Module "C#"

Submodule "C# Multithreading"

Concurrency and Asynchronous programming

UA Resource Development Unit

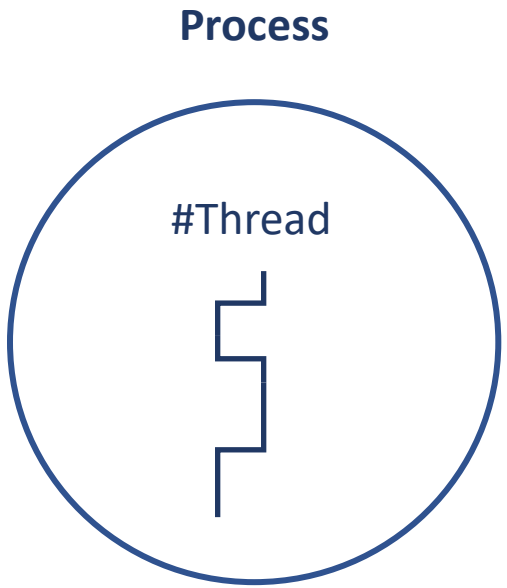
2020

Introduction

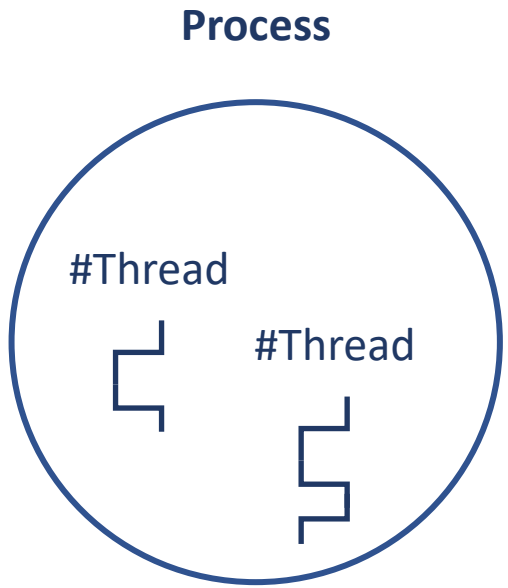
The most common concurrency scenarios:

- *Writing a responsive user interface*
- *Allowing requests to process simultaneously*
- *Parallel programming*
- *Speculative execution*

Threading



Single thread process



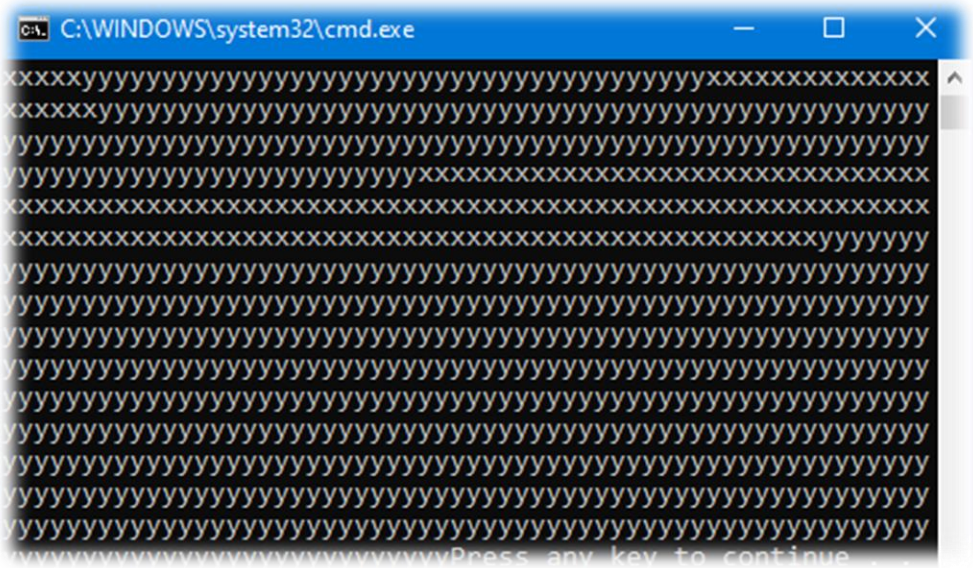
Multi thread process

Creating a Thread

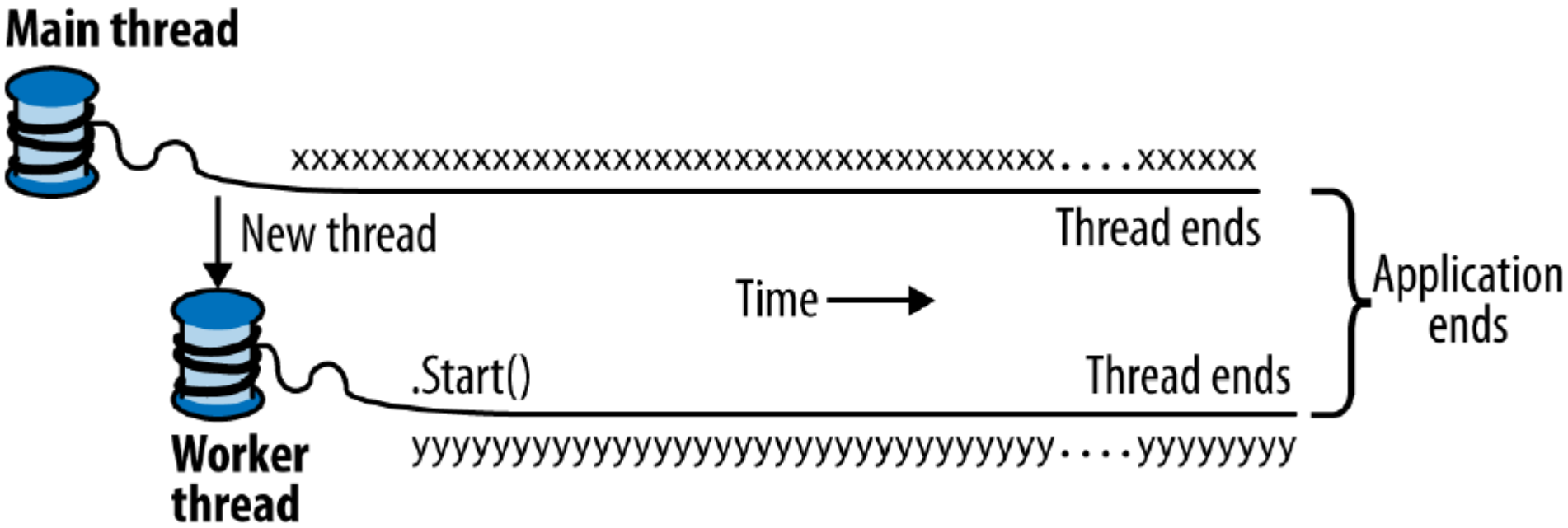
```
using System;
using System.Threading;

namespace A_SimpleThread
{
    class Program
    {
        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");
        }
    }
}
```



Creating a Thread



Thread state

```
static void ThreadLifeCycle()
{
    Thread t = new Thread(() =>
    {
        int x = 0;
        while (x < 100)
        {
            x++;
        }
    });
    t.Start();
    while (t.IsAlive)
    {
        Console.WriteLine("Thread is Alive");
    }
}
```

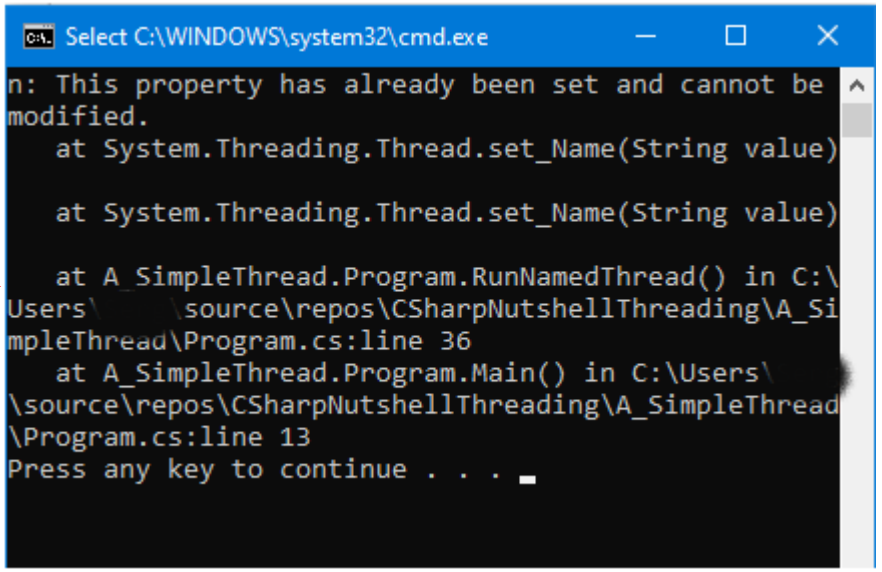
Thread's value **IsAlive** property will be **true** as long as delegate didn't finish its job

Thread properties

```
static void RunNamedThread()
{
    Thread t = new Thread(() =>
    {
        int x = 0;
        while (x < 100)
        {
            Console.Write(x++);
        }
    });
    t.Name = " Thread_X ";
    t.Start();
    Console.Write(t.Name);
    t.Name = "NewName";
}
```

Each thread has a **Name** property that you can set for the benefit of debugging.

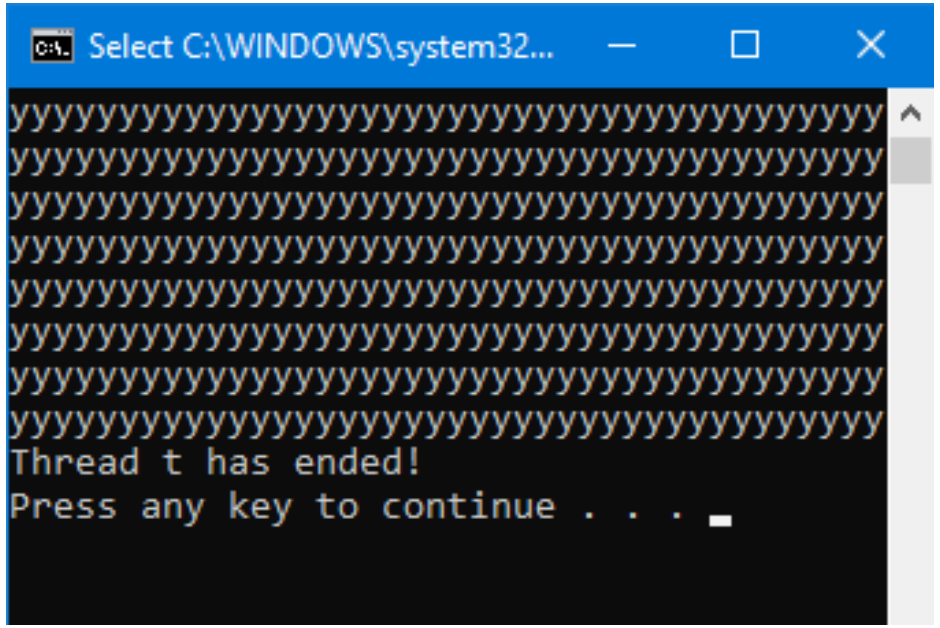
You can set a thread's name just **once**; attempts to change it later will throw an exception.



Join and Sleep

You can wait for another thread to end by calling its Join method

```
class Program
{
    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");
    }
}
```



Join and Sleep

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

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.

```
Thread.Sleep(0);
```

```
Thread.Yield();
```

While waiting on a **Sleep** or **Join**, a thread is ***blocked***.

Blocking and ThreadState

```
bool blocked = (someThread.ThreadState & ThreadState.WaitSleepJoin) != 0;
```

```
ThreadState.
```

AbortRequested
Background
Running
Stopped
StopRequested
Suspended
SuspendRequested
Unstarted
WaitSleepJoin

ThreadState.Aborted = 256
The thread state includes ThreadState.AbortRequested and the thread is now dead, but its st

Most of **ThreadState** values are either unused or deprecated.

The useful are: **Unstarted**, **Running**, **WaitSleepJoin**, **Stopped**

I/O-bound versus compute-bound

- Web-page load
- Console.ReadLine
- Thread.Sleep

vs

- Any CPU-intensive work

Local State

A separate copy of the *cycle* variable is created on each thread's memory stack

```
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('?');
}
```

Shared State

```
class ThreadTest
{
    bool _done;
    static void Main()
    {
        ThreadTest tt = new ThreadTest(); // Create a common instance
        new Thread(tt.Go).Start();
        tt.Go();
    }
    void Go() // Note that this is an instance method
    {
        if (!_done) { _done = true; Console.WriteLine("Done"); }
    }
}
```

Local variables captured by a lamda

```
class ThreadTest
{
    static void Main()
    {
        bool done = false;
        ThreadStart action = () =>
        {
            if (!done) { done = true; Console.WriteLine("Done"); }
        };
        new Thread(action).Start();
        action();
    }
}
```

Shared static fields

```
class ThreadTest
{
    static bool _done; // Static fields are shared between all threads
                       // in the same application domain.
    static void Main()
    {
        new Thread(Go).Start();
        Go();
    }
    static void Go()
    {
        if (!_done) { _done = true; Console.WriteLine("Done"); }
    }
}
```

Is it thread safe though?

How about that?

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

Avoid shared state if possible

Locking and Thread Safety

```
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; }
        }
    }
}
```

Passing Data to a Thread

Lambda wrapped method call

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

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

Straightforward lambda

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

Passing Data to a Thread

Long ago before lambda expressions existed (prior to C# 3.0)

```
class Program
{
    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);
    }


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

Thread's constructor overloads take either of these

Only one argument, also typecasting is usually needed

Lambda expressions and captured variables

```
static void Bad()
{
    for (int i = 0; i < 10; i++)
        new Thread(() => Console.Write(i)).Start();
    //The output is unknown! Here's a typical result:
    //0223557799
}

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

Lambda expressions and captured variables

Simple example

```
static void Main(string[] args)
{
    string text = "t1";
    Thread t1 = new Thread(() => Console.WriteLine(text));
    text = "t2";
    Thread t2 = new Thread(() => Console.WriteLine(text));
    t1.Start();
    t2.Start();
}
```

Both lambda expressions capture the **same** text variable, t2 is printed **twice**.

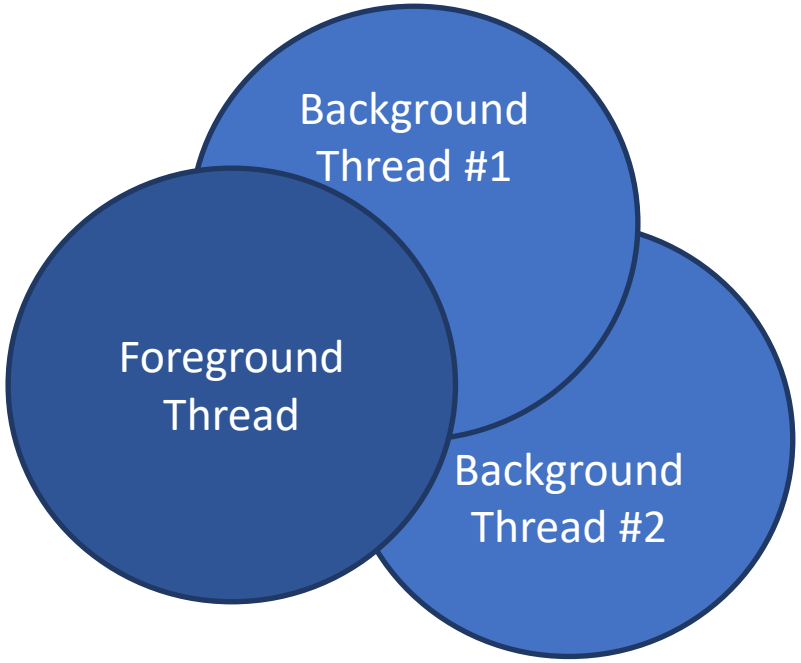
Exception Handling

```
class 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
}
```

Exception Handling

```
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
        // ...
    }
}
```

Foreground Versus Background Threads



A thread's foreground/background status has no relation to its *priority* (allocation of execution time).

Foreground Versus Background Threads

You can query or change a thread's background status using its **IsBackground** property:

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

Thread Priority

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

Thread t = new Thread(() =>
{
    int x = 0;
    while (x < int.MaxValue)
        x++;
});
t.Priority = ThreadPriority.Highest;
t.Start();


using System.Diagnostics;


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


Normally you don’t want to do this.
Just don’t mess up the priorities..
Ever!

Signaling

```
var signal = new ManualResetEvent(false);
new Thread(() =>
{
    Console.WriteLine("Waiting for signal...");
        signal.WaitOne();
        signal.Dispose();
        Console.WriteLine("Got signal!");
}).Start();
Thread.Sleep(2000);
signal.Set(); // "Open" the signal
```

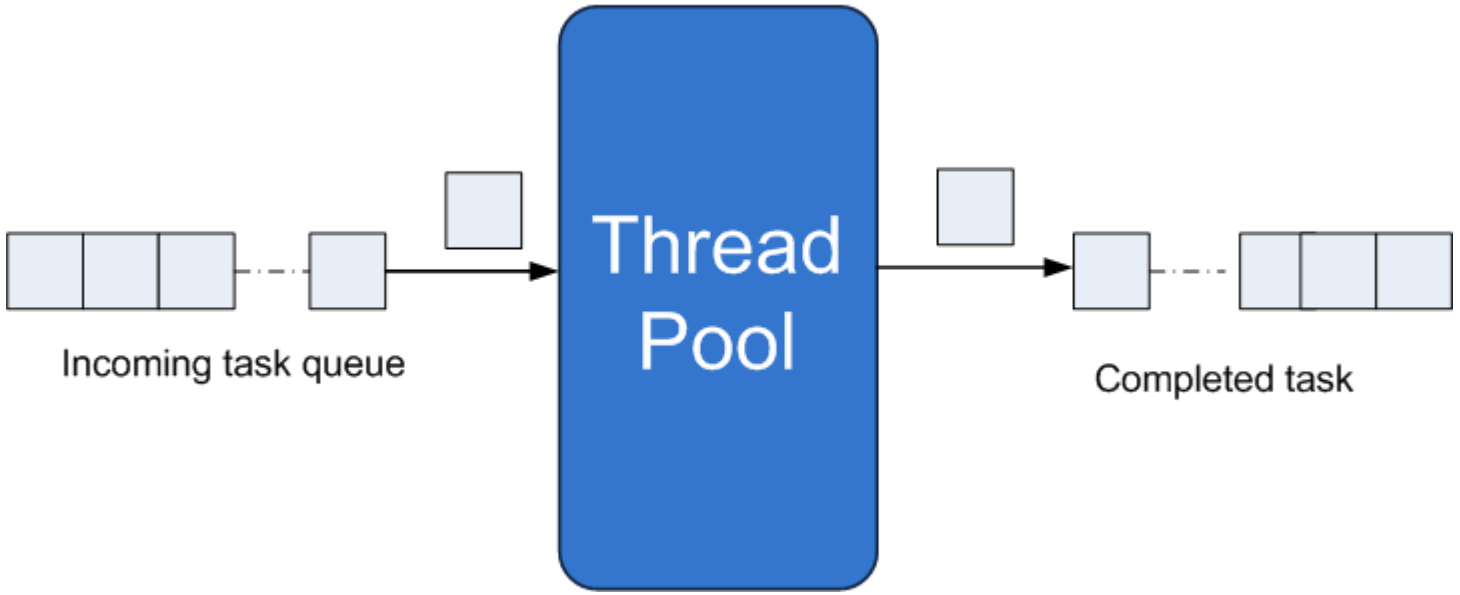
Threading in Rich-Client Applications

```
partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
        new Thread(Work).Start();
    }
    void Work()
    {
        Thread.Sleep(5000); // Simulate time-consuming task
        UpdateMessage("The answer");
    }
    void UpdateMessage(string message)
    {
        Action action = () => txtMessage.Text = message;
        Dispatcher.BeginInvoke(action);
    }
```

Synchronization Contexts

```
partial class MainWindow : Window
{
    SynchronizationContext _uiSyncContext;
    public MainWindow()
    {
        InitializeComponent();
        // Capture the synchronization context for the current UI thread:
        _uiSyncContext = SynchronizationContext.Current;
        new Thread(Work).Start();
    }
    void Work()
    {
        Thread.Sleep(5000); // Simulate time-consuming task
        UpdateMessage("The answer");
    }
    void UpdateMessage(string message)
    {
        // Marshal the delegate to the UI thread:
        _uiSyncContext.Post(_ => txtMessage.Text = message, null);
    }
}
```

The Thread Pool



The Thread Pool

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
- Pooled threads are always ***background threads***.
- Blocking pooled threads can degrade performance
- You are free to change the priority of a pooled thread—it will be restored to normal when released back to the pool.

Entering the thread pool

```
class Program
{
    static void Main(string[] args)
    {
        // Task is in System.Threading.Tasks
        Task.Run(() => Console.WriteLine("Hello from the thread pool"));
    }
}
```

Prior to Framework 4.0

```
    //System.Threading
    ThreadPool.QueueUserWorkItem(notUsed => Console.WriteLine("Hello"));
}
```

Thread pool use

- WCF, Remoting, ASP.NET, and ASMX Web Services application servers
- System.Timers.Timer and System.Threading.Timer
- The parallel programming constructs
- The (now redundant) BackgroundWorker class
- Asynchronous delegates (also now redundant)

Tasks

Limitations of using threads:

- While it's easy to pass data into a thread that you start, there's no easy way to get a "return value" back from a thread that you *Join*. You have to set up some kind of shared field. And if the operation throws an exception, catching and propagating that exception is equally painful.

- You can't tell a thread to start something else when it's finished; instead you must Join it (blocking your own thread in the process).

Tasks

- Task is a higher-lever abstraction
- Task respresents concurrent operation
- It may or may not be backed by a thread
- You can chain Tasks (through the use of *continuations).*
- Tasks can use ThreadPool
- TaskCompletionSource and callbacks

Starting a Task

Framework 4.0

```
Task.Factory.StartNew(() => Console.WriteLine("Foo"));
```

Framework 4.5

```
Task.Run(() => Console.WriteLine("Foo"));
```

Both are actually the same. **Task.Run** is sorta a shortcut to a **Task.Factory.StartNew**

Equivalent with thread would be:

```
new Thread(() => Console.WriteLine("Foo")).Start();
```

Q&A

UA .NET Online LAB