Asynchronous programming is very popular with the help of the async and await keywords in C#. When we are dealing with UI, and on button click, we use a long-running method like reading a large file or something else which will take a long time, in that case, the entire application must wait to complete the whole task. In other words, if any process is blocked in a synchronous application, the whole application gets blocked, and our application stops responding until the whole task completes.

Asynchronous programming is very helpful in this condition. By using Asynchronous programming, the Application can continue with the other work that does not depend on the completion of the entire task.

## Code examples of C# async await

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**

class Program

{

static void Main(string[] args)

{

Method1();

Method2();

Console.ReadKey();

}

public static async Task Method1()

{

await Task.Run(() =>

{

for (int i = 0; i < 100; i++)

{

Console.WriteLine(" Method 1");

// Do something

Task.Delay(100).Wait();

}

});

}

public static void Method2()

{

for (int i = 0; i < 25; i++)

{

Console.WriteLine(" Method 2");

// Do something

Task.Delay(100).Wait();

}

}

}

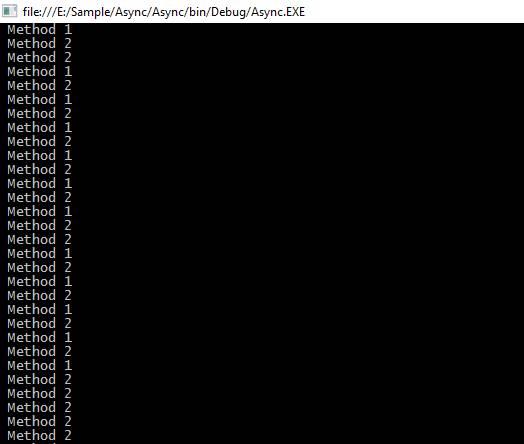
C#

Copy

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 the 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.

If we are using C# 7 or less, then we cannot use async keyword in the Main method for the console Application because it will give the error 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** **C# 7**

class Program

{

static void Main(string[] args)

{

callMethod();

Console.ReadKey();

}

public static async void callMethod()

{

Task<int> task = Method1();

Method2();

int count = await task;

Method3(count);

}

public static async Task<int> Method1()

{

int count = 0;

await Task.Run(() =>

{

for (int i = 0; i < 100; i++)

{

Console.WriteLine(" Method 1");

count += 1;

}

});

return count;

}

public static void Method2()

{

for (int i = 0; i < 25; i++)

{

Console.WriteLine(" Method 2");

}

}

public static void Method3(int count)

{

Console.WriteLine("Total count is " + count);

}

}

C#

Copy

**Code sample C# 9**

class Program

{

static async Task Main(string[] args)

{

await callMethod();

Console.ReadKey();

}

public static async Task callMethod()

{

Method2();

var count = await Method1();

Method3(count);

}

public static async Task<int> Method1()

{

int count = 0;

await Task.Run(() =>

{

for (int i = 0; i < 100; i++)

{

Console.WriteLine(" Method 1");

count += 1;

}

});

return count;

}

public static void Method2()

{

for (int i = 0; i < 25; i++)

{

Console.WriteLine(" Method 2");

}

}

public static void Method3(int count)

{

Console.WriteLine("Total count is " + count);

}

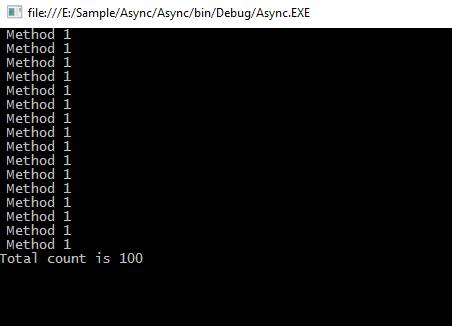
}

C#

Copy

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**

class Program

{ ke

static void Main()

{

Task task = new Task(CallMethod);

task.Start();

task.Wait();

Console.ReadLine();

}

static async void CallMethod()

{

string filePath = "E:\\sampleFile.txt";

Task<int> task = ReadFile(filePath);

Console.WriteLine(" Other Work 1");

Console.WriteLine(" Other Work 2");

Console.WriteLine(" Other Work 3");

int length = await task;

Console.WriteLine(" Total length: " + length);

Console.WriteLine(" After work 1");

Console.WriteLine(" After work 2");

}

static async Task<int> ReadFile(string file)

{

int length = 0;

Console.WriteLine(" File reading is stating");

using (StreamReader reader = new StreamReader(file))

{

// Reads all characters from the current position to the end of the stream asynchronously

// and returns them as one string.

string s = await reader.ReadToEndAsync();

length = s.Length;

}

Console.WriteLine(" File reading is completed");

return length;

}

}

C#

Copy

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, the 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. Still it has to wait for the line of code given below because we are using await keywords, and we are going to use the return value for the line of code given below.

int length = await task;

Console.WriteLine(" Total length: " + length);

C#

Copy

Subsequently, other lines of code will be executed sequentially.

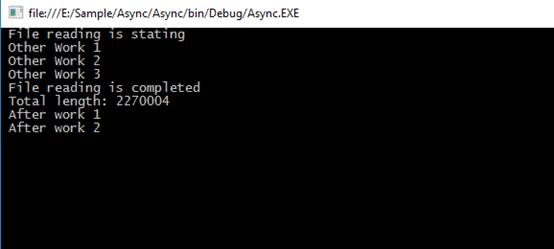
Console.WriteLine(" After work 1");

Console.WriteLine(" After work 2");

C#

Copy

**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 can use async and await keywords in C# to implement async programming in this easy way,

C# supports both synchronous and asynchronous methods. Let's learn the difference between synchronous and asynchronous and how to code in C#.

Interestingly enough, any method we normally create in C# is synchronous by default. For example, the following method fetches data from a database and binds it to a TextBox synchronously.

1. **private** **void** LoadData() {
2. // Create connection
3. SqlConnection conn = **new** SqlConnection(@ "network address= .; integrated
4. security = **true**; database = EmployeeDb ");
5. // Create command
6. **string** sql = @ "select EmpId,Name
7. from dbo.EmployeeDetails where EmpID <= 500 ";
8. // Data binding code goes here
9. **try** {
10. // Open connection
11. conn.Open();
12. // Execute query via ExecuteReader
13. SqlDataReader rdr = cmd.ExecuteReader();
14. **while** (rdr.Read()) {
15. txtReader.AppendText("\nEmpID: ");
16. txtReader.AppendText(rdr.GetValue(1) + "\t\t" + rdr.GetValue(0));
17. txtReader.AppendText("\n");
18. }
19. } **catch** (SqlException ex) {
20. MessageBox.Show(ex.Message + ex.StackTrace, "Exception Details");
21. } **finally** {
22. conn.Close();
23. }
24. }

What is Synchronous

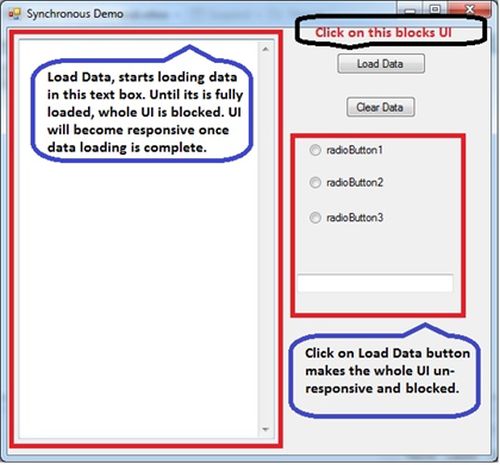
* Synchronous represents a set of activities that starts happening together at the same time.
* A synchronous call waits for the method to complete before continuing with program flow.

**How bad is it?**

* It badly impacts the UI that has just one thread to run its entire user interface code.
* Synchronous behavior leaves end users with a bad user experience and a blocked UI whenever the user attempts to perform some lengthy (time-consuming) operation.

Business Scenario and Problem Statement

Consider a real-world business case in which a UI binds data to the data grid by fetching it from the database. While data is being fetched and bound to the grid the rest of the UI is blocked. Any attempt of interaction with other UI controls will not be evident until the data loading is over. This UI blockage gets over when data fetch-and-binding is completely done. Refer to "Figure 1-1 Synchronous Behavior" below

  
Figure 1-1 Synchronous Behavior

Solution to the Synchronous Problem

A synchronous method call can create a delay in program execution that causes a bad user experience. Hence, an asynchronous approach (threads) will be better. An asynchronous method call (cretion of a thread) will return immediately so that the program can perform other operations while the called method completes its work in certain situations.  
  
The asynchronous method's behavior is different than synchronous ones because an asynchronous method is a separate thread. You create the thread; the thread starts executing, but control is immediately returned back to the thread that called them time; while the other thread continues to execute.  
  
In general, asynchronous programming makes sense in two cases as,

* If you are creating a UI intensive application in which the user experience is the prime concern. In this case, an asynchronous call allows the user interface to remain responsive. Unlike as shown in Figure 1-1.
* If you have other complex or expensive computational work to do, you can continue; interacting with the application UI while wait for the response back from the long-running task.

Asynchronous Patterns

There are various ways to use threads in applications. These recipes are known as Patterns.  
  
**Asynchronous Programming Model Pattern**

* Relies on two corresponding methods to represent an asynchronous operation: BeginMethodName and EndMethodName
* Most often you must have seen this while using delegates or method invocation from a Web Service.

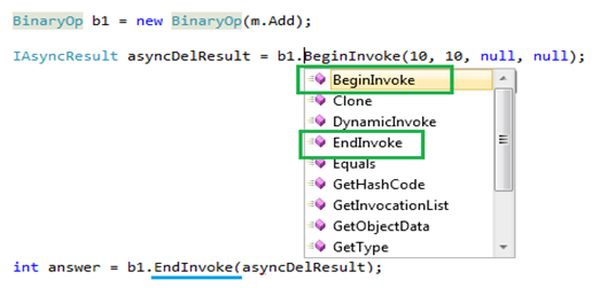


Figure 1-2 APM Pattern

**Event Based Asynchronous Pattern**

* The Event-based Asynchronous Pattern has a single MethodNameAsync method and a corresponding MethodNameCompleted event
* Basically, this pattern enforces a pair of methods and an event to collaborate and help the application execute a thread asynchronously

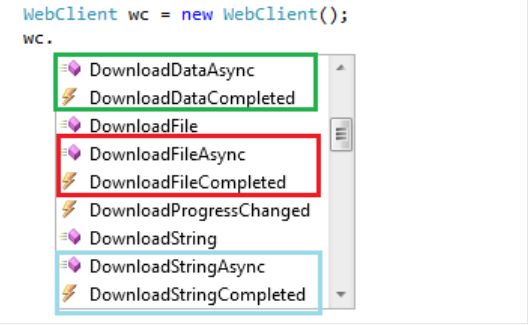


Figure 1-3 Event Based Pattern

**Task based Asynchronous Pattern**

* The Microsoft .NET Framework 4.0 introduces a new Task Parallel Library (TPL) for parallel computing and asynchronous programming. The namespace is "System.Threading.Tasks".
* A Task can represent an asynchronous operation and a Task provides an abstraction over creating and pooling threads.

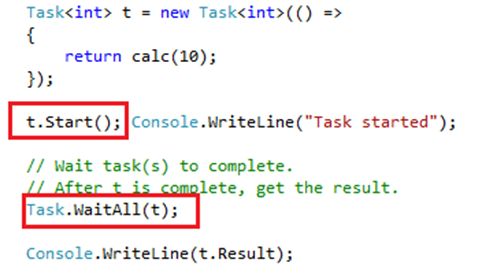


Figure 1-4 Task Based Pattern

C# 5.0 async and await based Asynchronous Pattern

* Two new keywords, async and await, were introduced in C# 5.0 and .NET 4.5. These are implemented at the compiler level and built on top of the "System.Threading.Tasks.Task" feature of .NET 4.0.
* To work with async and await, you must have Visual Studio 2012

1. async **void** LoadEmployee\_Click(**object** sender, RoutedEventArgs e) {
2. // ...
3. await viewer.LoadEmplployeeAsync();
4. // ...
5. }

**Problem with older Asynchrnous Patterns**  
With earlier patterns, the programmer needed to do all the plumbing and collaboration between a pair of methods (BeginMethod and EndMethod) or a method and an event (MethodAsync and MethodCompleted) to make them functional; see Figure 1-2 APM Pattern. This approach was a tedious job not only in terms of syntax but also from sequence of statements inside the method body.  
  
C# 5.0 async/await offers a completely different and easy way to do asynchronous programming. With this feature it's no longer the responsibility of the programmer to do the syntax related tedious work, rather this is now done by the keywords (C# 5.0 async / await) provided by the programming language.  
  
As a result, asynchronous code is easy to implement and retain its logical structure. Hence now it is as easy as writing your normal method without concern of any extra plumbing and so on. As shown in other asynchronous patterns in which you need to deal with a pair of methods or a combination of methods and events and so on.  
  
**Business Scenario**  
Consider a real-world business case, a WPF UI binding data to the data grid by fetching a large number of rows from a database. While data is being fetched and bound to a grid, the rest of the UI should continue to be responsive. Any attempt at interaction with other UI controls must **not** be blocked and data loading and binding must continue in parallel.. Refer to "Figure 1-1 Synchronous Behavior" below.

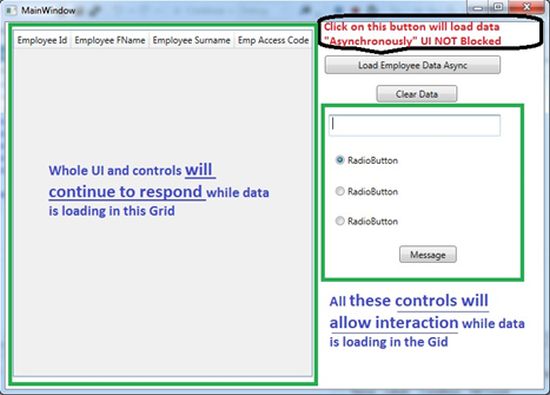


Figure 1-5 Asynchronous Behavior

**Let's Code**  
If you look at the code below, it looks like normal code as shown at the very beginning of this article. The differences worth noting are highlighted in yellow in the code block below.

1. Private async **void** LoadCustomersAsync() {
2. **using**(EmployeeDbEntities ent = **new** EmployeeDbEntities()) {
3. IdbConnection conn = (ent.Connection **as** EntityConnection).StoreConnection;
4. conn.Open();
5. **using**(DbCommand cmd = (DbCommand) conn.CreateCommand()) {
6. var query = from p **in** ent.EmployeeDetails
7. where p.Name.Contains("FN") && p.SurName.Contains("SN") && (p.Name + p.SurName).Length > 3
8. select p;
9. //Convert linq query to SQL statement for CommandText
10. **string** str = ((ObjectQuery) query).ToTraceString();
11. cmd.CommandText = str;
12. // Invoke Async flavor of ExecuteReader
13. var task = await cmd.ExecuteReaderAsync();
14. //translate retieved data to entity customer
15. var cust1 = await Task.Run(
16. () => ent.Translate < EmployeeDetails > (task).ToList < EmployeeDetails > ());
17. employeesDataGrid.ItemsSource = cust1;
18. }
19. }
20. }

As you noticed, the flow looks very natural and no extra plumbing appears in the code. Except async/await, task and of course the asynchronous flavor of the main function that is retrieving data from the database; in our case, ExecuteReaderAsync() is the method.  
  
This code will allow you to perform UI interaction; when data is being fetched and grid binding is taking place, refer to the Figure 1-6 async/await in action.

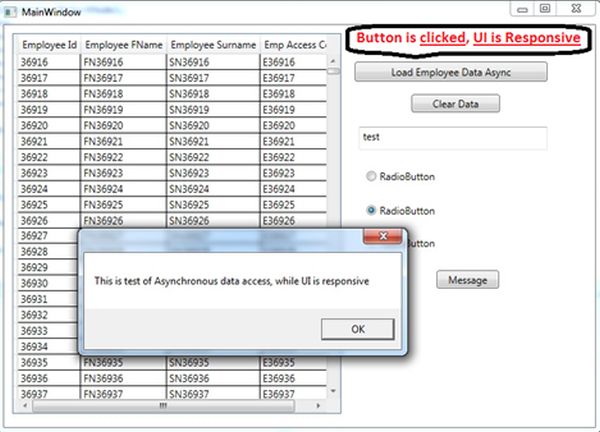


Figure 1-6 async/await in action (as you can see in image 36K + rows pulled)

**Legacy Operations**  
  
Microsoft suggests that with the release of .NET 4.5, the following commonly used methods should be considered as legacy operations. When possible and if you are usng .NET 4.5 then you must use async and await to do asynchronous programming in your application.

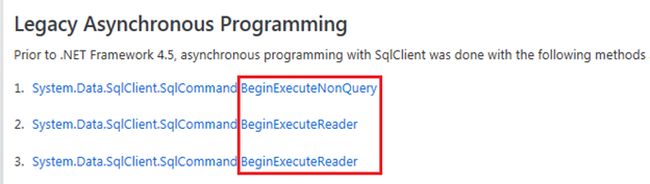


Figure 1-7 Legacy Operations

**What if you don't have Visual Studio 2012**Since Visual Studio 2012 is still not adopted by many development teams in various organizations and many developers still use Visual Studio 2010. So, can they use async and await syntax there?  
  
Microsoft released an async CTP that is supposed to work well with Visual Studio 2010 (without SP1) and allow the developers to use the same syntax.  
  
Search for "async CTP" in Bing or Google.

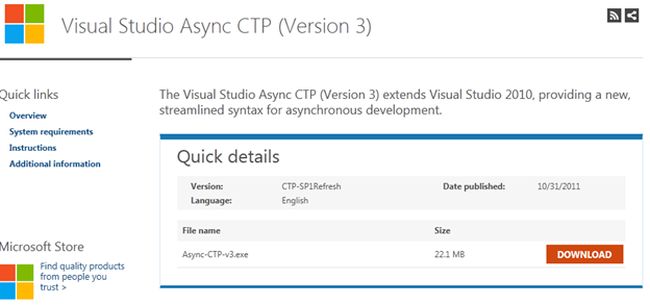


Figure 1-8 Async CTP download page

**Side-by-Side Comparison of various ways techniques**

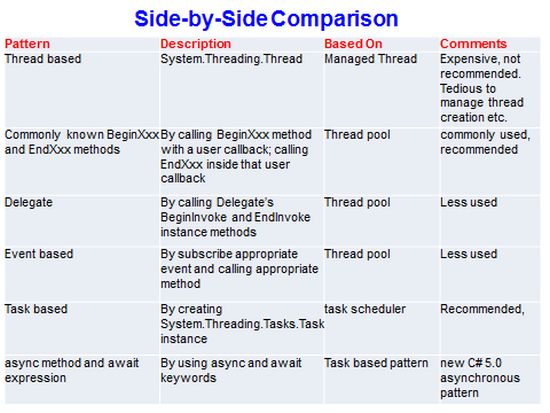


Figure 1-9 Side-by-Side comparison on various techniques

[Asynchronous Programming With C# (c-sharpcorner.com)](https://www.c-sharpcorner.com/UploadFile/84c85b/asynchronous-programming-with-C-Sharp-5-0/)