[Server & Tools Blogs](https://blogs.technet.microsoft.com/serverandtools) > [Developer Tools Blogs](https://blogs.msdn.microsoft.com/developer-tools) > Parallel Programming with .NET

* [Sign in](https://blogs.msdn.microsoft.com/wp-login.php?aadsso_action=login)

[Skip to content](https://blogs.msdn.microsoft.com/pfxteam/2012/04/12/asyncawait-faq/#content)

* [Executive Bloggers](https://blogs.msdn.microsoft.com/pfxteam/2012/04/12/asyncawait-faq/)
* [Visual Studio Products](https://blogs.msdn.microsoft.com/pfxteam/2012/04/12/asyncawait-faq/)
* [DevOps](https://blogs.msdn.microsoft.com/pfxteam/2012/04/12/asyncawait-faq/)
* [Languages](https://blogs.msdn.microsoft.com/pfxteam/2012/04/12/asyncawait-faq/)
* [Features](https://blogs.msdn.microsoft.com/pfxteam/2012/04/12/asyncawait-faq/)
* [.NET](https://blogs.msdn.microsoft.com/pfxteam/2012/04/12/asyncawait-faq/)
* [Platform Development](https://blogs.msdn.microsoft.com/pfxteam/2012/04/12/asyncawait-faq/)
* [Data Development](https://blogs.msdn.microsoft.com/pfxteam/2012/04/12/asyncawait-faq/)

[**Parallel Programming with .NET**](https://blogs.msdn.microsoft.com/pfxteam)

All about Async/Await, System.Threading.Tasks, System.Collections.Concurrent, System.Linq, and more…

[Parallel Programming with .NET](https://blogs.msdn.microsoft.com/pfxteam/)

**asdAsync/Await FAQыыыыы**

★★★★★

★★★★

★★★

★★

★

April 12, 2012 by [Stephen Toub - MSFT](https://social.msdn.microsoft.com/profile/Stephen+Toub+-+MSFT) // [22 Comments](https://blogs.msdn.microsoft.com/pfxteam/2012/04/12/asyncawait-faq/#comments)

* [0](https://blogs.msdn.microsoft.com/pfxteam/2012/04/12/asyncawait-faq/)
* [0](https://blogs.msdn.microsoft.com/pfxteam/2012/04/12/asyncawait-faq/)

From time to time, I receive questions from developers which highlight either a need for more information about the new “async” and “await” keywords in C# and Visual Basic. I’ve been cataloguing these questions, and I thought I’d take this opportunity to share my answers to them.

**Conceptual Overview**

Where can I get a good overview of the async/await keywords?

Generally, you can find lots of resources (links to articles, videos, blogs, etc.) on the Visual Studio Async page at <http://msdn.com/async>. To call out just a few specific resources, the October 2011 issue of MSDN Magazine included a trio of articles that provided a good introduction to the topic. If you read them all, I recommend you read them in the following order:

1. [Asynchronous Programming: Easier Asynchronous Programming with the New Visual Studio Async CTP](http://msdn.microsoft.com/en-us/magazine/hh456401.aspx)
2. [Asynchronous Programming: Pause and Play with Await](http://msdn.microsoft.com/en-us/magazine/hh456403.aspx)
3. [Asynchronous Programming: Understanding the Costs of Async and Await](http://msdn.microsoft.com/en-us/magazine/hh456402.aspx)

The .NET team blog also includes a good overview of asynchrony in .NET 4.5: [Async in 4.5: Worth the Await](http://blogs.msdn.com/b/dotnet/archive/2012/04/03/async-in-4-5-worth-the-await.aspx).

Why do I need the compiler to help me with asynchronous programming?

Anders Hejlsberg’s [Future directions for C# and Visual Basic](https://channel9.msdn.com/events/BUILD/BUILD2011/TOOL-816T) talk at //BUILD/ provides a great tour through why a compiler is really beneficial here. In short, the compiler takes on the responsibility of doing the kind of complicated transformations you’d otherwise be doing by hand as you manually invert your control flow using callbacks and continuation-passing style. You get to write your code using the language’s control flow constructs, just as you would if you were writing synchronous code, and the compiler under the covers applies the transformations necessary to use callbacks in order to avoid blocking threads.

To achieve the benefits of asynchrony, can’t I just wrap my synchronous methods in calls to Task.Run?

It depends on your goals for why you want to invoke the methods asynchronously. If your goal is simply to offload the work you’re doing to another thread, so as to, for example, maintain the responsiveness of your UI thread, then sure. If your goal is to help with scalability, then no, just wrapping a synchronous call in a Task.Run won’t help. For more information, see [Should I expose asynchronous wrappers for synchronous methods?](http://blogs.msdn.com/b/pfxteam/archive/2012/03/24/10287244.aspx) And if from your UI thread you want to offload work to a worker thread, and you use Task.Run to do so, you often typically want to do some work back on the UI thread once that background work is done, and these language features make that kind of coordination easy and seamless.

**The “Async” Keyword**

What does the “async” keyword do when applied to a method?

When you mark a method with the “async” keyword, you’re really telling the compiler two things:

1. You’re telling the compiler that you want to be able to use the “await” keyword inside the method (you can use the await keyword if and only if the method or lambda it’s in is marked as async). In doing so, you’re telling the compiler to compile the method using a state machine, such that the method will be able to suspend and then resume asynchronously at await points.
2. You’re telling the compiler to “lift” the result of the method or any exceptions that may occur into the return type. For a method that returns Task or Task<TResult>, this means that any returned value or exception that goes unhandled within the method is stored into the result task. For a method that returns void, this means that any exceptions are propagated to the caller’s context via whatever “SynchronizationContext” was current at the time of the method’s initial invocation.

Does using the “async” keyword on a method force all invocations of that method to be asynchronous?

No. When you invoke a method marked as “async”, it begins running synchronously on the curren thread. So, if you have a synchronous method that returns void and all you do to change it is mark it as “async”, invocations of that method will still run synchronously. This is true regardless of whether you leave the return type as “void” or change it to “Task”. Similarly, if you have a synchronous method that returns some TResult, and all you do is mark it as “async” and change the return type to be “Task<TResult>”, invocations of that method will still run synchronously.

Marking a method as “async” does not affect whether the method runs to completion synchronously or asynchronously. Rather, it enables the method to be split into multiple pieces, some of which may run asynchronously, such that the method may complete asynchronously. The boundaries of these pieces can occur only where you explicitly code one using the “await” keyword, so if “await” isn’t used at all in a method’s code, there will only be one piece, and since that piece will start running synchronously, it (and the whole method with it) will complete synchronously.

Does the “async” keyword cause the invocation of a method to queue to the ThreadPool? To create a new thread? To launch a rocket ship to Mars?

No. No. And no. See the previous questions. The “async” keyword indicates to the compiler that “await” may be used inside of the method, such that the method may suspend at an await point and have its execution resumed asynchronously when the awaited instance completes. This is why the compiler issues a warning if there are no “awaits” inside of a method marked as “async”.

Can I mark any method as “async”?

No. Only methods that return void, Task, or Task<TResult> can be marked as async. Further, not all such methods can be marked as “async”. For example, you can’t use “async”:

* On your application’s entry point method, e.g. Main. When you await an instance that’s not yet completed, execution returns to the caller of the method. In the case of Main, this would return out of Main, effectively ending the program.
* On a method attributed with:
  + [MethodImpl(MethodImplOptions.Synchronized)]. For a discussion of why this is disallowed, see [What’s New for Parallelism in .NET 4.5 Beta](http://blogs.msdn.com/b/pfxteam/archive/2012/02/29/10274035.aspx); attributing a method as Synchronized is akin to wrapping the entire body of the method with lock/SyncLock.
  + [SecurityCritical] and [SecuritySafeCritical]. When you compile an async method, the implementation / body of the method actually ends up in a compiler-generated MoveNext method, but the attributes for it remain on the signature you defined. That means that attributes like [SecuritySafeCritical] (which is meant to have a direct impact on what you’re able to do in the body of the method) would not work correctly, and thus they’re prohibited, at least for now.
* On a method with ref or out parameters.  The caller would expect those values to be set when the synchronous invocation of the method completes, but the implementation might not set them until its asynchronous completion much later.
* On a lambda used as an expression tree.  Async lambda expressions cannot be converted to expression trees.

Are there any conventions I should use when writing methods marked as “async”?

Yes. The Task-based Asynchronous Pattern (TAP) is entirely focused on how asynchronous methods that return Task or Task<TResult> should be exposed from libraries. This includes, but is not limited to, methods implemented using the “async” and “await” keywords. For an in-depth tour through the TAP, see the [Task-based Asynchronous Pattern](http://www.microsoft.com/download/en/details.aspx?id=19957) document.

Do I need to “Start” Tasks created by methods marked as “async”?

No.  Tasks returned from TAP methods are “hot”, meaning the tasks represent operations that are already in-progress.  Not only do you not need to call “.Start()” on such tasks, but doing so will fail if you try.  For more details, see [FAQ on Task.Start](http://blogs.msdn.com/b/pfxteam/archive/2012/01/14/10256832.aspx).

Do I need to “Dispose” Tasks created by methods marked as “async”?

No. In general, you don’t need to Dispose of any tasks.  See [Do I need to dispose of Tasks?](http://blogs.msdn.com/b/pfxteam/archive/2012/03/25/10287435.aspx).

How does “async” relate to the current SynchronizationContext?

For methods marked as “async” that return Task or Task<TResult>, there is no method-level interaction with the SynchronizationContext. However, for methods marked as “async” that return void, there is a potential interaction.

When an “async void” method is invoked, the prolog for the method’s invocation (as handled by the AsyncVoidMethodBuilder that is created by the compiler to represent the method’s lifetime) will capture the current SynchronizationContext (“capture” here means it accesses it and stores it). If there is a non-null SynchronizationContext, two things will be affected:

* The beginning of the method’s invocation will result in a call to the captured context’s OperationStarted method, and the completion of the method’s execution (whether synchronous or asynchronous) will result in a call to that captured context’s OperationCompleted method. This gives the context a chance to reference count outstanding asynchronous operations; if instead the method had returned a Task or Task<TResult>, the caller could have done the same tracking via that returned task.
* If the method completes due to an unhandled exception, the throwing of that exception will be Post’d to the captured SynchronizationContext. This gives the context a chance to deal with the failure. This is in contrast to a Task or Task<TResult>-returning async method, where the exception can be marshaled to the caller through the returned task.

If there isn’t a SynchronizationContext when the “async void” method is called, no context is captured, and then as there are no OperationStarted / OperationCompleted methods to call, none are invoked. In such a case, if an exception goes unhandled, the exception is propagated on the ThreadPool, which with default behavior will cause the process to be terminated.

**The “Await” Keyword**

What does the “await” keyword do?

The “await” keyword tells the compiler to insert a possible suspension/resumption point into a method marked as “async”.

Logically this means that when you write “await someObject;” the compiler will generate code that checks whether the operation represented by someObject has already completed. If it has, execution continues synchronously over the await point. If it hasn’t, the generated code will hook up a continuation delegate to the awaited object such that when the represented operation completes, that continuation delegate will be invoked. This continuation delegate will re-enter the method, picking up at this await location where the previous invocation left off. At this point, regardless of whether the awaited object had already completed by the time it was awaited, any result from the object will be extracted, or if the operation failed, any exception that occurred will be propagated.

In code, this means that when you write:

await someObject;

the compiler translates that into something like the following (this code is an approximation of what the compiler actually generates):

private class FooAsyncStateMachine : IAsyncStateMachine   
{   
    // Member fields for preserving “locals” and other necessary state   
    int $state;   
    TaskAwaiter $awaiter;   
    …   
    public void MoveNext()   
    {   
        // Jump table to get back to the right statement upon resumption   
        switch (this.$state)   
        {   
            …   
            case 2: goto Label2;   
            …   
        }   
        …   
        // Expansion of “await someObject;”   
        this.$awaiter = someObject.GetAwaiter();   
        if (!this.$awaiter.IsCompleted)   
        {   
            this.$state = 2;   
            this.$awaiter.OnCompleted(MoveNext);   
            return;   
            Label2:   
        }   
        this.$awaiter.GetResult();   
        …   
    }   
}

What are awaitables? What are awaiters?

While Task and Task<TResult> are two types very commonly awaited, they’re not the only ones that may be awaited.

An “awaitable” is any type that exposes a GetAwaiter method which returns a valid “awaiter”. This GetAwaiter method may be an instance method (as it is in the case of Task and Task<TResult>), or it may be an extension method.

An “awaiter” is any type returned from an awaitable’s GetAwaiter method and that conforms to a particular pattern. The awaiter must implement the System.Runtime.CompilerServices.INotifyCompletion interface, and optionally may implement the System.Runtime.CompilerServices.ICriticalNotifyCompletion interface. In addition to providing an implementation of the OnCompleted method that comes from INotifyCompletion (and optionally the UnsafeOnCompleted method that comes from ICriticalNotifyCompletion), an awaiter must also provide an IsCompleted Boolean property, as well as a parameterless GetResult method. GetResult returns void if the awaitable represents a void-returning operation, or it returns a TResult if the awaitable represents a TResult-returning operation.

Any type that follows the awaitable pattern may be awaited. For a discussion of several approaches to implementing custom awaitables, see [await anything;](http://blogs.msdn.com/b/pfxteam/archive/2011/01/13/10115642.aspx). You can also implement awaitables customized for very specific situations: for some examples, see [Advanced APM Consumption in Async Methods](http://blogs.msdn.com/b/pfxteam/archive/2012/01/23/10259822.aspx) and [Awaiting Socket Operations](http://blogs.msdn.com/b/pfxteam/archive/2011/12/15/10248293.aspx).

Where can’t I use “await”?

You can’t use await:

* Inside of a method or lambda not marked with “async”. The “async” keyword is what tells the compiler that it’s ok to use “await” inside of a method. (For a discussion of why the “async” keyword is required, see [Asynchrony in C# 5 Part Six: Whither async?](http://blogs.msdn.com/b/ericlippert/archive/2010/11/11/whither-async.aspx).)
* Inside of a property getter or setter. Properties are meant to return to the caller quickly, and thus are not expected to need asynchrony, which is geared for potentially long-running operations. If you must use asynchrony in your properties, you can do so by implementing an async method which is then used from your property.
* Inside of a lock/SyncLock block. For a discussion of why this is disallowed, as well as for a look at SemaphoreSlim.WaitAsync (which can be used with await), see [What’s New for Parallelism in .NET 4.5 Beta](http://blogs.msdn.com/b/pfxteam/archive/2012/02/29/10274035.aspx). You can also read about building a variety of custom async synchronization primitives at:
  + [Building Async Coordination Primitives, Part 1: AsyncManualResetEvent](http://blogs.msdn.com/b/pfxteam/archive/2012/02/11/10266920.aspx)
  + [Building Async Coordination Primitives, Part 2: AsyncAutoResetEvent](http://blogs.msdn.com/b/pfxteam/archive/2012/02/11/10266923.aspx)
  + [Building Async Coordination Primitives, Part 3: AsyncCountdownEvent](http://blogs.msdn.com/b/pfxteam/archive/2012/02/11/10266930.aspx)
  + [Building Async Coordination Primitives, Part 4: AsyncBarrier](http://blogs.msdn.com/b/pfxteam/archive/2012/02/11/10266932.aspx)
  + [Building Async Coordination Primitives, Part 5: AsyncSemaphore](http://blogs.msdn.com/b/pfxteam/archive/2012/02/12/10266983.aspx)
  + [Building Async Coordination Primitives, Part 6: AsyncLock](http://blogs.msdn.com/b/pfxteam/archive/2012/02/12/10266988.aspx)
  + [Building Async Coordination Primitives, Part 7: AsyncReaderWriterLock](http://blogs.msdn.com/b/pfxteam/archive/2012/02/12/10267069.aspx)
* Inside of an unsafe region. Note that you can use the “unsafe” keyword inside of a method marked as “async”, you just can’t use “await” inside of an unsafe region. This restriction has to do with how the compilers preserve state across an “await” point, and difficulties inherent to preserving unmanaged pointers.
* Inside of a catch or finally block. You can use “await” inside of a try block, regardless of whether it has associated catch or finally blocks, but you can’t use it inside of the catch or finally blocks. Doing so would disrupt the semantics of CLR exception handling.
* With most aspects of LINQ query comprehension syntax. “await” may only be used in a query expression within the first collection expression of the initial “from” clause or within the collection expression of a “join” clause.

Is “await task;” the same thing as “task.Wait()”?

No.

“task.Wait()” is a synchronous, potentially blocking call: it will not return to the caller of Wait() until the task has entered a final state, meaning that it’s completed in the RanToCompletion, Faulted, or Canceled state. In contrast, “await task;” tells the compiler to insert a potential suspension/resumption point into a method marked as “async”, such that if the task has not yet completed when it’s awaited, the async method should return to its caller, and its execution should resume when and only when the awaited task completes. Using “task.Wait()” when “await task;” would have been more appropriate can lead to unresponsive applications and deadlocks; see [Await, and UI, and deadlocks! Oh my!](http://blogs.msdn.com/b/pfxteam/archive/2011/01/13/10115163.aspx).

There are some other potential pitfalls to be aware of when using “async” and “await”. For some examples, see:

* [Potential pitfalls to avoid when passing around async lambdas](http://blogs.msdn.com/b/pfxteam/archive/2012/02/08/10265476.aspx)
* [Keeping Async Methods Alive](http://blogs.msdn.com/b/pfxteam/archive/2011/10/02/10219048.aspx)
* [Don’t Forget To Complete Your Tasks](http://blogs.msdn.com/b/pfxteam/archive/2011/10/02/10218999.aspx)
* [Are deadlocks still possible with await?](http://blogs.msdn.com/b/pfxteam/archive/2012/04/12/10293249.aspx)

Is there a functional difference between “task.Result” and “task.GetAwaiter().GetResult()”?

Yes, but only if the task completes non-successfully.  If the task ends in the RanToCompletion state, these are completely equivalent statements.  If, however, the task ends in the Faulted or Canceled state, the former will propagate the one or more exceptions wrapped in AggregateException, while the latter will propagate the exception directly (and if there are more than one in the task, it’ll just propagate one of them).  For background on why this difference exists, see [Task Exception Handling in .NET 4.5](http://blogs.msdn.com/b/pfxteam/archive/2011/09/28/10217876.aspx).

How does “await” relate to the current SynchronizationContext?

This is entirely up to the type being awaited. For a given await, the compiler generates code that ends up calling an awaiter’s OnCompleted method, passing in the continuation delegate to be executed. The compiler-generated code knows nothing about SynchronizationContext, and simply relies on the awaited object’s OnCompleted method to invoke the provided callback when the awaited operation completes. It’s the OnCompleted method, then, that’s responsible for making sure that the delegate is invoked in the “right place,” where “right place” is left entirely up to the awaiter.

The default behavior for awaiting a task (as implemented by the TaskAwaiter and TaskAwaiter<TResult> types returned from Task’s and Task<TResult>’s GetAwaiter methods, respectively) is to capture the current SynchronizationContext before suspending, and then when the awaited task completes, if there had been a current SynchronizationContext that got captured, to Post the invocation of the continuation delegate back to that SynchronizationContext. So, for example, if you use “await task;” on the UI thread of your application, OnCompleted when invoked will see a non-null current SynchronizationContext, and when the task completes, it’ll use that UI’s SynchronizationContext to marshal the invocation of the continuation delegate back to the UI thread.

If there isn’t a current SynchronizationContext when you await a Task, then the system will check to see if there’s a current TaskScheduler, and if there is, the continuation will be scheduled to that when the task completes.

If there isn’t such a context or scheduler to force the continuation back to, or if you do “await task.ConfigureAwait(false)” instead of just “await task;”, then the continuation won’t be forced back to the original context and will be allowed to run wherever the system deems appropriate. This typically means either running the continuation synchronously wherever the awaited task completes or running the continuation on the ThreadPool.

Can I use “await” in console apps?

Sure. You can’t use “await” inside of your Main method, however, as entry points can’t be marked as async. Instead, you can use “await” in other methods in your console app, and then if you call those methods from Main, you can synchronously wait (rather than asynchronously wait) for them to complete, e.g.

public static void Main()   
{   
    FooAsync().Wait();   
}

private static async Task FooAsync()   
{   
    await Task.Delay(1000);   
    Console.WriteLine(“Done with first delay”);   
    await Task.Delay(1000);   
}

You could also use a custom SynchronizationContext or TaskScheduler to achieve similar capabilities. For more information, see:

* [Await, SynchronizationContext, and Console Apps: Part 1](http://blogs.msdn.com/b/pfxteam/archive/2012/01/20/10259049.aspx)
* [Await, SynchronizationContext, and Console Apps: Part 2](http://blogs.msdn.com/b/pfxteam/archive/2012/01/21/10259307.aspx)
* [Await, SynchronizationContext, and Console Apps: Part 3](http://blogs.msdn.com/b/pfxteam/archive/2012/02/02/10263555.aspx)

Can I use “await” with other asynchronous patterns, like the Asynchronous Programming Model (APM) pattern and the Event-based Async Pattern (EAP)?

Sure. You can either implement a custom awaitable for your asynchronous operation, or you can convert the existing asynchronous operation to something that’s already awaitable, like Task or Task<TResult>. Here are some examples:

* [Tasks and the APM Pattern](http://blogs.msdn.com/b/pfxteam/archive/2009/06/09/9716439.aspx)
* [Tasks and the Event-based Asynchronous Pattern](http://blogs.msdn.com/b/pfxteam/archive/2009/06/19/9791857.aspx)
* [Advanced APM Consumption in Async Methods](http://blogs.msdn.com/b/pfxteam/archive/2012/01/23/10259822.aspx)
* [Implementing a SynchronizationContext.SendAsync method](http://blogs.msdn.com/b/pfxteam/archive/2012/01/20/10259082.aspx)
* [Awaiting Socket Operations](http://blogs.msdn.com/b/pfxteam/archive/2011/12/15/10248293.aspx)
* [await anything;](http://blogs.msdn.com/b/pfxteam/archive/2011/01/13/10115642.aspx)
* [The Nature of TaskCompletionSource<TResult>](http://blogs.msdn.com/b/pfxteam/archive/2009/06/02/9685804.aspx)

Does the code generated by async/await result in efficient asynchronous execution?

For the most part, yes, as a lot of work has been done to optimize the code generated by the compiler and the .NET Framework methods on which the generated code relies. For more information, including on best practices for minimizing the overhead of using tasks and async/await, see:

* [TPL Performance Improvements in .NET 4.5](http://blogs.msdn.com/b/pfxteam/archive/2011/11/10/10235962.aspx)
* [“The Zen of Async” at the MVP Summit 2012](http://blogs.msdn.com/b/pfxteam/archive/2012/03/03/10277034.aspx)
* [Asynchronous Programming: Understanding the Costs of Async and Await](http://msdn.microsoft.com/en-us/magazine/hh456402.aspx)

Top of Form

Bottom of Form

Search this blog Search all blogs

**Recent Posts**

* [New Task APIs in .NET 4.6](https://blogs.msdn.microsoft.com/pfxteam/2015/02/02/new-task-apis-in-net-4-6/) February 2, 2015
* [.NET memory allocation profiling and Tasks](https://blogs.msdn.microsoft.com/pfxteam/2013/04/04/net-memory-allocation-profiling-and-tasks/)April 4, 2013
* [Tasks, Monads, and LINQ](https://blogs.msdn.microsoft.com/pfxteam/2013/04/03/tasks-monads-and-linq/) April 3, 2013
* ["Invoke the method with await"… ugh!](https://blogs.msdn.microsoft.com/pfxteam/2013/03/13/invoke-the-method-with-await-ugh/) March 13, 2013

**[RSS](http://sxp.microsoft.com/feeds/3.0/devblogs)**[**Live Now on Developer Tools Blogs**](https://blogs.technet.microsoft.com/cloudplatform)

* [MWC Barcelona 2019—Lenovo announces new devices geared towards Firstline Workers](https://www.microsoft.com/en-us/microsoft-365/blog/2019/02/25/mwc-barcelona-2019-lenovo-announces-new-devices-geared-towards-firstline-workers/)
* [Bing Maps Autosuggest API Now Supports Business Suggestions in 8 New Countries](https://blogs.bing.com/maps/2019-02/bing-maps-autosuggest-supports-business-suggestions-in-8-new-countries)
* [National Bank of Canada reinvents workplace, empowers employees with Microsoft 365](https://officeblogs.wpengine.com/en-us/2019/02/25/national-bank-of-canada-reinvents-workplace-empowers-employees-with-microsoft-365/)

**Tags**

[.NET 4](https://blogs.msdn.microsoft.com/pfxteam/tag/net-4/) [.NET 4.5](https://blogs.msdn.microsoft.com/pfxteam/tag/net-4-5/) [Announcement](https://blogs.msdn.microsoft.com/pfxteam/tag/announcement/)[Article Summary](https://blogs.msdn.microsoft.com/pfxteam/tag/article-summary/) [Async](https://blogs.msdn.microsoft.com/pfxteam/tag/async/) [C++](https://blogs.msdn.microsoft.com/pfxteam/tag/c/)[Cancellation](https://blogs.msdn.microsoft.com/pfxteam/tag/cancellation/) [Code Samples](https://blogs.msdn.microsoft.com/pfxteam/tag/code-samples/)[Coordination Data Structures](https://blogs.msdn.microsoft.com/pfxteam/tag/coordination-data-structures/) [Dataflow](https://blogs.msdn.microsoft.com/pfxteam/tag/dataflow/) [Debugging](https://blogs.msdn.microsoft.com/pfxteam/tag/debugging/)[F#](https://blogs.msdn.microsoft.com/pfxteam/tag/f/) [FAQ](https://blogs.msdn.microsoft.com/pfxteam/tag/faq/) [Feedback Requested](https://blogs.msdn.microsoft.com/pfxteam/tag/feedback-requested/) [Media](https://blogs.msdn.microsoft.com/pfxteam/tag/media/)[Message Passing](https://blogs.msdn.microsoft.com/pfxteam/tag/message-passing/) [MSDN](https://blogs.msdn.microsoft.com/pfxteam/tag/msdn/) [New Feature?](https://blogs.msdn.microsoft.com/pfxteam/tag/new-feature/)[Parallel Extensions](https://blogs.msdn.microsoft.com/pfxteam/tag/parallel-extensions/)[ParallelExtensionsExtras](https://blogs.msdn.microsoft.com/pfxteam/tag/parallelextensionsextras/)[Parallelism Blockers](https://blogs.msdn.microsoft.com/pfxteam/tag/parallelism-blockers/) [PLINQ](https://blogs.msdn.microsoft.com/pfxteam/tag/plinq/)[Release](https://blogs.msdn.microsoft.com/pfxteam/tag/release/) [Silverlight](https://blogs.msdn.microsoft.com/pfxteam/tag/silverlight/) [Talks](https://blogs.msdn.microsoft.com/pfxteam/tag/talks/) [Task Parallel Library](https://blogs.msdn.microsoft.com/pfxteam/tag/task-parallel-library/) [Testing](https://blogs.msdn.microsoft.com/pfxteam/tag/testing/)[ThreadPool](https://blogs.msdn.microsoft.com/pfxteam/tag/threadpool/) [Tools](https://blogs.msdn.microsoft.com/pfxteam/tag/tools/) [Visual Studio](https://blogs.msdn.microsoft.com/pfxteam/tag/visual-studio/)[Visual Studio 2010](https://blogs.msdn.microsoft.com/pfxteam/tag/visual-studio-2010/)

**Videos**

**Related Resources**

* [Visual Studio Product Website](http://www.visualstudio.com/)
* [Visual Studio Developer Center](http://msdn.microsoft.com/vstudio)

**Archives**

 [February 2015](https://blogs.msdn.microsoft.com/pfxteam/2015/02/) (1)

 [April 2013](https://blogs.msdn.microsoft.com/pfxteam/2013/04/) (2)

 [All of 2015](https://blogs.msdn.microsoft.com/pfxteam/2015/) (1)

 [All of 2013](https://blogs.msdn.microsoft.com/pfxteam/2013/) (6)

 [All of 2012](https://blogs.msdn.microsoft.com/pfxteam/2012/) (46)

 [All of 2011](https://blogs.msdn.microsoft.com/pfxteam/2011/) (37)

 [All of 2010](https://blogs.msdn.microsoft.com/pfxteam/2010/) (67)

 [All of 2009](https://blogs.msdn.microsoft.com/pfxteam/2009/) (70)

 [All of 2008](https://blogs.msdn.microsoft.com/pfxteam/2008/) (54)

 [All of 2007](https://blogs.msdn.microsoft.com/pfxteam/2007/) (16)

**Tags**

* [.NET 4.5](https://blogs.msdn.microsoft.com/pfxteam/tag/net-4-5/)
* [Async](https://blogs.msdn.microsoft.com/pfxteam/tag/async/)
* [Parallel Extensions](https://blogs.msdn.microsoft.com/pfxteam/tag/parallel-extensions/)

**Join the conversation**

[Add Comment](https://blogs.msdn.microsoft.com/pfxteam/2012/04/12/asyncawait-faq/)

1. 7 years ago

[Log in to Reply](https://blogs.msdn.microsoft.com/pfxteam/wp-login.php?redirect_to=https%3A%2F%2Fblogs.msdn.microsoft.com%2Fpfxteam%2F2012%2F04%2F12%2Fasyncawait-faq%2F)



[James Chaldecott](https://social.msdn.microsoft.com/profile/James+Chaldecott)

RE: "If there isn’t a SynchronizationContext …In such a case, if an exception goes unhandled, the exception is propagated on the ThreadPool, which with default behavior will cause the process to be terminated."

I thought the new default behavior in 4.5 was to ignore unobserved exceptions? Is this different for async void methods?

1. 7 years ago

[Log in to Reply](https://blogs.msdn.microsoft.com/pfxteam/wp-login.php?redirect_to=https%3A%2F%2Fblogs.msdn.microsoft.com%2Fpfxteam%2F2012%2F04%2F12%2Fasyncawait-faq%2F)



[Rehan Saeed](https://social.msdn.microsoft.com/profile/Rehan+Saeed)

Good Article. Couple of questions.

1. is Task.RunAsynchronously() the same as running an async method synchronously?

2. Which of these implementations would you reccomend:

public interface IService

[

   async Task<Results> GetResults();

]

public interface IService

[

   // Left up to the caller to wrap in a task to run asyncronously.

   Results GetResults();

]

1. 7 years ago

[Log in to Reply](https://blogs.msdn.microsoft.com/pfxteam/wp-login.php?redirect_to=https%3A%2F%2Fblogs.msdn.microsoft.com%2Fpfxteam%2F2012%2F04%2F12%2Fasyncawait-faq%2F)



[Stephen Toub - MSFT](https://social.msdn.microsoft.com/profile/Stephen+Toub+-+MSFT)

James:

The change you allude to was just a change to the unobserved task exception behavior ([blogs.msdn.com/…/10217876.aspx](http://blogs.msdn.com/b/pfxteam/archive/2011/09/28/10217876.aspx)), where if a Task ends in the Faulted state and doesn't then have its exception observed, that exception will trigger the process to crash.  This doesn't apply to "async void" methods, as there is no Task… if it were "async Task", then any unhandled exception would be stored into the returned Task, but for "async void", the exception just propagates to whatever context is current.  And if there is no context, the ThreadPool is presumed (as that's what the base SynchronizationContext type targets).

BlackLightMobile:

Glad you liked the post.

I'm not sure what Task.RunAsynchronously is… there's no such method in .NET 4.5.  Are you referring to Task.Run?  If so, see [blogs.msdn.com/…/10287244.aspx](http://blogs.msdn.com/b/pfxteam/archive/2012/03/24/10287244.aspx). That post should help answer your second question as well.

1. 7 years ago

[Log in to Reply](https://blogs.msdn.microsoft.com/pfxteam/wp-login.php?redirect_to=https%3A%2F%2Fblogs.msdn.microsoft.com%2Fpfxteam%2F2012%2F04%2F12%2Fasyncawait-faq%2F)

https://secure.gravatar.com/avatar/abbd1a40eb3aa12cb685750866e4724f?s=40&d=mm&r=g

RichardDeeming

@Stephen: Looking at BlackLightMobile's second point, would I be right in assuming that you can't use "async" to decorate methods in an interface?

I'd expect to see something more like:

public interface IService

{

  Task<Results> GetResults();

}

class Service1 : IService

{

  public async Task<Results> GetResults()

  {

     …

  }

}

class Service2 : IService

{

  async Task<Results> IService.GetResults()

  {

     …

  }

}

1. 7 years ago

[Log in to Reply](https://blogs.msdn.microsoft.com/pfxteam/wp-login.php?redirect_to=https%3A%2F%2Fblogs.msdn.microsoft.com%2Fpfxteam%2F2012%2F04%2F12%2Fasyncawait-faq%2F)



[Stephen Toub - MSFT](https://social.msdn.microsoft.com/profile/Stephen+Toub+-+MSFT)

Richard: I didn't notice that "async" was used there.  You're correct that "async" can't be used on an interface declaration like that: the async keyword is about the method's implementation, not about its signature, so it doesn't apply to interface.  You can define an interface method to return Task or Task<TResult>, and then you can choose to implement that interface using async/await.

1. 7 years ago

[Log in to Reply](https://blogs.msdn.microsoft.com/pfxteam/wp-login.php?redirect_to=https%3A%2F%2Fblogs.msdn.microsoft.com%2Fpfxteam%2F2012%2F04%2F12%2Fasyncawait-faq%2F)



[Rehan Saeed](https://social.msdn.microsoft.com/profile/Rehan+Saeed)

@Richard Thanks for pointing that out.

@Stephen – I should have said Task with a small 't'.

task.RunAsynchronously

[msdn.microsoft.com/…/dd321435%28v=vs.110%29.aspx](http://msdn.microsoft.com/en-us/library/dd321435%28v=vs.110%29.aspx)

My service is almost always consumed asynchronously so I have everything wrapped in a Task. When someone wants to run it synchronously, they can either start the task and call Wait() or call RunAsyncronously.

I gues this is not good practice. I should just have normal methods and rely the caller to wrap the calls in a task.

Many thanks, I learned a lot from your blog.

1. 7 years ago

[Log in to Reply](https://blogs.msdn.microsoft.com/pfxteam/wp-login.php?redirect_to=https%3A%2F%2Fblogs.msdn.microsoft.com%2Fpfxteam%2F2012%2F04%2F12%2Fasyncawait-faq%2F)



[Stephen Toub - MSFT](https://social.msdn.microsoft.com/profile/Stephen+Toub+-+MSFT)

BlackLightMobile: Oh, you mean RunSynchronously, not RunAsynchronously.  RunSynchronously is really meant to be used within a library, not across library boundaries.  All tasks returned from public API surface area should already be beyond the Created state, and RunSynchronously only applies to tasks in the Created state.

1. 7 years ago

[Log in to Reply](https://blogs.msdn.microsoft.com/pfxteam/wp-login.php?redirect_to=https%3A%2F%2Fblogs.msdn.microsoft.com%2Fpfxteam%2F2012%2F04%2F12%2Fasyncawait-faq%2F)

https://secure.gravatar.com/avatar/7196dc5229aa83157b6f2db30e9b0ba7?s=40&d=mm&r=g

[hfeichtinger@hotmail.com](http://blogs.msdn.com/hfeichtinger_4000_hotmail.com/ProfileUrlRedirect.ashx)

I am developing communication servers. My server handles multiple OSI layer 2 protocols with devices connected via SerialPort or USB. So far I use SerialPort.ReadByte  as the basis of the protocol state machines and Threading.Timer single-shot callbacks for timeout dedection. The state machines use the event-based asynchronous pattern to trigger an answer to the calling clients via WCF.

WCF „events“ work fine inside a .NET environment.

Now I want to rewrite the server using OData in order to support smartphones.

All the async/await samples I’ve seen use asnyc methods of the underlying framework and multiple IO calls in a stateless way –the messages not related to each other.

In my case messages received form a state engine according to a protocol. I am the one building the framework.

cont.

1. 7 years ago

[Log in to Reply](https://blogs.msdn.microsoft.com/pfxteam/wp-login.php?redirect_to=https%3A%2F%2Fblogs.msdn.microsoft.com%2Fpfxteam%2F2012%2F04%2F12%2Fasyncawait-faq%2F)

https://secure.gravatar.com/avatar/7196dc5229aa83157b6f2db30e9b0ba7?s=40&d=mm&r=g

[hfeichtinger@hotmail.com](http://blogs.msdn.com/hfeichtinger_4000_hotmail.com/ProfileUrlRedirect.ashx)

part 2:

Question 1:

Does it make sense, to have a ReadByteAsynch method on the SerialPort?

Question 2:

Will an asnyc/await wrapper over Threading.Timer be as unprecise as System.Timer because theUI thread is used?

Question 3:

When building a framework, would you use async/await internally or only offer it on the API border to the callers? After all, the EAP works fine for years.

Question 4:

When will the OData client proxy code generators for Windows Phone, Iphone/IPad, and Android create async methods?

thank you very much

1. 7 years ago

[Log in to Reply](https://blogs.msdn.microsoft.com/pfxteam/wp-login.php?redirect_to=https%3A%2F%2Fblogs.msdn.microsoft.com%2Fpfxteam%2F2012%2F04%2F12%2Fasyncawait-faq%2F)



[Stephen Toub - MSFT](https://social.msdn.microsoft.com/profile/Stephen+Toub+-+MSFT)

Hi herbertf-

re: #1: I'm not familiar enough with SerialPort to know whether this would be appropriate or not.  You might consider asking in the MSDN Forums.  It looks like SerialPort does expose a "BaseStream", on which you could use ReadAsync/WriteAsync, etc.

re: #2. Are you referring to Task.Delay?  Task.Delay wraps System.Threading.Timer.  It itself doesn't use a UI thread.  If you "await Task.Delay(…);" from the UI thread, then the continuation will be marshaled back to the UI thread, but you could either use "await Task.Delay(….).ConfigureAwait(false);" to avoid that marshaling, or you could use the ContinueWith method on Task directly, e.g. "Task.Delay(…).ContinueWith(…);", as it doesn't involve the UI thread by default.

re: #3. You can certainly use async/await internally.

re: #4.  I don't have any information here.

1. 7 years ago

[Log in to Reply](https://blogs.msdn.microsoft.com/pfxteam/wp-login.php?redirect_to=https%3A%2F%2Fblogs.msdn.microsoft.com%2Fpfxteam%2F2012%2F04%2F12%2Fasyncawait-faq%2F)

https://secure.gravatar.com/avatar/3651e719491c63d4424134d66fe53db9?s=40&d=mm&r=g

robowarrior2

When the "async" method is first invoked a Task object is immediately created.

The method then executes up to the first "await" and quickly returns an as-yet uncompleted Task.

What kind of return? An empty Task object?, a placeholder? or what?

1. 7 years ago

[Log in to Reply](https://blogs.msdn.microsoft.com/pfxteam/wp-login.php?redirect_to=https%3A%2F%2Fblogs.msdn.microsoft.com%2Fpfxteam%2F2012%2F04%2F12%2Fasyncawait-faq%2F)



[Stephen Toub - MSFT](https://social.msdn.microsoft.com/profile/Stephen+Toub+-+MSFT)

robowarrior2, if the method synchronously executes to completion (i.e. there are no awaits in the method, or all of the awaits are on already completed operations), then the task returned will be one that's already completed; this may be a task cached for reuse by the Framework, or it may be a new one that's been created on the spot by the Framework (e.g. via Task.FromResult).  If the method does suspend due to awaiting a not-yet-completed operation, then the Task returned will be one that will later complete when the method completes asynchronously; this is very much like the kind of task returned from a TaskCompletionSource<TResult>'s Task property.

1. 7 years ago

[Log in to Reply](https://blogs.msdn.microsoft.com/pfxteam/wp-login.php?redirect_to=https%3A%2F%2Fblogs.msdn.microsoft.com%2Fpfxteam%2F2012%2F04%2F12%2Fasyncawait-faq%2F)

https://secure.gravatar.com/avatar/3651e719491c63d4424134d66fe53db9?s=40&d=mm&r=g

Joost

Hi,

Thanks for the excellent summary on async/await.

There is one question we have that we can't find an answer for, which is about memory/cache consistency in combination with the Task library. I do hope this a proper place to ask this question.

We are building a library for server-side I/O that solely uses asynchronous stream reads and writes. In that library we have a number of places where we update object fields. We don't use parallelism in the sense that the request handling code is setup as a chain of Task continuations where no two threads will access those fields simultaneously. As no two threads will access those fields in parallel we don't employ any locking mechanism. However, as each async operation may continue on a seperate thread, the fields may be accessed by (many) different threads.

Let me give an example snippet to base the concrete questions on:

class DoesThisNeedVolatile

{

   int totalRead = 0;

   public Task ReadSomeMoreAsync(Stream stream)

   {

…

Task<int> readTask = stream.ReadAsync(buffer, offset, length);

readTask.ContinueWith(rt =>

{

totalRead += rt.Result;

…

});

   }

   public async Task ReadSomeMoreAsync2(Stream stream)

   {

      …

      int read = await stream.ReadAsync(buffer, offset, length);

      totalRead += read;

      …

   }

}

The questions we have with the snippet above are about the visibility of the current value of the totalRead field amongst the different threads that may access it. This counts especially when the methods above are repeatedly invoked (by different threads, but never at the same time).

1. Given the fact that we don't employ parallelism, do we need any form of memory/cache synchronisation at all to make sure that the latest values of fields are correctly seen amongst the different threads/cores?

2. Does the Task library automatically include memory barriers at certain points? E.g. would it include a memory barrier before and/or after a continuation?

3. Is there difference in this respect between the "manual" continuations and the async/await pattern?

4. Should we make the totalRead field volatile to make these snippets work correctly?

Thank you for the great work on async, hope you can help us out here…

With kind regards,

Joost

1. 7 years ago

[Log in to Reply](https://blogs.msdn.microsoft.com/pfxteam/wp-login.php?redirect_to=https%3A%2F%2Fblogs.msdn.microsoft.com%2Fpfxteam%2F2012%2F04%2F12%2Fasyncawait-faq%2F)



[Stephen Toub - MSFT](https://social.msdn.microsoft.com/profile/Stephen+Toub+-+MSFT)

Hi Joost-

1. No.

2. Yes, TPL includes the appropriate barriers when tasks are queued and at the beginning/end of task execution so that values are appropriately made visible.

3. There's no difference in this respect.

4. No, you shouldn't need to do that.

1. 6 years ago

[Log in to Reply](https://blogs.msdn.microsoft.com/pfxteam/wp-login.php?redirect_to=https%3A%2F%2Fblogs.msdn.microsoft.com%2Fpfxteam%2F2012%2F04%2F12%2Fasyncawait-faq%2F)

https://secure.gravatar.com/avatar/3651e719491c63d4424134d66fe53db9?s=40&d=mm&r=g

Nilesh Avhad

Thanks for this a valuable information. Its very well written and  easy to understand  .

1. 6 years ago

[Log in to Reply](https://blogs.msdn.microsoft.com/pfxteam/wp-login.php?redirect_to=https%3A%2F%2Fblogs.msdn.microsoft.com%2Fpfxteam%2F2012%2F04%2F12%2Fasyncawait-faq%2F)

https://secure.gravatar.com/avatar/3651e719491c63d4424134d66fe53db9?s=40&d=mm&r=g

Matt Poland

A Tale of Two Threads

—————————

It was the best of DateTimes, it was the worst of DateTimes, it was the age of patterns, it was the age of code smells, it was the epoch of agile, it was the epoch of waterfall, it was the season of WinRT, it was the season of .Net, it was the spring of multi-core processors, it was the winter of chained callbacks, we had Moore's Law before us, we had dispatchers before us, we were all going to Silicon Valley, we were all going direct the other way – in short, programming was so far like present programming, that some of its noisiest bloggers insisted on its being received, for performance or for complexity, in the superlative degree of design only.

1. 6 years ago

[Log in to Reply](https://blogs.msdn.microsoft.com/pfxteam/wp-login.php?redirect_to=https%3A%2F%2Fblogs.msdn.microsoft.com%2Fpfxteam%2F2012%2F04%2F12%2Fasyncawait-faq%2F)

https://secure.gravatar.com/avatar/3651e719491c63d4424134d66fe53db9?s=40&d=mm&r=g

rqx

I have a problem when using async/await , description as bellow:

[stackoverflow.com/…/mvc-async-error-the-asynchronous-action-method-complete-cannot-be-executed-s](http://stackoverflow.com/questions/15007696/mvc-async-error-the-asynchronous-action-method-complete-cannot-be-executed-s)

1. 6 years ago

[Log in to Reply](https://blogs.msdn.microsoft.com/pfxteam/wp-login.php?redirect_to=https%3A%2F%2Fblogs.msdn.microsoft.com%2Fpfxteam%2F2012%2F04%2F12%2Fasyncawait-faq%2F)

https://secure.gravatar.com/avatar/3651e719491c63d4424134d66fe53db9?s=40&d=mm&r=g

MikeH

Do you test your code? Very bad for a FAQ!

The code with main async is wrong. The line "private static Task FooAsync() " should be "private static async Task FooAsync()"!!!

1. 6 years ago

[Log in to Reply](https://blogs.msdn.microsoft.com/pfxteam/wp-login.php?redirect_to=https%3A%2F%2Fblogs.msdn.microsoft.com%2Fpfxteam%2F2012%2F04%2F12%2Fasyncawait-faq%2F)



[Stephen Toub - MSFT](https://social.msdn.microsoft.com/profile/Stephen+Toub+-+MSFT)

@MikeH: Apologies if I've offended by accidentally omitting a word in a blog post code snippet.  I've fixed it.  Thank you for alerting me to the typo.

1. 5 years ago

[Log in to Reply](https://blogs.msdn.microsoft.com/pfxteam/wp-login.php?redirect_to=https%3A%2F%2Fblogs.msdn.microsoft.com%2Fpfxteam%2F2012%2F04%2F12%2Fasyncawait-faq%2F)



[Cesar Mello](https://social.msdn.microsoft.com/profile/Cesar+Mello)

Your posts are excellent, thank you very much!

1. 4 years ago

[Log in to Reply](https://blogs.msdn.microsoft.com/pfxteam/wp-login.php?redirect_to=https%3A%2F%2Fblogs.msdn.microsoft.com%2Fpfxteam%2F2012%2F04%2F12%2Fasyncawait-faq%2F)

https://secure.gravatar.com/avatar/f3b9817243953c0204d2695f7e7b9472?s=40&d=mm&r=g

Gizi1

Hi,

I am running a sequence of tasks with continuations. Some steps depend on results of previous steps.

The problem is handling the exceptions: I found out that if I use a cancellation token to skip step2 because there was an exception in step1, the final Aggregate exception will have a "System.Threading.Tasks.TaskCanceledException", rather then my exception at step1!

Can you advise how to handle this case?

Thanks, Gizi Ben-Tovim

1. 4 years ago

[Log in to Reply](https://blogs.msdn.microsoft.com/pfxteam/wp-login.php?redirect_to=https%3A%2F%2Fblogs.msdn.microsoft.com%2Fpfxteam%2F2012%2F04%2F12%2Fasyncawait-faq%2F)



[Stephen Toub - MSFT](https://social.msdn.microsoft.com/profile/Stephen+Toub+-+MSFT)

@Gizi1: This is one of the things async/await excels at, sequential composition.  You'd just:

T result = await step1();

await step2(result, cancellationToken);

await step3();

// etc.

Maybe I'm not understanding your scenario.  If this doesn't help, please post a short example that highlights the issues you're hitting.

© 2019 Microsoft Corporation.

* [Terms of Use](http://msdn.microsoft.com/cc300389)

* [Trademarks](http://www.microsoft.com/library/toolbar/3.0/trademarks/en-us.mspx)

* [Privacy & Cookies](http://go.microsoft.com/fwlink/?LinkId=248681)