An Async Primer

By Bill Wagner
August 2012

Introduction

The C# 5.0 release might seem small, given that the single major feature is the addition of async / await keywords to support asynchronous programming. While this doesn't seem to change the language specification very much, the addition of these features will have a profound effect on the programs you write every day.

Asynchronous methods mean that your methods don't always return complete results synchronously. They may return the promise of results at some future moment. Programming with future results changes the way you write code in profound ways. In this article, I'll provide an overview of the async features in C# 5, and discuss some of the ways you should change your regular programing habits to take advantage of async programming features.

One short article is not nearly enough space to cover asynchronous programming, and how these changes will affect you. This article will provide you with overview of the concepts you'll use writing and maintaining asynchronous programs using the new language and library features coming shortly.

I'm going to walk you through the tasks to convert a small WPF application from a synchronous model to an asynchronous model of execution. Download the Async Primer sample code here. I encourage you to run each of the versions in the solution as your read the article. Seeing how the program behaves differently as you modify it from synchronous to asynchronous execution will help you understand the discussion.

The application downloads and displays the RSS feeds from a few prominent C# bloggers. The initial version downloads those feeds synchronously:

```
private void retrieveAllFeeds()
{
   var webClient = new WebClient();


   foreach (var addr in feedAddrs)
   {
      var feed = new Feed(this);
      var content = webClient.DownloadString(addr);
      var xml = XElement.Parse(content);
```

```
        feed.Title = xml.TitleFromFeed();


        var items = from item in xml.FeedItems()
                    select new FeedItem
                    {
                       Title = item.Item1,
                       Description = item.Item2
                    };
        feed.AddItems(items);
        feeds.Add(feed);
    }
}
```

Run this sample, and you'll see that the display is unresponsive during the entire process of downloading the feeds. Even though the code executes a loop adding each feed, the display is not updated until all feeds have been downloaded and added to the list of feeds. That's because this code executes synchronously. You've pressed a button to initiate the downloads. That code executes on the UI thread. The UI thread is busy waiting for a response and cannot update its display while it is waiting for the feeds to return. Users will wonder if the application is hung, or if it is waiting for a response. The UI thread cannot process any messages (to draw windows, move the window, or anything else) until this method returns. This situation is the motivation for making it easier to write asynchronous code.

Introducing async and await

A very simple and small change will make a large difference to your users.

```
private async void retrieveAllFeeds()
{
    var webClient = new HttpClient();


    foreach (var addr in feedAddrs)
    {
        var feed = new Feed(this);
        Task<string> future = webClient.GetStringAsync(addr);
        string content = await future;
```

```
        var xml = XElement.Parse(content);
        feed.Title = xml.TitleFromFeed();

        var items = from item in xml.FeedItems()
                    select new FeedItem
                    {
                       Title = item.Item1,
                       Description = item.Item2
                    };
        feed.AddItems(items);
        feeds.Add(feed);
    }
}
```

The three changes above make this code asynchronous. Instead of using DownloadString, you call GetStringAsync (a member of HttpClient). The GetStringAsync method returns a Task <string>, where DownloadString returns the string. Task<string> represents a task that will produce a string at some later moment in time. The Task <string> can be *awaited*. Finally, the entire method must be marked with the *async* modifier because it now contains an await expression.

Run this version and you'll notice several changes immediately. The window updates as each feed is retrieved. You can interact with the window while the application is retrieving the feeds: you can move it, maximize it, or select an item.

Those small code changes have a pretty big effect. Let's concentrate on the two keywords: async, and await. The async modifier notifies the compiler that a method contains code that will execute asynchronously. The await keyword indicates where your code requires the result to be available.

Await instructs the compiler that the code which follows must not execute until the result of the awaited expression is available. In the example above the await expression indicates that the code following await relies on the string result of the DownloadStringTaskAsync() call. The compiler does a fair amount of work to rearrange code for you when you await something. My explanation will gloss over quite a few details, but it will give you a good conceptual view of what happens.

In most cases, this method returns when it reaches the await expression. But this method can't simply return; the rest of the method, including the remaining iterations on the loop would not execute. The compiler generates code to ensure that the remaining logic in the method executes when the result becomes available. The generated code performs the following logic:

1. If the result is already available, just keep on executing as though the method were synchronous.
2. Otherwise, do the following:
    a. Create a delegate contains the remaining logic in the method.
    b. Ensure that the delegate executes when the awaited expression completes.
    c. Return.

It sounds relatively simple. However, wrap your head around the code the compiler must generate for the await expression inside the loop, and you'll quickly realize that this is a very complicated problem for the compiler team to solve. What they've done is enabled us to write code in essentially the same style that we would write for synchronous execution. The compiler rewrites the code into the constructs necessary to transform that code into a correct asynchronous version.

The simple addition of async and await in the sample above instructs the compiler to perform that work. Going back to the sample, you have more work to do if you want to leverage these new features to the fullest.

The Importance of Task and the Await Pattern

Your most common use of await will be to await an expression that returns either at Task <T> or a Task. However, await is pattern based, not explicitly tied to the Task class or an interface implemented by Task and Task <T>. Any expression is awaitable if:

It is of type dynamic, or the type of the expression has an accessible (instance or extension method) GetAwaiter() with no parameters. Furthermore, the GetAwaiter() method must return a type that has following accessible instance methods:

bool IsCompleted { get; }
void OnCompleted(Action);
void GetResult() or T GetResult() for any type T.

This is a similar strategy to the way the LINQ expression pattern works. In the same way that you most often use the LINQ expression pattern through methods accessible via IEnumerable<T>, you'll most often use the await pattern with the Task and Task<T> types. The Task and Task<T> types support this pattern (Task's awaiter contains a void GetResult(), and Task<T>'s awaiter contains a T GetResult() method). For the remainder of the article, where I mention Task, everything I say applies to Task and Task<T>. In addition, WinRT contains types that satisfy this pattern: you can use the same syntax for asynchronous WinRT development.

The Task class embodies the concept of an asynchronous result for .NET developers. A Task object represents a result that may be available now, but if it is not available now, it will be available in the future (assuming the operation completes successfully). Task objects are how you will program the concepts related to this work that may not be completed yet.

The first asynchronous version of the feed aggregator does the minimum possible work to be asynchronous. All that's been achieved so far is to keep the UI responsive while the program waits for feeds to return. The program still requests and receives one feed before requesting any other feeds. One obvious improvement you can make is to start all the web requests before awaiting for any of the results. To do that, you'll need to store the Task<string> variables and await each of the tasks and adding each feed to the UI. Here's an updated version of retrieveAllFeeds(), and new method retrieveOneFeed() that starts all the web requests before awaiting:

```
private async Task retrieveAllFeeds()
{
   var tasks = new List<Task<Feed>>();
   foreach (var addr in feedAddrs)
   {
      var feedTask = retrieveOneFeed(addr);
      tasks.Add(feedTask);
   }
   Feed[] allFeeds = await Task.WhenAll(tasks);
   foreach (var feed in allFeeds)
      feeds.Add(feed);
}

private async Task<Feed> retrieveOneFeed(string url)
{
   var service = new HttpClient();
   var feed = new Feed(this);
   Task<string> future = service.GetStringAsync(url);
   string content = await future;
   var xml = XElement.Parse(content);
   feed.Title = xml.TitleFromFeed();

   var items = from item in xml.FeedItems()
               select new FeedItem
```

```
            {
                Title = item.Item1,
                Description = item.Item2
            };
    feed.AddItems(items);
    return feed;
}
```

Run this version, and you'll notice a new changes in behavior. First, you may notice the window updates a bit more quickly. Second, the gap between each feed being added is much smaller. That's because all of the web requests have been made before awaiting any of the results.

Let's begin with retrieveOneFeed and examine the changes from the code extracted from the previous version of retrieveAllFeeds. An await expression applied to a Task<T> has type T, or has type void when applied to a Task. Here, that means the variable 'content' is assigned to the string result of the task 'future'. Next, notice that retrieveOneFeed() returns a Task<Feed>, while the return statement returns an object of type Feed. The compiler generates the extra code to create the Task<string> returned by this async method, and generates the code to store the feed in that compiler generated Task<string> object.

The other changes are in the retrieveAllFeeds() method. This version now contains two loops: one that initiates all the web requests to download the content, and a second that processes each result. The first loop builds a list of tasks, and the second adds the feed content to the list of feeds.

Finally, this method now has the return type of Task instead of void. The compiler generates a Task object that callers can use to determine if the work has finished. Notice again that the method does not contain a return statement. The compiler generates the code to return the Task.

This last point is an important one to remember: In almost all cases, your async methods should return Task or Task<T>. Async methods that have a void return should be defined only for event handlers when the method signature enforces a void return type. In fact, you can use async lambda expressions as event handlers:

```
public RSSReaderViewModel()
{
    getFeeds = new DelegateCommand(async () => await retrieveAllFeeds(), () => true);
}
```

This version of the code moves from the "fire and forget" asynchronous code you saw in the previous

version. This version is manipulating task objects. As you delver further and further into writing async code with C# 5, you'll find yourself programming with Task and Task<T> much more often.

When Exceptions Happen

We've discussed how async methods return and yet still execute more code later: the portion of code before any await expression executes synchronously and returns when it must await on an asynchronous operation.

This introduces an interesting problem for handling exceptions. Consider the following changes to retrieveOneFeed (commented out):

```
private async Task<Feed> retrieveOneFeed(string url)
{
   //throw new ArgumentException("Just a trial exception");
   var service = new HttpClient();
   var feed = new Feed(this);
   var future = service.GetStringAsync(url);
   var content = await future;
   var xml = XElement.Parse(content);
   feed.Title = xml.TitleFromFeed();

   var items = from item in xml.FeedItems()
               select new FeedItem
               {
                  Title = item.Item1,
                  Description = item.Item2
               };
   feed.AddItems(items);
   // throw new InvalidOperationException("Just another trial exception");
   return feed;
}
```

The retrieveOneFeed() method contains code to throw exceptions either before or after the await. If

you uncomment both of the lines of code, could it throw in both locations? What should happen in any of those cases?

Let's examine this from the perspective of retrieveOneFeed() first. Most developers would read the code and assume that if the first line was uncommented, the rest of the retrieveOneMethod() method would not be executed. There's no await! Throwing an exception normally moves execution to the appropriate catch clause, after executing any finally clauses that apply. Under no circumstances would a developer expect the code following the await to execute. The language follows that expectation, and any code that follows the await is not executed if an exception is thrown earlier in the method.

Now, think about what you would expect at the calling code. Where would you expect to catch either of those potential exceptions? Should you catch exceptions that are thrown before the await where you call retrieveOneFeed()? Should you catch exceptions that occur after the await when you await the result? Should you place catch clauses in both places? (Remember that if the result is available when an async method awaits, the remaining code executes synchronously.) There are two possible choices for the language team to have made: They could have chosen to throw the exception when it was generated, either synchronously or asynchronously, depending on runtime conditions and whether the asynchronous operation completes before or after it is awaited. Or, the language could chose to always return any exceptions asynchronously, when the caller awaits the result.

The advantage of the first choice is that it does provide a bit more information: exceptions that happen synchronously would be thrown synchronously and exceptions that happen asynchronously would be thrown asynchronously. You can see how it might be possible to diagnose and recover from problems more quickly if that were the case.

There are also serious disadvantages with this idea. Without detailed knowledge of the implementation of an asynchronous methods, you can't know if a method might throw exceptions synchronously, asynchronously, or both. And, the same line of code might throw either synchronously or asynchronously, depending on when an operation completes.

Faced with those disadvantages, the team chose to always throw exceptions asynchronously. Exceptions that arise from an async method are retrown when code awaits that task. That means the calling code catches both synchronous and asynchronous exceptions by placing the await expression in a try block.

The feed aggregator needs one simple change to its retrieveAllFeeds() method:

```
private async Task retrieveAllFeeds()
{
   var tasks = new List<Task<Feed>>();
   foreach (var addr in feedAddrs)
   {
      var feedTask = retrieveOneFeed(addr);
      tasks.Add(feedTask);
   }
```

```
    try
    {
        foreach (var feed in await Task.WhenAll(tasks))
            feeds.Add(feed);
    }
    catch (ArgumentException e2)
    {
    }
    catch (InvalidOperationException e3)
    {
    }
}
```

Whether retrieveOneFeed() throws an exception before or after making the web request, this try clause will catch any errors. (To demonstrate that in the sample, I changed one of the feed addresses to a page that returns a 404 error). Readers that are familiar with the Task Parallel library may be surprised that catch clause looks for an ArgumentException, or an InvalidOperationException, not an AggregateException. The Task Parallel Library APIs (Task.Result, Task.Wait() and related methods) and PLINQ will collect the exceptions thrown from all asynchronous operations they use. If there are any exceptions (one, or more than one), the method will throw an AggregateException whose collection contains all the exceptions thrown by the asynchronous operation. When you use await, the code generated by the compiler unwraps the AggregateException and throws the underlying exception. By leveraging await, you avoid the extra work to handle the AggregateException type used by Task.Result, Task.Wait, and other Wait methods defined in the Task class. That's another reason to use await instead of the underlying Task methods.

You're Just Beginning

As I mentioned in the opening, the syntax added to support asynchronous programming seems minor: just two new keywords, async and await. But the move to asynchronous programming will have a profound effect on your everyday coding. Programmers assume that their code executes as a strict progression of cause to effect, but actually from a non-linear, non-subjective viewpoint – it's more like a big ball of wibbly wobbly … time-y wimey … stuff. With apologies to Dr Who, your programs now execute in a different manner than you thought. These features give you greater ability to keep your client applications responsive, and yet keep the code that expresses those concepts simple.