Contents

The Task asynchronous programming model in C#

6/21/2019 • 10 minutes to read • Edit Online

The Task asynchronous programming model (TAP) provides an abstraction over asynchronous code. You write code as a sequence of statements, just like always. You can read that code as though each statement completes before the next begins. The compiler performs a number of transformations because some of those statements may start work and return a Task that represents the ongoing work.

That's the goal of this syntax: enable code that reads like a sequence of statements, but executes in a much more complicated order based on external resource allocation and when tasks complete. It's analogous to how people give instructions for processes that include asynchronous tasks. Throughout this article, you'll use an example of instructions for making a breakfast to see how the `async` and `await` keywords make it easier to reason about code that includes a series of asynchronous instructions. You'd write the instructions something like the following list to explain how to make a breakfast:

1. Pour a cup of coffee.
2. Heat up a pan, then fry two eggs.
3. Fry three slices of bacon.
4. Toast two pieces of bread.
5. Add butter and jam to the toast.
6. Pour a glass of orange juice.

If you have experience with cooking, you'd execute those instructions **asynchronously**. You'd start warming the pan for eggs, then start the bacon. You'd put the bread in the toaster, then start the eggs. At each step of the process, you'd start a task, then turn your attention to tasks that are ready for your attention.

Cooking breakfast is a good example of asynchronous work that isn't parallel. One person (or thread) can handle all these tasks. Continuing the breakfast analogy, one person can make breakfast asynchronously by starting the next task before the first completes. The cooking progresses whether or not someone is watching it. As soon as you start warming the pan for the eggs, you can begin frying the bacon. Once the bacon starts, you can put the bread into the toaster.

For a parallel algorithm, you'd need multiple cooks (or threads). One would make the eggs, one the bacon, and so on. Each one would be focused on just that one task. Each cook (or thread) would be blocked synchronously waiting for bacon to be ready to flip, or the toast to pop.

Now, consider those same instructions written as C# statements:

```
static void Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");
    Egg eggs = FryEggs(2);
    Console.WriteLine("eggs are ready");
    Bacon bacon = FryBacon(3);
    Console.WriteLine("bacon is ready");
    Toast toast = ToastBread(2);
    ApplyButter(toast);
    ApplyJam(toast);
    Console.WriteLine("toast is ready");
    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");

    Console.WriteLine("Breakfast is ready!");
}
```

Computers don't interpret those instructions the same way people do. The computer will block on each statement until the work is complete before moving on to the next statement. That creates an unsatisfying breakfast. The later tasks wouldn't be started until the earlier tasks had completed. It would take much longer to create the breakfast, and some items would have gotten cold before being served.

If you want the computer to execute the above instructions asynchronously, you must write asynchronous code.

These concerns are important for the programs you write today. When you write client programs, you want the UI to be responsive to user input. Your application shouldn't make a phone appear frozen while it's downloading data from the web. When you write server programs, you don't want threads blocked. Those threads could be serving other requests. Using synchronous code when asynchronous alternatives exist hurts your ability to scale out less expensively. You pay for those blocked threads.

Successful modern applications require asynchronous code. Without language support, writing asynchronous code required callbacks, completion events, or other means that obscured the original intent of the code. The advantage of the synchronous code is that it's easy to understand. The step-by-step actions make it easy to scan and understand. Traditional asynchronous models forced you to focus on the asynchronous nature of the code, not on the fundamental actions of the code.

Don't block, await instead

The preceding code demonstrates a bad practice: constructing synchronous code to perform asynchronous operations. As written, this code blocks the thread executing it from doing any other work. It won't be interrupted while any of the tasks are in progress. It would be as though you stared at the toaster after putting the bread in. You'd ignore anyone talking to you until the toast popped.

Let's start by updating this code so that the thread doesn't block while tasks are running. The `await` keyword provides a non-blocking way to start a task, then continue execution when that task completes. A simple asynchronous version of the make a breakfast code would look like the following snippet:

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");
    Egg eggs = await FryEggs(2);
    Console.WriteLine("eggs are ready");
    Bacon bacon = await FryBacon(3);
    Console.WriteLine("bacon is ready");
    Toast toast = await ToastBread(2);
    ApplyButter(toast);
    ApplyJam(toast);
    Console.WriteLine("toast is ready");
    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");

    Console.WriteLine("Breakfast is ready!");
}
```

This code doesn't block while the eggs or the bacon are cooking. This code won't start any other tasks though. You'd still put the toast in the toaster and stare at it until it pops. But at least, you'd respond to anyone that wanted your attention. In a restaurant where multiple orders are placed, the cook could start another breakfast while the first is cooking.

Now, the thread working on the breakfast isn't blocked while awaiting any started task that hasn't yet finished. For some applications, this change is all that's needed. A GUI application still responds to the user with just this change. However, for this scenario, you want more. You don't want each of the component tasks to be executed sequentially. It's better to start each of the component tasks before awaiting the previous task's completion.

Start tasks concurrently

In many scenarios, you want to start several independent tasks immediately. Then, as each task finishes, you can continue other work that's ready. In the breakfast analogy, that's how you get breakfast done more quickly. You also get everything done close to the same time. You'll get a hot breakfast.

The System.Threading.Tasks.Task and related types are classes you can use to reason about tasks that are in progress. That enables you to write code that more closely resembles the way you'd actually create breakfast. You'd start cooking the eggs, bacon, and toast at the same time. As each requires action, you'd turn your attention to that task, take care of the next action, then await for something else that requires your attention.

You start a task and hold on to the Task object that represents the work. You'll `await` each task before working with its result.

Let's make these changes to the breakfast code. The first step is to store the tasks for operations when they start, rather than awaiting them:

```
Coffee cup = PourCoffee();
Console.WriteLine("coffee is ready");
Task<Egg> eggTask = FryEggs(2);
Egg eggs = await eggTask;
Console.WriteLine("eggs are ready");
Task<Bacon> baconTask = FryBacon(3);
Bacon bacon = await baconTask;
Console.WriteLine("bacon is ready");
Task<Toast> toastTask = ToastBread(2);
Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("toast is ready");
Juice oj = PourOJ();
Console.WriteLine("oj is ready");

Console.WriteLine("Breakfast is ready!");
```

Next, you can move the `await` statements for the bacon and eggs to the end of the method, before serving breakfast:

```
Coffee cup = PourCoffee();
Console.WriteLine("coffee is ready");
Task<Egg> eggTask = FryEggs(2);
Task<Bacon> baconTask = FryBacon(3);
Task<Toast> toastTask = ToastBread(2);
Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("toast is ready");
Juice oj = PourOJ();
Console.WriteLine("oj is ready");

Egg eggs = await eggTask;
Console.WriteLine("eggs are ready");
Bacon bacon = await baconTask;
Console.WriteLine("bacon is ready");

Console.WriteLine("Breakfast is ready!");
```

The preceding code works better. You start all the asynchronous tasks at once. You await each task only when you need the results. The preceding code may be similar to code in a web application that makes requests of different microservices, then combines the results into a single page. You'll make all the requests immediately, then `await` all those tasks and compose the web page.

Composition with tasks

You have everything ready for breakfast at the same time except the toast. Making the toast is the composition of an asynchronous operation (toasting the bread), and synchronous operations (adding the butter and the jam). Updating this code illustrates an important concept:

IMPORTANT

The composition of an asynchronous operation followed by synchronous work is an asynchronous operation. Stated another way, if any portion of an operation is asynchronous, the entire operation is asynchronous.

The preceding code showed you that you can use Task or Task<TResult> objects to hold running tasks. You `await` each task before using its result. The next step is to create methods that represent the combination of other work. Before serving breakfast, you want to await the task that represents toasting the bread before adding butter and jam. You can represent that work with the following code:

```
async Task<Toast> makeToastWithButterAndJamAsync(int number)
{
    var plainToast = await ToastBreadAsync(number);
    ApplyButter(plainToast);
    ApplyJam(plainToast);
    return plainToast;
}
```

The preceding method has the `async` modifier in its signature. That signals to the compiler that this method contains an `await` statement; it contains asynchronous operations. This method represents the task that toasts the bread, then adds butter and jam. This method returns a Task<TResult> that represents the composition of those three operations. The main block of code now becomes:

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");
    var eggsTask = FryEggsAsync(2);
    var baconTask = FryBaconAsync(3);
    var toastTask = makeToastWithButterAndJamAsync(2);

    var eggs = await eggsTask;
    Console.WriteLine("eggs are ready");
    var bacon = await baconTask;
    Console.WriteLine("bacon is ready");
    var toast = await toastTask;
    Console.WriteLine("toast is ready");
    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");

    Console.WriteLine("Breakfast is ready!");

    async Task<Toast> makeToastWithButterAndJamAsync(int number)
    {
        var plainToast = await ToastBreadAsync(number);
        ApplyButter(plainToast);
        ApplyJam(plainToast);
        return plainToast;
    }
}
```

The previous change illustrated an important technique for working with asynchronous code. You compose tasks by separating the operations into a new method that returns a task. You can choose when to await that task. You can start other tasks concurrently.

Await tasks efficiently

The series of `await` statements at the end of the preceding code can be improved by using methods of the `Task` class. One of those APIs is WhenAll, which returns a Task that completes when all the tasks in its argument list have completed, as shown in the following code:

```
await Task.WhenAll(eggTask, baconTask, toastTask);
Console.WriteLine("eggs are ready");
Console.WriteLine("bacon is ready");
Console.WriteLine("toast is ready");
Console.WriteLine("Breakfast is ready!");
```

Another option is to use WhenAny, which returns a `Task<Task>` that completes when any of its arguments completes. You can await the returned task, knowing that it has already finished. The following code shows how you could use WhenAny to await the first task to finish and then process its result. After processing the result from the completed task, you remove that completed task from the list of tasks passed to `WhenAny`.

```
var allTasks = new List<Task>{eggsTask, baconTask, toastTask};
while (allTasks.Any())
{
    Task finished = await Task.WhenAny(allTasks);
    if (finished == eggsTask)
    {
        Console.WriteLine("eggs are ready");
    }
    else if (finished == baconTask)
    {
        Console.WriteLine("bacon is ready");
    }
    else if (finished == toastTask)
    {
        Console.WriteLine("toast is ready");
    }
    allTasks.Remove(finished);
}
Console.WriteLine("Breakfast is ready!");
```

After all those changes, the final version of `Main` looks like the following code:

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");
    var eggsTask = FryEggsAsync(2);
    var baconTask = FryBaconAsync(3);
    var toastTask = makeToastWithButterAndJamAsync(2);

    var allTasks = new List<Task>{eggsTask, baconTask, toastTask};
    while (allTasks.Any())
    {
        Task finished = await Task.WhenAny(allTasks);
        if (finished == eggsTask)
        {
            Console.WriteLine("eggs are ready");
        }
        else if (finished == baconTask)
        {
            Console.WriteLine("bacon is ready");
        }
        else if (finished == toastTask)
        {
            Console.WriteLine("toast is ready");
        }
        allTasks.Remove(finished);
    }
    Console.WriteLine("Breakfast is ready!");

    async Task<Toast> makeToastWithButterAndJamAsync(int number)
    {
        var plainToast = await ToastBreadAsync(number);
        ApplyButter(plainToast);
        ApplyJam(plainToast);
        return plainToast;
    }
}
```

This final code is asynchronous. It more accurately reflects how a person would cook a breakfast. Compare the preceding code with the first code sample in this article. The core actions are still clear from reading the code. You can read this code the same way you'd read those instructions for making a breakfast at the beginning of this article. The language features for `async` and `await` provide the translation every person makes to follow those written instructions: start tasks as you can and don't block waiting for tasks to complete.

Task asynchronous programming model

6/25/2019 • 16 minutes to read • Edit Online

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".

```
async Task<int> AccessTheWebAsync()
{
    // You need to add a reference to System.Net.Http to declare client.
    using (HttpClient client = new HttpClient())
    {
        Task<string> getStringTask = client.GetStringAsync("https://docs.microsoft.com");

        DoIndependentWork();

        string urlContents = await getStringTask;

        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.

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

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.

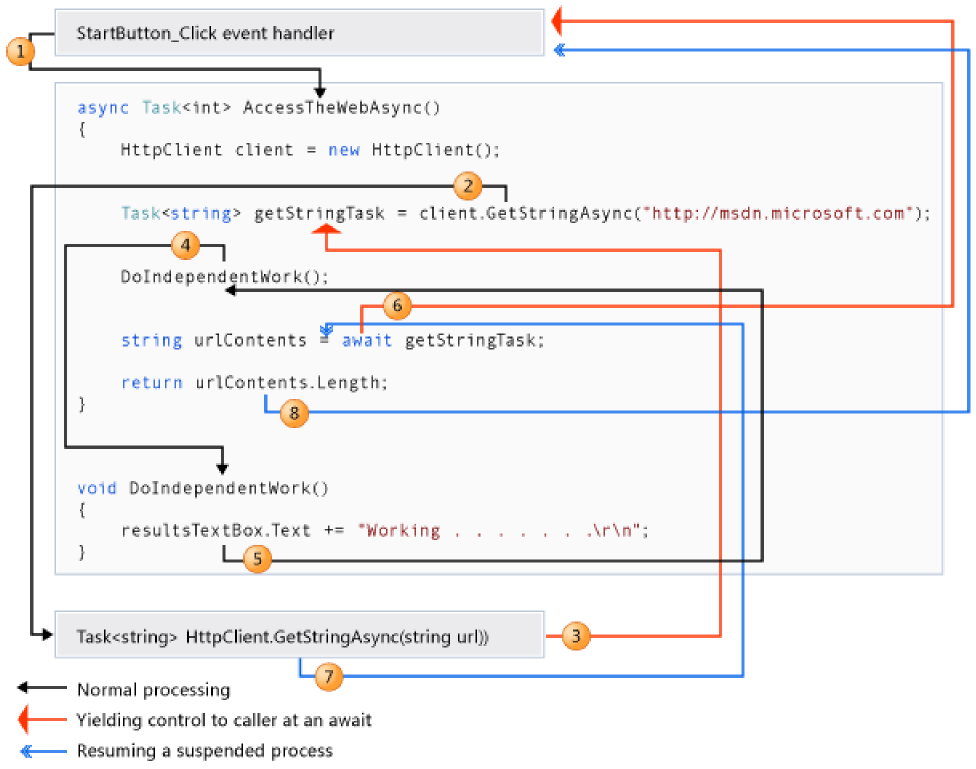
- The method usually includes at least one `await` expression, which marks a point where the method can't continue until the awaited asynchronous operation is complete. In the meantime, the method is suspended, and control returns to the method's caller. The next section of this topic illustrates what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


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

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.

```
// 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
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 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".

```
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 +=
                $"\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.
            using (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("https://docs.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: 25035.
```

See also

- async
- await
- Asynchronous programming
- Async overview

Walkthrough: Accessing the Web by Using async and await (C#)

5/23/2019 • 16 minutes to read • Edit Online

You can write asynchronous programs more easily and intuitively by using async/await features. 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#).

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).

NOTE

To run the examples, you must have Visual Studio 2012 or newer and the .NET Framework 4.5 or newer installed on your computer.

Create a WPF application

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

   The **New Project** dialog box opens.
3. In the **Installed Templates** pane, choose Visual C#, and then choose **WPF Application** from the list of project types.
4. In the **Name** text box, enter `AsyncExampleWPF`, and then choose the **OK** button.

   The new project appears in **Solution Explorer**.

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.

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.
3. At the top of the dialog box, verify that your project is targeting the .NET Framework 4.5 or higher.
4. In the **Assemblies** category, choose **Framework** if it isn't already chosen.
5. In the list of names, select the **System.Net.Http** check box.
6. Choose the **OK** button to close the dialog box.

Add necessary using directives

1. In **Solution Explorer**, open the shortcut menu for MainWindow.xaml.cs, and then choose **View Code**.
2. Add the following `using` directives at the top of the code file if they're not already present.

```
using System.Net.Http;
using System.Net;
using System.IO;
```

Create a synchronous app

1. In the design window, MainWindow.xaml, double-click the **Start** button to create the `startButton_Click` event handler in MainWindow.xaml.cs.
2. In MainWindow.xaml.cs, copy the following code into the body of `startButton_Click`:

```
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`.
3. 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.cs:

```
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 += $"\r\n\r\nTotal bytes returned:  {total}\r\n";
}

private List<string> SetUpURLList()
{
    var urls = new List<string>
    {
        "https://msdn.microsoft.com/library/windows/apps/br211380.aspx",
        "https://msdn.microsoft.com",
        "https://msdn.microsoft.com/library/hh290136.aspx",
        "https://msdn.microsoft.com/library/ee256749.aspx",
        "https://msdn.microsoft.com/library/hh290138.aspx",
        "https://msdn.microsoft.com/library/hh290140.aspx",
        "https://msdn.microsoft.com/library/dd470362.aspx",
        "https://msdn.microsoft.com/library/aa578028.aspx",
        "https://msdn.microsoft.com/library/ms404677.aspx",
        "https://msdn.microsoft.com/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 "https://".
    var displayURL = url.Replace("https://", "");
    resultsTextBox.Text += $"\n{displayURL,-58} {bytes,8}";
}
```

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/library/hh290136.aspx               225793
msdn.microsoft.com/library/ee256749.aspx               143577
msdn.microsoft.com/library/hh290138.aspx               237372
msdn.microsoft.com/library/hh290140.aspx               128279
msdn.microsoft.com/library/dd470362.aspx               157649
msdn.microsoft.com/library/aa578028.aspx               204457
msdn.microsoft.com/library/ms404677.aspx               176405
msdn.microsoft.com/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.

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 method GetResponse and to the Stream method CopyTo 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.

   > **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 method.

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

2. `GetResponseAsync` returns a Task<TResult>. In this case, the *task return variable*, `TResult`, has type WebResponse. 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 operator to the call to `GetResponseAsync`, as the following code shows.

   ```
   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.

   ```
   //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 How to: Extend the async Walkthrough by Using Task.WhenAll (C#).

3. Because you added the `await` operator in the previous step, a compiler error occurs. The operator can be used only in methods that are marked with the async 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.

   - 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. 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.

     ```
     await responseStream.CopyToAsync(content);
     ```

     The previous statement abbreviates the following two lines of code.

     ```
     // 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;
     ```

4. All that remains to be done in `GetURLContents` is to adjust the method signature. You can use the `await` operator only in methods that are marked with the async modifier. Add the modifier to mark the method as an *async method*, as the following code shows.

```
private async byte[] GetURLContents(string url)
```

5. The return type of an async method can only be Task, Task<TResult>, or `void` in C#. Typically, a return type of `void` is used only in an async event handler, where `void` is required. In other cases, you use `Task(T)` if the completed method has a return 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#).

   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<byte[]>`.
   - By convention, asynchronous methods have names that end in "Async," so rename the method `GetURLContentsAsync`.

   The following code shows these changes.

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

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

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` to the task that `GetURLContentsAsync` returns to obtain the byte array value.

   The following code shows these changes.

   ```
   byte[] urlContents = await GetURLContentsAsync(url);
   ```

   The previous assignment abbreviates the following two lines of code.

   ```
   // GetURLContentsAsync returns a Task<T>. At completion, the task
   // produces a byte array.
   //Task<byte[]> getContentsTask = GetURLContentsAsync(url);
   //byte[] urlContents = await getContentsTask;
   ```

2. Make the following changes in the method's signature:

   - Mark the method with the `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` statement.) However, the method must return a `Task` to be awaitable. Therefore, change the return type of the method from `void` to `Task`.

The following code shows these changes.

```
private async Task SumPageSizesAsync()
```

The conversion of `SumPageSizes` to `SumPageSizesAsync` is complete.

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.

```
// One-step async call.
await SumPageSizesAsync();

// Two-step async call.
//Task sumTask = SumPageSizesAsync();
//await sumTask;
```

3. To prevent accidentally reentering the operation, add the following statement at the top of `startButton_Click` to disable the **Start** button.

```
// Disable the button until the operation is complete.
startButton.IsEnabled = false;
```

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

```
// 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#).

4. Finally, add the `async` modifier to the declaration so that the event handler can await `SumPagSizesAsync`.

```
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`.

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

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).

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 method GetByteArrayAsync(String), 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.

   ```
   // Declare an HttpClient object and increase the buffer size. The
   // default buffer size is 65,536.
   HttpClient client =
       new HttpClient() { MaxResponseContentBufferSize = 1000000 };
   ```

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

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

3. Remove or comment out the `GetURLContentsAsync` method that you wrote.

4. 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 code

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.

```
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 +=
                $"\r\n\r\nTotal bytes returned:  {total}\r\n";
        }

        private List<string> SetUpURLList()
        {
            List<string> urls = new List<string>
            {
                "https://msdn.microsoft.com/library/windows/apps/br211380.aspx",
                "https://msdn.microsoft.com",
                "https://msdn.microsoft.com/library/hh290136.aspx",
                "https://msdn.microsoft.com/library/ee256749.aspx",
                "https://msdn.microsoft.com/library/hh290138.aspx",
                "https://msdn.microsoft.com/library/hh290140.aspx",
                "https://msdn.microsoft.com/library/dd470362.aspx",
                "https://msdn.microsoft.com/library/aa578028.aspx",
                "https://msdn.microsoft.com/library/ms404677.aspx",
                "https://msdn.microsoft.com/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 "https://".
            var displayURL = url.Replace("https://", "");
            resultsTextBox.Text += $"\n{displayURL,-58} {bytes,8}";
        }
    }
}
```

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

```
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 +=
                $"\r\n\r\nTotal bytes returned:  {total}\r\n";
        }

        private List<string> SetUpURLList()
        {
            List<string> urls = new List<string>
            {
                "https://msdn.microsoft.com/library/windows/apps/br211380.aspx",
                "https://msdn.microsoft.com",
                "https://msdn.microsoft.com/library/hh290136.aspx",
                "https://msdn.microsoft.com/library/ee256749.aspx",
                "https://msdn.microsoft.com/library/hh290138.aspx",
                "https://msdn.microsoft.com/library/hh290140.aspx",
                "https://msdn.microsoft.com/library/dd470362.aspx",
                "https://msdn.microsoft.com/library/aa578028.aspx",
                "https://msdn.microsoft.com/library/ms404677.aspx",
                "https://msdn.microsoft.com/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 "https://".
            var displayURL = url.Replace("https://", "");
            resultsTextBox.Text += $"\n{displayURL,-58} {bytes,8}";
        }
    }
}
```

See also

- Async Sample: Accessing the Web Walkthrough (C# and Visual Basic)
- async
- await
- Asynchronous Programming with async and await (C#)
- Async Return Types (C#)
- Task-based Asynchronous Programming (TAP)
- How to: Extend the async Walkthrough by Using Task.WhenAll (C#)
- How to: Make Multiple Web Requests in Parallel by Using async and await (C#)

How to: Extend the async Walkthrough by Using Task.WhenAll (C#)

4/28/2019 • 10 minutes to read • Edit Online

You can improve the performance of the async solution in Walkthrough: Accessing the Web by Using async and await (C#) by using the Task.WhenAll method. This method asynchronously awaits multiple asynchronous operations, which are represented as a collection of tasks.

You might have noticed in the walkthrough that the websites download at different rates. Sometimes one of the websites is very slow, which delays all the remaining downloads. When you run the asynchronous solutions that you build in the walkthrough, you can end the program easily if you don't want to wait, but a better option would be to start all the downloads at the same time and let faster downloads continue without waiting for the one that's delayed.

You apply the `Task.WhenAll` method to a collection of tasks. The application of `WhenAll` returns a single task that isn't complete until every task in the collection is completed. The tasks appear to run in parallel, but no additional threads are created. The tasks can complete in any order.

IMPORTANT

The following procedures describe extensions to the async applications that are developed in Walkthrough: Accessing the Web by Using async and await (C#). You can develop the applications by either completing the walkthrough or downloading the code from Developer Code Samples.

To run the example, you must have Visual Studio 2012 or later installed on your computer.

To add Task.WhenAll to your GetURLContentsAsync solution

1. Add the `ProcessURLAsync` method to the first application that's developed in Walkthrough: Accessing the Web by Using async and await (C#).

   - If you downloaded the code from Developer Code Samples, open the AsyncWalkthrough project, and then add `ProcessURLAsync` to the MainWindow.xaml.cs file.
   - If you developed the code by completing the walkthrough, add `ProcessURLAsync` to the application that includes the `GetURLContentsAsync` method. The MainWindow.xaml.cs file for this application is the first example in the "Complete Code Examples from the Walkthrough" section.

   The `ProcessURLAsync` method consolidates the actions in the body of the `foreach` loop in `SumPageSizesAsync` in the original walkthrough. The method asynchronously downloads the contents of a specified website as a byte array, and then displays and returns the length of the byte array.

```
private async Task<int> ProcessURLAsync(string url)
{
    var byteArray = await GetURLContentsAsync(url);
    DisplayResults(url, byteArray);
    return byteArray.Length;
}
```

2. Comment out or delete the `foreach` loop in `SumPageSizesAsync`, as the following code shows.

```
//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;
//}
```

3. Create a collection of tasks. The following code defines a query that, when executed by the ToArray method, creates a collection of tasks that download the contents of each website. The tasks are started when the query is evaluated.

   Add the following code to method `SumPageSizesAsync` after the declaration of `urlList`.

```
// Create a query.
IEnumerable<Task<int>> downloadTasksQuery =
    from url in urlList select ProcessURLAsync(url);

// Use ToArray to execute the query and start the download tasks.
Task<int>[] downloadTasks = downloadTasksQuery.ToArray();
```

4. Apply `Task.WhenAll` to the collection of tasks, `downloadTasks`. `Task.WhenAll` returns a single task that finishes when all the tasks in the collection of tasks have completed.

   In the following example, the `await` expression awaits the completion of the single task that `WhenAll` returns. The expression evaluates to an array of integers, where each integer is the length of a downloaded website. Add the following code to `SumPageSizesAsync`, just after the code that you added in the previous step.

```
// Await the completion of all the running tasks.
int[] lengths = await Task.WhenAll(downloadTasks);

//// The previous line is equivalent to the following two statements.
//Task<int[]> whenAllTask = Task.WhenAll(downloadTasks);
//int[] lengths = await whenAllTask;
```

5. Finally, use the Sum method to calculate the sum of the lengths of all the websites. Add the following line to `SumPageSizesAsync`.

```
int total = lengths.Sum();
```

To add Task.WhenAll to the HttpClient.GetByteArrayAsync solution

1. Add the following version of `ProcessURLAsync` to the second application that's developed in Walkthrough: Accessing the Web by Using async and await (C#).

   - If you downloaded the code from Developer Code Samples, open the AsyncWalkthrough_HttpClient project, and then add `ProcessURLAsync` to the MainWindow.xaml.cs file.

- If you developed the code by completing the walkthrough, add `ProcessURLAsync` to the application that uses the `HttpClient.GetByteArrayAsync` method. The MainWindow.xaml.cs file for this application is the second example in the "Complete Code Examples from the Walkthrough" section.

The `ProcessURLAsync` method consolidates the actions in the body of the `foreach` loop in `SumPageSizesAsync` in the original walkthrough. The method asynchronously downloads the contents of a specified website as a byte array, and then displays and returns the length of the byte array.

The only difference from the `ProcessURLAsync` method in the previous procedure is the use of the HttpClient instance, `client`.

```
async Task<int> ProcessURLAsync(string url, HttpClient client)
{
    byte[] byteArray = await client.GetByteArrayAsync(url);
    DisplayResults(url, byteArray);
    return byteArray.Length;
}
```

2. Comment out or delete the `For Each` or `foreach` loop in `SumPageSizesAsync`, as the following code shows.

```
//var total = 0;
//foreach (var url in urlList)
//{
//    // GetByteArrayAsync returns a Task<T>. At completion, the task
//    // produces a byte array.
//    byte[] urlContent = await client.GetByteArrayAsync(url);

//    // The previous line abbreviates the following two assignment
//    // statements.
//    Task<byte[]> getContentTask = client.GetByteArrayAsync(url);
//    byte[] urlContent = await getContentTask;

//    DisplayResults(url, urlContent);

//    // Update the total.
//    total += urlContent.Length;
//}
```

3. Define a query that, when executed by the ToArray method, creates a collection of tasks that download the contents of each website. The tasks are started when the query is evaluated.

   Add the following code to method `SumPageSizesAsync` after the declaration of `client` and `urlList`.

```
// Create a query.
IEnumerable<Task<int>> downloadTasksQuery =
    from url in urlList select ProcessURLAsync(url, client);

// Use ToArray to execute the query and start the download tasks.
Task<int>[] downloadTasks = downloadTasksQuery.ToArray();
```

4. Next, apply `Task.WhenAll` to the collection of tasks, `downloadTasks`. `Task.WhenAll` returns a single task that finishes when all the tasks in the collection of tasks have completed.

   In the following example, the `await` expression awaits the completion of the single task that `WhenAll` returns. When complete, the `await` expression evaluates to an array of integers, where each integer is the length of a downloaded website. Add the following code to `SumPageSizesAsync`, just after the code that you added in the previous step.

```
// Await the completion of all the running tasks.
int[] lengths = await Task.WhenAll(downloadTasks);

//// The previous line is equivalent to the following two statements.
//Task<int[]> whenAllTask = Task.WhenAll(downloadTasks);
//int[] lengths = await whenAllTask;
```

5. Finally, use the Sum method to get the sum of the lengths of all the websites. Add the following line to `SumPageSizesAsync`.

```
int total = lengths.Sum();
```

To test the Task.WhenAll solutions

- For either solution, choose the F5 key to run the program, and then choose the **Start** button. The output should resemble the output from the async solutions in Walkthrough: Accessing the Web by Using async and await (C#). However, notice that the websites appear in a different order each time.

Example

The following code shows the extensions to the project that uses the `GetURLContentsAsync` method to download content from the web.

```
// 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_WhenAll
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private async void startButton_Click(object sender, RoutedEventArgs e)
        {
            resultsTextBox.Clear();

            // Two-step async call.
            Task sumTask = SumPageSizesAsync();
            await sumTask;

            // One-step async call.
            //await SumPageSizesAsync();

            resultsTextBox.Text += "\r\nControl returned to startButton_Click.\r\n";
        }

        private async Task SumPageSizesAsync()
        {
            // Make a list of web addresses.
            List<string> urlList = SetUpURLList();

            // Create a query.
            IEnumerable<Task<int>> downloadTasksQuery =
                from url in urlList select ProcessURLAsync(url);

            // Use ToArray to execute the query and start the download tasks.
            Task<int>[] downloadTasks = downloadTasksQuery.ToArray();
            // You can do other work here before awaiting.

            // Await the completion of all the running tasks.
            int[] lengths = await Task.WhenAll(downloadTasks);

            //// The previous line is equivalent to the following two statements.
            //Task<int[]> whenAllTask = Task.WhenAll(downloadTasks);
            //int[] lengths = await whenAllTask;

            int total = lengths.Sum();

            //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 +=
                $"\r\n\r\nTotal bytes returned:  {total}\r\n";
        }

        private List<string> SetUpURLList()
        {
            List<string> urls = new List<string>
            {
                "https://msdn.microsoft.com",
                "https://msdn.microsoft.com/library/windows/apps/br211380.aspx",
                "https://msdn.microsoft.com/library/hh290136.aspx",
                "https://msdn.microsoft.com/library/ee256749.aspx",
                "https://msdn.microsoft.com/library/hh290138.aspx",
                "https://msdn.microsoft.com/library/hh290140.aspx",
                "https://msdn.microsoft.com/library/dd470362.aspx",
                "https://msdn.microsoft.com/library/aa578028.aspx",
                "https://msdn.microsoft.com/library/ms404677.aspx",
                "https://msdn.microsoft.com/library/ff730837.aspx"
            };
            return urls;
        }

        // The actions from the foreach loop are moved to this async method.
        private async Task<int> ProcessURLAsync(string url)
        {
            var byteArray = await GetURLContentsAsync(url);
            DisplayResults(url, byteArray);
            return byteArray.Length;
        }

        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())
            {
                // Get the data stream that is associated with the specified url.
                using (Stream responseStream = response.GetResponseStream())
                {
                    await responseStream.CopyToAsync(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 "https://".
            var displayURL = url.Replace("https://", "");
            resultsTextBox.Text += $"\n{displayURL,-58} {bytes,8}";
        }
    }
}
```

Example

The following code shows the extensions to the project that uses method `HttpClient.GetByteArrayAsync` to download content from the web.

```
// 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_HttpClient_WhenAll
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private async void startButton_Click(object sender, RoutedEventArgs e)
        {
            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";
        }

        private async Task SumPageSizesAsync()
        {
            // Make a list of web addresses.
            List<string> urlList = SetUpURLList();
            // Declare an HttpClient object and increase the buffer size. The
            // default buffer size is 65,536.
            HttpClient client = new HttpClient() { MaxResponseContentBufferSize = 1000000 };

            // Create a query.
            IEnumerable<Task<int>> downloadTasksQuery =
                from url in urlList select ProcessURLAsync(url, client);

            // Use ToArray to execute the query and start the download tasks.
            Task<int>[] downloadTasks = downloadTasksQuery.ToArray();

            // You can do other work here before awaiting.

            // Await the completion of all the running tasks.
            int[] lengths = await Task.WhenAll(downloadTasks);

            //// The previous line is equivalent to the following two statements.
            //Task<int[]> whenAllTask = Task.WhenAll(downloadTasks);
            //int[] lengths = await whenAllTask;

            int total = lengths.Sum();

            //var total = 0;
            //foreach (var url in urlList)
            //{
            //    // GetByteArrayAsync returns a Task<T>. At completion, the task
            //    // produces a byte array.
            //    byte[] urlContent = await client.GetByteArrayAsync(url);

            //    // The previous line abbreviates the following two assignment
            //    // statements.
            //    Task<byte[]> getContentTask = client.GetByteArrayAsync(url);
            //    byte[] urlContent = await getContentTask;

            //    DisplayResults(url, urlContent);

            //    // Update the total.
            //    total += urlContent.Length;
            //}

            // Display the total count for all of the web addresses.
            resultsTextBox.Text +=
                $"\r\n\r\nTotal bytes returned:  {total}\r\n";
        }

        private List<string> SetUpURLList()
        {
            List<string> urls = new List<string>
            {
                "https://msdn.microsoft.com",
                "https://msdn.microsoft.com/library/hh290136.aspx",
                "https://msdn.microsoft.com/library/ee256749.aspx",
                "https://msdn.microsoft.com/library/hh290138.aspx",
                "https://msdn.microsoft.com/library/hh290140.aspx",
                "https://msdn.microsoft.com/library/dd470362.aspx",
                "https://msdn.microsoft.com/library/aa578028.aspx",
                "https://msdn.microsoft.com/library/ms404677.aspx",
                "https://msdn.microsoft.com/library/ff730837.aspx"
            };
            return urls;
        }

        // The actions from the foreach loop are moved to this async method.
        async Task<int> ProcessURLAsync(string url, HttpClient client)
        {
            byte[] byteArray = await client.GetByteArrayAsync(url);
            DisplayResults(url, byteArray);
            return byteArray.Length;
        }

        private void DisplayResults(string url, byte[] content)
        {
            // Display the length of each web site. 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 "https://".
            var displayURL = url.Replace("https://", "");
            resultsTextBox.Text += $"\n{displayURL,-58} {bytes,8}";
        }
    }
}
```

See also

- Task.WhenAll
- Walkthrough: Accessing the Web by Using async and await (C#)

How to: Make Multiple Web Requests in Parallel by Using async and await (C#)

4/28/2019 • 5 minutes to read • Edit Online

In an async method, tasks are started when they're created. The await operator is applied to the task at the point in the method where processing can't continue until the task finishes. Often a task is awaited as soon as it's created, as the following example shows.

```
var result = await someWebAccessMethodAsync(url);
```

However, you can separate creating the task from awaiting the task if your program has other work to accomplish that doesn't depend on the completion of the task.

```
// The following line creates and starts the task.
var myTask = someWebAccessMethodAsync(url);

// While the task is running, you can do other work that doesn't depend
// on the results of the task.
// . . . . .

// The application of await suspends the rest of this method until the task is complete.
var result = await myTask;
```

Between starting a task and awaiting it, you can start other tasks. The additional tasks implicitly run in parallel, but no additional threads are created.

The following program starts three asynchronous web downloads and then awaits them in the order in which they're called. Notice, when you run the program, that the tasks don't always finish in the order in which they're created and awaited. They start to run when they're created, and one or more of the tasks might finish before the method reaches the await expressions.

NOTE

To complete this project, you must have Visual Studio 2012 or higher and the .NET Framework 4.5 or higher installed on your computer.

For another example that starts multiple tasks at the same time, see How to: Extend the async Walkthrough by Using Task.WhenAll (C#).

You can download the code for this example from Developer Code Samples.

To set up the project

1. To set up a WPF application, complete the following steps. You can find detailed instructions for these steps in Walkthrough: Accessing the Web by Using async and await (C#).

   - Create a WPF application that contains a text box and a button. Name the button `startButton`, and name the text box `resultsTextBox`.
   - Add a reference for System.Net.Http.
   - In the MainWindow.xaml.cs file, add a `using` directive for `System.Net.Http`.

To add the code

1. In the design window, MainWindow.xaml, double-click the button to create the `startButton_Click` event handler in MainWindow.xaml.cs.

2. Copy the following code, and paste it into the body of `startButton_Click` in MainWindow.xaml.cs.

```
resultsTextBox.Clear();
await CreateMultipleTasksAsync();
resultsTextBox.Text += "\r\n\r\nControl returned to startButton_Click.\r\n";
```

   The code calls an asynchronous method, `CreateMultipleTasksAsync`, which drives the application.

3. Add the following support methods to the project:

   - `ProcessURLAsync` uses an HttpClient method to download the contents of a website as a byte array. The support method, `ProcessURLAsync` then displays and returns the length of the array.

   - `DisplayResults` displays the number of bytes in the byte array for each URL. This display shows when each task has finished downloading.

   Copy the following methods, and paste them after the `startButton_Click` event handler in MainWindow.xaml.cs.

```
async Task<int> ProcessURLAsync(string url, HttpClient client)
{
    var byteArray = await client.GetByteArrayAsync(url);
    DisplayResults(url, byteArray);
    return byteArray.Length;
}

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 "https://".
    var displayURL = url.Replace("https://", "");
    resultsTextBox.Text += $"\n{displayURL,-58} {bytes,8}";
}
```

4. Finally, define method `CreateMultipleTasksAsync`, which performs the following steps.

   - The method declares an `HttpClient` object,which you need to access method GetByteArrayAsync in `ProcessURLAsync`.

   - The method creates and starts three tasks of type Task<TResult>, where `TResult` is an integer. As each task finishes, `DisplayResults` displays the task's URL and the length of the downloaded contents. Because the tasks are running asynchronously, the order in which the results appear might differ from the order in which they were declared.

   - The method awaits the completion of each task. Each `await` operator suspends execution of `CreateMultipleTasksAsync` until the awaited task is finished. The operator also retrieves the return value from the call to `ProcessURLAsync` from each completed task.

   - When the tasks have been completed and the integer values have been retrieved, the method sums the lengths of the websites and displays the result.

   Copy the following method, and paste it into your solution.

```
private async Task CreateMultipleTasksAsync()
{
    // Declare an HttpClient object, and increase the buffer size. The
    // default buffer size is 65,536.
    HttpClient client =
        new HttpClient() { MaxResponseContentBufferSize = 1000000 };

    // Create and start the tasks. As each task finishes, DisplayResults
    // displays its length.
    Task<int> download1 =
        ProcessURLAsync("https://msdn.microsoft.com", client);
    Task<int> download2 =
        ProcessURLAsync("https://msdn.microsoft.com/library/hh156528(VS.110).aspx", client);
    Task<int> download3 =
        ProcessURLAsync("https://msdn.microsoft.com/library/67w7t67f.aspx", client);

    // Await each task.
    int length1 = await download1;
    int length2 = await download2;
    int length3 = await download3;

    int total = length1 + length2 + length3;

    // Display the total count for the downloaded websites.
    resultsTextBox.Text += $"\r\n\r\nTotal bytes returned:  {total}\r\n";
}
```

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

   Run the program several times to verify that the three tasks don’t always finish in the same order and that the order in which they finish isn't necessarily the order in which they’re created and awaited.

Example

The following code contains the full example.

```
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 directive, and add a reference for System.Net.Http.
using System.Net.Http;

namespace AsyncExample_MultipleTasks
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private async void startButton_Click(object sender, RoutedEventArgs e)
        {
```

```
        {
            resultsTextBox.Clear();
            await CreateMultipleTasksAsync();
            resultsTextBox.Text += "\r\n\r\nControl returned to startButton_Click.\r\n";
        }

        private async Task CreateMultipleTasksAsync()
        {
            // Declare an HttpClient object, and increase the buffer size. The
            // default buffer size is 65,536.
            HttpClient client =
                new HttpClient() { MaxResponseContentBufferSize = 1000000 };

            // Create and start the tasks. As each task finishes, DisplayResults
            // displays its length.
            Task<int> download1 =
                ProcessURLAsync("https://msdn.microsoft.com", client);
            Task<int> download2 =
                ProcessURLAsync("https://msdn.microsoft.com/library/hh156528(VS.110).aspx", client);
            Task<int> download3 =
                ProcessURLAsync("https://msdn.microsoft.com/library/67w7t67f.aspx", client);

            // Await each task.
            int length1 = await download1;
            int length2 = await download2;
            int length3 = await download3;

            int total = length1 + length2 + length3;

            // Display the total count for the downloaded websites.
            resultsTextBox.Text += $"\r\n\r\nTotal bytes returned:  {total}\r\n";
        }

        async Task<int> ProcessURLAsync(string url, HttpClient client)
        {
            var byteArray = await client.GetByteArrayAsync(url);
            DisplayResults(url, byteArray);
            return byteArray.Length;
        }

        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 "https://".
            var displayURL = url.Replace("https://", "");
            resultsTextBox.Text += $"\n{displayURL,-58} {bytes,8}";
        }
    }
}
```

See also

- Walkthrough: Accessing the Web by Using async and await (C#)
- Asynchronous Programming with async and await (C#)
- How to: Extend the async Walkthrough by Using Task.WhenAll (C#)

Async Return Types (C#)

6/25/2019 • 7 minutes to read • Edit Online

Async methods can have the following return types:

- Task<TResult>, for an async method that returns a value.
- Task, for an async method that performs an operation but returns no value.
- `void`, for an event handler.
- Starting with C# 7.0, any type that has an accessible `GetAwaiter` method. The object returned by the `GetAwaiter` method must implement the System.Runtime.CompilerServices.ICriticalNotifyCompletion interface.

For more information about async methods, see Asynchronous Programming with async and await (C#).

Each return type is examined in one of the following sections, and you can find a full example that uses all three types at the end of the topic.

Task<TResult> Return Type

The Task<TResult> return type is used for an async method that contains a return (C#) statement in which the operand has type `TResult`.

In the following example, the `GetLeisureHours` async method contains a `return` statement that returns an integer. Therefore, the method declaration must specify a return type of `Task<int>`. The FromResult async method is a placeholder for an operation that returns a string.

```
using System;
using System.Linq;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      Console.WriteLine(ShowTodaysInfo().Result);
   }

   private static async Task<string> ShowTodaysInfo()
   {
      string ret = $"Today is {DateTime.Today:D}\n" +
                   "Today's hours of leisure: " +
                   $"{await GetLeisureHours()}";
      return ret;
   }

   static async Task<int> GetLeisureHours()
   {
       // Task.FromResult is a placeholder for actual work that returns a string.
       var today = await Task.FromResult<string>(DateTime.Now.DayOfWeek.ToString());

       // The method then can process the result in some way.
       int leisureHours;
       if (today.First() == 'S')
           leisureHours = 16;
       else
           leisureHours = 5;

       return leisureHours;
   }
}
// The example displays output like the following:
//       Today is Wednesday, May 24, 2017
//       Today's hours of leisure: 5
// </Snippet >
```

When `GetLeisureHours` is called from within an await expression in the `ShowTodaysInfo` method, the await expression retrieves the integer value (the value of `leisureHours`) that's stored in the task returned by the `GetLeisureHours` method. For more information about await expressions, see await.

You can better understand how this happens by separating the call to `GetLeisureHours` from the application of `await`, as the following code shows. A call to method `GetLeisureHours` that isn't immediately awaited returns a `Task<int>`, as you would expect from the declaration of the method. The task is assigned to the `integerTask` variable in the example. Because `integerTask` is a Task<TResult>, it contains a Result property of type `TResult`. In this case, `TResult` represents an integer type. When `await` is applied to `integerTask`, the await expression evaluates to the contents of the Result property of `integerTask`. The value is assigned to the `ret` variable.

IMPORTANT

The Result property is a blocking property. If you try to access it before its task is finished, the thread that's currently active is blocked until the task completes and the value is available. In most cases, you should access the value by using `await` instead of accessing the property directly.
The previous example retrieved the value of the Result property to block the main thread so that the `ShowTodaysInfo` method could finish execution before the application ended.

```
var integerTask = GetLeisureHours();

// You can do other work that does not rely on integerTask before awaiting.

string ret = $"Today is {DateTime.Today:D}\n" +
             "Today's hours of leisure: " +
             $"{await integerTask}";
```

Task Return Type

Async methods that don't contain a `return` statement or that contain a `return` statement that doesn't return an operand usually have a return type of Task. Such methods return `void` if they run synchronously. If you use a Task return type for an async method, a calling method can use an `await` operator to suspend the caller's completion until the called async method has finished.

In the following example, the `WaitAndApologize` async method doesn't contain a `return` statement, so the method returns a Task object. This enables `WaitAndApologize` to be awaited. Note that the Task type doesn't include a `Result` property because it has no return value.

```
using System;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      DisplayCurrentInfo().Wait();
   }

   static async Task DisplayCurrentInfo()
   {
      await WaitAndApologize();
      Console.WriteLine($"Today is {DateTime.Now:D}");
      Console.WriteLine($"The current time is {DateTime.Now.TimeOfDay:t}");
      Console.WriteLine("The current temperature is 76 degrees.");
   }

   static async Task WaitAndApologize()
   {
      // Task.Delay is a placeholder for actual work.
      await Task.Delay(2000);
      // Task.Delay delays the following line by two seconds.
      Console.WriteLine("\nSorry for the delay. . . .\n");
   }
}
// The example displays the following output:
//       Sorry for the delay. . . .
//
//       Today is Wednesday, May 24, 2017
//       The current time is 15:25:16.2935649
//       The current temperature is 76 degrees.
```

`WaitAndApologize` is awaited by using an await statement instead of an await expression, similar to the calling statement for a synchronous void-returning method. The application of an await operator in this case doesn't produce a value.

As in the previous Task<TResult> example, you can separate the call to `WaitAndApologize` from the application of an await operator, as the following code shows. However, remember that a `Task` doesn't have a `Result` property, and that no value is produced when an await operator is applied to a `Task`.

The following code separates calling the `WaitAndApologize` method from awaiting the task that the method returns.

```
Task wait = WaitAndApologize();

string output = $"Today is {DateTime.Now:D}\n" +
                $"The current time is {DateTime.Now.TimeOfDay:t}\n" +
                $"The current temperature is 76 degrees.\n";
await wait;
Console.WriteLine(output);
```

Void return type

You use the `void` return type in asynchronous event handlers, which require a `void` return type. For methods other than event handlers that don't return a value, you should return a Task instead, because an async method that returns `void` can't be awaited. Any caller of such a method must be able to continue to completion without waiting for the called async method to finish, and the caller must be independent of any values or exceptions that the async method generates.

The caller of a void-returning async method can't catch exceptions that are thrown from the method, and such unhandled exceptions are likely to cause your application to fail. If an exception occurs in an async method that returns a Task or Task<TResult>, the exception is stored in the returned task and is rethrown when the task is awaited. Therefore, make sure that any async method that can produce an exception has a return type of Task or Task<TResult> and that calls to the method are awaited.

For more information about how to catch exceptions in async methods, see the Exceptions in Async Methods section of the try-catch topic.

The following example shows the behavior of an async event handler. Note that in the example code, an async event handler must let the main thread know when it finishes. Then the main thread can wait for an async event handler to complete before exiting the program.

```
using System;
using System.Threading.Tasks;

public class NaiveButton
{
    public event EventHandler Clicked;

    public void Click()
    {
        Console.WriteLine("Somebody has clicked a button. Let's raise the event...");
        Clicked?.Invoke(this, EventArgs.Empty);
        Console.WriteLine("All listeners are notified.");
    }
}

public class AsyncVoidExample
{
    static TaskCompletionSource<bool> tcs;

    static async Task Main()
    {
        tcs = new TaskCompletionSource<bool>();
        var secondHandlerFinished = tcs.Task;

        var button = new NaiveButton();
        button.Clicked += Button_Clicked_1;
        button.Clicked += Button_Clicked_2_Async;
        button.Clicked += Button_Clicked_3;
        Console.WriteLine("About to click a button...");
        button.Click();
        Console.WriteLine("Button's Click method returned.");

        await secondHandlerFinished;
    }

    private static void Button_Clicked_1(object sender, EventArgs e)
    {
        Console.WriteLine("   Handler 1 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("   Handler 1 is done.");
    }

    private static async void Button_Clicked_2_Async(object sender, EventArgs e)
    {
        Console.WriteLine("   Handler 2 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("   Handler 2 is about to go async...");
        await Task.Delay(500);
        Console.WriteLine("   Handler 2 is done.");
        tcs.SetResult(true);
    }

    private static void Button_Clicked_3(object sender, EventArgs e)
    {
        Console.WriteLine("   Handler 3 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("   Handler 3 is done.");
    }
}

// Expected output:
// About to click a button...
// Somebody has clicked a button. Let's raise the event...
//    Handler 1 is starting...
//    Handler 1 is done.
//    Handler 2 is starting...
//    Handler 2 is about to go async...
//    Handler 3 is starting...
//    Handler 3 is done.
// All listeners are notified.
// Button's Click method returned.
//    Handler 2 is done.
```

Generalized async return types and ValueTask<TResult>

Starting with C# 7.0, an async method can return any type that has an accessible `GetAwaiter` method.

Because Task and Task<TResult> are reference types, memory allocation in performance-critical paths, particularly when allocations occur in tight loops, can adversely affect performance. Support for generalized return types means that you can return a lightweight value type instead of a reference type to avoid additional memory allocations.

.NET provides the System.Threading.Tasks.ValueTask<TResult> structure as a lightweight implementation of a generalized task-returning value. To use the System.Threading.Tasks.ValueTask<TResult> type, you must add the `System.Threading.Tasks.Extensions` NuGet package to your project. The following example uses the ValueTask<TResult> structure to retrieve the value of two dice rolls.

```
using System;
using System.Threading.Tasks;

class Program
{
   static Random rnd;

   static void Main()
   {
      Console.WriteLine($"You rolled {GetDiceRoll().Result}");
   }

   private static async ValueTask<int> GetDiceRoll()
   {
      Console.WriteLine("...Shaking the dice...");
      int roll1 = await Roll();
      int roll2 = await Roll();
      return roll1 + roll2;
   }

   private static async ValueTask<int> Roll()
   {
      if (rnd == null)
         rnd = new Random();

      await Task.Delay(500);
      int diceRoll = rnd.Next(1, 7);
      return diceRoll;
   }
}
// The example displays output like the following:
//       ...Shaking the dice...
//       You rolled 8
```

See also

- FromResult
- Walkthrough: Accessing the Web by Using async and await (C#)
- Control Flow in Async Programs (C#)
- async
- await

Control flow in async programs (C#)

4/28/2019 • 9 minutes to read • Edit Online

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.

In general, you mark methods that contain asynchronous code with the async (C#) modifier. In a method that's marked with an async modifier, you can use an await (C#) 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#).

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 method, GetStringAsync(String), 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.

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

    async Task<int> AccessTheWebAsync()
    {
        // TWO
        HttpClient client = new HttpClient();
        Task<string> getStringTask =
            client.GetStringAsync("https://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

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 or newer and the .NET Framework 4.5 or newer installed on your computer.

Download the program

You can download the application for this topic from Async Sample: Control Flow in Async Programs. 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.

Create the program Yourself

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.
3. Choose the **Installed** > **Visual C#** > **Windows Desktop** category, and then choose **WPF App** from the list of project templates.
4. Enter `AsyncTracer` as the name of the project, and then choose the **OK** button.

   The new project appears in **Solution Explorer**.
5. 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**.
6. In the **XAML** view of MainWindow.xaml, replace the code with the following code.

```
<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
" 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.
7. Add a reference for System.Net.Http.

8. In **Solution Explorer**, open the shortcut menu for MainWindow.xaml.cs, and then choose **View Code**.

9. In MainWindow.xaml.cs, replace the code with the following code.

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

        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("https://msdn.microsoft.com");

            resultsTextBox.Text += "\r\nTHREE: Back in AccessTheWebAsync.\r\n" +
                "           Task getStringTask is started.";
```

```
                            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;
        }
    }
}
```

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

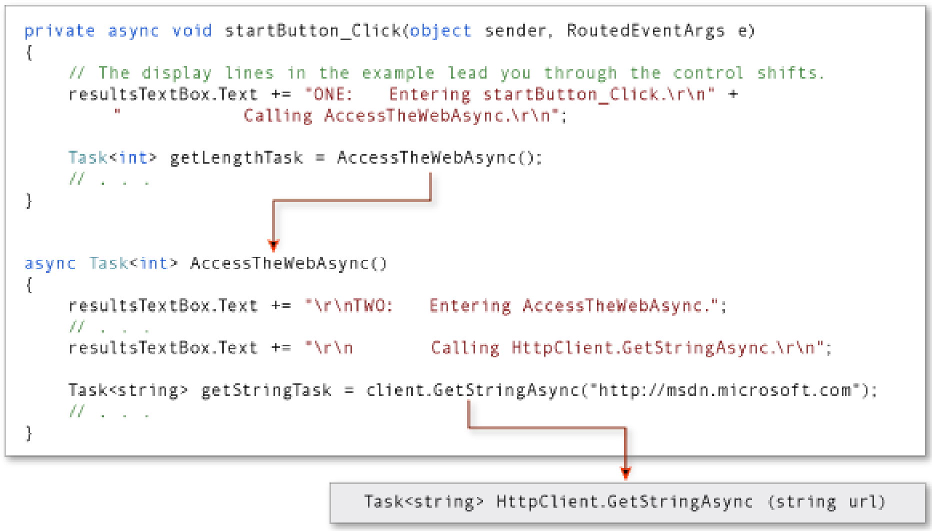
    The following output appears:

```
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

Steps ONE and TWO

The first two display lines trace the path as `startButton_Click` calls `AccessTheWebAsync`, and `AccessTheWebAsync` calls the asynchronous HttpClient method GetStringAsync(String). The following image outlines the calls from method to method.



```
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();
    // . . .
}


async Task<int> AccessTheWebAsync()
{
    resultsTextBox.Text += "\r\nTWO:   Entering AccessTheWebAsync.";
    // . . .
    resultsTextBox.Text += "\r\n       Calling HttpClient.GetStringAsync.\r\n";

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

```
Task<string> HttpClient.GetStringAsync (string url)
```

The return type of both `AccessTheWebAsync` and `client.GetStringAsync` is Task<TResult>. 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#).

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` 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

In `AccessTheWebAsync`, the asynchronous method GetStringAsync(String) 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.

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

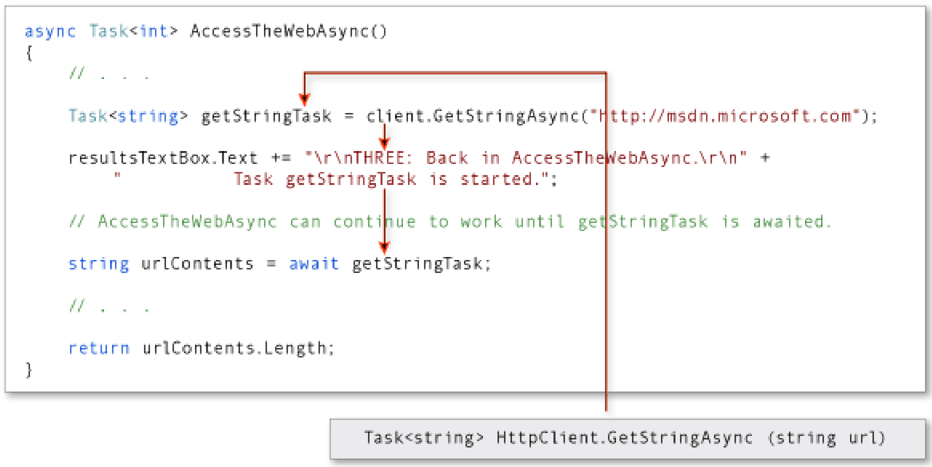
You can think of the task as a promise by `client.GetStringAsync` to produce an actual string eventually. In the meantime, if `AccessTheWebAsync` has work to do that doesn't depend on the promised string from `client.GetStringAsync`, that work can continue while `client.GetStringAsync` waits. In the 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.

```
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.



```
async Task<int> AccessTheWebAsync()
{
    // . . .

    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.

    string urlContents = await getStringTask;

    // . . .

    return urlContents.Length;
}
```

```
Task<string> HttpClient.GetStringAsync (string url)
```

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, the following assignment could replace the previous code that creates and then awaits `getStringTask`:

`string urlContents = await client.GetStringAsync("https://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

The declared return type of `AccessTheWebAsync` is `Task<int>`. 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.

```
Task<int> getLengthTask = AccessTheWebAsync();
```

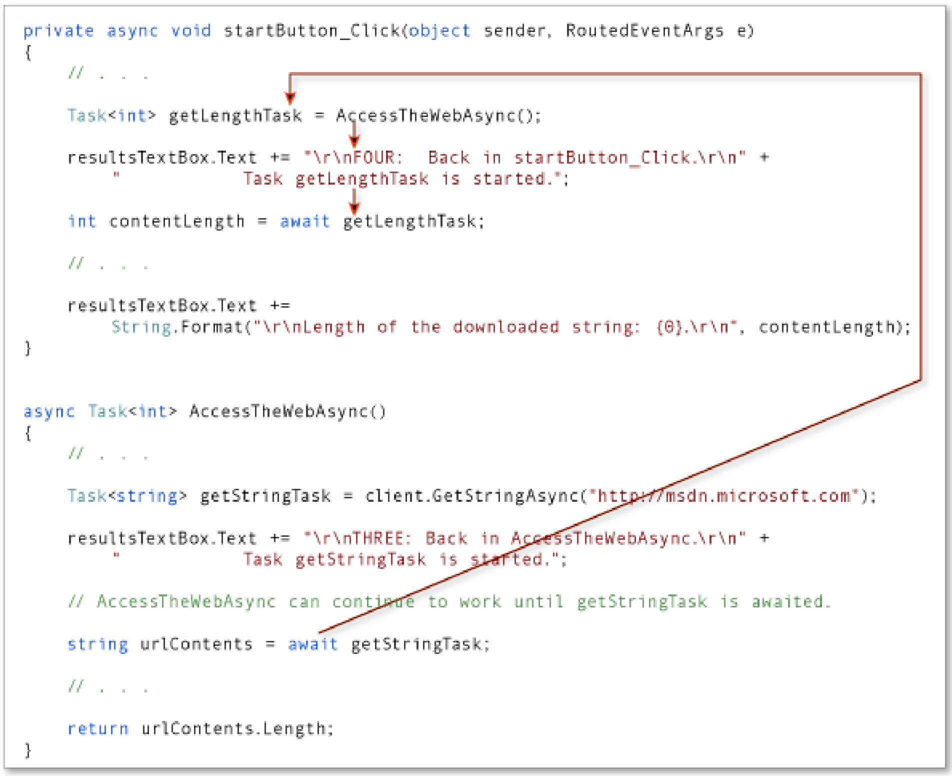
As in `AccessTheWebAsync`, `startButton_Click` can continue with work that doesn't depend on the results of the asynchronous task (`getLengthTask`) until the task is awaited. The following output lines represent that work.

```
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.

```
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.



```
private async void startButton_Click(object sender, RoutedEventArgs e)
{
    // . . .

    Task<int> getLengthTask = AccessTheWebAsync();

    resultsTextBox.Text += "\r\nFOUR:  Back in startButton_Click.\r\n" +
        "           Task getLengthTask is started.";

    int contentLength = await getLengthTask;

    // . . .

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


async Task<int> AccessTheWebAsync()
{
    // . . .

    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.

    string urlContents = await getStringTask;

    // . . .

    return urlContents.Length;
}
```

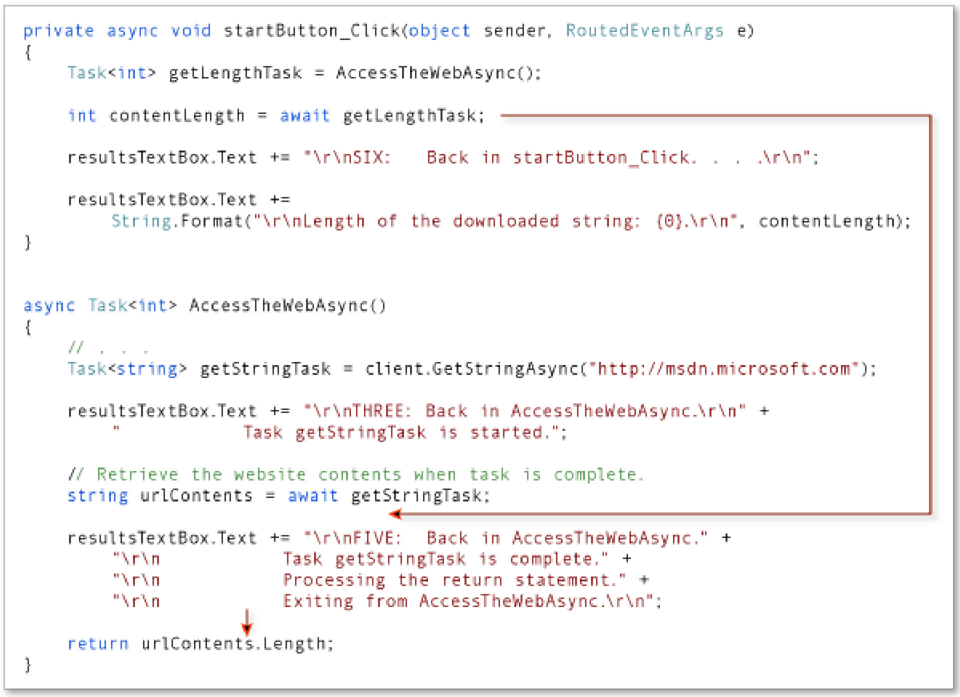
Step FIVE

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.



```
private async void startButton_Click(object sender, RoutedEventArgs e)
{
    Task<int> getLengthTask = AccessTheWebAsync();

    int contentLength = await getLengthTask;

    resultsTextBox.Text += "\r\nSIX:   Back in startButton_Click. . . .\r\n";

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


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

    resultsTextBox.Text += "\r\nTHREE: Back in AccessTheWebAsync.\r\n" +
        "           Task getStringTask is started.";

    // 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;
}
```

`AccessTheWebAsync` runs to completion, and control returns to `startButton_Click`, which is awaiting the completion.

Step SIX

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.

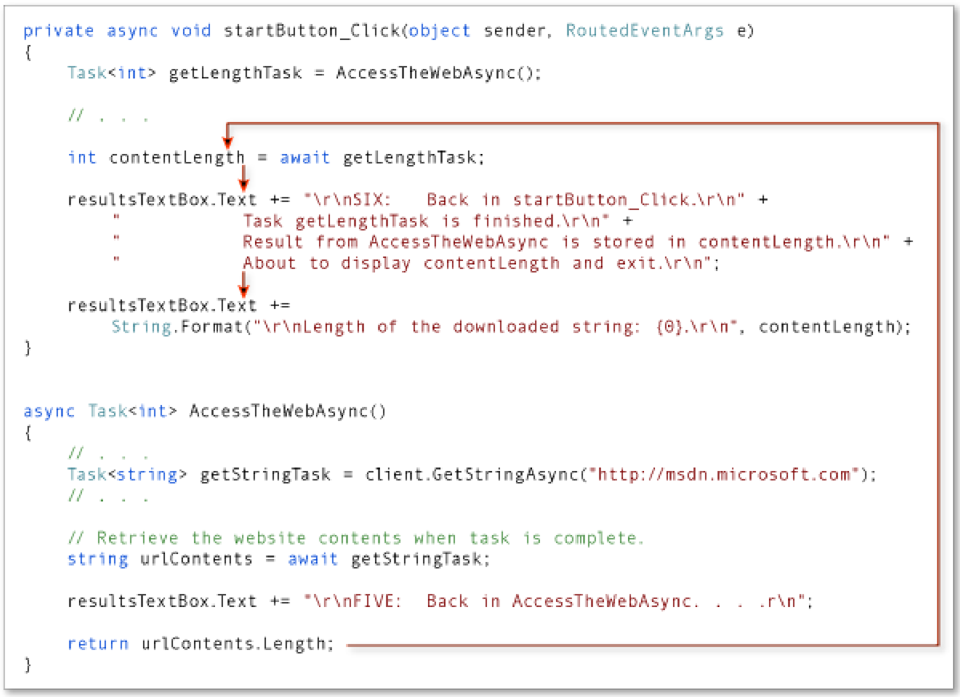
The following lines of output represent the resumption of processing in `startButton_Async`:

```
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.

```
int contentLength = await getLengthTask;
```

The following image shows the return of control from `AccessTheWebAsync` to `startButton_Click`.



```
private async void startButton_Click(object sender, RoutedEventArgs e)
{
    Task<int> getLengthTask = AccessTheWebAsync();

    // . . .

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

    // Retrieve the website contents when task is complete.
    string urlContents = await getStringTask;

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

    return urlContents.Length;
}
```

See also

- Asynchronous Programming with async and await (C#)
- Async Return Types (C#)
- Walkthrough: Accessing the Web by Using async and await (C#)
- Async Sample: Control Flow in Async Programs (C# and Visual Basic)

Fine-Tuning Your Async Application (C#)

4/28/2019 • 2 minutes to read • Edit Online

You can add precision and flexibility to your async applications by using the methods and properties that the Task type makes available. The topics in this section show examples that use CancellationToken and important `Task` methods such as Task.WhenAll and Task.WhenAny.

By using `WhenAny` and `WhenAll`, you can more easily start multiple tasks and await their completion by monitoring a single task.

- `WhenAny` returns a task that completes when any task in a collection is complete.

  For examples that use `WhenAny`, see Cancel Remaining Async Tasks after One Is Complete (C#) and Start Multiple Async Tasks and Process Them As They Complete (C#).

- `WhenAll` returns a task that completes when all tasks in a collection are complete.

  For more information and an example that uses `WhenAll`, see How to: Extend the async Walkthrough by Using Task.WhenAll (C#).

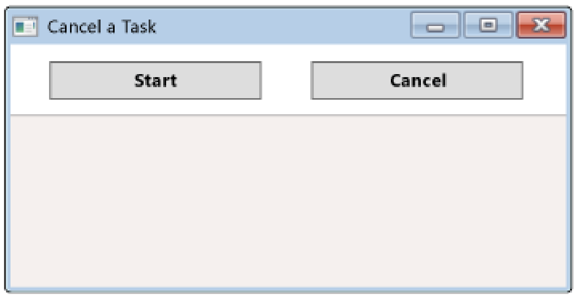
This section includes the following examples.

- 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#)

NOTE

To run the examples, you must have Visual Studio 2012 or newer and the .NET Framework 4.5 or newer installed on your computer.

The projects create a UI that contains a button that starts the process and a button that cancels it, as the following image shows. The buttons are named `startButton` and `cancelButton`.



You can download the complete Windows Presentation Foundation (WPF) projects from Async Sample: Fine Tuning Your Application.

See also

- Asynchronous Programming with async and await (C#)

Cancel an async task or a list of tasks (C#)

4/28/2019 • 10 minutes to read • Edit Online

You can set up a button that you can use to cancel an async application if you don't want to wait for it to finish. By following the examples in this topic, you can add a cancellation button to an application that downloads the contents of one website or a list of websites.

The examples use the UI that Fine-Tuning Your Async Application (C#) describes.

NOTE

To run the examples, you must have Visual Studio 2012 or newer and the .NET Framework 4.5 or newer installed on your computer.

Cancel a task

The first example associates the **Cancel** button with a single download task. If you choose the button while the application is downloading content, the download is canceled.

Download the example

You can download the complete Windows Presentation Foundation (WPF) project from Async Sample: Fine Tuning Your Application and then follow these steps.

1. Decompress the file that you downloaded, and then start Visual Studio.

2. On the menu bar, choose **File** > **Open** > **Project/Solution**.

3. In the **Open Project** dialog box, open the folder that holds the sample code that you decompressed, and then open the solution (.sln) file for AsyncFineTuningCS.

4. In **Solution Explorer**, open the shortcut menu for the **CancelATask** project, and then choose **Set as StartUp Project**.

5. Choose the **F5** key to run the project (or, press **Ctrl**+**F5** to run the project without debugging it).

TIP

If you don't want to download the project, you can review the MainWindow.xaml.cs files at the end of this topic.

Build the example

The following changes add a **Cancel** button to an application that downloads a website. If you don't want to download or build the example, you can review the final product in the "Complete Examples" section at the end of this topic. Asterisks mark the changes in the code.

To build the example yourself, step by step, follow the instructions in the "Downloading the Example" section, but choose **StarterCode** as the **StartUp Project** instead of **CancelATask**.

Then add the following changes to the MainWindow.xaml.cs file of that project.

1. Declare a `CancellationTokenSource` variable, `cts`, that’s in scope for all methods that access it.

```
public partial class MainWindow : Window
{
    // ***Declare a System.Threading.CancellationTokenSource.
    CancellationTokenSource cts;
```

2. Add the following event handler for the **Cancel** button. The event handler uses the CancellationTokenSource.Cancel method to notify `cts` when the user requests cancellation.

```
// ***Add an event handler for the Cancel button.
private void cancelButton_Click(object sender, RoutedEventArgs e)
{
    if (cts != null)
    {
        cts.Cancel();
    }
}
```

3. Make the following changes in the event handler for the **Start** button, `startButton_Click`.

   - Instantiate the `CancellationTokenSource`, `cts`.

   ```
   // ***Instantiate the CancellationTokenSource.
   cts = new CancellationTokenSource();
   ```

   - In the call to `AccessTheWebAsync`, which downloads the contents of a specified website, send the CancellationTokenSource.Token property of `cts` as an argument. The `Token` property propagates the message if cancellation is requested. Add a catch block that displays a message if the user chooses to cancel the download operation. The following code shows the changes.

   ```
   try
   {
       // ***Send a token to carry the message if cancellation is requested.
       int contentLength = await AccessTheWebAsync(cts.Token);
       resultsTextBox.Text += $"\r\nLength of the downloaded string: {contentLength}.\r\n";
   }
   // *** If cancellation is requested, an OperationCanceledException results.
   catch (OperationCanceledException)
   {
       resultsTextBox.Text += "\r\nDownload canceled.\r\n";
   }
   catch (Exception)
   {
       resultsTextBox.Text += "\r\nDownload failed.\r\n";
   }
   ```

4. In `AccessTheWebAsync`, use the HttpClient.GetAsync(String, CancellationToken) overload of the `GetAsync` method in the HttpClient type to download the contents of a website. Pass `ct`, the CancellationToken parameter of `AccessTheWebAsync`, as the second argument. The token carries the message if the user chooses the **Cancel** button.

   The following code shows the changes in `AccessTheWebAsync`.

```
// ***Provide a parameter for the CancellationToken.
async Task<int> AccessTheWebAsync(CancellationToken ct)
{
    HttpClient client = new HttpClient();

    resultsTextBox.Text += "\r\nReady to download.\r\n";

    // You might need to slow things down to have a chance to cancel.
    await Task.Delay(250);

    // GetAsync returns a Task<HttpResponseMessage>.
    // ***The ct argument carries the message if the Cancel button is chosen.
    HttpResponseMessage response = await
client.GetAsync("https://msdn.microsoft.com/library/dd470362.aspx", ct);

    // Retrieve the website contents from the HttpResponseMessage.
    byte[] urlContents = await response.Content.ReadAsByteArrayAsync();

    // The result of the method is the length of the downloaded website.
    return urlContents.Length;
}
```

5. If you don't cancel the program, it produces the following output.

```
Ready to download.
Length of the downloaded string: 158125.
```

   If you choose the **Cancel** button before the program finishes downloading the content, the program produces the following output.

```
Ready to download.
Download canceled.
```

Cancel a list of tasks

You can extend the previous example to cancel many tasks by associating the same `CancellationTokenSource` instance with each task. If you choose the **Cancel** button, you cancel all tasks that aren't yet complete.

Download the example

You can download the complete Windows Presentation Foundation (WPF) project from Async Sample: Fine Tuning Your Application and then follow these steps.

1. Decompress the file that you downloaded, and then start Visual Studio.
2. On the menu bar, choose **File** > **Open** > **Project/Solution**.
3. In the **Open Project** dialog box, open the folder that holds the sample code that you decompressed, and then open the solution (.sln) file for AsyncFineTuningCS.
4. In **Solution Explorer**, open the shortcut menu for the **CancelAListOfTasks** project, and then choose **Set as StartUp Project**.
5. Choose the **F5** key to run the project.

   Choose the **Ctrl**+**F5** keys to run the project without debugging it.

If you don't want to download the project, you can review the MainWindow.xaml.cs files at the end of this topic.

Build the example

To extend the example yourself, step by step, follow the instructions in the "Downloading the Example" section, but choose **CancelATask** as the **StartUp Project**. Add the following changes to that project. Asterisks mark the changes in the program.

1. Add a method to create a list of web addresses.

```
// ***Add a method that creates a list of web addresses.
private List<string> SetUpURLList()
{
    List<string> urls = new List<string>
    {
        "https://msdn.microsoft.com",
        "https://msdn.microsoft.com/library/hh290138.aspx",
        "https://msdn.microsoft.com/library/hh290140.aspx",
        "https://msdn.microsoft.com/library/dd470362.aspx",
        "https://msdn.microsoft.com/library/aa578028.aspx",
        "https://msdn.microsoft.com/library/ms404677.aspx",
        "https://msdn.microsoft.com/library/ff730837.aspx"
    };
    return urls;
}
```

2. Call the method in `AccessTheWebAsync`.

```
// ***Call SetUpURLList to make a list of web addresses.
List<string> urlList = SetUpURLList();
```

3. Add the following loop in `AccessTheWebAsync` to process each web address in the list.

```
// ***Add a loop to process the list of web addresses.
foreach (var url in urlList)
{
    // GetAsync returns a Task<HttpResponseMessage>.
    // Argument ct carries the message if the Cancel button is chosen.
    // ***Note that the Cancel button can cancel all remaining downloads.
    HttpResponseMessage response = await client.GetAsync(url, ct);

    // Retrieve the website contents from the HttpResponseMessage.
    byte[] urlContents = await response.Content.ReadAsByteArrayAsync();

    resultsTextBox.Text +=
        $"\r\nLength of the downloaded string: {urlContents.Length}.\r\n";
}
```

4. Because `AccessTheWebAsync` displays the lengths, the method doesn't need to return anything. Remove the return statement, and change the return type of the method to Task instead of Task<TResult>.

```
async Task AccessTheWebAsync(CancellationToken ct)
```

   Call the method from `startButton_Click` by using a statement instead of an expression.

```
await AccessTheWebAsync(cts.Token);
```

5. If you don’t cancel the program, it produces the following output.

```
Length of the downloaded string: 35939.

Length of the downloaded string: 237682.

Length of the downloaded string: 128607.

Length of the downloaded string: 158124.

Length of the downloaded string: 204890.

Length of the downloaded string: 175488.

Length of the downloaded string: 145790.

Downloads complete.
```

If you choose the **Cancel** button before the downloads are complete, the output contains the lengths of the downloads that completed before the cancellation.

```
Length of the downloaded string: 35939.

Length of the downloaded string: 237682.

Length of the downloaded string: 128607.

Downloads canceled.
```

Complete examples

The following sections contain the code for each of the previous examples. Notice that you must add a reference for System.Net.Http.

You can download the projects from Async Sample: Fine Tuning Your Application.

Example - Cancel a task

The following code is the complete MainWindow.xaml.cs file for the example that cancels a single task.

```
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;

// Add the following using directive for System.Threading.

using System.Threading;
namespace CancelATask
{
    public partial class MainWindow : Window
    {
        // ***Declare a System.Threading.CancellationTokenSource.
        CancellationTokenSource cts;

        public MainWindow()
        {
            InitializeComponent();
        }

        private async void startButton_Click(object sender, RoutedEventArgs e)
        {
            // ***Instantiate the CancellationTokenSource.
            cts = new CancellationTokenSource();

            resultsTextBox.Clear();

            try
            {
                // ***Send a token to carry the message if cancellation is requested.
                int contentLength = await AccessTheWebAsync(cts.Token);
                resultsTextBox.Text +=
                    $"\r\nLength of the downloaded string: {contentLength}.\r\n";
            }
            // *** If cancellation is requested, an OperationCanceledException results.
            catch (OperationCanceledException)
            {
                resultsTextBox.Text += "\r\nDownload canceled.\r\n";
            }
            catch (Exception)
            {
                resultsTextBox.Text += "\r\nDownload failed.\r\n";
            }

            // ***Set the CancellationTokenSource to null when the download is complete.
            cts = null;
        }

        // ***Add an event handler for the Cancel button.
        private void cancelButton_Click(object sender, RoutedEventArgs e)
        {
            if (cts != null)
            {
                cts.Cancel();
            }
        }

        // ***Provide a parameter for the CancellationToken.
        async Task<int> AccessTheWebAsync(CancellationToken ct)
        {
            HttpClient client = new HttpClient();

            resultsTextBox.Text += "\r\nReady to download.\r\n";

            // You might need to slow things down to have a chance to cancel.
            await Task.Delay(250);

            // GetAsync returns a Task<HttpResponseMessage>.
            // ***The ct argument carries the message if the Cancel button is chosen.
            HttpResponseMessage response = await
client.GetAsync("https://msdn.microsoft.com/library/dd470362.aspx", ct);

            // Retrieve the website contents from the HttpResponseMessage.
            byte[] urlContents = await response.Content.ReadAsByteArrayAsync();

            // The result of the method is the length of the downloaded website.
            return urlContents.Length;
        }
    }

    // Output for a successful download:
```

```
    // Ready to download.

    // Length of the downloaded string: 158125.

    // Or, if you cancel:

    // Ready to download.

    // Download canceled.
}
```

Example - Cancel a list of tasks

The following code is the complete MainWindow.xaml.cs file for the example that cancels a list of tasks.

```
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;

// Add the following using directive for System.Threading.
using System.Threading;

namespace CancelAListOfTasks
{
    public partial class MainWindow : Window
    {
        // Declare a System.Threading.CancellationTokenSource.
        CancellationTokenSource cts;

        public MainWindow()
        {
            InitializeComponent();
        }

        private async void startButton_Click(object sender, RoutedEventArgs e)
        {
            // Instantiate the CancellationTokenSource.
            cts = new CancellationTokenSource();

            resultsTextBox.Clear();

            try
            {
                await AccessTheWebAsync(cts.Token);
                // ***Small change in the display lines.
                resultsTextBox.Text += "\r\nDownloads complete.";
            }
            catch (OperationCanceledException)
            {
                resultsTextBox.Text += "\r\nDownloads canceled.";
            }
            catch (Exception)
            {
                resultsTextBox.Text += "\r\nDownloads failed.";
            }

            // Set the CancellationTokenSource to null when the download is complete.
            cts = null;
        }

        // Add an event handler for the Cancel button.
        private void cancelButton_Click(object sender, RoutedEventArgs e)
        {
            if (cts != null)
            {
                cts.Cancel();
            }
        }

        // Provide a parameter for the CancellationToken.
        // ***Change the return type to Task because the method has no return statement.
        async Task AccessTheWebAsync(CancellationToken ct)
        {
            // Declare an HttpClient object.
            HttpClient client = new HttpClient();

            // ***Call SetUpURLList to make a list of web addresses.
            List<string> urlList = SetUpURLList();

            // ***Add a loop to process the list of web addresses.
            foreach (var url in urlList)
            {
                // GetAsync returns a Task<HttpResponseMessage>.
                // Argument ct carries the message if the Cancel button is chosen.
                // ***Note that the Cancel button can cancel all remaining downloads.
                HttpResponseMessage response = await client.GetAsync(url, ct);

                // Retrieve the website contents from the HttpResponseMessage.
                byte[] urlContents = await response.Content.ReadAsByteArrayAsync();

                resultsTextBox.Text +=
                    $"\r\nLength of the downloaded string: {urlContents.Length}.\r\n";
            }
        }

        // ***Add a method that creates a list of web addresses.
        private List<string> SetUpURLList()
        {
            List<string> urls = new List<string>
            {
                "https://msdn.microsoft.com",
                "https://msdn.microsoft.com/library/hh290138.aspx",
                "https://msdn.microsoft.com/library/hh290140.aspx",
                "https://msdn.microsoft.com/library/dd470362.aspx",
                "https://msdn.microsoft.com/library/aa578028.aspx",
                "https://msdn.microsoft.com/library/ms404677.aspx",
                "https://msdn.microsoft.com/library/ff730837.aspx"
            };
            return urls;
        }
    }

    // Output if you do not choose to cancel:

    //Length of the downloaded string: 35939.

    //Length of the downloaded string: 237682.

    //Length of the downloaded string: 128607.

    //Length of the downloaded string: 158124.
```

```
        //Length of the downloaded string: 204890.

        //Length of the downloaded string: 175488.

        //Length of the downloaded string: 145790.

        //Downloads complete.

        // Sample output if you choose to cancel:

        //Length of the downloaded string: 35939.

        //Length of the downloaded string: 237682.

        //Length of the downloaded string: 128607.

        //Downloads canceled.
    }
```

See also

- CancellationTokenSource
- CancellationToken
- Asynchronous Programming with async and await (C#)
- Fine-Tuning Your Async Application (C#)
- Async Sample: Fine Tuning Your Application

Cancel async tasks after a period of time (C#)

4/9/2019 • 4 minutes to read • Edit Online

You can cancel an asynchronous operation after a period of time by using the CancellationTokenSource.CancelAfter method if you don't want to wait for the operation to finish. This method schedules the cancellation of any associated tasks that aren't complete within the period of time that's designated by the `CancelAfter` expression.

This example adds to the code that's developed in Cancel an Async Task or a List of Tasks (C#) to download a list of websites and to display the length of the contents of each one.

NOTE

To run the examples, you must have Visual Studio 2012 or newer and the .NET Framework 4.5 or newer installed on your computer.

Download the example

You can download the complete Windows Presentation Foundation (WPF) project from Async Sample: Fine Tuning Your Application and then follow these steps.

1. Decompress the file that you downloaded, and then start Visual Studio.
2. On the menu bar, choose **File** > **Open** > **Project/Solution**.
3. In the **Open Project** dialog box, open the folder that holds the sample code that you decompressed, and then open the solution (.sln) file for AsyncFineTuningCS.
4. In **Solution Explorer**, open the shortcut menu for the **CancelAfterTime** project, and then choose **Set as StartUp Project**.
5. Choose the **F5** key to run the project. (Or, press **Ctrl**+**F5** to run the project without debugging it).
6. Run the program several times to verify that the output might show output for all websites, no websites, or some web sites.

If you don't want to download the project, you can review the MainWindow.xaml.cs file at the end of this topic.

Build the example

The example in this topic adds to the project that's developed in Cancel an Async Task or a List of Tasks (C#) to cancel a list of tasks. The example uses the same UI, although the **Cancel** button isn't used explicitly.

To build the example yourself, step by step, follow the instructions in the "Downloading the Example" section, but choose **CancelAListOfTasks** as the **StartUp Project**. Add the changes in this topic to that project.

To specify a maximum time before the tasks are marked as canceled, add a call to `CancelAfter` to `startButton_Click`, as the following example shows. The addition is marked with asterisks.

```
private async void startButton_Click(object sender, RoutedEventArgs e)
{
    // Instantiate the CancellationTokenSource.
    cts = new CancellationTokenSource();

    resultsTextBox.Clear();

    try
    {
        // ***Set up the CancellationTokenSource to cancel after 2.5 seconds. (You
        // can adjust the time.)
        cts.CancelAfter(2500);

        await AccessTheWebAsync(cts.Token);
        resultsTextBox.Text += "\r\nDownloads succeeded.\r\n";
    }
    catch (OperationCanceledException)
    {
        resultsTextBox.Text += "\r\nDownloads canceled.\r\n";
    }
    catch (Exception)
    {
        resultsTextBox.Text += "\r\nDownloads failed.\r\n";
    }

    cts = null;
}
```

Run the program several times to verify that the output might show output for all websites, no websites, or some web sites. The following output is a sample.

```
Length of the downloaded string: 35990.

Length of the downloaded string: 407399.

Length of the downloaded string: 226091.

Downloads canceled.
```

Complete example

The following code is the complete text of the MainWindow.xaml.cs file for the example. Asterisks mark the elements that were added for this example.

Notice that you must add a reference for System.Net.Http.

You can download the project from Async Sample: Fine Tuning Your Application.

```
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;

// Add the following using directive.
using System.Threading;

namespace CancelAfterTime
{
    public partial class MainWindow : Window
    {
        // Declare a System.Threading.CancellationTokenSource.
        CancellationTokenSource cts;

        public MainWindow()
        {
            InitializeComponent();
        }

        private async void startButton_Click(object sender, RoutedEventArgs e)
        {
            // Instantiate the CancellationTokenSource.
            cts = new CancellationTokenSource();

            resultsTextBox.Clear();

            try
            {
                // ***Set up the CancellationTokenSource to cancel after 2.5 seconds. (You
                // can adjust the time.)
                cts.CancelAfter(2500);

                await AccessTheWebAsync(cts.Token);
                resultsTextBox.Text += "\r\nDownloads succeeded.\r\n";
            }
            catch (OperationCanceledException)
            {
                resultsTextBox.Text += "\r\nDownloads canceled.\r\n";
            }
            catch (Exception)
            {
                resultsTextBox.Text += "\r\nDownloads failed.\r\n";
            }

            cts = null;
        }

        // You can still include a Cancel button if you want to.
        private void cancelButton_Click(object sender, RoutedEventArgs e)
        {
            if (cts != null)
            {
                cts.Cancel();
            }
        }

        async Task AccessTheWebAsync(CancellationToken ct)
        {
            // Declare an HttpClient object.
            HttpClient client = new HttpClient();

            // Make a list of web addresses.
            List<string> urlList = SetUpURLList();

            foreach (var url in urlList)
            {
                // GetAsync returns a Task<HttpResponseMessage>.
                // Argument ct carries the message if the Cancel button is chosen.
                // Note that the Cancel button cancels all remaining downloads.
                HttpResponseMessage response = await client.GetAsync(url, ct);

                // Retrieve the website contents from the HttpResponseMessage.
                byte[] urlContents = await response.Content.ReadAsByteArrayAsync();

                resultsTextBox.Text +=
                    $"\r\nLength of the downloaded string: {urlContents.Length}.\r\n";
            }
        }

        private List<string> SetUpURLList()
        {
            List<string> urls = new List<string>
            {
                "https://msdn.microsoft.com",
                "https://msdn.microsoft.com/library/windows/apps/br211380.aspx",
                "https://msdn.microsoft.com/library/hh290136.aspx",
                "https://msdn.microsoft.com/library/ee256749.aspx",
                "https://msdn.microsoft.com/library/ms404677.aspx",
                "https://msdn.microsoft.com/library/ff730837.aspx"
            };
            return urls;
        }
    }

    // Sample Output:

    // Length of the downloaded string: 35990.

    // Length of the downloaded string: 407399.

    // Length of the downloaded string: 226091.

    // Downloads canceled.
}
```

See also

- Asynchronous Programming with async and await (C#)
- Walkthrough: Accessing the Web by Using async and await (C#)
- Cancel an Async Task or a List of Tasks (C#)
- Fine-Tuning Your Async Application (C#)
- Async Sample: Fine Tuning Your Application

Cancel Remaining Async Tasks after One Is Complete (C#)

7/12/2019 • 6 minutes to read • Edit Online

By using the Task.WhenAny method together with a CancellationToken, you can cancel all remaining tasks when one task is complete. The `WhenAny` method takes an argument that's a collection of tasks. The method starts all the tasks and returns a single task. The single task is complete when any task in the collection is complete.

This example demonstrates how to use a cancellation token in conjunction with `WhenAny` to hold onto the first task to finish from the collection of tasks and to cancel the remaining tasks. Each task downloads the contents of a website. The example displays the length of the contents of the first download to complete and cancels the other downloads.

NOTE

To run the examples, you must have Visual Studio 2012 or newer and the .NET Framework 4.5 or newer installed on your computer.

Downloading the Example

You can download the complete Windows Presentation Foundation (WPF) project from Async Sample: Fine Tuning Your Application and then follow these steps.

1. Decompress the file that you downloaded, and then start Visual Studio.
2. On the menu bar, choose **File**, **Open**, **Project/Solution**.
3. In the **Open Project** dialog box, open the folder that holds the sample code that you decompressed, and then open the solution (.sln) file for AsyncFineTuningCS.
4. In **Solution Explorer**, open the shortcut menu for the **CancelAfterOneTask** project, and then choose **Set as StartUp Project**.
5. Choose the F5 key to run the project.

   Choose the Ctrl+F5 keys to run the project without debugging it.
6. Run the program several times to verify that different downloads finish first.

If you don't want to download the project, you can review the MainWindow.xaml.cs file at the end of this topic.

Building the Example

The example in this topic adds to the project that's developed in Cancel an Async Task or a List of Tasks (C#) to cancel a list of tasks. The example uses the same UI, although the **Cancel** button isn't used explicitly.

To build the example yourself, step by step, follow the instructions in the "Downloading the Example" section, but choose **CancelAListOfTasks** as the **StartUp Project**. Add the changes in this topic to that project.

In the MainWindow.xaml.cs file of the **CancelAListOfTasks** project, start the transition by moving the processing steps for each website from the loop in `AccessTheWebAsync` to the following async method.

```
// ***Bundle the processing steps for a website into one async method.
async Task<int> ProcessURLAsync(string url, HttpClient client, CancellationToken ct)
{
    // GetAsync returns a Task<HttpResponseMessage>.
    HttpResponseMessage response = await client.GetAsync(url, ct);

    // Retrieve the website contents from the HttpResponseMessage.
    byte[] urlContents = await response.Content.ReadAsByteArrayAsync();

    return urlContents.Length;
}
```

In `AccessTheWebAsync`, this example uses a query, the ToArray method, and the `WhenAny` method to create and start an array of tasks. The application of `WhenAny` to the array returns a single task that, when awaited, evaluates to the first task to reach completion in the array of tasks.

Make the following changes in `AccessTheWebAsync`. Asterisks mark the changes in the code file.

1. Comment out or delete the loop.

2. Create a query that, when executed, produces a collection of generic tasks. Each call to `ProcessURLAsync` returns a Task<TResult> where `TResult` is an integer.

```
// ***Create a query that, when executed, returns a collection of tasks.
IEnumerable<Task<int>> downloadTasksQuery =
    from url in urlList select ProcessURLAsync(url, client, ct);
```

3. Call `ToArray` to execute the query and start the tasks. The application of the `WhenAny` method in the next step would execute the query and start the tasks without using `ToArray`, but other methods might not. The safest practice is to force execution of the query explicitly.

```
// ***Use ToArray to execute the query and start the download tasks.
Task<int>[] downloadTasks = downloadTasksQuery.ToArray();
```

4. Call `WhenAny` on the collection of tasks. `WhenAny` returns a `Task(Of Task(Of Integer))` or `Task<Task<int>>`. That is, `WhenAny` returns a task that evaluates to a single `Task(Of Integer)` or `Task<int>` when it's awaited. That single task is the first task in the collection to finish. The task that finished first is assigned to `firstFinishedTask`. The type of `firstFinishedTask` is Task<TResult> where `TResult` is an integer because that's the return type of `ProcessURLAsync`.

```
// ***Call WhenAny and then await the result. The task that finishes
// first is assigned to firstFinishedTask.
Task<int> firstFinishedTask = await Task.WhenAny(downloadTasks);
```

5. In this example, you're interested only in the task that finishes first. Therefore, use CancellationTokenSource.Cancel to cancel the remaining tasks.

```
// ***Cancel the rest of the downloads. You just want the first one.
cts.Cancel();
```

6. Finally, await `firstFinishedTask` to retrieve the length of the downloaded content.

```
var length = await firstFinishedTask;
resultsTextBox.Text += $"\r\nLength of the downloaded website:  {length}\r\n";
```

Run the program several times to verify that different downloads finish first.

Complete Example

The following code is the complete MainWindow.xaml.cs file for the example. Asterisks mark the elements that were added for this example.

Notice that you must add a reference for System.Net.Http.

You can download the project from Async Sample: Fine Tuning Your Application.

```
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;

// Add the following using directive.
using System.Threading;

namespace CancelAfterOneTask
{
    public partial class MainWindow : Window
    {
        // Declare a System.Threading.CancellationTokenSource.
        CancellationTokenSource cts;

        public MainWindow()
        {
            InitializeComponent();
        }

        private async void startButton_Click(object sender, RoutedEventArgs e)
        {
            // Instantiate the CancellationTokenSource.
            cts = new CancellationTokenSource();

            resultsTextBox.Clear();

            try
            {
                await AccessTheWebAsync(cts.Token);
                resultsTextBox.Text += "\r\nDownload complete.";
            }
            catch (OperationCanceledException)
            {
                resultsTextBox.Text += "\r\nDownload canceled.";
            }
            catch (Exception)
            {
                resultsTextBox.Text += "\r\nDownload failed.";
            }

            // Set the CancellationTokenSource to null when the download is complete.
            cts = null;
        }

        // You can still include a Cancel button if you want to.
        private void cancelButton_Click(object sender, RoutedEventArgs e)
        {
            if (cts != null)
            {
                cts.Cancel();
            }
        }

        // Provide a parameter for the CancellationToken.
        async Task AccessTheWebAsync(CancellationToken ct)
        {
            HttpClient client = new HttpClient();

            // Call SetUpURLList to make a list of web addresses.
            List<string> urlList = SetUpURLList();

            // ***Comment out or delete the loop.
            //foreach (var url in urlList)
            //{
            //    // GetAsync returns a Task<HttpResponseMessage>.
            //    // Argument ct carries the message if the Cancel button is chosen.
            //    // ***Note that the Cancel button can cancel all remaining downloads.
            //    HttpResponseMessage response = await client.GetAsync(url, ct);

            //    // Retrieve the website contents from the HttpResponseMessage.
            //    byte[] urlContents = await response.Content.ReadAsByteArrayAsync();

            //    resultsTextBox.Text +=
            //        $"\r\nLength of the downloaded string: {urlContents.Length}.\r\n";
            //}

            // ***Create a query that, when executed, returns a collection of tasks.
            IEnumerable<Task<int>> downloadTasksQuery =
                from url in urlList select ProcessURLAsync(url, client, ct);

            // ***Use ToArray to execute the query and start the download tasks.
            Task<int>[] downloadTasks = downloadTasksQuery.ToArray();

            // ***Call WhenAny and then await the result. The task that finishes
            // first is assigned to firstFinishedTask.
            Task<int> firstFinishedTask = await Task.WhenAny(downloadTasks);

            // ***Cancel the rest of the downloads. You just want the first one.
            cts.Cancel();

            // ***Await the first completed task and display the results.
            // Run the program several times to demonstrate that different
            // websites can finish first.
            var length = await firstFinishedTask;
            resultsTextBox.Text += $"\r\nLength of the downloaded website:  {length}\r\n";
        }

        // ***Bundle the processing steps for a website into one async method.
        async Task<int> ProcessURLAsync(string url, HttpClient client, CancellationToken ct)
        {
            // GetAsync returns a Task<HttpResponseMessage>.
            HttpResponseMessage response = await client.GetAsync(url, ct);

            // Retrieve the website contents from the HttpResponseMessage.
            byte[] urlContents = await response.Content.ReadAsByteArrayAsync();
```

```
                return urlContents.Length;
            }

            // Add a method that creates a list of web addresses.
            private List<string> SetUpURLList()
            {
                List<string> urls = new List<string>
                {
                    "https://msdn.microsoft.com",
                    "https://msdn.microsoft.com/library/hh290138.aspx",
                    "https://msdn.microsoft.com/library/hh290140.aspx",
                    "https://msdn.microsoft.com/library/dd470362.aspx",
                    "https://msdn.microsoft.com/library/aa578028.aspx",
                    "https://msdn.microsoft.com/library/ms404677.aspx",
                    "https://msdn.microsoft.com/library/ff730837.aspx"
                };
                return urls;
            }
        }
        // Sample output:

        // Length of the downloaded website:  158856

        // Download complete.
    }
```

See also

- WhenAny
- Fine-Tuning Your Async Application (C#)
- Asynchronous Programming with async and await (C#)
- Async Sample: Fine Tuning Your Application

Start Multiple Async Tasks and Process Them As They Complete (C#)

4/28/2019 • 5 minutes to read • Edit Online

By using Task.WhenAny, you can start multiple tasks at the same time and process them one by one as they're completed rather than process them in the order in which they're started.

The following example uses a query to create a collection of tasks. Each task downloads the contents of a specified website. In each iteration of a while loop, an awaited call to `WhenAny` returns the task in the collection of tasks that finishes its download first. That task is removed from the collection and processed. The loop repeats until the collection contains no more tasks.

NOTE

To run the examples, you must have Visual Studio (2012 or newer) and the .NET Framework 4.5 or newer installed on your computer.

Download an example solution

You can download the complete Windows Presentation Foundation (WPF) project from Async Sample: Fine Tuning Your Application and then follow these steps.

TIP

If you don't want to download the project, you can review the MainWindow.xaml.cs file at the end of this topic instead.

1. Extract the files that you downloaded from the .zip file, and then start Visual Studio.
2. On the menu bar, choose **File** > **Open** > **Project/Solution**.
3. In the **Open Project** dialog box, open the folder that holds the sample code you downloaded, and then open the solution (.sln) file for AsyncFineTuningCS.
4. In **Solution Explorer**, open the shortcut menu for the **ProcessTasksAsTheyFinish** project, and then choose **Set as StartUp Project**.
5. Choose the **F5** key to run the program (or, press **Ctrl**+**F5** keys to run the program without debugging it).
6. Run the project several times to verify that the downloaded lengths don't always appear in the same order.

Create the program yourself

This example adds to the code that's developed in Cancel Remaining Async Tasks after One Is Complete (C#), and it uses the same UI.

To build the example yourself, step by step, follow the instructions in the Downloading the Example section, but set **CancelAfterOneTask** as the startup project. Add the changes in this topic to the `AccessTheWebAsync` method in that project. The changes are marked with asterisks.

The **CancelAfterOneTask** project already includes a query that, when executed, creates a collection of tasks. Each call to `ProcessURLAsync` in the following code returns a Task<TResult>, where `TResult` is an integer:

```
IEnumerable<Task<int>> downloadTasksQuery = from url in urlList select ProcessURL(url, client, ct);
```

In the MainWindow.xaml.cs file of the project, make the following changes to the `AccessTheWebAsync` method.

- Execute the query by applying Enumerable.ToList instead of ToArray.

  ```
  List<Task<int>> downloadTasks = downloadTasksQuery.ToList();
  ```

- Add a `while` loop that performs the following steps for each task in the collection:

  1. Awaits a call to `WhenAny` to identify the first task in the collection to finish its download.

     ```
     Task<int> firstFinishedTask = await Task.WhenAny(downloadTasks);
     ```

  2. Removes that task from the collection.

     ```
     downloadTasks.Remove(firstFinishedTask);
     ```

  3. Awaits `firstFinishedTask`, which is returned by a call to `ProcessURLAsync`. The `firstFinishedTask` variable is a Task<TResult> where `TReturn` is an integer. The task is already complete, but you await it to retrieve the length of the downloaded website, as the following example shows.

     ```
     int length = await firstFinishedTask;
     resultsTextBox.Text += $"\r\nLength of the download:  {length}";
     ```

Run the program several times to verify that the downloaded lengths don't always appear in the same order.

Caution

You can use `WhenAny` in a loop, as described in the example, to solve problems that involve a small number of tasks. However, other approaches are more efficient if you have a large number of tasks to process. For more information and examples, see Processing tasks as they complete.

Complete example

The following code is the complete text of the MainWindow.xaml.cs file for the example. Asterisks mark the elements that were added for this example. Also, take note that you must add a reference for System.Net.Http.

You can download the project from Async Sample: Fine Tuning Your Application.

```
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;
```

```
using System.Net.Http;

// Add the following using directive.
using System.Threading;

namespace ProcessTasksAsTheyFinish
{
    public partial class MainWindow : Window
    {
        // Declare a System.Threading.CancellationTokenSource.
        CancellationTokenSource cts;

        public MainWindow()
        {
            InitializeComponent();
        }

        private async void startButton_Click(object sender, RoutedEventArgs e)
        {
            resultsTextBox.Clear();

            // Instantiate the CancellationTokenSource.
            cts = new CancellationTokenSource();

            try
            {
                await AccessTheWebAsync(cts.Token);
                resultsTextBox.Text += "\r\nDownloads complete.";
            }
            catch (OperationCanceledException)
            {
                resultsTextBox.Text += "\r\nDownloads canceled.\r\n";
            }
            catch (Exception)
            {
                resultsTextBox.Text += "\r\nDownloads failed.\r\n";
            }

            cts = null;
        }

        private void cancelButton_Click(object sender, RoutedEventArgs e)
        {
            if (cts != null)
            {
                cts.Cancel();
            }
        }

        async Task AccessTheWebAsync(CancellationToken ct)
        {
            HttpClient client = new HttpClient();

            // Make a list of web addresses.
            List<string> urlList = SetUpURLList();

            // ***Create a query that, when executed, returns a collection of tasks.
            IEnumerable<Task<int>> downloadTasksQuery =
                from url in urlList select ProcessURL(url, client, ct);

            // ***Use ToList to execute the query and start the tasks.
            List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

            // ***Add a loop to process the tasks one at a time until none remain.
            while (downloadTasks.Count > 0)
            {
                    // Identify the first task that completes.
                    Task<int> firstFinishedTask = await Task.WhenAny(downloadTasks);

                    // ***Remove the selected task from the list so that you don't
                    // ***Remove the selected task from the list so that you don't
                    // process it more than once.
                    downloadTasks.Remove(firstFinishedTask);

                    // Await the completed task.
                    int length = await firstFinishedTask;
                    resultsTextBox.Text += $"\r\nLength of the download:  {length}";
                }
            }

            private List<string> SetUpURLList()
            {
                List<string> urls = new List<string>
                {
                    "https://msdn.microsoft.com",
                    "https://msdn.microsoft.com/library/windows/apps/br211380.aspx",
                    "https://msdn.microsoft.com/library/hh290136.aspx",
                    "https://msdn.microsoft.com/library/dd470362.aspx",
                    "https://msdn.microsoft.com/library/aa578028.aspx",
                    "https://msdn.microsoft.com/library/ms404677.aspx",
                    "https://msdn.microsoft.com/library/ff730837.aspx"
                };
                return urls;
            }

            async Task<int> ProcessURL(string url, HttpClient client, CancellationToken ct)
            {
                // GetAsync returns a Task<HttpResponseMessage>.
                HttpResponseMessage response = await client.GetAsync(url, ct);

                // Retrieve the website contents from the HttpResponseMessage.
                byte[] urlContents = await response.Content.ReadAsByteArrayAsync();

                return urlContents.Length;
            }
        }
    }

    // Sample Output:

    // Length of the download:  226093
    // Length of the download:  412588
    // Length of the download:  175490
    // Length of the download:  204890
    // Length of the download:  158855
    // Length of the download:  145790
    // Length of the download:  44908
    // Downloads complete.
```

See also

- WhenAny
- Fine-Tuning Your Async Application (C#)
- Asynchronous Programming with async and await (C#)
- Async Sample: Fine Tuning Your Application

Handling Reentrancy in Async Apps (C#)

4/11/2019 • 17 minutes to read • Edit Online

When you include asynchronous code in your app, you should consider and possibly prevent reentrancy, which refers to reentering an asynchronous operation before it has completed. If you don't identify and handle possibilities for reentrancy, it can cause unexpected results.

In this topic

NOTE

To run the example, you must have Visual Studio 2012 or newer and the .NET Framework 4.5 or newer installed on your computer.

Recognizing Reentrancy

In the example in this topic, users choose a **Start** button to initiate an asynchronous app that downloads a series of websites and calculates the total number of bytes that are downloaded. A synchronous version of the example would respond the same way regardless of how many times a user chooses the button because, after the first time, the UI thread ignores those events until the app finishes running. In an asynchronous app, however, the UI thread continues to respond, and you might reenter the asynchronous operation before it has completed.

The following example shows the expected output if the user chooses the **Start** button only once. A list of the downloaded websites appears with the size, in bytes, of each site. The total number of bytes appears at the end.

```
1. msdn.microsoft.com/library/hh191443.aspx               83732
2. msdn.microsoft.com/library/aa578028.aspx              205273
3. msdn.microsoft.com/library/jj155761.aspx               29019
4. msdn.microsoft.com/library/hh290140.aspx              117152
5. msdn.microsoft.com/library/hh524395.aspx               68959
6. msdn.microsoft.com/library/ms404677.aspx              197325
7. msdn.microsoft.com                                            42972
8. msdn.microsoft.com/library/ff730837.aspx              146159

TOTAL bytes returned:  890591
```

However, if the user chooses the button more than once, the event handler is invoked repeatedly, and the download process is reentered each time. As a result, several asynchronous operations are running at the same time, the output interleaves the results, and the total number of bytes is confusing.

```
1. msdn.microsoft.com/library/hh191443.aspx                83732
2. msdn.microsoft.com/library/aa578028.aspx               205273
3. msdn.microsoft.com/library/jj155761.aspx                29019
4. msdn.microsoft.com/library/hh290140.aspx               117152
5. msdn.microsoft.com/library/hh524395.aspx                68959
1. msdn.microsoft.com/library/hh191443.aspx                83732
2. msdn.microsoft.com/library/aa578028.aspx               205273
6. msdn.microsoft.com/library/ms404677.aspx               197325
3. msdn.microsoft.com/library/jj155761.aspx                29019
7. msdn.microsoft.com                                            42972
4. msdn.microsoft.com/library/hh290140.aspx               117152
8. msdn.microsoft.com/library/ff730837.aspx               146159

TOTAL bytes returned:  890591

5. msdn.microsoft.com/library/hh524395.aspx                68959
1. msdn.microsoft.com/library/hh191443.aspx                83732
2. msdn.microsoft.com/library/aa578028.aspx               205273
6. msdn.microsoft.com/library/ms404677.aspx               197325
3. msdn.microsoft.com/library/jj155761.aspx                29019
4. msdn.microsoft.com/library/hh290140.aspx               117152
7. msdn.microsoft.com                                            42972
5. msdn.microsoft.com/library/hh524395.aspx                68959
8. msdn.microsoft.com/library/ff730837.aspx               146159

TOTAL bytes returned:  890591

6. msdn.microsoft.com/library/ms404677.aspx               197325
7. msdn.microsoft.com                                            42972
8. msdn.microsoft.com/library/ff730837.aspx               146159

TOTAL bytes returned:  890591
```

You can review the code that produces this output by scrolling to the end of this topic. You can experiment with the code by downloading the solution to your local computer and then running the WebsiteDownload project or by using the code at the end of this topic to create your own project. For more information and instructions, see Reviewing and Running the Example App.

Handling Reentrancy

You can handle reentrancy in a variety of ways, depending on what you want your app to do. This topic presents the following examples:

- Disable the Start Button

  Disable the **Start** button while the operation is running so that the user can't interrupt it.

- Cancel and Restart the Operation

  Cancel any operation that is still running when the user chooses the **Start** button again, and then let the most recently requested operation continue.

- Run Multiple Operations and Queue the Output

  Allow all requested operations to run asynchronously, but coordinate the display of output so that the results from each operation appear together and in order.

Disable the Start Button

You can block the **Start** button while an operation is running by disabling the button at the top of the `StartButton_Click` event handler. You can then reenable the button from within a `finally` block when the operation finishes so that users can run the app again.

To set up this scenario, make the following changes to the basic code that is provided in Reviewing and Running the Example App. You also can download the finished app from Async Samples: Reentrancy in .NET Desktop Apps. The name of the project is DisableStartButton.

```
private async void StartButton_Click(object sender, RoutedEventArgs e)
{
    // This line is commented out to make the results clearer in the output.
    //ResultsTextBox.Text = "";

    // ***Disable the Start button until the downloads are complete.
    StartButton.IsEnabled = false;

    try
    {
        await AccessTheWebAsync();
    }
    catch (Exception)
    {
        ResultsTextBox.Text += "\r\nDownloads failed.";
    }
    // ***Enable the Start button in case you want to run the program again.
    finally
    {
        StartButton.IsEnabled = true;
    }
}
```

As a result of the changes, the button doesn't respond while `AccessTheWebAsync` is downloading the websites, so the process can't be reentered.

Cancel and Restart the Operation

Instead of disabling the **Start** button, you can keep the button active but, if the user chooses that button again, cancel the operation that's already running and let the most recently started operation continue.

For more information about cancellation, see Fine-Tuning Your Async Application (C#).

To set up this scenario, make the following changes to the basic code that is provided in Reviewing and Running the Example App. You also can download the finished app from Async Samples: Reentrancy in .NET Desktop Apps. The name of the project is CancelAndRestart.

1. Declare a CancellationTokenSource variable, `cts`, that's in scope for all methods.

   ```
   public partial class MainWindow : Window   // Or class MainPage
   {
       // *** Declare a System.Threading.CancellationTokenSource.
       CancellationTokenSource cts;
   ```

2. In `StartButton_Click`, determine whether an operation is already underway. If the value of `cts` is null, no operation is already active. If the value isn't null, the operation that is already running is canceled.

   ```
   // *** If a download process is already underway, cancel it.
   if (cts != null)
   {
       cts.Cancel();
   }
   ```

3. Set `cts` to a different value that represents the current process.

```
// *** Now set cts to a new value that you can use to cancel the current process
// if the button is chosen again.
CancellationTokenSource newCTS = new CancellationTokenSource();
cts = newCTS;
```

4. At the end of `StartButton_Click`, the current process is complete, so set the value of `cts` back to null.

```
// *** When the process is complete, signal that another process can begin.
if (cts == newCTS)
    cts = null;
```

The following code shows all the changes in `StartButton_Click`. The additions are marked with asterisks.

```
private async void StartButton_Click(object sender, RoutedEventArgs e)
{
    // This line is commented out to make the results clearer in the output.
    //ResultsTextBox.Clear();

    // *** If a download process is already underway, cancel it.
    if (cts != null)
    {
        cts.Cancel();
    }

    // *** Now set cts to cancel the current process if the button is chosen again.
    CancellationTokenSource newCTS = new CancellationTokenSource();
    cts = newCTS;

    try
    {
        // ***Send cts.Token to carry the message if there is a cancellation request.
        await AccessTheWebAsync(cts.Token);

    }
    // *** Catch cancellations separately.
    catch (OperationCanceledException)
    {
        ResultsTextBox.Text += "\r\nDownloads canceled.\r\n";
    }
    catch (Exception)
    {
        ResultsTextBox.Text += "\r\nDownloads failed.\r\n";
    }
    // *** When the process is complete, signal that another process can proceed.
    if (cts == newCTS)
        cts = null;
}
```

In `AccessTheWebAsync`, make the following changes.

- Add a parameter to accept the cancellation token from `StartButton_Click`.
- Use the GetAsync method to download the websites because `GetAsync` accepts a CancellationToken argument.
- Before calling `DisplayResults` to display the results for each downloaded website, check `ct` to verify that the current operation hasn't been canceled.

The following code shows these changes, which are marked with asterisks.

```
// *** Provide a parameter for the CancellationToken from StartButton_Click.
async Task AccessTheWebAsync(CancellationToken ct)
{
    // Declare an HttpClient object.
    HttpClient client = new HttpClient();

    // Make a list of web addresses.
    List<string> urlList = SetUpURLList();

    var total = 0;
    var position = 0;

    foreach (var url in urlList)
    {
        // *** Use the HttpClient.GetAsync method because it accepts a
        // cancellation token.
        HttpResponseMessage response = await client.GetAsync(url, ct);

        // *** Retrieve the website contents from the HttpResponseMessage.
        byte[] urlContents = await response.Content.ReadAsByteArrayAsync();

        // *** Check for cancellations before displaying information about the
        // latest site.
        ct.ThrowIfCancellationRequested();

        DisplayResults(url, urlContents, ++position);

        // Update the total.
        total += urlContents.Length;
    }

    // Display the total count for all of the websites.
    ResultsTextBox.Text +=
        $"\r\n\r\nTOTAL bytes returned:  {total}\r\n";
}
```

If you choose the **Start** button several times while this app is running, it should produce results that resemble the following output.

```
1. msdn.microsoft.com/library/hh191443.aspx               83732
2. msdn.microsoft.com/library/aa578028.aspx              205273
3. msdn.microsoft.com/library/jj155761.aspx               29019
4. msdn.microsoft.com/library/hh290140.aspx              122505
5. msdn.microsoft.com/library/hh524395.aspx               68959
6. msdn.microsoft.com/library/ms404677.aspx              197325
Download canceled.

1. msdn.microsoft.com/library/hh191443.aspx               83732
2. msdn.microsoft.com/library/aa578028.aspx              205273
3. msdn.microsoft.com/library/jj155761.aspx               29019
Download canceled.

1. msdn.microsoft.com/library/hh191443.aspx               83732
2. msdn.microsoft.com/library/aa578028.aspx              205273
3. msdn.microsoft.com/library/jj155761.aspx               29019
4. msdn.microsoft.com/library/hh290140.aspx              117152
5. msdn.microsoft.com/library/hh524395.aspx               68959
6. msdn.microsoft.com/library/ms404677.aspx              197325
7. msdn.microsoft.com                                           42972
8. msdn.microsoft.com/library/ff730837.aspx              146159

TOTAL bytes returned:  890591
```

To eliminate the partial lists, uncomment the first line of code in `StartButton_Click` to clear the text box each time the user restarts the operation.

Run Multiple Operations and Queue the Output

This third example is the most complicated in that the app starts another asynchronous operation each time that the user chooses the **Start** button, and all the operations run to completion. All the requested operations download websites from the list asynchronously, but the output from the operations is presented sequentially. That is, the actual downloading activity is interleaved, as the output in Recognizing Reentrancy shows, but the list of results for each group is presented separately.

The operations share a global Task, `pendingWork`, which serves as a gatekeeper for the display process.

To set up this scenario, make the following changes to the basic code that is provided in Reviewing and Running the Example App. You also can download the finished app from Async Samples: Reentrancy in .NET Desktop Apps. The name of the project is QueueResults.

The following output shows the result if the user chooses the **Start** button only once. The letter label, A, indicates that the result is from the first time the **Start** button is chosen. The numbers show the order of the URLs in the list of download targets.

```
#Starting group A.
#Task assigned for group A.

A-1. msdn.microsoft.com/library/hh191443.aspx                87389
A-2. msdn.microsoft.com/library/aa578028.aspx               209858
A-3. msdn.microsoft.com/library/jj155761.aspx                30870
A-4. msdn.microsoft.com/library/hh290140.aspx               119027
A-5. msdn.microsoft.com/library/hh524395.aspx                71260
A-6. msdn.microsoft.com/library/ms404677.aspx               199186
A-7. msdn.microsoft.com                                            53266
A-8. msdn.microsoft.com/library/ff730837.aspx               148020

TOTAL bytes returned:  918876

#Group A is complete.
```

If the user chooses the **Start** button three times, the app produces output that resembles the following lines. The information lines that start with a pound sign (#) trace the progress of the application.

```
#Starting group A.
#Task assigned for group A.

A-1. msdn.microsoft.com/library/hh191443.aspx                87389
A-2. msdn.microsoft.com/library/aa578028.aspx               207089
A-3. msdn.microsoft.com/library/jj155761.aspx                30870
A-4. msdn.microsoft.com/library/hh290140.aspx               119027
A-5. msdn.microsoft.com/library/hh524395.aspx                71259
A-6. msdn.microsoft.com/library/ms404677.aspx               199185

#Starting group B.
#Task assigned for group B.

A-7. msdn.microsoft.com                                            53266

#Starting group C.
#Task assigned for group C.

A-8. msdn.microsoft.com/library/ff730837.aspx               148010

TOTAL bytes returned:  916095

B-1. msdn.microsoft.com/library/hh191443.aspx                87389
B-2. msdn.microsoft.com/library/aa578028.aspx               207089
B-3. msdn.microsoft.com/library/jj155761.aspx                30870
B-4. msdn.microsoft.com/library/hh290140.aspx               119027
B-5. msdn.microsoft.com/library/hh524395.aspx                71260
B-6. msdn.microsoft.com/library/ms404677.aspx               199186

#Group A is complete.

B-7. msdn.microsoft.com                                            53266
B-8. msdn.microsoft.com/library/ff730837.aspx               148010

TOTAL bytes returned:  916097

C-1. msdn.microsoft.com/library/hh191443.aspx                87389
C-2. msdn.microsoft.com/library/aa578028.aspx               207089

#Group B is complete.

C-3. msdn.microsoft.com/library/jj155761.aspx                30870
C-4. msdn.microsoft.com/library/hh290140.aspx               119027
C-5. msdn.microsoft.com/library/hh524395.aspx                72765
C-6. msdn.microsoft.com/library/ms404677.aspx               199186
C-7. msdn.microsoft.com                                            56190
C-8. msdn.microsoft.com/library/ff730837.aspx               148010

TOTAL bytes returned:  920526

#Group C is complete.
```

Groups B and C start before group A has finished, but the output for the each group appears separately. All the output for group A appears first, followed by all the output for group B, and then all the output for group C. The app always displays the groups in order and, for each group, always displays the information about the individual websites in the order that the URLs appear in the list of URLs.

However, you can't predict the order in which the downloads actually happen. After multiple groups have been started, the download tasks that they generate are all active. You can't assume that A-1 will be downloaded before B-1, and you can't assume that A-1 will be downloaded before A-2.

Global Definitions

The sample code contains the following two global declarations that are visible from all methods.

```
public partial class MainWindow : Window  // Class MainPage in Windows Store app.
{
    // ***Declare the following variables where all methods can access them.
    private Task pendingWork = null;
    private char group = (char)('A' - 1);
```

The `Task` variable, `pendingWork`, oversees the display process and prevents any group from interrupting another group's display operation. The character variable, `group`, labels the output from different groups to verify that results appear in the expected order.

The Click Event Handler

The event handler, `StartButton_Click`, increments the group letter each time the user chooses the **Start** button. Then the handler calls `AccessTheWebAsync` to run the downloading operation.

```
private async void StartButton_Click(object sender, RoutedEventArgs e)
{
    // ***Verify that each group's results are displayed together, and that
    // the groups display in order, by marking each group with a letter.
    group = (char)(group + 1);
    ResultsTextBox.Text += $"\r\n\r\n#Starting group {group}.";

    try
    {
        // *** Pass the group value to AccessTheWebAsync.
        char finishedGroup = await AccessTheWebAsync(group);

        // The following line verifies a successful return from the download and
        // display procedures.
        ResultsTextBox.Text += $"\r\n\r\n#Group {finishedGroup} is complete.\r\n";
    }
    catch (Exception)
    {
        ResultsTextBox.Text += "\r\nDownloads failed.";
    }
}
```

The AccessTheWebAsync Method

This example splits `AccessTheWebAsync` into two methods. The first method, `AccessTheWebAsync`, starts all the download tasks for a group and sets up `pendingWork` to control the display process. The method uses a Language Integrated Query (LINQ query) and ToArray to start all the download tasks at the same time.

`AccessTheWebAsync` then calls `FinishOneGroupAsync` to await the completion of each download and display its length.

`FinishOneGroupAsync` returns a task that's assigned to `pendingWork` in `AccessTheWebAsync`. That value prevents interruption by another operation before the task is complete.

```
private async Task<char> AccessTheWebAsync(char grp)
{
    HttpClient client = new HttpClient();

    // Make a list of the web addresses to download.
    List<string> urlList = SetUpURLList();

    // ***Kick off the downloads. The application of ToArray activates all the download tasks.
    Task<byte[]>[] getContentTasks = urlList.Select(url => client.GetByteArrayAsync(url)).ToArray();

    // ***Call the method that awaits the downloads and displays the results.
    // Assign the Task that FinishOneGroupAsync returns to the gatekeeper task, pendingWork.
    pendingWork = FinishOneGroupAsync(urlList, getContentTasks, grp);

    ResultsTextBox.Text += $"\r\n#Task assigned for group {grp}. Download tasks are active.\r\n";

    // ***This task is complete when a group has finished downloading and displaying.
    await pendingWork;

    // You can do other work here or just return.
    return grp;
}
```

The FinishOneGroupAsync Method

This method cycles through the download tasks in a group, awaiting each one, displaying the length of the downloaded website, and adding the length to the total.

The first statement in `FinishOneGroupAsync` uses `pendingWork` to make sure that entering the method doesn't interfere with an operation that is already in the display process or that's already waiting. If such an operation is in progress, the entering operation must wait its turn.

```
private async Task FinishOneGroupAsync(List<string> urls, Task<byte[]>[] contentTasks, char grp)
{
    // ***Wait for the previous group to finish displaying results.
    if (pendingWork != null) await pendingWork;

    int total = 0;

    // contentTasks is the array of Tasks that was created in AccessTheWebAsync.
    for (int i = 0; i < contentTasks.Length; i++)
    {
        // Await the download of a particular URL, and then display the URL and
        // its length.
        byte[] content = await contentTasks[i];
        DisplayResults(urls[i], content, i, grp);
        total += content.Length;
    }

    // Display the total count for all of the websites.
    ResultsTextBox.Text +=
        $"\r\n\r\nTOTAL bytes returned:  {total}\r\n";
}
```

Points of Interest

The information lines that start with a pound sign (#) in the output clarify how this example works.

The output shows the following patterns.

- A group can be started while a previous group is displaying its output, but the display of the previous group's output isn't interrupted.

```
#Starting group A.
#Task assigned for group A. Download tasks are active.

A-1. msdn.microsoft.com/library/hh191443.aspx                87389
A-2. msdn.microsoft.com/library/aa578028.aspx               207089
A-3. msdn.microsoft.com/library/jj155761.aspx                30870
A-4. msdn.microsoft.com/library/hh290140.aspx               119037
A-5. msdn.microsoft.com/library/hh524395.aspx                71260

#Starting group B.
#Task assigned for group B. Download tasks are active.

A-6. msdn.microsoft.com/library/ms404677.aspx               199186
A-7. msdn.microsoft.com                                            53078
A-8. msdn.microsoft.com/library/ff730837.aspx               148010

TOTAL bytes returned:  915919

B-1. msdn.microsoft.com/library/hh191443.aspx                87388
B-2. msdn.microsoft.com/library/aa578028.aspx               207089
B-3. msdn.microsoft.com/library/jj155761.aspx                30870

#Group A is complete.

B-4. msdn.microsoft.com/library/hh290140.aspx               119027
B-5. msdn.microsoft.com/library/hh524395.aspx                71260
B-6. msdn.microsoft.com/library/ms404677.aspx               199186
B-7. msdn.microsoft.com                                            53078
B-8. msdn.microsoft.com/library/ff730837.aspx               148010

TOTAL bytes returned:  915908
```

- The `pendingWork` task is null at the start of `FinishOneGroupAsync` only for group A, which started first. Group A hasn't yet completed an await expression when it reaches `FinishOneGroupAsync`. Therefore, control hasn't returned to `AccessTheWebAsync`, and the first assignment to `pendingWork` hasn't occurred.

- The following two lines always appear together in the output. The code is never interrupted between starting a group's operation in `StartButton_Click` and assigning a task for the group to `pendingWork`.

  ```
  #Starting group B.
  #Task assigned for group B. Download tasks are active.
  ```

  After a group enters `StartButton_Click`, the operation doesn't complete an await expression until the operation enters `FinishOneGroupAsync`. Therefore, no other operation can gain control during that segment of code.

Reviewing and Running the Example App

To better understand the example app, you can download it, build it yourself, or review the code at the end of this topic without implementing the app.

> **NOTE**
>
> To run the example as a Windows Presentation Foundation (WPF) desktop app, you must have Visual Studio 2012 or newer and the .NET Framework 4.5 or newer installed on your computer.

Downloading the App

1. Download the compressed file from Async Samples: Reentrancy in .NET Desktop Apps.

2. Decompress the file that you downloaded, and then start Visual Studio.

3. On the menu bar, choose **File**, **Open**, **Project/Solution**.

4. Navigate to the folder that holds the decompressed sample code, and then open the solution (.sln) file.

5. In **Solution Explorer**, open the shortcut menu for the project that you want to run, and then choose **Set as StartUpProject**.

6. Choose the CTRL+F5 keys to build and run the project.

Building the App

The following section provides the code to build the example as a WPF app.

To build a WPF app

1. Start Visual Studio.

2. On the menu bar, choose **File**, **New**, **Project**.

   The **New Project** dialog box opens.

3. In the **Installed Templates** pane, expand **Visual C#**, and then expand **Windows**.

4. In the list of project types, choose **WPF Application**.

5. Name the project `WebsiteDownloadWPF`, and then choose the **OK** button.

   The new project appears in **Solution Explorer**.

6. 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**.

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

```
<Window x:Class="WebsiteDownloadWPF.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:WebsiteDownloadWPF"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Grid Width="517" Height="360">
        <Button x:Name="StartButton" Content="Start" HorizontalAlignment="Left" Margin="-1,0,0,0"
VerticalAlignment="Top" Click="StartButton_Click" Height="53" Background="#FFA89B9B" FontSize="36"
Width="518"  />
        <TextBox x:Name="ResultsTextBox" HorizontalAlignment="Left" Margin="-1,53,0,-36"
TextWrapping="Wrap" VerticalAlignment="Top" Height="343" FontSize="10"
ScrollViewer.VerticalScrollBarVisibility="Visible" Width="518" FontFamily="Lucida Console" />
    </Grid>
</Window>
```

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

8. Add a reference for System.Net.Http.

9. In **Solution Explorer**, open the shortcut menu for MainWindow.xaml.cs, and then choose **View Code**.

10. In MainWindow.xaml.cs, replace the code with the following code.

```
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.Threading;

namespace WebsiteDownloadWPF
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private async void StartButton_Click(object sender, RoutedEventArgs e)
        {
            // This line is commented out to make the results clearer in the output.
            //ResultsTextBox.Text = "";

            try
            {
                await AccessTheWebAsync();
            }
            catch (Exception)
            {
                ResultsTextBox.Text += "\r\nDownloads failed.";
            }
        }

        private async Task AccessTheWebAsync()
        {
            // Declare an HttpClient object.
            HttpClient client = new HttpClient();

            // Make a list of web addresses.
            List<string> urlList = SetUpURLList();

            var total = 0;
            var position = 0;

            foreach (var url in urlList)
            {
                // GetByteArrayAsync returns a task. At completion, the task
                // produces a byte array.
                byte[] urlContents = await client.GetByteArrayAsync(url);

                DisplayResults(url, urlContents, ++position);

                // Update the total.
                total += urlContents.Length;
            }

            // Display the total count for all of the websites.
            ResultsTextBox.Text +=
                $"\r\n\r\nTOTAL bytes returned:  {total}\r\n";
        }
        private List<string> SetUpURLList()
        {
            List<string> urls = new List<string>
            {
                "https://msdn.microsoft.com/library/hh191443.aspx",
                "https://msdn.microsoft.com/library/aa578028.aspx",
                "https://msdn.microsoft.com/library/jj155761.aspx",
                "https://msdn.microsoft.com/library/hh290140.aspx",
                "https://msdn.microsoft.com/library/hh524395.aspx",
                "https://msdn.microsoft.com/library/ms404677.aspx",
                "https://msdn.microsoft.com",
                "https://msdn.microsoft.com/library/ff730837.aspx"
            };
            return urls;
        }

        private void DisplayResults(string url, byte[] content, int pos)
        {
            // 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.

            // Strip off the "https://".
            var displayURL = url.Replace("https://", "");
            // Display position in the URL list, the URL, and the number of bytes.
            ResultsTextBox.Text += $"\n{pos}. {displayURL,-58} {content.Length,8}";
        }
    }
}
```

11. Choose the CTRL+F5 keys to run the program, and then choose the **Start** button several times.

12. Make the changes from Disable the Start Button, Cancel and Restart the Operation, or Run Multiple Operations and Queue the Output to handle the reentrancy.

See also

- Walkthrough: Accessing the Web by Using async and await (C#)
- Asynchronous Programming with async and await (C#)

Using Async for File Access (C#)

4/28/2019 • 5 minutes to read • Edit Online

You can use the async feature to access files. By using the async feature, you can call into asynchronous methods without using callbacks or splitting your code across multiple methods or lambda expressions. To make synchronous code asynchronous, you just call an asynchronous method instead of a synchronous method and add a few keywords to the code.

You might consider the following reasons for adding asynchrony to file access calls:

- Asynchrony makes UI applications more responsive because the UI thread that launches the operation can perform other work. If the UI thread must execute code that takes a long time (for example, more than 50 milliseconds), the UI may freeze until the I/O is complete and the UI thread can again process keyboard and mouse input and other events.
- Asynchrony improves the scalability of ASP.NET and other server-based applications by reducing the need for threads. If the application uses a dedicated thread per response and a thousand requests are being handled simultaneously, a thousand threads are needed. Asynchronous operations often don't need to use a thread during the wait. They use the existing I/O completion thread briefly at the end.
- The latency of a file access operation might be very low under current conditions, but the latency may greatly increase in the future. For example, a file may be moved to a server that's across the world.
- The added overhead of using the Async feature is small.
- Asynchronous tasks can easily be run in parallel.

Running the Examples

To run the examples in this topic, you can create a **WPF Application** or a **Windows Forms Application** and then add a **Button**. In the button's `Click` event, add a call to the first method in each example.

In the following examples, include the following `using` statements.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Text;
using System.Threading.Tasks;
```

Use of the FileStream Class

The examples in this topic use the FileStream class, which has an option that causes asynchronous I/O to occur at the operating system level. By using this option, you can avoid blocking a ThreadPool thread in many cases. To enable this option, you specify the `useAsync=true` or `options=FileOptions.Asynchronous` argument in the constructor call.

You can't use this option with StreamReader and StreamWriter if you open them directly by specifying a file path. However, you can use this option if you provide them a Stream that the FileStream class opened. Note that asynchronous calls are faster in UI apps even if a ThreadPool thread is blocked, because the UI thread isn't blocked during the wait.

Writing Text

The following example writes text to a file. At each await statement, the method immediately exits. When the file I/O is complete, the method resumes at the statement that follows the await statement. Note that the async modifier is in the definition of methods that use the await statement.

```
public async void ProcessWrite()
{
    string filePath = @"temp2.txt";
    string text = "Hello World\r\n";

    await WriteTextAsync(filePath, text);
}

private async Task WriteTextAsync(string filePath, string text)
{
    byte[] encodedText = Encoding.Unicode.GetBytes(text);

    using (FileStream sourceStream = new FileStream(filePath,
        FileMode.Append, FileAccess.Write, FileShare.None,
        bufferSize: 4096, useAsync: true))
    {
        await sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
    };
}
```

The original example has the statement `await sourceStream.WriteAsync(encodedText, 0, encodedText.Length);`, which is a contraction of the following two statements:

```
Task theTask = sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
await theTask;
```

The first statement returns a task and causes file processing to start. The second statement with the await causes the method to immediately exit and return a different task. When the file processing later completes, execution returns to the statement that follows the await. For more information, see Control Flow in Async Programs (C#).

Reading Text

The following example reads text from a file. The text is buffered and, in this case, placed into a StringBuilder. Unlike in the previous example, the evaluation of the await produces a value. The ReadAsync method returns a Task<Int32>, so the evaluation of the await produces an `Int32` value (`numRead`) after the operation completes. For more information, see Async Return Types (C#).

```
public async void ProcessRead()
{
    string filePath = @"temp2.txt";

    if (File.Exists(filePath) == false)
    {
        Debug.WriteLine("file not found: " + filePath);
    }
    else
    {
        try
        {
            string text = await ReadTextAsync(filePath);
            Debug.WriteLine(text);
        }
        catch (Exception ex)
        {
            Debug.WriteLine(ex.Message);
        }
    }
}

private async Task<string> ReadTextAsync(string filePath)
{
    using (FileStream sourceStream = new FileStream(filePath,
        FileMode.Open, FileAccess.Read, FileShare.Read,
        bufferSize: 4096, useAsync: true))
    {
        StringBuilder sb = new StringBuilder();

        byte[] buffer = new byte[0x1000];
        int numRead;
        while ((numRead = await sourceStream.ReadAsync(buffer, 0, buffer.Length)) != 0)
        {
            string text = Encoding.Unicode.GetString(buffer, 0, numRead);
            sb.Append(text);
        }

        return sb.ToString();
    }
}
```

Parallel Asynchronous I/O

The following example demonstrates parallel processing by writing 10 text files. For each file, the WriteAsync method returns a task that is then added to a list of tasks. The `await Task.WhenAll(tasks);` statement exits the method and resumes within the method when file processing is complete for all of the tasks.

The example closes all FileStream instances in a `finally` block after the tasks are complete. If each `FileStream` was instead created in a `using` statement, the `FileStream` might be disposed of before the task was complete.

Note that any performance boost is almost entirely from the parallel processing and not the asynchronous processing. The advantages of asynchrony are that it doesn't tie up multiple threads, and that it doesn't tie up the user interface thread.

```csharp
public async void ProcessWriteMult()
{
    string folder = @"tempfolder\";
    List<Task> tasks = new List<Task>();
    List<FileStream> sourceStreams = new List<FileStream>();

    try
    {
        for (int index = 1; index <= 10; index++)
        {
            string text = "In file " + index.ToString() + "\r\n";

            string fileName = "thefile" + index.ToString("00") + ".txt";
            string filePath = folder + fileName;

            byte[] encodedText = Encoding.Unicode.GetBytes(text);

            FileStream sourceStream = new FileStream(filePath,
                FileMode.Append, FileAccess.Write, FileShare.None,
                bufferSize: 4096, useAsync: true);

            Task theTask = sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
            sourceStreams.Add(sourceStream);

            tasks.Add(theTask);
        }

        await Task.WhenAll(tasks);
    }

    finally
    {
        foreach (FileStream sourceStream in sourceStreams)
        {
            sourceStream.Close();
        }
    }
}
```

When using the WriteAsync and ReadAsync methods, you can specify a CancellationToken, which you can use to cancel the operation mid-stream. For more information, see Fine-Tuning Your Async Application (C#) and Cancellation in Managed Threads.

See also

- Asynchronous Programming with async and await (C#)
- Async Return Types (C#)
- Control Flow in Async Programs (C#)