**Asynchronous Programming with Async and Await (C# and Visual Basic)**

**Visual Studio 2013**

[Other Versions](javascript:;)

Description: http://i.msdn.microsoft.com/Areas/Epx/Content/Images/ImageSprite.png

* [Visual Studio 2012](http://msdn.microsoft.com/en-us/library/hh191443(d=printer,v=vs.110).aspx)

You can avoid performance bottlenecks and enhance the overall responsiveness of your application by using asynchronous programming. However, traditional techniques for writing asynchronous applications can be complicated, making them difficult to write, debug, and maintain.

Visual Studio 2012 introduces a simplified approach, async programming, that leverages asynchronous support in the .NET Framework 4.5 and the Windows Runtime. The compiler does the difficult work that the developer used to do, and your application retains a logical structure that resembles synchronous code. As a result, you get all the advantages of asynchronous programming with a fraction of the effort.

This topic contains the following sections.

* [Async Improves Responsiveness](http://msdn.microsoft.com/en-us/library/hh191443(d=printer).aspx#BKMK_WhentoUseAsynchrony)
* [Async Methods Are Easier to Write](http://msdn.microsoft.com/en-us/library/hh191443(d=printer).aspx#BKMK_HowtoWriteanAsyncMethod)
* [What Happens in an Async Method](http://msdn.microsoft.com/en-us/library/hh191443(d=printer).aspx#BKMK_WhatHappensUnderstandinganAsyncMethod)
* [API Async Methods](http://msdn.microsoft.com/en-us/library/hh191443(d=printer).aspx#BKMK_APIAsyncMethods)
* [Threads](http://msdn.microsoft.com/en-us/library/hh191443(d=printer).aspx#BKMK_Threads)
* [Async and Await](http://msdn.microsoft.com/en-us/library/hh191443(d=printer).aspx#BKMK_AsyncandAwait)
* [Return Types and Parameters](http://msdn.microsoft.com/en-us/library/hh191443(d=printer).aspx#BKMK_ReturnTypesandParameters)
* [Naming Convention](http://msdn.microsoft.com/en-us/library/hh191443(d=printer).aspx#BKMK_NamingConvention)
* [Related Topics](http://msdn.microsoft.com/en-us/library/hh191443(d=printer).aspx#BKMK_RelatedTopics)
* [Complete Example](http://msdn.microsoft.com/en-us/library/hh191443(d=printer).aspx#BKMK_CompleteExample)
* [Related Topics](http://msdn.microsoft.com/en-us/library/hh191443(d=printer).aspx#seeAlsoToggle)

This topic provides an overview of when and how to use async programming and includes links to support topics that contain details and examples.

[Async Improves Responsiveness](javascript:void(0))

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. 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. The listed APIs from the .NET Framework 4.5 and the Windows Runtime contain methods that support async programming.

|  |  |
| --- | --- |
| **Application area** | **Supporting APIs that contain async methods** |
| Web access | [HttpClient](http://msdn.microsoft.com/en-us/library/system.net.http.httpclient.aspx) , [SyndicationClient](http://go.microsoft.com/fwlink/p/?LinkId=259441) |
| Working with files | [StorageFile](http://go.microsoft.com/fwlink/p/?LinkId=248220), [StreamWriter](http://msdn.microsoft.com/en-us/library/system.io.streamwriter.aspx), [StreamReader](http://msdn.microsoft.com/en-us/library/system.io.streamreader.aspx), [XmlReader](http://msdn.microsoft.com/en-us/library/system.xml.xmlreader.aspx) |
| Working with images | [MediaCapture](http://go.microsoft.com/fwlink/p/?LinkId=261839), [BitmapEncoder](http://go.microsoft.com/fwlink/p/?LinkId=261840), [BitmapDecoder](http://go.microsoft.com/fwlink/p/?LinkId=261841) |
| WCF programming | [Synchronous and Asynchronous Operations](http://go.microsoft.com/fwlink/p/?LinkID=192382) |
|  |  |

Asynchrony proves especially valuable for applications that access the UI thread because all UI-related activity usually shares one thread. If any process is blocked in a synchronous application, all are blocked. Your application stops responding, and you might conclude that it has failed when instead it's just waiting.

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.

The async-based approach adds the equivalent of an automatic transmission to the list of options that you can choose from when designing asynchronous operations. That is, you get all the benefits of traditional asynchronous programming but with much less effort from the developer.

[Async Methods Are Easier to Write](javascript:void(0))

The [Async](http://msdn.microsoft.com/en-us/library/hh191564.aspx) and [Await](http://msdn.microsoft.com/en-us/library/hh191564.aspx) keywords in Visual Basic and the [async](http://msdn.microsoft.com/en-us/library/hh156513.aspx) and [await](http://msdn.microsoft.com/en-us/library/hh156528.aspx) keywords in C# are the heart of async programming. By using those two keywords, you can use resources in the .NET Framework or the Windows Runtime to create an asynchronous method almost as easily as you create a synchronous method. Asynchronous methods that you define by using async and await are referred to as async methods.

The following example shows an async method. Almost everything in the code should look completely familiar to you. The comments call out the features that you add to create the asynchrony.

You can find the complete example file at the end of this topic, and you can download the sample from [Async Sample: Example from "Asynchronous Programming with Async and Await"](http://go.microsoft.com/fwlink/?LinkID=261549) .

C#

[VB](http://msdn.microsoft.com/en-us/library/hh191443(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-1)

// 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();

// 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.

string urlContents = await getStringTask;

// The return statement specifies an integer result.

// Any methods that are awaiting AccessTheWebAsync retrieve the length value.

return urlContents.Length;

}

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.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh191443(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-2)

string urlContents = await client.GetStringAsync();

The following characteristics summarize what makes the previous example an async method.

* 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>](http://msdn.microsoft.com/en-us/library/dd321424.aspx) if your method has a return statement in which the operand has type TResult.
  + [Task](http://msdn.microsoft.com/en-us/library/system.threading.tasks.task.aspx) if your method has no return statement or has a return statement with no operand.
  + Void (a [Sub](http://msdn.microsoft.com/en-us/library/831f9wka.aspx) in Visual Basic) if you're writing an async event handler.

For more information, see "Return Types and Parameters" later in this topic.

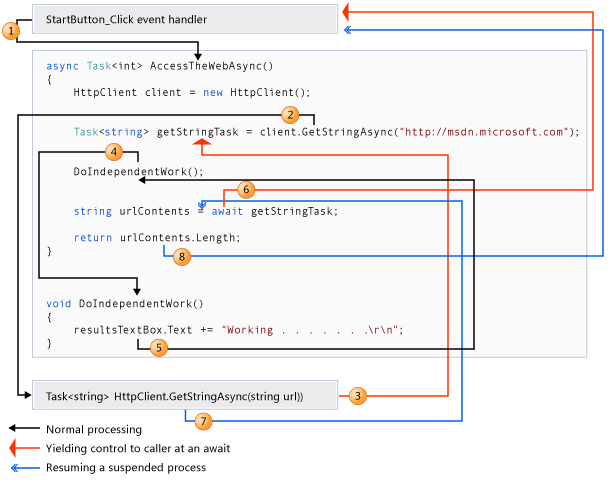
* The method usually includes at least one await expression, which marks a point where the method can't continue until the awaited asynchronous operation is complete. In the meantime, the method is suspended, and control returns to the method's caller. The next section of this topic illustrates what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

In async methods, you use the provided keywords and types to indicate what you want to do, and the compiler does the rest, including keeping track of what must happen when control returns to an await point in a suspended method. Some routine processes, such as loops and exception handling, can be difficult to handle in traditional asynchronous code. In an async method, you write these elements much as you would in a synchronous solution, and the problem is solved.

For more information about asynchrony in previous versions of the .NET Framework, see [TPL and Traditional .NET Framework Asynchronous Programming](http://msdn.microsoft.com/en-us/library/dd997423.aspx).

[What Happens in an Async Method](javascript:void(0))

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.



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](http://msdn.microsoft.com/en-us/library/system.net.http.httpclient.aspx) instance and calls the [GetStringAsync](http://msdn.microsoft.com/en-us/library/system.net.http.httpclient.getstringasync.aspx) 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.

GetStringAsync returns a [Task<TResult>](http://msdn.microsoft.com/en-us/library/dd321424.aspx) 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.

1. Because getStringTask hasn't been awaited yet, AccessTheWebAsync can continue with other work that doesn't depend on the final result from GetStringAsync. That work is represented by a call to the synchronous method DoIndependentWork.
2. DoIndependentWork is a synchronous method that does its work and returns to its caller.
3. 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.

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. The task represents a promise to produce an integer result that's the length of the downloaded string.

|  |
| --- |
| **Description: NoteNote** |
| If GetStringAsync (and therefore getStringTask) is complete before AccessTheWebAsync awaits it, control remains in AccessTheWebAsync. The expense of suspending and then returning to AccessTheWebAsync would be wasted if the called asynchronous process (getStringTask) has already completed and AccessTheWebSync doesn't have to wait for the final result. |

Inside the caller (the event handler in this example), the processing pattern continues. The caller might do other work that doesn't depend on the result from AccessTheWebAsync before awaiting that result, or the caller might await immediately. The event handler is waiting for AccessTheWebAsync, and AccessTheWebAsync is waiting for GetStringAsync.

1. GetStringAsync completes and produces a string result. The string result isn't returned by the call to GetStringAsync in the way that you might expect. (Remember that the method already returned a task in step 3.) Instead, 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.
2. 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. In the full example at the end of the topic, you can confirm that the event handler retrieves and prints the value of the length result.

If you are new to asynchronous programming, take a minute to consider the difference between synchronous and asynchronous behavior. 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.

For more information about control flow, see [Control Flow in Async Programs (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/hh873191.aspx).

[API Async Methods](javascript:void(0))

You might be wondering where to find methods such as GetStringAsync that support async programming. 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](http://msdn.microsoft.com/en-us/library/system.threading.tasks.task.aspx) or [Task<TResult>](http://msdn.microsoft.com/en-us/library/dd321424.aspx). For example, the System.IO.Stream class contains methods such as [CopyToAsync](http://msdn.microsoft.com/en-us/library/system.io.stream.copytoasync.aspx), [ReadAsync](http://msdn.microsoft.com/en-us/library/system.io.stream.readasync.aspx), and [WriteAsync](http://msdn.microsoft.com/en-us/library/system.io.stream.writeasync.aspx) alongside the synchronous methods [CopyTo](http://msdn.microsoft.com/en-us/library/system.io.stream.copyto.aspx), [Read](http://msdn.microsoft.com/en-us/library/system.io.stream.read.aspx), and [Write](http://msdn.microsoft.com/en-us/library/system.io.stream.write.aspx).

The Windows Runtime also contains many methods that you can use with async and await in Windows Store apps. For more information and example methods, see [Quickstart: using the await operator for asynchronous programming](http://go.microsoft.com/fwlink/?LinkId=248545), [Asynchronous programming (Windows Store apps)](http://go.microsoft.com/fwlink/?LinkId=259592), and [WhenAny: Bridging between the .NET Framework and the Windows Runtime (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/jj635140.aspx).

[Threads](javascript:void(0))

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. 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. You can use [Task.Run](http://msdn.microsoft.com/en-us/library/system.threading.tasks.task.run.aspx) to move CPU-bound work to a background thread, but a background thread doesn't help with a process that's just waiting for results to become available.

The async-based approach to asynchronous programming is preferable to existing approaches in almost every case. In particular, this approach is better than [BackgroundWorker](http://msdn.microsoft.com/en-us/library/system.componentmodel.backgroundworker.aspx) for IO-bound operations because the code is simpler and you don't have to guard against race conditions. In combination with [Task.Run](http://msdn.microsoft.com/en-us/library/system.threading.tasks.task.run.aspx), async programming is better than [BackgroundWorker](http://msdn.microsoft.com/en-us/library/system.componentmodel.backgroundworker.aspx) for CPU-bound operations because async programming separates the coordination details of running your code from the work that Task.Run transfers to the threadpool.

[Async and Await](javascript:void(0))

If you specify that a method is an async method by using an [Async](http://msdn.microsoft.com/en-us/library/hh191564.aspx) or [async](http://msdn.microsoft.com/en-us/library/hh156513.aspx) modifier, you enable the following two capabilities.

* The marked async method can use [Await](http://msdn.microsoft.com/en-us/library/hh156570.aspx) or [await](http://msdn.microsoft.com/en-us/library/hh156528.aspx) 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, and finally blocks don’t run.

* 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.

Async , async, Await, and await are contextual keywords. For more information and examples, see the following topics:

* [Async (Visual Basic)](http://msdn.microsoft.com/en-us/library/hh191564.aspx)
* [async (C# Reference)](http://msdn.microsoft.com/en-us/library/hh156513.aspx)
* [Await Operator (Visual Basic)](http://msdn.microsoft.com/en-us/library/hh156570.aspx)
* [await (C# Reference)](http://msdn.microsoft.com/en-us/library/hh156528.aspx)

[Return Types and Parameters](javascript:void(0))

In .NET Framework programming, an async method typically returns a [Task](http://msdn.microsoft.com/en-us/library/system.threading.tasks.task.aspx) or a [Task<TResult>](http://msdn.microsoft.com/en-us/library/dd321424.aspx). 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>](http://msdn.microsoft.com/en-us/library/dd321424.aspx) as the return type if the method contains a [Return](http://msdn.microsoft.com/en-us/library/2e34641s.aspx) (Visual Basic) or [return](http://msdn.microsoft.com/en-us/library/1h3swy84.aspx) (C#) 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.

The following example shows how you declare and call a method that returns a [Task<TResult>](http://msdn.microsoft.com/en-us/library/dd321424.aspx) or a [Task](http://msdn.microsoft.com/en-us/library/system.threading.tasks.task.aspx).

C#

[VB](http://msdn.microsoft.com/en-us/library/hh191443(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-3)

// 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();

// 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();

Each returned task represents ongoing work. A task encapsulates information about the state of the asynchronous process and, eventually, either the final result from the process or the exception that the process raises if it doesn't succeed.

An async method can also be a Sub method (Visual Basic) or have a void return type (C#). This return type is used primarily to define event handlers, where a void return type is required. Async event handlers often serve as the starting point for async programs.

An async method that’s a Sub procedure or 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 [ByRef](http://msdn.microsoft.com/en-us/library/c84t73c2.aspx) parameters in Visual Basic or [ref](http://msdn.microsoft.com/en-us/library/14akc2c7.aspx) or [out](http://msdn.microsoft.com/en-us/library/t3c3bfhx.aspx) parameters in C#, but the method can call methods that have such parameters.

For more information and examples, see [Async Return Types (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/hh524395.aspx). For more information about how to catch exceptions in async methods, see [try-catch (C# Reference)](http://msdn.microsoft.com/en-us/library/0yd65esw.aspx) or [Try...Catch...Finally Statement (Visual Basic)](http://msdn.microsoft.com/en-us/library/fk6t46tz.aspx).

Asynchronous APIs in Windows Runtime programming have one of the following return types, which are similar to tasks:

* [IAsyncOperation](http://go.microsoft.com/fwlink/p/?LinkId=261896), which corresponds to [Task<TResult>](http://msdn.microsoft.com/en-us/library/dd321424.aspx)
* [IAsyncAction](http://go.microsoft.com/fwlink/p/?LinkId=261897), which corresponds to [Task](http://msdn.microsoft.com/en-us/library/system.threading.tasks.task.aspx)
* [IAsyncActionWithProgress](http://go.microsoft.com/fwlink/p/?LinkId=261898)
* [IAsyncOperationWithProgress](http://go.microsoft.com/fwlink/p/?LinkID=259454)

For more information and an example, see [Quickstart: using the await operator for asynchronous programming](http://go.microsoft.com/fwlink/p/?LinkId=248545).

[Naming Convention](javascript:void(0))

By convention, you append "Async" to the names of methods that have an Async or async modifier.

You can ignore the convention where an event, base class, or interface contract suggests a different name. For example, you shouldn’t rename common event handlers, such as Button1\_Click.

[Related Topics](javascript:void(0))

|  |  |  |
| --- | --- | --- |
| **Title** | **Description** | **Sample** |
| [Walkthrough: Accessing the Web by Using Async and Await (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/hh300224.aspx) | Shows how to convert a synchronous WPF solution to an asynchronous WPF solution. The application downloads a series of websites. | [Async Sample: Accessing the Web Walkthrough (C# and Visual Basic)](http://go.microsoft.com/fwlink/p/?LinkID=255191&clcid=0x409) |
| [How to: Extend the Async Walkthrough by Using Task.WhenAll (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/hh556530.aspx) | Adds [Task.WhenAll](http://msdn.microsoft.com/en-us/library/system.threading.tasks.task.whenall.aspx) to the previous walkthrough. The use of WhenAll starts all the downloads at the same time. |  |
| [How to: Make Multiple Web Requests in Parallel by Using Async and Await (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/hh696703.aspx) | Demonstrates how to start several tasks at the same time. | [Async Sample: Make Multiple Web Requests in Parallel (C# and Visual Basic)](http://go.microsoft.com/fwlink/p/?LinkID=254906&clcid=0x409) |
| [Async Return Types (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/hh524395.aspx) | Illustrates the types that async methods can return and explains when each type is appropriate. |  |
| [Control Flow in Async Programs (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/hh873191.aspx) | Traces in detail the flow of control through a succession of await expressions in an asynchronous program. | [Async Sample: Control Flow in Async Programs (C# and Visual Basic)](http://go.microsoft.com/fwlink/p/?LinkID=255285&clcid=0x409) |
| [Fine-Tuning Your Async Application (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/jj155761.aspx) | Shows how to add the following functionality to your async solution:   * [Cancel an Async Task or a List of Tasks (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/jj155759.aspx) * [Cancel Async Tasks after a Period of Time (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/jj155760.aspx) * [Cancel Remaining Async Tasks after One Is Complete (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/jj155758.aspx) * [Start Multiple Async Tasks and Process Them As They Complete (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/jj155756.aspx) | [Async Sample: Fine Tuning Your Application (C# and Visual Basic)](http://go.microsoft.com/fwlink/p/?LinkID=255046&clcid=0x409) |
| [Handling Reentrancy in Async Apps (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/jj651641.aspx) | Shows how to handle cases in which an active asynchronous operation is restarted while it’s running. |  |
| [WhenAny: Bridging between the .NET Framework and the Windows Runtime (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/jj635140.aspx) | Shows how to bridge between Task types in the .NET Framework and IAsyncOperations in the Windows Runtime so that you can use [WhenAny](http://msdn.microsoft.com/en-us/library/system.threading.tasks.task.whenany.aspx) with a Windows Runtime method. | [Async Sample: Bridging between .NET and Windows Runtime (AsTask and WhenAny)](http://go.microsoft.com/fwlink/p/?LinkID=260638) |
| [Async Cancellation: Bridging between the .NET Framework and the Windows Runtime (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/jj631575.aspx) | Shows how to bridge between Task types in the .NET Framework and IAsyncOperations in the Windows Runtime so that you can use [CancellationTokenSource](http://msdn.microsoft.com/en-us/library/system.threading.cancellationtokensource.aspx) with a Windows Runtime method. | [Async Sample: Bridging between .NET and Windows Runtime (AsTask & Cancellation)](http://go.microsoft.com/fwlink/p/?LinkId=263004) |
| [Using Async for File Access (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/jj155757.aspx) | Lists and demonstrates the benefits of using async and await to access files. |  |
| [Walkthrough: Using the Debugger with Async Methods](http://msdn.microsoft.com/en-us/library/jj155813.aspx) | Demonstrates the control flow at an await statement, and demonstrates the behavior of the Step Into, Step Over, and Step Out commands within async methods. |  |
| [Task-based Asynchronous Pattern (TAP)](http://msdn.microsoft.com/en-us/library/hh873175.aspx) | Describes a new pattern for asynchrony in the .NET Framework. The pattern is based on the [Task](http://msdn.microsoft.com/en-us/library/system.threading.tasks.task.aspx) and [Task<TResult>](http://msdn.microsoft.com/en-us/library/dd321424.aspx) types. |  |
| [Quickstart: Calling asynchronous APIs in C# or Visual Basic](http://go.microsoft.com/fwlink/p/?LinkId=248545) | Shows how to use async and await in a Windows Store app. |  |
| [Asynchronous programming (Windows Store apps)](http://go.microsoft.com/fwlink/p/?LinkId=259592) | Provides an overview of asynchronous programming in the Windows Runtime. |  |
| [Async Videos on Channel 9](http://go.microsoft.com/fwlink/p/?LinkID=267466) | Provides links to a variety of videos about async programming. |  |

[Complete Example](javascript:void(0))

The following code is the MainWindow.xaml.vb or MainWindow.xaml.cs file from the Windows Presentation Foundation (WPF) application that this topic discusses. You can download the sample from [Async Sample: Example from "Asynchronous Programming with Async and Await"](http://go.microsoft.com/fwlink/p/?LinkID=261549).

C#

[VB](http://msdn.microsoft.com/en-us/library/hh191443(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-4)

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Data;

using System.Windows.Documents;

using System.Windows.Input;

using System.Windows.Media;

using System.Windows.Media.Imaging;

using System.Windows.Navigation;

using System.Windows.Shapes;

// 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.

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);

}

// 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();

// 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.

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";

}

}

}

// Sample Output:

// Working . . . . . . .

// Length of the downloaded string: 41564.

# Walkthrough: Accessing the Web by Using Async and Await (C# and Visual Basic)

**Visual Studio 2013**

[Other Versions](javascript:;)

Description: http://i.msdn.microsoft.com/Areas/Epx/Content/Images/ImageSprite.png

* [Visual Studio 2012](http://msdn.microsoft.com/en-us/library/hh300224(d=printer,v=vs.110).aspx)

This topic has not yet been rated - [Rate this topic](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx#feedback)

You can write asynchronous programs more easily and intuitively by using features that are introduced in Visual Studio 2012. You can write asynchronous code that looks like synchronous code and let the compiler handle the difficult callback functions and continuations that asynchronous code usually entails.

For more information about the Async feature, see [Asynchronous Programming with Async and Await (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/hh191443.aspx).

This walkthrough starts with a synchronous Windows Presentation Foundation (WPF) application that sums the number of bytes in a list of websites. The walkthrough then converts the application to an asynchronous solution by using the new features.

If you don't want to build the applications yourself, you can download "Async Sample: Accessing the Web Walkthrough (C# and Visual Basic)" from [Developer Code Samples](http://go.microsoft.com/fwlink/?LinkId=255191).

In this walkthrough, you complete the following tasks:

* [Create a WPF application](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx#BKMK_CreateWPFApp) .
* [Design a simple WPF MainWindow window](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx#BKMK_DesignWPFMainWin) .
* [Add a reference](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx#BKMK_AddReference) .
* [Add Imports statements or using directives](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx#BKMK_AddStatesandDirs) .
* [Create a synchronous solution](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx#BKMK_CreatSynchApp) .
* [Test the synchronous solution](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx#BKMK_TestSynchSol) .
* [Convert GetURLContents to an asynchronous method](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx#BKMK_ConvertGtBtArr) .
* [Convert SumPageSizes to an asynchronous method](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx#BKMK_ConvertSumPagSzs) .
* [Convert startButton\_Click to an asynchronous method](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx#BKMK_Cnvrtbttn1) .
* [Test the asynchronous solution](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx#BKMK_testAsynchSolution) .
* [Replace GetURLContentsAsync with a .NET Framework method](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx#BKMK_ReplaceGetByteArrayAsync) .
* [Complete Code Examples from the Walkthrough](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx#BKMK_CompleteCodeExamples)

[Prerequisites](javascript:void(0))

Visual Studio 2012 must be installed on your computer. For more information, see the [Microsoft website](http://go.microsoft.com/fwlink/?LinkId=235233).

### To create a WPF application

1. Start Visual Studio.
2. On the menu bar, choose File, New, Project.

The New Project dialog box opens.

1. In the Installed Templates pane, choose Visual Basic or Visual C#, and then choose WPF Application from the list of project types.
2. In the Name text box, enter AsyncExampleWPF, and then choose the OK button.

The new project appears in Solution Explorer.

### To design a simple WPF MainWindow

1. In the Visual Studio Code Editor, choose the MainWindow.xaml tab.
2. If the Toolbox window isn’t visible, open the View menu, and then choose Toolbox.
3. Add a Button control and a TextBox control to the MainWindow window.
4. Highlight the TextBox control and, in the Properties window, set the following values:
   * Set the Name property to resultsTextBox.
   * Set the Height property to 250.
   * Set the Width property to 500.
   * On the Text tab, specify a monospaced font, such as Lucida Console or Global Monospace.
5. Highlight the Button control and, in the Properties window, set the following values:
   * Set the Name property to startButton.
   * Change the value of the Content property from Button to Start.
6. Position the text box and the button so that both appear in the MainWindow window.

For more information about the WPF XAML Designer, see [Creating a UI by using XAML Designer](http://msdn.microsoft.com/en-us/library/hh921077.aspx).

### To add a reference

1. In Solution Explorer, highlight your project's name.
2. On the menu bar, choose Project, Add Reference.

The Reference Manager dialog box appears.

1. At the top of the dialog box, verify that your project is targeting the .NET Framework 4.5.
2. In the Assemblies area, choose Framework if it isn’t already chosen.
3. In the list of names, select the System.Net.Http check box.
4. Choose the OK button to close the dialog box.

### To add necessary Imports statements or using directives

1. In Solution Explorer, open the shortcut menu for MainWindow.xaml.vb or MainWindow.xaml.cs, and then choose View Code.
2. Add the following Imports statements (Visual Basic) or using directives (C#) at the top of the code file if they’re not already present.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-1)

using System.Net.Http;

using System.Net;

using System.IO;

### To create a synchronous application

1. In the design window, MainWindow.xaml, double-click the Start button to create the startButton\_Click event handler in MainWindow.xaml.vb or MainWindow.xaml.cs. As an alternative, highlight the Start button, choose the Event handlers for the selected elements icon in the Properties window, and then enter startButton\_Click in the Click text box.
2. In MainWindow.xaml.vb or MainWindow.xaml.cs, copy the following code into the body of startButton\_Click.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-2)

resultsTextBox.Clear();

SumPageSizes();

resultsTextBox.Text += "\r\nControl returned to startButton\_Click.";

The code calls the method that drives the application, SumPageSizes, and displays a message when control returns to startButton\_Click.

1. The code for the synchronous solution contains the following four methods:
   * SumPageSizes, which gets a list of webpage URLs from SetUpURLList and then calls GetURLContents and DisplayResults to process each URL.
   * SetUpURLList, which makes and returns a list of web addresses.
   * GetURLContents, which downloads the contents of each website and returns the contents as a byte array.
   * DisplayResults, which displays the number of bytes in the byte array for each URL.

Copy the following four methods, and then paste them under the startButton\_Click event handler in MainWindow.xaml.vb or MainWindow.xaml.cs.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-3)

private void SumPageSizes()

{

// Make a list of web addresses.

List<string> urlList = SetUpURLList();

var total = 0;

foreach (var url in urlList)

{

// GetURLContents returns the contents of url as a byte array.

byte[] urlContents = GetURLContents(url);

DisplayResults(url, urlContents);

// Update the total.

total += urlContents.Length;

}

// Display the total count for all of the web addresses.

resultsTextBox.Text +=

string.Format("\r\n\r\nTotal bytes returned: {0}\r\n", total);

}

private List<string> SetUpURLList()

{

var urls = new List<string>

{

"http://msdn.microsoft.com/library/windows/apps/br211380.aspx",

"http://msdn.microsoft.com",

"http://msdn.microsoft.com/en-us/library/hh290136.aspx",

"http://msdn.microsoft.com/en-us/library/ee256749.aspx",

"http://msdn.microsoft.com/en-us/library/hh290138.aspx",

"http://msdn.microsoft.com/en-us/library/hh290140.aspx",

"http://msdn.microsoft.com/en-us/library/dd470362.aspx",

"http://msdn.microsoft.com/en-us/library/aa578028.aspx",

"http://msdn.microsoft.com/en-us/library/ms404677.aspx",

"http://msdn.microsoft.com/en-us/library/ff730837.aspx"

};

return urls;

}

private byte[] GetURLContents(string url)

{

// The downloaded resource ends up in the variable named content.

var content = new MemoryStream();

// Initialize an HttpWebRequest for the current URL.

var webReq = (HttpWebRequest)WebRequest.Create(url);

// Send the request to the Internet resource and wait for

// the response.

// Note: you can't use HttpWebRequest.GetResponse in a Windows Store app.

using (WebResponse response = webReq.GetResponse())

{

// Get the data stream that is associated with the specified URL.

using (Stream responseStream = response.GetResponseStream())

{

// Read the bytes in responseStream and copy them to content.

responseStream.CopyTo(content);

}

}

// Return the result as a byte array.

return content.ToArray();

}

private void DisplayResults(string url, byte[] content)

{

// Display the length of each website. The string format

// is designed to be used with a monospaced font, such as

// Lucida Console or Global Monospace.

var bytes = content.Length;

// Strip off the "http://".

var displayURL = url.Replace("http://", "");

resultsTextBox.Text += string.Format("\n{0,-58} {1,8}", displayURL, bytes);

}

### To test the synchronous solution

* Choose the F5 key to run the program, and then choose the Start button.

Output that resembles the following list should appear.

msdn.microsoft.com/library/windows/apps/br211380.aspx 383832

msdn.microsoft.com 33964

msdn.microsoft.com/en-us/library/hh290136.aspx 225793

msdn.microsoft.com/en-us/library/ee256749.aspx 143577

msdn.microsoft.com/en-us/library/hh290138.aspx 237372

msdn.microsoft.com/en-us/library/hh290140.aspx 128279

msdn.microsoft.com/en-us/library/dd470362.aspx 157649

msdn.microsoft.com/en-us/library/aa578028.aspx 204457

msdn.microsoft.com/en-us/library/ms404677.aspx 176405

msdn.microsoft.com/en-us/library/ff730837.aspx 143474

Total bytes returned: 1834802

Control returned to startButton\_Click.

Notice that it takes a few seconds to display the counts. During that time, the UI thread is blocked while it waits for requested resources to download. As a result, you can't move, maximize, minimize, or even close the display window after you choose the Start button. These efforts fail until the byte counts start to appear. If a website isn’t responding, you have no indication of which site failed. It is difficult even to stop waiting and close the program.

Compare this behavior to the [behavior after you convert the app](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx#BKMK_CompleteCodeExamples) to an asynchronous solution.

### To convert GetURLContents to an asynchronous method

1. To convert the synchronous solution to an asynchronous solution, the best place to start is in GetURLContents because the calls to the [HttpWebRequest](http://msdn.microsoft.com/en-us/library/system.net.httpwebrequest.aspx) method [GetResponse](http://msdn.microsoft.com/en-us/library/system.net.httpwebrequest.getresponse.aspx) and to the [Stream](http://msdn.microsoft.com/en-us/library/system.io.stream.aspx) method [CopyTo](http://msdn.microsoft.com/en-us/library/system.io.stream.copyto.aspx) are where the application accesses the web. The .NET Framework makes the conversion easy by supplying asynchronous versions of both methods.

For more information about the methods that are used in GetURLContents, see [WebRequest](http://msdn.microsoft.com/en-us/library/system.net.webrequest.aspx).

|  |
| --- |
| **Note** |
| As you follow the steps in this walkthrough, several compiler errors appear. You can ignore them and continue with the walkthrough. |

Change the method that's called in the third line of GetURLContents from GetResponse to the asynchronous, task-based [GetResponseAsync](http://msdn.microsoft.com/en-us/library/system.net.webrequest.getresponseasync.aspx) method.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-5)

using (WebResponse response = webReq.GetResponseAsync())

1. GetResponseAsync returns a [Task<TResult>](http://msdn.microsoft.com/en-us/library/dd321424.aspx). In this case, the task return variable, TResult, has type [WebResponse](http://msdn.microsoft.com/en-us/library/system.net.webresponse.aspx). The task is a promise to produce an actual WebResponse object after the requested data has been downloaded and the task has run to completion.

To retrieve the WebResponse value from the task, apply an [Await](http://msdn.microsoft.com/en-us/library/hh156570.aspx) (Visual Basic) or [await](http://msdn.microsoft.com/en-us/library/hh156528.aspx) (C#) operator to the call to GetResponseAsync, as the following code shows.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-6)

using (WebResponse response = await webReq.GetResponseAsync())

The await operator suspends the execution of the current method, GetURLContents, until the awaited task is complete. In the meantime, control returns to the caller of the current method. In this example, the current method is GetURLContents, and the caller is SumPageSizes. When the task is finished, the promised WebResponse object is produced as the value of the awaited task and assigned to the variable response.

The previous statement can be separated into the following two statements to clarify what happens.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-7)

//Task<WebResponse> responseTask = webReq.GetResponseAsync();

//using (WebResponse response = await responseTask)

The call to webReq.GetResponseAsync returns a Task(Of WebResponse) or Task<WebResponse>. Then an await operator is applied to the task to retrieve the WebResponse value.

If your async method has work to do that doesn’t depend on the completion of the task, the method can continue with that work between these two statements, after the call to the async method and before the await operator is applied. For examples, see [How to: Make Multiple Web Requests in Parallel by Using Async and Await (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/hh696703.aspx) and [How to: Extend the Async Walkthrough by Using Task.WhenAll (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/hh556530.aspx).

1. Because you added the Await or await operator in the previous step, a compiler error occurs. The operator can be used only in methods that are marked with the [Async](http://msdn.microsoft.com/en-us/library/hh191564.aspx) (Visual Basic) or [async](http://msdn.microsoft.com/en-us/library/hh156513.aspx) (C#) modifier. Ignore the error while you repeat the conversion steps to replace the call to CopyTo with a call to CopyToAsync.
   * Change the name of the method that’s called to [CopyToAsync](http://msdn.microsoft.com/en-us/library/system.io.stream.copytoasync.aspx).
   * The CopyTo or CopyToAsync method copies bytes to its argument, content, and doesn’t return a meaningful value. In the synchronous version, the call to CopyTo is a simple statement that doesn't return a value. The asynchronous version, CopyToAsync, returns a [Task](http://msdn.microsoft.com/en-us/library/system.threading.tasks.task.aspx). The task functions like "Task(void)" and enables the method to be awaited. Apply Await or await to the call to CopyToAsync, as the following code shows.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-8)

await responseStream.CopyToAsync(content);

The previous statement abbreviates the following two lines of code.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-9)

// CopyToAsync returns a Task, not a Task<T>.

//Task copyTask = responseStream.CopyToAsync(content);

// When copyTask is completed, content contains a copy of

// responseStream.

//await copyTask;

1. All that remains to be done in GetURLContents is to adjust the method signature. You can use the Await or await operator only in methods that are marked with the [Async](http://msdn.microsoft.com/en-us/library/hh191564.aspx) (Visual Basic) or [async](http://msdn.microsoft.com/en-us/library/hh156513.aspx) (C#) modifier. Add the modifier to mark the method as an async method, as the following code shows.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-10)

private async byte[] GetURLContents(string url)

1. The return type of an async method can only be [Task](http://msdn.microsoft.com/en-us/library/system.threading.tasks.task.aspx), [Task<TResult>](http://msdn.microsoft.com/en-us/library/dd321424.aspx), or void in C#. In Visual Basic, the method must be a Function that returns a Task or a Task(Of T), or the method must be a Sub. Typically, a Sub method (Visual Basic) or a return type of void (C#) is used only in an async event handler, where Sub or void is required. In other cases, you use Task(T) if the completed method has a [Return](http://msdn.microsoft.com/en-us/library/2e34641s.aspx) or [return](http://msdn.microsoft.com/en-us/library/1h3swy84.aspx) statement that returns a value of type T, and you use Task if the completed method doesn’t return a meaningful value. You can think of the Task return type as meaning "Task(void)."

For more information, see [Async Return Types (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/hh524395.aspx).

Method GetURLContents has a return statement, and the statement returns a byte array. Therefore, the return type of the async version is Task(T), where T is a byte array. Make the following changes in the method signature:

* + Change the return type to Task(Of Byte()) (Visual Basic) or Task<byte[]> (C#).
  + By convention, asynchronous methods have names that end in "Async," so rename the method GetURLContentsAsync.

The following code shows these changes.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-11)

private async Task<byte[]> GetURLContentsAsync(string url)

With those few changes, the conversion of GetURLContents to an asynchronous method is complete.

### To convert SumPageSizes to an asynchronous method

1. Repeat the steps from the previous procedure for SumPageSizes. First, change the call to GetURLContents to an asynchronous call.
   * Change the name of the method that’s called from GetURLContents to GetURLContentsAsync, if you haven't already done so.
   * Apply Await or await to the task that GetURLContentsAsync returns to obtain the byte array value.

The following code shows these changes.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-12)

byte[] urlContents = await GetURLContentsAsync(url);

The previous assignment abbreviates the following two lines of code.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-13)

// GetURLContentsAsync returns a Task<T>. At completion, the task

// produces a byte array.

//Task<byte[]> getContentsTask = GetURLContentsAsync(url);

//byte[] urlContents = await getContentsTask;

1. Make the following changes in the method's signature:
   * Mark the method with the Async or async modifier.
   * Add "Async" to the method name.
   * There is no task return variable, T, this time because SumPageSizesAsync doesn’t return a value for T. (The method has no Return or return statement.) However, the method must return a Task to be awaitable. Therefore, make one of the following changes:
     + In Visual Basic, change the method type from Sub to Function. The return type of the function is Task.
     + In C#, change the return type of the method from void to Task.

The following code shows these changes.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-14)

private async Task SumPageSizesAsync()

The conversion of SumPageSizes to SumPageSizesAsync is complete.

### To convert startButton\_Click to an asynchronous method

1. In the event handler, change the name of the called method from SumPageSizes to SumPageSizesAsync, if you haven’t already done so.
2. Because SumPageSizesAsync is an async method, change the code in the event handler to await the result.

The call to SumPageSizesAsync mirrors the call to CopyToAsync in GetURLContentsAsync. The call returns a Task, not a Task(T).

As in previous procedures, you can convert the call by using one statement or two statements. The following code shows these changes.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-15)

// One-step async call.

await SumPageSizesAsync();

// Two-step async call.

//Task sumTask = SumPageSizesAsync();

//await sumTask;

1. To prevent accidentally reentering the operation, add the following statement at the top of startButton\_Click to disable the Start button.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-16)

// Disable the button until the operation is complete.

startButton.IsEnabled = false;

You can reenable the button at the end of the event handler.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-17)

// Reenable the button in case you want to run the operation again.

startButton.IsEnabled = true;

For more information about reentrancy, see [Handling Reentrancy in Async Apps (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/jj651641.aspx).

1. Finally, add the Async or async modifier to the declaration so that the event handler can await SumPagSizesAsync.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-18)

private async void startButton\_Click(object sender, RoutedEventArgs e)

Typically, the names of event handlers aren’t changed. The return type isn’t changed to Task because event handlers must return void in C# or be Sub procedures in Visual Basic. Therefore, the return type to Task.

The conversion of the project from synchronous to asynchronous processing is complete.

### To test the asynchronous solution

1. Choose the F5 key to run the program, and then choose the Start button.
2. Output that resembles the output of the synchronous solution should appear. However, notice the following differences.
   * The results don’t all occur at the same time, after the processing is complete. For example, both programs contain a line in startButton\_Click that clears the text box. The intent is to clear the text box between runs if you choose the Start button for a second time, after one set of results has appeared. In the synchronous version, the text box is cleared just before the counts appear for the second time, when the downloads are completed and the UI thread is free to do other work. In the asynchronous version, the text box clears immediately after you choose the Start button.
   * Most importantly, the UI thread isn’t blocked during the downloads. You can move or resize the window while the web resources are being downloaded, counted, and displayed. If one of the websites is slow or not responding, you can cancel the operation by choosing the Close button (the x in the red field in the upper-right corner).

### To replace method GetURLContentsAsync with a .NET Framework method

1. The .NET Framework 4.5 provides many async methods that you can use. One of them, the [HttpClient](http://msdn.microsoft.com/en-us/library/system.net.http.httpclient.aspx) method [GetByteArrayAsync(String)](http://msdn.microsoft.com/en-us/library/hh551751.aspx), does just what you need for this walkthrough. You can use it instead of the GetURLContentsAsync method that you created in an earlier procedure.

The first step is to create an HttpClient object in method SumPageSizesAsync. Add the following declaration at the start of the method.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-19)

// Declare an HttpClient object and increase the buffer size. The

// default buffer size is 65,536.

HttpClient client =

new HttpClient() { MaxResponseContentBufferSize = 1000000 };

1. In SumPageSizesAsync, replace the call to your GetURLContentsAsync method with a call to the HttpClient method.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-20)

byte[] urlContents = await client.GetByteArrayAsync(url);

1. Remove or comment out the GetURLContentsAsync method that you wrote.
2. Choose the F5 key to run the program, and then choose the Start button.

The behavior of this version of the project should match the behavior that the "To test the asynchronous solution" procedure describes but with even less effort from you.

[Example](javascript:void(0))

The following code contains the full example of the conversion from a synchronous to an asynchronous solution by using the asynchronous GetURLContentsAsync method that you wrote. Notice that it strongly resembles the original, synchronous solution.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-21)

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Data;

using System.Windows.Documents;

using System.Windows.Input;

using System.Windows.Media;

using System.Windows.Media.Imaging;

using System.Windows.Navigation;

using System.Windows.Shapes;

// Add the following using directives, and add a reference for System.Net.Http.

using System.Net.Http;

using System.IO;

using System.Net;

namespace AsyncExampleWPF

{

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

}

private async void startButton\_Click(object sender, RoutedEventArgs e)

{

// Disable the button until the operation is complete.

startButton.IsEnabled = false;

resultsTextBox.Clear();

// One-step async call.

await SumPageSizesAsync();

// Two-step async call.

//Task sumTask = SumPageSizesAsync();

//await sumTask;

resultsTextBox.Text += "\r\nControl returned to startButton\_Click.\r\n";

// Reenable the button in case you want to run the operation again.

startButton.IsEnabled = true;

}

private async Task SumPageSizesAsync()

{

// Make a list of web addresses.

List<string> urlList = SetUpURLList();

var total = 0;

foreach (var url in urlList)

{

byte[] urlContents = await GetURLContentsAsync(url);

// The previous line abbreviates the following two assignment statements.

// GetURLContentsAsync returns a Task<T>. At completion, the task

// produces a byte array.

//Task<byte[]> getContentsTask = GetURLContentsAsync(url);

//byte[] urlContents = await getContentsTask;

DisplayResults(url, urlContents);

// Update the total.

total += urlContents.Length;

}

// Display the total count for all of the websites.

resultsTextBox.Text +=

string.Format("\r\n\r\nTotal bytes returned: {0}\r\n", total);

}

private List<string> SetUpURLList()

{

List<string> urls = new List<string>

{

"http://msdn.microsoft.com/library/windows/apps/br211380.aspx",

"http://msdn.microsoft.com",

"http://msdn.microsoft.com/en-us/library/hh290136.aspx",

"http://msdn.microsoft.com/en-us/library/ee256749.aspx",

"http://msdn.microsoft.com/en-us/library/hh290138.aspx",

"http://msdn.microsoft.com/en-us/library/hh290140.aspx",

"http://msdn.microsoft.com/en-us/library/dd470362.aspx",

"http://msdn.microsoft.com/en-us/library/aa578028.aspx",

"http://msdn.microsoft.com/en-us/library/ms404677.aspx",

"http://msdn.microsoft.com/en-us/library/ff730837.aspx"

};

return urls;

}

private async Task<byte[]> GetURLContentsAsync(string url)

{

// The downloaded resource ends up in the variable named content.

var content = new MemoryStream();

// Initialize an HttpWebRequest for the current URL.

var webReq = (HttpWebRequest)WebRequest.Create(url);

// Send the request to the Internet resource and wait for

// the response.

using (WebResponse response = await webReq.GetResponseAsync())

// The previous statement abbreviates the following two statements.

//Task<WebResponse> responseTask = webReq.GetResponseAsync();

//using (WebResponse response = await responseTask)

{

// Get the data stream that is associated with the specified url.

using (Stream responseStream = response.GetResponseStream())

{

// Read the bytes in responseStream and copy them to content.

await responseStream.CopyToAsync(content);

// The previous statement abbreviates the following two statements.

// CopyToAsync returns a Task, not a Task<T>.

//Task copyTask = responseStream.CopyToAsync(content);

// When copyTask is completed, content contains a copy of

// responseStream.

//await copyTask;

}

}

// Return the result as a byte array.

return content.ToArray();

}

private void DisplayResults(string url, byte[] content)

{

// Display the length of each website. The string format

// is designed to be used with a monospaced font, such as

// Lucida Console or Global Monospace.

var bytes = content.Length;

// Strip off the "http://".

var displayURL = url.Replace("http://", "");

resultsTextBox.Text += string.Format("\n{0,-58} {1,8}", displayURL, bytes);

}

}

}

The following code contains the full example of the solution that uses the HttpClient method, GetByteArrayAsync.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh300224(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-22)

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Data;

using System.Windows.Documents;

using System.Windows.Input;

using System.Windows.Media;

using System.Windows.Media.Imaging;

using System.Windows.Navigation;

using System.Windows.Shapes;

// Add the following using directives, and add a reference for System.Net.Http.

using System.Net.Http;

using System.IO;

using System.Net;

namespace AsyncExampleWPF

{

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

}

private async void startButton\_Click(object sender, RoutedEventArgs e)

{

resultsTextBox.Clear();

// Disable the button until the operation is complete.

startButton.IsEnabled = false;

// One-step async call.

await SumPageSizesAsync();

//// Two-step async call.

//Task sumTask = SumPageSizesAsync();

//await sumTask;

resultsTextBox.Text += "\r\nControl returned to startButton\_Click.\r\n";

// Reenable the button in case you want to run the operation again.

startButton.IsEnabled = true;

}

private async Task SumPageSizesAsync()

{

// Declare an HttpClient object and increase the buffer size. The

// default buffer size is 65,536.

HttpClient client =

new HttpClient() { MaxResponseContentBufferSize = 1000000 };

// Make a list of web addresses.

List<string> urlList = SetUpURLList();

var total = 0;

foreach (var url in urlList)

{

// GetByteArrayAsync returns a task. At completion, the task

// produces a byte array.

byte[] urlContents = await client.GetByteArrayAsync(url);

// The following two lines can replace the previous assignment statement.

//Task<byte[]> getContentsTask = client.GetByteArrayAsync(url);

//byte[] urlContents = await getContentsTask;

DisplayResults(url, urlContents);

// Update the total.

total += urlContents.Length;

}

// Display the total count for all of the websites.

resultsTextBox.Text +=

string.Format("\r\n\r\nTotal bytes returned: {0}\r\n", total);

}

private List<string> SetUpURLList()

{

List<string> urls = new List<string>

{

"http://msdn.microsoft.com/library/windows/apps/br211380.aspx",

"http://msdn.microsoft.com",

"http://msdn.microsoft.com/en-us/library/hh290136.aspx",

"http://msdn.microsoft.com/en-us/library/ee256749.aspx",

"http://msdn.microsoft.com/en-us/library/hh290138.aspx",

"http://msdn.microsoft.com/en-us/library/hh290140.aspx",

"http://msdn.microsoft.com/en-us/library/dd470362.aspx",

"http://msdn.microsoft.com/en-us/library/aa578028.aspx",

"http://msdn.microsoft.com/en-us/library/ms404677.aspx",

"http://msdn.microsoft.com/en-us/library/ff730837.aspx"

};

return urls;

}

private void DisplayResults(string url, byte[] content)

{

// Display the length of each website. The string format

// is designed to be used with a monospaced font, such as

// Lucida Console or Global Monospace.

var bytes = content.Length;

// Strip off the "http://".

var displayURL = url.Replace("http://", "");

resultsTextBox.Text += string.Format("\n{0,-58} {1,8}", displayURL, bytes);

}

}

}

**async (C# Reference)**

**Visual Studio 2013**

[Other Versions](javascript:;)

Description: http://i.msdn.microsoft.com/Areas/Epx/Content/Images/ImageSprite.png

* [Visual Studio 2012](http://msdn.microsoft.com/en-us/library/hh156513(d=printer,v=vs.110).aspx)

By using the async modifier, you specify that a method, [lambda expression](http://msdn.microsoft.com/en-us/library/bb397687.aspx), or [anonymous method](http://msdn.microsoft.com/en-us/library/0yw3tz5k.aspx) is asynchronous. If you use this modifier on a method or expression, it's referred to as an async method.

C#

public async Task<int> ExampleMethodAsync()

{

// . . . .

}

If you're new to asynchronous programming, you can find an introduction in [Asynchronous Programming with Async and Await (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/hh191443.aspx).

An async method provides a convenient way to do potentially long-running work without blocking the caller's thread. The caller (say, M1) of an async method can resume its work without waiting for the async method to finish. However, M1 typically uses the await keyword so that it returns immediately, allowing M1’s caller to resume work or return to the thread’s synchronization context (or message pump).

string contents = await contentsTask;

The method runs synchronously until it reaches its first await expression, at which point the method is suspended until the awaited task is complete. In the meantime, control returns to the caller of the method, as the example in the next section shows.

If the method that the async keyword modifies doesn't contain an await expression or statement, the method executes synchronously. A compiler warning alerts you to any async methods that don't contain await, because that situation might indicate an error. See [Compiler Warning (level 1) CS4014](http://msdn.microsoft.com/en-us/library/hh873131.aspx).

The async keyword is contextual in that it's a keyword only when it modifies a method, a lambda expression, or an anonymous method. In all other contexts, it's interpreted as an identifier.

[Example](javascript:void(0))

The following example shows the structure and flow of control between an async event handler, StartButton\_Click, and an async method, ExampleMethodAsync. The result from the async method is the length of a downloaded website. The code is suitable for a Windows Presentation Foundation (WPF) app or Windows Store app that you create in Visual Studio 2013; see the code comments for setting up the app.

C#

// You can run this code in Visual Studio 2013 as a WPF app or a Windows Store app.

// You need a button (StartButton) and a textbox (ResultsTextBox).

// Remember to set the names and handler so that you have something like this:

// <Button Content="Button" HorizontalAlignment="Left" Margin="88,77,0,0" VerticalAlignment="Top" Width="75"

// Click="StartButton\_Click" Name="StartButton"/>

// <TextBox HorizontalAlignment="Left" Height="137" Margin="88,140,0,0" TextWrapping="Wrap"

// Text="TextBox" VerticalAlignment="Top" Width="310" Name="ResultsTextBox"/>

// To run the code as a WPF app:

// paste this code into the MainWindow class in MainWindow.xaml.cs,

// add a reference to System.Net.Http, and

// add a using directive for System.Net.Http.

// To run the code as a Windows Store app:

// paste this code into the MainPage class in MainPage.xaml.cs, and

// add using directives for System.Net.Http and System.Threading.Tasks.

private async void StartButton\_Click(object sender, RoutedEventArgs e)

{

// ExampleMethodAsync returns a Task<int>, which means that the method

// eventually produces an int result. However, ExampleMethodAsync returns

// the Task<int> value as soon as it reaches an await.

ResultsTextBox.Text += "\n";

try

{

int length = await ExampleMethodAsync();

// Note that you could put "await ExampleMethodAsync()" in the next line where

// "length" is, but due to when '+=' fetches the value of ResultsTextBox, you

// would not see the global side effect of ExampleMethodAsync setting the text.

ResultsTextBox.Text += String.Format("Length: {0}\n", length);

}

catch (Exception)

{

// Process the exception if one occurs.

}

}

public async Task<int> ExampleMethodAsync()

{

var httpClient = new HttpClient();

int exampleInt = (await httpClient.GetStringAsync("http://msdn.microsoft.com")).Length;

ResultsTextBox.Text += "Preparing to finish ExampleMethodAsync.\n";

// After the following return statement, any method that's awaiting

// ExampleMethodAsync (in this case, StartButton\_Click) can get the

// integer result.

return exampleInt;

}

// Output:

// Preparing to finish ExampleMethodAsync.

// Length: 53292

|  |
| --- |
| **Important** |
| For more information about tasks and the code that executes while waiting for a task, see [Asynchronous Programming with Async and Await (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/hh191443.aspx). For a full WPF example that uses similar elements, see [Walkthrough: Accessing the Web by Using Async and Await (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/hh300224.aspx). You can download the walkthrough code from [Developer Code Samples](http://go.microsoft.com/fwlink/?LinkId=255191). |

[Return Types](javascript:void(0))

An async method can have a return type of [Task](http://msdn.microsoft.com/en-us/library/system.threading.tasks.task.aspx), [Task<TResult>](http://msdn.microsoft.com/en-us/library/dd321424.aspx), or [void](http://msdn.microsoft.com/en-us/library/yah0tteb.aspx). The method can't declare any [ref](http://msdn.microsoft.com/en-us/library/14akc2c7.aspx) or [out](http://msdn.microsoft.com/en-us/library/t3c3bfhx.aspx) parameters, but it can call methods that have such parameters.

You specify Task<TResult> as the return type of an async method if the [return](http://msdn.microsoft.com/en-us/library/1h3swy84.aspx) statement of the method specifies an operand of type TResult. You use Task if no meaningful value is returned when the method is completed. That is, a call to the method returns a Task, but when the Task is completed, any await expression that's awaiting the Task evaluates to void.

You use the void return type primarily to define event handlers, which require that return type. The caller of a void-returning async method can't await it and can't catch exceptions that the method throws.

For more information and examples, see [Async Return Types (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/hh524395.aspx).

**Control Flow in Async Programs (C# and Visual Basic)**

**Visual Studio 2013**

[Other Versions](javascript:;)

Description: http://i.msdn.microsoft.com/Areas/Epx/Content/Images/ImageSprite.png

* [Visual Studio 2012](http://msdn.microsoft.com/en-us/library/hh873191(d=printer,v=vs.110).aspx)

You can write and maintain asynchronous programs more easily by using the Async and Await keywords. However, the results might surprise you if you don't understand how your program operates. This topic traces the flow of control through a simple async program to show you when control moves from one method to another and what information is transferred each time.

|  |
| --- |
| **Note** |
| The Async and Await keywords were introduced in Visual Studio 2012. |

In general, you mark methods that contain asynchronous code with the [Async (Visual Basic)](http://msdn.microsoft.com/en-us/library/hh191564.aspx) or [async (C#)](http://msdn.microsoft.com/en-us/library/hh156513.aspx) modifier. In a method that's marked with an async modifier, you can use an [Await (Visual Basic)](http://msdn.microsoft.com/en-us/library/hh156570.aspx) or [await (C#)](http://msdn.microsoft.com/en-us/library/hh156528.aspx) operator to specify where the method pauses to wait for a called asynchronous process to complete. For more information, see [Asynchronous Programming with Async and Await (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/hh191443.aspx).

The following example uses async methods to download the contents of a specified website as a string and to display the length of the string. The example contains the following two methods.

* startButton\_Click, which calls AccessTheWebAsync and displays the result.
* AccessTheWebAsync, which downloads the contents of a website as a string and returns the length of the string. AccessTheWebAsync uses an asynchronous [HttpClient](http://msdn.microsoft.com/en-us/library/system.net.http.httpclient.aspx) method, [GetStringAsync(String)](http://msdn.microsoft.com/en-us/library/hh551746.aspx), to download the contents.

Numbered display lines appear at strategic points throughout the program to help you understand how the program runs and to explain what happens at each point that is marked. The display lines are labeled "ONE" through "SIX." The labels represent the order in which the program reaches these lines of code.

The following code shows an outline of the program.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh873191(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-1)

public partial class MainWindow : Window

{

// . . .

private async void startButton\_Click(object sender, RoutedEventArgs e)

{

// ONE

Task<int> getLengthTask = AccessTheWebAsync();

// FOUR

int contentLength = await getLengthTask;

// SIX

resultsTextBox.Text +=

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

}

async Task<int> AccessTheWebAsync()

{

// TWO

HttpClient client = new HttpClient();

Task<string> getStringTask =

client.GetStringAsync("http://msdn.microsoft.com");

// THREE

string urlContents = await getStringTask;

// FIVE

return urlContents.Length;

}

}

Each of the labeled locations, "ONE" through "SIX," displays information about the current state of the program. The following output is produced.

ONE: Entering startButton\_Click.

Calling AccessTheWebAsync.

TWO: Entering AccessTheWebAsync.

Calling HttpClient.GetStringAsync.

THREE: Back in AccessTheWebAsync.

Task getStringTask is started.

About to await getStringTask & return a Task<int> to startButton\_Click.

FOUR: Back in startButton\_Click.

Task getLengthTask is started.

About to await getLengthTask -- no caller to return to.

FIVE: Back in AccessTheWebAsync.

Task getStringTask is complete.

Processing the return statement.

Exiting from AccessTheWebAsync.

SIX: Back in startButton\_Click.

Task getLengthTask is finished.

Result from AccessTheWebAsync is stored in contentLength.

About to display contentLength and exit.

Length of the downloaded string: 33946.

[Set Up the Program](javascript:void(0))

You can download the code that this topic uses from MSDN, or you can build it yourself.

|  |
| --- |
| **Note** |
| To run the example, you must have Visual Studio 2012, Visual Studio 2013, Visual Studio Express 2012, Visual Studio Express 2013 for Windows, or the .NET Framework 4.5 or 4.5.1 installed on your computer. |

[Download the Program](javascript:void(0))

You can download the application for this topic from [Async Sample: Control Flow in Async Programs](http://go.microsoft.com/fwlink/?LinkId=255285). The following steps open and run the program.

1. Unzip the downloaded file, and then start Visual Studio.
2. On the menu bar, choose File, Open, Project/Solution.
3. Navigate to the folder that holds the unzipped sample code, open the solution (.sln) file, and then choose the F5 key to build and run the project.

[Build the Program Yourself](javascript:void(0))

The following Windows Presentation Foundation (WPF) project contains the code example for this topic.

To run the project, perform the following steps:

1. Start Visual Studio.
2. On the menu bar, choose File, New, Project.

The New Project dialog box opens.

1. In the Installed Templates pane, choose Visual Basic or Visual C#, and then choose WPF Application from the list of project types.
2. Enter AsyncTracer as the name of the project, and then choose the OK button.

The new project appears in Solution Explorer.

1. In the Visual Studio Code Editor, choose the MainWindow.xaml tab.

If the tab isn’t visible, open the shortcut menu for MainWindow.xaml in Solution Explorer, and then choose View Code.

1. In the XAML view of MainWindow.xaml, replace the code with the following code.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh873191(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-3)

<Window

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008" xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006" mc:Ignorable="d" x:Class="AsyncTracer.MainWindow"

Title="Control Flow Trace" Height="350" Width="592">

<Grid>

<Button x:Name="startButton" Content="Start&#xa;" HorizontalAlignment="Left" Margin="250,10,0,0" VerticalAlignment="Top" Width="75" Height="24" Click="startButton\_Click" d:LayoutOverrides="GridBox"/>

<TextBox x:Name="resultsTextBox" HorizontalAlignment="Left" TextWrapping="Wrap" VerticalAlignment="Bottom" Width="576" Height="265" FontFamily="Lucida Console" FontSize="10" VerticalScrollBarVisibility="Visible" Grid.ColumnSpan="3"/>

</Grid>

</Window>

A simple window that contains a text box and a button appears in the Design view of MainWindow.xaml.

1. Add a reference for [System.Net.Http](http://msdn.microsoft.com/en-us/library/system.net.http.aspx).
2. In Solution Explorer, open the shortcut menu for MainWindow.xaml.vb or MainWindow.xaml.cs, and then choose View Code.
3. In MainWindow.xaml.vb or MainWindow.xaml.cs, replace the code with the following code.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh873191(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-4)

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Data;

using System.Windows.Documents;

using System.Windows.Input;

using System.Windows.Media;

using System.Windows.Media.Imaging;

using System.Windows.Navigation;

using System.Windows.Shapes;

// Add a using directive and a reference for System.Net.Http;

using System.Net.Http;

namespace AsyncTracer

{

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

}

private async void startButton\_Click(object sender, RoutedEventArgs e)

{

// The display lines in the example lead you through the control shifts.

resultsTextBox.Text += "ONE: Entering startButton\_Click.\r\n" +

" Calling AccessTheWebAsync.\r\n";

Task<int> getLengthTask = AccessTheWebAsync();

resultsTextBox.Text += "\r\nFOUR: Back in startButton\_Click.\r\n" +

" Task getLengthTask is started.\r\n" +

" About to await getLengthTask -- no caller to return to.\r\n";

int contentLength = await getLengthTask;

resultsTextBox.Text += "\r\nSIX: Back in startButton\_Click.\r\n" +

" Task getLengthTask is finished.\r\n" +

" Result from AccessTheWebAsync is stored in contentLength.\r\n" +

" About to display contentLength and exit.\r\n";

resultsTextBox.Text +=

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

}

async Task<int> AccessTheWebAsync()

{

resultsTextBox.Text += "\r\nTWO: Entering AccessTheWebAsync.";

// Declare an HttpClient object.

HttpClient client = new HttpClient();

resultsTextBox.Text += "\r\n Calling HttpClient.GetStringAsync.\r\n";

// GetStringAsync returns a Task<string>.

Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");

resultsTextBox.Text += "\r\nTHREE: Back in AccessTheWebAsync.\r\n" +

" Task getStringTask is started.";

// AccessTheWebAsync can continue to work until getStringTask is awaited.

resultsTextBox.Text +=

"\r\n About to await getStringTask and return a Task<int> to startButton\_Click.\r\n";

// Retrieve the website contents when task is complete.

string urlContents = await getStringTask;

resultsTextBox.Text += "\r\nFIVE: Back in AccessTheWebAsync." +

"\r\n Task getStringTask is complete." +

"\r\n Processing the return statement." +

"\r\n Exiting from AccessTheWebAsync.\r\n";

return urlContents.Length;

}

}

}

1. Choose the F5 key to run the program, and then choose the Start button.

The following output should appear.

ONE: Entering startButton\_Click.

Calling AccessTheWebAsync.

TWO: Entering AccessTheWebAsync.

Calling HttpClient.GetStringAsync.

THREE: Back in AccessTheWebAsync.

Task getStringTask is started.

About to await getStringTask & return a Task<int> to startButton\_Click.

FOUR: Back in startButton\_Click.

Task getLengthTask is started.

About to await getLengthTask -- no caller to return to.

FIVE: Back in AccessTheWebAsync.

Task getStringTask is complete.

Processing the return statement.

Exiting from AccessTheWebAsync.

SIX: Back in startButton\_Click.

Task getLengthTask is finished.

Result from AccessTheWebAsync is stored in contentLength.

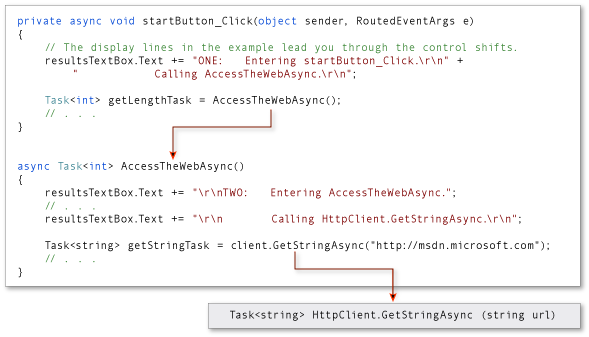
About to display contentLength and exit.

Length of the downloaded string: 33946.

[Trace the Program](javascript:void(0))

[Steps ONE and TWO](javascript:void(0))

The first two display lines trace the path as startButton\_Click calls AccessTheWebAsync, and AccessTheWebAsync calls the asynchronous [HttpClient](http://msdn.microsoft.com/en-us/library/system.net.http.httpclient.aspx) method [GetStringAsync(String)](http://msdn.microsoft.com/en-us/library/hh551746.aspx). The following image outlines the calls from method to method.



The return type of both AccessTheWebAsync and client.GetStringAsync is [Task<TResult>](http://msdn.microsoft.com/en-us/library/dd321424.aspx). For AccessTheWebAsync, TResult is an integer. For GetStringAsync, TResult is a string. For more information about async method return types, see [Async Return Types (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/hh524395.aspx).

A task-returning async method returns a task instance when control shifts back to the caller. Control returns from an async method to its caller either when an Await or await operator is encountered in the called method or when the called method ends. The display lines that are labeled "THREE" through "SIX" trace this part of the process.

[Step THREE](javascript:void(0))

In AccessTheWebAsync, the asynchronous method [GetStringAsync(String)](http://msdn.microsoft.com/en-us/library/hh551746.aspx) is called to download the contents of the target webpage. Control returns from client.GetStringAsync to AccessTheWebAsync when client.GetStringAsync returns.

The client.GetStringAsync method returns a task of string that’s assigned to the getStringTask variable in AccessTheWebAsync. The following line in the example program shows the call to client.GetStringAsync and the assignment.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh873191(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-6)

Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");

You can think of the task as a promise by client.GetStringAsync to produce an actual string eventually. In the meantime, if AccessTheWebAsync has work to do that doesn't depend on the promised string from client.GetStringAsync, that work can continue while client.GetStringAsync waits. In the example, the following lines of output, which are labeled "THREE,” represent the opportunity to do independent work

THREE: Back in AccessTheWebAsync.

Task getStringTask is started.

About to await getStringTask & return a Task<int> to startButton\_Click.

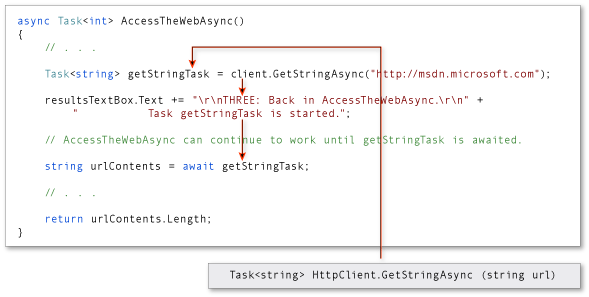
The following statement suspends progress in AccessTheWebAsync when getStringTask is awaited.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh873191(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-8)

string urlContents = await getStringTask;

The following image shows the flow of control from client.GetStringAsync to the assignment to getStringTask and from the creation of getStringTask to the application of an await operator.



The await expression suspends AccessTheWebAsync until client.GetStringAsync returns. In the meantime, control returns to the caller of AccessTheWebAsync, startButton\_Click.

|  |
| --- |
| **Note** |
| Typically, you await the call to an asynchronous method immediately. For example, one of the following assignments could replace the previous code that creates and then awaits getStringTask:   * Visual Basic: Dim urlContents As String = Await client.GetStringAsync("http://msdn.microsoft.com") * C#: string urlContents = await client.GetStringAsync("http://msdn.microsoft.com");   In this topic, the await operator is applied later to accommodate the output lines that mark the flow of control through the program. |

[Step FOUR](javascript:void(0))

The declared return type of AccessTheWebAsync is Task(Of Integer) in Visual Basic and Task<int> in C#. Therefore, when AccessTheWebAsync is suspended, it returns a task of integer to startButton\_Click. You should understand that the returned task isn’t getStringTask. The returned task is a new task of integer that represents what remains to be done in the suspended method, AccessTheWebAsync. The task is a promise from AccessTheWebAsync to produce an integer when the task is complete.

The following statement assigns this task to the getLengthTask variable.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh873191(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-9)

Task<int> getLengthTask = AccessTheWebAsync();

As in AccessTheWebAsync, startButton\_Click can continue with work that doesn’t depend on the results of the asynchronous task (getLengthTask) until the task is awaited. The following output lines represent that work.

FOUR: Back in startButton\_Click.

Task getLengthTask is started.

About to await getLengthTask -- no caller to return to.

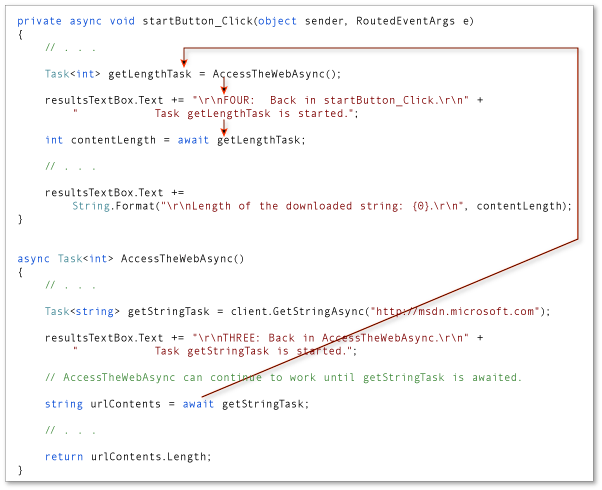
Progress in startButton\_Click is suspended when getLengthTask is awaited. The following assignment statement suspends startButton\_Click until AccessTheWebAsync is complete.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh873191(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-11)

int contentLength = await getLengthTask;

In the following illustration, the arrows show the flow of control from the await expression in AccessTheWebAsync to the assignment of a value to getLengthTask, followed by normal processing in startButton\_Click until getLengthTask is awaited.



[Step FIVE](javascript:void(0))

When client.GetStringAsync signals that it’s complete, processing in AccessTheWebAsync is released from suspension and can continue past the await statement. The following lines of output represent the resumption of processing.

FIVE: Back in AccessTheWebAsync.

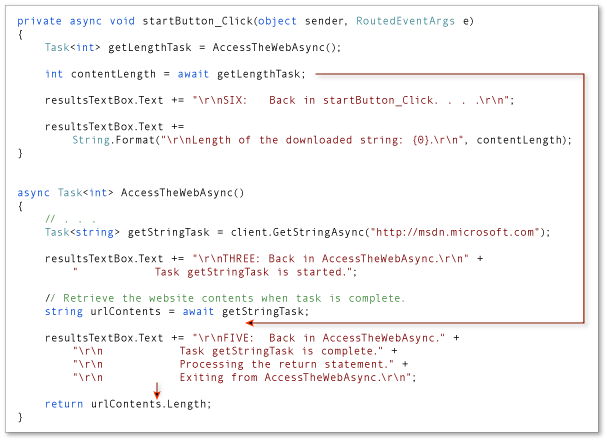
Task getStringTask is complete.

Processing the return statement.

Exiting from AccessTheWebAsync.

The operand of the return statement, urlContents.Length, is stored in the task that AccessTheWebAsync returns. The await expression retrieves that value from getLengthTask in startButton\_Click.

The following image shows the transfer of control after client.GetStringAsync (and getStringTask) are complete.



AccessTheWebAsync runs to completion, and control returns to startButton\_Click, which is awaiting the completion.

[Step SIX](javascript:void(0))

When AccessTheWebAsync signals that it’s complete, processing can continue past the await statement in startButton\_Async. In fact, the program has nothing more to do.

The following lines of output represent the resumption of processing in startButton\_Async:

SIX: Back in startButton\_Click.

Task getLengthTask is finished.

Result from AccessTheWebAsync is stored in contentLength.

About to display contentLength and exit.

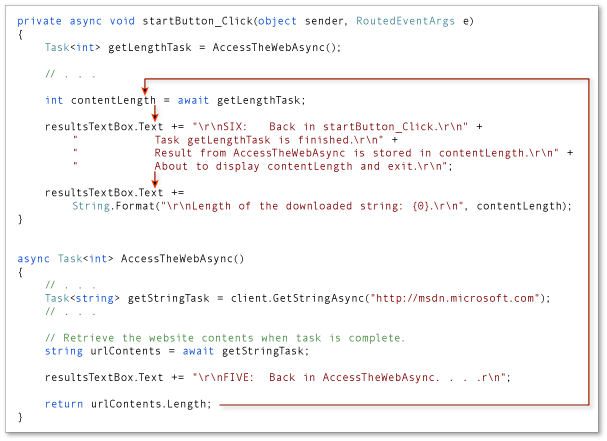
The await expression retrieves from getLengthTask the integer value that’s the operand of the return statement in AccessTheWebAsync. The following statement assigns that value to the contentLength variable.

C#

[VB](http://msdn.microsoft.com/en-us/library/hh873191(d=printer).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-14)

int contentLength = await getLengthTask;

The following image shows the return of control from AccessTheWebAsync to startButton\_Click.



**await (C# Reference)**

**Visual Studio 2013**

The await operator is applied to a task in an asynchronous method to suspend the execution of the method until the awaited task completes. The task represents ongoing work.

The asynchronous method in which await is used must be modified by the [async](http://msdn.microsoft.com/en-us/library/hh156513.aspx) keyword. Such a method, defined by using the async modifier, and usually containing one or more await expressions, is referred to as an async method.

|  |
| --- |
| **Description: NoteNote** |
| The async and await keywords were introduced in Visual Studio 2012. For an introduction to async programming, see [Asynchronous Programming with Async and Await (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/hh191443.aspx). |

The task to which the await operator is applied typically is the return value from a call to a method that implements the [Task-Based Asynchronous Pattern](http://go.microsoft.com/fwlink/?LinkId=204847). Examples include values of type [Task](http://msdn.microsoft.com/en-us/library/system.threading.tasks.task.aspx) or [Task<TResult>](http://msdn.microsoft.com/en-us/library/dd321424.aspx).

In the following code, the [HttpClient](http://msdn.microsoft.com/en-us/library/system.net.http.httpclient.aspx) method [GetByteArrayAsync](http://msdn.microsoft.com/en-us/library/system.net.http.httpclient.getbytearrayasync.aspx) returns a Task<byte[]>, getContentsTask. The task is a promise to produce the actual byte array when the task is complete. The await operator is applied to getContentsTask to suspend execution in SumPageSizesAsync until getContentsTask is complete. In the meantime, control is returned to the caller of SumPageSizesAsync. When getContentsTask is finished, the await expression evaluates to a byte array.

C#

private async Task SumPageSizesAsync()

{

// To use the HttpClient type in desktop apps, you must include a using directive and add a

// reference for the System.Net.Http namespace.

HttpClient client = new HttpClient();

// . . .

Task<byte[]> getContentsTask = client.GetByteArrayAsync(url);

byte[] urlContents = await getContentsTask;

// Equivalently, now that you see how it works, you can write the same thing in a single line.

//byte[] urlContents = await client.GetByteArrayAsync(url);

// . . .

}

|  |
| --- |
| **Description: Important noteImportant** |
| For the complete example, see [Walkthrough: Accessing the Web by Using Async and Await (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/hh300224.aspx). You can download the sample from [Developer Code Samples](http://go.microsoft.com/fwlink/?LinkID=255191&clcid=0x409) on the Microsoft website. The example is in the AsyncWalkthrough\_HttpClient project. |

As shown in the previous example, if await is applied to the result of a method call that returns a Task<TResult>, then the type of the await expression is TResult. If await is applied to the result of a method call that returns a Task, then the type of the await expression is void. The following example illustrates the difference.

C#

// Keyword await used with a method that returns a Task<TResult>.

TResult result = await AsyncMethodThatReturnsTaskTResult();

// Keyword await used with a method that returns a Task.

await AsyncMethodThatReturnsTask();

An await expression does not block the thread on which it is executing. Instead, it causes the compiler to sign up the rest of the async method as a continuation on the awaited task. Control then returns to the caller of the async method. When the task completes, it invokes its continuation, and execution of the async method resumes where it left off.

An await expression can occur only in the body of an immediately enclosing method, lambda expression, or anonymous method that is marked by an async modifier. The term await serves as a keyword only in that context. Elsewhere, it is interpreted as an identifier. Within the method, lambda expression, or anonymous method, an await expression cannot occur in the body of a synchronous function, in a query expression, in the catch or finally block of an [exception handling statement](http://msdn.microsoft.com/en-us/library/s7fekhdy.aspx), in the block of a [lock statement](http://msdn.microsoft.com/en-us/library/c5kehkcz.aspx), or in an [unsafe](http://msdn.microsoft.com/en-us/library/chfa2zb8.aspx) context.

[Exceptions](javascript:void(0))

Most async methods return a [Task](http://msdn.microsoft.com/en-us/library/system.threading.tasks.task.aspx) or [Task<TResult>](http://msdn.microsoft.com/en-us/library/dd321424.aspx). The properties of the returned task carry information about its status and history, such as whether the task is complete, whether the async method caused an exception or was canceled, and what the final result is. The await operator accesses those properties.

If you await a task-returning async method that causes an exception, the await operator rethrows the exception.

If you await a task-returning async method that's canceled, the await operator rethrows an [OperationCanceledException](http://msdn.microsoft.com/en-us/library/system.operationcanceledexception.aspx).

A single task that is in a faulted state can reflect multiple exceptions. For example, the task might be the result of a call to [Task.WhenAll](http://msdn.microsoft.com/en-us/library/system.threading.tasks.task.whenall.aspx). When you await such a task, the await operation rethrows only one of the exceptions. However, you can't predict which of the exceptions is rethrown.

For examples of error handling in async methods, see [try-catch (C# Reference)](http://msdn.microsoft.com/en-us/library/0yd65esw.aspx).

[Example](javascript:void(0))

The following Windows Forms example illustrates the use of await in an async method, WaitAsynchronouslyAsync. Contrast the behavior of that method with the behavior of WaitSynchronously. Without an await operator applied to a task, WaitSynchronously runs synchronously despite the use of the async modifier in its definition and a call to [Thread.Sleep](http://msdn.microsoft.com/en-us/library/system.threading.thread.sleep.aspx) in its body.

C#

private async void button1\_Click(object sender, EventArgs e)

{

// Call the method that runs asynchronously.

string result = await WaitAsynchronouslyAsync();

// Call the method that runs synchronously.

//string result = await WaitSynchronously ();

// Display the result.

textBox1.Text += result;

}

// The following method runs asynchronously. The UI thread is not

// blocked during the delay. You can move or resize the Form1 window

// while Task.Delay is running.

public async Task<string> WaitAsynchronouslyAsync()

{

await Task.Delay(10000);

return "Finished";

}

// The following method runs synchronously, despite the use of async.

// You cannot move or resize the Form1 window while Thread.Sleep

// is running because the UI thread is blocked.

public async Task<string> WaitSynchronously()

{

// Add a using directive for System.Threading.

Thread.Sleep(10000);

return "Finished";

}