**Asynchronous programming patterns**

# Asynchronous Programming Model (APM)

An asynchronous operation that uses the [IAsyncResult](https://docs.microsoft.com/en-us/dotnet/api/system.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](https://docs.microsoft.com/en-us/dotnet/api/system.io.filestream) class provides the [BeginRead](https://docs.microsoft.com/en-us/dotnet/api/system.io.filestream.beginread) and [EndRead](https://docs.microsoft.com/en-us/dotnet/api/system.io.filestream.endread) methods to asynchronously read bytes from a file. These methods implement the asynchronous version of the [Read](https://docs.microsoft.com/en-us/dotnet/api/system.io.filestream.read) method.

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](https://docs.microsoft.com/en-us/dotnet/api/system.iasyncresult) interface.

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](https://docs.microsoft.com/en-us/dotnet/api/system.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.

## **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.

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](https://docs.microsoft.com/en-us/dotnet/api/system.iasyncresult) parameter. Callers must pass the instance returned by the corresponding call to **Begin***OperationName*.

If the asynchronous operation represented by the [IAsyncResult](https://docs.microsoft.com/en-us/dotnet/api/system.iasyncresult) object has not completed when **End***OperationName* is called, **End***OperationName* blocks the calling thread until the asynchronous operation is complete.

**/\***

**The following example demonstrates using asynchronous methods to**

**get Domain Name System information for the specified host computer.**

**\*/**

**using System;**

**using System.Net;**

**using System.Net.Sockets;**

**namespace Examples.AdvancedProgramming.AsynchronousOperations**

**{**

**public class BlockUntilOperationCompletes**

**{**

**public static void Main(string[] args)**

**{**

**// Make sure the caller supplied a host name.**

**if (args.Length == 0 || args[0].Length == 0)**

**{**

**// Print a message and exit.**

**Console.WriteLine("You must specify the name of a host computer.");**

**return;**

**}**

**// Start the asynchronous request for DNS information.**

**// This example does not use a delegate or user-supplied object**

**// so the last two arguments are null.**

**IAsyncResult result = Dns.BeginGetHostEntry(args[0], null, null);**

**Console.WriteLine("Processing your request for information...");**

**// Do any additional work that can be done here.**

**try**

**{**

**// EndGetHostByName blocks until the process completes.**

**IPHostEntry host = Dns.EndGetHostEntry(result);**

**string[] aliases = host.Aliases;**

**IPAddress[] addresses = host.AddressList;**

**if (aliases.Length > 0)**

**{**

**Console.WriteLine("Aliases");**

**for (int i = 0; i < aliases.Length; i++)**

**{**

**Console.WriteLine("{0}", aliases[i]);**

**}**

**}**

**if (addresses.Length > 0)**

**{**

**Console.WriteLine("Addresses");**

**for (int i = 0; i < addresses.Length; i++)**

**{**

**Console.WriteLine("{0}",addresses[i].ToString());**

**}**

**}**

**}**

**catch (SocketException e)**

**{**

**Console.WriteLine("An exception occurred while processing the request: {0}", e.Message);**

**}**

**}**

**}**

**}**

# Event-based Asynchronous Pattern Overview

The Event-based Asynchronous Pattern makes available the advantages of multithreaded applications while hiding many of the complex issues inherent in multithreaded design. Using a class that supports this pattern can allow you to:

* Perform time-consuming tasks, such as downloads and database operations, "in the background," without interrupting your application.
* Execute multiple operations simultaneously, receiving notifications when each completes.
* Wait for resources to become available without stopping ("hanging") your application.
* Communicate with pending asynchronous operations using the familiar events-and-delegates model. For more information on using event handlers and delegates, see [Events](https://docs.microsoft.com/en-us/dotnet/standard/events/index).

A class that supports the Event-based Asynchronous Pattern will have one or more methods named MethodName**Async**. These methods may mirror synchronous versions, which perform the same operation on the current thread. The class may also have a MethodName**Completed** event and it may have a MethodName**AsyncCancel** (or simply **CancelAsync**) method.

If you want your application to keep running while the image is loading, you can call the [LoadAsync](https://docs.microsoft.com/en-us/dotnet/api/system.windows.forms.picturebox.loadasync) method and handle the [LoadCompleted](https://docs.microsoft.com/en-us/dotnet/api/system.windows.forms.picturebox.loadcompleted) event, just as you would handle any other event. When you call the [LoadAsync](https://docs.microsoft.com/en-us/dotnet/api/system.windows.forms.picturebox.loadasync) method, your application will continue to run while the download proceeds on a separate thread ("in the background"). Your event handler will be called when the image-loading operation is complete, and your event handler can examine the [AsyncCompletedEventArgs](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.asynccompletedeventargs) parameter to determine if the download completed successfully.

**public class AsyncExample**

**{**

**// Synchronous methods.**

**public int Method1(string param);**

**public void Method2(double param);**

**// Asynchronous methods.**

**public void Method1Async(string param);**

**public void Method1Async(string param, object userState);**

**public event Method1CompletedEventHandler Method1Completed;**

**public void Method2Async(double param);**

**public void Method2Async(double param, object userState);**

**public event Method2CompletedEventHandler Method2Completed;**

**public void CancelAsync(object userState);**

**public bool IsBusy { get; }**

**// Class implementation not shown.**

**}**

# Task-based Asynchronous Pattern (TAP)

## **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 Asyncsuffix and also requires one or more events, event handler delegate types, and EventArg-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.

A TAP method returns either a [System.Threading.Tasks.Task](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task) or a [System.Threading.Tasks.Task<TResult>](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1), 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<TResult>](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1), and should use a tuple or a custom data structure to accommodate multiple values.

## **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.

* 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.

## **Exceptions**

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.

## **Task Status**

The [Task](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task) class provides a life cycle for asynchronous operations, and that cycle is represented by the [TaskStatus](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskstatus) enumeration. To support corner cases of types that derive from [Task](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task) and [Task<TResult>](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1), and to support the separation of construction from scheduling, the [Task](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task) class exposes a [Start](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.start) method. Tasks that are created by the public [Task](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task) constructors are referred to as cold tasks, because they begin their life cycle in the non-scheduled [Created](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskstatus#System_Threading_Tasks_TaskStatus_Created) state and are scheduled only when [Start](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.start) is called on these instances.

namespace System.Threading.Tasks

{

//

// Summary:

// Represents the current stage in the lifecycle of a System.Threading.Tasks.Task.

public enum TaskStatus

{

//

// Summary:

// The task has been initialized but has not yet been scheduled.

Created = 0,

//

// Summary:

// The task is waiting to be activated and scheduled internally by the .NET Framework

// infrastructure.

WaitingForActivation = 1,

//

// Summary:

// The task has been scheduled for execution but has not yet begun executing.

WaitingToRun = 2,

//

// Summary:

// The task is running but has not yet completed.

Running = 3,

//

// Summary:

// The task has finished executing and is implicitly waiting for attached child

// tasks to complete.

WaitingForChildrenToComplete = 4,

//

// Summary:

// The task completed execution successfully.

RanToCompletion = 5,

//

// Summary:

// The task acknowledged cancellation by throwing an OperationCanceledException

// with its own CancellationToken while the token was in signaled state, or the

// task's CancellationToken was already signaled before the task started executing.

// For more information, see Task Cancellation.

Canceled = 6,

//

// Summary:

// The task completed due to an unhandled exception.

Faulted = 7

}

}

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](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskstatus" \l "System_Threading_Tasks_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**](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.start)**on the**[**Task**](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.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](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.start) on any [Task](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task) that is returned from a TAP method. Calling [Start](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.start) on an active task results in an [InvalidOperationException](https://docs.microsoft.com/en-us/dotnet/api/system.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](https://docs.microsoft.com/en-us/dotnet/api/system.threading.cancellationtoken) instance).

public Task ReadAsync(byte [] buffer, int offset, int count, CancellationToken cancellationToken)

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](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskstatus#System_Threading_Tasks_TaskStatus_Canceled) state; there is no available result and no exception is thrown.

If a cancellation token has requested cancellation before the TAP method that accepts that token is called, the TAP method should return a [Canceled](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskstatus#System_Threading_Tasks_TaskStatus_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](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskstatus#System_Threading_Tasks_TaskStatus_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](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskstatus" \l "System_Threading_Tasks_TaskStatus_RanToCompletion) or [Faulted](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskstatus#System_Threading_Tasks_TaskStatus_Faulted) state.

## **Progress Reporting (Optional)**

Update a user interface with information about the progress of the asynchronous operation.

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<T>](https://docs.microsoft.com/en-us/dotnet/api/system.iprogress-1) interface:

public Task ReadAsync(byte[] buffer, int offset, int count, IProgress<long> progress)

## **Hybrid approach - Task**

public Task<int> MethodAsync(string input)

{

if (input == null) throw new ArgumentNullException("input");

return MethodAsyncInternal(input);

}

private async Task<int> MethodAsyncInternal(string input)

{

// code that uses await goes here

return value;

}

## **Suspending Execution with Await**

[Await](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/await) keyword in C# is used to asynchronously await [Task](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task) and [Task<TResult>](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1) objects. When you're awaiting a [Task](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task), the await expression is of type void. When you're awaiting a [Task<TResult>](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1), the await expression is of type TResult. An await expression must occur inside the body of an asynchronous method.

The await functionality installs a callback on the task by using a continuation. This callback resumes the asynchronous method at the point of suspension. When the asynchronous method is resumed, if the awaited operation completed successfully and was a [Task<TResult>](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1), its TResult is returned. If the [Task](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task) or [Task<TResult>](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1) that was awaited ended in the [Canceled](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskstatus#System_Threading_Tasks_TaskStatus_Canceled) state, an [OperationCanceledException](https://docs.microsoft.com/en-us/dotnet/api/system.operationcanceledexception) exception is thrown. If the [Task](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task) or [Task<TResult>](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1) that was awaited ended in the [Faulted](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskstatus#System_Threading_Tasks_TaskStatus_Faulted) state, the exception that caused it to fault is thrown.

## **Configuring Suspension and Resumption with ConfigureAwait**

You can also use the [Task.ConfigureAwait](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.configureawait) method for better control over suspension and resumption in an asynchronous method. As mentioned previously, by default, the current context is captured at the time an asynchronous method is suspended, and that captured context is used to invoke the asynchronous method’s continuation upon resumption. In many cases, this is the exact behavior you want. In other cases, you may not care about the continuation context, and you can achieve better performance by avoiding such posts back to the original context. To enable this, use the [Task.ConfigureAwait](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.configureawait) method to inform the await operation not to capture and resume on the context, but to continue execution wherever the asynchronous operation that was being awaited completed.

await someTask.ConfigureAwait(continueOnCapturedContext:false);

## **Canceling an Asynchronous Operation**

A cancellation token is created through a cancellation token source ([CancellationTokenSource](https://docs.microsoft.com/en-us/dotnet/api/system.threading.cancellationtokensource) object). The source’s [Token](https://docs.microsoft.com/en-us/dotnet/api/system.threading.cancellationtokensource.token) property returns the cancellation token that will be signaled when the source’s [Cancel](https://docs.microsoft.com/en-us/dotnet/api/system.threading.cancellationtokensource.cancel) method is called.

var cts = new CancellationTokenSource();

string result = await DownloadStringAsync(url, cts.Token);

… // at some point later, potentially on another thread

cts.Cancel();

## **Monitoring Progress**

Some asynchronous methods expose progress through a progress interface passed into the asynchronous method. For example, consider a function which asynchronously downloads a string of text, and along the way raises progress updates that include the percentage of the download that has completed thus far.

private async void btnDownload\_Click(object sender, RoutedEventArgs e)

{

btnDownload.IsEnabled = false;

try

{

txtResult.Text = await DownloadStringAsync(txtUrl.Text,

new Progress<int>(p => pbDownloadProgress.Value = p));

}

finally { btnDownload.IsEnabled = true; }

}

## **Task.Run**

public async void button1\_Click(object sender, EventArgs e)

{

textBox1.Text = await Task.Run(() =>

{

// … do compute-bound work here

return answer;

});

}

## **Task.FromResult**

Use the [FromResult](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.fromresult) method in scenarios where data may already be available and just needs to be returned from a task-returning method lifted into a [Task<TResult>](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1):

public Task<int> GetValueAsync(string key)

{

int cachedValue;

return TryGetCachedValue(out cachedValue) ?

Task.FromResult(cachedValue) :

GetValueAsyncInternal();

}

private async Task<int> GetValueAsyncInternal(string key)

{

…

}