Asynchronous Programming with async and await

async Programming

- async programming, that leverages asynchronous support in the .NET Framework 4.5 and the Windows Runtime.
- Asynchrony is essential for activities that are potentially blocking, such as when your application accesses the web.
- Access to a web resource sometimes is slow or delayed. If such an activity is blocked within a synchronous process, the entire application must wait

async Programming

- In an asynchronous process, the application can continue with other work that doesn't depend on the web resource until the potentially blocking task finishes.
- The following table shows typical areas where asynchronous programming improves responsiveness.

async Programming

Application area	Supporting APIs that contain async methods
Web access	HttpClient, SyndicationClient
Working with files	StorageFile, StreamWriter, StreamReader, XmlReader
Working with images	MediaCapture, BitmapEncoder, BitmapDecoder
WCF programming	Synchronous and Asynchronous Operations

async Programming

- Asynchrony proves especially valuable for applications that access the UI thread because all UI-related activity usually shares one thread.
- When you use asynchronous methods, the application continues to respond to the UI.
- When you use asynchronous methods, the application continues to respond to the UI.
- You can resize or minimize a window, for example, or you can close the application if you don't want to wait for it to finish.

async Methods

- The async and await keywords in C# are the heart of async programming.
- Asynchronous methods that you define by using async and await are referred to as async methods.
- The following example shows an async method.

async Programming Example

```
async Task<int> AccessTheWebAsync()
{// Three things to note in the signature:
//  - The method has an async modifier.
//  - The return type is Task or Task<T>.
//    Here, it is Task<int> because the return statement returns an integer.
//  - The method name ends in "Async.“
// You need to add a reference to System.Net.Http to declare client.
   HttpClient client = new HttpClient();
// GetStringAsync returns a Task<string>. That means that when you await the
   // task you'll get a string (urlContents).
   Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");
```

async Programming Example

```
// You can do work here that doesn't rely on the string from GetStringAsync.
   DoIndependentWork();
// The await operator suspends AccessTheWebAsync.
   //  - AccessTheWebAsync can't continue until getStringTask is complete.
   //  - Meanwhile, control returns to the caller of AccessTheWebAsync.
   //  - Control resumes here when getStringTask is complete.
   //  - The await operator then retrieves the string result from getStringTask.
   string urlContents = await getStringTask;
// The return statement specifies an integer result.
   // Any methods that are awaiting AccessTheWebAsync retrieve the length value.
   return urlContents.Length;
}
```

async Programming Example

- If AccessTheWebAsync doesn't have any work that it can do between calling GetStringAsync and awaiting its completion, you can simplify your code by calling and awaiting in the following single statement.

```
string urlContents = await client.GetStringAsync();
```

async Method Summary

- The method signature includes an **Async** or **async** modifier.
- The name of an async method, by convention, ends with an "Async" suffix.
- The return type is one of the following types:
  - Task<TResult> if your method has a return statement in which the operand has type TResult.
  - Task if your method has no return statement or has a return statement with no operand.
  - Void (a Sub in Visual Basic) if you're writing an async event handler.

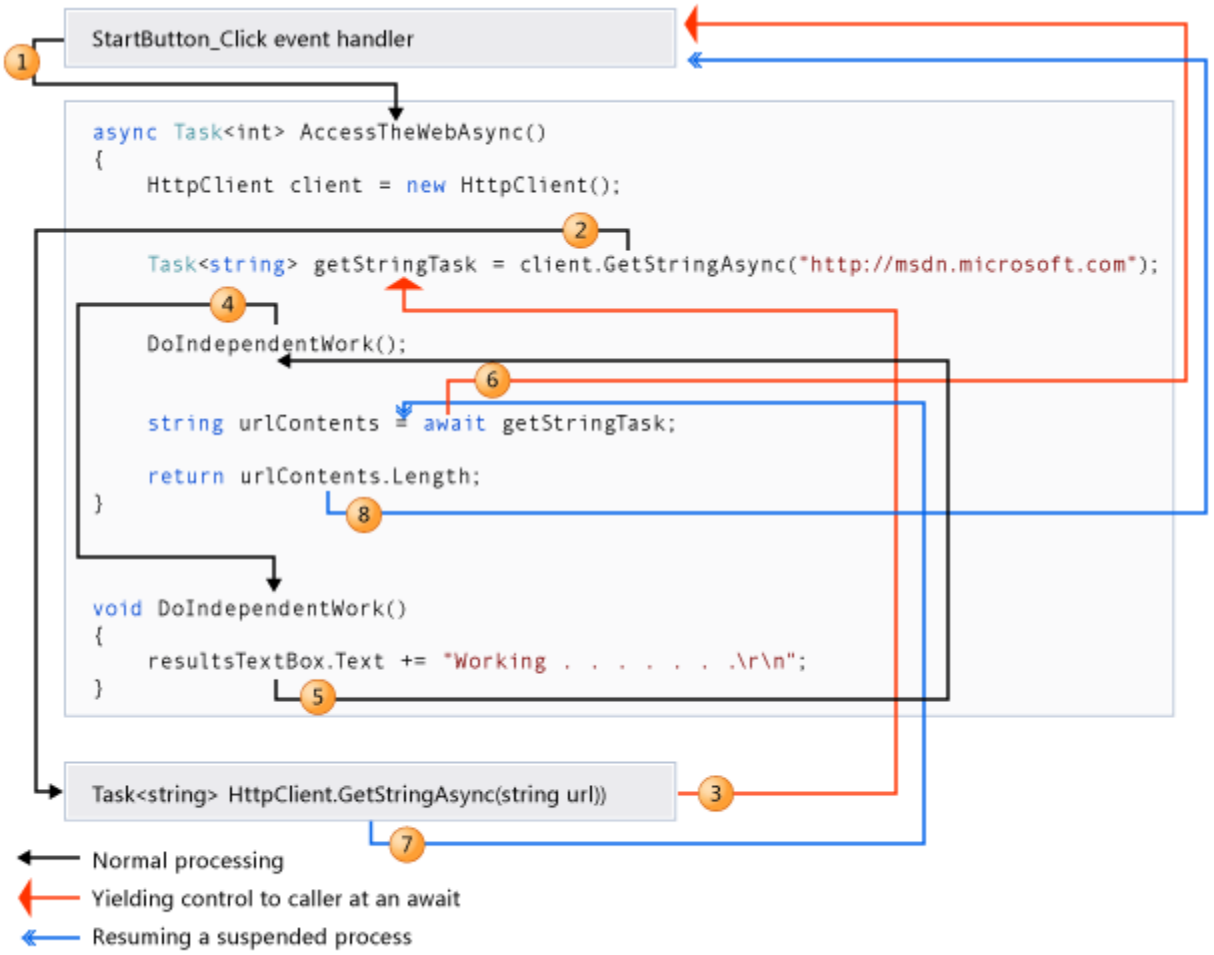
async Method Summary

- The method usually includes at least one await expression, which marks a point where the method can't continue until the awaited asynchronous operation is complete.
- In the meantime, the method is suspended, and control returns to the method's caller.
- The next section of this topic illustrates what happens at the suspension point.

What Happens in an Async Method

The most important thing to understand in asynchronous programming is how the control flow moves from method to method. The following diagram leads you through the process.

What Happens in an Async Method



What Happens in an Async Method

The numbers in the diagram correspond to the following steps.

1. An event handler calls and awaits the AccessTheWebAsync async method.
2. AccessTheWebAsync creates an HttpClient instance and calls the GetStringAsync asynchronous method to download the contents of a website as a string.
3. Something happens in **GetStringAsync** that suspends its progress. Perhaps it must wait for a website to download or some other blocking activity. To avoid blocking resources, **GetStringAsync** yields control to its caller, AccessTheWebAsync.

What Happens in an Async Method

GetStringAsync returns a Task<TResult> where TResult is a string, and AccessTheWebAsync assigns the task to the getStringTask variable.

The task represents the ongoing process for the call to **GetStringAsync**, with a commitment to produce an actual string value when the work is complete.

4. Because getStringTask hasn't been awaited yet, AccessTheWebAsync can continue with other work that doesn't depend on the final result from **GetStringAsync**.

What Happens in an Async Method

That work is represented by a call to the synchronous method DoIndependentWork.

5. DoIndependentWork is a synchronous method that does its work and returns to its caller.
6. AccessTheWebAsync has run out of work that it can do without a result from getStringTask. AccessTheWebAsync next wants to calculate and return the length of the downloaded string, but the method can't calculate that value until the method has the string.

What Happens in an Async Method

Therefore, AccessTheWebAsync uses an await operator to suspend its progress and to yield control to the method that called AccessTheWebAsync. AccessTheWebAsync returns a **Task(Of Integer)** or **Task<int>** to the caller.

7. **GetStringAsync** completes and produces a string result.
   The method already returned a task in step 3.
   The string result is stored in the task that represents the completion of the method, getStringTask.
   The await operator retrieves the result from getStringTask.
   The assignment statement assigns the retrieved result to urlContents.

What Happens in an Async Method

8. When AccessTheWebAsync has the string result, the method can calculate the length of the string. Then the work of AccessTheWebAsync is also complete, and the waiting event handler can resume.

A synchronous method returns when its work is complete (step 5), but an async method returns a task value when its work is suspended (steps 3 and 6). When the async method eventually completes its work, the task is marked as completed and the result, if any, is stored in the task.

API Async Methods

- The .NET Framework 4.5 contains many members that work with async and await.
- You can recognize these members by the "Async" suffix that's attached to the member name and a return type of Task or Task<TResult>.
- For example, the System.IO.Stream class contains methods such as CopyToAsync, ReadAsync, and WriteAsync alongside the synchronous methods CopyTo, Read, and Write.

Threads

- Async methods are intended to be non-blocking operations.
- An await expression in an async method doesn’t block the current thread while the awaited task is running.
- Instead, the expression signs up the rest of the method as a continuation and returns control to the caller of the async method.
- The async and await keywords don't cause additional threads to be created.

Threads

- Async methods don't require multithreading because an async method doesn't run on its own thread.
- The method runs on the current synchronization context and uses time on the thread only when the method is active.

Async and Await

If you specify that a method is an async method by using an async modifier, you enable the following two capabilities.

- The marked async method can use Await or await to designate suspension points.
- The await operator tells the compiler that the async method can't continue past that point until the awaited asynchronous process is complete.
- In the meantime, control returns to the caller of the async method.
- The suspension of an async method at an await expression doesn't constitute an exit from the method.

Async and Await

- The marked async method can itself be awaited by methods that call it.

An async method typically contains one or more occurrences of an await operator, but the absence of await expressions doesn't cause a compiler error.

If an async method doesn't use an await operator to mark a suspension point, the method executes as a synchronous method does, despite the async modifier.
The compiler issues a warning for such methods.

Return Types and Parameters

- In .NET Framework programming, an async method typically returns a Task or a Task<TResult>.
- Inside an async method, an await operator is applied to a task that's returned from a call to another async method.
- You specify Task<TResult> as the return type if the method contains a return statement that specifies an operand of type **TResult**.
- You use **Task** as the return type if the method has no return statement or has a return statement that doesn't return an operand

Task<TResult> or a Task.

The following example shows how you declare and call a method that returns a Task<TResult> or a Task.

```
// Signature specifies Task<TResult>
async Task<int> TaskOfTResult_MethodAsync()
{
   int hours;
   // . . .
   // Return statement specifies an integer result.
   return hours;
}
// Calls to TaskOfTResult_MethodAsync
Task<int> returnedTaskTResult = TaskOfTResult_MethodAsync();
int intResult = await returnedTaskTResult;
// or, in a single statement
int intResult = await TaskOfTResult_MethodAsync();
```

Task<TResult> or a Task.

```
// Signature specifies Task
async Task Task_MethodAsync()
{
    // . . .
    // The method has no return statement.
}

// Calls to Task_MethodAsync
Task returnedTask = Task_MethodAsync();
await returnedTask;
// or, in a single statement
await Task_MethodAsync();
```

Task<TResult> or a Task.

- An async method can have a **void** return type.
- This return type is used primarily to define event handlers, where a **void** return type is required.
- An async method that has a **void** return type can’t be awaited, and the caller of a void-returning method can't catch any exceptions that the method throws.
- An async method can't declare ref or out parameters in C#, but the method can call methods that have such parameters.

Naming Convention

- By convention, you append "Async" to the names of methods that have an **async** modifier.
- You can ignore the convention where an event, base class, or interface contract suggests a different name.

Complete Example

The following code is the MainWindow.xaml.vb or MainWindow.xaml.cs file from the Windows Presentation Foundation (WPF) application that this topic discusses.

```
// Add a using directive and a reference for System.Net.Http;
using System.Net.Http;
namespace AsyncFirstExample
{
    public partial class MainWindow : Window
    {
        // Mark the event handler with async so you can use await in it.
```

Complete Example

```
private async void StartButton_Click(object sender, RoutedEventArgs
e)
      {
         // Call and await separately.
         //Task<int> getLengthTask = AccessTheWebAsync();
         //// You can do independent work here.
         //int contentLength = await getLengthTask;

         int contentLength = await AccessTheWebAsync();

         resultsTextBox.Text +=
            String.Format("\r\nLength of the downloaded string:
{0}.\r\n", contentLength);
      }
```

Complete Example

```
// Three things to note in the signature:
      //  - The method has an async modifier.
      //  - The return type is Task or Task<T>. (See "Return Types" section.)
      //    Here, it is Task<int> because the return statement returns an integer.
      //  - The method name ends in "Async."
      async Task<int> AccessTheWebAsync()
      {
         // You need to add a reference to System.Net.Http to declare client.
         HttpClient client = new HttpClient();
```

Complete Example

```
// GetStringAsync returns a Task<string>. That means that when you await the
        // task you'll get a string (urlContents).
        Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");

        // You can do work here that doesn't rely on the string from GetStringAsync.
        DoIndependentWork();

// The await operator suspends AccessTheWebAsync.
//  - AccessTheWebAsync can't continue until getStringTask is complete.
//  - Meanwhile, control returns to the caller of AccessTheWebAsync.
//  - Control resumes here when getStringTask is complete.
//  - The await operator then retrieves the string result from getStringTask.
```

Complete Example

```
string urlContents = await getStringTask;
  // The return statement specifies an integer result.
 // Any methods that are awaiting AccessTheWebAsync retrieve the
//length value.
          return urlContents.Length;
      }
void DoIndependentWork()
      {
          resultsTextBox.Text += "Working . . . . . .\r\n";
      }
   }
}
```

Complete Example

- // Sample Output:
-
- // Working . . . . . .
-
- // Length of the downloaded string: 41564.