**await (C# Reference)**

**Visual Studio 2013**

[Other Versions](javascript:;)

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

* [Visual Studio 2012](http://msdn.microsoft.com/en-us/library/hh156528(d=printer,v=vs.110).aspx)

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.

|  |
| --- |
| **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);

// . . .

}

|  |
| --- |
| **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";

}