Asynchronous Programming with Async and Await (Visual Basic)

Visual Studio 2015

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

Visual Studio 2012 introduced a simplified approach, async programming, that leverages asynchronous support in the .NET Framework 4.5 and higher as well as in 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 when your application accesses the web. Access to a web resource sometimes is slow or delayed. If such an activity is blocked within a synchronous process, the entire application must wait. In an asynchronous process, the application can continue with other work that doesn't depend on the web resource until the potentially blocking task finishes.

The following table shows typical areas where asynchronous programming improves responsiveness. The listed APIs from the .NET Framework 4.5 and the Windows Runtime contain methods that support async programming.

Application area	Supporting APIs that contain async methods
Web access	HttpClient, SyndicationClient
Working with files	StorageFile, StreamWriter, StreamReader, XmlReader
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 Visual Basic are the heart of async programming. By using those two keywords, you can use resources in the .NET Framework or the Windows Runtime to create an asynchronous method almost as easily as you create a synchronous method. Asynchronous methods that you define by using **Async** and **Await** are referred to as async methods.

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

You can find 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".

VB

```
' Three things to note in the signature:
'  - The method has an Async modifier.
'  - The return type is Task or Task(Of T). (See "Return Types" section.)
'    Here, it is Task(Of Integer) because the return statement returns an integer.
'  - The method name ends in "Async."
Async Function AccessTheWebAsync() As Task(Of Integer)

    ' You need to add a reference to System.Net.Http to declare client.
    Dim client As HttpClient = New HttpClient()

    ' GetStringAsync returns a Task(Of String). That means that when you await the
    ' task you'll get a string (urlContents).
    Dim getStringTask As Task(Of String) =
client.GetStringAsync("http://msdn.microsoft.com")


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

    ' The Await operator suspends AccessTheWebAsync.
    '  - AccessTheWebAsync can't continue until getStringTask is complete.
    '  - Meanwhile, control returns to the caller of AccessTheWebAsync.
    '  - Control resumes here when getStringTask is complete.
    '  - The Await operator then retrieves the string result from getStringTask.
    Dim urlContents As String = Await getStringTask

    ' The return statement specifies an integer result.
    ' Any methods that are awaiting AccessTheWebAsync retrieve the length value.
    Return urlContents.Length
End Function
```

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.

VB

```
Dim urlContents As String = Await client.GetStringAsync()
```

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(Of 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.
  - Sub if you're writing an async event handler.

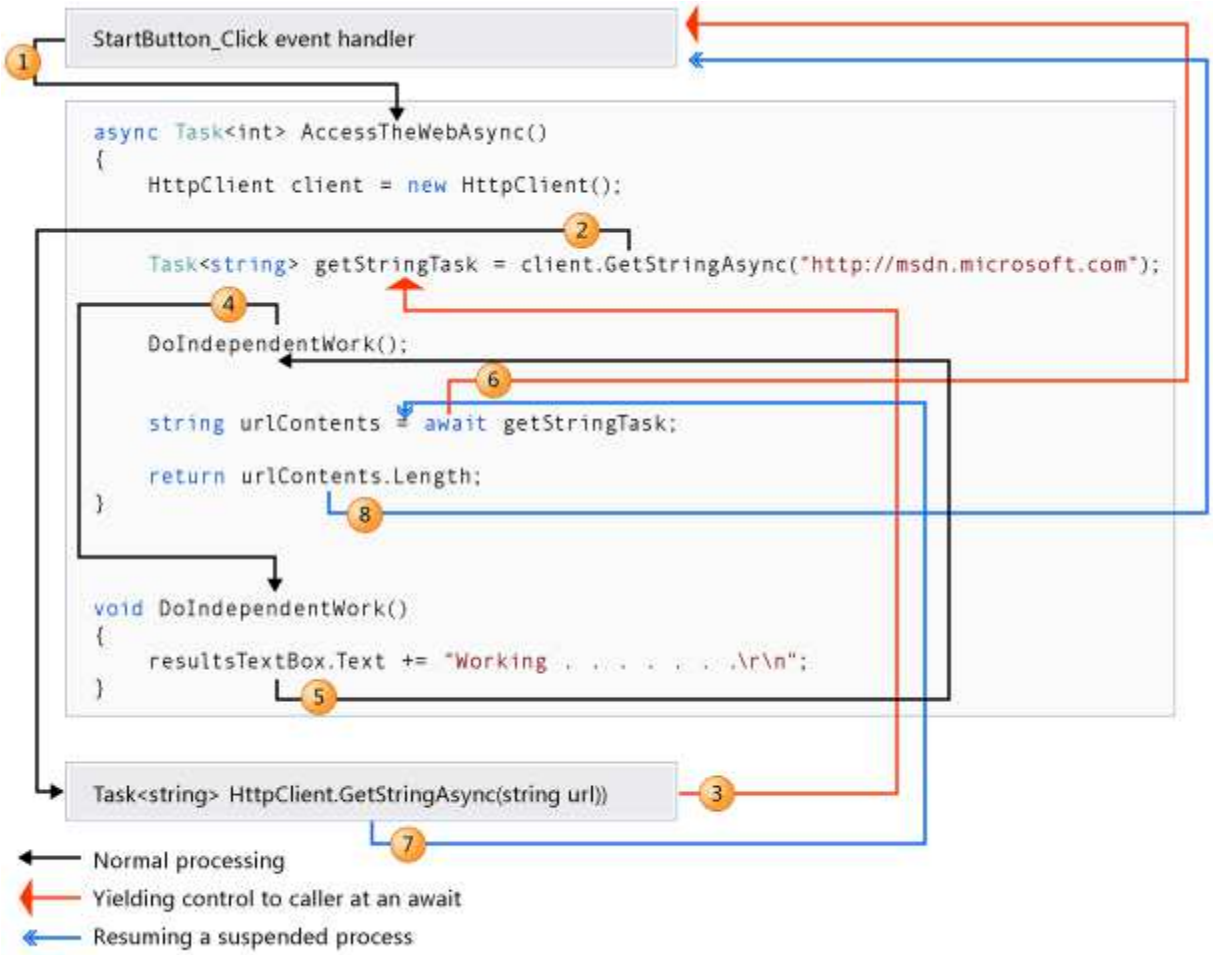
  For more information, see "Return Types and Parameters" later in this topic.
- The method usually includes at least one await expression, which marks a point where the method can't continue until the awaited asynchronous operation is complete. In the meantime, the method is suspended, and control returns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

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

For more information about asynchrony in previous versions of the .NET Framework, see TPL and Traditional .NET Framework Asynchronous Programming.

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.



The numbers in the diagram correspond to the following steps.

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

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

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

   GetStringAsync returns a Task(Of 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>** (**Task(Of Integer)** in Visual Basic) 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`) is complete before `AccessTheWebAsync` awaits it, control remains in `AccessTheWebAsync`. The expense of suspending and then returning to `AccessTheWebAsync` would be wasted if the called asynchronous process (`getStringTask`) has already completed and AccessTheWebSync doesn't have to wait for the final result.

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

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 (Visual Basic).

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 contains many members that work with **Async** and **Await**. You can recognize these members by the "Async" suffix that's attached to the member name and a return type of Task or Task(Of 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 and example methods, see Quickstart: using the await operator for asynchronous programming, Asynchronous programming (Windows Store apps), and WhenAny: Bridging between the .NET Framework and the Windows Runtime (Visual Basic).

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 BackgroundWorker for IO-bound operations because the code is simpler and you don't have to guard against race conditions. In combination with Task.Run, 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 an 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 (Visual Basic)
- Await Operator (Visual Basic)

Return Types and Parameters

In .NET Framework programming, an async method typically returns a Task or a Task(Of 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(Of 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.

The following example shows how you declare and call a method that returns a Task(Of TResult) or a Task.

VB

```
' Signature specifies Task(Of Integer)
Async Function TaskOfTResult_MethodAsync() As Task(Of Integer)

    Dim hours As Integer
    ' . . .
    ' Return statement specifies an integer result.
    Return hours
End Function

' Calls to TaskOfTResult_MethodAsync
Dim returnedTaskTResult As Task(Of Integer) = TaskOfTResult_MethodAsync()
Dim intResult As Integer = Await returnedTaskTResult
' or, in a single statement
Dim intResult As Integer = Await TaskOfTResult_MethodAsync()


' Signature specifies Task
Async Function Task_MethodAsync() As Task

    ' . . .
    ' The method has no return statement.
End Function

' Calls to Task_MethodAsync
Task returnedTask = Task_MethodAsync()
Await returnedTask
' or, in a single statement
Await Task_MethodAsync()
```

Each returned task represents ongoing work. A task encapsulates information about the state of the asynchronous

process and, eventually, either the final result from the process or the exception that the process raises if it doesn't succeed.

An async method can also be a **Sub** method. This return type is used primarily to define event handlers, where a return type is required. Async event handlers often serve as the starting point for async programs.

An async method that's a **Sub** procedure can't be awaited, and the caller can't catch any exceptions that the method throws.

An async method can't declare ByRef parameters, but the method can call methods that have such parameters.

For more information and examples, see Async Return Types (Visual Basic). For more information about how to catch exceptions in async methods, see Try...Catch...Finally Statement (Visual Basic).

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

- IAsyncOperation, which corresponds to Task(Of TResult)
- IAsyncAction, which corresponds to Task
- IAsyncActionWithProgress
- IAsyncOperationWithProgress

For more information and an example, see Quickstart: using the await operator for asynchronous programming.

Naming Convention

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

You can ignore the convention where an event, base class, or interface contract suggests a different name. 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 (Visual Basic)	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 (Visual Basic)	Adds Task.WhenAll to the previous walkthrough. The use of **WhenAll** starts all the downloads at the same time.	
How to: Make Multiple Web	Demonstrates how to start several tasks at the same	Async Sample: Make

Requests in Parallel by Using Async and Await (Visual Basic)	time.	Multiple Web Requests in Parallel
Async Return Types (Visual Basic)	Illustrates the types that async methods can return and explains when each type is appropriate.	
Control Flow in Async Programs (Visual Basic)	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 (Visual Basic)	Shows how to add the following functionality to your async solution: • Cancel an Async Task or a List of Tasks (Visual Basic) • Cancel Async Tasks after a Period of Time (Visual Basic) • Cancel Remaining Async Tasks after One Is Complete (Visual Basic) • Start Multiple Async Tasks and Process Them As They Complete (Visual Basic)	Async Sample: Fine Tuning Your Application
Handling Reentrancy in Async Apps (Visual Basic)	Shows how to handle cases in which an active asynchronous operation is restarted while it's running.	
WhenAny: Bridging between the .NET Framework and the Windows Runtime (Visual Basic)	Shows how to bridge between Task types in the .NET Framework and IAsyncOperations in the Windows Runtime so that you can use WhenAny(Of TResult) 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 (Visual Basic)	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 (Visual Basic)	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(Of 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.vb 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".

VB

```
' Add an Imports statement and a reference for System.Net.Http
Imports System.Net.Http

Class MainWindow

    ' Mark the event handler with async so you can use Await in it.
    Private Async Sub StartButton_Click(sender As Object, e As RoutedEventArgs)

        ' Call and await separately.
        'Task<int> getLengthTask = AccessTheWebAsync();
        '' You can do independent work here.
        'int contentLength = await getLengthTask;

        Dim contentLength As Integer = Await AccessTheWebAsync()

        ResultsTextBox.Text &=
            String.Format(vbCrLf & "Length of the downloaded string: {0}." & vbCrLf,
contentLength)
    End Sub


    ' Three things to note in the signature:
    '  - The method has an Async modifier.
    '  - The return type is Task or Task(Of T). (See "Return Types" section.)
    '    Here, it is Task(Of Integer) because the return statement returns an integer.
    '  - The method name ends in "Async."
    Async Function AccessTheWebAsync() As Task(Of Integer)

        ' You need to add a reference to System.Net.Http to declare client.
        Dim client As HttpClient = New HttpClient()

        ' GetStringAsync returns a Task(Of String). That means that when you await the
        ' task you'll get a string (urlContents).
        Dim getStringTask As Task(Of String) =
client.GetStringAsync("http://msdn.microsoft.com")


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

        ' The Await operator suspends AccessTheWebAsync.
        '  - AccessTheWebAsync can't continue until getStringTask is complete.
        '  - Meanwhile, control returns to the caller of AccessTheWebAsync.
        '  - Control resumes here when getStringTask is complete.
        '  - The Await operator then retrieves the string result from getStringTask.
        Dim urlContents As String = Await getStringTask

        ' The return statement specifies an integer result.
```

```
        ' Any methods that are awaiting AccessTheWebAsync retrieve the length value.
        Return urlContents.Length
    End Function


    Sub DoIndependentWork()
        ResultsTextBox.Text &= "Working . . . . . ." & vbCrLf
    End Sub
End Class

' Sample Output:

' Working . . . . . .

' Length of the downloaded string: 41763.
```

See Also

Await Operator (Visual Basic)
Async (Visual Basic)

Await Operator (Visual Basic)

Visual Studio 2015

You apply the **Await** operator to an operand in an asynchronous method or lambda expression to suspend execution of the method until the awaited task completes. The task represents ongoing work.

The method in which **Await** is used must have an Async modifier. Such a method, defined by using the **Async** modifier, and usually containing one or more **Await** expressions, is referred to as an *async method*.

Note

The **Async** and **Await** keywords were introduced in Visual Studio 2012. For an introduction to async programming, see Asynchronous Programming with Async and Await (C# and Visual Basic).

Typically, the task to which you apply the **Await** operator is the return value from a call to a method that implements the Task-Based Asynchronous Pattern, that is, a Task or a Task(Of TResult).

In the following code, the HttpClient method GetByteArrayAsync returns `getContentsTask`, a **Task(Of Byte())**. The task is a promise to produce the actual byte array when the operation is complete. The **Await** operator is applied to `getContentsTask` to suspend execution in `SumPageSizesAsync` until `getContentsTask` is complete. In the meantime, control is returned to the caller of `SumPageSizesAsync`. When `getContentsTask` is finished, the **Await** expression evaluates to a byte array.

VB

```
Private Async Function SumPageSizesAsync() As Task

    ' To use the HttpClient type in desktop apps, you must include a using directive and
add a
    ' reference for the System.Net.Http namespace.
    Dim client As HttpClient = New HttpClient()
    ' . . .
    Dim getContentsTask As Task(Of Byte()) = client.GetByteArrayAsync(url)
    Dim urlContents As Byte() = Await getContentsTask

    ' Equivalently, now that you see how it works, you can write the same thing in a
single line.
    'Dim urlContents As Byte() = Await client.GetByteArrayAsync(url)
    ' . . .
End Function
```

Important

For the complete example, see Walkthrough: Accessing the Web by Using Async and Await (C# and Visual Basic). You can download the sample from Developer Code Samples on the Microsoft website. The example is in the AsyncWalkthrough_HttpClient project.

If **Await** is applied to the result of a method call that returns a **Task(Of TResult)**, the type of the **Await** expression is TResult. If **Await** is applied to the result of a method call that returns a **Task**, the **Await** expression doesn't return a value. The following example illustrates the difference.

VB

```
' Await used with a method that returns a Task(Of TResult).
Dim result As TResult = Await AsyncMethodThatReturnsTaskTResult()

' Await used with a method that returns a Task.
Await AsyncMethodThatReturnsTask()
```

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

An **Await** expression can occur only in the body of an immediately enclosing method or lambda expression that is marked by an **Async** modifier. The term *Await* serves as a keyword only in that context. Elsewhere, it is interpreted as an identifier. Within the async method or lambda expression, an **Await** expression cannot occur in a query expression, in the **catch** or **finally** block of a Try...Catch...Finally statement, in the loop control variable expression of a **For** or **For Each** loop, or in the body of a SyncLock statement.

Exceptions

Most async methods return a Task or Task(Of TResult). The properties of the returned task carry information about its status and history, such as whether the task is complete, whether the async method caused an exception or was canceled, and what the final result is. The **Await** operator accesses those properties.

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

If you await a task-returning async method that is canceled, the **Await** operator rethrows an OperationCanceledException.

A single task that is in a faulted state can reflect multiple exceptions. For example, the task might be the result of a call to Task.WhenAll. When you await such a task, the await operation rethrows only one of the exceptions. However, you can't predict which of the exceptions is rethrown.

For examples of error handling in async methods, see Try...Catch...Finally Statement (Visual Basic).

Example

The following Windows Forms example illustrates the use of **Await** in an async method, `WaitAsynchronouslyAsync`.

Contrast the behavior of that method with the behavior of `WaitSynchronously`. Without an **Await** operator, `WaitSynchronously` runs synchronously despite the use of the **Async** modifier in its definition and a call to Thread.Sleep in its body.

VB

```
Private Async Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    ' Call the method that runs asynchronously.
    Dim result As String = Await WaitAsynchronouslyAsync()

    ' Call the method that runs synchronously.
    'Dim result As String = Await WaitSynchronously()

    ' Display the result.
    TextBox1.Text &= result
End Sub

' The following method runs asynchronously. The UI thread is not
' blocked during the delay. You can move or resize the Form1 window
' while Task.Delay is running.
Public Async Function WaitAsynchronouslyAsync() As Task(Of String)
    Await Task.Delay(10000)
    Return "Finished"
End Function

' The following method runs synchronously, despite the use of Async.
' You cannot move or resize the Form1 window while Thread.Sleep
' is running because the UI thread is blocked.
Public Async Function WaitSynchronously() As Task(Of String)
    ' Import System.Threading for the Sleep method.
    Thread.Sleep(10000)
    Return "Finished"
End Function
```

See Also

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

Async (Visual Basic)

Visual Studio 2015

The **Async** modifier indicates that the method or lambda expression that it modifies is asynchronous. Such methods are referred to as *async methods*.

An async method provides a convenient way to do potentially long-running work without blocking the caller's thread. The caller of an async method can resume its work without waiting for the async method to finish.

Note

The **Async** and **Await** keywords were introduced in Visual Studio 2012. For an introduction to async programming, see Asynchronous Programming with Async and Await (C# and Visual Basic).

The following example shows the structure of an async method. By convention, async method names end in "Async."

VB

```
Public Async Function ExampleMethodAsync() As Task(Of Integer)
    ' . . .

    ' At the Await expression, execution in this method is suspended and,
    ' if AwaitedProcessAsync has not already finished, control returns
    ' to the caller of ExampleMethodAsync. When the awaited task is
    ' completed, this method resumes execution.
    Dim exampleInt As Integer = Await AwaitedProcessAsync()

    ' . . .

    ' The return statement completes the task. Any method that is
    ' awaiting ExampleMethodAsync can now get the integer result.
    Return exampleInt
End Function
```

Typically, a method modified by the **Async** keyword contains at least one Await expression or statement. The method runs synchronously until it reaches the first **Await**, at which point it suspends until the awaited task completes. In the meantime, control is returned to the caller of the method. If the method doesn't contain an **Await** expression or statement, the method isn't suspended and executes as a synchronous method does. A compiler warning alerts you to any async methods that don't contain **Await** because that situation might indicate an error. For more information, see the compiler error.

The **Async** keyword is an unreserved keyword. It is a keyword when it modifies a method or a lambda expression. In all other contexts, it is interpreted as an identifier.

Return Types

An async method is either a Sub procedure, or a Function procedure that has a return type of Task or Task(Of TResult). The method cannot declare any ByRef parameters.

You specify **Task(Of TResult)** for the return type of an async method if the Return statement of the method has an operand of type TResult. You use **Task** if no meaningful value is returned when the method is completed. That is, a call to the method returns a **Task**, but when the **Task** is completed, any **Await** statement that's awaiting the **Task** doesn't produce a result value.

Async subroutines are used primarily to define event handlers where a **Sub** procedure is required. The caller of an async subroutine can't await it and can't catch exceptions that the method throws.

For more information and examples, see Async Return Types (C# and Visual Basic).

Example

The following examples show an async event handler, an async lambda expression, and an async method. For a full example that uses these elements, see Walkthrough: Accessing the Web by Using Async and Await (C# and Visual Basic). You can download the walkthrough code from Developer Code Samples.

VB

```
' An event handler must be a Sub procedure.
Async Sub button1_Click(sender As Object, e As RoutedEventArgs) Handles button1.Click
    textBox1.Clear()
    ' SumPageSizesAsync is a method that returns a Task.
    Await SumPageSizesAsync()
    textBox1.Text = vbCrLf & "Control returned to button1_Click."
End Sub


' The following async lambda expression creates an equivalent anonymous
' event handler.
AddHandler button1.Click, Async Sub(sender, e)
                              textBox1.Clear()
                              ' SumPageSizesAsync is a method that returns a Task.
                              Await SumPageSizesAsync()
                              textBox1.Text = vbCrLf & "Control returned to
button1_Click."
                          End Sub


' The following async method returns a Task(Of T).
' A typical call awaits the Byte array result:
'      Dim result As Byte() = Await GetURLContents("http://msdn.com")
Private Async Function GetURLContentsAsync(url As String) As Task(Of Byte())

    ' The downloaded resource ends up in the variable named content.
    Dim content = New MemoryStream()
```

```
    ' Initialize an HttpWebRequest for the current URL.
    Dim webReq = CType(WebRequest.Create(url), HttpWebRequest)

    ' Send the request to the Internet resource and wait for
    ' the response.
    Using response As WebResponse = Await webReq.GetResponseAsync()
        ' Get the data stream that is associated with the specified URL.
        Using responseStream As Stream = response.GetResponseStream()
            ' Read the bytes in responseStream and copy them to content.
            ' CopyToAsync returns a Task, not a Task<T>.
            Await responseStream.CopyToAsync(content)
        End Using
    End Using

    ' Return the result as a byte array.
    Return content.ToArray()
End Function
```

See Also

AsyncStateMachineAttribute
Await Operator (Visual Basic)
Asynchronous Programming with Async and Await (C# and Visual Basic)
Walkthrough: Accessing the Web by Using Async and Await (C# and Visual Basic)

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

Visual Studio 2015

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

For more information about the Async feature, see Asynchronous Programming with Async and Await (Visual Basic).

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

If you don't want to build the applications yourself, you can download "Async Sample: Accessing the Web Walkthrough (C# and Visual Basic)" from Developer Code Samples.

In this walkthrough, you complete the following tasks:

Prerequisites

Visual Studio 2012 or later must be installed on your computer. For more information, see the Microsoft website.

To 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 Basic, 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**.

To design a simple WPF MainWindow

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

   For more information about the WPF XAML Designer, see Creating a UI by using XAML Designer in Visual Studio.

To add a reference

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

   The **Reference Manager** dialog box appears.
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** area, 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.

To add necessary Imports statements

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

VB

```
Imports System.Net.Http
Imports System.Net
Imports System.IO
```

To create a synchronous application

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

VB

```
resultsTextBox.Clear()
SumPageSizes()
resultsTextBox.Text &= vbCrLf & "Control 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.vb:

VB

```
Private Sub SumPageSizes()
```

```
    ' Make a list of web addresses.
    Dim urlList As List(Of String) = SetUpURLList()

    Dim total = 0
    For Each url In urlList
        ' GetURLContents returns the contents of url as a byte array.
        Dim urlContents As Byte() = GetURLContents(url)

        DisplayResults(url, urlContents)

        ' Update the total.
        total += urlContents.Length
    Next

    ' Display the total count for all of the web addresses.
    resultsTextBox.Text &= String.Format(vbCrLf & vbCrLf & "Total bytes returned:
{0}" & vbCrLf, total)
End Sub


Private Function SetUpURLList() As List(Of String)

    Dim urls = New List(Of String) From
        {
            "http://msdn.microsoft.com/library/windows/apps/br211380.aspx",
            "http://msdn.microsoft.com",
            "http://msdn.microsoft.com/en-us/library/hh290136.aspx",
            "http://msdn.microsoft.com/en-us/library/ee256749.aspx",
            "http://msdn.microsoft.com/en-us/library/hh290138.aspx",
            "http://msdn.microsoft.com/en-us/library/hh290140.aspx",
            "http://msdn.microsoft.com/en-us/library/dd470362.aspx",
            "http://msdn.microsoft.com/en-us/library/aa578028.aspx",
            "http://msdn.microsoft.com/en-us/library/ms404677.aspx",
            "http://msdn.microsoft.com/en-us/library/ff730837.aspx"
        }
    Return urls
End Function


Private Function GetURLContents(url As String) As Byte()

    ' The downloaded resource ends up in the variable named content.
    Dim content = New MemoryStream()

    ' Initialize an HttpWebRequest for the current URL.
    Dim webReq = CType(WebRequest.Create(url), HttpWebRequest)

    ' 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 response As WebResponse = webReq.GetResponse()
        ' Get the data stream that is associated with the specified URL.
        Using responseStream As Stream = response.GetResponseStream()
```

```
            ' Read the bytes in responseStream and copy them to content.
            responseStream.CopyTo(content)
        End Using
    End Using

    ' Return the result as a byte array.
    Return content.ToArray()
End Function


Private Sub DisplayResults(url As String, content As Byte())

    ' 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.
    Dim bytes = content.Length
    ' Strip off the "http://".
    Dim displayURL = url.Replace("http://", "")
    resultsTextBox.Text &= String.Format(vbCrLf & "{0,-58} {1,8}", displayURL, bytes)
End Sub
```

To test the synchronous solution

1. 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.

To convert GetURLContents to an asynchronous method

1. To convert the synchronous solution to an asynchronous solution, the best place to start is in `GetURLContents` because the calls to the HttpWebRequest 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.

   VB

   ```
   Using response As WebResponse = webReq.GetResponseAsync()
   ```

2. **GetResponseAsync** returns a Task(Of 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.

   VB

   ```
   Using response As WebResponse = 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.

   VB

   ```
   'Dim responseTask As Task(Of WebResponse) = webReq.GetResponseAsync()
   'Using response As WebResponse = 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 (Visual Basic) and How to: Extend the Async Walkthrough by Using Task.WhenAll (Visual Basic).

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.

     VB

     ```
     Await responseStream.CopyToAsync(content)
     ```

     The previous statement abbreviates the following two lines of code.

     VB

     ```
     ' CopyToAsync returns a Task, not a Task<T>.
     'Dim copyTask As Task = 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.

   VB

   ```
   Private Async Function GetURLContents(url As String) As Byte()
   ```

5. The return type of an async method can only be Task, Task(Of TResult). In Visual Basic, the method must be a **Function** that returns a **Task** or a **Task(Of T)**, or the method must be a **Sub**. Typically, a **Sub** method is used only in an async event handler, where **Sub** 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.

   For more information, see Async Return Types (Visual Basic).

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

   - Change the return type to `Task(Of Byte())`.

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

The following code shows these changes.

VB

```
Private Async Function GetURLContentsAsync(url As String) As Task(Of Byte())
```

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

To convert SumPageSizes to an asynchronous method

1. Repeat the steps from the previous procedure for `SumPageSizes`. First, change the call to `GetURLContents` to an asynchronous call.

   - Change the name of the method that's called from `GetURLContents` to `GetURLContentsAsync`, if you haven't already done so.
   - Apply **Await** to the task that `GetURLContentsAsync` returns to obtain the byte array value.

   The following code shows these changes.

   VB

   ```
   Dim urlContents As Byte() = Await GetURLContentsAsync(url)
   ```

   The previous assignment abbreviates the following two lines of code.

   VB

   ```
   ' GetURLContentsAsync returns a task. At completion, the task
   ' produces a byte array.
   'Dim getContentsTask As Task(Of Byte()) = GetURLContentsAsync(url)
   'Dim urlContents As Byte() = 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 method type from **Sub** to **Function**. The return type of the function is **Task**.

   The following code shows these changes.

   VB

```
Private Async Function SumPageSizesAsync() As Task
```

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

To convert startButton_Click to an asynchronous method

1. In the event handler, change the name of the called method from `SumPageSizes` to `SumPageSizesAsync`, if you haven't already done so.

2. Because `SumPageSizesAsync` is an async method, change the code in the event handler to await the result.

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

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

   VB

   ```
   '' One-step async call.
   Await SumPageSizesAsync()

   ' Two-step async call.
   'Dim sumTask As Task = 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.

   VB

   ```
   ' Disable the button until the operation is complete.
   startButton.IsEnabled = False
   ```

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

   VB

   ```
   ' 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 (Visual Basic).

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

   VB

```
Async Sub startButton_Click(sender As Object, e As RoutedEventArgs) Handles
startButton.Click
```

Typically, the names of event handlers aren't changed. The return type isn't changed to **Task** because event handlers must be **Sub** procedures in Visual Basic.

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

To test the asynchronous solution

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

To replace method GetURLContentsAsync with a .NET Framework method

1. The .NET Framework 4.5 provides many async methods that you can use. One of them, the HttpClient 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.

   VB

   ```
   ' Declare an HttpClient object and increase the buffer size. The
   ' default buffer size is 65,536.
   Dim client As HttpClient =
       New HttpClient() With {.MaxResponseContentBufferSize = 1000000}
   ```

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

   VB

```
Dim urlContents As Byte() = 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

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.

VB

```
' Add the following Imports statements, and add a reference for System.Net.Http.
Imports System.Net.Http
Imports System.Net
Imports System.IO

Class MainWindow

    Async Sub startButton_Click(sender As Object, e As RoutedEventArgs) Handles
startButton.Click

        ' Disable the button until the operation is complete.
        startButton.IsEnabled = False

        resultsTextBox.Clear()

        '' One-step async call.
        Await SumPageSizesAsync()

        ' Two-step async call.
        'Dim sumTask As Task = SumPageSizesAsync()
        'Await sumTask

        resultsTextBox.Text &= vbCrLf & "Control returned to button1_Click."

        ' Reenable the button in case you want to run the operation again.
        startButton.IsEnabled = True
    End Sub


    Private Async Function SumPageSizesAsync() As Task

        ' Make a list of web addresses.
        Dim urlList As List(Of String) = SetUpURLList()
```

```
        Dim total = 0
        For Each url In urlList
            Dim urlContents As Byte() = Await GetURLContentsAsync(url)

            ' The previous line abbreviates the following two assignment statements.

            '//<snippet21>
            ' GetURLContentsAsync returns a task. At completion, the task
            ' produces a byte array.
            'Dim getContentsTask As Task(Of Byte()) = GetURLContentsAsync(url)
            'Dim urlContents As Byte() = Await getContentsTask

            DisplayResults(url, urlContents)

            ' Update the total.
            total += urlContents.Length
        Next

        ' Display the total count for all of the websites.
        resultsTextBox.Text &= String.Format(vbCrLf & vbCrLf &
                                             "Total bytes returned:  {0}" & vbCrLf,
total)
    End Function


    Private Function SetUpURLList() As List(Of String)

        Dim urls = New List(Of String) From
            {
                "http://msdn.microsoft.com/library/windows/apps/br211380.aspx",
                "http://msdn.microsoft.com",
                "http://msdn.microsoft.com/en-us/library/hh290136.aspx",
                "http://msdn.microsoft.com/en-us/library/ee256749.aspx",
                "http://msdn.microsoft.com/en-us/library/hh290138.aspx",
                "http://msdn.microsoft.com/en-us/library/hh290140.aspx",
                "http://msdn.microsoft.com/en-us/library/dd470362.aspx",
                "http://msdn.microsoft.com/en-us/library/aa578028.aspx",
                "http://msdn.microsoft.com/en-us/library/ms404677.aspx",
                "http://msdn.microsoft.com/en-us/library/ff730837.aspx"
            }
        Return urls
    End Function


    Private Async Function GetURLContentsAsync(url As String) As Task(Of Byte())

        ' The downloaded resource ends up in the variable named content.
        Dim content = New MemoryStream()

        ' Initialize an HttpWebRequest for the current URL.
        Dim webReq = CType(WebRequest.Create(url), HttpWebRequest)

        ' Send the request to the Internet resource and wait for
        ' the response.
```

```
        Using response As WebResponse = Await webReq.GetResponseAsync()

            ' The previous statement abbreviates the following two statements.

            'Dim responseTask As Task(Of WebResponse) = webReq.GetResponseAsync()
            'Using response As WebResponse = Await responseTask

            ' Get the data stream that is associated with the specified URL.
            Using responseStream As Stream = 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>.
                'Dim copyTask As Task = responseStream.CopyToAsync(content)

                ' When copyTask is completed, content contains a copy of
                ' responseStream.
                'Await copyTask
            End Using
        End Using

        ' Return the result as a byte array.
        Return content.ToArray()
    End Function


    Private Sub DisplayResults(url As String, content As Byte())

        ' 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.
        Dim bytes = content.Length
        ' Strip off the "http://".
        Dim displayURL = url.Replace("http://", "")
        resultsTextBox.Text &= String.Format(vbCrLf & "{0,-58} {1,8}", displayURL,
bytes)
    End Sub

End Class
```

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

VB

```
' Add the following Imports statements, and add a reference for System.Net.Http.
Imports System.Net.Http
Imports System.Net
Imports System.IO

Class MainWindow
```

```
    Async Sub startButton_Click(sender As Object, e As RoutedEventArgs) Handles
startButton.Click

        resultsTextBox.Clear()

        ' Disable the button until the operation is complete.
        startButton.IsEnabled = False

        ' One-step async call.
        Await SumPageSizesAsync()

        '' Two-step async call.
        'Dim sumTask As Task = SumPageSizesAsync()
        'Await sumTask

        resultsTextBox.Text &= vbCrLf & "Control returned to button1_Click."

        ' Reenable the button in case you want to run the operation again.
        startButton.IsEnabled = True
    End Sub


    Private Async Function SumPageSizesAsync() As Task

        ' Declare an HttpClient object and increase the buffer size. The
        ' default buffer size is 65,536.
        Dim client As HttpClient =
            New HttpClient() With {.MaxResponseContentBufferSize = 1000000}

        ' Make a list of web addresses.
        Dim urlList As List(Of String) = SetUpURLList()

        Dim total = 0
        For Each url In urlList
            ' GetByteArrayAsync returns a task. At completion, the task
            ' produces a byte array.
            Dim urlContents As Byte() = Await client.GetByteArrayAsync(url)

            ' The following two lines can replace the previous assignment statement.
            'Dim getContentsTask As Task(Of Byte()) = client.GetByteArrayAsync(url)
            'Dim urlContents As Byte() = Await getContentsTask

            DisplayResults(url, urlContents)

            ' Update the total.
            total += urlContents.Length
        Next

        ' Display the total count for all of the websites.
        resultsTextBox.Text &= String.Format(vbCrLf & vbCrLf &
                                             "Total bytes returned:  {0}" & vbCrLf,
total)
    End Function
```

```
    Private Function SetUpURLList() As List(Of String)

        Dim urls = New List(Of String) From
            {
                "http://msdn.microsoft.com/library/windows/apps/br211380.aspx",
                "http://msdn.microsoft.com",
                "http://msdn.microsoft.com/en-us/library/hh290136.aspx",
                "http://msdn.microsoft.com/en-us/library/ee256749.aspx",
                "http://msdn.microsoft.com/en-us/library/hh290138.aspx",
                "http://msdn.microsoft.com/en-us/library/hh290140.aspx",
                "http://msdn.microsoft.com/en-us/library/dd470362.aspx",
                "http://msdn.microsoft.com/en-us/library/aa578028.aspx",
                "http://msdn.microsoft.com/en-us/library/ms404677.aspx",
                "http://msdn.microsoft.com/en-us/library/ff730837.aspx"
            }
        Return urls
    End Function


    Private Sub DisplayResults(url As String, content As Byte())

        ' 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.
        Dim bytes = content.Length
        ' Strip off the "http://".
        Dim displayURL = url.Replace("http://", "")
        resultsTextBox.Text &= String.Format(vbCrLf & "{0,-58} {1,8}", displayURL,
bytes)
    End Sub

End Class
```

See Also

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

How to: Extend the Async Walkthrough by Using Task.WhenAll (Visual Basic)

Visual Studio 2015

You can improve the performance of the async solution in Walkthrough: Accessing the Web by Using Async and Await (Visual Basic) 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 (Visual Basic). 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 (Visual Basic).

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

   - If you developed the code by completing the walkthrough, add `ProcessURLAsync` to the application that includes the `GetURLContentsAsync` method. The MainWindow.xaml.vb 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 **For Each** 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.

   VB

```
Private Async Function ProcessURLAsync(url As String) As Task(Of Integer)
```

```
    Dim byteArray = Await GetURLContentsAsync(url)
    DisplayResults(url, byteArray)
    Return byteArray.Length
End Function
```

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

VB

```
'Dim total = 0
'For Each url In urlList

'    Dim urlContents As Byte() = Await GetURLContentsAsync(url)

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

'    ' GetURLContentsAsync returns a task. At completion, the task
'    ' produces a byte array.
'    'Dim getContentsTask As Task(Of Byte()) = GetURLContentsAsync(url)
'    'Dim urlContents As Byte() = Await getContentsTask

'    DisplayResults(url, urlContents)

'    ' Update the total.
'    total += urlContents.Length
'Next
```

3. Create a collection of tasks. The following code defines a query that, when executed by the ToArray(Of TSource) 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`.

VB

```
' Create a query.
Dim downloadTasksQuery As IEnumerable(Of Task(Of Integer)) =
    From url In urlList Select ProcessURLAsync(url)

' Use ToArray to execute the query and start the download tasks.
Dim downloadTasks As Task(Of Integer)() = 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.

VB

```
' Await the completion of all the running tasks.
Dim lengths As Integer() = Await Task.WhenAll(downloadTasks)

'' The previous line is equivalent to the following two statements.
'Dim whenAllTask As Task(Of Integer()) = Task.WhenAll(downloadTasks)
'Dim lengths As Integer() = 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`.

VB

```
Dim 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 (Visual Basic).

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

   - If you developed the code by completing the walkthrough, add `ProcessURLAsync` to the application that uses the `HttpClient.GetByteArrayAsync` method. The MainWindow.xaml.vb 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 **For Each** 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`.

VB

```
Private Async Function ProcessURLAsync(url As String, client As HttpClient) As
Task(Of Integer)

    Dim byteArray = Await client.GetByteArrayAsync(url)
    DisplayResults(url, byteArray)
    Return byteArray.Length
End Function
```

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

VB

```
'Dim total = 0
'For Each url In urlList
```

```
'    ' GetByteArrayAsync returns a task. At completion, the task
'    ' produces a byte array.
'    Dim urlContents As Byte() = Await client.GetByteArrayAsync(url)

'    ' The following two lines can replace the previous assignment statement.
'    'Dim getContentsTask As Task(Of Byte()) = client.GetByteArrayAsync(url)
'    'Dim urlContents As Byte() = Await getContentsTask

'    DisplayResults(url, urlContents)

'    ' Update the total.
'    total += urlContents.Length
'NextNext
```

3. Define a query that, when executed by the ToArray(Of TSource) 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`.

VB

```
' Create a query.
Dim downloadTasksQuery As IEnumerable(Of Task(Of Integer)) =
    From url In urlList Select ProcessURLAsync(url, client)

' Use ToArray to execute the query and start the download tasks.
Dim downloadTasks As Task(Of Integer)() = 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.

VB

```
' Await the completion of all the running tasks.
Dim lengths As Integer() = Await Task.WhenAll(downloadTasks)

'' The previous line is equivalent to the following two statements.
'Dim whenAllTask As Task(Of Integer()) = Task.WhenAll(downloadTasks)
'Dim lengths As Integer() = 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`.

VB

```
Dim 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 (Visual Basic). 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.

VB

```
' Add the following Imports statements, and add a reference for System.Net.Http.
Imports System.Net.Http
Imports System.Net
Imports System.IO


Class MainWindow

    Async Sub startButton_Click(sender As Object, e As RoutedEventArgs) Handles
startButton.Click

        resultsTextBox.Clear()

        ' One-step async call.
        Await SumPageSizesAsync()

        '' Two-step async call.
        'Dim sumTask As Task = SumPageSizesAsync()
        'Await sumTask

        resultsTextBox.Text &= vbCrLf & "Control returned to button1_Click."
    End Sub


    Private Async Function SumPageSizesAsync() As Task

        ' Make a list of web addresses.
        Dim urlList As List(Of String) = SetUpURLList()

        ' Create a query.
        Dim downloadTasksQuery As IEnumerable(Of Task(Of Integer)) =
            From url In urlList Select ProcessURLAsync(url)

        ' Use ToArray to execute the query and start the download tasks.
        Dim downloadTasks As Task(Of Integer)() = downloadTasksQuery.ToArray()

        ' You can do other work here before awaiting.

        ' Await the completion of all the running tasks.
```

```
    Dim lengths As Integer() = Await Task.WhenAll(downloadTasks)

    '' The previous line is equivalent to the following two statements.
    'Dim whenAllTask As Task(Of Integer()) = Task.WhenAll(downloadTasks)
    'Dim lengths As Integer() = Await whenAllTask

    Dim total = lengths.Sum()

    'Dim total = 0
    'For Each url In urlList

    '    Dim urlContents As Byte() = Await GetURLContentsAsync(url)

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

    '    ' GetURLContentsAsync returns a task. At completion, the task
    '    ' produces a byte array.
    '    'Dim getContentsTask As Task(Of Byte()) = GetURLContentsAsync(url)
    '    'Dim urlContents As Byte() = Await getContentsTask

    '    DisplayResults(url, urlContents)

    '    ' Update the total.
    '    total += urlContents.Length
    'NextNext

    ' Display the total count for all of the web addresses.
    resultsTextBox.Text &= String.Format(vbCrLf & vbCrLf &
                                         "Total bytes returned:  {0}" & vbCrLf, total)
End Function


Private Function SetUpURLList() As List(Of String)

    Dim urls = New List(Of String) From
        {
            "http://msdn.microsoft.com",
            "http://msdn.microsoft.com/en-us/library/hh290136.aspx",
            "http://msdn.microsoft.com/en-us/library/ee256749.aspx",
            "http://msdn.microsoft.com/en-us/library/hh290138.aspx",
            "http://msdn.microsoft.com/en-us/library/hh290140.aspx",
            "http://msdn.microsoft.com/en-us/library/dd470362.aspx",
            "http://msdn.microsoft.com/en-us/library/aa578028.aspx",
            "http://msdn.microsoft.com/en-us/library/ms404677.aspx",
            "http://msdn.microsoft.com/en-us/library/ff730837.aspx"
        }
    Return urls
End Function


' The actions from the foreach loop are moved to this async method.
Private Async Function ProcessURLAsync(url As String) As Task(Of Integer)

    Dim byteArray = Await GetURLContentsAsync(url)
```

```
        DisplayResults(url, byteArray)
        Return byteArray.Length
    End Function


    Private Async Function GetURLContentsAsync(url As String) As Task(Of Byte())

        ' The downloaded resource ends up in the variable named content.
        Dim content = New MemoryStream()

        ' Initialize an HttpWebRequest for the current URL.
        Dim webReq = CType(WebRequest.Create(url), HttpWebRequest)

        ' Send the request to the Internet resource and wait for
        ' the response.
        Using response As WebResponse = Await webReq.GetResponseAsync()
            ' Get the data stream that is associated with the specified URL.
            Using responseStream As Stream = response.GetResponseStream()
                ' Read the bytes in responseStream and copy them to content.
                ' CopyToAsync returns a Task, not a Task<T>.
                Await responseStream.CopyToAsync(content)
            End Using
        End Using

        ' Return the result as a byte array.
        Return content.ToArray()
    End Function



    Private Sub DisplayResults(url As String, content As Byte())

        ' 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.
        Dim bytes = content.Length
        ' Strip off the "http://".
        Dim displayURL = url.Replace("http://", "")
        resultsTextBox.Text &= String.Format(vbCrLf & "{0,-58} {1,8}", displayURL, bytes)
    End Sub

End Class
```

Example

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

VB

```
' Add the following Imports statements, and add a reference for System.Net.Http.
Imports System.Net.Http
Imports System.Net
```

```
Imports System.IO

Class MainWindow

    Async Sub startButton_Click(sender As Object, e As RoutedEventArgs) Handles
startButton.Click

        resultsTextBox.Clear()

        '' One-step async call.
        Await SumPageSizesAsync()

        '' Two-step async call.
        'Dim sumTask As Task = SumPageSizesAsync()
        'Await sumTask

        resultsTextBox.Text &= vbCrLf & "Control returned to button1_Click."
    End Sub


    Private Async Function SumPageSizesAsync() As Task

        ' Declare an HttpClient object and increase the buffer size. The
        ' default buffer size is 65,536.
        Dim client As HttpClient =
            New HttpClient() With {.MaxResponseContentBufferSize = 1000000}

        ' Make a list of web addresses.
        Dim urlList As List(Of String) = SetUpURLList()

        ' Create a query.
        Dim downloadTasksQuery As IEnumerable(Of Task(Of Integer)) =
            From url In urlList Select ProcessURLAsync(url, client)

        ' Use ToArray to execute the query and start the download tasks.
        Dim downloadTasks As Task(Of Integer)() = downloadTasksQuery.ToArray()

        ' You can do other work here before awaiting.

        ' Await the completion of all the running tasks.
        Dim lengths As Integer() = Await Task.WhenAll(downloadTasks)

        '' The previous line is equivalent to the following two statements.
        'Dim whenAllTask As Task(Of Integer()) = Task.WhenAll(downloadTasks)
        'Dim lengths As Integer() = Await whenAllTask

        Dim total = lengths.Sum()

        ''<snippet7>
        'Dim total = 0
        'For Each url In urlList
        '    ' GetByteArrayAsync returns a task. At completion, the task
        '    ' produces a byte array.
        '    '<snippet31>
```

```
        '    Dim urlContents As Byte() = Await client.GetByteArrayAsync(url)
        '    '</snippet31>

        '    ' The following two lines can replace the previous assignment statement.
        '    'Dim getContentsTask As Task(Of Byte()) = client.GetByteArrayAsync(url)
        '    'Dim urlContents As Byte() = Await getContentsTask

        '    DisplayResults(url, urlContents)

        '    ' Update the total.
        '    total += urlContents.Length
        'NextNext

        ' Display the total count for all of the web addresses.
        resultsTextBox.Text &= String.Format(vbCrLf & vbCrLf &
                                             "Total bytes returned:  {0}" & vbCrLf, total)
    End Function


    Private Function SetUpURLList() As List(Of String)

        Dim urls = New List(Of String) From
            {
                "http://www.msdn.com",
                "http://msdn.microsoft.com/en-us/library/hh290136.aspx",
                "http://msdn.microsoft.com/en-us/library/ee256749.aspx",
                "http://msdn.microsoft.com/en-us/library/hh290138.aspx",
                "http://msdn.microsoft.com/en-us/library/hh290140.aspx",
                "http://msdn.microsoft.com/en-us/library/dd470362.aspx",
                "http://msdn.microsoft.com/en-us/library/aa578028.aspx",
                "http://msdn.microsoft.com/en-us/library/ms404677.aspx",
                "http://msdn.microsoft.com/en-us/library/ff730837.aspx"
            }
        Return urls
    End Function

    Private Async Function ProcessURLAsync(url As String, client As HttpClient) As
Task(Of Integer)

        Dim byteArray = Await client.GetByteArrayAsync(url)
        DisplayResults(url, byteArray)
        Return byteArray.Length
    End Function


    Private Sub DisplayResults(url As String, content As Byte())

        ' 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.
        Dim bytes = content.Length
        ' Strip off the "http://".
        Dim displayURL = url.Replace("http://", "")
        resultsTextBox.Text &= String.Format(vbCrLf & "{0,-58} {1,8}", displayURL, bytes)
```

```
        End Sub

    End Class
```

See Also

Task.WhenAll
Walkthrough: Accessing the Web by Using Async and Await (Visual Basic)

How to: Make Multiple Web Requests in Parallel by Using Async and Await (Visual Basic)

Visual Studio 2015

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.

VB

```
Dim 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.

VB

```
' The following line creates and starts the task.
Dim myTask = someWebAccessMethodAsync(url)

' While the task is running, you can do other work that does not depend
' on the results of the task.
' . . . . .

' The application of Await suspends the rest of this method until the task is
' complete.
Dim 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 (Visual Basic).

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 (Visual Basic).
    - 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.vb file, add an **Imports** statement 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.vb.
2. Copy the following code, and paste it into the body of `startButton_Click` in MainWindow.xaml.vb.

VB

```
resultsTextBox.Clear()
Await CreateMultipleTasksAsync()
resultsTextBox.Text &= vbCrLf & "Control returned to button1_Click."
```

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.vb.

VB

```
Private Async Function ProcessURLAsync(url As String, client As HttpClient) As
Task(Of Integer)

    Dim byteArray = Await client.GetByteArrayAsync(url)
    DisplayResults(url, byteArray)
    Return byteArray.Length
End Function
```

```
Private Sub DisplayResults(url As String, content As Byte())

    ' 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.
    Dim bytes = content.Length
    ' Strip off the "http://".
    Dim displayURL = url.Replace("http://", "")
    resultsTextBox.Text &= String.Format(vbCrLf & "{0,-58} {1,8}", displayURL, bytes)
End Sub
```

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(Of 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.

VB

```
Private Async Function CreateMultipleTasksAsync() As Task

    ' Declare an HttpClient object, and increase the buffer size. The
    ' default buffer size is 65,536.
    Dim client As HttpClient =
        New HttpClient() With {.MaxResponseContentBufferSize = 1000000}

    ' Create and start the tasks. As each task finishes, DisplayResults
    ' displays its length.
    Dim download1 As Task(Of Integer) =
        ProcessURLAsync("http://msdn.microsoft.com", client)
    Dim download2 As Task(Of Integer) =
        ProcessURLAsync("http://msdn.microsoft.com/en-us/library
/hh156528(VS.110).aspx", client)
    Dim download3 As Task(Of Integer) =
        ProcessURLAsync("http://msdn.microsoft.com/en-us/library/67w7t67f.aspx",
client)

    ' Await each task.
```

```
        Dim length1 As Integer = Await download1
        Dim length2 As Integer = Await download2
        Dim length3 As Integer = Await download3

        Dim total As Integer = length1 + length2 + length3

        ' Display the total count for all of the websites.
        resultsTextBox.Text &= String.Format(vbCrLf & vbCrLf &
                                             "Total bytes returned:  {0}" & vbCrLf,
    total)
    End Function
```

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.

VB

```
' Add the following Imports statements, and add a reference for System.Net.Http.
Imports System.Net.Http


Class MainWindow

    Async Sub startButton_Click(sender As Object, e As RoutedEventArgs) Handles
startButton.Click
        resultsTextBox.Clear()
        Await CreateMultipleTasksAsync()
        resultsTextBox.Text &= vbCrLf & "Control returned to button1_Click."
    End Sub


    Private Async Function CreateMultipleTasksAsync() As Task

        ' Declare an HttpClient object, and increase the buffer size. The
        ' default buffer size is 65,536.
        Dim client As HttpClient =
            New HttpClient() With {.MaxResponseContentBufferSize = 1000000}

        ' Create and start the tasks. As each task finishes, DisplayResults
        ' displays its length.
        Dim download1 As Task(Of Integer) =
            ProcessURLAsync("http://msdn.microsoft.com", client)
        Dim download2 As Task(Of Integer) =
            ProcessURLAsync("http://msdn.microsoft.com/en-us/library
/hh156528(VS.110).aspx", client)
        Dim download3 As Task(Of Integer) =
```

```
            ProcessURLAsync("http://msdn.microsoft.com/en-us/library/67w7t67f.aspx",
client)

        ' Await each task.
        Dim length1 As Integer = Await download1
        Dim length2 As Integer = Await download2
        Dim length3 As Integer = Await download3

        Dim total As Integer = length1 + length2 + length3

        ' Display the total count for all of the websites.
        resultsTextBox.Text &= String.Format(vbCrLf & vbCrLf &
                                             "Total bytes returned:  {0}" & vbCrLf, total)
    End Function


    Private Async Function ProcessURLAsync(url As String, client As HttpClient) As
Task(Of Integer)

        Dim byteArray = Await client.GetByteArrayAsync(url)
        DisplayResults(url, byteArray)
        Return byteArray.Length
    End Function


    Private Sub DisplayResults(url As String, content As Byte())

        ' 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.
        Dim bytes = content.Length
        ' Strip off the "http://".
        Dim displayURL = url.Replace("http://", "")
        resultsTextBox.Text &= String.Format(vbCrLf & "{0,-58} {1,8}", displayURL, bytes)
    End Sub
End Class
```

See Also

Walkthrough: Accessing the Web by Using Async and Await (Visual Basic)
Asynchronous Programming with Async and Await (Visual Basic)
How to: Extend the Async Walkthrough by Using Task.WhenAll (Visual Basic)

Async Return Types (Visual Basic)

Visual Studio 2015

Async methods have three possible return types: Task(Of TResult), Task, and void. In Visual Basic, the void return type is written as a Sub procedure. For more information about async methods, see Asynchronous Programming with Async and Await (Visual Basic).

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.

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.

Task(T) Return Type

The Task(Of TResult) return type is used for an async method that contains a Return statement in which the operand has type **TResult**.

In the following example, the `TaskOfT_MethodAsync` async method contains a return statement that returns an integer. Therefore, the method declaration must specify a return type of **Task(Of Integer)**.

VB

```
' TASK(OF T) EXAMPLE
Async Function TaskOfT_MethodAsync() As Task(Of Integer)

    ' The body of an async method is expected to contain an awaited
    ' asynchronous call.
    ' Task.FromResult is a placeholder for actual work that returns a string.
    Dim today As String = Await Task.FromResult(Of String)
(DateTime.Now.DayOfWeek.ToString())

    ' The method then can process the result in some way.
    Dim leisureHours As Integer
    If today.First() = "S" Then
        leisureHours = 16
    Else
        leisureHours = 5
    End If

    ' Because the return statement specifies an operand of type Integer, the
    ' method must have a return type of Task(Of Integer).
    Return leisureHours
```

```
    End Function
```

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

The following code calls and awaits method `TaskOfT_MethodAsync`. The result is assigned to the `result1` variable.

VB

```
' Call and await the Task(Of T)-returning async method in the same statement.
Dim result1 As Integer = Await TaskOfT_MethodAsync()
```

You can better understand how this happens by separating the call to `TaskOfT_MethodAsync` from the application of **Await**, as the following code shows. A call to method `TaskOfT_MethodAsync` that isn't immediately awaited returns a **Task(Of Integer)**, 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(Of 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 `result2` variable.

Warning

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.

VB

```
' Call and await in separate statements.
Dim integerTask As Task(Of Integer) = TaskOfT_MethodAsync()

' You can do other work that does not rely on resultTask before awaiting.
textBox1.Text &= String.Format("Application can continue working while the Task(Of T)
runs. . . . " & vbCrLf)

Dim result2 As Integer = Await integerTask
```

The display statements in the following code verify that the values of the `result1` variable, the `result2` variable, and the **Result** property are the same. Remember that the **Result** property is a blocking property and shouldn't be accessed before its task has been awaited.

VB

```
' Display the values of the result1 variable, the result2 variable, and
' the resultTask.Result property.
textBox1.Text &= String.Format(vbCrLf & "Value of result1 variable:   {0}" & vbCrLf,
result1)
textBox1.Text &= String.Format("Value of result2 variable:   {0}" & vbCrLf, result2)
```

```
    textBox1.Text &= String.Format("Value of resultTask.Result:  {0}" & vbCrLf,
    integerTask.Result)
```

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 would be Sub procedures if they were written to 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, async method `Task_MethodAsync` doesn't contain a return statement. Therefore, you specify a return type of **Task** for the method, which enables `Task_MethodAsync` to be awaited. The definition of the **Task** type doesn't include a **Result** property to store a return value.

VB

```
' TASK EXAMPLE
Async Function Task_MethodAsync() As Task

    ' The body of an async method is expected to contain an awaited
    ' asynchronous call.
    ' Task.Delay is a placeholder for actual work.
    Await Task.Delay(2000)
    textBox1.Text &= String.Format(vbCrLf & "Sorry for the delay. . . ." & vbCrLf)

    ' This method has no return statement, so its return type is Task.
End Function
```

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

The following code calls and awaits method `Task_MethodAsync`.

VB

```
' Call and await the Task-returning async method in the same statement.
Await Task_MethodAsync()
```

As in the previous Task(Of TResult) example, you can separate the call to `Task_MethodAsync` 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 `Task_MethodAsync` from awaiting the task that `Task_MethodAsync` returns.

VB

```
' Call and await in separate statements.
Dim simpleTask As Task = Task_MethodAsync()
```

```
    ' You can do other work that does not rely on simpleTask before awaiting.
    textBox1.Text &= String.Format(vbCrLf & "Application can continue working while the
    Task runs. . . ." & vbCrLf)

    Await simpleTask
```

Void Return Type

The primary use of **Sub** procedures is in event handlers, where there is no return type (referred to as a void return type in other languages). A void return also can be used to override void-returning methods or for methods that perform activities that can be categorized as "fire and forget." However, you should return a **Task** wherever possible, because a void-returning async method 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(Of TResult), the exception is stored in the returned task, and 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(Of TResult) and that calls to the method are awaited.

For more information about how to catch exceptions in async methods, see Try...Catch...Finally Statement (Visual Basic).

The following code defines an async event handler.

VB

```
' SUB EXAMPLE
Async Sub button1_Click(sender As Object, e As RoutedEventArgs) Handles button1.Click

    textBox1.Clear()

    ' Start the process and await its completion. DriverAsync is a
    ' Task-returning async method.
    Await DriverAsync()

    ' Say goodbye.
    textBox1.Text &= vbCrLf & "All done, exiting button-click event handler."
End Sub
```

Complete Example

The following Windows Presentation Foundation (WPF) project contains the code examples from 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. In the **Installed**, **Templates** category, choose **Visual Basic**, and then choose **Windows**. Choose **WPF Application** from the list of project types.

4. Enter **AsyncReturnTypes** 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 is not visible, open the shortcut menu for MainWindow.xaml in **Solution Explorer**, and then choose **Open**.

6. In the **XAML** window of MainWindow.xaml, replace the code with the following code.

VB

```
<Window x:Class="MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Button x:Name="button1" Content="Start" HorizontalAlignment="Left"
Margin="214,28,0,0" VerticalAlignment="Top" Width="75"
HorizontalContentAlignment="Center" FontWeight="Bold" FontFamily="Aharoni"
Click="button1_Click"/>
        <TextBox x:Name="textBox1" Margin="0,80,0,0" TextWrapping="Wrap"
FontFamily="Lucida Console"/>

    </Grid>
</Window>
```

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

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

8. Replace the code in MainWindow.xaml.vb with the following code.

VB

```
Class MainWindow

    ' SUB EXAMPLE
    Async Sub button1_Click(sender As Object, e As RoutedEventArgs) Handles
button1.Click

        textBox1.Clear()
```

```
        ' Start the process and await its completion. DriverAsync is a
        ' Task-returning async method.
        Await DriverAsync()

        ' Say goodbye.
        textBox1.Text &= vbCrLf & "All done, exiting button-click event handler."
    End Sub


    Async Function DriverAsync() As Task

        ' Task(Of T)
        ' Call and await the Task(Of T)-returning async method in the same
statement.
        Dim result1 As Integer = Await TaskOfT_MethodAsync()

        ' Call and await in separate statements.
        Dim integerTask As Task(Of Integer) = TaskOfT_MethodAsync()

        ' You can do other work that does not rely on resultTask before awaiting.
        textBox1.Text &= String.Format("Application can continue working while the
Task(Of T) runs. . . . " & vbCrLf)

        Dim result2 As Integer = Await integerTask

        ' Display the values of the result1 variable, the result2 variable, and
        ' the resultTask.Result property.
        textBox1.Text &= String.Format(vbCrLf & "Value of result1 variable:   {0}"
& vbCrLf, result1)
        textBox1.Text &= String.Format("Value of result2 variable:   {0}" &
vbCrLf, result2)
        textBox1.Text &= String.Format("Value of resultTask.Result:  {0}" &
vbCrLf, integerTask.Result)

        ' Task
        ' Call and await the Task-returning async method in the same statement.
        Await Task_MethodAsync()

        ' Call and await in separate statements.
        Dim simpleTask As Task = Task_MethodAsync()

        ' You can do other work that does not rely on simpleTask before awaiting.
        textBox1.Text &= String.Format(vbCrLf & "Application can continue working
while the Task runs. . . ." & vbCrLf)

        Await simpleTask
    End Function


    ' TASK(OF T) EXAMPLE
    Async Function TaskOfT_MethodAsync() As Task(Of Integer)

        ' The body of an async method is expected to contain an awaited
        ' asynchronous call.
```

```
        ' Task.FromResult is a placeholder for actual work that returns a string.
        Dim today As String = Await Task.FromResult(Of String)
(DateTime.Now.DayOfWeek.ToString())

        ' The method then can process the result in some way.
        Dim leisureHours As Integer
        If today.First() = "S" Then
            leisureHours = 16
        Else
            leisureHours = 5
        End If

        ' Because the return statement specifies an operand of type Integer, the
        ' method must have a return type of Task(Of Integer).
        Return leisureHours
    End Function


    ' TASK EXAMPLE
    Async Function Task_MethodAsync() As Task

        ' The body of an async method is expected to contain an awaited
        ' asynchronous call.
        ' Task.Delay is a placeholder for actual work.
        Await Task.Delay(2000)
        textBox1.Text &= String.Format(vbCrLf & "Sorry for the delay. . . ." &
vbCrLf)

        ' This method has no return statement, so its return type is Task.
    End Function

End Class
```

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

   The following output should appear.

```
Application can continue working while the Task<T> runs. . . .

Value of result1 variable:   5
Value of result2 variable:   5
Value of integerTask.Result: 5

Sorry for the delay. . . .

Application can continue working while the Task runs. . . .

Sorry for the delay. . . .

All done, exiting button-click event handler.
```

See Also

- FromResult(Of TResult)
- Walkthrough: Accessing the Web by Using Async and Await (Visual Basic)
- Control Flow in Async Programs (Visual Basic)
- Async (Visual Basic)
- Await Operator (Visual Basic)

Control Flow in Async Programs (Visual Basic)

Visual Studio 2015

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

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

In general, you mark methods that contain asynchronous code with the Async (Visual Basic) modifier. In a method that's marked with an async modifier, you can use an Await (Visual Basic) 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 (Visual Basic).

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.

VB

```
Class MainWindow

    Private Async Sub StartButton_Click(sender As Object, e As RoutedEventArgs) Handles
StartButton.Click

        ' ONE
        Dim getLengthTask As Task(Of Integer) = AccessTheWebAsync()

        ' FOUR
```

```
        Dim contentLength As Integer = Await getLengthTask

        ' SIX
        ResultsTextBox.Text &=
            String.Format(vbCrLf & "Length of the downloaded string: {0}." & vbCrLf,
contentLength)

    End Sub


    Async Function AccessTheWebAsync() As Task(Of Integer)

        ' TWO
        Dim client As HttpClient = New HttpClient()
        Dim getStringTask As Task(Of String) =
            client.GetStringAsync("http://msdn.microsoft.com")

        ' THREE
        Dim urlContents As String = Await getStringTask

        ' FIVE
        Return urlContents.Length
    End Function

End Class
```

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.

Build 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. In the **Installed Templates** pane, choose **Visual Basic**, and then choose **WPF Application** from the list of project types.
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.

VB

```
<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="MainWindow"
    Title="Control Flow Trace" Height="350" Width="525">
    <Grid>
        <Button x:Name="StartButton" Content="Start" HorizontalAlignment="Left"
Margin="221,10,0,0" VerticalAlignment="Top" Width="75"/>
        <TextBox x:Name="ResultsTextBox" HorizontalAlignment="Left"
TextWrapping="Wrap" VerticalAlignment="Bottom" Width="510" Height="265"
FontFamily="Lucida Console" FontSize="10" VerticalScrollBarVisibility="Visible"
d:LayoutOverrides="HorizontalMargin"/>

    </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.vb, and then choose **View Code**.

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

VB

```
' Add an Imports statement and a reference for System.Net.Http.
Imports System.Net.Http

Class MainWindow

    Private Async Sub StartButton_Click(sender As Object, e As RoutedEventArgs)
Handles StartButton.Click

        ' The display lines in the example lead you through the control shifts.
        ResultsTextBox.Text &= "ONE:   Entering StartButton_Click." & vbCrLf &
            "           Calling AccessTheWebAsync." & vbCrLf

        Dim getLengthTask As Task(Of Integer) = AccessTheWebAsync()

        ResultsTextBox.Text &= vbCrLf & "FOUR:  Back in StartButton_Click." &
vbCrLf &
            "           Task getLengthTask is started." & vbCrLf &
```

```
            "           About to await getLengthTask -- no caller to return to."
& vbCrLf

        Dim contentLength As Integer = Await getLengthTask

        ResultsTextBox.Text &= vbCrLf & "SIX:   Back in StartButton_Click." &
vbCrLf &
            "           Task getLengthTask is finished." & vbCrLf &
            "           Result from AccessTheWebAsync is stored in
contentLength." & vbCrLf &
            "           About to display contentLength and exit." & vbCrLf

        ResultsTextBox.Text &=
            String.Format(vbCrLf & "Length of the downloaded string: {0}." &
vbCrLf, contentLength)
    End Sub


    Async Function AccessTheWebAsync() As Task(Of Integer)

        ResultsTextBox.Text &= vbCrLf & "TWO:   Entering AccessTheWebAsync."

        ' Declare an HttpClient object.
        Dim client As HttpClient = New HttpClient()

        ResultsTextBox.Text &= vbCrLf & "           Calling
HttpClient.GetStringAsync." & vbCrLf

        ' GetStringAsync returns a Task(Of String).
        Dim getStringTask As Task(Of String) =
client.GetStringAsync("http://msdn.microsoft.com")

        ResultsTextBox.Text &= vbCrLf & "THREE: Back in AccessTheWebAsync." &
vbCrLf &
            "           Task getStringTask is started."

        ' AccessTheWebAsync can continue to work until getStringTask is awaited.

        ResultsTextBox.Text &=
            vbCrLf & "           About to await getStringTask & return a Task(Of
Integer) to StartButton_Click." & vbCrLf

        ' Retrieve the website contents when task is complete.
        Dim urlContents As String = Await getStringTask

        ResultsTextBox.Text &= vbCrLf & "FIVE:  Back in AccessTheWebAsync." &
            vbCrLf & "           Task getStringTask is complete." &
            vbCrLf & "           Processing the return statement." &
            vbCrLf & "           Exiting from AccessTheWebAsync." & vbCrLf

        Return urlContents.Length
    End Function

End Class
```

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

    The following output should appear.

```
ONE:   Entering startButton_Click.
           Calling AccessTheWebAsync.

TWO:   Entering AccessTheWebAsync.
           Calling HttpClient.GetStringAsync.

THREE: Back in AccessTheWebAsync.
           Task getStringTask is started.
           About to await getStringTask & return a Task<int> to
startButton_Click.

FOUR:  Back in startButton_Click.
           Task getLengthTask is started.
           About to await getLengthTask -- no caller to return to.

FIVE:  Back in AccessTheWebAsync.
           Task getStringTask is complete.
           Processing the return statement.
           Exiting from AccessTheWebAsync.

SIX:   Back in startButton_Click.
           Task getLengthTask is finished.
           Result from AccessTheWebAsync is stored in contentLength.
           About to display contentLength and exit.

Length of the downloaded string: 33946.
```

Trace the Program

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.

The return type of both `AccessTheWebAsync` and `client.GetStringAsync` is Task(Of 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 (Visual Basic).

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.

VB

```
Dim getStringTask As Task(Of String) =
client.GetStringAsync("http://msdn.microsoft.com")
```

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

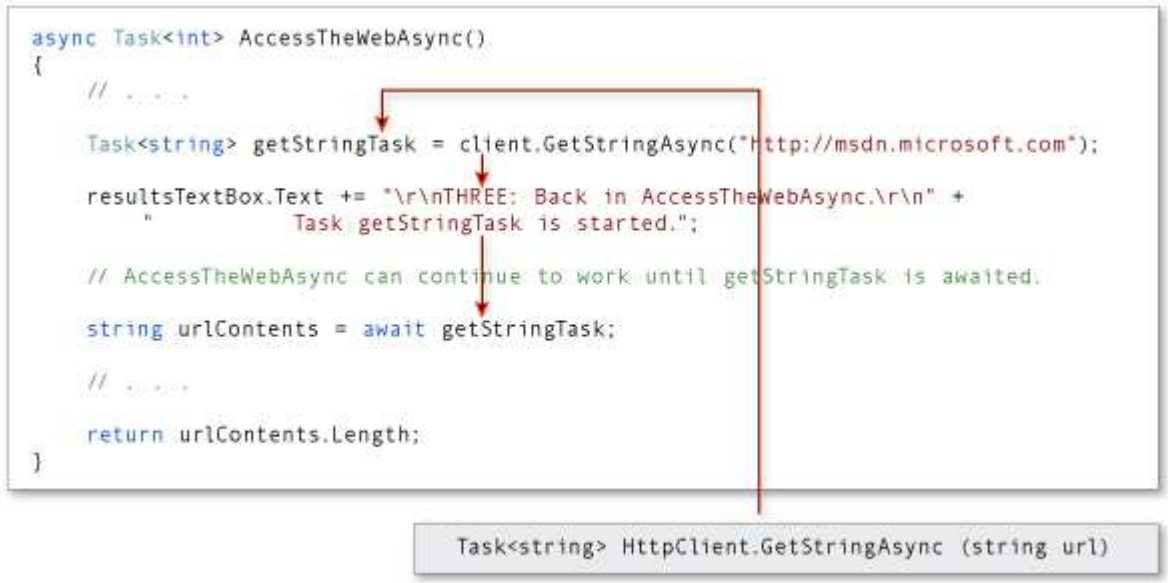
```
THREE: Back in AccessTheWebAsync.
           Task getStringTask is started.
           About to await getStringTask & return a Task<int> to startButton_Click.
```

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

VB

```
Dim urlContents As String = Await getStringTask
```

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



The await expression suspends `AccessTheWebAsync` until `client.GetStringAsync` returns. In the meantime, control returns to the caller of `AccessTheWebAsync`, `startButton_Click`.

Note

Typically, you await the call to an asynchronous method immediately. For example, the following assignment could replace the previous code that creates and then awaits `getStringTask`: `Dim urlContents As String = Await client.GetStringAsync("http://msdn.microsoft.com")`

In this topic, the await operator is applied later to accommodate the output lines that mark the flow of control through the program.

Step FOUR

The declared return type of `AccessTheWebAsync` is **Task(Of Integer)**. 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.

VB

```
Dim getLengthTask As Task(Of Integer) = 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.

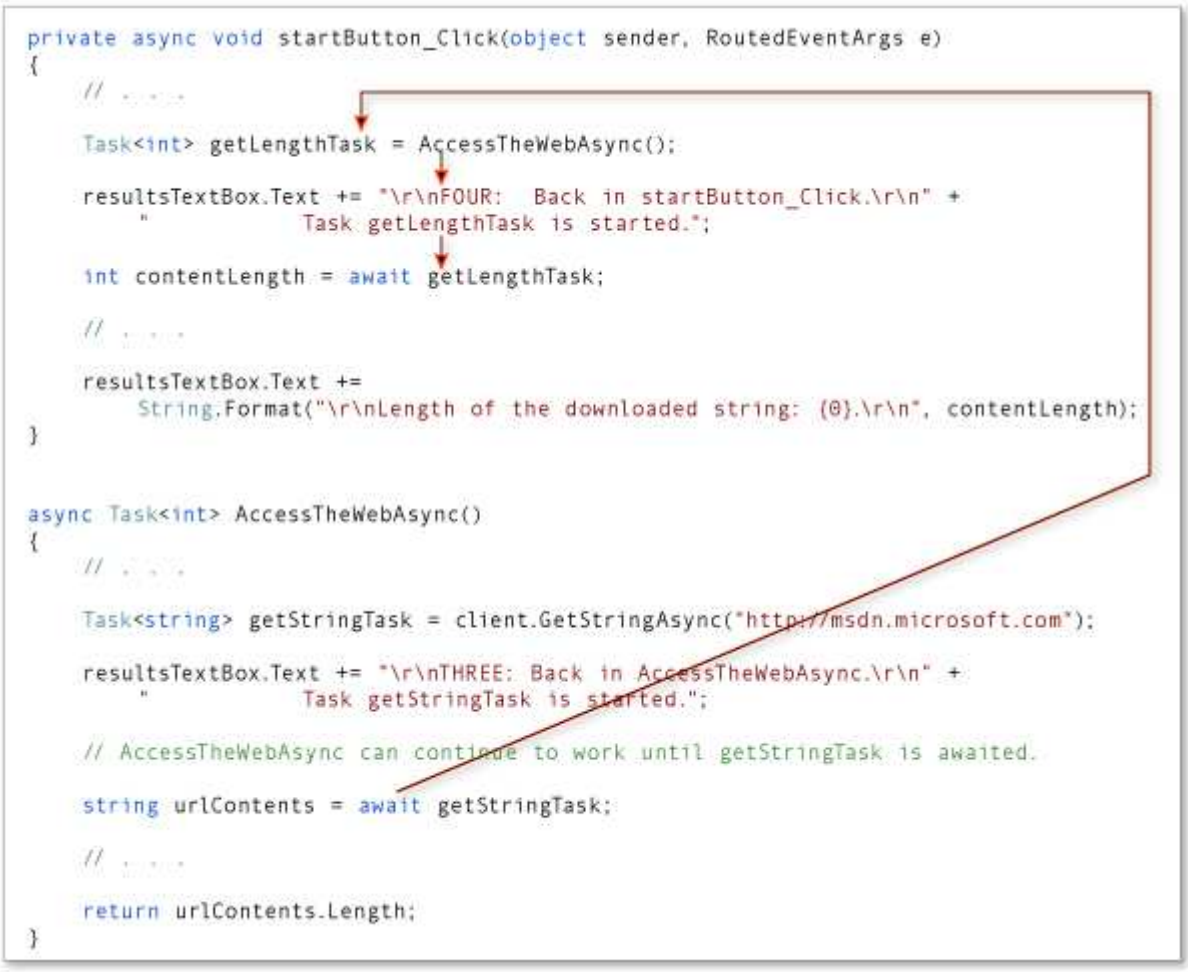
```
FOUR:  Back in startButton_Click.
           Task getLengthTask is started.
           About to await getLengthTask -- no caller to return to.
```

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

VB

```
Dim contentLength As Integer = Await getLengthTask
```

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



Step FIVE

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`:

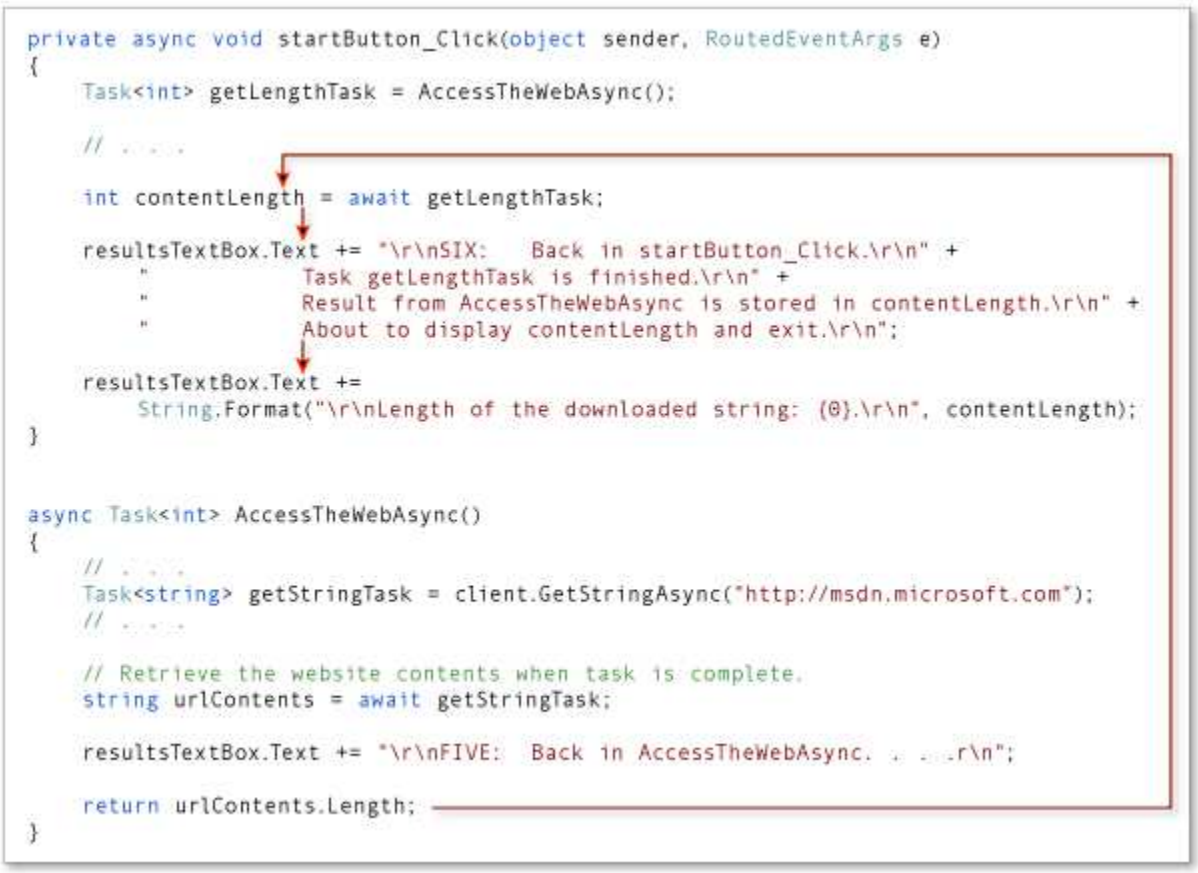
```
SIX:   Back in startButton_Click.
           Task getLengthTask is finished.
           Result from AccessTheWebAsync is stored in contentLength.
           About to display contentLength and exit.
```

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.

VB

```
Dim contentLength As Integer = 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 (Visual Basic)
- Async Return Types (Visual Basic)
- Walkthrough: Accessing the Web by Using Async and Await (Visual Basic)
- Async Sample: Control Flow in Async Programs (C# and Visual Basic)

Handling Reentrancy in Async Apps (Visual Basic)

Visual Studio 2015

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/en-us/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 5b54de66-6be3-459e-b869-65070b020645#BKMD_SettingUpTheExample.

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:

- 5b54de66-6be3-459e-b869-65070b020645#BKMK_DisableTheStartButton

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

- 5b54de66-6be3-459e-b869-65070b020645#BKMK_CancelAndRestart

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

- 5b54de66-6be3-459e-b869-65070b020645#BKMK_RunMultipleOperations

  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.

The following code shows these changes, which are marked with asterisks. You can add the changes to the code at the end of this topic, or you can download the finished app from Async Samples: Reentrancy in .NET Desktop Apps. The project name is DisableStartButton.

VB

```
Private Async Sub StartButton_Click(sender As Object, e As RoutedEventArgs)
    ' 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 ex As Exception
        ResultsTextBox.Text &= vbCrLf & "Downloads failed."
    ' ***Enable the Start button in case you want to run the program again.
    Finally
        StartButton.IsEnabled = True

    End Try
End Sub
```

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 (Visual Basic).

To set up this scenario, make the following changes to the basic code that is provided in 5b54de66-6be3-459e-b869-65070b020645#BKMD_SettingUpTheExample. You also can download the finished app from Async Samples: Reentrancy in .NET Desktop Apps. The name of this project is CancelAndRestart.

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

VB

```
Class MainWindow // Or Class MainPage

    ' *** Declare a System.Threading.CancellationTokenSource.
    Dim cts As CancellationTokenSource
```

2. In `StartButton_Click`, determine whether an operation is already underway. If the value of `cts` is **Nothing**, no operation is already active. If the value isn't **Nothing**, the operation that is already running is canceled.

VB

```
' *** If a download process is already underway, cancel it.
If cts IsNot Nothing Then
    cts.Cancel()
End If
```

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

VB

```
' *** Now set cts to cancel the current process if the button is chosen again.
Dim newCTS As CancellationTokenSource = New CancellationTokenSource()
cts = newCTS
```

4. At the end of `StartButton_Click`, the current process is complete, so set the value of `cts` back to **Nothing**.

VB

```
    ' *** When the process completes, signal that another process can proceed.
    If cts Is newCTS Then
        cts = Nothing
    End If
```

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

VB

```
Private Async Sub StartButton_Click(sender As Object, e As RoutedEventArgs)

    ' This line is commented out to make the results clearer.
    'ResultsTextBox.Text = ""


    ' *** If a download process is underway, cancel it.
    If cts IsNot Nothing Then
        cts.Cancel()
    End If

    ' *** Now set cts to cancel the current process if the button is chosen again.
    Dim newCTS As CancellationTokenSource = New CancellationTokenSource()
    cts = newCTS

    Try
        ' *** Send a token to carry the message if the operation is canceled.
        Await AccessTheWebAsync(cts.Token)


    Catch ex As OperationCanceledException
        ResultsTextBox.Text &= vbCrLf & "Download canceled." & vbCrLf

    Catch ex As Exception
        ResultsTextBox.Text &= vbCrLf & "Downloads failed."
    End Try

    ' *** When the process is complete, signal that another process can proceed.
    If cts Is newCTS Then
        cts = Nothing
    End If
End Sub
```

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.

VB

```
' *** Provide a parameter for the CancellationToken from StartButton_Click.
Private Async Function AccessTheWebAsync(ct As CancellationToken) As Task

    ' Declare an HttpClient object.
    Dim client = New HttpClient()

    ' Make a list of web addresses.
    Dim urlList As List(Of String) = SetUpURLList()

    Dim total = 0
    Dim position = 0

    For Each url In urlList
        ' *** Use the HttpClient.GetAsync method because it accepts a
        ' cancellation token.
        Dim response As HttpResponseMessage = Await client.GetAsync(url, ct)

        ' *** Retrieve the website contents from the HttpResponseMessage.
        Dim urlContents As Byte() = Await response.Content.ReadAsByteArrayAsync()

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

        position += 1
        DisplayResults(url, urlContents, position)

        ' Update the total.
        total += urlContents.Length
    Next

    ' Display the total count for all of the websites.
    ResultsTextBox.Text &=
        String.Format(vbCrLf & vbCrLf & "TOTAL bytes returned:  " & total & vbCrLf)
End Function
```

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 5b54de66-6be3-459e-b869-65070b020645#BKMK_RecognizingReentrancy 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.

You can run this example by pasting the changes into the code in 5b54de66-6be3-459e-b869-65070b020645#BKMK_BuildingTheApp, or you can follow the instructions in 5b54de66-6be3-459e-b869-65070b020645#BKMK_DownloadingTheApp to download the sample and then run the QueueResults project.

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.

VB

```
Class MainWindow    ' Class MainPage in Windows Store app.

    ' ***Declare the following variables where all methods can access them.
    Private pendingWork As Task = Nothing
    Private group As Char = ChrW(AscW("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.

VB

```
Private Async Sub StartButton_Click(sender As Object, e As RoutedEventArgs)
    ' ***Verify that each group's results are displayed together, and that
    ' the groups display in order, by marking each group with a letter.
```

```
    group = ChrW(AscW(group) + 1)
    ResultsTextBox.Text &= String.Format(vbCrLf & vbCrLf & "#Starting group {0}.",
group)

    Try
        ' *** Pass the group value to AccessTheWebAsync.
        Dim finishedGroup As Char = Await AccessTheWebAsync(group)

        ' The following line verifies a successful return from the download and
        ' display procedures.
        ResultsTextBox.Text &= String.Format(vbCrLf & vbCrLf & "#Group {0} is
complete." & vbCrLf, finishedGroup)

    Catch ex As Exception
        ResultsTextBox.Text &= vbCrLf & "Downloads failed."

    End Try
End Sub
```

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(Of TSource) 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.

VB

```
Private Async Function AccessTheWebAsync(grp As Char) As Task(Of Char)

    Dim client = New HttpClient()

    ' Make a list of the web addresses to download.
    Dim urlList As List(Of String) = SetUpURLList()

    ' ***Kick off the downloads. The application of ToArray activates all the
download tasks.
    Dim getContentTasks As Task(Of Byte())() =
        urlList.Select(Function(addr) client.GetByteArrayAsync(addr)).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 &=
```

```
            String.Format(vbCrLf & "#Task assigned for group {0}. Download tasks are
active." & vbCrLf, grp)

        ' ***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
    End Function
```

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.

VB

```
    Private Async Function FinishOneGroupAsync(urls As List(Of String), contentTasks As
    Task(Of Byte())(), grp As Char) As Task

        ' Wait for the previous group to finish displaying results.
        If pendingWork IsNot Nothing Then
            Await pendingWork
        End If

        Dim total = 0

        ' contentTasks is the array of Tasks that was created in AccessTheWebAsync.
        For i As Integer = 0 To contentTasks.Length - 1
            ' Await the download of a particular URL, and then display the URL and
            ' its length.
            Dim content As Byte() = Await contentTasks(i)
            DisplayResults(urls(i), content, i, grp)
            total += content.Length
        Next

        ' Display the total count for all of the websites.
        ResultsTextBox.Text &=
            String.Format(vbCrLf & vbCrLf & "TOTAL bytes returned:  " & total & vbCrLf)
    End Function
```

You can run this example by pasting the changes into the code in 5b54de66-6be3-459e-b869-65070b020645#BKMK_BuildingTheApp, or you can follow the instructions in 5b54de66-6be3-459e-b869-65070b020645#BKMK_DownloadingTheApp to download the sample, and then run the QueueResults project.

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 **Nothing** 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 Basic**, 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.

VB

```
<Window x:Class="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.vb, and then choose **View Code**.

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

VB

```
' Add the following Imports statements, and add a reference for System.Net.Http.
Imports System.Net.Http
Imports System.Threading

Class MainWindow
```

```
    Private Async Sub StartButton_Click(sender As Object, e As RoutedEventArgs)
        ' This line is commented out to make the results clearer in the output.
        'ResultsTextBox.Text = ""

        Try
            Await AccessTheWebAsync()

        Catch ex As Exception
            ResultsTextBox.Text &= vbCrLf & "Downloads failed."

        End Try
    End Sub


    Private Async Function AccessTheWebAsync() As Task

        ' Declare an HttpClient object.
        Dim client = New HttpClient()

        ' Make a list of web addresses.
        Dim urlList As List(Of String) = SetUpURLList()

        Dim total = 0
        Dim position = 0

        For Each url In urlList
            ' GetByteArrayAsync returns a task. At completion, the task
            ' produces a byte array.
            Dim urlContents As Byte() = Await client.GetByteArrayAsync(url)

            position += 1
            DisplayResults(url, urlContents, position)

            ' Update the total.
            total += urlContents.Length
        Next

        ' Display the total count for all of the websites.
        ResultsTextBox.Text &=
            String.Format(vbCrLf & vbCrLf & "TOTAL bytes returned:  " & total &
vbCrLf)
    End Function


    Private Function SetUpURLList() As List(Of String)
        Dim urls = New List(Of String) From
        {
            "http://msdn.microsoft.com/en-us/library/hh191443.aspx",
            "http://msdn.microsoft.com/en-us/library/aa578028.aspx",
            "http://msdn.microsoft.com/en-us/library/jj155761.aspx",
            "http://msdn.microsoft.com/en-us/library/hh290140.aspx",
            "http://msdn.microsoft.com/en-us/library/hh524395.aspx",
            "http://msdn.microsoft.com/en-us/library/ms404677.aspx",
            "http://msdn.microsoft.com",
```

```
            "http://msdn.microsoft.com/en-us/library/ff730837.aspx"
        }
        Return urls
    End Function


    Private Sub DisplayResults(url As String, content As Byte(), pos As Integer)
        ' 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 "http:'".
        Dim displayURL = url.Replace("http://", "")
        ' Display position in the URL list, the URL, and the number of bytes.
        ResultsTextBox.Text &= String.Format(vbCrLf & "{0}. {1,-58} {2,8}", pos,
displayURL, content.Length)
    End Sub
End Class
```

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

12. Make the changes from 5b54de66-6be3-459e-b869-65070b020645#BKMK_DisableTheStartButton, 5b54de66-6be3-459e-b869-65070b020645#BKMK_CancelAndRestart, or 5b54de66-6be3-459e-b869-65070b020645#BKMK_RunMultipleOperations to handle the reentrancy.

See Also

Walkthrough: Accessing the Web by Using Async and Await (Visual Basic)
Asynchronous Programming with Async and Await (Visual Basic)

Using Async for File Access (Visual Basic)

Visual Studio 2015

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 **Imports** statements.

VB

```
Imports System
Imports System.Collections.Generic
Imports System.Diagnostics
Imports System.IO
Imports System.Text
Imports 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.

VB

```
Public Async Sub ProcessWrite()
    Dim filePath = "temp2.txt"
    Dim text = "Hello World" & ControlChars.CrLf

    Await WriteTextAsync(filePath, text)
End Sub

Private Async Function WriteTextAsync(filePath As String, text As String) As Task
    Dim encodedText As Byte() = Encoding.Unicode.GetBytes(text)

    Using sourceStream As New FileStream(filePath,
        FileMode.Append, FileAccess.Write, FileShare.None,
        bufferSize:=4096, useAsync:=True)

        Await sourceStream.WriteAsync(encodedText, 0, encodedText.Length)
    End Using
End Function
```

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

VB

```
Dim theTask As Task = 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 (Visual Basic).

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 (Visual Basic).

VB

```
Public Async Sub ProcessRead()
    Dim filePath = "temp2.txt"

    If File.Exists(filePath) = False Then
        Debug.WriteLine("file not found: " & filePath)
    Else
        Try
            Dim text As String = Await ReadTextAsync(filePath)
            Debug.WriteLine(text)
        Catch ex As Exception
            Debug.WriteLine(ex.Message)
        End Try
    End If
End Sub

Private Async Function ReadTextAsync(filePath As String) As Task(Of String)

    Using sourceStream As New FileStream(filePath,
        FileMode.Open, FileAccess.Read, FileShare.Read,
        bufferSize:=4096, useAsync:=True)

        Dim sb As New StringBuilder

        Dim buffer As Byte() = New Byte(&H1000) {}
        Dim numRead As Integer
        numRead = Await sourceStream.ReadAsync(buffer, 0, buffer.Length)
        While numRead <> 0
            Dim text As String = Encoding.Unicode.GetString(buffer, 0, numRead)
            sb.Append(text)

            numRead = Await sourceStream.ReadAsync(buffer, 0, buffer.Length)
        End While

        Return sb.ToString
    End Using
End Function
```

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 **Imports** 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.

VB

```
Public Async Sub ProcessWriteMult()
    Dim folder = "tempfolder\"
    Dim tasks As New List(Of Task)
    Dim sourceStreams As New List(Of FileStream)

    Try
        For index = 1 To 10
            Dim text = "In file " & index.ToString & ControlChars.CrLf

            Dim fileName = "thefile" & index.ToString("00") & ".txt"
            Dim filePath = folder & fileName

            Dim encodedText As Byte() = Encoding.Unicode.GetBytes(text)

            Dim sourceStream As New FileStream(filePath,
                FileMode.Append, FileAccess.Write, FileShare.None,
                bufferSize:=4096, useAsync:=True)

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

            tasks.Add(theTask)
        Next

        Await Task.WhenAll(tasks)
    Finally
        For Each sourceStream As FileStream In sourceStreams
            sourceStream.Close()
        Next
    End Try
End Sub
```

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 (Visual Basic) and Cancellation in Managed Threads.

See Also

Asynchronous Programming with Async and Await (Visual Basic)
Async Return Types (Visual Basic)
Control Flow in Async Programs (Visual Basic)