**Async and await in c# example**

With asynchronous programming, you can divide your logic into awaitable tasks, where you can perform some long running operations such as reading large file, doing an API call, downloading a resource from web or performing a complex calculation without having to block the execution of your application on UI or service.

.Net framework gives you simple and easy to use keywords which are the async and await modifiers to transform your code from synchronous to asynchronous.

The main benefits of asynchronous programming using async / await include the following:

* Increase the performance and **responsiveness** of your application, particularly when you have long-running operations that do not require to block the execution. In this case, you can perform other work while waiting for the result from the long running task.
* **Organize your code** in a neat and readable way significantly better than boilerplate code of the traditional thread creation and handling. with async / await , you write less code and your code will be more maintainable than using the previous asynchronous programming methods such as using plain tasks.

**Why write Asynchronous ?**

Usually, you want to write asynchronous code for 2 different types of purposes or needs:

* **I/O bound code.** This is when you want to do an input/output operation, particularly, downloading big resource from network, reading a huge file or accessing a database resouce. In this case, you use the await keyword on an async method that returns a Task or Task<T>.
* **CPU bound code.** This is when you want to do a heavy in-app calculation, such as calculating and displaying the remaining distance to reach the finish line of a car racing game. Image what will happen if the was done synchronously and the UI was blocked while calculating the remaining distance when the car is moving?! Therefore, In the CPU bound case, you use the await keyword on an async method that will be running on a background thread using the method Task.Run()

In both scenarios, your application's UI or service's responsiveness or running state should not be blocked or affected.

**Structure of Async and Await**

Turning a normal method to become asynchronous and make it use the async / await keywords, should be achieved by the following changes:

* Method definition should include the keyword async, this keyword by itself doesn't do anything except enabling you to use the keyword await within the method.
* Method return type should change to return either void or Task or Task<T> , where T is the return data type, so in this example it will become

public async Task<String> GetUserNameAsync(){ }

* According to the naming convention, an asynchronous method name should end with the word 'Async' , so if you have a method with name GetUserName, it should become GetUserNameAsync

When you add the keyword async to the method definition, it will enable you to use the await keyword within this method, which means you can await the method in the way you need. and if you do not include the keyword await, then the method will be treated as a normal or synchronous method.

An important note here is that even though returning void in an async method is allowed, it should not be used in most cases, as the other 2 return types Task and Task<T> represent void and T subsequently, after the awaitable method completes and returns result. So the use of void as return type should be only limited for **event handlers**.

using System;

using System.Net.Http;

using System.Threading.Tasks;

namespace AsyncApp

{

class Program

{

private const string URL = "https://docs.microsoft.com/en-us/dotnet/csharp/csharp";

static void Main(string[] args)

{

DoSynchronousWork();

var someTask = DoSomethingAsync();

DoSynchronousWorkAfterAwait();

someTask.Wait(); //this is a blocking call, use it only on Main method

Console.ReadLine();

}

public static void DoSynchronousWork()

{

// You can do whatever work is needed here

Console.WriteLine("1. Doing some work synchronously");

}

static async Task DoSomethingAsync() //A Task return type will eventually yield a void

{

Console.WriteLine("2. Async task has started...");

await GetStringAsync(); // we are awaiting the Async Method GetStringAsync

}

static async Task GetStringAsync()

{

using (var httpClient = new HttpClient())

{

Console.WriteLine("3. Awaiting the result of GetStringAsync of Http Client...");

string result = await httpClient.GetStringAsync(URL); //execution pauses here while awaiting GetStringAsync to complete

//From this line and below, the execution will resume once the above awaitable is done

//using await keyword, it will do the magic of unwrapping the Task<string> into string (result variable)

Console.WriteLine("4. The awaited task has completed. Let's get the content length...");

Console.WriteLine($"5. The length of http Get for {URL}");

Console.WriteLine($"6. {result.Length} character");

}

}

static void DoSynchronousWorkAfterAwait()

{

//This is the work we can do while waiting for the awaited Async Task to complete

Console.WriteLine("7. While waiting for the async task to finish, we can do some unrelated work...");

for (var i = 0; i <= 5; i++)

{

for (var j = i; j <= 5; j++)

{

Console.Write("\*");

}

Console.WriteLine();

}

}

}

}

**Sample examples of async and await keyword in C#**

We are going to take a console Application for our demonstration.

**Example 1**

In this example, we are going to take two methods, which are not dependent on each other.

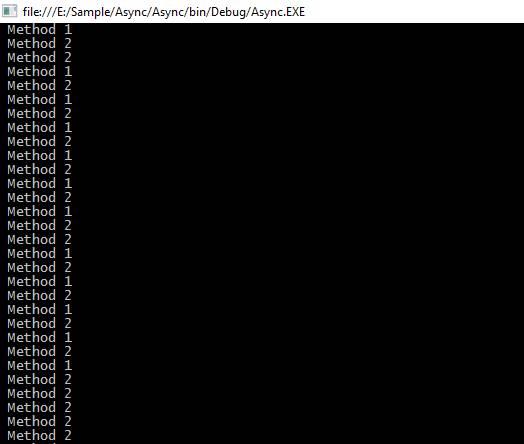
**Code sample**

1. **class** Program
2. {
3. **static** **void** Main(**string**[] args)
4. {
5. Method1();
6. Method2();
7. Console.ReadKey();
8. }
10. **public** **static** async Task Method1()
11. {
12. await Task.Run(() =>
13. {
14. **for** (**int** i = 0; i < 100; i++)
15. {
16. Console.WriteLine(" Method 1");
17. }
18. });
19. }

22. **public** **static** **void** Method2()
23. {
24. **for** (**int** i = 0; i < 25; i++)
25. {
26. Console.WriteLine(" Method 2");
27. }
28. }
29. }

In the code given above, Method1 and Method2 are not dependent on each other and we are calling from the Main method.

Here, we can clearly see Method1 and Method2 are not waiting for each other.

**Output  
  
**

Now, coming to the second example, suppose we have Method3, which is dependent on Method1

**Example 2**

In this example, Method1 is returning total length as an integer value and we are passing a parameter as a length in a Method3, which is coming from Method1.

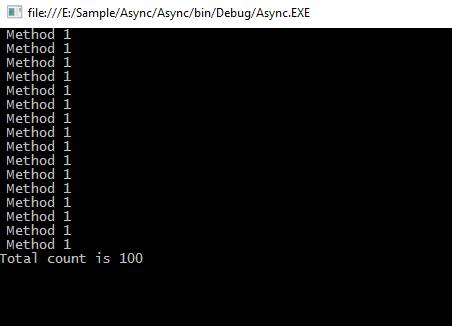
Here, we have to use await keyword before passing a parameter in Method3 and for it, we have to use the async keyword from the calling method.

We can not use await keyword without async and we cannot use async keyword in the Main method for the console Application because it will give the error given below.  
  
  
We are going to create a new method as callMethod and in this method, we are going to call our all Methods as Method1, Method2 and Method3 respectively.

**Code sample**

1. **class** Program
2. {
3. **static** **void** Main(**string**[] args)
4. {
5. callMethod();
6. Console.ReadKey();
7. }
9. **public** **static** async **void** callMethod()
10. {
11. Task<**int**> task = Method1();
12. Method2();
13. **int** count = await task;
14. Method3(count);
15. }
17. **public** **static** async Task<**int**> Method1()
18. {
19. **int** count = 0;
20. await Task.Run(() =>
21. {
22. **for** (**int** i = 0; i < 100; i++)
23. {
24. Console.WriteLine(" Method 1");
25. count += 1;
26. }
27. });
28. **return** count;
29. }
31. **public** **static** **void** Method2()
32. {
33. **for** (**int** i = 0; i < 25; i++)
34. {
35. Console.WriteLine(" Method 2");
36. }
37. }
39. **public** **static** **void** Method3(**int** count)
40. {
41. Console.WriteLine("Total count is " + count);
42. }
43. }

In the code given above, Method3 requires one parameter, which is the return type of Method1. Here, await keyword is playing a vital role for waiting of Method1 task completion.

**Output  
  
**

**Real time example**

There are some supporting API's from the .NET Framework 4.5 and the Windows runtime contains methods that support async programming.

We can use all of these in the real time project with the help of async and await keyword for the faster execution of the task.

Some APIs that contain async methods are HttpClient, SyndicationClient, StorageFile, StreamWriter, StreamReader, XmlReader, MediaCapture, BitmapEncoder, BitmapDecoder etc.

In this example, we are going to read all the characters from a large text file asynchronously and get the total length of all the characters.

**Sample code**

1. **class** Program
2. {
3. **static** **void** Main()
4. {
5. Task task = **new** Task(CallMethod);
6. task.Start();
7. task.Wait();
8. Console.ReadLine();
9. }
11. **static** async **void** CallMethod()
12. {
13. **string** filePath = "E:\\sampleFile.txt";
14. Task<**int**> task = ReadFile(filePath);
16. Console.WriteLine(" Other Work 1");
17. Console.WriteLine(" Other Work 2");
18. Console.WriteLine(" Other Work 3");
20. **int** length = await task;
21. Console.WriteLine(" Total length: " + length);
23. Console.WriteLine(" After work 1");
24. Console.WriteLine(" After work 2");
25. }
27. **static** async Task<**int**> ReadFile(**string** file)
28. {
29. **int** length = 0;
31. Console.WriteLine(" File reading is stating");
32. **using** (StreamReader reader = **new** StreamReader(file))
33. {
34. // Reads all characters from the current position to the end of the stream asynchronously
35. // and returns them as one string.
36. **string** s = await reader.ReadToEndAsync();
38. length = s.Length;
39. }
40. Console.WriteLine(" File reading is completed");
41. **return** length;
42. }
43. }

In the code given above, we are calling a ReadFile method to read the contents of a text file and get the length of the total characters present in the text file.

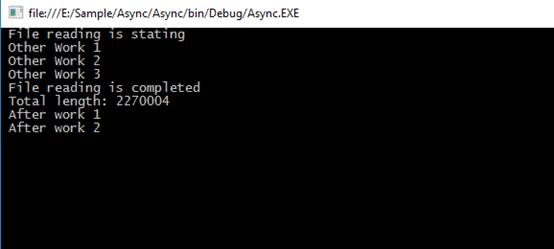
In our sampleText.txt, file contains too many characters, so It will take a long time to read all the characters.

Here, we are using async programming to read all the contents from the file, so it will not wait to get a return value from this method and execute the other lines of code but it has to wait for the line of code given below because we are using await keyword and we are going to use the return value for the line of code given below..

1. **int** length = await task;
2. Console.WriteLine(" Total length: " + length);

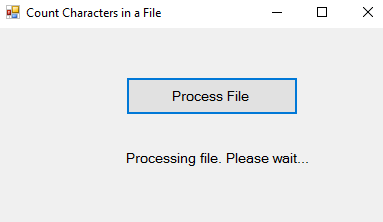
Subsequently, other lines of code will be executed sequentially.

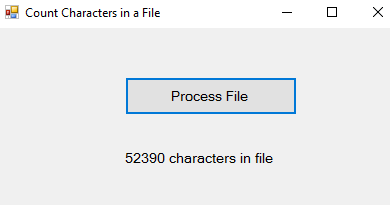
1. Console.WriteLine(" After work 1");
2. Console.WriteLine(" After work 2");

**Output  
  
**

Here, we have to understand very important points that if we are not using await keyword, then the method works as a synchronous method. The compiler will show the warning to us but it will not show any error.

we will discuss **async**and **await**keywords and their purpose with an example.  
  
Let us create a simple Windows Forms Application that counts the number of characters in a given file. Let us assume the file is very big and it takes around 5 seconds to read and count the number of characters in the file. When the **"Process File Button"** is clicked, the application should display the message **"Processing File. Please wait"**.



As soon as the application finishes processing the file it should display the the number of characters as shown below.   
  
   
  
Another improtant requirement is that the application should remain responsive throughout the entire process, i.e when the application is busy processing the file the application should not hang and we should still be able to interact with the application. We should be able to click with in the other controls on the form, move the form around on the screen, resize it if required etc.  
  
Let us first create the Windows Forms Application without using async and await keywords and see how it behaves. Here are the steps.

1. In your D: drive, create a new folder. Name it **Data**. In the folder create a new Text Document. Name it **Data.txt**. Type some text in the file and save it. The application that we are going to create, counts the number of characters in this file.  
  
2. Create a New **"Windows Forms Application"**. Name it AsyncExample.  
  
3. Drag and Drop a **"Button"** on the Form and set the following properties  
   Name = btnProcessFIle  
   Font - Size = 10  
   Text = Process File  
  
4. Drag and Drop a **"Label"** on the Form and set the following properties  
   Name = lblCount  
   Font - Size = 10  
   Text = ""  
  
5. Double Click on the **"Button"** control to generate the "Click" event handler  
  
6. Copy and paste the following code in Form1.cs

using System;

using System.IO;

using System.Threading;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace AsyncExample

{

    public partial class Form1 : Form

    {

        public Form1()

        {

            InitializeComponent();

        }

        private int CountCharacters()

        {

            int count = 0;

            // Create a StreamReader and point it to the file to read

            using (StreamReader reader = new StreamReader("C:\\Data\\Data.txt"))

            {

                string content = reader.ReadToEnd();

                count = content.Length;

                // Make the program look busy for 5 seconds

                Thread.Sleep(5000);

            }

            return count;

        }

        private void btnProcessFIle\_Click(object sender, EventArgs e)

        {

            lblCount.Text = "Processing file. Please wait...";

            int count = CountCharacters();

            lblCount.Text = count.ToString() + " characters in file";

        }

    }

}

7. Run the application and click the **"Process File"** button. You will notice the following problems.

* The application does not display the status message i.e "Processing file. Please wait."
* While the application is busy processing the file, it becomes unresponsive. You cannot move the form window or resize it.

These problems can be very easily fixed by using the **async** and **await**keywords. Notice only the **btnProcessFIle\_Click**() event handler method needs to change.

// Make the method async by using the async keyword

private async void btnProcessFIle\_Click(object sender, EventArgs e)

{

    // Create a task to execute CountCharacters() function

    // CountCharacters() function returns int, so we created Task<int>

    Task<int> task = new Task<int>(CountCharacters);

    task.Start();

    lblCount.Text = "Processing file. Please wait...";

    // Wait until the long running task completes

    int count = await task;

    lblCount.Text = count.ToString() + " characters in file";

}

Now, when we click the **"Process File"** button, notice

* The application displays the status message (**"Processing file. Please wait"**) immediately.
* Even when the application is busy processing the file, it is responsive. You can move the form window around or resize it.

**So what is the use of async and await keywords in C#**  
async and await keywords are used to create asynchronous methods. The async keyword specifies that a method is an asynchronous method and the await keyword specifies a suspension point. The await operator signalls 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.  
  
An async method typically contains one or more occurrences of an await operator, but the absence of await expressions doesn’t cause a compiler error.  
  
**You may have a few questions at this point.**  
1. Can't we achieve the same thing using a Thread.   
2. What is the difference between a Thread and a Task  
3. When to use a Task over Thread and vice-versa

### c# wait for thread to finish without blocking

we discussed creating a simple responsive windows forms application using Task, and async & await keywords. In this video we will discuss how to do the same using a Thread instead of Task.

To use a Thread instead of a Task we only need to change btnProcessFile\_Click() method as shown below. 

private void btnProcessFile\_Click(object sender, EventArgs e)

{

    int count = 0;

    Thread thread = new Thread(() => { count = CountCharacters(); });

    thread.Start();

    lblCount.Text = "Processing file. Please wait...";

    lblCount.Text = count.ToString() + " characters in file";

}

At this point the application does not work as expected. We have two problems with the above code.  
1. We do not see the message, "Processing file. Please wait." at all  
2. It displays "0 characters in file"  
  
**Why is this happening**  
The Main thread i.e the UI thread has created a worker thread which executes CountCharacters() function. The worker thread takes at least 5 seconds to complete. In the mean time the Main thread continues executing the following 2 lines of code.

lblCount.Text = "Processing file. Please wait...";

lblCount.Text = count.ToString() + " characters in file";

**But why didn't we see the message "Processing file. Please wait..."**  
This is because, the UI thread executes the above 2 lines of code so fast that the second message overwrites the first message and at that speed it is impossible for a human eye to spot the overwriting.  
  
**How to solve the above two problems**  
It is very simple. The Main thread has to wait for the worker thread to finish it's work before the UI thread can display the second message. We achieve this by using Join() method on the worker thread.

private void btnProcessFile\_Click(object sender, EventArgs e)

{

    int count = 0;

    Thread thread = new Thread(() => { count = CountCharacters(); });

    thread.Start();

    lblCount.Text = "Processing file. Please wait...";

    // Join() blocks the Main thread (UI Thread)

    thread.Join();

    lblCount.Text = count.ToString() + " characters in file";

}

At this point run the application and test it. We have fixed the above two problems but introduced a new problem. While the application is busy processing the file, the UI is blocked i.e we cannot move the form around or resize it.  
  
You may be thinking why can't we move the code that updates the label control Text property into the worker thread as shown below. This is dangerous because, the thread that has created the control must modify the control. In our case the Main thread (i.e UI Thread) is the thread that has created the label control so only the Main thread should set it's Text property and not the worker thread. If you run the application it may or may not work as expected. If it is working, it is only working by blind luck. 

private void btnProcessFile\_Click(object sender, EventArgs e)

{

    int count = 0;

    Thread thread = new Thread(() =>

    {

        count = CountCharacters();

        // This is dangerous

        lblCount.Text = count.ToString() + " characters in file";

    });

    thread.Start();

    lblCount.Text = "Processing file. Please wait...";

}

The right way to achieve this is by using BeginInvoke() method as shown below. BeginInvoke() method asks the UI thread to set the Text property of the label control in a type safe manner.

private void btnProcessFile\_Click(object sender, EventArgs e)

{

    int count = 0;

    Thread thread = new Thread(() =>

    {

        count = CountCharacters();

        Action action = () => lblCount.Text = count.ToString() + " characters in file";

        this.BeginInvoke(action);

    });

    thread.Start();

    lblCount.Text = "Processing file. Please wait...";

}

In the example above, notice that the Action delegate points to a piece of code. The Action delegate is then passed to the BeginInvoke() method which asks the UI thread to execute that piece of code asynchronously in a type safe manner. The above code can also be rewritten as shown below.

int characterCount = 0;

private void btnProcessFile\_Click(object sender, EventArgs e)

{

    Thread thread = new Thread(() =>

    {

        characterCount = CountCharacters();

        // Action delegate points to SetLabelTextProperty method

        // Signature of SetLabelTextProperty() method should match

        // with the signature of Action delegate

        Action action = new Action(SetLabelTextProperty);

        this.BeginInvoke(action);

    });

    thread.Start();

    lblCount.Text = "Processing file. Please wait...";

}

private void SetLabelTextProperty()

{

    lblCount.Text = characterCount.ToString() + " characters in file";

}

Asynchronous implementation is very easy with tasks, and async & await keywords. Though the above example is a very simple example, notice the code is already getting relatively complicated. Imagine if we have multiple threads, and we want to use the result of one thread from another thread and so on and so forth. It can get painful and complicated. In our previous video, we have seen how easy it is to achieve exactly the same thing using a Task.