**Why It’s Important**

However, we developers used to write methods, I would say synchronous methods and try to call them asynchronously, through many ways (Thread, ThreadStart, ThreadPool, BackgroundWorker, etc.), but writing asynchronous methods in nature was somehow hard to do and maintain.

The feature I am to talk about enables us to create asynchronous methods so easily on the fly, as a matter of fact, it helps us to change our traditional synchronous methods into asynchronous ones.

**The Task Class**

The feature can be summarized by the following example where we changed the method to an asynchronous one inherently or by nature, simply by changing the return type.

Let’s consider a long running method:

public static IEnumerable<int> getPrimes(int min, int count)

{

return Enumerable.Range(min, count).Where

(n => Enumerable.Range(2, (int)Math.Sqrt(n) - 1).All(i =>

n % i > 0));

}

Depending on the parameters min and count, this method can take a long time.

One way to make it asynchronous is to simply change the return type as in the following example:

public static Task<IEnumerable<int>> getPrimesAsync(int min, int count)

{

return Task.Run (()=> Enumerable.Range(min, count).Where

(n => Enumerable.Range(2, (int)Math.Sqrt(n) - 1).All(i =>

n % i > 0)));

}

In the example above, please notice that we changed the name of the method itself by adding Async and that is the convention to be followed.

The return type is Task in case it was void and Task<T> where T is the return type of method.

**What is Task?**

Task is simply an implementation to IAsynchResult, that’s why I had to mention the article I wrote before or the problem I encountered and the solution I had to reach.

Now when a method returns Task<T> then it is awaitable and that means you can call it using the keyword await, and that means whenever you call it using await, the execution control will come back to you immediately and there will be no impact on the responsiveness of your application.

Let's see how we call these methods above:

private static void PrintPrimaryNumbers()

{

for (int i = 0; i < 10; i++)

getPrimes(i \* 100000 + 1, i \* 1000000)

.ToList().

ForEach(x => Trace.WriteLine(x));

}

PrintPrimaryNumbers() is a method that we can call directly and traditionally, I am calling it here 10 times, it will be called consequently and we will see how long it will take to finish.

private static async void PrintPrimaryNumbersAsync()

{

for (int i = 0; i < 10; i++)

{

var result = await getPrimesAsync(i \* 100000 + 1, i \* 1000000);

result.ToList().ForEach(x => Trace.WriteLine(x));

}

}

Whilst PrintPrimaryNumbersAsync() is decorated by the keyword async and it calls getPrimesAsync asynchronously.

Once it calls, the execution immediately returns to the caller (the main thread).. and once any of the other threads is done, it will get the control back (in our case, write the primary numbers in found in the range supplied).

To make the picture clearer, run and minimize the range like in the source files attached.

Now let’s see the main function:

static void Main(string[] args)

{

DateTime t1 = DateTime.Now;

PrintPrimaryNumbers();

var ts1 = DateTime.Now.Subtract(t1);

Trace.WriteLine("Finished Sync and started Async");

var t2 = DateTime.Now;

PrintPrimaryNumbersAsync();

var ts2 = DateTime.Now.Subtract(t2);

Trace.WriteLine(string.Format(

"It took {0} for the sync call and {1} for the Async one", ts1, ts2));

Trace.WriteLine("Any Key to terminate!!");

}

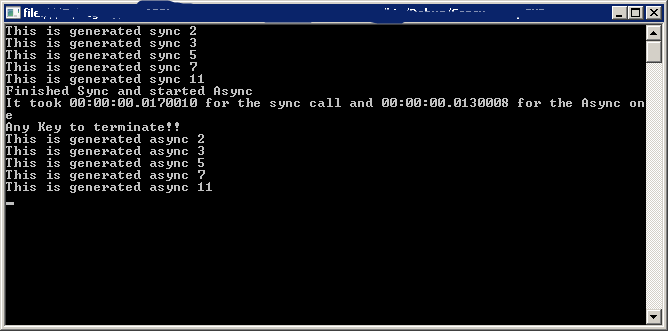
Can you tell the difference between ts1 and ts2?

Here is the result:

Finished Sync and started Async

It took 00:32:16.1627422 for the sync call and 00:00:00.0050003 for the Async one

If you choose a small range for test, you can see something like:



It is very important to notice that the time measured here is not the time that took the async operation to complete, it is the time that took the async operation to kick in or start, but it did not block the main thread and that was the nice catch about it, if you wait for this and waited for the results to come and then measure the time then you will know the time accurately taken by the async call.

Usually we use Task.WaitAny(...) or Task.WaitAll(...) to keep the main thread waiting till the asynchronous calls to finish, and that is what I have not done here and maybe should have.

Console.ReadLine() will hold the main thread till you click any key, expecting that you will wait to see the results before you terminate it.

In short, the example shows you that you can easily run asynchronous calls without blocking the main thread but it did not show how to get the results in an adequate manner.

**Summary**

.NET platform 4.5 has made some revolutionary technique in the asynchronous programming and gave up some old techniques that are now called obsolete (BackgroundWorker, Event Asynchronous Programming, Asynchronous Programming Model APM).

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace ContextRunner

{

static class Program

{

/// <summary>

/// The main entry point for the application.

/// </summary>

[STAThread]

static void Main()

{

Application.EnableVisualStyles();

Application.SetCompatibleTextRenderingDefault(false);

var context = SynchronizationContext.Current;

if (context == null)

MessageBox.Show("No context for this thread");

else

MessageBox.Show("We got a context");

Form1 form = new Form1();

context = SynchronizationContext.Current;

if (context == null)

MessageBox.Show("No context for this thread");

else

MessageBox.Show("We got a context");

Application.Run(new Form1());

}

}

}

**Post aync result to the main UI thread**

using System;

using System.Diagnostics;

using System.Threading;

using System.Windows.Forms;

namespace ContextRunner

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

private void button1\_Click(object sender, EventArgs e)

{

Trace.WriteLine("mToolStripButtonThreads\_Click thread: " + Thread.CurrentThread.ManagedThreadId);

// grab the sync context associated to this

// thread (the UI thread), and save it in uiContext

// note that this context is set by the UI thread

// during Form creation (outside of your control)

// also note, that not every thread has a sync context attached to it.

SynchronizationContext uiContext = SynchronizationContext.Current;

// create a thread and associate it to the run method

Thread thread = new Thread(Run);

// start the thread, and pass it the UI context,

// so this thread will be able to update the UI from within the thread

thread.Start(uiContext);

}

private void Run(object state)

{

Trace.WriteLine("Run thread: " + Thread.CurrentThread.ManagedThreadId);

SynchronizationContext uiContext = state as SynchronizationContext;

for (int i = 0; i < 10; i++)

{

// normally you would do some code here to grab items from the database.

// or some long computation

Thread.Sleep(10);

// use the ui context to execute the UpdateUI method,

// this insure that the UpdateUI method will run on the UI thread.

uiContext.Post(UpdateUI, "line " + i.ToString());

//try

//{

// uiContext.Send(UpdateUI, "line " + i.ToString());

//}

//catch (Exception e)

//{

// Trace.WriteLine(e.Message);

//}

}

}

/// <summary>

/// This method is executed on the main UI thread.

/// </summary>

private void UpdateUI(object state)

{

Trace.WriteLine("UpdateUI thread:" + Thread.CurrentThread.ManagedThreadId);

string text = state as string;

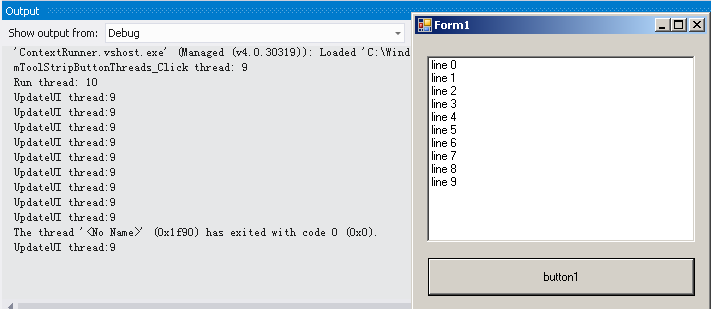
mListBox.Items.Add(text);

//throw new Exception("Boom");

}

}

}



// \*\*\*

//This means that Post will not wait for the execution of the delegate to complete.

//Post will "Fire and Forget" about the execution code within the delegate.

//It also means that you cannot catch exceptions as we did with the Send method.

//Suppose an exception is thrown, it will be the UI thread that will get it;

//unhanding the exception will terminate the UI thread.

\*\*\* When try-catch added, exception will be thrown to the Run function, not arrived UI thread, in this case.

**APM, EAP->TPL**

* The Asynchronous Programming Model (**APM**) is the model you see with BeginMethod(...) and EndMethod(...) pairs.
* The Event-based Asynchronous Pattern (**EAP**) is the model you see with MethodAsync(...) and CancelAsync(...) pairs. There's usually a Completed event. BackgroundWorker is a good example of this pattern. (EAP was introduced in the .NET Framework version 2.0.)
* As of C# 4.5, both have been replaced by the async/await pattern, which is using the Task Parallelism Library (**TPL**). You will see them marked with Async after the method name and usually returning an **awaitable** Task or Task<TResult>. If you are able to target .NET 4.5, you should definitely use this pattern over the APM or EAP design.

