstarted State** − It is the situation when the instance of the thread is created but the Start method is not called.
- **The Ready State** − It is the situation when the thread is ready to run and waiting CPU cycle.
- **The Not Runnable State** − A thread is not executable, when
  - Sleep method has been called
  - Wait method has been called
  - Blocked by I/O operations
- **The Dead State** − It is the situation when the thread completes execution or is aborted.

The Main Thread

In C#, the **System.Threading.Thread** class is used for working with threads. It allows creating and accessing individual threads in a multithreaded application. The first thread to be executed in a process is called the **main** thread.

When a C# program starts execution, the main thread is automatically created. The threads created using the **Thread** class are called the child threads of the main thread. You can access a thread using the **CurrentThread** property of the Thread class.
The following program demonstrates main thread execution −

```
using System;
using System.Threading;

namespace MultithreadingApplication {
   class MainThreadProgram {
      static void Main(string[] args) {
         Thread th = Thread.CurrentThread;
         th.Name = "MainThread";

         Console.WriteLine("This is {0}", th.Name);
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result −

```
This is MainThread
```

Properties and Methods of the Thread Class

The following table shows some most commonly used **properties** of the **Thread** class −

Sr.No.	Property & Description

1	**CurrentContext** Gets the current context in which the thread is executing.
2	**CurrentCulture** Gets or sets the culture for the current thread.
3	**CurrentPrinciple** Gets or sets the thread's current principal (for role-based security).
4	**CurrentThread** Gets the currently running thread.
5	**CurrentUICulture** Gets or sets the current culture used by the Resource Manager to look up culture-specific resources at run-time.
6	**ExecutionContext** Gets an ExecutionContext object that contains information about the various contexts of the current thread.
7	**IsAlive** Gets a value indicating the execution status of the current thread.
8	**IsBackground** Gets or sets a value indicating whether or not a thread is a background thread.
9	**IsThreadPoolThread** Gets a value indicating whether or not a thread belongs to the managed thread pool.
10	**ManagedThreadId** Gets a unique identifier for the current managed thread.
11	**Name** Gets or sets the name of the thread.
12	**Priority** Gets or sets a value indicating the scheduling priority of a thread.
13	**ThreadState** Gets a value containing the states of the current thread.

The following table shows some of the most commonly used **methods** of the **Thread** class −

Sr.No.	Method & Description
1	**public void Abort()** Raises a ThreadAbortException in the thread on which it is invoked, to begin the process of terminating the thread. Calling this method usually terminates the thread.
2	**public static LocalDataStoreSlot AllocateDataSlot()** Allocates an unnamed data slot on all the threads. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead.
3	**public static LocalDataStoreSlot AllocateNamedDataSlot(string name)** Allocates a named data slot on all threads. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead.
4	**public static void BeginCriticalRegion()** Notifies a host that execution is about to enter a region of code in which the effects of a thread abort or unhandled exception might jeopardize other tasks in the application domain.
5	**public static void BeginThreadAffinity()** Notifies a host that managed code is about to execute instructions that depend on the identity of the current physical operating system thread.
6	**public static void EndCriticalRegion()** Notifies a host that execution is about to enter a region of code in which the effects of a thread abort or unhandled exception are limited to the current task.
7	**public static void EndThreadAffinity()**

	Notifies a host that managed code has finished executing instructions that depend on the identity of the current physical operating system thread.
8	**public static void FreeNamedDataSlot(string name)** Eliminates the association between a name and a slot, for all threads in the process. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead.
9	**public static Object GetData(LocalDataStoreSlot slot)** Retrieves the value from the specified slot on the current thread, within the current thread's current domain. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead.
10	**public static AppDomain GetDomain()** Returns the current domain in which the current thread is running.
11	**public static AppDomain GetDomainID()** Returns a unique application domain identifier
12	**public static LocalDataStoreSlot GetNamedDataSlot(string name)** Looks up a named data slot. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead.
13	**public void Interrupt()** Interrupts a thread that is in the WaitSleepJoin thread state.
14	**public void Join()** Blocks the calling thread until a thread terminates, while continuing to perform standard COM and SendMessage pumping. This method has different overloaded forms.
15	**public static void MemoryBarrier()** Synchronizes memory access as follows: The processor executing the current thread cannot reorder instructions in such a way that memory accesses prior to the call to MemoryBarrier execute after memory accesses that follow the call to MemoryBarrier.
16	**public static void ResetAbort()** Cancels an Abort requested for the current thread.
17	**public static void SetData(LocalDataStoreSlot slot, Object data)** Sets the data in the specified slot on the currently running thread, for that thread's current domain. For better performance, use fields marked with the ThreadStaticAttribute attribute instead.
18	**public void Start()** Starts a thread.
19	**public static void Sleep(int millisecondsTimeout)** Makes the thread pause for a period of time.
20	**public static void SpinWait(int iterations)** Causes a thread to wait the number of times defined by the iterations parameter
21	**public static byte VolatileRead(ref byte address)** **public static double VolatileRead(ref double address)** **public static int VolatileRead(ref int address)** **public static Object VolatileRead(ref Object address)** Reads the value of a field. The value is the latest written by any processor in a computer, regardless of the number of processors or the state of processor cache. This method has different overloaded forms. Only some are given above.
22	**public static void VolatileWrite(ref byte address,byte value)** **public static void VolatileWrite(ref double address, double value)** **public static void VolatileWrite(ref int address, int value)** **public static void VolatileWrite(ref Object address, Object value)**

Writes a value to a field immediately, so that the value is visible to all processors in the computer. This method has different overloaded forms. Only some are given above.

23 **public static bool Yield()**
Causes the calling thread to yield execution to another thread that is ready to run on the current processor. The operating system selects the thread to yield to.

Creating Threads

Threads are created by extending the Thread class. The extended Thread class then calls the **Start()** method to begin the child thread execution.
The following program demonstrates the concept −
Live Demo
```
using System;
using System.Threading;

namespace MultithreadingApplication {
   class ThreadCreationProgram {
      public static void CallToChildThread() {
         Console.WriteLine("Child thread starts");
      }
      static void Main(string[] args) {
         ThreadStart childref = new ThreadStart(CallToChildThread);
         Console.WriteLine("In Main: Creating the Child thread");
         Thread childThread = new Thread(childref);
         childThread.Start();
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result −
```
In Main: Creating the Child thread
Child thread starts
```

Managing Threads

The Thread class provides various methods for managing threads.
The following example demonstrates the use of the **sleep()** method for making a thread pause for a specific period of time.
Live Demo
```
using System;
using System.Threading;

namespace MultithreadingApplication {
   class ThreadCreationProgram {
      public static void CallToChildThread() {
         Console.WriteLine("Child thread starts");

         // the thread is paused for 5000 milliseconds
         int sleepfor = 5000;

         Console.WriteLine("Child Thread Paused for {0} seconds", sleepfor /
1000);
         Thread.Sleep(sleepfor);
         Console.WriteLine("Child thread resumes");
      }

      static void Main(string[] args) {
         ThreadStart childref = new ThreadStart(CallToChildThread);
         Console.WriteLine("In Main: Creating the Child thread");
         Thread childThread = new Thread(childref);
         childThread.Start();
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result −

```
In Main: Creating the Child thread
Child thread starts
Child Thread Paused for 5 seconds
Child thread resumes
```

Destroying Threads

The **Abort()** method is used for destroying threads.

The runtime aborts the thread by throwing a **ThreadAbortException**. This exception cannot be caught, the control is sent to the *finally* block, if any.

The following program illustrates this −

```
using System;
using System.Threading;

namespace MultithreadingApplication {
   class ThreadCreationProgram {
      public static void CallToChildThread() {
         try {
            Console.WriteLine("Child thread starts");

            // do some work, like counting to 10
            for (int counter = 0; counter <= 10; counter++) {
               Thread.Sleep(500);
               Console.WriteLine(counter);
            }

            Console.WriteLine("Child Thread Completed");
         } catch (ThreadAbortException e) {
            Console.WriteLine("Thread Abort Exception");
         } finally {
            Console.WriteLine("Couldn't catch the Thread Exception");
         }
      }
      static void Main(string[] args) {
         ThreadStart childref = new ThreadStart(CallToChildThread);
         Console.WriteLine("In Main: Creating the Child thread");

         Thread childThread = new Thread(childref);
         childThread.Start();

         //stop the main thread for some time
         Thread.Sleep(2000);

         //now abort the child
         Console.WriteLine("In Main: Aborting the Child thread");

         childThread.Abort();
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result −

```
In Main: Creating the Child thread
Child thread starts
0
```

```
1
2
In Main: Aborting the Child thread
Thread Abort Exception
Couldn't catch the Thread Exception
```

Threading in C#

Joseph Albahari

Part 1	Part 2	Part 3	Part 4	Part 5
Getting Started	Basic Synchronization	Using Threads	Advanced Threading	Parallel Programming

Download PDF

Part 1: Getting Started

Introduction and Concepts

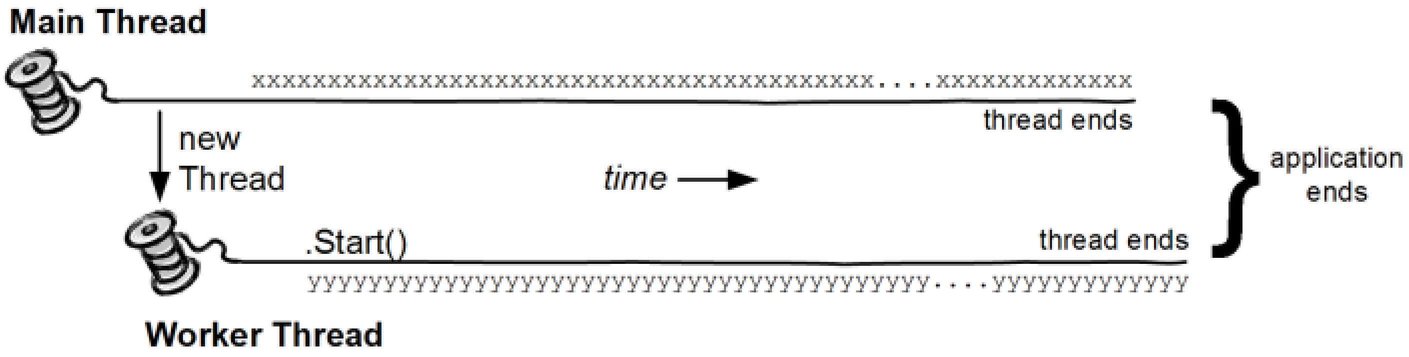
C# supports parallel execution of code through multithreading. A thread is an independent execution path, able to run simultaneously with other threads.

A C# *client* program (Console, WPF, or Windows Forms) starts in a single thread created automatically by the CLR and operating system (the "main" thread), and is made multithreaded by creating additional threads. Here's a simple example and its output:
All examples assume the following namespaces are imported:

```
using System;
using System.Threading;
class ThreadTest
{
  static void Main()
  {
    Thread t = new Thread (WriteY);          // Kick off a new thread
    t.Start();                               // running WriteY()

    // Simultaneously, do something on the main thread.
    for (int i = 0; i < 1000; i++) Console.Write ("x");
  }

  static void WriteY()
  {
    for (int i = 0; i < 1000; i++) Console.Write ("y");
  }
}
```

```
xxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
...
```

The main thread creates a new thread `t` on which it runs a method that repeatedly prints the character "y". Simultaneously, the main thread repeatedly prints the character "x":


Main Thread
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx....xxxxxxxxxxxxx
thread ends
new Thread
time
application ends
.Start()
thread ends
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy....yyyyyyyyyyyyy
Worker Thread

Once started, a thread’s `IsAlive` property returns `true`, until the point where the thread ends. A thread ends when the delegate passed to the `Thread`’s constructor finishes executing. Once ended, a thread cannot restart.

The CLR assigns each thread its own memory stack so that local variables are kept separate. In the next example, we define a method with a local variable, then call the method simultaneously on the main thread and a newly created thread:

```
static void Main()
{
  new Thread (Go).Start();      // Call Go() on a new thread
  Go();                         // Call Go() on the main thread
}

static void Go()
{
  // Declare and use a local variable - 'cycles'
  for (int cycles = 0; cycles < 5; cycles++) Console.Write ('?');
}
```

```
??????????
```

A separate copy of the cycles variable is created on each thread's memory stack, and so the output is, predictably, ten question marks.
Threads share data if they have a common reference to the same object instance. For example:

```
class ThreadTest
{
  bool done;

  static void Main()
  {
    ThreadTest tt = new ThreadTest();   // Create a common instance
    new Thread (tt.Go).Start();
    tt.Go();
  }

  // Note that Go is now an instance method
  void Go()
  {
     if (!done) { done = true; Console.WriteLine ("Done"); }
  }
}
```

Because both threads call `Go()` on the same `ThreadTest` instance, they share the `done` field. This results in "Done" being printed once instead of twice:

```
Done
```

Static fields offer another way to share data between threads. Here’s the same example with `done` as a static field:

```
class ThreadTest
{
  static bool done;    // Static fields are shared between all threads
  static void Main()
  {
    new Thread (Go).Start();
    Go();
  }

  static void Go()
  {
    if (!done) { done = true; Console.WriteLine ("Done"); }
  }
}
```

Both of these examples illustrate another key concept: that of thread safety (or rather, lack of it!) The output is actually indeterminate: it's possible (though unlikely) that "Done" could be printed twice. If, however, we swap the order of statements in the `Go` method, the odds of "Done" being printed twice go up dramatically:

```
static void Go()
{
  if (!done) { Console.WriteLine ("Done"); done = true; }
}
Done
Done   (usually!)
```

The problem is that one thread can be evaluating the `if` statement right as the other thread is executing the `WriteLine` statement — before it's had a chance to set `done` to true.
The remedy is to obtain an exclusive lock while reading and writing to the common field. C# provides the lock statement for just this purpose:

```
class ThreadSafe
{
  static bool done;
  static readonly object locker = new object();

  static void Main()
  {
    new Thread (Go).Start();
    Go();
  }

  static void Go()
  {
    lock (locker)
    {
      if (!done) { Console.WriteLine ("Done"); done = true; }
    }
  }
}
```

When two threads simultaneously contend a lock (in this case, `locker`), one thread waits, or blocks, until the lock becomes available. In this case, it ensures only one thread can enter the critical section of code at a time, and "Done" will be printed just once. Code that's protected in such a manner — from indeterminacy in a multithreading context — is called thread-safe.

Shared data is the primary cause of complexity and obscure errors in multithreading. Although often essential, it pays to keep it as simple as possible.
A thread, while *blocked*, doesn't consume CPU resources.

Join and Sleep

You can wait for another thread to end by calling its `Join` method. For example:

```
static void Main()
{
  Thread t = new Thread (Go);
  t.Start();
  t.Join();
  Console.WriteLine ("Thread t has ended!");
}

static void Go()
{
  for (int i = 0; i < 1000; i++) Console.Write ("y");
}
```

This prints "y" 1,000 times, followed by "Thread t has ended!" immediately afterward. You can include a timeout when calling `Join`, either in milliseconds or as a `TimeSpan`. It then returns `true` if the thread ended or `false` if it timed out.

`Thread.Sleep` pauses the current thread for a specified period:

```
Thread.Sleep (TimeSpan.FromHours (1));  // sleep for 1 hour
Thread.Sleep (500);                     // sleep for 500 milliseconds
```

While waiting on a `Sleep` or `Join`, a thread is *blocked* and so does not consume CPU resources.
`Thread.Sleep(0)` relinquishes the thread's current time slice immediately, voluntarily handing over the CPU to other threads. Framework 4.0's new `Thread.Yield()` method does the same thing — except that it relinquishes only to threads running on the *same* processor.
`Sleep(0)` or `Yield` is occasionally useful in production code for advanced performance tweaks. It's also an excellent diagnostic tool for helping to uncover thread safety issues: if inserting `Thread.Yield()` anywhere in your code makes or breaks the program, you almost certainly have a bug.

How Threading Works

Multithreading is managed internally by a thread scheduler, a function the CLR typically delegates to the operating system. A thread scheduler ensures all active threads are allocated appropriate execution time, and that threads that are waiting or blocked (for instance, on an exclusive lock or on user input) do not consume CPU time.

On a single-processor computer, a thread scheduler performs *time-slicing* — rapidly switching execution between each of the active threads. Under Windows, a time-slice is typically in the tens-of-milliseconds region — much larger than the CPU overhead in actually switching context between one thread and another (which is typically in the few-microseconds region).

On a multi-processor computer, multithreading is implemented with a mixture of time-slicing and genuine concurrency, where different threads run code simultaneously on different CPUs. It's almost certain there will still be some time-slicing, because of the operating system's need to service its own threads — as well as those of other applications.

A thread is said to be *preempted* when its execution is interrupted due to an external factor such as time-slicing. In most situations, a thread has no control over when and where it's preempted.

Threads vs Processes

A thread is analogous to the operating system process in which your application runs. Just as processes run in parallel on a computer, threads run in parallel *within a single process*. Processes are fully isolated from each other; threads have just a limited degree of isolation. In particular, threads share (heap) memory with other threads running in the same application. This, in part, is why threading is useful: one thread can fetch data in the background, for instance, while another thread can display the data as it arrives.

Threading's Uses and Misuses

Multithreading has many uses; here are the most common:

Maintaining a responsive user interface

> By running time-consuming tasks on a parallel "worker" thread, the main UI thread is free to continue processing keyboard and mouse events.

Making efficient use of an otherwise blocked CPU

> Multithreading is useful when a thread is awaiting a response from another computer or piece of hardware. While one thread is blocked while performing the task, other threads can take advantage of the otherwise unburdened computer.

Parallel programming

> Code that performs intensive calculations can execute faster on multicore or multiprocessor computers if the workload is shared among multiple threads in a "divide-and-conquer" strategy (see Part 5).

Speculative execution

> On multicore machines, you can sometimes improve performance by predicting something that might need to be done, and then doing it ahead of time. LINQPad uses this technique to speed up the creation of new queries. A variation is to run a number of different algorithms in parallel that all solve the same task. Whichever one finishes first "wins" — this is effective when you can't know ahead of time which algorithm will execute fastest.

Allowing requests to be processed simultaneously

> On a server, client requests can arrive concurrently and so need to be handled in parallel (the .NET Framework creates threads for this automatically if you use ASP.NET, WCF, Web Services, or Remoting). This can also be useful on a client (e.g., handling peer-to-peer networking — or even multiple requests from the user).

With technologies such as ASP.NET and WCF, you may be unaware that multithreading is even taking place — unless you access shared data (perhaps via static fields) without appropriate locking, running afoul of thread safety.

Threads also come with strings attached. The biggest is that multithreading can increase complexity. Having lots of threads does not in and of itself create much complexity; it's the interaction between threads (typically via shared data) that does. This applies whether or not the interaction is intentional, and can cause long development cycles and an ongoing susceptibility to intermittent and nonreproducible bugs. For this reason, it pays to keep interaction to a minimum, and to stick to simple and proven designs wherever possible. This article focuses largely on dealing with just these complexities; remove the interaction and there's much less to say!

A good strategy is to encapsulate multithreading logic into reusable classes that can be independently examined and tested. The Framework itself offers many higher-level threading constructs, which we cover later.

Threading also incurs a resource and CPU cost in scheduling and switching threads (when there are more active threads than CPU cores) — and there's also a creation/tear-down cost.

Multithreading will not always speed up your application — it can even slow it down if used excessively or inappropriately. For example, when heavy disk I/O is involved, it can be faster to have a couple of worker threads run tasks in sequence than to have 10 threads executing at once. (In Signaling with Wait and Pulse, we describe how to implement a producer/consumer queue, which provides just this functionality.)

Creating and Starting Threads

As we saw in the introduction, threads are created using the `Thread` class's constructor, passing in a `ThreadStart` delegate which indicates where execution should begin. Here's how the `ThreadStart` delegate is defined:

```
public delegate void ThreadStart();
```

Calling `Start` on the thread then sets it running. The thread continues until its method returns, at which point the thread ends. Here's an example, using the expanded C# syntax for creating a `TheadStart` delegate:

```
class ThreadTest
{
  static void Main()
  {
    Thread t = new Thread (new ThreadStart (Go));

    t.Start();    // Run Go() on the new thread.
    Go();         // Simultaneously run Go() in the main thread.
  }

  static void Go()
  {
    Console.WriteLine ("hello!");
  }
}
```

In this example, thread `t` executes `Go()` — at (much) the same time the main thread calls `Go()`. The result is two near-instant hellos.

A thread can be created more conveniently by specifying just a method group — and allowing C# to infer the `ThreadStart` delegate:

```
Thread t = new Thread (Go);    // No need to explicitly use ThreadStart
```

Another shortcut is to use a lambda expression or anonymous method:

```
static void Main()
{
  Thread t = new Thread ( () => Console.WriteLine ("Hello!") );
  t.Start();
}
```

Passing Data to a Thread

The easiest way to pass arguments to a thread's target method is to execute a lambda expression that calls the method with the desired arguments:

```
static void Main()
{
  Thread t = new Thread ( () => Print ("Hello from t!") );
  t.Start();
}

static void Print (string message)
{
  Console.WriteLine (message);
}
```

With this approach, you can pass in any number of arguments to the method. You can even wrap the entire implementation in a multi-statement lambda:

```
new Thread (() =>
{
  Console.WriteLine ("I'm running on another thread!");
  Console.WriteLine ("This is so easy!");
}).Start();
```

You can do the same thing almost as easily in C# 2.0 with anonymous methods:

```
new Thread (delegate()
{
  ...
}).Start();
```

Another technique is to pass an argument into `Thread`'s `Start` method:

```
static void Main()
{
  Thread t = new Thread (Print);
  t.Start ("Hello from t!");
}

static void Print (object messageObj)
{
  string message = (string) messageObj;   // We need to cast here
  Console.WriteLine (message);
}
```

This works because `Thread`'s constructor is overloaded to accept either of two delegates:

```
public delegate void ThreadStart();
public delegate void ParameterizedThreadStart (object obj);
```

The limitation of `ParameterizedThreadStart` is that it accepts only one argument. And because it's of type `object`, it usually needs to be cast.

Lambda expressions and captured variables

As we saw, a lambda expression is the most powerful way to pass data to a thread. However, you must be careful about accidentally modifying *captured variables* after starting the thread, because these variables are shared. For instance, consider the following:

```
for (int i = 0; i < 10; i++)
  new Thread (() => Console.Write (i)).Start();
```

The output is nondeterministic! Here's a typical result:

```
0223557799
```

The problem is that the `i` variable refers to the *same* memory location throughout the loop's lifetime. Therefore, each thread calls `Console.Write` on a variable whose value may change as it is running! This is analogous to the problem we describe in "Captured Variables" in Chapter 8 of C# 4.0 in a Nutshell. The problem is less about multithreading and more about C#'s rules for capturing variables (which are somewhat undesirable in the case of `for` and `foreach` loops).

The solution is to use a temporary variable as follows:

```
for (int i = 0; i < 10; i++)
{
  int temp = i;
  new Thread (() => Console.Write (temp)).Start();
}
```

Variable `temp` is now local to each loop iteration. Therefore, each thread captures a different memory location and there's no problem. We can illustrate the problem in the earlier code more simply with the following example:

```
string text = "t1";
Thread t1 = new Thread ( () => Console.WriteLine (text) );

text = "t2";
Thread t2 = new Thread ( () => Console.WriteLine (text) );

t1.Start();
t2.Start();
```

Because both lambda expressions capture the same `text` variable, `t2` is printed twice:

```
t2
t2
```

Naming Threads

Each thread has a `Name` property that you can set for the benefit of debugging. This is particularly useful in Visual Studio, since the thread's name is displayed in the Threads Window and Debug Location toolbar. You can set a thread's name just once; attempts to change it later will throw an exception.

The static `Thread.CurrentThread` property gives you the currently executing thread. In the following example, we set the main thread's name:

```
class ThreadNaming
{
  static void Main()
  {
    Thread.CurrentThread.Name = "main";
    Thread worker = new Thread (Go);
    worker.Name = "worker";
    worker.Start();
    Go();
  }

  static void Go()
  {
    Console.WriteLine ("Hello from " + Thread.CurrentThread.Name);
  }
}
```

Foreground and Background Threads

By default, threads you create explicitly are *foreground threads*. Foreground threads keep the application alive for as long as any one of them is running, whereas *background threads* do not. Once all foreground threads finish, the application ends, and any background threads still running abruptly terminate.

A thread's foreground/background status has no relation to its priority or allocation of execution time.
You can query or change a thread's background status using its `IsBackground` property. Here's an example:

```
class PriorityTest
{
  static void Main (string[] args)
  {
    Thread worker = new Thread ( () => Console.ReadLine() );
    if (args.Length > 0) worker.IsBackground = true;
    worker.Start();
  }
}
```

If this program is called with no arguments, the worker thread assumes foreground status and will wait on the `ReadLine` statement for the user to press Enter. Meanwhile, the main thread exits, but the application keeps running because a foreground thread is still alive.

On the other hand, if an argument is passed to `Main()`, the worker is assigned background status, and the program exits almost immediately as the main thread ends (terminating the `ReadLine`).

When a process terminates in this manner, any `finally` blocks in the execution stack of background threads are circumvented. This is a problem if your program employs `finally` (or `using`) blocks to perform cleanup work such as releasing resources or deleting temporary files. To avoid this, you can explicitly wait out such background threads upon exiting an application. There are two ways to accomplish this:

- If you've created the thread yourself, call `Join` on the thread.
- If you're on a pooled thread, use an event wait handle.

In either case, you should specify a timeout, so you can abandon a renegade thread should it refuse to finish for some reason. This is your backup exit strategy: in the end, you want your application to close — without the user having to enlist help from the Task Manager!

If a user uses the Task Manager to forcibly end a .NET process, all threads "drop dead" as though they were background threads. This is observed rather than documented behavior, and it could vary depending on the CLR and operating system version.

Foreground threads don't require this treatment, but you must take care to avoid bugs that could cause the thread not to end. A common cause for applications failing to exit properly is the presence of active foreground threads.

Thread Priority

A thread's `Priority` property determines how much execution time it gets relative to other active threads in the operating system, on the following scale:

```
enum ThreadPriority { Lowest, BelowNormal, Normal, AboveNormal, Highest }
```

This becomes relevant only when multiple threads are simultaneously active.
Think carefully before elevating a thread's priority — it can lead to problems such as resource starvation for other threads.

Elevating a thread's priority doesn't make it capable of performing real-time work, because it's still throttled by the application's process priority. To perform real-time work, you must also elevate the process priority using the `Process` class in `System.Diagnostics` (we didn't tell you how to do this):

```
using (Process p = Process.GetCurrentProcess())
  p.PriorityClass = ProcessPriorityClass.High;
```

`ProcessPriorityClass.High` is actually one notch short of the highest priority: `Realtime`. Setting a process priority to `Realtime` instructs the OS that you never want the process to yield CPU time to another process. If your program enters an accidental infinite loop, you might find even the operating system locked out, with nothing short of the power button left to rescue you! For this reason, `High` is usually the best choice for real-time applications.

If your real-time application has a user interface, elevating the process priority gives screen updates excessive CPU time, slowing down the entire computer (particularly if the UI is complex). Lowering the main thread's priority in conjunction with raising the process's priority ensures that the real-time thread doesn't get preempted by screen redraws, but doesn't solve the problem of starving other applications of CPU time, because the operating system will still allocate disproportionate resources to the process as a whole. An ideal solution is to have the real-time worker and user interface run as separate applications with different process priorities, communicating via Remoting or memory-mapped files. Memory-mapped files are ideally suited to this task; we explain how they work in Chapters 14 and 25 of C# 4.0 in a Nutshell.

Even with an elevated process priority, there's a limit to the suitability of the managed environment in handling hard real-time requirements. In addition to the issues of latency introduced by automatic garbage collection, the operating system may present additional challenges — even for unmanaged applications — that are best solved with dedicated hardware or a specialized real-time platform.

Exception Handling

Any `try`/`catch`/`finally` blocks in scope when a thread is created are of no relevance to the thread when it starts executing. Consider the following program:

```
public static void Main()
{
  try
  {
    new Thread (Go).Start();
  }
  catch (Exception ex)
  {
    // We'll never get here!
    Console.WriteLine ("Exception!");
  }
}

static void Go() { throw null; }   // Throws a NullReferenceException
```

The `try/catch` statement in this example is ineffective, and the newly created thread will be encumbered with an unhandled `NullReferenceException`. This behavior makes sense when you consider that each thread has an independent execution path.

The remedy is to move the exception handler into the `Go` method:
public static void Main()

```
{
   new Thread (Go).Start();
}

static void Go()
{
  try
  {
    // ...
    throw null;    // The NullReferenceException will get caught below
    // ...
  }
  catch (Exception ex)
  {
    // Typically log the exception, and/or signal another thread
    // that we've come unstuck
    // ...
  }
}
```

You need an exception handler on all thread entry methods in production applications — just as you do (usually at a higher level, in the execution stack) on your main thread. An unhandled exception causes the whole application to shut down. With an ugly dialog!

In writing such exception handling blocks, rarely would you *ignore* the error: typically, you'd log the details of the exception, and then perhaps display a dialog allowing the user to automatically submit those details to your web server. You then might shut down the application — because it's possible that the error corrupted the program's state. However, the cost of doing so is that the user will lose his recent work — open documents, for instance.

The "global" exception handling events for WPF and Windows Forms applications (`Application.DispatcherUnhandledException` and `Application.ThreadException`) fire only for exceptions thrown on the main UI thread. You still must handle exceptions on worker threads manually.

`AppDomain.CurrentDomain.UnhandledException` fires on any unhandled exception, but provides no means of preventing the application from shutting down afterward.
There are, however, some cases where you don't need to handle exceptions on a worker thread, because the .NET Framework does it for you. These are covered in upcoming sections, and are:

- Asynchronous delegates
- `BackgroundWorker`
- The Task Parallel Library (conditions apply)

Thread Pooling

Whenever you start a thread, a few hundred microseconds are spent organizing such things as a fresh private local variable stack. Each thread also consumes (by default) around 1 MB of memory. The *thread pool* cuts these overheads by sharing and recycling threads, allowing multithreading to be applied at a very granular level without a performance penalty. This is useful when leveraging multicore processors to execute computationally intensive code in parallel in "divide-and-conquer" style.

The thread pool also keeps a lid on the total number of worker threads it will run simultaneously. Too many active threads throttle the operating system with administrative burden and render CPU caches ineffective. Once a limit is reached, jobs queue up and start only when another finishes. This makes arbitrarily concurrent applications possible, such as a web server. (The *asynchronous method pattern* is an advanced technique that takes this further by making highly efficient use of the pooled threads; we describe this in Chapter 23 of C# 4.0 in a Nutshell).
There are a number of ways to enter the thread pool:

- Via the Task Parallel Library (from Framework 4.0)
- By calling `ThreadPool.QueueUserWorkItem`
- Via asynchronous delegates
- Via `BackgroundWorker`

The following constructs use the thread pool *indirectly*:
- WCF, Remoting, ASP.NET, and ASMX Web Services application servers
- `System.Timers.Timer` and `System.Threading.Timer`
- Framework methods that end in *Async*, such as those on `WebClient` (the *event-based asynchronous pattern*), and most `Begin`XXX methods (the *asynchronous programming model* pattern)
- PLINQ

The *Task Parallel Library* (TPL) and PLINQ are sufficiently powerful and high-level that you'll want to use them to assist in multithreading even when thread pooling is unimportant. We discuss these in detail in Part 5; right now, we'll look briefly at how you can use the `Task` class as a simple means of running a delegate on a pooled thread.

There are a few things to be wary of when using pooled threads:
- You cannot set the `Name` of a pooled thread, making debugging more difficult (although you can attach a description when debugging in Visual Studio's Threads window).
- Pooled threads are always *background threads* (this is usually not a problem).
- Blocking a pooled thread may trigger additional latency in the early life of an application unless you call `ThreadPool.SetMinThreads` (see Optimizing the Thread Pool).

You are free to change the priority of a pooled thread — it will be restored to normal when released back to the pool.
You can query if you're currently executing on a pooled thread via the property `Thread.CurrentThread.IsThreadPoolThread`.

Entering the Thread Pool via TPL

You can enter the thread pool easily using the `Task` classes in the Task Parallel Library. The `Task` classes were introduced in Framework 4.0: if you're familiar with the older constructs, consider the nongeneric `Task` class a replacement for `ThreadPool.QueueUserWorkItem`, and the generic

`Task<TResult>` a replacement for asynchronous delegates. The newer constructs are faster, more convenient, and more flexible than the old.

To use the nongeneric `Task` class, call `Task.Factory.StartNew`, passing in a delegate of the target method:

```
static void Main()    // The Task class is in System.Threading.Tasks
{
  Task.Factory.StartNew (Go);
}

static void Go()
{
  Console.WriteLine ("Hello from the thread pool!");
}
```

`Task.Factory.StartNew` returns a `Task` object, which you can then use to monitor the task — for instance, you can wait for it to complete by calling its `Wait` method.

Any unhandled exceptions are conveniently rethrown onto the host thread when you call a task's `Wait method`. (If you don't call `Wait` and instead abandon the task, an unhandled exception will shut down the process as with an ordinary thread.)

The generic `Task<TResult>` class is a subclass of the nongeneric `Task`. It lets you get a return value back from the task after it finishes executing. In the following example, we download a web page using `Task<TResult>`:

```
static void Main()
{
  // Start the task executing:
  Task<string> task = Task.Factory.StartNew<string>
    ( () => DownloadString ("http://www.linqpad.net") );

  // We can do other work here and it will execute in parallel:
  RunSomeOtherMethod();

  // When we need the task's return value, we query its Result property:
  // If it's still executing, the current thread will now block (wait)
  // until the task finishes:
  string result = task.Result;
}

static string DownloadString (string uri)
{
  using (var wc = new System.Net.WebClient())
    return wc.DownloadString (uri);
}
```

(The `<string>` type argument highlighted is for clarity: it would be *inferred* if we omitted it.)
Any unhandled exceptions are automatically rethrown when you query the task's `Result` property, wrapped in an `AggregateException`. However, if you fail to query its `Result` property (and don't call `Wait`) any unhandled exception will take the process down.
The Task Parallel Library has many more features, and is particularly well suited to leveraging multicore processors. We'll resume our discussion of TPL in Part 5.

Entering the Thread Pool Without TPL

You can't use the Task Parallel Library if you're targeting an earlier version of the .NET Framework (prior to 4.0). Instead, you must use one of the older constructs for entering the thread pool: `ThreadPool.QueueUserWorkItem` and asynchronous delegates. The difference between the two is that asynchronous delegates let you return data from the thread. Asynchronous delegates also marshal any exception back to the caller.

QueueUserWorkItem

To use `QueueUserWorkItem`, simply call this method with a delegate that you want to run on a pooled thread:

```
static void Main()
{
  ThreadPool.QueueUserWorkItem (Go);
  ThreadPool.QueueUserWorkItem (Go, 123);
  Console.ReadLine();
}

static void Go (object data)   // data will be null with the first call.
{
  Console.WriteLine ("Hello from the thread pool! " + data);
}
```

```
Hello from the thread pool!
Hello from the thread pool! 123
```

Our target method, `Go`, must accept a single `object` argument (to satisfy the `WaitCallback` delegate). This provides a convenient way of passing data to the method, just like with `ParameterizedThreadStart`. Unlike with `Task`, `QueueUserWorkItem` doesn't return an object to help you subsequently manage execution. Also, you must explicitly deal with exceptions in the target code — unhandled exceptions will take down the program.

Asynchronous delegates

`ThreadPool.QueueUserWorkItem` doesn’t provide an easy mechanism for getting return values back from a thread after it has finished executing. Asynchronous delegate invocations (asynchronous delegates for short) solve this, allowing any number of typed arguments to be passed in both directions. Furthermore, unhandled exceptions on asynchronous delegates are conveniently rethrown on the original thread (or more accurately, the thread that calls `EndInvoke`), and so they don’t need explicit handling.

Don’t confuse asynchronous delegates with asynchronous methods (methods starting with *Begin* or *End*, such as `File.BeginRead`/`File.EndRead`). Asynchronous methods follow a similar protocol outwardly, but they exist to solve a much harder problem, which we describe in Chapter 23 of C# 4.0 in a Nutshell.

Here’s how you start a worker task via an asynchronous delegate:

1. Instantiate a delegate targeting the method you want to run in parallel (typically one of the predefined `Func` delegates).
2. Call `BeginInvoke` on the delegate, saving its `IAsyncResult` return value.

   `BeginInvoke` returns immediately to the caller. You can then perform other activities while the pooled thread is working.
3. When you need the results, call `EndInvoke` on the delegate, passing in the saved `IAsyncResult` object.

In the following example, we use an asynchronous delegate invocation to execute concurrently with the main thread, a simple method that returns a string’s length:

```
static void Main()
{
  Func<string, int> method = Work;
  IAsyncResult cookie = method.BeginInvoke ("test", null, null);
  //
  // ... here's where we can do other work in parallel...
  //
  int result = method.EndInvoke (cookie);
  Console.WriteLine ("String length is: " + result);
}
```

```
static int Work (string s) { return s.Length; }
```

`EndInvoke` does three things. First, it waits for the asynchronous delegate to finish executing, if it hasn't already. Second, it receives the return value (as well as any `ref` or `out` parameters). Third, it throws any unhandled worker exception back to the calling thread.

If the method you're calling with an asynchronous delegate has no return value, you are still (technically) obliged to call `EndInvoke`. In practice, this is open to debate; there are no `EndInvoke` police to administer punishment to noncompliers! If you choose not to call `EndInvoke`, however, you'll need to consider exception handling on the worker method to avoid silent failures.
You can also specify a callback delegate when calling `BeginInvoke` — a method accepting an `IAsyncResult` object that's automatically called upon completion. This allows the instigating thread to "forget" about the asynchronous delegate, but it requires a bit of extra work at the callback end:

```
static void Main()
{
  Func<string, int> method = Work;
  method.BeginInvoke ("test", Done, method);
  // ...
  //
}

static int Work (string s) { return s.Length; }

static void Done (IAsyncResult cookie)
{
  var target = (Func<string, int>) cookie.AsyncState;
  int result = target.EndInvoke (cookie);
  Console.WriteLine ("String length is: " + result);
}
```

The final argument to `BeginInvoke` is a user state object that populates the `AsyncState` property of `IAsyncResult`. It can contain anything you like; in this case, we're using it to pass the `method` delegate to the completion callback, so we can call `EndInvoke` on it.

Optimizing the Thread Pool

The thread pool starts out with one thread in its pool. As tasks are assigned, the pool manager "injects" new threads to cope with the extra concurrent workload, up to a maximum limit. After a sufficient period of inactivity, the pool manager may "retire" threads if it suspects that doing so will lead to better throughput.

You can set the upper limit of threads that the pool will create by calling `ThreadPool.SetMaxThreads`; the defaults are:

- 1023 in Framework 4.0 in a 32-bit environment
- 32768 in Framework 4.0 in a 64-bit environment
- 250 per core in Framework 3.5
- 25 per core in Framework 2.0

(These figures may vary according to the hardware and operating system.) The reason there are that many is to ensure progress should some threads be blocked (idling while awaiting some condition, such as a response from a remote computer).

You can also set a lower limit by calling `ThreadPool.SetMinThreads`. The role of the lower limit is subtler: it's an advanced optimization technique that instructs the pool manager not to *delay* in the allocation of threads until reaching the lower limit. Raising the minimum thread count improves concurrency when there are blocked threads (see sidebar).

The default lower limit is one thread per processor core — the minimum that allows full CPU utilization. On server environments, though (such ASP.NET under IIS), the lower limit is typically much higher — as much as 50 or more.

How Does the Minimum Thread Count Work?
Increasing the thread pool's minimum thread count to *x* doesn't actually force *x* threads to be created right away — threads are created only on demand. Rather, it instructs the pool manager to create up to *x* threads the *instant* they are required. The question, then, is why would the thread pool otherwise delay in creating a thread when it's needed?

The answer is to prevent a brief burst of short-lived activity from causing a full allocation of threads, suddenly swelling an application's memory footprint. To illustrate, consider a quad-core computer running a client application that enqueues 40 tasks at once. If each task performs a 10 ms calculation, the whole thing will be over in 100 ms, assuming the work is divided among the four cores. Ideally, we'd want the 40 tasks to run on *exactly four threads*:

- Any less and we'd not be making maximum use of all four cores.
- Any more and we'd be wasting memory and CPU time creating unnecessary threads.

And this is exactly how the thread pool works. Matching the thread count to the core count allows a program to retain a small memory footprint without hurting performance — as long as the threads are efficiently used (which in this case they are).
But now suppose that instead of working for 10 ms, each task queries the Internet, waiting half a second for a response while the local CPU is idle. The pool manager's thread-economy strategy breaks down; it would now do better to create more threads, so all the Internet queries could happen simultaneously.

Fortunately, the pool manager has a backup plan. If its queue remains stationary for more than half a second, it responds by creating more threads — one every half-second — up to the capacity of the thread pool.

The half-second delay is a two-edged sword. On the one hand, it means that a one-off burst of brief activity doesn't make a program suddenly consume an extra unnecessary 40 MB (or more) of memory. On the other hand, it can needlessly delay things when a pooled thread blocks, such as when querying a database or calling `WebClient.DownloadFile`. For this reason, you can tell the pool manager not to delay in the allocation of the first *x* threads, by calling `SetMinThreads`, for instance:

```
ThreadPool.SetMinThreads (50, 50);
```

(The second value indicates how many threads to assign to I/O completion ports, which are used by the APM, described in Chapter 23 of C# 4.0 in a Nutshell.)
The default value is one thread per core.

The concept of multithreading

Multithreading is an important concept in programming languages, and C# too, which is how to make the thread of the program running parallelly to each other. For simplicity you see an example below:

HelloThread.cs

?

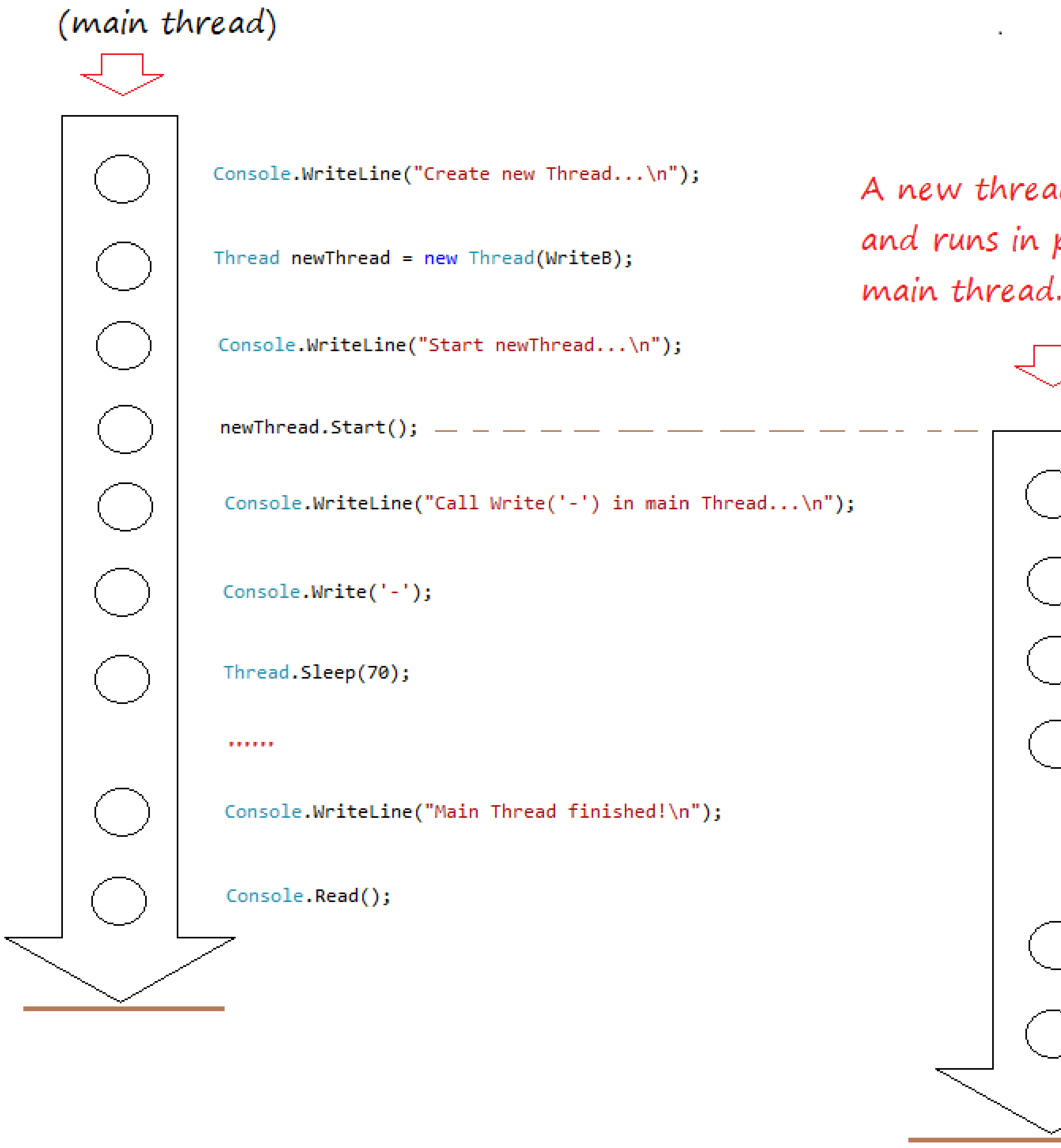
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;
```

```
namespace MultithreadingTutorial
{
    class HelloThread
    {
        public static void Main(string[] args)
        {

            Console.WriteLine("Create new Thread...\n");

            // Creates a child thread, which runs in parallel with the main thread.
            Thread newThread = new Thread(WriteB);

            Console.WriteLine("Start newThread...\n");

            // Start the thread
            newThread.Start();

            Console.WriteLine("Call Write('-') in main Thread...\n");

            // In the main thread print out character '-'
            for (int i = 0; i < 50; i++)
            {
                Console.Write('-');

                // Sleep 70 millisenconds.
                Thread.Sleep(70);
            }


            Console.WriteLine("Main Thread finished!\n");
            Console.Read();
        }



        public static void WriteB()
        {
            // Print out the 'B' 100 times.
            for (int i = 0; i < 100; i++)
            {
                Console.Write('B');

                // Sleep 100 millisenconds
                Thread.Sleep(100);
            }

        }
    }


}
```

And you run this class:

file:///E:/CSHARP_TUTORIAL/MySolution/MultithreadingTutorial/bin/Debug/M...

The operating principle of the thread is explained in the following illustration:



```
Console.WriteLine("Create new Thread...\n");

Thread newThread = new Thread(WriteB);

Console.WriteLine("Start newThread...\n");

newThread.Start();

Console.WriteLine("Call Write('-') in main Thread...\n");

Console.Write('-');

Thread.Sleep(70);

......

Console.WriteLine("Main Thread finished!\n");

Console.Read();
```

2- Pass parameters to Thread

Above you have become familiar with **HelloThread** example, you have created an object wrapped a static method to execute this method in parallel with the main thread.
Static method can become a parameter passed into constructor of the **Thread** class if that method don't have parameters, or with a single parameter object type.
?

```
// A static method, have no parameter.
public static void LetGo()
{
      // ...
}
```

```
// A static method, has only one parameter type of object.
public static void GetGo(object value)
{
      // ...
}
```

This next example, I'll create a **thread** to wrap a static method with one parameter (object type). Running thread and pass value to the parameter.

MyWork.cs

?

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;

namespace MultithreadingTutorial
{
    class MyWork
    {

        public static void DoWork(object ch)
        {
            for (int i = 0; i < 100; i++)
            {

                Console.Write(ch);

                // Sleep 50 milliseconds
                Thread.Sleep(50);
            }
        }



    }
}
```

ThreadParamDemo.cs

?

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;

namespace MultithreadingTutorial
{
    class ThreadParamDemo
    {

        public static void Main(string[] args)
        {

            Console.WriteLine("Create new thread.. \n");

            // Create a Thread object to wrap around the static method MyWork.DoWork
            Thread workThread = new Thread(MyWork.DoWork);

            Console.WriteLine("Start workThread...\n");

            // Run workThread,
            // and pass parameter to MyWork.DoWork.
            workThread.Start("*");


            for (int i = 0; i < 20; i++)
            {
                Console.Write(".");

                // Sleep 30 milliseconds.
                Thread.Sleep(30);
            }

            Console.WriteLine("MainThread ends");
            Console.Read();
        }
    }

}
```

Running **ThreadParamDemo**:

file:///E:/CSHARP_TUTORIAL/MySolution/MultithreadingTutorial/bin/Debug/M...

3- Thread uses non-static method

You can also create a thread to execute (using) a none-static method. See for example:

Worker.cs

?

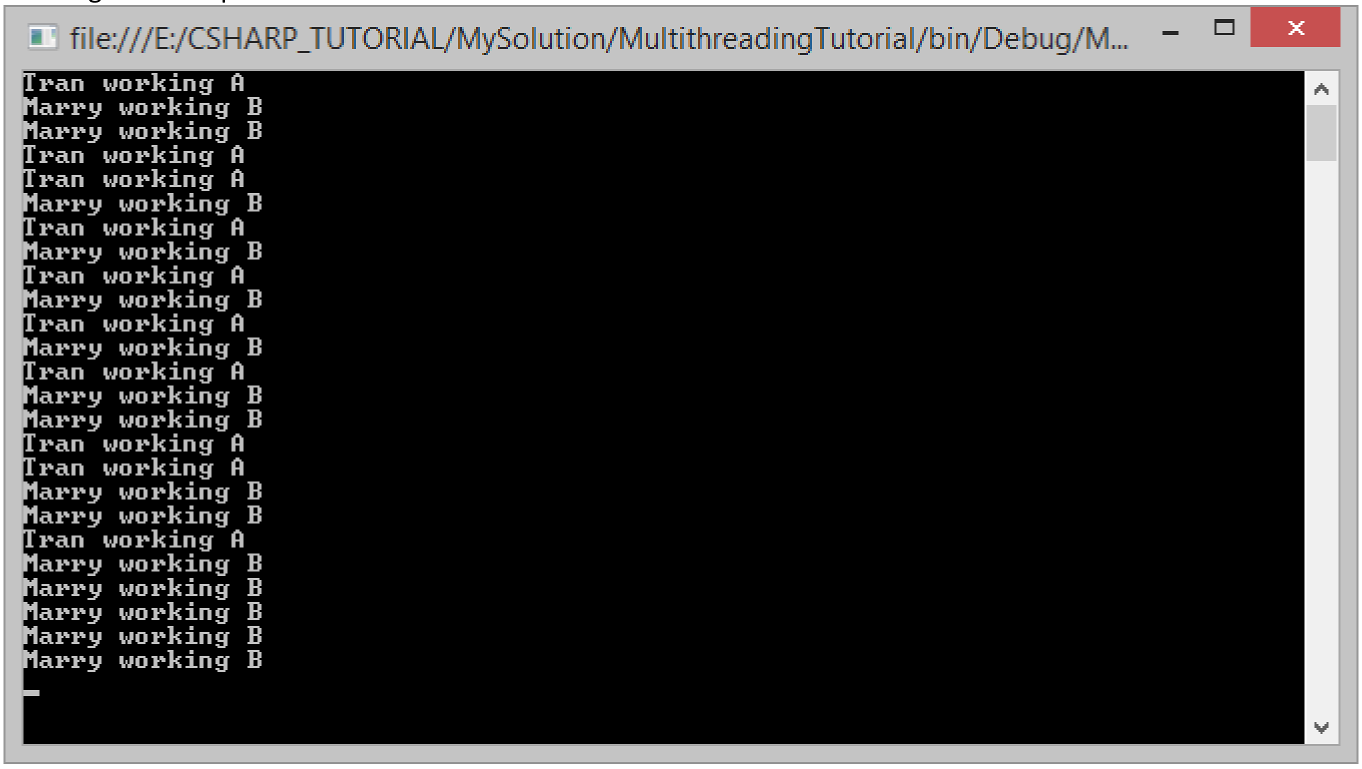
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;

namespace MultithreadingTutorial
{
    class Worker
    {
        private string name;
        private int loop;

        public Worker(string name, int loop)
        {
            this.name = name;
            this.loop = loop;
        }

        public void DoWork(object value)
        {
            for (int i = 0; i < loop; i++)
            {
                Console.WriteLine(name + " working " + value);
                Thread.Sleep(50);
            }

        }

    }

}
```

WorkerTest.cs

?

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;

namespace MultithreadingTutorial
{
    class WorkerTest
    {

        public static void Main(string[] args)
        {
            Worker w1 = new Worker("Tran",10);

            // Create a Thread object.
            Thread workerThread1 = new Thread(w1.DoWork);

            // Pass parameter to DoWork method.
            workerThread1.Start("A");


            Worker w2 = new Worker("Marry",15);
            // Create a Thread object.
            Thread workerThread2 = new Thread(w2.DoWork);

            // Pass parameter to DoWork method.
            workerThread2.Start("B");

            Console.Read();
        }
    }
}
```

Running the example:



file:///E:/CSHARP_TUTORIAL/MySolution/MultithreadingTutorial/bin/Debug/M...

```
Tran working A
Marry working B
Marry working B
Tran working A
Tran working A
Marry working B
Tran working A
Marry working B
Tran working A
Marry working B
Tran working A
Marry working B
Tran working A
Marry working B
Marry working B
Tran working A
Tran working A
Marry working B
Marry working B
Tran working A
Marry working B
Marry working B
Marry working B
Marry working B
Marry working B
_
```

4- ThreadStart Delegate

ThreadStart is a Delegate, it is initiated by wrap a method. And it is passed as a parameter to initialize the **Thread** object.

With **.Net** < 2.0, to start a thread, you need to create **ThreadStart**, which is a delegate.

Beginning in version 2.0 of the **.NET Framework**, it is not necessary to create a delegate explicitly. You only need to specify the name of the method in the **Thread** constructor.

Programmer.cs

?

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;

namespace MultithreadingTutorial
```

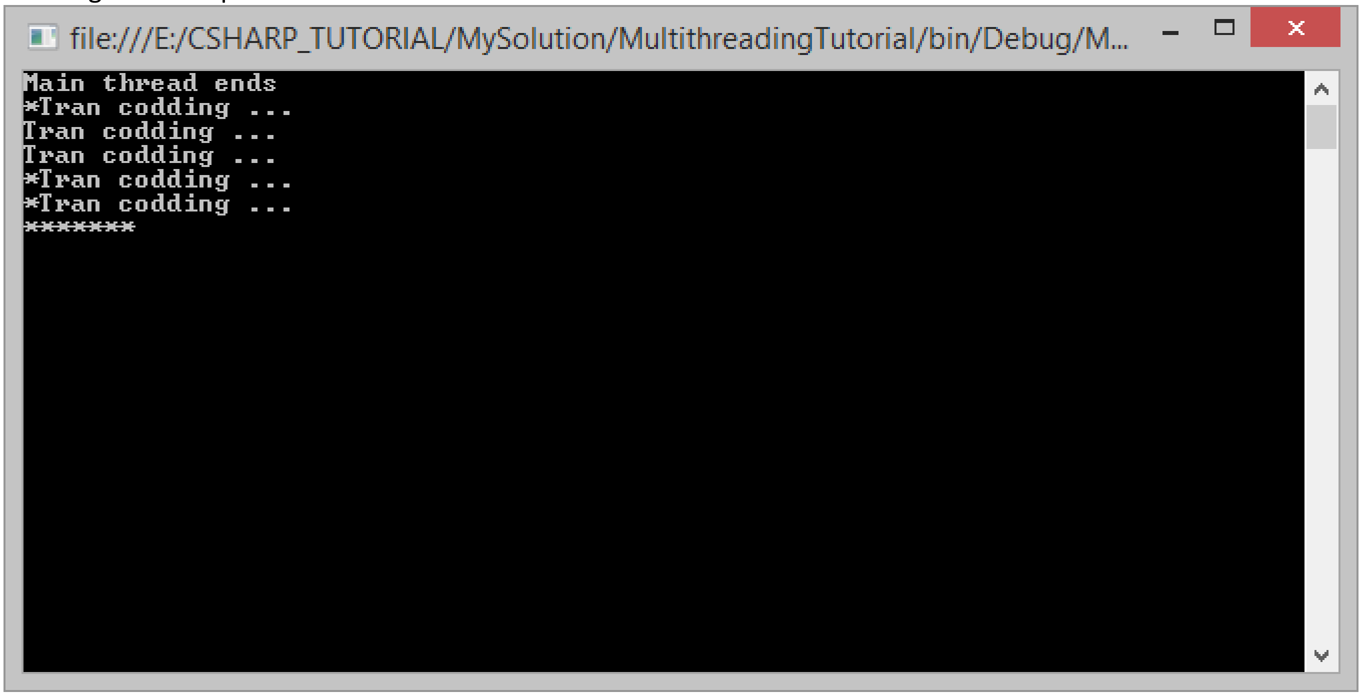
```
{
    class Programmer
    {
        private string name;
        public Programmer(string name)
        {
            this.name= name;
        }

        // This is none static method, no parameters.
        public void DoCode()
        {
            for (int i = 0; i < 5; i++)
            {
                Console.WriteLine(name + " codding ... ");
                Thread.Sleep(50);
            }

        }
    }

}
```

ThreadStartDemo.cs

?

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;


namespace MultithreadingTutorial
{
    class ThreadStartDemo
    {


        public static void Main(string[] args)
        {
            // Create a ThreadStart object that wrap a static method.
            // (It can only wrap method have no parameters)
            // (ThreadStart is a delegate).
            ThreadStart threadStart1 = new ThreadStart(DoWork);

            // Create a thread wrap threadStart1.
            Thread workThread = new Thread(threadStart1);

            // Start thread
            workThread.Start();

            // Create Programmer object.
            Programmer tran = new Programmer("Tran");

            // You can also create a ThreadStart object that wrap a non-
static method.
            // (It can only wrap method have no parameters)
            ThreadStart threadStart2 = new ThreadStart(tran.DoCode);
            // Create a Thread wrap threadStart2.
            Thread progThread = new Thread(threadStart2);

            progThread.Start();

            Console.WriteLine("Main thread ends");
            Console.Read();
        }


        public static void DoWork()
        {
            for (int i = 0; i < 10; i++)
            {
                Console.Write("*");
                Thread.Sleep(100);
            }
        }
    }
}
```

Running the example:



file:///E:/CSHARP_TUTORIAL/MySolution/MultithreadingTutorial/bin/Debug/M...

```
Main thread ends
*Tran codding ...
Tran codding ...
Tran codding ...
*Tran codding ...
*Tran codding ...
*******
```

5- Thread with anonymous code

In the above, you have created Thread using a specific method. You can create a thread to execute any code

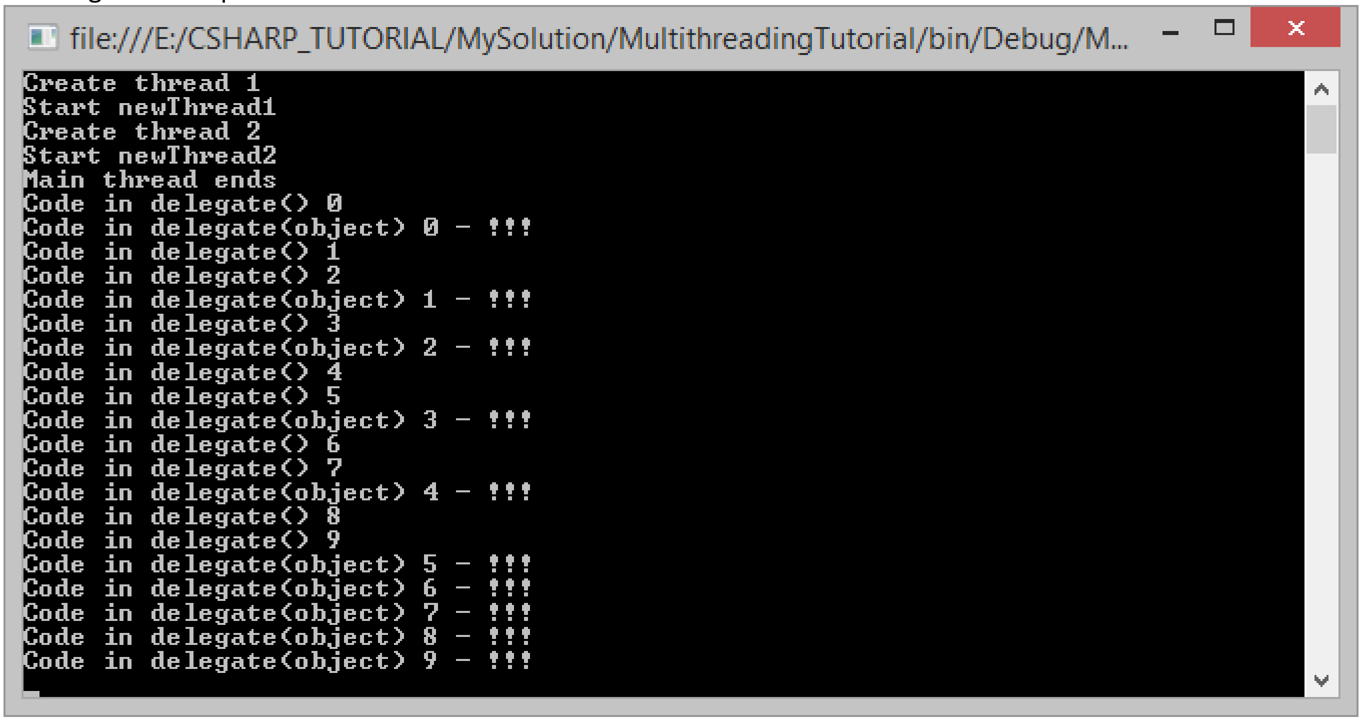
?

```
// Use delegate() to create anonymous method.
delegate()
{
     //  ...
}
```

5
Example:
ThreadUsingSnippetCode.cs
?

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;

namespace MultithreadingTutorial
{
    class ThreadUsingSnippetCode
    {

        public static void Main(string[] args)
        {

            Console.WriteLine("Create thread 1");

            // Creates a thread to execute a snippet code.
            Thread newThread1 = new Thread(
                delegate()
                {
                    for (int i = 0; i < 10; i++)
                    {
                        Console.WriteLine("Code in delegate() " + i);
                        Thread.Sleep(50);
                    }

                }
            );

            Console.WriteLine("Start newThread1");


            // Start thread.
            newThread1.Start();

            Console.WriteLine("Create thread 2");

            // Creates a thread to execute a snippet code.
            Thread newThread2 = new Thread(
                delegate(object value)
                {
                    for (int i = 0; i < 10; i++)
                    {
                        Console.WriteLine("Code in delegate(object) " +
i + " - " + value);
                        Thread.Sleep(100);
                    }

                }
            );

            Console.WriteLine("Start newThread2");

            // Start thread 2.
            // Pass parameter to delegate().
            newThread2.Start("!!!");

            Console.WriteLine("Main thread ends");
            Console.Read();

        }
    }
}
```

Running the example:



file:///E:/CSHARP_TUTORIAL/MySolution/MultithreadingTutorial/bin/Debug/M...

```
Create thread 1
Start newThread1
Create thread 2
Start newThread2
Main thread ends
Code in delegate() 0
Code in delegate(object) 0 - !!!
Code in delegate() 1
Code in delegate() 2
Code in delegate(object) 1 - !!!
Code in delegate() 3
Code in delegate(object) 2 - !!!
Code in delegate() 4
Code in delegate() 5
Code in delegate(object) 3 - !!!
Code in delegate() 6
Code in delegate() 7
Code in delegate(object) 4 - !!!
Code in delegate() 8
Code in delegate() 9
Code in delegate(object) 5 - !!!
Code in delegate(object) 6 - !!!
Code in delegate(object) 7 - !!!
Code in delegate(object) 8 - !!!
Code in delegate(object) 9 - !!!
```

6- Name the thread

In multi-threaded programming, you can name the stream, it is really useful in case of debugging , to know the code that is being executed in which thread.

In a thread, you can call ***Thread.CurrentThread.Name*** to retrieve the name of the thread that take place at that time.

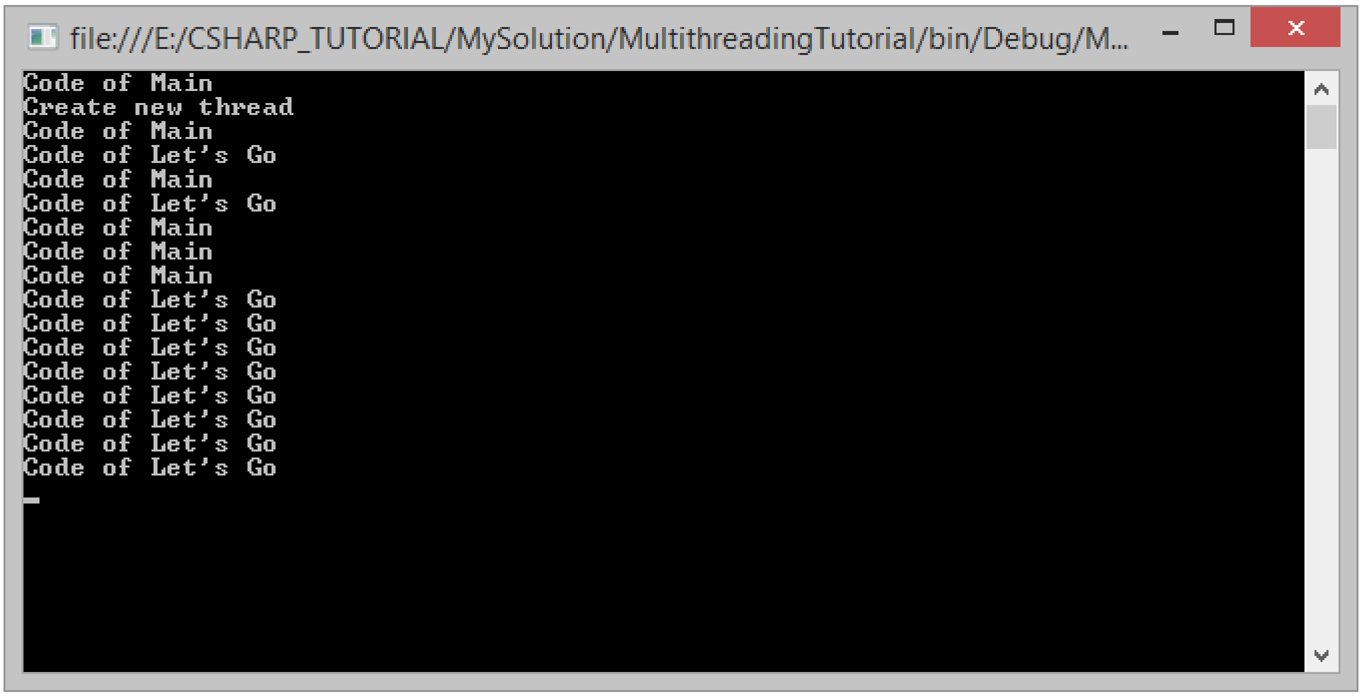
See illustration:

NamingThreadDemo.cs

?

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;

namespace MultithreadingTutorial
{
    class NamingThreadDemo
    {

        public static void Main(string[] args)
        {
            // Set name to current thread
            // (Main thread)
            Thread.CurrentThread.Name = "Main";

            Console.WriteLine("Code of "+ Thread.CurrentThread.Name);

            Console.WriteLine("Create new thread");

            // Create a thread.
            Thread letgoThread = new Thread(LetGo);

            // Set name to thread
            letgoThread.Name = "Let's Go";

            letgoThread.Start();

            for (int i = 0; i < 5; i++)
            {
                Console.WriteLine("Code of " + Thread.CurrentThread.Name);
                Thread.Sleep(30);
            }

            Console.Read();
        }


        public static void LetGo()
        {
            for (int i = 0; i < 10; i++)
            {
                Console.WriteLine("Code of " + Thread.CurrentThread.Name);
                Thread.Sleep(50);
            }
        }
    }

}
```

Running the example:


file:///E:/CSHARP_TUTORIAL/MySolution/MultithreadingTutorial/bin/Debug/M...

```
Code of Main
Create new thread
Code of Main
Code of Let's Go
Code of Main
Code of Let's Go
Code of Main
Code of Main
Code of Main
Code of Let's Go
Code of Let's Go
Code of Let's Go
Code of Let's Go
Code of Let's Go
Code of Let's Go
Code of Let's Go
Code of Let's Go
```

7- Priority between threads

there are 5 priority of a thread in C # , they are defined in the **ThreadPriority** enum .

** ThreadPriority enum **

?

```
enum ThreadPriority {
    Lowest,
    BelowNormal,
    Normal,
    AboveNormal,
    Highest
}
```

Usually with high-speed computers, if the thread only execute little amount of work, you find it very difficult to detect the difference between high-priority threads and threads have low priority.

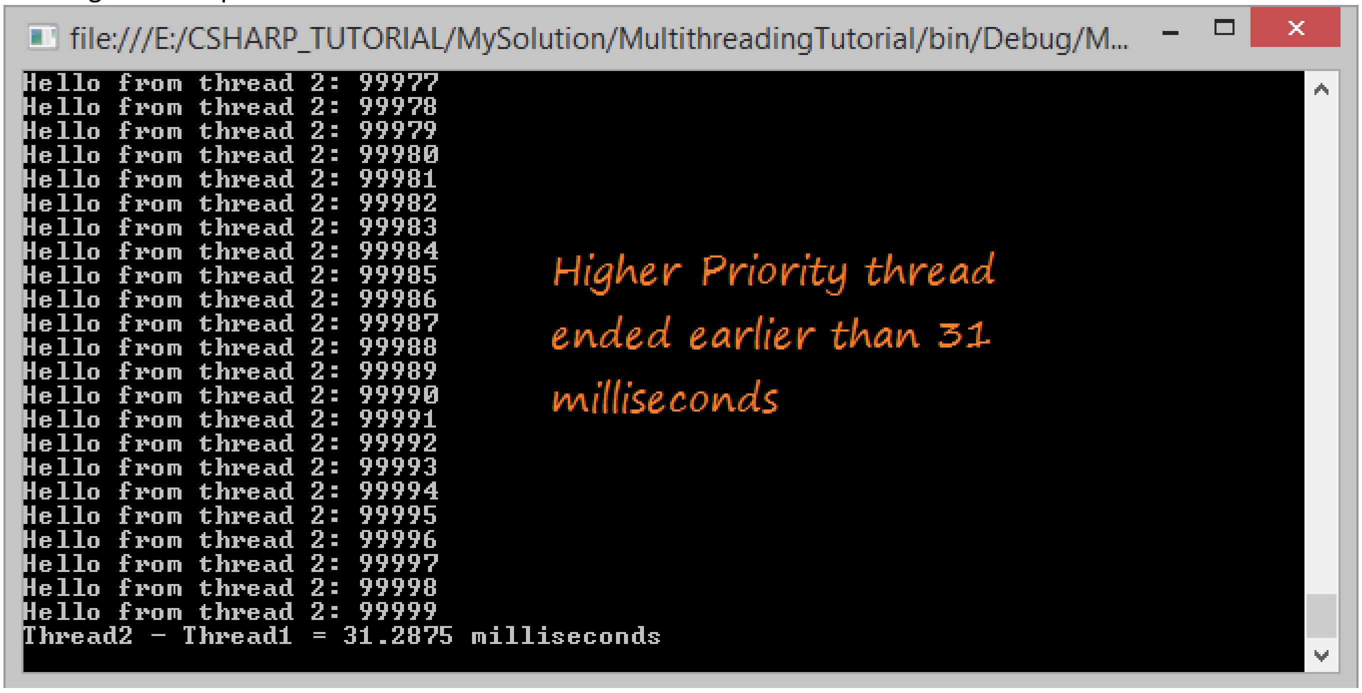
The example below has two threads, each prints the text of 100K lines (numbers large enough to see the difference).

ThreadPriorityDemo.cs

?

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;

namespace MultithreadingTutorial
{
    class ThreadPriorityDemo
    {
        private static DateTime endsDateTime1;
        private static DateTime endsDateTime2;


        public static void Main(string[] args)
        {
            endsDateTime1 = DateTime.Now;
            endsDateTime2 = DateTime.Now;

            Thread thread1 = new Thread(Hello1);

            // Set highest priority for thread1
            thread1.Priority = ThreadPriority.Highest;

            Thread thread2 = new Thread(Hello2);

            // Set the lowest priority for thread2.
            thread2.Priority = ThreadPriority.Lowest;


            thread2.Start(); thread1.Start();

            Console.Read();
        }


        public static void Hello1()
        {
            for (int i = 0; i < 100000; i++)
            {
                Console.WriteLine("Hello from thread 1: "+ i);
            }
            // The time of thread1 ends.
            endsDateTime1 = DateTime.Now;

            PrintInterval();
        }

        public static void Hello2()
        {
            for (int i = 0; i < 100000; i++)
            {
                Console.WriteLine("Hello from thread 2: "+ i);
            }
            // The time of thread2 ends.
            endsDateTime2 = DateTime.Now;

            PrintInterval();
        }

        private static void PrintInterval()
        {
            // Interval (Milliseconds)
            TimeSpan interval = endsDateTime2 - endsDateTime1;


            Console.WriteLine("Thread2 - Thread1 = " +
interval.TotalMilliseconds + " milliseconds");
        }
    }

}
```

Running the example:



file:///E:/CSHARP_TUTORIAL/MySolution/MultithreadingTutorial/bin/Debug/M...

```
Hello from thread 2: 99977
Hello from thread 2: 99978
Hello from thread 2: 99979
Hello from thread 2: 99980
Hello from thread 2: 99981
Hello from thread 2: 99982
Hello from thread 2: 99983
Hello from thread 2: 99984
Hello from thread 2: 99985
Hello from thread 2: 99986
Hello from thread 2: 99987
Hello from thread 2: 99988
Hello from thread 2: 99989
Hello from thread 2: 99990
Hello from thread 2: 99991
Hello from thread 2: 99992
Hello from thread 2: 99993
Hello from thread 2: 99994
Hello from thread 2: 99995
Hello from thread 2: 99996
Hello from thread 2: 99997
Hello from thread 2: 99998
Hello from thread 2: 99999
Thread2 - Thread1 = 31.2875 milliseconds
```

8- Using Join()

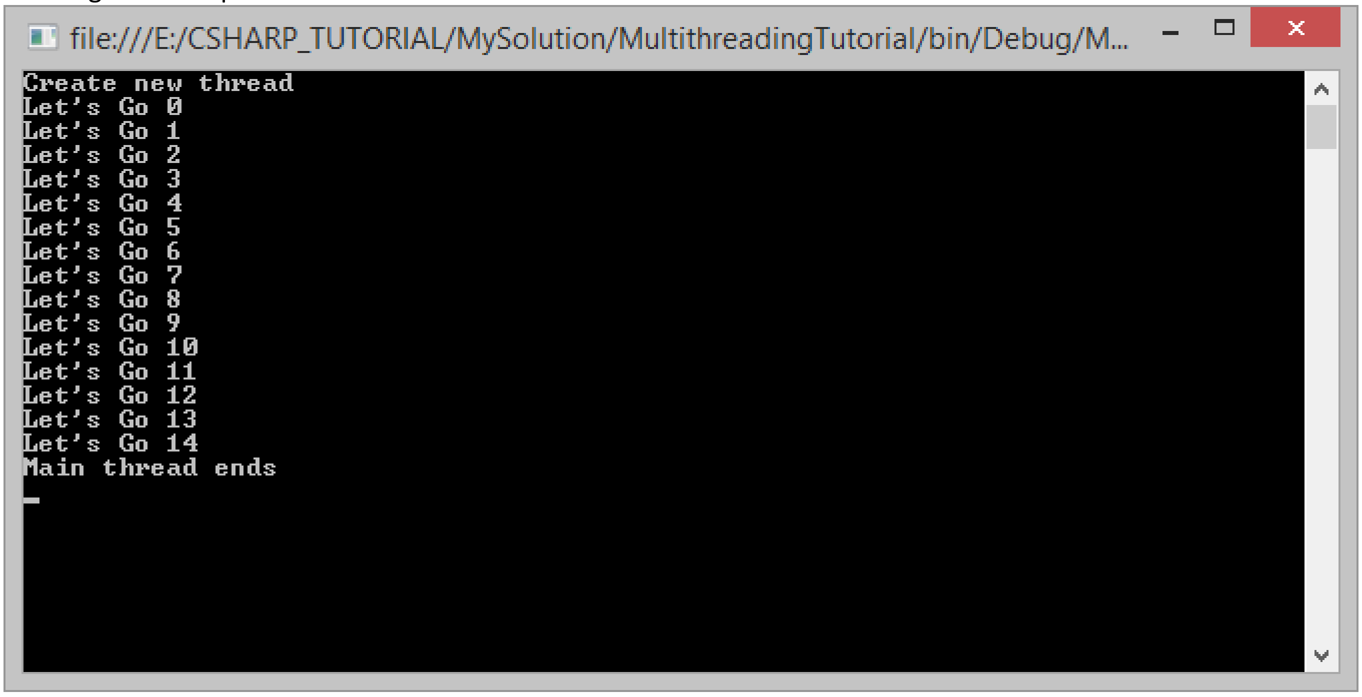
Thread.Join() is a method notifying that please wait for this thread to be completed before the parent thread continues to run.

ThreadJoinDemo.cs

?

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;


namespace MultithreadingTutorial
{
    class ThreadJoinDemo
    {

        public static void Main(string[] args)
        {
            Console.WriteLine("Create new thread");

            Thread letgoThread = new Thread(LetGo);

            // Start Thread.
            letgoThread.Start();

            // Tell to the parent thread (here is main thread)
            // wait for the letgoThread to finish, then continue running.
            letgoThread.Join();

            // This statement must wait for letgoThread to end, then
continue running.
            Console.WriteLine("Main thread ends");
            Console.Read();
        }
        public static void LetGo()
        {
            for (int i = 0; i < 15; i++)
            {
                Console.WriteLine("Let's Go " + i);
            }
        }
    }
}
```

Running the example:



file:///E:/CSHARP_TUTORIAL/MySolution/MultithreadingTutorial/bin/Debug/M...

```
Create new thread
Let's Go 0
Let's Go 1
Let's Go 2
Let's Go 3
Let's Go 4
Let's Go 5
Let's Go 6
Let's Go 7
Let's Go 8
Let's Go 9
Let's Go 10
Let's Go 11
Let's Go 12
Let's Go 13
Let's Go 14
Main thread ends
```

9- Using Yield()

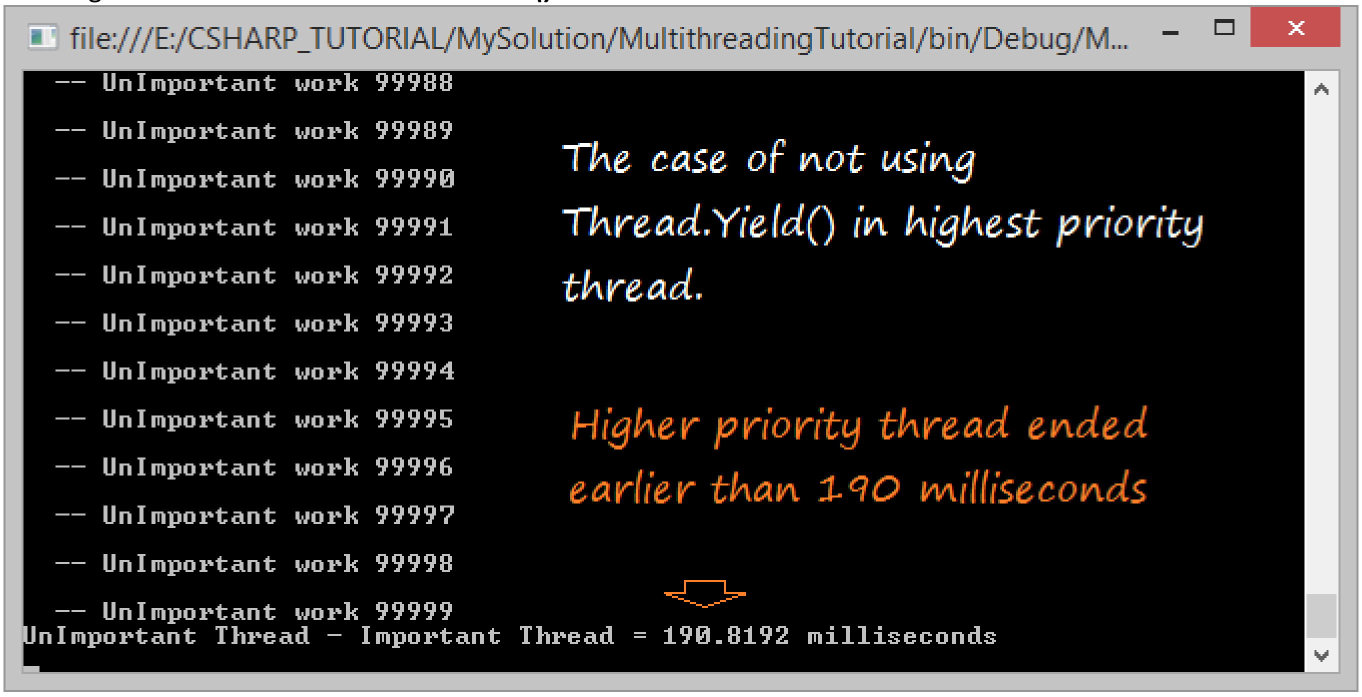
Theoretically, **to 'yield' means to let go, to give up, to surrender**. A yielding thread tells the operator system that it's willing to let other threads be scheduled in its place. This indicates that it's not doing something too critical. Note that *it's only a hint*, though, and not guaranteed to have any effect at all.

So, **Yield()** methods is used when you see that thread is free, it's not doing anything important, it suggests operating system give priority temporarily to the other thread.
The example below, there are two threads, each thread print out a text 100K times (the numbers are large enough to see the difference). One thread is the highest priority, and other thread is lowest priority. See completion time of 2 threads.
ThreadYieldDemo.cs
?

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;
```

```

namespace MultithreadingTutorial
{
    class ThreadYieldDemo
    {

        private static DateTime  importantEndTime;
        private static DateTime  unImportantEndTime;

        public static void Main(string[] args)
        {
            importantEndTime = DateTime.Now;
            unImportantEndTime = DateTime.Now;

            Console.WriteLine("Create thread 1");

            Thread importantThread = new Thread(ImportantWork);

            // Set the highest priority for this thread.
            importantThread.Priority = ThreadPriority.Highest;

            Console.WriteLine("Create thread 2");

            Thread unImportantThread = new Thread(UnImportantWork);

            // Set the lowest priority for this thread.
            unImportantThread.Priority = ThreadPriority.Lowest;

            // Start threads.
            unImportantThread.Start();
            importantThread.Start();


            Console.Read();

        }

        // A important job which requires high priority.
        public static void ImportantWork()
        {
            for (int i = 0; i < 100000; i++)
            {
                Console.WriteLine("\n Important work " + i);

                // Notifying the operating system,
                // this thread gives priority to other threads.
                Thread.Yield();
            }
            // The end time of this thread.
            importantEndTime = DateTime.Now;
            PrintTime();
        }

        public static void UnImportantWork()
        {
            for (int i = 0; i < 100000; i++)
            {
                Console.WriteLine("\n  -- UnImportant work " + i);
            }
            // The end time of this thread.
            unImportantEndTime = DateTime.Now;
            PrintTime();
        }

        private static void PrintTime()
        {
            // Interval (Milliseconds)
            TimeSpan interval = unImportantEndTime - importantEndTime;

            Console.WriteLine("UnImportant Thread - Important Thread =
" + interval.TotalMilliseconds +" milliseconds");
        }

    }
}
```

Running class in the absence of ***Thread.Yield()***:



```
file:///E:/CSHARP_TUTORIAL/MySolution/MultithreadingTutorial/bin/Debug/M...
 -- UnImportant work 99988
 -- UnImportant work 99989
 -- UnImportant work 99990
 -- UnImportant work 99991
 -- UnImportant work 99992
 -- UnImportant work 99993
 -- UnImportant work 99994
 -- UnImportant work 99995
 -- UnImportant work 99996
 -- UnImportant work 99997
 -- UnImportant work 99998
 -- UnImportant work 99999
UnImportant Thread - Important Thread = 190.8192 milliseconds
```

The case of not using Thread.Yield() in highest priority thread.
Higher priority thread ended earlier than 190 milliseconds

Runs the above class in case the higher priority thread continuously calls **Thread.Yield()** to request that the system temporarily give priority to the other threads.

Introduction To Multithreading In C#

This article is a complete introduction to Multithreading in C#. This tutorial explains what a thread in C# is and how C# threading works.

- facebook
- twitter
- linkedIn
- Reddit
  -
    - Email
    - Bookmark

multithreading.ZIP
In working in New York and talking to programmers all over Wall Street, I've noticed a common thread of knowledge expected in most real time programming applications. That knowledge is known as multithreading. As I have migrated around the programming world, and performed interviews on potential programming candidates, it never ceases to amaze me how little is known about multithreading or why or how threading is applied. In a series of excellent articles written by Vance Morrison, MSDN has tried to address this problem: (See the August Issue of MSDN, What every Developer Must Know about Multithreaded Apps, and the October issue Understand the Impact of Low-Lock Techniques in Multithreaded Apps.

In this article I will attempt to give an introductory discussion on threading, why it is used, and how you use it in .NET. I hope to once and for all unveil the mystery behind multithreading, and in explaining it, help avert potential threading disasters in your code.

What is a thread?

Every application runs with at least one thread. So what is a thread? A thread is nothing more than a process. My guess is that the word thread comes from the Greek mythology of supernatural Muses weaving threads on a loom, where each thread represents a path in time of someone's life. If you mess with that thread, then you disturb the fabric of life or change the process of life. On the computer, a thread is a process moving through time. The process performs sets of sequential steps, each step executing a line of code. Because the steps are sequential, each step takes a given amount of time. The time it takes to complete a series of steps is the sum of the time it takes to perform each programming step.

What are multithreaded applications?

For a long time, most programming applications (except for embedded system programs) were single-threaded. That means there was only one thread in the entire application. You could never do computation A until completing computation B. A program starts at step 1 and continues sequentially (step 2, step 3, step 4) until it hits the final step (call it step 10). A multithreaded application allows you to run several threads, each thread running in its own process. So theoretically you can run step 1 in one thread and at the same time run step 2 in another thread. At the same time you could run step 3 in its own thread, and even step 4 in its own thread. Hence step 1, step 2, step 3, and step 4 would run concurrently. Theoretically, if all four steps took about the same time, you could finish your program in a quarter of the time it takes to run a single thread (assuming you had a 4 processor machine). So why isn't every program multithreaded? Because along with speed, you face complexity. Imagine if step 1 somehow depends on the information in step 2. The program might not run correctly if step 1 finishes calculating before step 2 or visa versa.

An Unusual Analogy

Another way to think of multiple threading is to consider the human body. Each one of the body's organs (heart, lungs, liver, brain) are all involved in processes. Each process is running simultaneously. Imagine if each organ ran as a step in a process: first the heart, then the brain, then the liver, then the lungs. We would probably drop dead. So the human body is like one big multithreaded application. All organs are processes running simultaneously, and all of these processes depend upon one another. All of these processes communicate through nerve signals, blood flow and chemical triggers. As with all multithreaded applications, the human body is very complex. If some processes don't get information from other processes, or certain processes slow down or speed up, we end up with a medical problem. That's why (as with all multithreaded applications) these processes need to be synchronized properly in order to function normally.

When to Thread

Multiple threading is most often used in situations where you want programs to run more efficiently. For example, let's say your Window Form program contains a method (call it method_A) inside it that takes more than a second to run and needs to run repetitively. Well, if the entire program ran in a single thread, you would notice times when button presses didn't work correctly, or your typing was a bit sluggish. If method_A was computationally intensive enough, you might even notice certain parts of your Window Form not working at all. This unacceptable program behavior is a sure sign that you need multithreading in your program. Another common scenario where you would need threading is in a messaging system. If you have numerous messages being sent into your application, you need to capture them at the same time your main processing program is running and distribute them appropriately. You can't efficiently capture a series of messages at the same time you are doing any heavy processing, because otherwise you may miss messages. Multiple threading can also be used in an assembly line fashion where several processes run simultaneously. For example once process collects data in a thread, one process filters the data, and one process matches the data against a database. Each of these scenarios are common uses for multithreading and will significantly improve performance of similar applications running in a single thread.

When not to Thread

It is possible that when a beginning programmer first learns threading, they may be fascinated with the possibility of using threading in their program. They may actually become *thread-happy.* Let me elaborate,

Day 1) Programmer learns the that they can spawn a thread and begins creating a single new thread in their program, *Cool*!
Day 2) Programmer says, "I can make this even more efficient by spawning other threads in parts of my program!"
Day 3) P: "Wow, I can even fork threads within threads and REALLY improve efficiency!!"
Day 4) P: "I seem to be getting some odd results, but that's okay, we'll just ignore them for now."
Day 5) "Hmmmm, sometimes my widgetX variable has a value, but other times it never seems to get set at all, must be my computer isn't working, I'll just run the debugger".
Day 9) "This darn (stronger language) program is jumping all over the place!! I can't figure out what is going on!"
Week 2) Sometimes the program just sits there and does absolutely nothing! H-E-L-P!!!!!

Sound familiar? Almost anyone who has attempted to design a multi-threaded program for the first time, even with good design knowledge of threading, has probably experienced at least 1 or 2 of these daily bullet points. I am not insinuating that threading is a bad thing, I'm just pointing out that in the process of creating threading efficiency in your programs, be *very, very careful.* Because unlike a single threaded program, you are handling many processes at the same time, and multiple processes, with multiple dependent variables, can be very tricky to follow. Think of multithreading like you would think of juggling. Juggling a single ball in your hand (although kind of boring) is fairly simple. However, if you are challenged to put two of those balls in the air, the task is a bit more difficult. 3, 4, and 5, balls are progressively more difficult. As the ball count increases, you have a better and better chance of really *dropping the ball.* Juggling a lot of balls at once requires knowledge, skill, and precise timing. So does multiple threading.

Figure 1 - Multithreading is like juggling processes

Problems with Threading

If every process in your program was mutually exclusive - that is, no process depended in any way upon another, then multiple threading would be very easy and very few problems would occur. Each process would run along in its own happy course and not bother the other processes. However, when more than one process needs to read or write the memory used by other processes, problems can occur. For example let's say there are two processes, process #1 and process #2. Both processes share variable X.

If thread process #1 writes variable X with the value 5 first and thread process #2 writes variable X with value -3 next, the final value of X is -3. However if process #2 writes variable X with value -3 first and then process #1 writes variable X with value 5, the final value of X is 5. So you see, if the process that allows you to set X has no knowledge of process #1 or process #2, X can end up with different final values depending upon which thread got to X first. In a single threaded program, there is no way this could happen, because everything follows in sequence. In a single threaded program, since no processes are running in parallel, X is always set by method #1 first, (if it is called first) and then set by method #2. There are no surprises in a single threaded program, it's just step by step. With a mulithreaded program, two threads can enter a piece of code at the same time, and wreak havoc on the results. The problem with threads is that you need some way to control one thread accessing a shared piece of memory while another thread running at the same time is allowed to enter the same code and manipulate the shared data.

Thread Safety

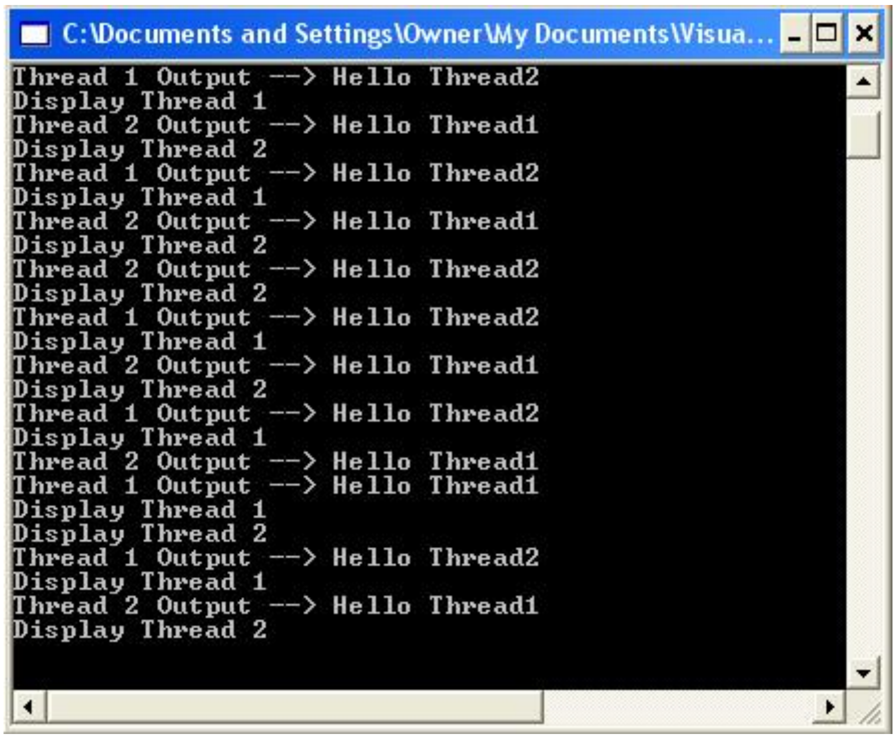
Imagine if every time you juggled three balls, the ball in the air, by some freak of nature, was never allowed to reach your right hand until the ball already sitting in your right hand was thrown. Boy, would juggling be a lot easier! This is what thread safety is all about. In our program we force one thread to wait inside our code block while the other thread is finishing its business. This activity, known as thread blocking or synchronizing threads, allows us to control the timing of simultaneous threads running inside our program. In C# we lock on a particular part of memory (usually an instance of an object) and don't allow any thread to enter code of this object's memory until another thread is done using the object. By now you are probably thirsting for a code example, so here you go.

Let's take a look at a two-threaded scenario. In our example, we will create two threads in C#: Thread 1 and Thread 2, both running in their own while loop. The threads won't do anything useful, they will just print out a message saying which thread they are part of. We will utilize a shared memory class member called _threadOutput. _threadOutput will be assigned a message based upon the thread in which it is running. Listing #1 shows the two threads contained in DisplayThread1 and DisplayThread2 respectively.

Listing 1 - Creating two threads sharing a common variable in memory

```
// shared memory variable between the two threads
// used to indicate which thread we are in
private string _threadOutput = "";

/// <summary>
/// Thread 1: Loop continuously,
/// Thread 1: Displays that we are in thread 1
/// </summary>
void DisplayThread1()
{
    while (_stopThreads == false)
    {
        Console.WriteLine("Display Thread 1");

        // Assign the shared memory to a message about thread #1
        _threadOutput = "Hello Thread1";


        Thread.Sleep(1000);  // simulate a lot of processing

        // tell the user what thread we are in thread #1, and display shared memory
        Console.WriteLine("Thread 1 Output --> {0}", _threadOutput);

    }
}

/// <summary>
/// Thread 2: Loop continuously,
/// Thread 2: Displays that we are in thread 2
/// </summary>
void DisplayThread2()
{
    while (_stopThreads == false)
    {
      Console.WriteLine("Display Thread 2");


     // Assign the shared memory to a message about thread #2
      _threadOutput = "Hello Thread2";


      Thread.Sleep(1000);  // simulate a lot of processing

     // tell the user we are in thread #2
      Console.WriteLine("Thread 2 Output --> {0}", _threadOutput);

    }
}
Class1()
{
    // construct two threads for our demonstration;
    Thread thread1 = new Thread(new ThreadStart(DisplayThread1));
    Thread thread2 = new Thread(new ThreadStart(DisplayThread2));

    // start them
    thread1.Start();
    thread2.Start();
}
```

The results of this code is shown in figure 2. Look carefully at the results. You will notice that the program gives some surprising output(if we were looking at this from a single-threaded mindset). Although we clearly assigned _threadOutput to a string with a number corresponding to the thread to which it belongs, that is not what it looks like in the console:



```
Thread 1 Output --> Hello Thread2
Display Thread 1
Thread 2 Output --> Hello Thread1
Display Thread 2
Thread 1 Output --> Hello Thread2
Display Thread 1
Thread 2 Output --> Hello Thread1
Display Thread 2
Thread 2 Output --> Hello Thread2
Display Thread 2
Thread 1 Output --> Hello Thread2
Display Thread 1
Thread 2 Output --> Hello Thread1
Display Thread 2
Thread 1 Output --> Hello Thread2
Display Thread 1
Thread 2 Output --> Hello Thread1
Thread 1 Output --> Hello Thread1
Display Thread 1
Display Thread 2
Thread 1 Output --> Hello Thread2
Display Thread 1
Thread 2 Output --> Hello Thread1
Display Thread 2
```

Figure 2 - Unusual output from our two thread example.

We would expect to see the following from our code,

Thread 1 Output --> Hello Thread 1 and Thread 2 Output --> Hello Thread 2, but for the most part, the results are completely unpredictable.

Sometimes we see Thread 2 Output --> Hello Thread 1 and Thread 1 Output --> Hello Thread 2. The thread output does not match the code! Even though, we look at the code and follow it with our eyes, _threadOutput = "Hello Thread 2", Sleep, Write "Thread 2 --> Hello Thread 2", this sequence we expect does not necessarily produce the final result.

Explanation

The reason we see the results we do is because in a multithreaded program such as this one, the code theoretically is executing the two methods DisplayThread1 and DisplayThread2, simultaneously. Each method shares the variable, _threadOutput. So it is possible that although _threadOutput is assigned a value "Hello Thread1" in thread #1 and displays _threadOutput two lines later to the console, that somewhere in between the time thread #1 assigns it and displays it, thread #2 assigns _threadOutput the value "Hello Thread2". Not only are these strange results, possible, they are quite frequent as seen in the output shown in figure 2. This painful threading problem is an all too common bug in thread programming known as a *race condition.* This example is a very simple example of the well-known threading problem. It is possible for this problem to be hidden from the programmer much more indirectly such as through referenced variables or collections pointing to thread-unsafe variables. Although in figure 2 the symptoms are blatant, a race condition can appear much more rarely and be intermittent once a minute, once an hour, or appear three days later. The race is probably the programmer's worst nightmare because of its infrequency and because it can be *very very* hard to reproduce.

Winning the Race

The best way to avoid race conditions is to write thread-safe code. If your code is thread-safe, you can prevent some nasty threading issues from cropping up. There are several defenses for writing thread-safe code. One is to share memory as little as possible. If you create an instance of a class, and it runs in one thread, and then you create another instance of the same class, and it runs in another thread, the classes are thread safe as long as they don't contain any static variables. The two classes each create their own memory for their own fields, hence no shared memory. If you do have static variables in your class or the instance of your class is shared by several other threads, then you must find a way to make sure one thread cannot use the memory of that variable until the other class is done using it. The way we prevent one thread from affecting the memory of the other class while one is occupied with that memory is called *locking.*

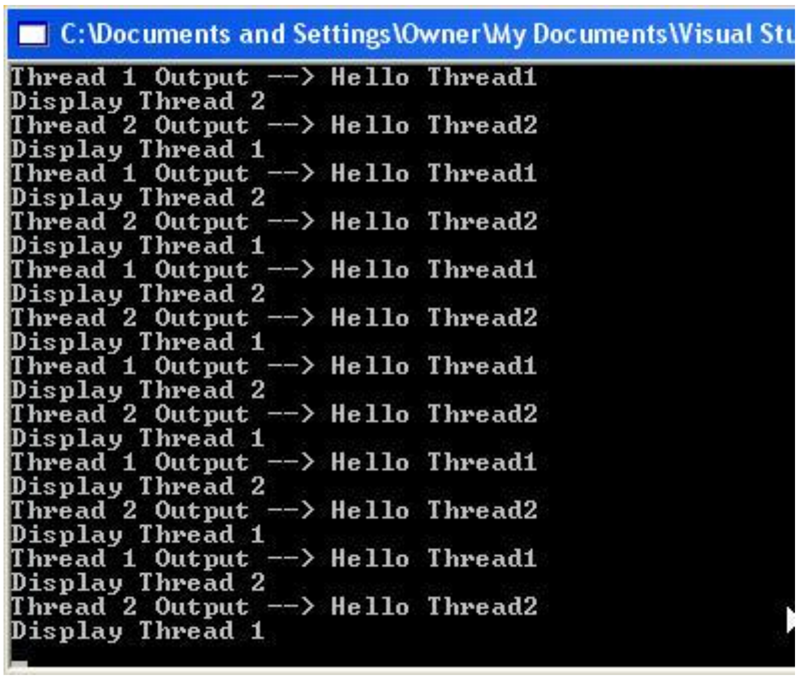
C# allows us to lock our code with either a Monitor class or a lock { } construct. (The lock construct actually internally implements the Monitor class through a try-finally block, but it hides these details from the programmer). In our example in listing 1, we can lock the sections of code from the point in which we populate the shared _threadOutput variable all the way to the actual output to the console. We lock our critical section of code in both threads so we don't have a race in one or the other. The quickest and dirtiest way to lock inside a method is to lock on this pointer. Locking on this pointer will lock on the entire class instance, so any thread trying to modify a field of the class while inside the lock will be *blocked*. Blocking means that the thread trying to change the variable will sit and wait until the lock is released on the locked thread. The thread is released from the lock upon reaching the last bracket in the lock { } construct.

Listing 2 - Synchronizing two Threads by locking them

```
/// <summary>
/// Thread 1, Displays that we are in thread 1 (locked)
 /// </summary>
 void DisplayThread1()
 {
      while (_stopThreads == false)
      {
        // lock on the current instance of the class for thread #1
          lock (this)
          {
              Console.WriteLine("Display Thread 1");
              _threadOutput = "Hello Thread1";
              Thread.Sleep(1000);  // simulate a lot of processing
              // tell the user what thread we are in thread #1
              Console.WriteLine("Thread 1 Output --> {0}", _threadOutput);
          }// lock released  for thread #1 here
      }
 }

/// <summary>
/// Thread 1, Displays that we are in thread 1 (locked)
 /// </summary>
 void DisplayThread2()
 {
      while (_stopThreads == false)
      {

         // lock on the current instance of the class for thread #2
```

```
            lock (this)
            {
                Console.WriteLine("Display Thread 2");
                _threadOutput = "Hello Thread2";
                Thread.Sleep(1000);  // simulate a lot of processing
                // tell the user what thread we are in thread #1
                Console.WriteLine("Thread 2 Output --> {0}", _threadOutput);
            } // lock released  for thread #2 here
        }
 }
```

The results of locking the two threads is shown in figure 3. Note that all thread output is nicely synchronized. You always get a result saying Thread 1 Output --> Hello Thread 1 and Thread 2 Output --> Hello Thread 2. Note, however, that thread locking does come at a price. When you lock a thread, you force the other thread to wait until the lock is released. In essence, you've slowed down the program, because while the other thread is waiting to use the shared memory, the first thread isn't doing anything in the program. Therefore you need to use locks sparingly; don't just go and lock every method you have in your code if they are not involved in shared memory. Also be careful when you use locks, because you don't want to get into the situation where thread #1 is waiting for a lock to be released by thread #2, and thread #2 is waiting for a lock to be released by thread #1. When this situation happens, both threads are blocked and the program appears frozen. This situation is known as *deadlock* and is almost as bad a situation as a race condition because it can also happen at unpredictable, intermittent periods in the program.



```
Thread 1 Output --> Hello Thread1
Display Thread 2
Thread 2 Output --> Hello Thread2
Display Thread 1
Thread 1 Output --> Hello Thread1
Display Thread 2
Thread 2 Output --> Hello Thread2
Display Thread 1
Thread 1 Output --> Hello Thread1
Display Thread 2
Thread 2 Output --> Hello Thread2
Display Thread 1
Thread 1 Output --> Hello Thread1
Display Thread 2
Thread 2 Output --> Hello Thread2
Display Thread 1
Thread 1 Output --> Hello Thread1
Display Thread 2
Thread 2 Output --> Hello Thread2
Display Thread 1
Thread 1 Output --> Hello Thread1
Display Thread 2
Thread 2 Output --> Hello Thread2
Display Thread 1
```

Figure 3 - Synchronizing the dual thread program using locks

Alternative Solution

.NET provides many mechanisms to help you control threads. Another way to keep a thread blocked while another thread is processing a piece of shared memory is to use an AutoResetEvent. The AutoResetEvent class has two methods, Set and WaitOne. These two methods can be used together for controlling the blocking of a thread. When an AutoResetEvent is initialized with false, the program will stop at the line of code that calls WaitOne until the Set method is called on the AutoResetEvent. After the Set method is executed on the AutoResetEvent, the thread becomes unblocked and is allowed to proceed past WaitOne. The next time WaitOne is called, it has automatically been reset, so the program will again wait (block) at the line of code in which the WaitOne method is executing. You can use this "stop and trigger" mechanism to block on one thread until another thread is ready to free the blocked thread by calling Set. Listing 3 shows our same two threads using the AutoResetEvent to block each other while the blocked thread waits and the unblocked thread executes to display _threadOutput to the Console. Initially, _blockThread1 is initialized to signal false, while _blockThread2 is initialized to signal true. This means that _blockThread2 will be allowed to proceed through the WaitOne call the first time through the loop in DisplayThread_2, while _blockThread1 will block on its WaitOne call in DisplayThread_1. When the _blockThread2 reaches the end of the loop in thread 2, it signals _blockThread1 by calling Set in order to release thread 1 from its block. Thread 2 then waits in its WaitOne call until Thread 1 reaches the end of its loop and calls Set on _blockThread2. The Set called in Thread 1 releases the block on thread 2 and the process starts again. Note that if we had set both AutoResetEvents (_blockThread1 and _blockThread2) initially to signal false, then both threads would be waiting to proceed through the loop without any chance to trigger each other, and we would experience a *deadlock.*

Listing 3 - Alternatively Blocking threads with the AutoResetEvent

```
AutoResetEvent _blockThread1 = new AutoResetEvent(false);
AutoResetEvent _blockThread2 = new AutoResetEvent(true);

/// <summary>
/// Thread 1, Displays that we are in thread 1
/// </summary>
void DisplayThread_1()
{
    while (_stopThreads == false)
    {
          // block thread 1  while the thread 2 is executing
           _blockThread1.WaitOne();

           // Set was called to free the block on thread 1, continue executing the code
             Console.WriteLine("Display Thread 1");

             _threadOutput = "Hello Thread 1";
             Thread.Sleep(1000);  // simulate a lot of processing

              // tell the user what thread we are in thread #1
             Console.WriteLine("Thread 1 Output --> {0}", _threadOutput);

           // finished executing the code in thread 1, so unblock thread 2
             _blockThread2.Set();
    }
}

/// <summary>
/// Thread 2, Displays that we are in thread 2
/// </summary>
void DisplayThread_2()
{
    while (_stopThreads == false)
    {
        // block thread 2  while thread 1 is executing
            _blockThread2.WaitOne();

        // Set was called to free the block on thread 2, continue executing the code
            Console.WriteLine("Display Thread 2");

            _threadOutput = "Hello Thread 2";
            Thread.Sleep(1000);  // simulate a lot of processing

            // tell the user we are in thread #2
            Console.WriteLine("Thread 2 Output --> {0}", _threadOutput);

        // finished executing the code in thread 2, so unblock thread 1
          _blockThread1.Set();
    }
}
```

The output produced by listing 3 is the same output as our locking code, shown in figure 3, but the AutoResetEvent gives us some more dynamic control over how one thread can notify another thread when the current thread is done the processing.

Conclusion

As we are pushing the theoretical limits of microprocessor speed, technology needs to find new ways to be able to optimize speed and performance in computer technology. With the invention of multiple processor chips and the inroads into parallel programming, understanding multithreading can prepare you for the paradigm needed to handle these more recent technologies that will bring us the advantage we need to continue to challenge Moore's Law. C# and .NET give us the ability to support multithreading and parallel processing. If we understand how to use to utilize these tools skillfully, we can prepare for these hardware promises of the future in our own day-to-day programming activities. In the meantime, *sharp*en your knowledge of threading, so you can *.net* the possibilities.

How to create Threads in C#

In C#, a multi-threading system is built upon the Thread class, which encapsulates the execution of threads. This class contains several methods and properties which helps in managing and creating threads and this class is defined under `System.Threading` namespace. The `System.Threading` namespace provides classes and interfaces that are used in multi-thread programming.

Some commonly used classes in this namespace are :

Class Name	Description
Mutex	It is a synchronization primitive that can also be used for IPS (interprocess synchronization).
Monitor	This class provides a mechanism that access objects in synchronize manner.
Semaphore	This class is used to limit the number of threads that can access a resource or pool of resources concurrently.
Thread	This class is used to creates and controls a thread, sets its priority, and gets its status.
ThreadPool	This class provides a pool of threads that can be used to execute tasks, post work items, process asynchronous I/O, wait on behalf of other threads, and process timers.
ThreadLocal	This class provides thread-local storage of data.
Timer	This class provides a mechanism for executing a method on a thread pool thread at specified intervals. You are not allowed to inherit this class.
Volatile	This class contains methods for performing volatile memory operations.

Steps to create a thread in a C# Program:

1. First of all import `System.Threading` namespace, it plays an important role in creating a thread in your program as you have no need to write the fully qualified name of class everytime.
2. `Using System;`
3. `Using System.Threading`
4. Now, create and initialize the thread object in your main method.
5. `public static void main()`
6. `{`
7. `    Thread thr = new Thread(job1);`
8. `}`

Or

You can also use ThreadStart constructor for initializing a new instance.

```
public static void main()
{
     Thread thr = new Thread(new ThreadStart(job1));
}
```

9. Now you can call your thread object.

```
public static void main()
{
     Thread thr = new Thread(job1);
     thr.Start();
}
```

Below programs illustrate the practical implementations of above steps:

Example 1:

filter_none

edit

play_arrow

brightness_4

```
// C# program to illustrate the
// creation of thread using
// non-static method
using System;
using System.Threading;

public class ExThread {

    // Non-static method
    public void mythread1()
    {
        for (int z = 0; z < 3; z++) {
            Console.WriteLine("First Thread");
        }
    }
}

// Driver Class
public class GFG {

    // Main Method
    public static void Main()
    {
        // Creating object of ExThread class
        ExThread obj = new ExThread();

        // Creating thread
        // Using thread class
        Thread thr = new Thread(new ThreadStart(obj.mythread1));
        thr.Start();
    }
}
```

Output:

```
First Thread
First Thread
First Thread
```

Explanation: In the above example, we have a class named as *ExThread* that contain a non-static method named as *mythread1()*. So we create an instance, i.e. *obj* of ExThread class and refer it in the constructor of *ThreadStart* class as given in this statement `Thread a = new Thread(new ThreadStart(obj.mythread1));`. Using `Thread a = new Thread(new ThreadStart(obj.mythread1));` statement we will create a thread named as *thr* and initialize the work of this thread. By using `thr.Start();` statement.

Example 2:

filter_none
edit
play_arrow
brightness_4

```
// C# program to illustrate the creation
// of thread using static method
using System;
using System.Threading;

public class ExThread {

    // Static method for thread a
    public static void thread1()
    {
        for (int z = 0; z < 5; z++) {
            Console.WriteLine(z);
        }
    }

    // static method for thread b
    public static void thread2()
    {
        for (int z = 0; z < 5; z++) {
            Console.WriteLine(z);
        }
    }
}

// Driver Class
public class GFG {

    // Main method
    public static void Main()
    {
        // Creating and initializing threads
        Thread a = new Thread(ExThread.thread1);
        Thread b = new Thread(ExThread.thread2);
        a.Start();
        b.Start();
    }
}
```

Output :

```
0
1
2
3
4
0
1
2
3
4
```

Explanation: In the above example, we have a class named as *ExThread* and contain two static methods named as *thread1()* and *thread2()*. So we do not need to create an instance of *ExThread* class. Here we call these methods using a class name, like `ExThread.thread1`, `ExThread.thread2`. By using `Thread a = new Thread(ExThread.thread1);` statement we create and initialize the work of thread *a*, similarly for thread *b*. By using `a.Start();` and `b.Start();` statements, *a* and *b* threads scheduled for execution.

Note: The output of these programs may vary due to context switching.

How to Write Your First Multi-threaded Application with C#

Languages Frameworks and Tools
C#

Introduction

CPU with multiple cores have become more and more common these days. To use full potential of the machine, writing multi-threaded applications is now really important. Multi-threaded applications are difficult to code and test, expensive to maintain as they are prone to deadlock, race conditions, and so many other multi-threaded hazards.
Writing threaded application is not that tough these days as it used to be in the past. Dot net has already done much of the heavy lifting for us so that we can focus more on application specifics.
There are various ways to write multi-threaded application in C#.

Starting Thread

```
public class SimpleThreadExample
{
    public void StartMultipleThread()
    {
        DateTime startTime = DateTime.Now;

        Thread t1 = new Thread(() =>
        {
            int numberOfSeconds = 0;
            while (numberOfSeconds < 5)
            {
                Thread.Sleep(1000);

                numberOfSeconds++;
            }

            Console.WriteLine("I ran for 5 seconds");
        });

        Thread t2 = new Thread(() =>
        {
            int numberOfSeconds = 0;
            while (numberOfSeconds < 8)
            {
                Thread.Sleep(1000);

                numberOfSeconds++;
            }

            Console.WriteLine("I ran for 8 seconds");
        });


        //parameterized thread
        Thread t3 = new Thread(p =>
        {
            int numberOfSeconds = 0;
            while (numberOfSeconds < Convert.ToInt32(p))
            {
                Thread.Sleep(1000);

                numberOfSeconds++;
            }

            Console.WriteLine("I ran for {0} seconds", numberOfSeconds);
        });

        t1.Start();
        t2.Start();
        //passing parameter to parameterized thread
        t3.Start(20);

        //wait for t1 to fimish
        t1.Join();

        //wait for t2 to finish
        t2.Join();

        //wait for t3 to finish
        t3.Join();


        Console.WriteLine("All Threads Exited in {0} secods", (DateTime.Now -
startTime).TotalSeconds);
    }
}
```

Terminating Thread

`Thread.Abort()` method can be used to destroy the running thread. But in practice, it is better to let the CLR do the dirty work for you. You can have a flag which would break your long running process.

```
public class DestroyThreadExample
{
    public bool IsCancelled { get; set; }

    public Thread MyThread { get; set; }

    public void StartThread()
    {
        MyThread = new Thread(() =>
        {
            int numberOfSeconds = 0;
            while (numberOfSeconds < 8)
            {
                if (IsCancelled == false)
                {
                    break;
                }

                Thread.Sleep(1000);

                numberOfSeconds++;
            }

            Console.WriteLine("I ran for {0} seconds", numberOfSeconds);
        });
    }

    //Destroy thread
    public void Abort()
    {
        MyThread.Abort();
    }

    //Graceful exit
    public void GracefulAbort()
    {
        IsCancelled = true;
    }
}
```

Responsive UI

One of the most popular use cases of threading is to make the UI responsive. Lets see how we can achieve that with the simple thread and windows form.

```
public partial class Form1 : Form
{
    public delegate void UpdateLabel(string label);

    public bool IsCancelled { get; set; }

    public Form1()
    {
        InitializeComponent();
    }

    private void UpdateUI(string labelText)
    {
        lblStopWatch.Text = labelText;
    }

    private void btnStart_Click(object sender, EventArgs e)
    {
        DateTime startTime = DateTime.Now;

        IsCancelled = false;

        Thread t = new Thread(() =>
        {
            while (IsCancelled==false)
            {
                Thread.Sleep(1000);

                string      timeElapsedInstring      =      (DateTime.Now      -
startTime).ToString(@"hh\:mm\:ss");

                lblStopWatch.Invoke(new                  UpdateLabel(UpdateUI),
timeElapsedInstring);

            }
        });


        t.Start();
    }

    private void btnStop_Click(object sender, EventArgs e)
    {
        IsCancelled = true;
    }
}
```

Noticed the delegate being used to update the UI?
`lblStopWatch.Invoke(new UpdateLabel(UpdateUI), timeElapsedInstring);`
This is because you can't update UI from any other thread other than the UI thread. I think this is very common scenario every one would have dealt with while working on windows forms.
There is another easiest way to update the UI.

BackgroundWorker

There is another easy way to create a thread specifically to update the UI: BackgroundWorker

```
public partial class Form2 : Form
{
    BackgroundWorker workerThread = null;

    bool _keepRunning = false;

    public Form2()
    {
        InitializeComponent();

        InstantiateWorkerThread();
    }

    private void InstantiateWorkerThread()
    {
        workerThread = new BackgroundWorker();
        workerThread.ProgressChanged += WorkerThread_ProgressChanged;
        workerThread.DoWork += WorkerThread_DoWork;
        workerThread.RunWorkerCompleted += WorkerThread_RunWorkerCompleted;
        workerThread.WorkerReportsProgress = true;
        workerThread.WorkerSupportsCancellation = true;
    }

    private void WorkerThread_ProgressChanged(object sender,
ProgressChangedEventArgs e)
    {
        lblStopWatch.Text = e.UserState.ToString();
    }

    private void WorkerThread_RunWorkerCompleted(object sender,
RunWorkerCompletedEventArgs e)
    {
        if(e.Cancelled)
        {
            lblStopWatch.Text = "Cancelled";
        }
        else
        {
            lblStopWatch.Text = "Stopped";
        }
    }

    private void WorkerThread_DoWork(object sender, DoWorkEventArgs e)
    {
        DateTime startTime = DateTime.Now;

        _keepRunning = true;

        while (_keepRunning)
        {
            Thread.Sleep(1000);
            string       timeElapsedInstring       =       (DateTime.Now       -
startTime).ToString(@"hh\:mm\:ss");

            workerThread.ReportProgress(0, timeElapsedInstring);

            if(workerThread.CancellationPending)
            {
                // this is important as it set the cancelled property of
RunWorkerCompletedEventArgs to true
                e.Cancel = true;
                break;
            }
        }
    }

    private void btnStart_Click(object sender, EventArgs e)
    {
        workerThread.RunWorkerAsync();
    }

    private void btnStop_Click(object sender, EventArgs e)
    {
        _keepRunning = false;
    }

    private void btnCancel_Click(object sender, EventArgs e)
    {
        workerThread.CancelAsync();
    }
}
```

In this approach, instead of creating the plain old thread and using delegate to update the UI, we have BackgroundWorker component which does the work for us. It supports multiple events to run long running process (DoWork), update the UI (ProgressChanged) and you will know when the background thread has actually ended (RunWorkerCompleted). In the plain old thread, knowing the end of the thread is tricky and you have to rely either of Thread. Join or use some other wait handles.

Problems Introduced by Threading

While writing the multi-threaded application, there are a bunch of known issues that we should be able to handle. Deadlocks and race conditions are few to name. It is necessary to maintain synchronized access to different resources to make sure we are not corrupting the output. For example, if a file in the filesystem is being modified by multiple threads, the application must allow only one thread to modify the file at a time, otherwise the file might get corrupted. One way of doing it is using Lock keyword. If we are accessing the shared resource around the Lock statement, it will allow only one thread to execute the code within the lock block.

```
public class LockExample
{

    public int SharedResource { get; set; }

    public object _locker = 0;

    public void StartThreadAccessingSharedResource()
    {
        Thread t1 = new Thread(() =>
        {
            lock (_locker)
            {
                SharedResource++;
            }

        });

        Thread t2 = new Thread(() =>
        {
            lock (_locker)
            {
                SharedResource--;
            }

        });

        t1.Start();
        t2.Start();
    }
}
```
csharp

In this example we have two threads accessing the same resource. Using the lock keyword will guarantee that the shared variable will be accessed by one one thread at a time. While thread t1 is executing the code within the lock block, thread t2 will be waiting.

Conclusion

These basics of C# multi-threading programming will prepare you for advance topics like Concurrent data structure, Wait handles, Tasks, and asynchronous programming with C#.
Associated code can be accessed on GitHub.

A beginner's guide to threading in C#

by Pete Bosch in Developer on May 31, 2002, 12:00 AM PST

Threading can increase your application's performance and scalability if it is implemented correctly. Find out the basic concepts and see how you can use them in your applications.

Using multiple threads can help you achieve greater performance, scalability, and responsiveness in your applications—but you need to be careful. This article begins a series on the tools and techniques involved in threading. I'll start with an introduction to the concept and a survey of some of the more common constructs and how to use them.

The yin and yang of threading

It's been said that one of the best things about Java is that it makes threading easy, but—at the same time—that one of the worst things about Java is that it makes threading easy. When Microsoft developed C#, it brought this ease-of-use dilemma to a whole new platform. There are more primitives to play with in C# than in Java, but the basic Java primitives of the *Thread* object and synchronization monitors are there in similar form and function and provide more than enough steel to hoist yourself on your own petard. So be very, very careful in making the decision to utilize explicit multithreading in your application.

Why not to multithread

The first point to remember when deciding whether to avoid multithreading is that, unless you are doing weekend play-coding, do not use threading simply because it is cool. It gets hot soon enough, and if you're not careful, your boss will, too. Second, you should not use multithreading to make things faster until you have proven to yourself (and hopefully a few others around you) that a single threaded implementation is unacceptably slow. And finally, before venturing into an explicitly multithreaded mechanism, remember that Microsoft provides an apartment model that allows an object, written as a single-threaded construct, to run in a multithreaded environment. So you may not need to explicitly code for multithreading. The apartment model is a subject for another article.

If not done right, multithreading opens a Pandora's box of ill effects. With no apparent repeatability, values can turn to utter garbage. Counters can fail to increment. Your application can suddenly freeze. Resources such as database connections can unexpectedly close or become exhausted. Some of the most challenging puzzles in a senior developer's career arise from sleuthing a threading issue. The big problem is that these puzzles usually take time to solve, which can have a serious effect on product delivery dates, or worse, on product reliability.

Why to multithread

You may have a good candidate for multithreading if your application has operations like any of the following:

- · In series, can take an unacceptably long time to complete
- · Can be made parallel
- · May spend an appreciable amount of their execution time waiting for network, file system, or user or other I/O response

But before crossing the Rubicon, make sure that each of the above three circumstances holds.

If your code is fast enough, but you think you could make it *really* fast (you *do* have performance specifications, don't you?), resist the urge. If you aren't sure that you can make your operations parallel (such as performing simultaneous database updates into the same table, when your database does table-level locking), fight the temptation. If you don't know whether your application is spending a lot of time waiting for input or output to complete, determine that first. Three threads, each performing a computation of pi to the millionth place, will actually take longer to complete than repeating the computation three times in the same thread. This scenario fails the third criterion above—there are no idle cycles during one computation that a second parallel computation might be able to use.

The one exception to this rule is that if you are writing for a multiprocessor machine, you might stand to gain by making suitable operations parallel, even if each of the operations is a CPU hog.

Basic thread management tools

Having provided ample warning and set the stage for when and when not to use multiple threads, I will now describe some of the tools you have at your disposal if you choose to do so.

Thread

The .NET libraries provide an object called *System.Threading.Thread*, which represents a single thread of execution. You can start a thread, seeking to accomplish a task in that thread while the current thread continues. This would be useful for an application that needed to print a document or save a large file but wanted to acknowledge the user's request and return control to the user. We demonstrate this mechanism in **Listing A**.

We first create a method, *SayHello*, that does what we want to accomplish. Its signature must match that of the *System.Threading.ThreadStart* delegate. Note the *Thread.Sleep(int numMillisecs)* call in the *SayHello* method. This is a useful construct and will appear often in these samples.

In the Main routine, we create a new thread with a *ThreadStart* delegate made from the *SayHello* method, and call *Start* on that thread. The thread we created is started, and our main thread continues on to completion in this example.

Many times, you will have a slightly different task to perform in each thread and will want to pass each thread a parameter of some sort to differentiate its task from that of the others. While there are several reasonable ways of doing this, the most straightforward is to create a *Task* object that holds the thread, the unique parameter, and the work method that provides the *ThreadStart* delegate. From the *work* method, you can read the supplied parameter, as it is a member of the *Task* object and is therefore unique to that thread. By making the thread a public field, you have full access to all the thread's members without having to write additional wrapper code. See **Listing B** for an example of this technique.

You can even provide a return value of sorts from the *Task* object by defining a field in the task to hold it, setting the value before the thread completes, and reading it from the thread that started the task after the task completes.

You can pause one thread, waiting for other threads to complete what they are doing. You might do this when you want to collect return values as described above or when you spread a database update across three separate threads but don't want to proceed until all threads are done. This technique is shown in **Listing C**.

Here, we build on the code from Listing A. This time, we launch two threads, each with the same task as before. Following the calling of both threads' *Start()* methods, though, we call their *Join()* methods. Calling *Join()* on a thread causes the calling thread to pause execution until the called thread has completed. So the *thread1.Start()* method causes the main thread to pause until *thread1* has completed. We then do the same thing to *thread2*. As a result, the main thread does not complete until both *thread1* and *thread2* have completed.

Here are two parting thoughts on this example. First, a thread may not call *Join* on another thread until that thread has been started. Second, there are two more forms of *Join* that allow specification of a timeout after which the calling thread will continue even if the called thread is still running.

Computer science frequently employs the concept of a watchdog—an entity whose responsibility is to ensure the correct function, or handling of incorrect functions, in another entity. A common pattern is the watchdog timer, usually responsible for making sure that another task completes in a reasonable time. **Listing D** shows a simple mechanism for implementing a watchdog timer.

After *thread1* is started, we join with it, but provide a 10-second timeout. Since *thread1* has a 15-second pause built in, it will still be alive when the join expires. The main thread tests *thread1.IsAlive* and, if it's still alive, terminates the thread.

Synchronization and monitors

Synchronization refers to the practice of ensuring that only one thread executes in a section of code at a time. While discussion of all of them is beyond the scope of this article, a surprising number of constructs must take place inside a single threaded block to be reliably safe. Unfortunately, most of them work fine almost all of the time if outside of such a block, so the old "If it compiles and I get the answer I expect, it's right" mantra doesn't hold here. This is part of why multithreading is so dangerous.

A monitor is the most basic synchronization construct. Any object can have a monitor associated with it, and no monitor can be associated with more than one object. Monitors have a "lock," which may be acquired by only one thread at a time. It must be released by that thread before another thread can acquire it. You can guard a section of code by declaring an object that is visible to all threads, such as a class field, and having a section of code acquire the lock from that monitor before performing some operation and then release the lock when it completes. This construct is demonstrated in **Listing E**.

We declare an object, *myLockObject*, whose sole purpose is to provide a monitor for synchronization. In the *SayHello* method, we allow both threads to print "Hello" whenever they want. However, we control the printing of "Wonderful" and "World" with a monitor associated with *myMonitorObject* so that one thread must complete both prints before another is allowed to begin.

Two other techniques are available for accomplishing this mechanism—the *lock()* keyword and the *MethodImplAttribute* attribute. See **Listing F** for an example.

We replace the *Monitor.Enter(…)* and *Monitor.Exit(…)* constructs with a *lock(…){ … }* construct. These constructs are identical in effect—the latter is simply shorthand for the former. We also add a method, *SayHello2(),* which has an attribute attached to it, *MethodImpl*. This attribute specifies that the entire method is to be synchronized. This is equivalent to forcing the calling code to acquire a lock on the monitor associated with the type object that contains the method before it is allowed to make the call. This is cleaner than enclosing the method body in a *lock(){…}* statement. Note that the documentation defines the attribute as being called *MethodImplAttribute*, but its implementation has it called *MethodImpl* instead. According to the stated convention for declaring attributes, it appears that a developer at Microsoft may have goofed.

Summary
This article has covered a lot of ground. I have discussed the reasons for and against explicitly using multiple threads, as well as shown some of the primitive constructs you will need if you choose to do threading. I introduced the *Thread* object and explained how to run several threads to accomplish the task of your choice. I described the monitor concept and showed how to use it to achieve synchronization around a block of code. I also described two shorthand means of accomplishing the same thing in specific cases, the *lock* keyword and the *MethodImpl* attribute.

In future articles, I will describe several other basic constructs, implement a thread pool, and explore more advanced constructs, such as thread-local storage and overlapped I/O.

C# Multithreading

Multitasking is the simultaneous execution of multiple tasks or processes over a certain time interval. Windows operating system is an example of multitasking because it is capable of running more than one process at a time like running Google Chrome, Notepad, VLC player etc. at the same time. The operating system uses a term known as an *process* to execute all these applications at the same time. A process is a part of an operating system which is responsible for executing an application. Every program that executes on your system is a process and to run the code inside the application a process uses a term known as an *thread*.

A thread is a lightweight process, or in other words, a thread is a unit which executes the code under the program. So every program has logic and a thread is responsible for executing this logic. Every program by default carries one thread to executes the logic of the program and the thread is known as the *Main Thread*, so every program or application is by default single threaded model. This single-threaded model has a drawback. The single thread runs all the process present in the program in synchronizing manner, means one after another. So, the second process waits until the first process completes its execution, it consumes more time in processing.
For example, we have a class named as *Geek* and this class contains two different methods, i.e *method1*, *method2*. Now the main thread is responsible for executing all these methods, so the main thread executes all these methods one by one.

Example:
filter_none
edit
play_arrow
brightness_4

```
// C# program to illustrate the
// concept of single threaded model
using System;
using System.Threading;
```

```
public class Geek {

    // static method one
    public static void method1()
    {

        // It prints numbers from 0 to 10
        for (int I = 0; I <= 10; I++) {

            Console.WriteLine("Method1 is : {0}", I);

            // When the value of I is equal to
            // 5 then this method sleeps for
            // 6 seconds and after 6 seconds
            // it resumes its working
            if (I == 5) {
                Thread.Sleep(6000);
            }
        }
    }

    // static method two
    public static void method2()
    {

        // It prints numbers from 0 to 10
        for (int J = 0; J <= 10; J++) {

            Console.WriteLine("Method2 is : {0}", J);
        }
    }
}

// Driver Class
public class GFG {

    // Main Method
    static public void Main()
    {

        // Calling static methods
        Geek.method1();
        Geek.method2();
    }
}
```
Output:
```
Method1 is : 0
Method1 is : 1
Method1 is : 2
Method1 is : 3
Method1 is : 4
Method1 is : 5
Method1 is : 6
Method1 is : 7
Method1 is : 8
Method1 is : 9
Method1 is : 10
Method2 is : 0
Method2 is : 1
Method2 is : 2
Method2 is : 3
Method2 is : 4
```

```
Method2 is : 5
Method2 is : 6
Method2 is : 7
Method2 is : 8
Method2 is : 9
Method2 is : 10
```

Explanation: Here, first of all, *method1* executes. In *method1* , *for* loop starts from 0 when the value of i is equal to 5 then the method goes into sleep for 6 seconds and after 6 seconds it resumes its process and print remaining value. Until *method2* is in the waiting state. *method2* start its working when *method1* complete its assigned task. So to overcome the drawback of single threaded model `multithreading` is introduced.

Multi-threading is a process which contains multiple threads within a single process. Here each thread performs different activities. For example, we have a class and this call contains two different methods, now using multithreading each method is executed by a separate thread. So the major advantage of multithreading is it works simultaneously, means multiple tasks executes at the same time. And also maximizing the utilization of the CPU because multithreading works on time-sharing concept mean each thread takes its own time for execution and does not affect the execution of another thread, this time interval is given by the operating system.

Example:
filter_none
edit
play_arrow
brightness_4

```
// C# program to illustrate the
// concept of multithreading
using System;
using System.Threading;

public class GFG {

    // static method one
    public static void method1()
    {

        // It prints numbers from 0 to 10
        for (int I = 0; I <= 10; I++) {
            Console.WriteLine("Method1 is : {0}", I);

            // When the value of I is equal to 5 then
            // this method sleeps for 6 seconds
            if (I == 5) {
                Thread.Sleep(6000);
            }
        }
    }

    // static method two
    public static void method2()
    {
        // It prints numbers from 0 to 10
        for (int J = 0; J <= 10; J++) {
            Console.WriteLine("Method2 is : {0}", J);
        }
    }

    // Main Method
    static public void Main()
    {

        // Creating and initializing threads
        Thread thr1 = new Thread(method1);
        Thread thr2 = new Thread(method2);
        thr1.Start();
        thr2.Start();
    }
}
```

Output :

```
Method1 is : 0
Method1 is : 1
Method1 is : 2
Method1 is : 3
Method2 is : 0
Method2 is : 1
Method2 is : 2
Method2 is : 3
Method2 is : 4
Method2 is : 5
Method2 is : 6
Method2 is : 7
Method2 is : 8
Method2 is : 9
Method2 is : 10
Method1 is : 4
Method1 is : 5
Method1 is : 6
Method1 is : 7
Method1 is : 8
Method1 is : 9
Method1 is : 10
```

Explanation: Here, we create and initialize two threads, i.e *thr1* and *thr2* using Thread class. Now using `thr1.Start();` and `thr2.Start();` we start the execution of both the threads. Now both thread runs simultaneously and the processing of *thr2* does not depend upon the processing of *thr1* like in the single threaded model.

Note: Output may vary due to context switching.

Advantages of Multithreading:

- It executes multiple process simultaneously.
- Maximize the utilization of CPU resources.
- Time sharing between multiple process.

How to create Threads in C#

In C#, a multi-threading system is built upon the Thread class, which encapsulates the execution of threads. This class contains several methods and properties which helps in managing and creating threads and this class is defined under `System.Threading` namespace. The `System.Threading` namespace provides classes and interfaces that are used in multi-thread programming.

Some commonly used classes in this namespace are :

Class Name	Description
Mutex	It is a synchronization primitive that can also be used for IPS (interprocess synchronization).
Monitor	This class provides a mechanism that access objects in synchronize manner.

Semaphore	This class is used to limit the number of threads that can access a resource or pool of resources concurrently.
Thread	This class is used to creates and controls a thread, sets its priority, and gets its status.
ThreadPool	This class provides a pool of threads that can be used to execute tasks, post work items, process asynchronous I/O, wait on behalf of other threads, and process timers.
ThreadLocal	This class provides thread-local storage of data.
Timer	This class provides a mechanism for executing a method on a thread pool thread at specified intervals. You are not allowed to inherit this class.
Volatile	This class contains methods for performing volatile memory operations.

Steps to create a thread in a C# Program:

1. First of all import `System.Threading` namespace, it plays an important role in creating a thread in your program as you have no need to write the fully qualified name of class everytime.
2. `Using System;`
3. `Using System.Threading`
4. Now, create and initialize the thread object in your main method.
5. `public static void main()`
6. `{`
7. `    Thread thr = new Thread(job1);`
8. `}`

   Or

   You can also use ThreadStart constructor for initializing a new instance.
   ```
   public static void main()
   {
        Thread thr = new Thread(new ThreadStart(job1));
   }
   ```
9. Now you can call your thread object.
10. `public static void main()`
11. `{`
12. `    Thread thr = new Thread(job1);`
13. `    thr.Start();`
14. `}`

Below programs illustrate the practical implementations of above steps:

Example 1:

filter_none
edit
play_arrow
brightness_4

```
// C# program to illustrate the
// creation of thread using
// non-static method
using System;
using System.Threading;

public class ExThread {

    // Non-static method
    public void mythread1()
    {
        for (int z = 0; z < 3; z++) {
            Console.WriteLine("First Thread");
        }
    }
}

// Driver Class
public class GFG {
```

```
    // Main Method
    public static void Main()
    {
        // Creating object of ExThread class
        ExThread obj = new ExThread();

        // Creating thread
        // Using thread class
        Thread thr = new Thread(new ThreadStart(obj.mythread1));
        thr.Start();
    }
}
```

Output:

```
First Thread
First Thread
First Thread
```

Explanation: In the above example, we have a class named as *ExThread* that contain a non-static method named as *mythread1()*. So we create an instance, i.e. *obj* of ExThread class and refer it in the constructor of *ThreadStart* class as given in this statement `Thread a = new Thread(new ThreadStart(obj.mythread1));`. Using T`hread a = new Thread(new ThreadStart(obj.mythread1));` statement we will create a thread named as *thr* and initialize the work of this thread. By using `thr.Start();` statement.

Example 2:

filter_none
edit
play_arrow
brightness_4

```
// C# program to illustrate the creation
// of thread using static method
using System;
using System.Threading;

public class ExThread {

    // Static method for thread a
    public static void thread1()
    {
        for (int z = 0; z < 5; z++) {
            Console.WriteLine(z);
        }
    }

    // static method for thread b
    public static void thread2()
    {
        for (int z = 0; z < 5; z++) {
            Console.WriteLine(z);
        }
    }
}

// Driver Class
public class GFG {

    // Main method
    public static void Main()
    {
        // Creating and initializing threads
        Thread a = new Thread(ExThread.thread1);
        Thread b = new Thread(ExThread.thread2);
        a.Start();
        b.Start();
    }
}
```

Output :

```
0
1
2
3
4
0
1
2
3
4
```

Explanation: In the above example, we have a class named as *ExThread* and contain two static methods named as *thread1()* and *thread2()*. So we do not need to create an instance of *ExThread* class. Here we call these methods using a class name, like `ExThread.thread1`, `ExThread.thread2`. By using `Thread a = new Thread(ExThread.thread1);` statement we create and initialize the work of thread *a*, similarly for thread *b*. By using `a.Start();` and `b.Start();` statements, *a* and *b* threads scheduled for execution.

Note: The output of these programs may vary due to context switching.

Joseph Albahari has a great article called Threading in C#. This is really cool blog post about start to learn for threading in C#. Joseph clearly explained:

- Introduction and Concepts
- Join and Sleep
- How Threading Works
- Creating and Starting Threads
- Thread Pooling

And check this out article from Codeproject.

- Getting Started

You can create and start a new thread by instantiating a `Thread` object and calling its Start method. The simplest constructor for Thread takes a `ThreadStart` delegate: a parameterless method indicating where execution should begin.

```
using System;
using System.Threading;

class ThreadTest
{
   static void Main()
   {
      Thread t = new Thread (WriteY); // Kick off a new thread
      t.Start(); // running WriteY()
      // Simultaneously, do something on the main thread.
      for (int i = 0; i < 1000; i++) Console.Write ("x");
   }
   static void WriteY()
   {
      for (int i = 0; i < 1000; i++) Console.Write ("y");
      }
}

// Output:
xxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxx
```

```
xxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
...
```