What is Asynchronous Programming?

When a computer is running a program, the operating system or the application allocates a group of resources; known as a thread; to run the application code.

In **synchronous** programming, all the code from on application is running on a single thread. Therefore, code has to execute synchronously which means code is executed in a one action at a time. One operation has to finish before the processor moves on to execute the next operation. That means if one operation is taking a long time, the program will freeze and stop waiting for it to finish before it moves to the next operation.

An **asynchronous** operation is an operation that runs on a separate thread that gets initiated from another thread. The thread that initiates an asynchronous operation does not need to wait for that operation to complete before it can continue. In that sense, resources can be allocated to other tasks that can be executed in the meantime.

Why use Asynchronous programming?

you might have already deduced this, asynchronous programming makes the best and most efficient utilization of the machine resources when it comes to having long running operations in an application. For a UI application, Asynchronous programming makes the application responsive because the UI thread is not blocked by a CPU or I/O heavy operation which makes it more interactive and user friendly. Asynchronous programming also takes advantage of parallel computing which is supported in most of today's machines. This helps programmers write parallel processing applications easily without having to write complicated code that is hard to maintain.

When to use Asynchronous Programming

As we mentioned, Asynchronous code is best used for long running operations. Those operations can be CPU bound or I/O bound operations. Examples of long running operations are:

* I/O operations that include Network requests for data retrieval.
* CPU heavy operations like scientific calculations using huge data sets.
* I/O operations like disk access operations including reading and writing to disks.

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 so that your program UI does not freeze waiting for this operation to finish.

What applications are best candidates for asynchronous programming?

* **Desktop User Interface Applications**, a desktop UI application is an application that is expected to be interactive, users should be able to interact and communicate with the various pieces of the application UI mostly at all times. Nothing is worse for an UI application than a frozen control that the user is unable to interact with because of a long running network operation like a web service call. The time spent waiting for information to travel across the network is blocking the UI resources and thus the application appears to be frozen or dead.  
  Desktop / UI modern frameworks give special precedence to the thread that uses the UI. Async code is very important for these technologies to build better UI applications. Examples of UI technologies that use asynchronous programming are:
  + Windows Forms (WinForms)
  + Windows Presentation Foundation (WPF)
  + Universal Windows Platform (UWP)
* **Web Server Applications**, a web server application does not deal with UI but often times, it needs to run remote database queries or run some calculations on large amounts of data to generate a data analysis report. Using Asynchronous programming for these tasks allows the server code to do the task efficiently specially that web servers are usually handling multiple requests from multiple clients at the same time. Asynchronous programming helps avoid situations where threads are simply waiting to do something while the rest of the application is getting overwhelmed by clients requests resulting in less than ideal performance as well as delayed responsiveness to clients requests.

Asynchronous Programming is not always the solution

There are problems that asynchronous code does not solve. There are also problems that Asynchronous code can cause. For example:

1. Asynchronous code does not automatically improve performance. If a network task takes a certain amount of time, asynchronous code is not going to change that. In fact, there is some amount of (small) overhead for the framework managing the process. Rather, asynchronous code helps manage resources more efficiently.
2. Using Asynchronous code imposes an overhead on the system. When threads are used, the system needs a way to manage them like thread saving, scheduling, listening, locking, resuming, ...etc. There are two types of overhead that exist when using threads, *memory overhead* which every thread reserves from a machine's virtual memory when initiated, and a *scheduler overhead* which is the way the operating system uses to manage choosing which thread should be executed on which CPU and when. These overheads can slow down the entire system if threads are excessively used.
3. If you don't have a desktop / UI application, or your code is not network- or I/O-bound, you won't see much benefit.
4. If your application is CPU-bound (it is slowing down because of heavy compute processes), multi-threading / task-based asynchrony is a better solution. We will discuss that in module 2.

## .Net Async Method

 Bookmark this page

## .Net build-in Async Method

You now know that you can avoid performance bottlenecks and enhance the overall responsiveness of your application by using asynchronous programming. However, traditional techniques for writing asynchronous applications can be complicated, making them difficult to write, debug, and maintain.

C# 5 introduced a simplified approach, async programming, that leverages asynchronous support in the .NET Framework 4.5 and higher, .NET Core, and the Windows Runtime. This simplified model makes it easier to perform asynchronous operations. The .Net compiler does the difficult work that the developers used to do, which enables them to concentrate on the business logic of the application. As a result, you get all the advantages of asynchronous programming with a fraction of the effort.

#### async and await keywords

Asynchronous operations are closely related to tasks. The .NET Framework support for async programming makes it easier to perform asynchronous operations by transparently creating new tasks and coordinating their actions on your behalf. In particular, the built-in async and await keywords are the heart of async programming in C#. They enable you to invoke an asynchronous operation and wait for the result within a single method, without blocking the thread.  
Asynchronous methods that you define by using async and await are referred to as async methods.

The following example shows an async method:

// Three things to note in the signature:

// - The method has an async modifier.

// - The return type is Task or Task<T>.

// Here, it is Task<int> because the return statement returns an integer.

// - The method name ends in "Async."

async Task<int> AccessTheWebAsync()

{

// You need to add a reference to System.Net.Http to declare client.

HttpClient client = new HttpClient();

// GetStringAsync returns a Task<string>. That means that when you await the task you'll get a string (urlContents).

Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");

// You can do work here that doesn't rely on the string from GetStringAsync.

DoIndependentWork();

// The await operator suspends AccessTheWebAsync.

// - AccessTheWebAsync can't continue until getStringTask is complete.

// - Meanwhile, control returns to the caller of AccessTheWebAsync.

// - Control resumes here when getStringTask is complete.

// - The await operator then retrieves the string result from getStringTask.

string urlContents = await getStringTask;

// The return statement specifies an integer result.

// Any methods that are awaiting AccessTheWebAsync retrieve the length value.

return urlContents.Length;

}

If AccessTheWebAsync doesn't have any work that it can do between calling GetStringAsync and awaiting its completion, you can simplify your code by calling and awaiting in the following single statement.

string urlContents = await client.GetStringAsync();

## What makes an async method?

 Bookmark this page

##### The following summarizes what makes the previous example an async method:

1. The method signature includes an async modifier.
2. The name of an async method, by convention, ends with an “Async” suffix.
3. The return type is one of the following types:

a. Task<TResult>  
b. Task  
c. void

1. The method includes at least one await expression:

string urlContents = await getStringTask;

which marks a point where the method can't continue until the awaited asynchronous operation is complete. In the meantime, the method is suspended, and control returns to the method's caller.

As mentioned, the core of asynchronous programming are the Task and Task<T> objects, which model asynchronous operations. They are supported by the async and await keywords. In C#, a Task object can be thought of as an ongoing operation. An object of the subclass Task<T>is an operation that will have a result of type T at some point in the future. It is like a promise of a T when the awaited operation completes.

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. This type of work is covered in Module 2 of this course.

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.

Lets look at two examples that show how to use the built -in async method in .Net 4.5 or higher and .Net Core

#### 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:

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);

};

#### 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.

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);

};

## Async method support in .Net Classes

 Bookmark this page

### API async methods

You might be wondering where to find methods such as GetStringAsync that support async programming. The .NET Framework 4.5 or higher and .NET Core contain many members that work with async and await. You can recognize them by the “Async” suffix that’s appended to the member name, and by their return type of Task or Task<TResult>. For example, the System.IO.Stream class contains methods such as CopyToAsync, ReadAsync, and WriteAsync alongside the synchronous methods CopyTo, Read, and Write.

The Windows Runtime also contains many methods that you can use with async and await in Windows apps.

The .Net Core framework has a rich support for async methods. Types in the .Net Core framework that have support for the async methods include (please use the links for complete references for using those types):

### Network-oriented classes with async methods

1. [**HttpClient:**](https://docs.microsoft.com/en-us/dotnet/api/system.net.http.httpclient?view=netcore-2.0) Provides a base class for sending HTTP requests and receiving HTTP responses from a resource identified by a URI.
2. [**WebClient:**](https://docs.microsoft.com/en-us/dotnet/api/system.net.webclient?view=netcore-2.0) provides common methods for sending data to and receiving data from a resource identified by a URI.

### Files I/O-oriented classes with async method

1. [**StreamWriter:**](https://docs.microsoft.com/en-us/dotnet/api/system.io.streamwriter?view=netcore-2.0) provides methods for writing characters to a stream in a particular encoding
2. [**StreamReader:**](https://docs.microsoft.com/en-us/dotnet/api/system.io.streamreader?view=netcore-2.0) this class allows reading of characters from a byte stream in a particular encoding.
3. [**XmlReader:**](https://docs.microsoft.com/en-us/dotnet/api/system.xml.xmlreader?view=netcore-2.0) a class that represents a reader that provides fast, non-cached, forward-only access to XML data.

## Characteristics of an Async Method in .Net Core 2

 Bookmark this page

## Writing async methods in C# and .Net Core 2

To write async methods in C# there are few steps you need to accomplish so that the compiler, as well as the runtime know that this is asynchronous code:

**1- Use the async keyword in the method declaration.**

Use the async modifier to specify that a method, lambda expression, or anonymous method is asynchronous. If you use this modifier on a method or expression, it's referred to as an async method. The following example defines an async method named ExampleMethodAsync:

public async Task<int> ExampleMethodAsync()

{

// . . . .

}

**2- Use the await keyword in the code calling an asynchronous process/method.**  
The await operator is applied to a task in an asynchronous method to insert a suspension point in the execution of the method until the awaited task completes. The task represents ongoing work or a promise. It is not until you await on an async method call that it is actually executing. When you call an async method without using await, you only receive a promise (Task) that you can await on later. This task will give you the promised return type after it completes. In C# syntax, await acts as a unary operator. It is placed to the left of an expression and means to wait for that expression asynchronously with no blocking. If you apply await to a Task<T>, it becomes an await expression, and the whole expression has type T. That means you can assign the result of awaiting to a variable of type T and use it in the rest of the method.

This code calls the async method ExampleMethodAsync above.

int resultValue = await ExampleMethodAsync();

await can **only** be used in an async method modified by the async keyword.

If await is applied to the result of a method call that returns a Task<TResult>, then the type of the await expression is TResult. If await is applied to the result of a method call that returns a Task, then the type of the await expression is void. The following example illustrates the difference.

// await keyword used with a method that returns a Task<TResult>.

TResult result = await AsyncMethodThatReturnsTaskTResult();

// await keyword used with a method that returns a Task.

await AsyncMethodThatReturnsTask();

// await keyword used with a method that returns a ValueTask<TResult>.

TResult result = await AsyncMethodThatReturnsValueTaskTResult();

**3- Use one of the correct return types in the async method in order to get results, we will dive deeper in return types in module 3.**  
return types for an async method need to be one of the following:  
- Task<TResult> if your method has a return statement in which the operand has type TResult.  
- Task if your method has no return statement or has a return statement with no operand.  
- void if you're writing an async event handler.  
- Any other type that has a GetAwaiter method (starting with C# 7).

**4- Adding the “Async” suffix at the end of the method name. This is not required but is considered a convention for writing async methods in C#.**

public async Task<int> ExampleMethodAsync()

{

// . . . .

}

**5- Include at least one await expression in the async method code.**  
The marked async method can use await to designate suspension points. The await operator tells the compiler that the async method can't continue past that point until the awaited asynchronous process is complete. In the meantime, control returns to the caller of the async method. The suspension of an async method at an await expression doesn't constitute an exit from the method, and finally blocks don’t run.

public async Task<int> ExampleMethodAsync()

{

//some code

string pageContents = await client.GetStringAsync(uri);

//some more code

return pageContents.Length;

}

Although an async method typically contains one or more occurrences of an await operator, the absence of await expressions doesn’t cause a compiler error. If an async method doesn’t use an await operator to mark a suspension point, the method executes as a synchronous method does, despite the async modifier. The compiler issues a warning for such methods.

## Characteristics of an Async Method in .Net Core 2

 Bookmark this page

## Writing async methods in C# and .Net Core 2

To write async methods in C# there are few steps you need to accomplish so that the compiler, as well as the runtime know that this is asynchronous code:

**1- Use the async keyword in the method declaration.**

Use the async modifier to specify that a method, lambda expression, or anonymous method is asynchronous. If you use this modifier on a method or expression, it's referred to as an async method. The following example defines an async method named ExampleMethodAsync:

public async Task<int> ExampleMethodAsync()

{

// . . . .

}

**2- Use the await keyword in the code calling an asynchronous process/method.**  
The await operator is applied to a task in an asynchronous method to insert a suspension point in the execution of the method until the awaited task completes. The task represents ongoing work or a promise. It is not until you await on an async method call that it is actually executing. When you call an async method without using await, you only receive a promise (Task) that you can await on later. This task will give you the promised return type after it completes. In C# syntax, await acts as a unary operator. It is placed to the left of an expression and means to wait for that expression asynchronously with no blocking. If you apply await to a Task<T>, it becomes an await expression, and the whole expression has type T. That means you can assign the result of awaiting to a variable of type T and use it in the rest of the method.

This code calls the async method ExampleMethodAsync above.

int resultValue = await ExampleMethodAsync();

await can **only** be used in an async method modified by the async keyword.

If await is applied to the result of a method call that returns a Task<TResult>, then the type of the await expression is TResult. If await is applied to the result of a method call that returns a Task, then the type of the await expression is void. The following example illustrates the difference.

// await keyword used with a method that returns a Task<TResult>.

TResult result = await AsyncMethodThatReturnsTaskTResult();

// await keyword used with a method that returns a Task.

await AsyncMethodThatReturnsTask();

// await keyword used with a method that returns a ValueTask<TResult>.

TResult result = await AsyncMethodThatReturnsValueTaskTResult();

**3- Use one of the correct return types in the async method in order to get results, we will dive deeper in return types in module 3.**  
return types for an async method need to be one of the following:  
- Task<TResult> if your method has a return statement in which the operand has type TResult.  
- Task if your method has no return statement or has a return statement with no operand.  
- void if you're writing an async event handler.  
- Any other type that has a GetAwaiter method (starting with C# 7).

**4- Adding the “Async” suffix at the end of the method name. This is not required but is considered a convention for writing async methods in C#.**

public async Task<int> ExampleMethodAsync()

{

// . . . .

}

**5- Include at least one await expression in the async method code.**  
The marked async method can use await to designate suspension points. The await operator tells the compiler that the async method can't continue past that point until the awaited asynchronous process is complete. In the meantime, control returns to the caller of the async method. The suspension of an async method at an await expression doesn't constitute an exit from the method, and finally blocks don’t run.

public async Task<int> ExampleMethodAsync()

{

//some code

string pageContents = await client.GetStringAsync(uri);

//some more code

return pageContents.Length;

}

Although an async method typically contains one or more occurrences of an await operator, the absence of await expressions doesn’t cause a compiler error. If an async method doesn’t use an await operator to mark a suspension point, the method executes as a synchronous method does, despite the async modifier. The compiler issues a warning for such methods.

## A Glance at legacy

 Bookmark this page

While this is not the focus of our course, it is worth mentioning that in earlier versions of C# and .Net Framework, asynchronous programming used to be achieved by using callbacks and delegates. This was the standard method of asynchronous coding in the .NET Framework, before async/await languages features came out in C#5 and .Net 4.5. You may come across these in legacy code if you have to deal with it. However, it is not recommended for new code to use callbacks or delegates for asynchronous operations. Async methods with the async/await keywords is the standard way to go now.

Some routine processes, such as loops and exception handling, can be difficult to handle in traditional asynchronous code. In an async method, you write these elements much as you would in a synchronous solution, and the problem is solved.

##### Other methods of asynchronoud coding:

Legacy code may use patterns such as Asynchronous Programming Model (APM) and Event-based Asynchronous Pattern (EAP). The [Task Parallel Library (TPL)](https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/tpl-and-traditional-async-programming) can be used in various ways in conjunction with either of the asynchronous patterns. We will examine these patterns more closely in Module 4. For now, you just need to know that async/await represent the evolution of asynchronous pattern and should be your go-forward.