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!!");
}
```

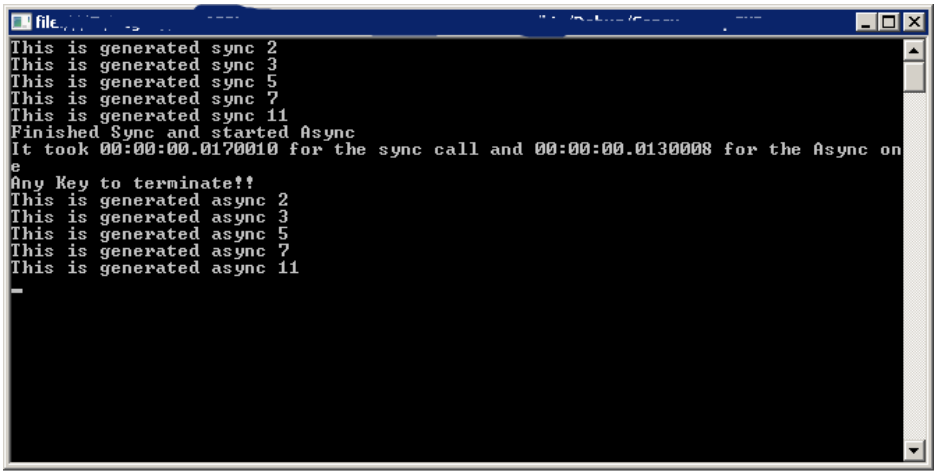
Can you tell the difference between `ts1` and `ts2`?

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).