

Asynchronous Programming Patterns	1
Asynchronous Programming Model (APM)	3
Event-based Asynchronous Pattern (EAP)	6
Task-based Asynchronous Pattern (TAP)	8

Asynchronous Programming Patterns

.NET Framework (current version)

The .NET Framework provides three patterns for performing asynchronous operations:

- Asynchronous Programming Model (APM) pattern (also called the [IAsyncResult](#) pattern), where asynchronous operations require [Begin](#) and [End](#) methods (for example, [BeginWrite](#) and [EndWrite](#) for asynchronous write operations). This pattern is no longer recommended for new development. For more information, see [Asynchronous Programming Model \(APM\)](#).
- Event-based Asynchronous Pattern (EAP), which requires a method that has the [Async](#) suffix, and also requires one or more events, event handler delegate types, and [EventArgs](#)-derived types. EAP was introduced in the .NET Framework 2.0. It is no longer recommended for new development. For more information, see [Event-based Asynchronous Pattern \(EAP\)](#).
- Task-based Asynchronous Pattern (TAP), which uses a single method to represent the initiation and completion of an asynchronous operation. TAP was introduced in the .NET Framework 4 and is the recommended approach to asynchronous programming in the .NET Framework. The [async](#) and [await](#) keywords in C# and the [Async](#) and [Await](#) operators in Visual Basic Language add language support for TAP. For more information, see [Task-based Asynchronous Pattern \(TAP\)](#).

Comparing Patterns

For a quick comparison of how the three patterns model asynchronous operations, consider a **Read** method that reads a specified amount of data into a provided buffer starting at a specified offset:

C#

```
public class MyClass
{
    public int Read(byte [] buffer, int offset, int count);
}
```

The APM counterpart of this method would expose the [BeginRead](#) and [EndRead](#) methods:

C#

```
public class MyClass
{
    public IAsyncResult BeginRead(
        byte [] buffer, int offset, int count,
        AsyncCallback callback, object state);
    public int EndRead(IAsyncResult asyncResult);
}
```

The EAP counterpart would expose the following set of types and members:

C#

```
public class MyClass
{
    public void ReadAsync(byte [] buffer, int offset, int count);
    public event ReadCompletedEventHandler ReadCompleted;
}
```

The TAP counterpart would expose the following single `ReadAsync` method:

C#

```
public class MyClass
{
    public Task<int> ReadAsync(byte [] buffer, int offset, int count);
}
```

For a comprehensive discussion of TAP, APM, and EAP, see the links provided in the next section.

Related Topics

Title	Description
Asynchronous Programming Model (APM)	Describes the legacy model that uses the IAsyncResult interface to provide asynchronous behavior. This model is no longer recommended for new development.
Event-based Asynchronous Pattern (EAP)	Describes the event-based legacy model for providing asynchronous behavior. This model is no longer recommended for new development.
Task-based Asynchronous Pattern (TAP)	Describes the new asynchronous pattern based on the System.Threading.Tasks namespace. This model is the recommended approach to asynchronous programming in the .NET Framework 4 and later versions.

Asynchronous Programming Model (APM)

.NET Framework (current version)

An asynchronous operation that uses the [IAsyncResult](#) design pattern is implemented as two methods named **BeginOperationName** and **EndOperationName** 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 **BeginOperationName**, an application can continue executing instructions on the calling thread while the asynchronous operation takes place on a different thread. For each call to **BeginOperationName**, the application should also call **EndOperationName** to get the results of the operation.

Beginning an Asynchronous Operation

The **BeginOperationName** 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 BeginOperationName instead of completing on a separate ThreadPool thread.
IsCompleted	A value that indicates whether the asynchronous operation has completed.

A **BeginOperationName** 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 **BeginOperationName** method signature. The **BeginOperationName** 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 **BeginOperationName** 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 **BeginOperationName** method signature.

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

Ending an Asynchronous Operation

The **EndOperationName** method ends asynchronous operation *OperationName*. The return value of the **EndOperationName** 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 **EndOperationName** 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 **EndOperationName** method also includes an [IAsyncResult](#) parameter. Callers must pass the instance returned by the corresponding call to **BeginOperationName**.

If the asynchronous operation represented by the [IAsyncResult](#) object has not completed when **EndOperationName** is called, **EndOperationName** blocks the calling thread until the asynchronous operation is complete. Exceptions thrown by the asynchronous operation are thrown from the **EndOperationName** method. The effect of calling the **EndOperationName** method multiple times with the same [IAsyncResult](#) is not defined. Likewise, calling the **EndOperationName** 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 **EndOperationName** 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 **EndOperationName** 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](#)

© 2016 Microsoft

Event-based Asynchronous Pattern (EAP)

.NET Framework (current version)

There are a number of ways to expose asynchronous features to client code. The Event-based Asynchronous Pattern prescribes one way for classes to present asynchronous behavior.

Note

Starting with the .NET Framework 4, the Task Parallel Library provides a new model for asynchronous and parallel programming. For more information, see [Parallel Programming in the .NET Framework](#).

In This Section

[Event-based Asynchronous Pattern Overview](#)

Describes how the Event-based Asynchronous Pattern makes available the advantages of multithreaded applications while hiding many of the complex issues inherent in multithreaded design.

[Implementing the Event-based Asynchronous Pattern](#)

Describes the standardized way to package a class that has asynchronous features.

[Best Practices for Implementing the Event-based Asynchronous Pattern](#)

Describes the requirements for exposing asynchronous features according to the Event-based Asynchronous Pattern.

[Deciding When to Implement the Event-based Asynchronous Pattern](#)

Describes how to determine when you should choose to implement the Event-based Asynchronous Pattern instead of the [IAsyncResult](#) pattern.

[Walkthrough: Implementing a Component That Supports the Event-based Asynchronous Pattern](#)

Illustrates how to create a component that implements the Event-based Asynchronous Pattern. It is implemented using helper classes from the [System.ComponentModel](#) namespace, which ensures that the component works correctly under any application model.

[How to: Use Components That Support the Event-based Asynchronous Pattern](#)

Describes how to use a component that supports the Event-based Asynchronous Pattern.

Reference

[AsyncOperation](#)

Describes the [AsyncOperation](#) class and has links to all its members.

[AsyncOperationManager](#)

Describes the [AsyncOperationManager](#) class and has links to all its members.

[BackgroundWorker](#)

Describes the [BackgroundWorker](#) component and has links to all its members.

Related Sections

[Task Parallel Library \(TPL\)](#)

Describes a programming model for asynchronous and parallel operations.

[Managed Threading](#)

Describes multithreading features in the .NET Framework.

[Threading \(C# and Visual Basic\)](#)

Describes multithreading features in the C# and Visual Basic languages.

See Also

[Managed Threading Best Practices](#)

[Handling and Raising Events](#)

[Multithreading in Components](#)

[Asynchronous Programming Design Patterns](#)

© 2016 Microsoft

Task-based Asynchronous Pattern (TAP)

.NET Framework (current version)

The Task-based Asynchronous Pattern (TAP) is based on the [System.Threading.Tasks.Task](#) and [System.Threading.Tasks.Task\(Of TResult\)](#) types in the [System.Threading.Tasks](#) namespace, which are used to represent arbitrary asynchronous operations. TAP is the recommended asynchronous design pattern for new development.

Naming, Parameters, and Return Types

TAP uses a single method to represent the initiation and completion of an asynchronous operation. This is in contrast to the Asynchronous Programming Model (APM or **IAsyncResult**) pattern, which requires **Begin** and **End** methods, and in contrast to the Event-based Asynchronous Pattern (EAP), which requires a method that has the **Async** suffix and also requires one or more events, event handler delegate types, and **EventArgs**-derived types. Asynchronous methods in TAP include the **Async** suffix after the operation name; for example, **GetAsync** for a get operation. If you're adding a TAP method to a class that already contains that method name with the **Async** suffix, use the suffix **TaskAsync** instead. For example, if the class already has a **GetAsync** method, use the name **GetTaskAsync**.

The TAP method returns either a [System.Threading.Tasks.Task](#) or a [System.Threading.Tasks.Task\(Of TResult\)](#), based on whether the corresponding synchronous method returns void or a type **TResult**.

The parameters of a TAP method should match the parameters of its synchronous counterpart, and should be provided in the same order. However, *out* and *ref* parameters are exempt from this rule and should be avoided entirely. Any data that would have been returned through an *out* or *ref* parameter should instead be returned as part of the **TResult** returned by [Task\(Of TResult\)](#), and should use a tuple or a custom data structure to accommodate multiple values. Methods that are devoted exclusively to the creation, manipulation, or combination of tasks (where the asynchronous intent of the method is clear in the method name or in the name of the type to which the method belongs) need not follow this naming pattern; such methods are often referred to as *combinators*. Examples of combinators include [WhenAll](#) and [WhenAny](#), and are discussed in the [Using the Built-in Task-based Combinators](#) section of the article [Consuming the Task-based Asynchronous Pattern](#).

For examples of how the TAP syntax differs from the syntax used in legacy asynchronous programming patterns such as the Asynchronous Programming Model (APM) and the Event-based Asynchronous Pattern (EAP), see [Asynchronous Programming Patterns](#).

Initiating an Asynchronous Operation

An asynchronous method that is based on TAP can do a small amount of work synchronously, such as validating arguments and initiating the asynchronous operation, before it returns the resulting task. Synchronous work should be kept to the minimum so the asynchronous method can return quickly. Reasons for a quick return include the following:

- Asynchronous methods may be invoked from user interface (UI) threads, and any long-running synchronous work could harm the responsiveness of the application.
- Multiple asynchronous methods may be launched concurrently. Therefore, any long-running work in the

synchronous portion of an asynchronous method could delay the initiation of other asynchronous operations, thereby decreasing the benefits of concurrency.

In some cases, the amount of work required to complete the operation is less than the amount of work required to launch the operation asynchronously. Reading from a stream where the read operation can be satisfied by data that is already buffered in memory is an example of such a scenario. In such cases, the operation may complete synchronously, and may return a task that has already been completed.

Exceptions

An asynchronous method should raise an exception to be thrown out of the asynchronous method call only in response to a usage error. Usage errors should never occur in production code. For example, if passing a null reference (**Nothing** in Visual Basic) as one of the method's arguments causes an error state (usually represented by an [ArgumentNullException](#) exception), you can modify the calling code to ensure that a null reference is never passed. For all other errors, exceptions that occur when an asynchronous method is running should be assigned to the returned task, even if the asynchronous method happens to complete synchronously before the task is returned. Typically, a task contains at most one exception. However, if the task represents multiple operations (for example, [WhenAll](#)), multiple exceptions may be associated with a single task.

Target Environment

When you implement a TAP method, you can determine where asynchronous execution occurs. You may choose to execute the workload on the thread pool, implement it by using asynchronous I/O (without being bound to a thread for the majority of the operation's execution), run it on a specific thread (such as the UI thread), or use any number of potential contexts. A TAP method may even have nothing to execute, and may just return a [Task](#) that represents the occurrence of a condition elsewhere in the system (for example, a task that represents data arriving at a queued data structure). The caller of the TAP method may block waiting for the TAP method to complete by synchronously waiting on the resulting task, or may run additional (continuation) code when the asynchronous operation completes. The creator of the continuation code has control over where that code executes. You may create the continuation code either explicitly, through methods on the [Task](#) class (for example, [ContinueWith](#)) or implicitly, by using language support built on top of continuations (for example, **await** in C#, **Await** in Visual Basic, **AwaitValue** in F#).

Task Status

The [Task](#) class provides a life cycle for asynchronous operations, and that cycle is represented by the [TaskStatus](#) enumeration. To support corner cases of types that derive from [Task](#) and [Task\(Of TResult\)](#), and to support the separation of construction from scheduling, the [Task](#) class exposes a [Start](#) method. Tasks that are created by the public [Task](#) constructors are referred to as *cold tasks*, because they begin their life cycle in the non-scheduled [Created](#) state and are scheduled only when [Start](#) is called on these instances. All other tasks begin their life cycle in a hot state, which means that the asynchronous operations they represent have already been initiated and their task status is an enumeration value other than [TaskStatus.Created](#). All tasks that are returned from TAP methods must be activated. If a TAP method internally uses a task's constructor to instantiate the task to be returned, the TAP method must call [Start](#) on the [Task](#) object before returning it. Consumers of a TAP method may safely assume that the returned task is active and should not try to call [Start](#) on any [Task](#) that is returned from a TAP method. Calling [Start](#) on an active task results in an [InvalidOperationException](#) exception.

Cancellation (Optional)

In TAP, cancellation is optional for both asynchronous method implementers and asynchronous method consumers. If an operation allows cancellation, it exposes an overload of the asynchronous method that accepts a cancellation token ([CancellationToken](#) instance). By convention, the parameter is named *cancellationToken*.

VB

```
Public Function ReadAsync(buffer() As Byte, offset As Integer,
                          count As Integer,
                          cancellationToken As CancellationToken) _
    As Task
```

The asynchronous operation monitors this token for cancellation requests. If it receives a cancellation request, it may choose to honor that request and cancel the operation. If the cancellation request results in work being ended prematurely, the TAP method returns a task that ends in the [Canceled](#) state; there is no available result and no exception is thrown. The [Canceled](#) state is considered to be a final (completed) state for a task, along with the [Faulted](#) and [RanToCompletion](#) states. Therefore, if a task is in the [Canceled](#) state, its [IsCompleted](#) property returns **true**. When a task completes in the [Canceled](#) state, any continuations registered with the task are scheduled or executed, unless a continuation option such as [NotOnCanceled](#) was specified to opt out of continuation. Any code that is asynchronously waiting for a canceled task through use of language features continues to run but receives an [OperationCanceledException](#) or an exception derived from it. Code that is blocked synchronously waiting on the task through methods such as [Wait](#) and [WaitAll](#) also continue to run with an exception.

If a cancellation token has requested cancellation before the TAP method that accepts that token is called, the TAP method should return a [Canceled](#) task. However, if cancellation is requested while the asynchronous operation is running, the asynchronous operation need not accept the cancellation request. The returned task should end in the [Canceled](#) state only if the operation ends as a result of the cancellation request. If cancellation is requested but a result or an exception is still produced, the task should end in the [RanToCompletion](#) or [Faulted](#) state. For asynchronous methods used by a developer who wants cancellation first and foremost, you don't have to provide an overload that doesn't accept a cancellation token. For methods that cannot be canceled, do not provide overloads that accept a cancellation token; this helps indicate to the caller whether the target method is actually cancelable. Consumer code that does not desire cancellation may call a method that accepts a [CancellationToken](#) and provide [None](#) as the argument value. [None](#) is functionally equivalent to the default [CancellationToken](#).

Progress Reporting (Optional)

Some asynchronous operations benefit from providing progress notifications; these are typically used to update a user interface with information about the progress of the asynchronous operation. In TAP, progress is handled through an [IProgress\(Of T\)](#) interface, which is passed to the asynchronous method as a parameter that is usually named *progress*. Providing the progress interface when the asynchronous method is called helps eliminate race conditions that result from incorrect usage (that is, when event handlers that are incorrectly registered after the operation starts may miss updates). More importantly, the progress interface supports varying implementations of progress, as determined by the consuming code. For example, the consuming code may only care about the latest progress update, or may want to buffer all updates, or may want to invoke an action for each update, or may want to control whether the invocation is marshaled to a particular thread. All these options may be achieved by using a different implementation of the interface, customized to the particular consumer's needs. As with cancellation, TAP implementations should provide an [IProgress\(Of T\)](#) parameter only if the API supports progress notifications. For example, if the [ReadAsync](#) method discussed earlier in this article is able to report intermediate progress in the form of the number of bytes read thus far, the progress callback could be an [IProgress\(Of T\)](#) interface:

VB

```
Public Function ReadAsync(buffer() As Byte, offset As Integer,
                          count As Integer,
                          progress As IProgress(Of Long)) As Task
```

If a [FindFilesAsync](#) method returns a list of all files that meet a particular search pattern, the progress callback could provide an estimate of the percentage of work completed as well as the current set of partial results. It could do this either with a tuple:

VB

```
Public Function FindFilesAsync(pattern As String,
                               progress As IProgress(Of Tuple(Of Double,
ReadOnlyCollection(Of List(Of FileInfo)))) _
                               As Task(Of ReadOnlyCollection(Of FileInfo))
```

or with a data type that is specific to the API:

VB

```
Public Function FindFilesAsync(pattern As String,
                               progress As IProgress(Of FindFilesProgressInfo)) _
                               As Task(Of ReadOnlyCollection(Of FileInfo))
```

In the latter case, the special data type is usually suffixed with [ProgressInfo](#).

If TAP implementations provide overloads that accept a *progress* parameter, they must allow the argument to be **null**, in which case no progress will be reported. TAP implementations should report the progress to the [Progress\(Of T\)](#) object synchronously, which enables the asynchronous method to quickly provide progress, and allow the consumer of the progress to determine how and where best to handle the information. For example, the progress instance could choose to marshal callbacks and raise events on a captured synchronization context.

IProgress<T> Implementations

The .NET Framework 4.5 provides a single [IProgress\(Of T\)](#) implementation: [Progress\(Of T\)](#). The [Progress\(Of T\)](#) class is declared as follows:

VB

```
Public Class Progress(Of T) : Inherits IProgress(Of T)
    Public Sub New()
    Public Sub New(handler As Action(Of T))
    Protected Overridable Sub OnReport(value As T)
    Public Event ProgressChanged As EventHandler(Of T)
End Class
```

An instance of [Progress\(Of T\)](#) exposes a [ProgressChanged](#) event, which is raised every time the asynchronous operation reports a progress update. The [ProgressChanged](#) event is raised on the [SynchronizationContext](#) object that was captured when the [Progress\(Of T\)](#) instance was instantiated. If no synchronization context was available, a default context that targets the thread pool is used. Handlers may be registered with this event. A single handler may also be provided to the [Progress\(Of T\)](#) constructor for convenience, and behaves just like an event handler for the [ProgressChanged](#) event. Progress updates are raised asynchronously to avoid delaying the asynchronous operation while event handlers are executing. Another [IProgress\(Of T\)](#) implementation could choose to apply different semantics.

Choosing the Overloads to Provide

If a TAP implementation uses both the optional [CancellationToken](#) and optional [IProgress\(Of T\)](#) parameters, it could potentially require up to four overloads:

VB

```
Public MethodNameAsync(...) As Task
Public MethodNameAsync(..., cancellationToken As Cancellation_token) As Task
Public MethodNameAsync(..., progress As IProgress(Of T)) As Task
Public MethodNameAsync(..., cancellationToken As Cancellation_token,
    progress As IProgress(Of T)) As Task
```

However, many TAP implementations provide neither cancellation or progress capabilities, so they require a single method:

VB

```
Public MethodNameAsync(...) As Task
```

If a TAP implementation supports either cancellation or progress but not both, it may provide two overloads:

VB

```
Public MethodNameAsync(...) As Task
Public MethodNameAsync(..., cancellationToken As CancellationToken) As Task

' ... or ...

Public MethodNameAsync(...) As Task
Public MethodNameAsync(..., progress As IProgress(Of T)) As Task
```

If a TAP implementation supports both cancellation and progress, it may expose all four overloads. However, it may provide only the following two:

VB

```
Public MethodNameAsync(...) As Task
Public MethodNameAsync(..., cancellationToken As CancellationToken,
    progress As IProgress(Of T)) As Task
```

To compensate for the two missing intermediate combinations, developers may pass [None](#) or a default [CancellationToken](#) for the *cancellationToken* parameter and **null** for the *progress* parameter.

If you expect every usage of the TAP method to support cancellation or progress, you may omit the overloads that don't accept the relevant parameter.

If you decide to expose multiple overloads to make cancellation or progress optional, the overloads that don't support cancellation or progress should behave as if they passed [None](#) for cancellation or **null** for progress to the overload that does support these.

Related Topics

Title	Description
Asynchronous Programming Patterns	Introduces the three patterns for performing asynchronous operations: the Task-based Asynchronous Pattern (TAP), the Asynchronous Programming Model (APM), and the Event-based Asynchronous Pattern (EAP).
Implementing the Task-based Asynchronous Pattern	Describes how to implement the Task-based Asynchronous Pattern (TAP) in three ways: by using the C# and Visual Basic compilers in Visual Studio, manually, or through a combination of the compiler and manual methods.
Consuming the Task-based Asynchronous Pattern	Describes how you can use tasks and callbacks to achieve waiting without blocking.
Interop with Other Asynchronous Patterns and	Describes how to use the Task-based Asynchronous Pattern (TAP) to implement the Asynchronous Programming Model (APM) and Event-based Asynchronous Pattern

Types	(EAP).
-------	--------