https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/task-asynchronous-programming-model

The async and await keywords in C# are the heart of async programming. By using those two keywords, you can use resources in .NET Framework, .NET Core, or the Windows Runtime to create an asynchronous method almost as easily as you create a synchronous method. Asynchronous methods that you define by using the async keyword are referred to as *async methods*.

The following example shows an async method. Almost everything in the code should look familiar to you.

You can find a complete Windows Presentation Foundation (WPF) example available for download from Asynchronous programming with async and await in C#.

C#Copy
```
public async Task<int> GetUrlContentLengthAsync()
{
   var client = new HttpClient();

   Task<string> getStringTask =
      client.GetStringAsync("https://docs.microsoft.com/dotnet");

   DoIndependentWork();

   string contents = await getStringTask;

   return contents.Length;
}

void DoIndependentWork()
{
   Console.WriteLine("Working...");
}
```

You can learn several practices from the preceding sample. Start with the method signature. It includes the async modifier. The return type is Task<int> (See "Return Types" section for more options). The method name ends in Async. In the body of the method, GetStringAsync returns a Task<string>. That means that when you await the task you'll get a string (contents). Before awaiting the task, you can do work that doesn't rely on the string from GetStringAsync.

Pay close attention to the await operator. It suspends GetUrlContentLengthAsync:

- GetUrlContentLengthAsync can't continue until getStringTask is complete.
- Meanwhile, control returns to the caller of GetUrlContentLengthAsync.
- Control resumes here when getStringTask is complete.

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

The return statement specifies an integer result. Any methods that are awaiting GetUrlContentLengthAsync retrieve the length value.

If GetUrlContentLengthAsync 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.

C#Copy
```
string contents = await client.GetStringAsync("https://docs.microsoft.com/dotnet");
```

The following characteristics summarize what makes the previous example an async method:

- The method signature includes an async modifier.
- The name of an async method, by convention, ends with an "Async" suffix.
- The return type is one of the following types:
  - 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# 7.0).

  For more information, see the Return types and parameters section.

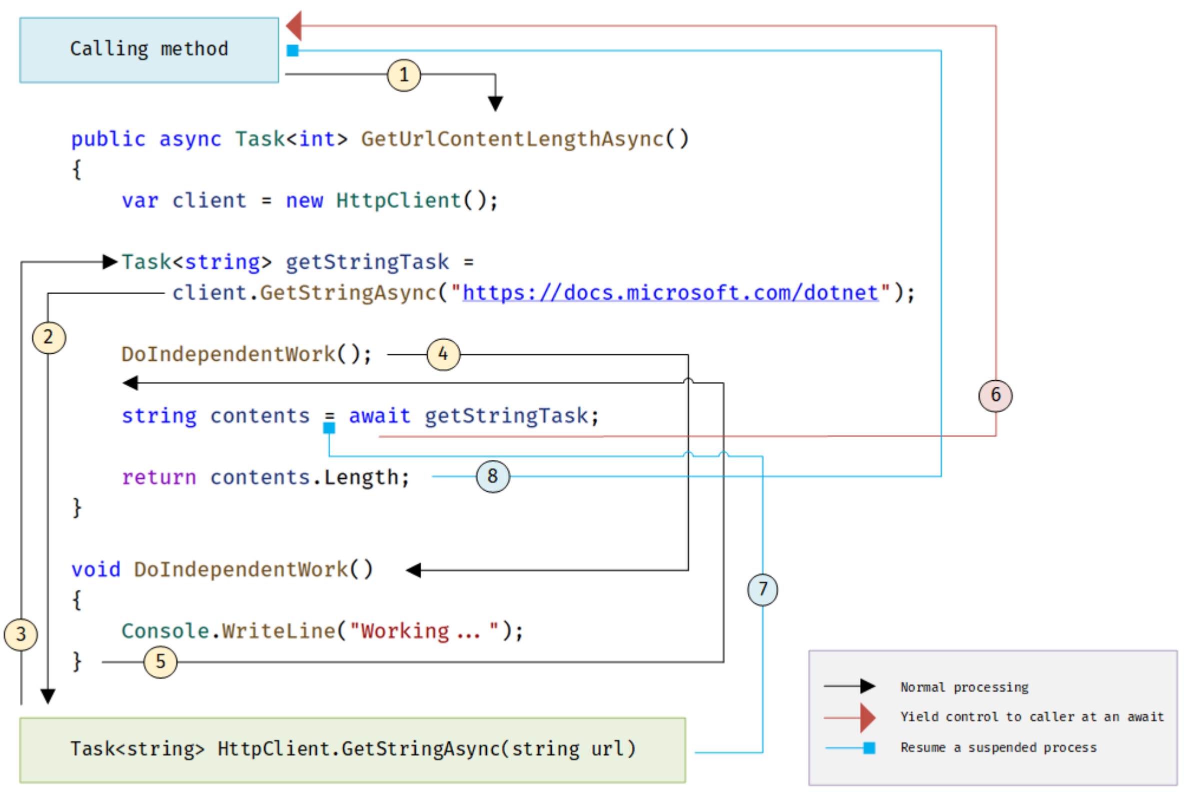
- The method usually includes at least one await expression, which marks a point where the method can't continue until the awaited asynchronous operation is complete. In the meantime, the method is suspended, and control returns to the method's caller. The next section of this topic illustrates what happens at the suspension point.

In async methods, you use the provided keywords and types to indicate what you want to do, and the compiler does the rest, including keeping track of what must happen when control returns to an await point in a suspended method. 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.

For more information about asynchrony in previous versions of .NET Framework, see TPL and traditional .NET Framework asynchronous programming.

What happens in an async method

The most important thing to understand in asynchronous programming is how the control flow moves from method to method. The following diagram leads you through the process:


```
public async Task<int> GetUrlContentLengthAsync()
{
    var client = new HttpClient();

    Task<string> getStringTask =
        client.GetStringAsync("https://docs.microsoft.com/dotnet");

    DoIndependentWork();

    string contents = await getStringTask;

    return contents.Length;
}

void DoIndependentWork()
{
    Console.WriteLine("Working ... ");
}
```

```
Task<string> HttpClient.GetStringAsync(string url)
```
Calling method
1
2
3
4
5
6
7
8
Normal processing
Yield control to caller at an await
Resume a suspended process

The numbers in the diagram correspond to the following steps, initiated when a calling method calls the async method.

1. A calling method calls and awaits the GetUrlContentLengthAsync async method.
2. GetUrlContentLengthAsync creates an HttpClient instance and calls the GetStringAsync asynchronous method to download the contents of a website as a string.
3. Something happens in GetStringAsync that suspends its progress. Perhaps it must wait for a website to download or some other blocking activity. To avoid blocking resources, GetStringAsync yields control to its caller, GetUrlContentLengthAsync.

   GetStringAsync returns a Task<TResult>, where TResult is a string, and GetUrlContentLengthAsync assigns the task to the getStringTask variable. The task represents the ongoing process for the call to GetStringAsync, with a commitment to produce an actual string value when the work is complete.

4. Because getStringTask hasn't been awaited yet, GetUrlContentLengthAsync can continue with other work that doesn't depend on the final result from GetStringAsync. That work is represented by a call to the synchronous method DoIndependentWork.
5. DoIndependentWork is a synchronous method that does its work and returns to its caller.
6. GetUrlContentLengthAsync has run out of work that it can do without a result from getStringTask. GetUrlContentLengthAsync next wants to calculate and return the length of the downloaded string, but the method can't calculate that value until the method has the string.

Therefore, GetUrlContentLengthAsync uses an await operator to suspend its progress and to yield control to the method that
called GetUrlContentLengthAsync. GetUrlContentLengthAsync returns a Task<int> to the caller. The task represents a promise to produce an integer result that's the length of the downloaded string.

Note

If GetStringAsync (and therefore getStringTask) completes
before GetUrlContentLengthAsync awaits it, control remains
in GetUrlContentLengthAsync. The expense of suspending and then returning
to GetUrlContentLengthAsync would be wasted if the called asynchronous
process getStringTask has already completed and GetUrlContentLengthAsync doesn't have to wait for the final result.

Inside the calling method the processing pattern continues. The caller might do other work that doesn't depend on the result from GetUrlContentLengthAsync before awaiting that result, or the caller might await immediately. The calling method is waiting for GetUrlContentLengthAsync, and GetUrlContentLengthAsync is waiting
for GetStringAsync.

7. GetStringAsync completes and produces a string result. The string result isn't returned by the call to GetStringAsync in the way that you might expect. (Remember that the method already returned a task in step 3.) Instead, the string result is stored in the task that represents the completion of the method, getStringTask. The await operator retrieves the result from getStringTask. The assignment statement assigns the retrieved result to contents.
8. When GetUrlContentLengthAsync has the string result, the method can calculate the length of the string. Then the work of GetUrlContentLengthAsync is also complete, and the waiting event handler can resume. In the full example at the end of the topic, you can confirm that the event handler retrieves and prints the value of the length result. If you are new to asynchronous programming, take a minute to consider the difference between synchronous and asynchronous behavior. A synchronous method returns when its work is complete (step 5), but an async method returns a task value when its work is suspended (steps 3 and 6). When the async method eventually completes its work, the task is marked as completed and the result, if any, is stored in the task.