Task asynchronous programming model

05/22/2017 • 17 minutes to read • +4

In this article

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.

C# 5 introduced a simplified approach, async programming, that leverages asynchronous support in the .NET Framework 4.5 and higher, .NET Core, 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 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

Asynchrony is essential for activities that are potentially blocking, such as web access. Access to a web resource sometimes is slow or delayed. If such an activity is blocked in 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 .NET and the Windows Runtime contain methods that support async programming.

Application area	**.NET types with async methods**	**Windows Runtime types with async methods**
Web access	HttpClient	SyndicationClient
Working with files	StreamWriter, StreamReader, XmlReader	StorageFile
Working with images		MediaCapture, BitmapEncoder, BitmapDecoder
WCF programming	Synchronous and Asynchronous Operations	

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

The async and await keywords in C# are the heart of async programming. By using those two keywords, you can use resources in the .NET Framework, .NET Core, 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 the `async` keyword are referred to as *async methods*.

The following example shows an async method. Almost everything in the code should look completely familiar to you.

You can find a complete Windows Presentation Foundation (WPF) 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"]().

C# Copy

```
async Task<int> AccessTheWebAsync()
{
    // You need to add a reference to System.Net.Http to declare client.
    var 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("https://docs.microsoft.com/dotnet");

    // 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;
}
```

You can learn several practices from the preceding sample. Start with the method signature. It includes the `async` modifier. The return type is `Task<int>` (See "Return Types" section for more options). The method name ends in `Async`. In the body of the method, `GetStringAsync` returns a `Task<string>`. That means that when you `await` the task you'll get a `string` (`urlContents`). Before awaiting the task, you can do work that doesn't rely on the `string` from `GetStringAsync`.

Pay close attention to the `await` operator. It 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`.

The return statement specifies an integer result. Any methods that are awaiting `AccessTheWebAsync` retrieve the length value.

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# Copy

```csharp
string urlContents = await
client.GetStringAsync("https://docs.microsoft.com/dotnet");
```

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

- The method signature includes an `async` modifier.
- The name of an async method, by convention, ends with an "Async" suffix.
- The return type is one of the following types:
  - Task<TResult> if your method has a return statement in which the operand has type `TResult`.
  - Task if your method has no return statement or has a return statement with no operand.
  - `void` if you're writing an async event handler.
  - Any other type that has a `GetAwaiter` method (starting with C# 7.0).

  For more information, see the Return Types and Parameters section.
- 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

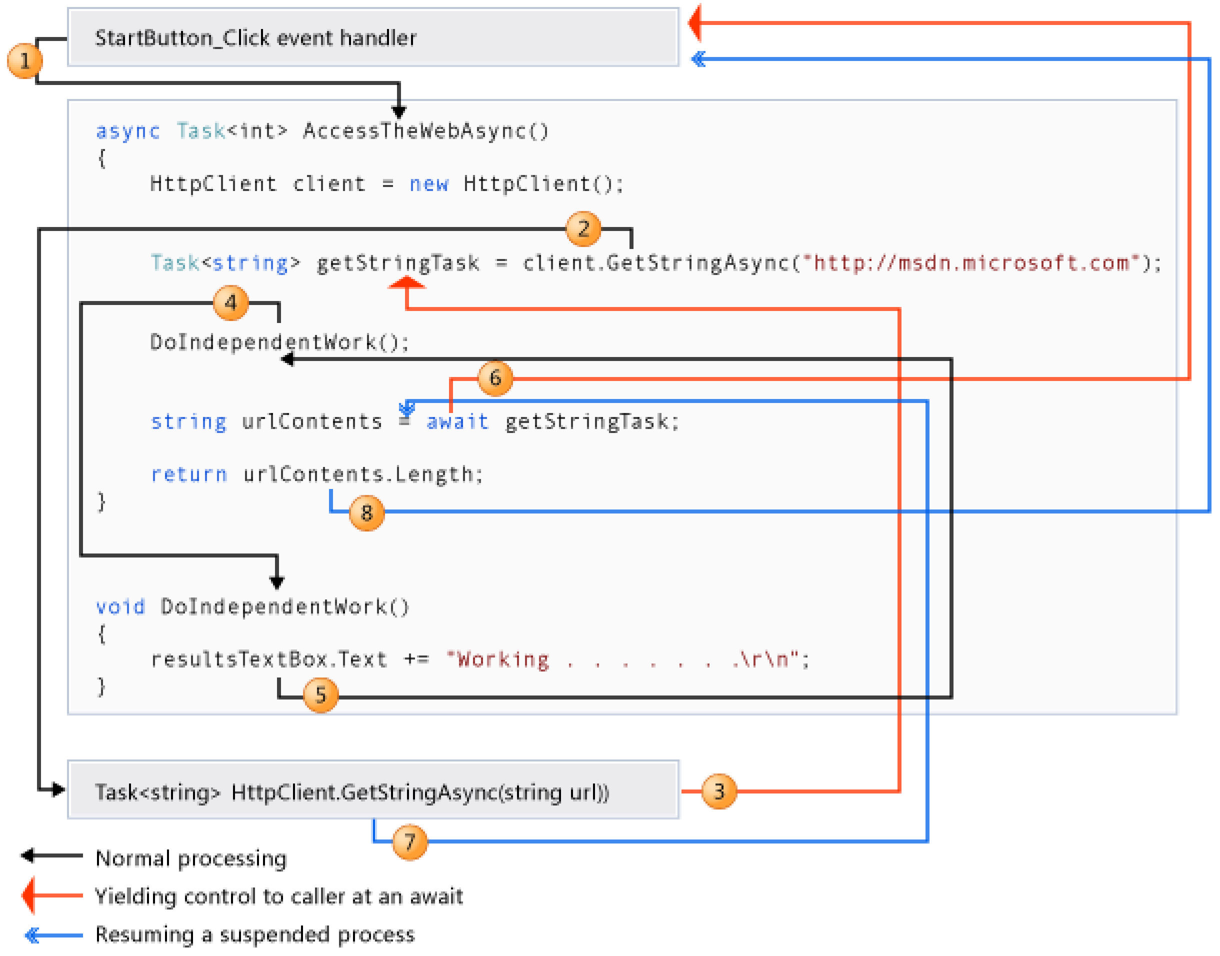
complete. In the meantime, the method is suspended, and control returns to the method's caller. The next section of this topic illustrates what happens at the suspension point.

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.

What happens in an async method

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



```
async Task<int> AccessTheWebAsync()
{
    HttpClient client = new HttpClient();

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

    DoIndependentWork();

    string urlContents = await getStringTask;

    return urlContents.Length;
}

void DoIndependentWork()
{
    resultsTextBox.Text += "Working . . . . . . .\r\n";
}
```

StartButton_Click event handler
Task<string> HttpClient.GetStringAsync(string url))
1
2
3
4
5
6
7
8
Normal processing
Yielding control to caller at an await
Resuming a suspended process

The numbers in the diagram correspond to the following steps, initiated when the user clicks the "start" button.

1. An event handler calls and awaits the `AccessTheWebAsync` async method.

2. `AccessTheWebAsync` creates an HttpClient instance and calls the GetStringAsync asynchronous method to download the contents of a website as a string.

3. Something happens in `GetStringAsync` that suspends its progress. Perhaps it must wait for a website to download or some other blocking activity. To avoid blocking resources, `GetStringAsync` yields control to its caller, `AccessTheWebAsync`.

   `GetStringAsync` returns a Task<TResult>, where `TResult` is a string, and `AccessTheWebAsync` assigns the task to the `getStringTask` variable. The task represents the ongoing process for the call to `GetStringAsync`, with a commitment to produce an actual string value when the work is complete.

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

represented by a call to the synchronous method `DoIndependentWork`.

5. `DoIndependentWork` is a synchronous method that does its work and returns to its caller.

6. `AccessTheWebAsync` has run out of work that it can do without a result from `getStringTask`. `AccessTheWebAsync` next wants to calculate and return the length of the downloaded string, but the method can't calculate that value until the method has the string.

   Therefore, `AccessTheWebAsync` uses an await operator to suspend its progress and to yield control to the method that called `AccessTheWebAsync`. `AccessTheWebAsync` returns a `Task<int>` to the caller. The task represents a promise to produce an integer result that's the length of the downloaded string.

   > **Note**
   >
   > If `GetStringAsync` (and therefore `getStringTask`) completes 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 `AccessTheWebAsync` 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`.

7. `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`.

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

event handler can resume. 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#).

API async methods

You might be wondering where to find methods such as `GetStringAsync` that support async programming. The .NET Framework 4.5 or higher and .NET Core contain many members that work with `async` and `await`. You can recognize them by the "Async" suffix that's appended to the member name, and by their return type of Task or Task<TResult>. For example, the `System.IO.Stream` class contains methods such as CopyToAsync, ReadAsync, and WriteAsync alongside the synchronous methods CopyTo, Read, and Write.

The Windows Runtime also contains many methods that you can use with `async` and `await` in Windows apps. For more information, see Threading and async programming for UWP development, and Asynchronous programming (Windows Store apps) and Quickstart: Calling asynchronous APIs in C# or Visual Basic if you use earlier versions of the Windows Runtime.

Threads

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

The `async` and `await` keywords don't cause additional threads to be created. 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 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 the BackgroundWorker class for I/O-bound operations because the code is simpler and you don't have to guard against race conditions. In combination with the Task.Run method, async programming is better than BackgroundWorker 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

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

- The marked async method can use await to designate suspension points. The `await` operator tells the compiler that the async method can't continue past that point until the awaited asynchronous process is complete. In the meantime, control returns to the caller of the async method.

  The suspension of an async method at an `await` expression doesn't constitute an exit from the method, 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` and `await` are contextual keywords. For more information and examples, see the following topics:

- async
- await

Return types and parameters

An async method typically returns a Task or a Task<TResult>. Inside an async method, an `await` operator is applied to a task that's returned from a call to another async method.

You specify Task<TResult> as the return type if the method contains a return statement that specifies an operand of type `TResult`.

You use Task as the return type if the method has no return statement or has a return statement that doesn't return an operand.

Starting with C# 7.0, you can also specify any other return type, provided that the type includes a `GetAwaiter` method. ValueTask<TResult> is an example of such a type. It is available in the System.Threading.Tasks.Extension NuGet package.

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

C# Copy

```
// Signature specifies Task<TResult>
async Task<int> GetTaskOfTResultAsync()
{
    int hours = 0;
    await Task.Delay(0);
    // Return statement specifies an integer result.
    return hours;
}

// Calls to GetTaskOfTResultAsync
Task<int> returnedTaskTResult = GetTaskOfTResultAsync();
int intResult = await returnedTaskTResult;
// or, in a single statement
int intResult = await GetTaskOfTResultAsync();

// Signature specifies Task
async Task GetTaskAsync()
{
    await Task.Delay(0);
    // The method has no return statement.
}

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

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 have a `void` return type. 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 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 in, ref or out parameters, but the method can call methods that have such parameters. Similarly, an async method can't return a value by reference, although it can call methods with ref return values.

For more information and examples, see Async Return Types (C#). For more information about how to catch exceptions in async methods, see try-catch.

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

- IAsyncOperation<TResult>, which corresponds to Task<TResult>
- IAsyncAction, which corresponds to Task
- IAsyncActionWithProgress<TProgress>
- IAsyncOperationWithProgress<TResult, TProgress>

Naming convention

By convention, methods that return commonly awaitable types (e.g. `Task`, `Task<T>`, `ValueTask`, `ValueTask<T>`) should have names that end with "Async". Methods that start an asynchronous operation but do not return an awaitable type should not have names that end with "Async", but may start with "Begin", "Start", or some other verb to suggest this method does not return or throw the result of the operation.

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 and samples (Visual Studio)

Title	Description	Sample

Title	Description	Sample
Walkthrough: Accessing the Web by Using async and await (C#)	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
How to extend the async walkthrough by using Task.WhenAll (C#)	Adds Task.WhenAll 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#)	Demonstrates how to start several tasks at the same time.	Async Sample: Make Multiple Web Requests in Parallel
Async Return Types (C#)	Illustrates the types that async methods can return and explains when each type is appropriate.	
Control Flow in Async Programs (C#)	Traces in detail the flow of control through a succession of await expressions in an asynchronous program.	Async Sample: Control Flow in Async Programs
Fine-Tuning Your Async Application (C#)	Shows how to add the following functionality to your async solution: - Cancel an Async Task or a List of Tasks (C#) - Cancel Async Tasks after a Period of Time (C#) - Cancel Remaining Async Tasks after One Is Complete (C#) - Start Multiple Async Tasks and Process Them As They Complete (C#)	Async Sample: Fine Tuning Your Application
Handling Reentrancy in Async Apps (C#)	Shows how to handle cases in which an active asynchronous operation is restarted while it's running.	

Title	Description	Sample
WhenAny: Bridging between the .NET Framework and the Windows Runtime	Shows how to bridge between Task types in the .NET Framework and IAsyncOperations in the Windows Runtime so that you can use WhenAny with a Windows Runtime method.	Async Sample: Bridging between .NET and Windows Runtime (AsTask and WhenAny)
Async Cancellation: Bridging between the .NET Framework and the Windows Runtime	Shows how to bridge between Task types in the .NET Framework and IAsyncOperations in the Windows Runtime so that you can use CancellationTokenSource with a Windows Runtime method.	Async Sample: Bridging between .NET and Windows Runtime (AsTask & Cancellation)
Using Async for File Access (C#)	Lists and demonstrates the benefits of using async and await to access files.	
Task-based Asynchronous Pattern (TAP)	Describes a new pattern for asynchrony in the .NET Framework. The pattern is based on the Task and Task<TResult> types.	
Async Videos on Channel 9	Provides links to a variety of videos about async programming.	

Complete example

The following code is the *MainWindow.xaml.cs* file from the WPF application that this article discusses. You can download the sample from Async Sample: Example from "Asynchronous Programming with Async and Await".

C# Copy

```csharp
using System;
using System.Threading.Tasks;
using System.Windows;

// 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 +=
                $"\r\nLength of the downloaded string: {contentLength}.\r\n";
        }

        // 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.
            var 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("https://docs.microsoft.com/dotnet");

            // 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 += "\r\nWorking . . . . . . .\r\n";
        }
    }
}

// Sample Output:

// Working . . . . . . .

// Length of the downloaded string: 41564.
```

See also

- async
- await
- Asynchronous programming
- Async overview

Is this page helpful?

Yes No