Asynchronous Programming Model (APM)

.NET Framework (current version)

An asynchronous operation that uses the IAsyncResult design pattern is implemented as two methods named **Begin***OperationName* and **End***OperationName* that begin and end the asynchronous operation *OperationName* respectively. For example, the FileStream class provides the BeginRead and EndRead methods to asynchronously read bytes from a file. These methods implement the asynchronous version of the Read method.

> **Note**
>
> Starting with the .NET Framework 4, the Task Parallel Library provides a new model for asynchronous and parallel programming. For more information, see Task Parallel Library (TPL) and Task-based Asynchronous Pattern (TAP)).

After calling **Begin***OperationName*, an application can continue executing instructions on the calling thread while the asynchronous operation takes place on a different thread. For each call to **Begin***OperationName*, the application should also call **End***OperationName* to get the results of the operation.

Beginning an Asynchronous Operation

The **Begin***OperationName* method begins asynchronous operation *OperationName* and returns an object that implements the IAsyncResult interface. IAsyncResult objects store information about an asynchronous operation. The following table shows information about an asynchronous operation.

Member	Description
AsyncState	An optional application-specific object that contains information about the asynchronous operation.
AsyncWaitHandle	A WaitHandle that can be used to block application execution until the asynchronous operation completes.
CompletedSynchronously	A value that indicates whether the asynchronous operation completed on the thread used to call **Begin***OperationName* instead of completing on a separate ThreadPool thread.
IsCompleted	A value that indicates whether the asynchronous operation has completed.

A **Begin***OperationName* method takes any parameters declared in the signature of the synchronous version of the method that are passed by value or by reference. Any out parameters are not part of the **Begin***OperationName* method signature. The **Begin***OperationName* method signature also includes two additional parameters. The first of these defines an AsyncCallback delegate that references a method that is called when the asynchronous operation completes. The caller

can specify **null** (**Nothing** in Visual Basic) if it does not want a method invoked when the operation completes. The second additional parameter is a user-defined object. This object can be used to pass application-specific state information to the method invoked when the asynchronous operation completes. If a **Begin***OperationName* method takes additional operation-specific parameters, such as a byte array to store bytes read from a file, the AsyncCallback and application state object are the last parameters in the **Begin***OperationName* method signature.

Begin*OperationName* returns control to the calling thread immediately. If the **Begin***OperationName* method throws exceptions, the exceptions are thrown before the asynchronous operation is started. If the **Begin***OperationName* method throws exceptions, the callback method is not invoked.

Ending an Asynchronous Operation

The **End***OperationName* method ends asynchronous operation *OperationName*. The return value of the **End***OperationName* method is the same type returned by its synchronous counterpart and is specific to the asynchronous operation. For example, the EndRead method returns the number of bytes read from a FileStream and the EndGetHostByName method returns an IPHostEntry object that contains information about a host computer. The **End***OperationName* method takes any out or ref parameters declared in the signature of the synchronous version of the method. In addition to the parameters from the synchronous method, the **End***OperationName* method also includes an IAsyncResult parameter. Callers must pass the instance returned by the corresponding call to **Begin***OperationName*.

If the asynchronous operation represented by the IAsyncResult object has not completed when **End***OperationName* is called, **End***OperationName* blocks the calling thread until the asynchronous operation is complete. Exceptions thrown by the asynchronous operation are thrown from the **End***OperationName* method. The effect of calling the **End***OperationName* method multiple times with the same IAsyncResult is not defined. Likewise, calling the **End***OperationName* method with an IAsyncResult that was not returned by the related Begin method is also not defined.

Note
For either of the undefined scenarios, implementers should consider throwing InvalidOperationException.

Note
Implementers of this design pattern should notify the caller that the asynchronous operation completed by setting IsCompleted to true, calling the asynchronous callback method (if one was specified) and signaling the AsyncWaitHandle.

Application developers have several design choices for accessing the results of the asynchronous operation. The correct choice depends on whether the application has instructions that can execute while the operation completes. If an application cannot perform any additional work until it receives the results of the asynchronous operation, the application must block until the results are available. To block until an asynchronous operation completes, you can use one of the following approaches:

- Call **End***OperationName* from the application's main thread, blocking application execution until the operation is complete. For an example that illustrates this technique, see Blocking Application Execution by Ending an Async Operation.
- Use the AsyncWaitHandle to block application execution until one or more operations are complete. For an

example that illustrates this technique, see Blocking Application Execution Using an AsyncWaitHandle.

Applications that do not need to block while the asynchronous operation completes can use one of the following approaches:

- Poll for operation completion status by checking the IsCompleted property periodically and calling **End***OperationName* when the operation is complete. For an example that illustrates this technique, see Polling for the Status of an Asynchronous Operation.
- Use an AsyncCallback delegate to specify a method to be invoked when the operation is complete. For an example that illustrates this technique, see Using an AsyncCallback Delegate to End an Asynchronous Operation.

See Also

Event-based Asynchronous Pattern (EAP)
Calling Synchronous Methods Asynchronously
Using an AsyncCallback Delegate and State Object

Calling Asynchronous Methods Using IAsyncResult

.NET Framework (current version)

Types in the .NET Framework and third-party class libraries can provide methods that allow an application to continue executing while performing asynchronous operations in threads other than the main application thread. The following sections describe and provide code examples that demonstrate the different ways you can call asynchronous methods that use the IAsyncResult design pattern.

- Blocking Application Execution by Ending an Async Operation.
- Blocking Application Execution Using an AsyncWaitHandle.
- Polling for the Status of an Asynchronous Operation.
- Using an AsyncCallback Delegate to End an Asynchronous Operation.

See Also

Event-based Asynchronous Pattern (EAP)
Event-based Asynchronous Pattern Overview

Blocking Application Execution by Ending an Async Operation

.NET Framework (current version)

Applications that cannot continue to do other work while waiting for the results of an asynchronous operation must block until the operation completes. Use one of the following options to block your application's main thread while waiting for an asynchronous operation to complete:

- Call the asynchronous operations **End***OperationName* method. This approach is demonstrated in this topic.
- Use the AsyncWaitHandle property of the IAsyncResult returned by the asynchronous operation's **Begin***OperationName* method. For an example that demonstrates this approach, see Blocking Application Execution Using an AsyncWaitHandle.

Applications that use the **End***OperationName* method to block until an asynchronous operation is complete will typically call the **Begin***OperationName* method, perform any work that can be done without the results of the operation, and then call **End***OperationName*.

Example

The following code example demonstrates using asynchronous methods in the Dns class to retrieve Domain Name System information for a user-specified computer. Note that **null** (**Nothing** in Visual Basic) is passed for the BeginGetHostByName *requestCallback* and *stateObject* parameters because these arguments are not required when using this approach.

VB

```
' The following example demonstrates using asynchronous methods to
' get Domain Name System information for the specified host computer.

Imports System
Imports System.Net
Imports System.Net.Sockets

Namespace Examples.AdvancedProgramming.AsynchronousOperations
    Public Class BlockUntilOperationCompletes
        Public Shared Sub Main(args() as String)
            ' Make sure the caller supplied a host name.
            If(args.Length = 0)
                ' Print a message and exit.
                Console.WriteLine("You must specify the name of a host computer.")
                End
            End If
            ' Start the asynchronous request for DNS information.
            ' This example does not use a delegate or user-supplied object
```

```
            ' so the last two arguments are Nothing.
            Dim result as IAsyncResult = Dns.BeginGetHostEntry(args(0), Nothing,
Nothing)
            Console.WriteLine("Processing your request for information...")
            ' Do any additional work that can be done here.
            Try
                ' EndGetHostByName blocks until the process completes.
                Dim host as IPHostEntry = Dns.EndGetHostEntry(result)
                Dim  aliases() as String = host.Aliases
                Dim addresses() as IPAddress= host.AddressList
                Dim i as Integer
                If aliases.Length > 0
                    Console.WriteLine("Aliases")
                    For i = 0 To aliases.Length -1
                        Console.WriteLine("{0}", aliases(i))
                    Next i
                End If
                If addresses.Length > 0
                    Console.WriteLine("Addresses")
                    For i = 0 To addresses.Length -1
                        Console.WriteLine("{0}", addresses(i).ToString())
                    Next i
                End If
            Catch e as SocketException
                  Console.WriteLine("An exception occurred while processing the
request: {0}", e.Message)
            End Try
        End Sub
    End Class
End Namespace
```

See Also

Event-based Asynchronous Pattern (EAP)
Event-based Asynchronous Pattern Overview

Blocking Application Execution Using an AsyncWaitHandle

.NET Framework (current version)

Applications that cannot continue to do other work while waiting for the results of an asynchronous operation must block until the operation completes. Use one of the following options to block your application's main thread while waiting for an asynchronous operation to complete:

- Use the AsyncWaitHandle property of the IAsyncResult returned by the asynchronous operation's **Begin***OperationName* method. This approach is demonstrated in this topic.
- Call the asynchronous operation's **End***OperationName* method. For an example that demonstrates this approach, see Blocking Application Execution by Ending an Async Operation.

Applications that use one or more WaitHandle objects to block until an asynchronous operation is complete will typically call the **Begin***OperationName* method, perform any work that can be done without the results of the operation, and then block until the asynchronous operation(s) completes. An application can block on a single operation by calling one of the WaitOne methods using the AsyncWaitHandle. To block while waiting for a set of asynchronous operations to complete, store the associated AsyncWaitHandle objects in an array and call one of the WaitAll methods. To block while waiting for any one of a set of asynchronous operations to complete, store the associated AsyncWaitHandle objects in an array and call one of the WaitAny methods.

Example

The following code example demonstrates using asynchronous methods in the DNS class to retrieve Domain Name System information for a user-specified computer. The example demonstrates blocking using the WaitHandle associated with the asynchronous operation. Note that **null** (**Nothing** in Visual Basic) is passed for the BeginGetHostByName *requestCallback* and *stateObject* parameters because these are not required when using this approach.

VB

```
' The following example demonstrates using asynchronous methods to
' get Domain Name System information for the specified host computer.

Imports System
Imports System.Net
Imports System.Net.Sockets
Imports System.Threading

namespace Examples.AdvancedProgramming.AsynchronousOperations
    Public Class WaitUntilOperationCompletes

        Public Shared Sub Main(args() as String)
            ' Make sure the caller supplied a host name.
```

```
            If(args.Length = 0)
                ' Print a message and exit.
                Console.WriteLine("You must specify the name of a host computer.")
                End
            End If
            ' Start the asynchronous request for DNS information.
            Dim result as IAsyncResult= Dns.BeginGetHostEntry(args(0), Nothing, Nothing)
            Console.WriteLine("Processing request for information...")
            ' Wait until the operation completes.
            result.AsyncWaitHandle.WaitOne()
            ' The operation completed. Process the results.
            Try
                ' Get the results.
                Dim host as IPHostEntry = Dns.EndGetHostEntry(result)
                Dim  aliases() as String = host.Aliases
                Dim addresses() as IPAddress= host.AddressList
                Dim i as Integer
                If aliases.Length > 0
                    Console.WriteLine("Aliases")
                    For i = 0 To aliases.Length -1
                        Console.WriteLine("{0}", aliases(i))
                    Next i
                End If
                If addresses.Length > 0
                    Console.WriteLine("Addresses")
                    For i = 0 To addresses.Length -1
                        Console.WriteLine("{0}", addresses(i).ToString())
                    Next i
                End If
            Catch e as SocketException
                  Console.WriteLine("An exception occurred while processing the
request: {0}" _
                    , e.Message)
            End Try
        End Sub
    End Class
End Namespace
```

See Also

Event-based Asynchronous Pattern (EAP)
Event-based Asynchronous Pattern Overview

Polling for the Status of an Asynchronous Operation

.NET Framework (current version)

Applications that can do other work while waiting for the results of an asynchronous operation should not block waiting until the operation completes. Use one of the following options to continue executing instructions while waiting for an asynchronous operation to complete:

- Use the IsCompleted property of the IAsyncResult returned by the asynchronous operation's **Begin***OperationName* method to determine whether the operation has completed. This approach is known as polling and is demonstrated in this topic.
- Use an AsyncCallback delegate to process the results of the asynchronous operation in a separate thread. For an example that demonstrates this approach, see Using an AsyncCallback Delegate to End an Asynchronous Operation.

Example

The following code example demonstrates using asynchronous methods in the Dns class to retrieve Domain Name System information for a user-specified computer. This example starts the asynchronous operation and then prints periods (".") at the console until the operation is complete. Note that **null** (**Nothing** in Visual Basic) is passed for the BeginGetHostByName AsyncCallback and Object parameters because these arguments are not required when using this approach.

VB

```
'The following example demonstrates using asynchronous methods to
'get Domain Name System information for the specified host computer.
'This example polls to detect the end of the asynchronous operation.

Imports System
Imports System.Net
Imports System.Net.Sockets
Imports System.Threading

Namespace Examples.AdvancedProgramming.AsynchronousOperations

    Public Class PollUntilOperationCompletes
        Shared Sub UpdateUserInterface()
            ' Print a period to indicate that the application
            ' is still working on the request.
            Console.Write(".")
        End Sub
        Public Shared Sub Main(args() as String)
            ' Make sure the caller supplied a host name.
```

```
            If(args.Length = 0)
                ' Print a message and exit.
                Console.WriteLine("You must specify the name of a host computer.")
                End
            End If
            ' Start the asynchronous request for DNS information.
            Dim result as IAsyncResult= Dns.BeginGetHostEntry(args(0), Nothing, Nothing)
            Console.WriteLine("Processing request for information...")

            ' Poll for completion information.
            ' Print periods (".") until the operation completes.
            Do while result.IsCompleted <> True
                UpdateUserInterface()
            Loop
            ' The operation is complete. Process the results.
            ' Print a new line.
            Console.WriteLine()
            Try
                Dim host as IPHostEntry = Dns.EndGetHostEntry(result)
                Dim  aliases() as String = host.Aliases
                Dim addresses() as IPAddress = host.AddressList
                Dim i as Integer
                If aliases.Length > 0
                    Console.WriteLine("Aliases")
                    For i = 0 To aliases.Length -1
                        Console.WriteLine("{0}", aliases(i))
                    Next i
                End If
                If addresses.Length > 0
                    Console.WriteLine("Addresses")
                    For i = 0 To addresses.Length -1
                        Console.WriteLine("{0}", addresses(i).ToString())
                    Next i
                End If
            Catch e as SocketException
                  Console.WriteLine("An exception occurred while processing the
request: {0}", e.Message)
            End Try
        End Sub
    End Class
End Namespace
```

See Also

Event-based Asynchronous Pattern (EAP)
Event-based Asynchronous Pattern Overview

Using an AsyncCallback Delegate to End an Asynchronous Operation

.NET Framework (current version)

Applications that can do other work while waiting for the results of an asynchronous operation should not block waiting until the operation completes. Use one of the following options to continue executing instructions while waiting for an asynchronous operation to complete:

- Use an AsyncCallback delegate to process the results of the asynchronous operation in a separate thread. This approach is demonstrated in this topic.
- Use the IsCompleted property of the IAsyncResult returned by the asynchronous operation's **Begin***OperationName* method to determine whether the operation has completed. For an example that demonstrates this approach, see Polling for the Status of an Asynchronous Operation.

Example

The following code example demonstrates using asynchronous methods in the Dns class to retrieve Domain Name System (DNS) information for user-specified computers. This example creates an AsyncCallback delegate that references the `ProcessDnsInformation` method. This method is called once for each asynchronous request for DNS information.

Note that the user-specified host is passed to the BeginGetHostByName Object parameter. For an example that demonstrates defining and using a more complex state object, see Using an AsyncCallback Delegate and State Object.

VB

```
'The following example demonstrates using asynchronous methods to
'get Domain Name System information for the specified host computers.
'This example uses a delegate to obtain the results of each asynchronous
'operation.

Imports System
Imports System.Net
Imports System.Net.Sockets
Imports System.Threading
Imports System.Collections.Specialized
Imports System.Collections

Namespace Examples.AdvancedProgramming.AsynchronousOperations

    Public Class UseDelegateForAsyncCallback

        Dim Shared requestCounter as Integer
        Dim Shared hostData as ArrayList = new ArrayList()
```

```
        Dim Shared hostNames as StringCollection = new StringCollection()
        Shared Sub UpdateUserInterface()

            ' Print a message to indicate that the application
            ' is still working on the remaining requests.
            Console.WriteLine("{0} requests remaining.", requestCounter)
        End Sub
        Public Shared Sub Main()
            ' Create the delegate that will process the results of the
            ' asynchronous request.
            Dim callBack as AsyncCallback
            Dim host as string
            Dim i, j, k as Integer
            callback = AddressOf ProcessDnsInformation
            Do
                Console.Write(" Enter the name of a host computer or <enter> to finish:
")
                host = Console.ReadLine()
                If host.Length > 0
                    ' Increment the request counter in a thread safe manner.
                    Interlocked.Increment(requestCounter)
                    ' Start the asynchronous request for DNS information.
                    Dns.BeginGetHostEntry(host, callBack, host)
                End If
            Loop While (host.Length > 0)

            ' The user has entered all of the host names for lookup.
            ' Now wait until the threads complete.
            Do While requestCounter > 0

                UpdateUserInterface()
            Loop

            ' Display the results.
            For i = 0 To hostNames.Count -1
                Dim dataObject as Object = hostData (i)
                Dim message as String

                ' Was a SocketException was thrown?
                If TypeOf dataObject is String
                    message = CType(dataObject, String)
                    Console.WriteLine("Request for {0} returned message: {1}", _
                        hostNames(i), message)
                Else
                    ' Get the results.
                    Dim h as IPHostEntry = CType(dataObject, IPHostEntry)
                    Dim aliases() as String = h.Aliases
                    Dim addresses() as IPAddress = h.AddressList
                    If aliases.Length > 0
                        Console.WriteLine("Aliases for 0}", hostNames(i))
                        For j = 0 To aliases.Length -1
                            Console.WriteLine("{0}", aliases(j))
                        Next j
                    End If
```

```
                    If addresses.Length > 0
                        Console.WriteLine("Addresses for {0}", hostNames(i))
                        For k = 0 To addresses.Length -1
                            Console.WriteLine("{0}",addresses(k).ToString())
                        Next k
                    End If
                End If
            Next i
        End Sub

        ' The following method is called when each asynchronous operation completes.
        Shared Sub ProcessDnsInformation(result as IAsyncResult)

            Dim hostName as String = CType(result.AsyncState, String)
            hostNames.Add(hostName)
            Try
                ' Get the results.
                Dim host as IPHostEntry = Dns.EndGetHostEntry(result)
                hostData.Add(host)
            ' Store the exception message.
            Catch e as SocketException
                hostData.Add(e.Message)
            Finally
                ' Decrement the request counter in a thread-safe manner.
                Interlocked.Decrement(requestCounter)
            End Try
        End Sub
    End Class
End Namespace
```

See Also

Event-based Asynchronous Pattern (EAP)
Event-based Asynchronous Pattern Overview
Calling Asynchronous Methods Using IAsyncResult
Using an AsyncCallback Delegate and State Object

Asynchronous Programming Using Delegates

.NET Framework (current version)

Delegates enable you to call a synchronous method in an asynchronous manner. When you call a delegate synchronously, the **Invoke** method calls the target method directly on the current thread. If the **BeginInvoke** method is called, the common language runtime (CLR) queues the request and returns immediately to the caller. The target method is called asynchronously on a thread from the thread pool. The original thread, which submitted the request, is free to continue executing in parallel with the target method. If a callback method has been specified in the call to the **BeginInvoke** method, the callback method is called when the target method ends. In the callback method, the **EndInvoke** method obtains the return value and any input/output or output-only parameters. If no callback method is specified when calling **BeginInvoke**, **EndInvoke** can be called from the thread that called **BeginInvoke**.

Important

Compilers should emit delegate classes with **Invoke**, **BeginInvoke**, and **EndInvoke** methods using the delegate signature specified by the user. The **BeginInvoke** and **EndInvoke** methods should be decorated as native. Because these methods are marked as native, the CLR automatically provides the implementation at class load time. The loader ensures that they are not overridden.

In This Section

Calling Synchronous Methods Asynchronously
: Discusses the use of delegates to make asynchronous calls to ordinary methods, and provides simple code examples that show the four ways to wait for an asynchronous call to return.

Related Sections

Event-based Asynchronous Pattern (EAP)
: Describes asynchronous programming with the .NET Framework.

See Also

Delegate

Calling Synchronous Methods Asynchronously

.NET Framework (current version)

The .NET Framework enables you to call any method asynchronously. To do this you define a delegate with the same signature as the method you want to call; the common language runtime automatically defines **BeginInvoke** and **EndInvoke** methods for this delegate, with the appropriate signatures.

> **Note**
>
> Asynchronous delegate calls, specifically the **BeginInvoke** and **EndInvoke** methods, are not supported in the .NET Compact Framework.

The **BeginInvoke** method initiates the asynchronous call. It has the same parameters as the method that you want to execute asynchronously, plus two additional optional parameters. The first parameter is an AsyncCallback delegate that references a method to be called when the asynchronous call completes. The second parameter is a user-defined object that passes information into the callback method. **BeginInvoke** returns immediately and does not wait for the asynchronous call to complete. **BeginInvoke** returns an IAsyncResult, which can be used to monitor the progress of the asynchronous call.

The **EndInvoke** method retrieves the results of the asynchronous call. It can be called any time after **BeginInvoke**. If the asynchronous call has not completed, **EndInvoke** blocks the calling thread until it completes. The parameters of **EndInvoke** include the **out** and **ref** parameters (<Out> **ByRef** and **ByRef** in Visual Basic) of the method that you want to execute asynchronously, plus the IAsyncResult returned by **BeginInvoke**.

> **Note**
>
> The IntelliSense feature in Visual Studio 2005 displays the parameters of **BeginInvoke** and **EndInvoke**. If you are not using Visual Studio or a similar tool, or if you are using C# with Visual Studio 2005, see Asynchronous Programming Model (APM) for a description of the parameters defined for these methods.

The code examples in this topic demonstrate four common ways to use **BeginInvoke** and **EndInvoke** to make asynchronous calls. After calling **BeginInvoke** you can do the following:

- Do some work and then call **EndInvoke** to block until the call completes.
- Obtain a WaitHandle using the IAsyncResult.AsyncWaitHandle property, use its WaitOne method to block execution until the WaitHandle is signaled, and then call **EndInvoke**.
- Poll the IAsyncResult returned by **BeginInvoke** to determine when the asynchronous call has completed, and then call **EndInvoke**.
- Pass a delegate for a callback method to **BeginInvoke**. The method is executed on a ThreadPool thread when the

asynchronous call completes. The callback method calls **EndInvoke**.

Important
No matter which technique you use, always call **EndInvoke** to complete your asynchronous call.

Defining the Test Method and Asynchronous Delegate

The code examples that follow demonstrate various ways of calling the same long-running method, `TestMethod`, asynchronously. The `TestMethod` method displays a console message to show that it has begun processing, sleeps for a few seconds, and then ends. `TestMethod` has an **out** parameter to demonstrate the way such parameters are added to the signatures of **BeginInvoke** and **EndInvoke**. You can handle **ref** parameters similarly.

The following code example shows the definition of `TestMethod` and the delegate named `AsyncMethodCaller` that can be used to call `TestMethod` asynchronously. To compile the code examples, you must include the definitions for `TestMethod` and the `AsyncMethodCaller` delegate.

VB

```
Imports System
Imports System.Threading
Imports System.Runtime.InteropServices

Namespace Examples.AdvancedProgramming.AsynchronousOperations
    Public Class AsyncDemo
        ' The method to be executed asynchronously.
        Public Function TestMethod(ByVal callDuration As Integer, _
                <Out> ByRef threadId As Integer) As String
            Console.WriteLine("Test method begins.")
            Thread.Sleep(callDuration)
            threadId = Thread.CurrentThread.ManagedThreadId()
            return String.Format("My call time was {0}.", callDuration.ToString())
        End Function
    End Class

    ' The delegate must have the same signature as the method
    ' it will call asynchronously.
    Public Delegate Function AsyncMethodCaller(ByVal callDuration As Integer, _
        <Out> ByRef threadId As Integer) As String
End Namespace
```

Waiting for an Asynchronous Call with EndInvoke

The simplest way to execute a method asynchronously is to start executing the method by calling the delegate's **BeginInvoke** method, do some work on the main thread, and then call the delegate's **EndInvoke** method. **EndInvoke** might block the calling thread because it does not return until the asynchronous call completes. This is a good technique

to use with file or network operations.

Important

Because **EndInvoke** might block, you should never call it from threads that service the user interface.

VB

```
Imports System
Imports System.Threading
Imports System.Runtime.InteropServices

Namespace Examples.AdvancedProgramming.AsynchronousOperations
    Public Class AsyncMain
        Shared Sub Main()
            ' The asynchronous method puts the thread id here.
            Dim threadId As Integer

            ' Create an instance of the test class.
            Dim ad As New AsyncDemo()

            ' Create the delegate.
            Dim caller As New AsyncMethodCaller(AddressOf ad.TestMethod)

            ' Initiate the asynchronous call.
            Dim result As IAsyncResult = caller.BeginInvoke(3000, _
                threadId, Nothing, Nothing)

            Thread.Sleep(0)
            Console.WriteLine("Main thread {0} does some work.", _
                 Thread.CurrentThread.ManagedThreadId)

            ' Call EndInvoke to Wait for the asynchronous call to complete,
            ' and to retrieve the results.
            Dim returnValue As String = caller.EndInvoke(threadId, result)

            Console.WriteLine("The call executed on thread {0}, with return value
""{1}"".", _
                threadId, returnValue)
        End Sub
    End Class

End Namespace

'This example produces output similar to the following:
'
'Main thread 1 does some work.
'Test method begins.
'The call executed on thread 3, with return value "My call time was 3000.".
```

Waiting for an Asynchronous Call with WaitHandle

You can obtain a WaitHandle by using the AsyncWaitHandle property of the IAsyncResult returned by **BeginInvoke**. The WaitHandle is signaled when the asynchronous call completes, and you can wait for it by calling the WaitOne method.

If you use a WaitHandle, you can perform additional processing before or after the asynchronous call completes, but before calling **EndInvoke** to retrieve the results.

Note

The wait handle is not closed automatically when you call **EndInvoke**. If you release all references to the wait handle, system resources are freed when garbage collection reclaims the wait handle. To free the system resources as soon as you are finished using the wait handle, dispose of it by calling the WaitHandle.Close method. Garbage collection works more efficiently when disposable objects are explicitly disposed.

VB

```
Imports System
Imports System.Threading
Imports System.Runtime.InteropServices

Namespace Examples.AdvancedProgramming.AsynchronousOperations

    Public Class AsyncMain
        Shared Sub Main()
            ' The asynchronous method puts the thread id here.
            Dim threadId As Integer

            ' Create an instance of the test class.
            Dim ad As New AsyncDemo()

            ' Create the delegate.
            Dim caller As New AsyncMethodCaller(AddressOf ad.TestMethod)

            ' Initiate the asynchronous call.
            Dim result As IAsyncResult = caller.BeginInvoke(3000, _
                threadId, Nothing, Nothing)

            Thread.Sleep(0)
            Console.WriteLine("Main thread {0} does some work.", _
                Thread.CurrentThread.ManagedThreadId)
            ' Perform additional processing here and then
            ' wait for the WaitHandle to be signaled.
            result.AsyncWaitHandle.WaitOne()

            ' Call EndInvoke to retrieve the results.
            Dim returnValue As String = caller.EndInvoke(threadId, result)

            ' Close the wait handle.
            result.AsyncWaitHandle.Close()
```

```
            Console.WriteLine("The call executed on thread {0}, with return value
""{1}"".", _
                threadId, returnValue)
        End Sub
    End Class
End Namespace

'This example produces output similar to the following:
'
'Main thread 1 does some work.
'Test method begins.
'The call executed on thread 3, with return value "My call time was 3000.".
```

Polling for Asynchronous Call Completion

You can use the IsCompleted property of the IAsyncResult returned by **BeginInvoke** to discover when the asynchronous call completes. You might do this when making the asynchronous call from a thread that services the user interface. Polling for completion allows the calling thread to continue executing while the asynchronous call executes on a ThreadPool thread.

VB

```
Imports System
Imports System.Threading
Imports System.Runtime.InteropServices

Namespace Examples.AdvancedProgramming.AsynchronousOperations

    Public Class AsyncMain
        Shared Sub Main()
            ' The asynchronous method puts the thread id here.
            Dim threadId As Integer

            ' Create an instance of the test class.
            Dim ad As New AsyncDemo()

            ' Create the delegate.
            Dim caller As New AsyncMethodCaller(AddressOf ad.TestMethod)

            ' Initiate the asynchronous call.
            Dim result As IAsyncResult = caller.BeginInvoke(3000, _
                threadId, Nothing, Nothing)

            ' Poll while simulating work.
            While result.IsCompleted = False
                Thread.Sleep(250)
                Console.Write(".")
            End While

            ' Call EndInvoke to retrieve the results.
```

```
            Dim returnValue As String = caller.EndInvoke(threadId, result)

            Console.WriteLine(vbCrLf & _
                "The call executed on thread {0}, with return value ""{1}"".", _
                threadId, returnValue)
        End Sub
    End Class
End Namespace

' This example produces output similar to the following:
'
'Test method begins.
'.............
'The call executed on thread 3, with return value "My call time was 3000.".
```

Executing a Callback Method When an Asynchronous Call Completes

If the thread that initiates the asynchronous call does not need to be the thread that processes the results, you can execute a callback method when the call completes. The callback method is executed on a ThreadPool thread.

To use a callback method, you must pass **BeginInvoke** an AsyncCallback delegate that represents the callback method. You can also pass an object that contains information to be used by the callback method. In the callback method, you can cast the IAsyncResult, which is the only parameter of the callback method, to an AsyncResult object. You can then use the AsyncResult.AsyncDelegate property to get the delegate that was used to initiate the call so that you can call **EndInvoke**.

Notes on the example:

- The *threadId* parameter of `TestMethod` is an **out** parameter (<Out> **ByRef** in Visual Basic), so its input value is never used by `TestMethod`. A dummy variable is passed to the **BeginInvoke** call. If the *threadId* parameter were a **ref** parameter (**ByRef** in Visual Basic), the variable would have to be a class-level field so that it could be passed to both **BeginInvoke** and **EndInvoke**.
- The state information that is passed to **BeginInvoke** is a format string, which the callback method uses to format an output message. Because it is passed as type Object, the state information must be cast to its proper type before it can be used.
- The callback is made on a ThreadPool thread. ThreadPool threads are background threads, which do not keep the application running if the main thread ends, so the main thread of the example has to sleep long enough for the callback to finish.

VB

```
Imports System
Imports System.Threading
Imports System.Runtime.Remoting.Messaging

Namespace Examples.AdvancedProgramming.AsynchronousOperations
```

```
    Public Class AsyncMain

        Shared Sub Main()

            ' Create an instance of the test class.
            Dim ad As New AsyncDemo()

            ' Create the delegate.
            Dim caller As New AsyncMethodCaller(AddressOf ad.TestMethod)

            ' The threadId parameter of TestMethod is an <Out> parameter, so
            ' its input value is never used by TestMethod. Therefore, a dummy
            ' variable can be passed to the BeginInvoke call. If the threadId
            ' parameter were a ByRef parameter, it would have to be a class-
            ' level field so that it could be passed to both BeginInvoke and
            ' EndInvoke.
            Dim dummy As Integer = 0

            ' Initiate the asynchronous call, passing three seconds (3000 ms)
            ' for the callDuration parameter of TestMethod; a dummy variable
            ' for the <Out> parameter (threadId); the callback delegate; and
            ' state information that can be retrieved by the callback method.
            ' In this case, the state information is a string that can be used
            ' to format a console message.
            Dim result As IAsyncResult = caller.BeginInvoke(3000, _
                dummy, _
                AddressOf CallbackMethod, _
                "The call executed on thread {0}, with return value ""{1}""." )

            Console.WriteLine("The main thread {0} continues to execute...", _
                Thread.CurrentThread.ManagedThreadId)

            ' The callback is made on a ThreadPool thread. ThreadPool threads
            ' are background threads, which do not keep the application running
            ' if the main thread ends. Comment out the next line to demonstrate
            ' this.
            Thread.Sleep(4000)

            Console.WriteLine("The main thread ends.")
        End Sub

        ' The callback method must have the same signature as the
        ' AsyncCallback delegate.
        Shared Sub CallbackMethod(ByVal ar As IAsyncResult)
            ' Retrieve the delegate.
            Dim result As AsyncResult = CType(ar, AsyncResult)
            Dim caller As AsyncMethodCaller = CType(result.AsyncDelegate,
AsyncMethodCaller)

            ' Retrieve the format string that was passed as state
            ' information.
            Dim formatString As String = CType(ar.AsyncState, String)

            ' Define a variable to receive the value of the <Out> parameter.
```

```
            ' If the parameter were ByRef rather than <Out> then it would have to
            ' be a class-level field so it could also be passed to BeginInvoke.
            Dim threadId As Integer = 0

            ' Call EndInvoke to retrieve the results.
            Dim returnValue As String = caller.EndInvoke(threadId, ar)

            ' Use the format string to format the output message.
            Console.WriteLine(formatString, threadId, returnValue)
        End Sub
    End Class
End Namespace

' This example produces output similar to the following:
'
'The main thread 1 continues to execute...
'Test method begins.
'The call executed on thread 3, with return value "My call time was 3000.".
'The main thread ends.
```

See Also

Delegate
Event-based Asynchronous Pattern (EAP)