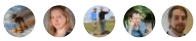
Asynchronous programming

06/20/2016 • 10 minutes to read •  +12

In this article

If you have any I/O-bound needs (such as requesting data from a network or accessing a database), you'll want to utilize asynchronous programming. You could also have CPU-bound code, such as performing an expensive calculation, which is also a good scenario for writing async code.

C# has a language-level asynchronous programming model which allows for easily writing asynchronous code without having to juggle callbacks or conform to a library which supports asynchrony. It follows what is known as the Task-based Asynchronous Pattern (TAP).

Basic Overview of the Asynchronous Model

The core of async programming is the `Task` and `Task<T>` objects, which model asynchronous operations. They are supported by the `async` and `await` keywords. The model is fairly simple in most cases:

For I/O-bound code, you `await` an operation which returns a `Task` or `Task<T>` inside of an `async` method.

For CPU-bound code, you `await` an operation which is started on a background thread with the `Task.Run` method.

The `await` keyword is where the magic happens. It yields control to the caller of the method that performed `await`, and it ultimately allows a UI to be responsive or a service to be elastic.

There are other ways to approach async code than `async` and `await` outlined in the TAP article linked above, but this document will focus on the language-level constructs from this point forward.

I/O-Bound Example: Downloading data from a web service

You may need to download some data from a web service when a button is pressed, but don't want to block the UI thread. It can be accomplished simply like this:

C# Copy

```
private readonly HttpClient _httpClient = new HttpClient();

downloadButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI as the request
    // from the web service is happening.
    //
    // The UI thread is now free to perform other work.
    var stringData = await _httpClient.GetStringAsync(URL);
    DoSomethingWithData(stringData);
};
```

And that's it! The code expresses the intent (downloading some data asynchronously) without getting bogged down in interacting with Task objects.

CPU-bound Example: Performing a Calculation for a Game

Say you're writing a mobile game where pressing a button can inflict damage on many enemies on the screen. Performing the damage calculation can be expensive, and doing it on the UI thread would make the game appear to pause as the calculation is performed!

The best way to handle this is to start a background thread which does the work using `Task.Run`, and `await` its result. This will allow the UI to feel smooth as the work is being done.

C# Copy

```
private DamageResult CalculateDamageDone()
{
    // Code omitted:
    //
    // Does an expensive calculation and returns
    // the result of that calculation.
```

```
}

calculateButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI while CalculateDamageDone()
    // performs its work.  The UI thread is free to perform other work.
    var damageResult = await Task.Run(() => CalculateDamageDone());
    DisplayDamage(damageResult);
};
```

And that's it! This code cleanly expresses the intent of the button's click event, it doesn't require managing a background thread manually, and it does so in a non-blocking way.

What happens under the covers

There's a lot of moving pieces where asynchronous operations are concerned. If you're curious about what's happening underneath the covers of `Task` and `Task<T>`, checkout the Async in-depth article for more information.

On the C# side of things, the compiler transforms your code into a state machine which keeps track of things like yielding execution when an `await` is reached and resuming execution when a background job has finished.

For the theoretically-inclined, this is an implementation of the Promise Model of asynchrony.

Key Pieces to Understand

- Async code can be used for both I/O-bound and CPU-bound code, but differently for each scenario.
- Async code uses `Task<T>` and `Task`, which are constructs used to model work being done in the background.
- The `async` keyword turns a method into an async method, which allows you to use the `await` keyword in its body.
- When the `await` keyword is applied, it suspends the calling method and yields control back to its caller until the awaited task is complete.
- `await` can only be used inside an async method.

Recognize CPU-Bound and I/O-Bound Work

The first two examples of this guide showed how you can use `async` and `await` for I/O-bound and CPU-bound work. It's key that you can identify when a job you need to do is I/O-bound or CPU-bound, because it can greatly affect the performance of your code and could potentially lead to misusing certain constructs.

Here are two questions you should ask before you write any code:

1. Will your code be "waiting" for something, such as data from a database?

   If your answer is "yes", then your work is **I/O-bound**.

2. Will your code be performing a very expensive computation?

   If you answered "yes", then your work is **CPU-bound**.

If the work you have is **I/O-bound**, use `async` and `await` *without* `Task.Run`. You *should not* use the Task Parallel Library. The reason for this is outlined in the Async in Depth article.

If the work you have is **CPU-bound** and you care about responsiveness, use `async` and `await` but spawn the work off on another thread *with* `Task.Run`. If the work is appropriate for concurrency and parallelism, you should also consider using the Task Parallel Library.

Additionally, you should always measure the execution of your code. For example, you may find yourself in a situation where your CPU-bound work is not costly enough compared with the overhead of context switches when multithreading. Every choice has its tradeoff, and you should pick the correct tradeoff for your situation.

More Examples

The following examples demonstrate various ways you can write async code in C#. They cover a few different scenarios you may come across.

Extracting Data from a Network

This snippet downloads the HTML from the homepage at www.dotnetfoundation.org and counts the number of times the string ".NET" occurs in the HTML. It uses ASP.NET MVC to define a web controller method which performs this task, returning the number.

> **Note**

If you plan on doing HTML parsing in production code, don't use regular expressions. Use a parsing library instead.

C# Copy

```
private readonly HttpClient _httpClient = new HttpClient();

[HttpGet]
[Route("DotNetCount")]
public async Task<int> GetDotNetCountAsync()
{
    // Suspends GetDotNetCountAsync() to allow the caller (the web server)
    // to accept another request, rather than blocking on this one.
    var html = await
_httpClient.GetStringAsync("https://dotnetfoundation.org");

    return Regex.Matches(html, @"\.NET").Count;
}
```

Here's the same scenario written for a Universal Windows App, which performs the same task when a Button is pressed:

C# Copy

```
private readonly HttpClient _httpClient = new HttpClient();

private async void SeeTheDotNets_Click(object sender, RoutedEventArgs e)
{
    // Capture the task handle here so we can await the background task later.
    var getDotNetFoundationHtmlTask =
_httpClient.GetStringAsync("https://www.dotnetfoundation.org");

    // Any other work on the UI thread can be done here, such as enabling a
Progress Bar.
    // This is important to do here, before the "await" call, so that the user
    // sees the progress bar before execution of this method is yielded.
    NetworkProgressBar.IsEnabled = true;
    NetworkProgressBar.Visibility = Visibility.Visible;

    // The await operator suspends SeeTheDotNets_Click, returning control to
its caller.
    // This is what allows the app to be responsive and not block the UI
thread.
    var html = await getDotNetFoundationHtmlTask;
    int count = Regex.Matches(html, @"\.NET").Count;

    DotNetCountLabel.Text = $"Number of .NETs on dotnetfoundation.org:
```

```
{count}";

    NetworkProgressBar.IsEnabled = false;
    NetworkProgressBar.Visibility = Visibility.Collapsed;
}
```

Waiting for Multiple Tasks to Complete

You may find yourself in a situation where you need to retrieve multiple pieces of data concurrently. The `Task` API contains two methods, `Task.WhenAll` and `Task.WhenAny` which allow you to write asynchronous code which performs a non-blocking wait on multiple background jobs.

This example shows how you might grab `User` data for a set of `userId`s.

C# Copy

```
public async Task<User> GetUserAsync(int userId)
{
    // Code omitted:
    //
    // Given a user Id {userId}, retrieves a User object corresponding
    // to the entry in the database with {userId} as its Id.
}

public static async Task<IEnumerable<User>> GetUsersAsync(IEnumerable<int>
userIds)
{
    var getUserTasks = new List<Task<User>>();

    foreach (int userId in userIds)
    {
        getUserTasks.Add(GetUserAsync(userId));
    }

    return await Task.WhenAll(getUserTasks);
}
```

Here's another way to write this a bit more succinctly, using LINQ:

C# Copy

```
public async Task<User> GetUserAsync(int userId)
{
    // Code omitted:
    //
```

```
    // Given a user Id {userId}, retrieves a User object corresponding
    // to the entry in the database with {userId} as its Id.
}

public static async Task<User[]> GetUsersAsync(IEnumerable<int> userIds)
{
    var getUserTasks = userIds.Select(id => GetUserAsync(id));
    return await Task.WhenAll(getUserTasks);
}
```

Although it's less code, take care when mixing LINQ with asynchronous code. Because LINQ uses deferred (lazy) execution, async calls won't happen immediately as they do in a `foreach()` loop unless you force the generated sequence to iterate with a call to `.ToList()` or `.ToArray()`.

Important Info and Advice

Although async programming is relatively straightforward, there are some details to keep in mind which can prevent unexpected behavior.

- `async` **methods need to have an** `await` **keyword in their body or they will never yield!**

This is important to keep in mind. If `await` is not used in the body of an `async` method, the C# compiler will generate a warning, but the code will compile and run as if it were a normal method. Note that this would also be incredibly inefficient, as the state machine generated by the C# compiler for the async method would not be accomplishing anything.

- **You should add "Async" as the suffix of every async method name you write.**

This is the convention used in .NET to more-easily differentiate synchronous and asynchronous methods. Note that certain methods which aren't explicitly called by your code (such as event handlers or web controller methods) don't necessarily apply. Because these are not explicitly called by your code, being explicit about their naming isn't as important.

- `async void` **should only be used for event handlers.**

`async void` is the only way to allow asynchronous event handlers to work because events do not have return types (thus cannot make use of `Task` and `Task<T>`). Any other use of `async void` does not follow the TAP model and can be challenging to use, such as:

- Exceptions thrown in an `async void` method can't be caught outside of that method.
- `async void` methods are very difficult to test.
- `async void` methods can cause bad side effects if the caller isn't expecting them to be async.
- **Tread carefully when using async lambdas in LINQ expressions**

Lambda expressions in LINQ use deferred execution, meaning code could end up executing at a time when you're not expecting it to. The introduction of blocking tasks into this can easily result in a deadlock if not written correctly. Additionally, the nesting of asynchronous code like this can also make it more difficult to reason about the execution of the code. Async and LINQ are powerful, but should be used together as carefully and clearly as possible.

- **Write code that awaits Tasks in a non-blocking manner**

Blocking the current thread as a means to wait for a Task to complete can result in deadlocks and blocked context threads, and can require significantly more complex error-handling. The following table provides guidance on how to deal with waiting for Tasks in a non-blocking way:

Use this...	Instead of this...	When wishing to do this
`await`	`Task.Wait` or `Task.Result`	Retrieving the result of a background task
`await Task.WhenAny`	`Task.WaitAny`	Waiting for any task to complete
`await Task.WhenAll`	`Task.WaitAll`	Waiting for all tasks to complete
`await Task.Delay`	`Thread.Sleep`	Waiting for a period of time

- **Write less stateful code**

Don't depend on the state of global objects or the execution of certain methods. Instead, depend only on the return values of methods. Why?

- Code will be easier to reason about.
- Code will be easier to test.
- Mixing async and synchronous code is far simpler.
- Race conditions can typically be avoided altogether.
- Depending on return values makes coordinating async code simple.

- (Bonus) it works really well with dependency injection.

A recommended goal is to achieve complete or near-complete Referential Transparency in your code. Doing so will result in an extremely predictable, testable, and maintainable codebase.

Other Resources

- Async in-depth provides more information about how Tasks work.
- Asynchronous programming with async and await (C#)
- Lucian Wischik's Six Essential Tips for Async are a wonderful resource for async programming

---

Is this page helpful?

Yes No
