Tutorial No. 2

Understanding a Simple Async Program

HTTP://BLOGS.MSDN.COM/B/CSHARPFAQ/ARCHIVE/2012/06/26/UNDERSTANDING-A-SIMPLE-ASYNC-PROGRAM.ASPX

by Alan Berman

The Async feature in Visual Studio 2012 RC makes it easy to invoke asynchronous methods.

If you mark a method with or async modifier, you can use the await operator in the method. When control reaches an await expression in the async method, control returns to the caller, and progress in the method is suspended until the awaited task completes. When the task is complete, execution can resume in the method.

Using asynchronous methods instead of synchronous methods can provide benefits. Asynchrony makes UI applications more responsive because the UI thread that launches that operation can perform other work. Asynchrony also improves the scalability of server-based application by reducing the need for threads.

This blog provides a simple async example and describes what occurs when it runs. It also provides an example of exception handling for a call to an async method.

Running the Examples

To run the examples in this blog, you can do the following:

1. Install Visual Studio 2012 RC, which can be accessed from Visual Studio Asynchronous Programming.
2. Create **WPF Application** or **Windows Forms Application**.
3. In your project, add a reference to **System.Net.Http**. This allows you to use the HttpClient class in the first example,
4. Add a **Button** to the application. Modify the Button_Click event handler to add the async modifier. The Button_Click event handlers in the examples are from a **WPF Application**, however they can be modified for a **Windows Form Application**.
5. Include the following **using** statements.

```
using System.Diagnostics;
using System.Net.Http;
using System.Threading.Tasks;
```

Example of async and await

The following example illustrates use of the async modifier and the await operator. An explanation is provided below. Note that startButton_Click is marked with the async modifier

```
private async void startButton_Click(object sender, RoutedEventArgs e)
{
    Task<string> getWebPageTask = GetWebPageAsync("http://msdn.microsoft.com");

    Debug.WriteLine("In startButton_Click before await");
    string webText = await getWebPageTask;
    Debug.WriteLine("Characters received: " + webText.Length.ToString());
}

private async Task<string> GetWebPageAsync(string url)
{
    // Start an async task.
    Task<string> getStringTask = (new HttpClient()).GetStringAsync(url);

    // Await the task. This is what happens:
    // 1. Execution immediately returns to the calling method, returning a
    //    different task from the task created in the previous statement.
    //    Execution in this method is suspended.
    // 2. When the task created in the previous statement completes, the
    //    result from the GetStringAsync method is produced by the Await
    //    statement, and execution continues within this method.
    Debug.WriteLine("In GetWebPageAsync before await");
    string webText = await getStringTask;
    Debug.WriteLine("In GetWebPageAsync after await");
```

```
    return webText;
}

// Output:
//   In GetWebPageAsync before await
//   In startButton_Click before await
//   In GetWebPageAsync after await
//   Characters received: 44306
```

Control Flow in Example

In the above example, the startButton_Click method calls the GetWebPageAsync method. In the GetWebPageAsync method, the following occurs:

1. The GetWebPageAsync method calls the GetStringAsync method, which returns a task.
2. The "await getStringTask" expression causes the GetWebPageAsync method to immediately exit and return a different task. Execution in the GetWebPageAsync method is suspended.
3. When the awaited task (getStringTask) completes at a later time, processing resumes after the await statement in the GetWebPageAsync method.

The startButton_Click method also contains an await expression, which is "await getWebPageTask". Because GetWebPageAsync is an async method, the task for the call to GetWebPageAsync needs to be awaited. startButton_Click needs to be defined with the async modifier because it has an await expression.

Return Types in Example

In the above example, the GetWebPageAsync method has a **return** statement that returns a **string**. Therefore, the method declaration of GetWebPageAsync has a return type of **Task<string>**. Because the return type is **Task<string>**, the evaluation of the await expression in startButton_Click produces a **string**, as the following statement demonstrates: string webText = await getWebPageTask; .

Similarly, the GetWebPageAsync method calls the GetStringAsync method, which is defined with a return type of**Task<string>**. The call to GetStringAsync returns a **Task<string>** and the evaluation of the await expression produces a**string**.

The startButton_Click method has a return type of void.

Note: An async method can have a return type of Task<TResult>, Task, or **void.** The **void** return type is used primarily to define event handlers, where a **void** return type is required. An async method that returns void can't be awaited, and the caller of a void-returning method can't catch exceptions that are thrown by the method.

Simplified Code

In the above example, the GetWebPageAsync method includes the following two statements:

```
Task<string> getStringTask = (new HttpClient()).GetStringAsync(url);
string webText = await getStringTask;
```

The above two statements can be condensed into one statement, as follows:

```
string webText = await (new HttpClient()).GetStringAsync(url);
```

The condensed statement also returns a task and awaits the task, however the task is not stored in a variable. While the condensed statement is more concise, the uncondensed code facilitates a greater understanding of the control flow of your code, and of the associated tasks.

Exception Handling

The following example illustrates exception handling for an async method. To catch an exception that applies to an async task, the **await** expression is in a **try** block, and the exception is caught in a **catch** block.

Uncomment the "throw new Exception" line in the example to demonstrate exception handling. The exception is caught in the **catch** block, and the task's **IsFaulted** property is set to **true**, and the task's **Exception.InnerException** property is set to the exception.

Uncomment the "throw new OperationCancelledException" line to demonstrate what happens when you cancel an asynchronous process. The exception is caught in the **catch** block and the

task's **IsCanceled** property is set to **true**. Under some conditions that don't apply to this example, **IsFaulted** is set to true and **IsCanceled** is set to false.

```
private async void testExceptionButton_Click(object sender, RoutedEventArgs e)
{
    Task<string> theTask = DelayAsync();

    try
    {
        string result = await theTask;
        Debug.WriteLine("Result: " + result);
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Exception Message: " + ex.Message);
    }
    Debug.WriteLine("Task IsCanceled: " + theTask.IsCanceled);
    Debug.WriteLine("Task IsFaulted:  " + theTask.IsFaulted);
    if (theTask.Exception != null)
    {
        Debug.WriteLine("Task Exception Message: "
            + theTask.Exception.Message);
        Debug.WriteLine("Task Inner Exception Message: "
            + theTask.Exception.InnerException.Message);
    }
}

private async Task<string> DelayAsync()
{
    await Task.Delay(100);

    // Uncomment each of the following lines to
    // demonstrate exception handling.

    //throw new OperationCanceledException("canceled");
    //throw new Exception("Something happened.");
```

```
    return "Done";
}

// Output when no exception is thrown in the awaited method:
//   Result: Done
//   Task IsCanceled: False
//   Task IsFaulted:  False

// Output when an Exception is thrown in the awaited method:
//   Exception Message: Something happened.
//   Task IsCanceled: False
//   Task IsFaulted:  True
//   Task Exception Message: One or more errors occurred.
//   Task Inner Exception Message: Something happened.

// Output when a OperationCanceledException or TaskCanceledException
// is thrown in the awaited method:
//   Exception Message: canceled
//   Task IsCanceled: True
//   Task IsFaulted:  False
```

See Also

- Async Feature Control Flow blog
- Simultaneous Async Tasks blog
- Using Async for File Access blog

Resources

- Details and Download Page
- Forum for Feedback and Questions
- Microsoft Connect for Bugs and Suggestions