Async Programming in Visual Studio 2010

By Bill Wagner
November 2011

Introduction

The latest MSDN magazine (Oct 2011) features three articles on the upcoming language and library features that support asynchronous programming. Those articles and the recent Visual Studio 11 CTP are great ways to learn about these new features, and grow your skills in async programming in C#.

Many developers I work with can't necessarily use the new CTP just yet. For them, an equally important question is how to structure your current production codebases to support asynchronous operations while working on the released versions of the C# language and .NET libraries. In this article, I'll discuss how you can use current language features and libraries to support asynchronous programming. Most importantly, I'll show you how to do so in a way that makes it as easy as possible to adopt the upcoming language features quickly when you're ready to update your production codebase. You'll want to adopt the new language features as soon as you can; the new features greatly improve the experience.

Simple mapping of keywords to features

Eric Lippert's article in the Oct 2011 issues of MSDN Magazine (http://msdn.microsoft.com/en-us/magazine/hh456401.aspx) provides a wonderful description of how the new language features make it easier to write asynchronous code. He showed this sample code in his article, demonstrating the new async features in the language:

```
async void ServeBreakfast(Diner diner)
{
   var order = await ObtainOrderAsync(diner);
   var ingredients = await ObtainIngredientsAsync(order);
   var recipe = await ObtainRecipeAsync(order);
   var meal = await recipe.PrepareAsync(ingredients);
   diner.Give(meal);
}
```

Both of the new keywords, async and await are used in this small sample. The async keyword specifies that the method behaves like a resumable method. It returns before it has completed all its work, but it

will resume and finish when it can. When an async method returns a value (Task<T>), it returns a *future.* A future is an object that will, at some later time, contain the results of method. In this case, The above snippet returns void, which I'll discuss later. Usually async methods will return a Task<T>, where T is the type of the results being generated. A 'future' is an object that represents the results of work that may not have completed. The caller can examine the properties of the future to see if the work has completed, and if it has, the future will contain the results. The choice of leveraging Task<T> for this purpose often leads to the misconception that async methods run in parallel and create new threads. That's not correct. The languages teams chose to leverage the Task<T> type, from the Task Parallel Library (TPL), because its public contract accurately represents the concept of a future. The thing doing that work may be a background thread, an I/O completion thread, a graphic processing unit, or even the current thread. It doesn't matter to the caller.

Await signifies two related things. First, it says that everything remaining in the method becomes the *continuation* for the method. A 'continuation' represents where the method will continue once the expression being awaited completes its work. Secondly, it tells the method to return. The method will return a future (if it returns anything), which represents the future result of the awaited expression, and the remaining work in the method expressed as a continuation.

The async and await keywords can feel like magic. But there is no magic; there is only code the compiler creates for you. By translating this example into C# 4, you'll see conceptually what it does. (The actual code is somewhat more complicated). You'll also see how to add these features into your current codebase in such a way that your code will adapt easily when you're ready to integrate the next version of C# into your applications. Let's get started.

I find it easier to unwind these methods by starting at the end. The last step is to prepare the meal (asynchronously), and when the meal is ready, serve it to the diner. The Task<T>.ContinueWith() method is how to express what happens when a method finishes in C# 4. ContinueWith() takes an Action<Task<T>> as its parameter. That means you have to examine the result to retrieve the actual result. Of course, the entire method is asynchronous, so it still must await for the last step to finish. Here's that first refactoring:

```
async void ServeBreakfast(Diner diner)
{
   var order = await ObtainOrderAsync(diner);
   var ingredients = await ObtainIngredientsAsync(order);
   var recipe = await ObtainRecipeAsync(order);
   var finished = recipe.PrepareAsync(ingredients)
      .ContinueWith(m => diner.Give(m.Result));
   await finished;
}
```

You'll begin to notice a pattern here very quickly. Working up, you can see that the previous line obtains all the ingredients, and then continues to work on preparation (and serving):

```
async void ServeBreakfast(Diner diner)
{
   var order = await ObtainOrderAsync(diner);
   var ingredients = await ObtainIngredientsAsync(order);
   var finished = ObtainRecipeAsync(order)
      .ContinueWith(r => r.Result.PrepareAsync(ingredients)
         .ContinueWith(m => diner.Give(m.Result)
      )
   );
   await finished;
}
```

Now that you've seen two of them, and you think you've got it down, it's time to throw some changes at you. Notice that the second call above gets the ingredients, which aren't needed until later. You need to keep track of that result, but you don't need to wait for it. You can just use the Task<T> to continue that asynchronous work without waiting:

```
async void ServeBreakfast(Diner diner)
{
   var order = await ObtainOrderAsync(diner);
   var ingredients = ObtainIngredientsAsync(order);
   var finished = ObtainRecipeAsync(order)
      .ContinueWith(r => r.Result.PrepareAsync(ingredients.Result)
         .ContinueWith(m => diner.Give(m.Result)
      )
   );
   await finished;
}
```

Notice that I'm just calling ingredients. Result to retrieve the result of obtaining the ingredients. That's different than awaiting the result: Task.Result will block until the results are available, and will not return. That's ok in this case, because we would be awaiting it immediately. Next, let's remove one more await from the method:

```
async void ServeBreakfast(Diner diner)
{
   var order = ObtainOrderAsync(diner);
   var ingredients = ObtainIngredientsAsync(order.Result);
   var finished = ObtainRecipeAsync(order.Result)
      .ContinueWith(r => r.Result.PrepareAsync(ingredients.Result)
         .ContinueWith(m => diner.Give(m.Result)
      )
   );
   await finished;
}
```

OK, we're getting close. That await at the end still isn't valid C# 4. One possible change is to wait for the final results in the ServeBreakfast method:

```
void ServeBreakfast(Diner diner)
{
   var order = ObtainOrderAsync(diner);
   var ingredients = ObtainIngredientsAsync(order.Result);
   var finished = ObtainRecipeAsync(order.Result)
      .ContinueWith(r => r.Result.PrepareAsync(ingredients.Result)
         .ContinueWith(m => diner.Give(m.Result)
      )
   );
   finished.Wait();
}
```

That's now valid C# 4, but it's made another semantic change to the method. This version does not return a future, but blocks until the diner has been served his breakfast. To fix that, we need to change the signature of the method to return a Task, which does behave like a future:

```
Task ServeBreakfast(Diner diner)
{
   var order = ObtainOrderAsync(diner);
   var ingredients = ObtainIngredientsAsync(order.Result);
   var finished = ObtainRecipeAsync(order.Result)
      .ContinueWith(r => r.Result.PrepareAsync(ingredients.Result)
         .ContinueWith(m => diner.Give(m.Result)
      )
   );
   return finished;
}
```

This version performs much the same function as the original. The caller must hold onto the task object, and decide when to wait for that task to finish. The C# 5 compiler does not actually make the translation I just made. In C# 5, void returning async methods return void, not a Task. The caller cannot await its completion. Void returning async methods should be used only for 'fire and forget' scenarios. Specifically, these would be in event handlers. The vast majority of the async methods you will write (in C# 4 or C# 5) will return Task<T>.

However, I made many simplifications to the code that the compiler will create on your behalf. The most important change is that C# 5 will generate all the code to handle exceptions thrown by asynchronous tasks. I have not converted that code to this version. Also, I've not shown any of the work to handle marshaling the continuations onto the same thread that started the method. That work is also done by the compiler in C# 5.0.

One simple rule: Leverage Task<T>

This simplified back-porting of a C# 5 sample highlights the most important recommendation I can give you so that you structure your code today so that it's ready for C# 5.0. The new compiler functionality makes extensive use of the capabilities in the Task<T> and Task classes. By working with those classes, and using those types for the returns of your asynchronous methods, your code will be structured so that it will be easiest to migrate to the new syntax when you're ready to adopt C# 5 as part of your production toolset. Your asynchronous methods should return Task<T>, for whatever type of data they produce. They can also return Task, for those asynchronous operations that produce side effects, but do not return results. The most common way to start an asynchronous task in C# 4 is to use the

Task.Factory.StartNew method. For example, the ObtainRecipeAsync method might look something like this:

```
private Task<Recipe> ObtainRecipeAsync(Order order)
{
   return Task.Factory.StartNew(() =>
      {
         // elided
         return new Recipe(order);
      });
}
```

Locations where you want to use await can be translated into Task.ContinueWith() calls. That's not exactly how the compiler handles await, but it does get the job done with the current tools. For more details on the code that the compiler does generate with C# 5, see Mads Torgersen's article in the Oct 2011 issue of MSDN Magazine (http://msdn.microsoft.com/en-us/magazine/hh456403.aspx). ContinueWith() specifies the continuation actions that should be performed after the original task completes its work.

The Task class also does have basic support for handling exceptions that are thrown from asynchronous operations. In C# 4, you'll need to examine some of the properties of the Task object that manages your asynchronous operations in order to handle exceptions. I've made more modification to the ServeBreakfast method in order to handle any exceptions that might occur while obtaining the order:

```
Task ServeBreakfast(Diner diner)
{
   var order = ObtainOrderAsync(diner);
   order.Wait();
   if (order.IsFaulted)
      throw order.Exception;
   var ingredients = ObtainIngredientsAsync(order.Result);
   var finished = ObtainRecipeAsync(order.Result)
      .ContinueWith(r => r.Result.PrepareAsync(ingredients.Result)
         .ContinueWith(m => diner.Give(m.Result)
      )
```

```
    );
    return finished;
}
```

This code waits for the order task (obtaining the order) to finish. Then, it checks to see if it is faulted. If it is faulted, this routine rethrows the exception that was generated by the async task. Otherwise, it continues on its way, doing the remaining work. You would do the same thing with the other tasks in order to check each asynchronous operation for errors before continuing. Here's the final version, after propagating any exceptions from any of the asynchronous tasks:

```
Task ServeBreakfast(Diner diner)
{
    var order = ObtainOrderAsync(diner);
    order.Wait();
    if (order.IsFaulted)
        throw order.Exception;
    var ingredients = ObtainIngredientsAsync(order.Result);
    var recipe = ObtainRecipeAsync(order.Result);
    Task.WaitAll(ingredients, recipe);
    if (ingredients.IsFaulted)
        throw ingredients.Exception;
    if (recipe.IsFaulted)
        throw recipe.Exception;
    var finished = recipe.ContinueWith(r =>
        r.Result.PrepareAsync(ingredients.Result)
            .ContinueWith(m => diner.Give(m.Result)
        )
    );
    return finished;
}
```

This final version comes closer to approximating the functionality of the original version. It's still missing

several features that Eric and Mads cover in their MSDN magazine articles. Those features include cancellation, and marshaling continuations to the UI thread. Those features can be replicated using methods on the Task class. However, they are quite a bit more work.

By leveraging the Task class, you can create asynchronous code today that will be easy to upgrade to the C# 5 language features when you're ready. You'll learn more about writing asynchronous algorithms, and how you can take advantage of asynchrony in your applications. Windows Phone and Silverlight APIs already expect you leverage async programming to maintain a responsive UI. You'll also be better prepared for WinRT programming, where more of the APIs are available only in an asynchronous method.

The next version of C# will contain great new features to make it easier to create asynchronous code that reads and feels like our familiar synchronous models. But that's no reason to wait. Download and try the VS 11 CTP. Equally important, start using Task<T> and Task in your current work with C# 4 and Visual Studio 2010. You'll learn valuable experience with asynchronous programming, and you will be able to more easily adapt your current codebase to C# 5 when you're ready.