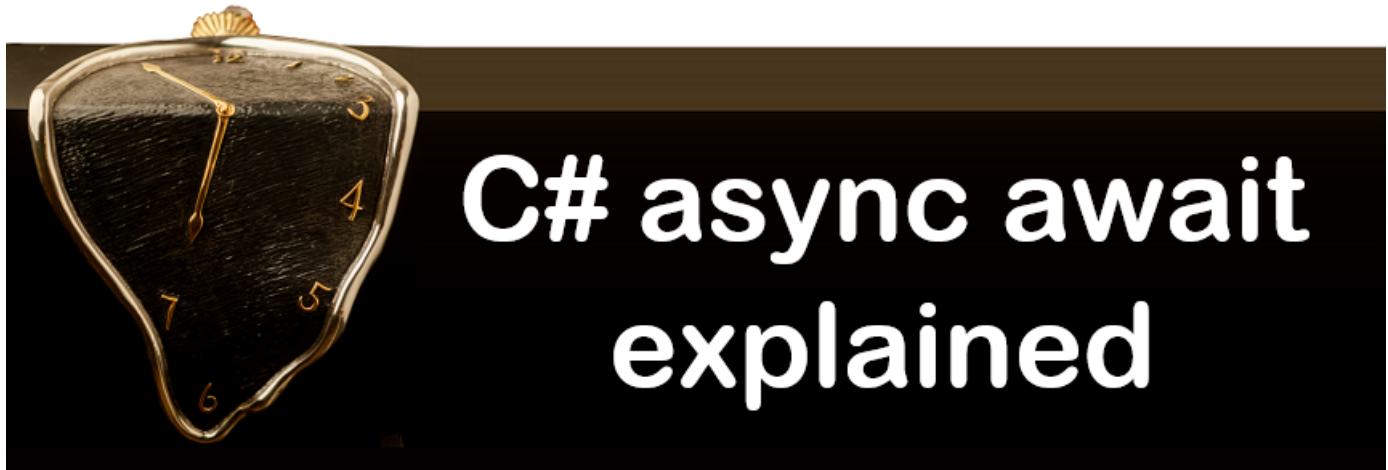


NEW Version v2020.1.2

NDepend

Improve your .NET code quality with NDepend

C# async await explained



In 2012, C#5 was released. This version introduced two new keywords `async` and `await`. At that time CPU clock speed reached an upper limit imposed by physical laws. But chip makers started to deliver CPU with several cores that can run tasks in parallel. Thus C# needed a way to ease asynchronous programming.

The `async` and `await` keywords make asynchronous programming *almost* too easy. Many programmers use them often without really understanding the runtime workflow. This is a great thing, they can focus more on the business of their applications and less on asynchronous details. But some disconcerting behaviors might (and will) happen. Thus it is preferable that one understands the logic behind `async` and `await` and what can influence it. This is the goal of the present article.

Calling an async method

Here is a small C# program that illustrates the `async` and `await` keywords. Two tasks A and B runs simultaneously. Task A runs within a method marked as `async`, while B is executed after calling the `async` method.

C#

```

1  class Program {
2
3      static async Task Main() {
4          Console.WriteLine($"Start Program");
5
6          Task<int> taskA = MethodAAsync();
7
8          for (int i = 0; i < 5; i++) {
9              Console.WriteLine($" B{i}");
10             Task.Delay(50).Wait();
11         }
12
13         Console.WriteLine("Wait for taskA termination");
14
15         await taskA;
16
17         Console.WriteLine($"The result of taskA is {taskA.Result}");
18         Console.ReadKey();
19     }
20
21     static async Task<int> MethodAAsync() {
22         for (int i = 0; i < 5; i++) {
23             Console.WriteLine($" A{i}");
24             await Task.Delay(100);
25         }
26         int result = 123;
27         Console.WriteLine($" A returns result {result}");
28         return result;
29     }
30
31     // Convenient helper to print colorful threadId on console
32     static void Console.WriteLine(string str) {
33         int threadId = Thread.CurrentThread.ManagedThreadId;
34         Console.ForegroundColor = threadId == 1 ? ConsoleColor.White : ConsoleColor.Cyan;
35         Console.WriteLine(
36             $"{str}{new string(' ', 26 - str.Length)} Thread {threadId}");
37     }
38 }

```

Here is the result:

```

Start Program      Thread 1
A0                 Thread 1
B0                 Thread 1
B1                 Thread 1
A1                 Thread 7
B2                 Thread 1
B3                 Thread 1
A2                 Thread 4
B4                 Thread 1
A3                 Thread 7
Wait for taskA termination Thread 1
A4                 Thread 7
A returns result 123 Thread 7
The result of taskA is 123 Thread 7

```

Before explaining in details how the two occurrences of the keyword `async` modify the workflow, let's make some remarks:

The method with the modifier `async` is named `MethodAAsync()`. The method name suffix `Async` is not mandatory but widely used, also in the .NET Base Class Library (BCL). This suffix can be ignored in methods with common name like `Button1_Click()` or `Main()`. By the way `Main()` is also marked with `async` in the code above, more on this point later.

In the `async` method `MethodAAsync()`, once the keyword `await` is met for the first time the remaining of the task is actually executed by some random threads obtained from the runtime thread pool.

As a consequence the call to the `async` method `MethodAAsync()` is not blocking the main thread. First it prints `A0` on the console and then returns to run the task `B` synchronously while the task `A` continues on other threads.

This is why the `async` method `MethodAAsync()` returns a `Task<int>` named `taskA`. This task represents the remaining course of `MethodAAsync()` that will print `A1`, `A2`, `A3`, `A4` and then returns an integer result.

Thread 1 and then Thread 4 and 7 are involved to run task `A`. Each time the keyword `await` is executed, one cannot predict the pool thread that will be used to run the code remaining. Keep in mind that this behavior results from running within a console application context where there is no `SynchronizationContext` (this will be explained in a later section).

Similarly, in the main method the code after `await taskA;` is executed on a random pool thread. Here it appears to be the same thread that executed the last part of `MethodAAsync()`.

First let's explain the easy role of the `async` keyword. Then we'll have a closer look at the influence of the `await` keyword.

The easy role of the `async` keyword

It is important to note that only the keyword `await` does mysterious things here. `async` is just here to decorate a method to tell the C# compiler that this method contains at least one `await` keyword. The C# compiler could be smart enough to detect that a method contains an `await` keyword. However `async` was introduced both for readability and for backward compatibility to avoid breaking existing code that used `await` as a variable name:

```

class Program {
    0 references
    static /*async*/ Task Main() {
        Console.WriteLine(str:"Start Program");

        Task<int> taskA = MethodAAsync();

        for (int i = 0; i < 5; i++) {
            Console.WriteLine(str:" B{i}");
            Task.Delay(50).Wait();
        }

        Console.WriteLine(str:"Wait for taskA termination");

        await taskA;
    }
}

```

CS0246: The type or namespace name 'await' could not be found (are you missing a using directive or an assembly reference?)

Cannot resolve symbol 'await'

Show potential fixes (Ctrl+;)

Consequently, an async method with no await keyword is executed synchronously. A warning is emitted in this situation.

```

1 reference
static async Task<int> MethodAAsync() {
    for (int i = 0; i < 5; i++) {
        Console.WriteLine(str:" B{i}");
        //await Task.Delay(100);
        Task.Delay(100).Wait();
    }
    int result = 123;
    Console.WriteLine(str:" A returns result {result}");
    return result;
}

```

This async method lacks 'await' operators and will run synchronously. Consider using the 'await' operator to await non-blocking API calls, or 'await Task.Run(...)' to do CPU-bound work on a background thread

From now **keep in mind that the keyword async is just a decorator that tells the C# compiler that the method contains at least one occurrence of the await keyword**. By the way, since the main method also contains the await keyword it must also be declared as async and also returns a Task. A main method can be declared as async since C# 7.1.

Understanding the await workflow

In the short program above there are two occurrences of the keyword await, in the Main() method and in the MethodAAsync() method. We now know that await can only be mentioned

within a method with the modifier `async`. Also in both places the keyword `await` is immediately followed by a `Task` or `Task<TResult>` object. To understand the `await` workflow there are 3 points to carefully take account:

1) The caller point of view: Once the keyword `await` is met for the first time in an `async` method, the currently executing thread immediately returns. This is why we say that a call to an `async` method is not blocking. It means that when a thread is calling an `async` method, it might not use its result immediately. Instead it got a **promise of result**, which is the `Task<TResult>` object returned. The thread can do some work (task B here) and then `await` on the task later when it finally needs the result. By the way [the similar javascript construct is called a promise](#).

2) The awaited asynchronous task: `await` is called on a task object, that is not the task returned by the `async` method.

The task might be started at that point as in `await Task.Delay(100);` that simulates a computation intensive task or an I/O bound task. It could be replaced with something like `await Task.Run(() => { ...computation intensive task running on a pool thread... });`.

Or the task might already be running, as in the `await taskA;` in the `Main()` method.

3) The task returned by the `async` method is the code remaining once the awaited task terminates: The beauty is that the `await` keyword doesn't lead to any wasted thread awaiting the task ending. When the task finishes (eventually with a result in the case `Task<TResult>`) the infrastructure behind the `await` keyword chooses a thread to resume the remaining code in the `async` method that is after the keyword `await`. This remaining code to run is nested in a task object. This is the task object returned by the `async` method.

In the `async` method `MethodAAsync()` the code after the `await` keyword is the remaining loops and then the code that returns the result.

In the `async` `Main()` method, the code after the `await` keyword is `Console.WriteLine($"The result of taskA is {taskA.Result}");` followed by `Console.ReadKey();`.

What's often not well understood is that there are really 2 tasks involved in an `async` method:

The task following the `await` keyword that runs the CPU bound or I/O bound code.

The task returned by the `async` method that represents the remaining code to run upon the awaited task termination.

In fact in this short program above, there are much more than 2 tasks involved at runtime! These few lines of code are more subtle than they look because in `MethodAAsync()`, the keyword `await` is met in each loop and each time `await Task.Delay(100);` simulates a new task. As a consequence at each loop a new task is created to run the remaining code once the task `Task.Delay(100);` terminates. So **taskA returned by `MethodAAsync()` is concretely a chain of tasks** and each loop can be ran by a different thread. We can see in the console output that the pool threads with IDs 7 and 4 are involved to run sub-tasks of taskA. Notice that the first loop that prints A0 executed by the main thread is not a part of taskA.

The magic behind the C# `await` keyword

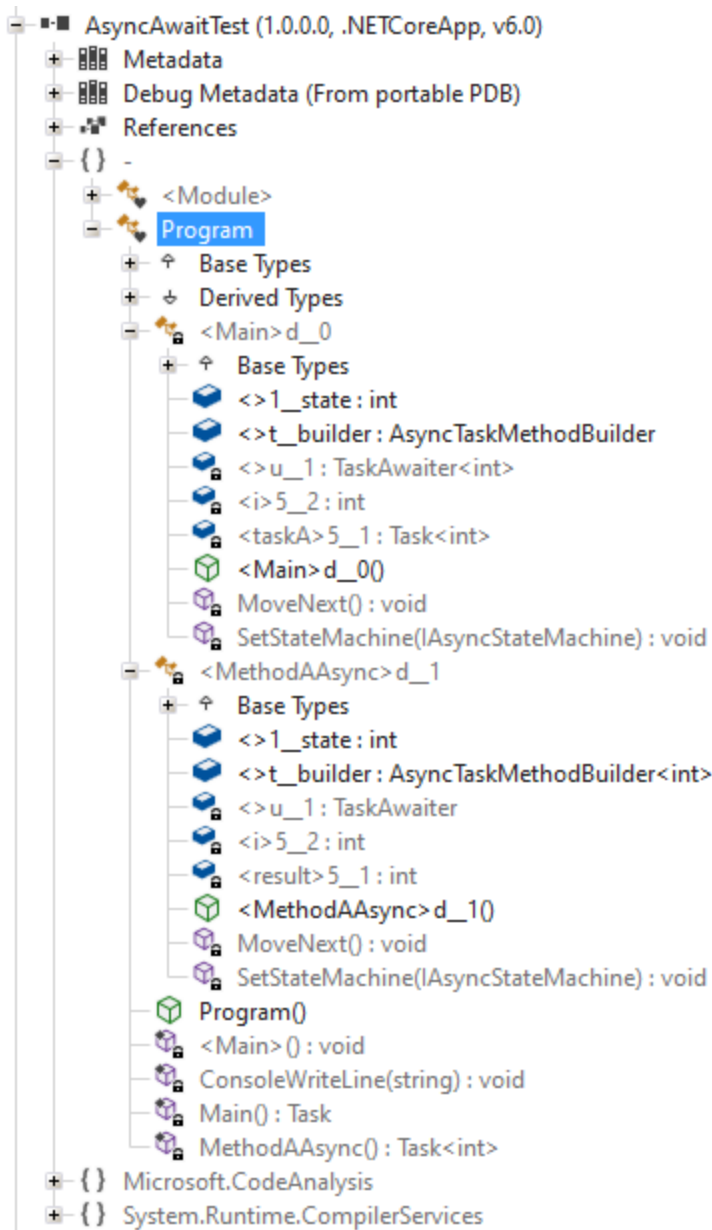
Now that we detailed the await keyword workflow we can measure how powerful it is. Some magic does occur under the hood to resume the execution once the task finishes. Let's have a look at the thread stack trace after `await taskA;` in the main method.

```
C#
1    ...
2    Console.WriteLine("Wait for taskA termination");
3    await taskA;
4
5    Console.WriteLine(new System.Diagnostics.StackTrace());
6
7    Console.WriteLine($"The result of taskA is {taskA.Result}");
8    Console.ReadKey();
9 }
```

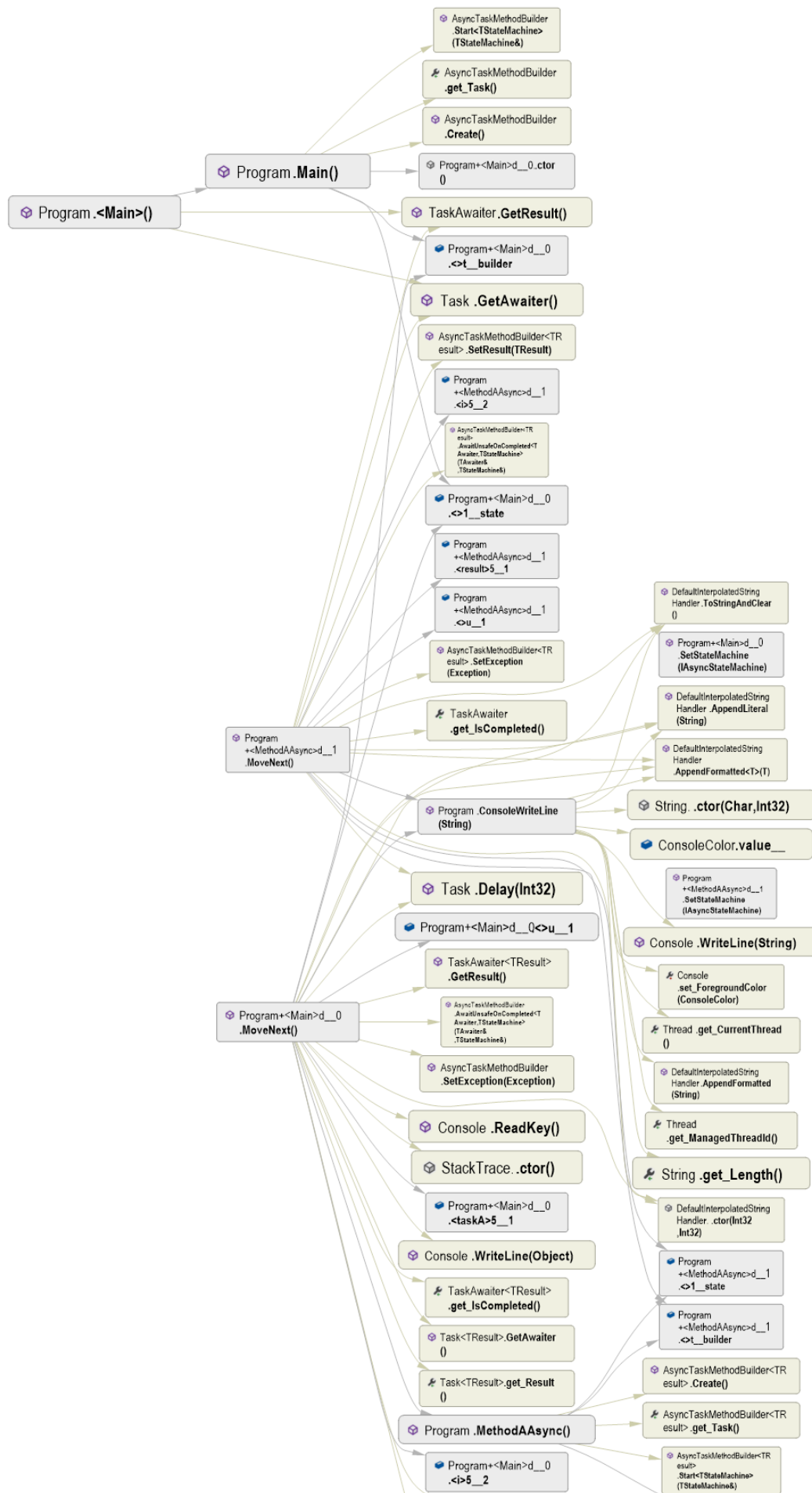
Here it is:

```
Start Program          Thread 1
A0                    Thread 1
B0                    Thread 1
B1                    Thread 1
A1                    Thread 4
B2                    Thread 1
B3                    Thread 1
A2                    Thread 4
B4                    Thread 1
Wait for taskA termination Thread 1
A3                    Thread 4
A4                    Thread 4
A returns result 123   Thread 4
  at Program.Main()
  at System.Runtime.CompilerServices.AsyncTaskMethodBuilder`1.AsyncStateMachineBox`1.ExecutionContextCallback(Object s)
  at System.Threading.ExecutionContext.RunInternal(ExecutionContext executionContext, ContextCallback callback, Object state)
  at System.Runtime.CompilerServices.AsyncTaskMethodBuilder`1.AsyncStateMachineBox`1.MoveNext(Thread threadPoolThread)
  at System.Runtime.CompilerServices.AsyncTaskMethodBuilder`1.AsyncStateMachineBox`1.MoveNext()
  at System.Threading.Tasks.AwaitTaskContinuation.RunOrScheduleAction(IAsyncStateMachineBox box, Boolean allowInlining)
  at System.Threading.Tasks.Task.RunContinuations(Object continuationObject)
  at System.Threading.Tasks.Task.FinishContinuations()
  at System.Threading.Tasks.Task`1.TrySetResult(TResult result)
  at System.Runtime.CompilerServices.AsyncTaskMethodBuilder`1.SetExistingTaskResult(Task`1 task, TResult result)
  at System.Runtime.CompilerServices.AsyncTaskMethodBuilder`1.SetResult(TResult result)
  at Program.MethodAAsync()
  at System.Runtime.CompilerServices.AsyncTaskMethodBuilder`1.AsyncStateMachineBox`1.ExecutionContextCallback(Object s)
  at System.Threading.ExecutionContext.RunInternal(ExecutionContext executionContext, ContextCallback callback, Object state)
```

The simple line `await taskA;` leads the C# compiler to generate a lot of code to pilot the runtime. Identifiers like `AsyncState...` and `MoveNext()` shows that a state machine is created for us to let the magic of code continuation happens seamlessly. Here is the assembly decompiled with ILSpy. We can see that a class is generated by the compiler for each usage of the await keyword:



Here is a [call graph generated by NDepend](#) of the methods of the [Task Parallel Library \(TPL\)](#) called by the generated code. To obtain such graph with methods and fields generated by the compiler, the following setting must be disabled first: *NDepend > Project Properties > Analysis > Merge Code Generated by Compiler into Application Code*

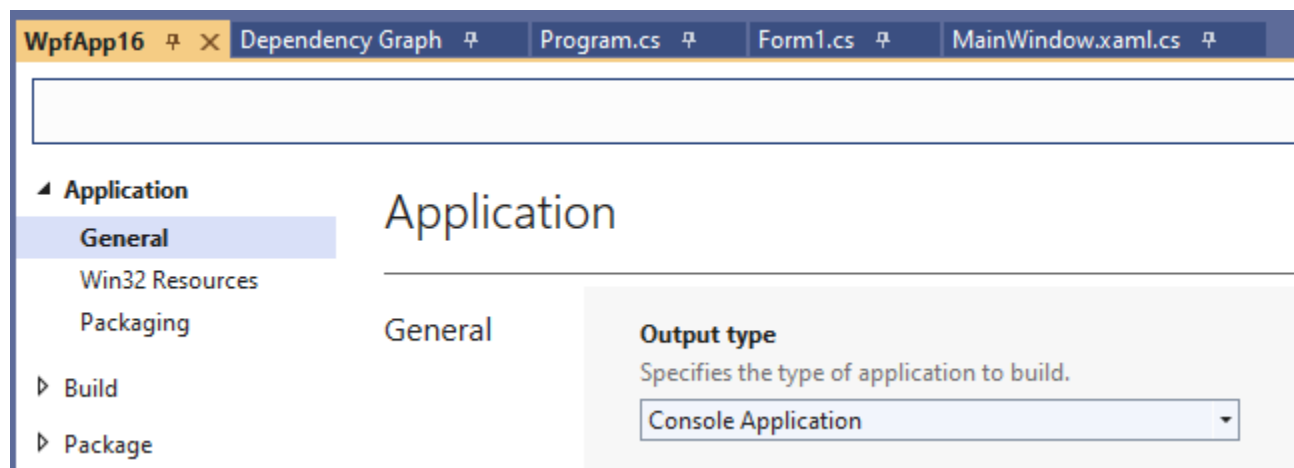




The details of what the C# compiler generates when it meets the keyword `await` is outside the scope of this article [but you can deep dive in it in this Microsoft article](#). Just keep in mind that the code executed after an `await` keyword can eventually be executed by a *random thread* and that a lot of code that calls the TPL is generated to make this happen. Let's explain how the *random thread* is chosen by the runtime.

The SynchronizationContext

So far we only demonstrated code executed in the context of a console application. **The context in which some asynchronous code runs actually influences its workflow a lot.** For example let's run the same code in the context of a WPF application. Since it is convenient to keep the console output to show results of our experiments, let's set the output type of our WPF assembly to *Console Application*, so a console is shown when the WPF app starts.




Now let's execute the exact same code from within a WPF button click event handler:

```

C#
1 public partial class MainWindow : Window {
2     public MainWindow() {
3         InitializeComponent();
4     }
5     private async void Button_Click(object sender, RoutedEventArgs e) {
6         Console.WriteLine($"Start Program");
7         Task<int> taskA = MethodAAsync();
8         ...
  
```

Here is the surprising result: the main thread is used to run everything! And task A loops are postponed after task B loops (except the first one).

Start Program	Thread 1
A0	Thread 1
B0	Thread 1
B1	Thread 1
B2	Thread 1
B3	Thread 1
B4	Thread 1
Wait for taskA termination	Thread 1
A1	Thread 1
A2	Thread 1
A3	Thread 1
A4	Thread 1
A returns result 123	Thread 1
The result of taskA is 123	Thread 1



This is totally different than what we had with our console application. The key is that in a WPF context (and also in a Winforms context) there is a synchronization context object, that can be obtained through `SynchronizationContext.Current`.

```
private async void Button_Click(object sender, RoutedEventArgs e) {
    var sc:SynchronizationContext? = SynchronizationContext.Current;
    Console.WriteLine($"Start Program");
}
```

Debugger state: `sc` is of type `{System.Windows.Threading.DispatcherSynchronizationContext}`.

There is no synchronization context in a console application.

0 references

```
static async Task Main() {
    var sc:SynchronizationContext? = SynchronizationContext.Current;
    Console.WriteLine($"Start Program");
}
```

Debugger state: `sc` is `null`.

The WPF and Winforms `SynchronizationContext` behavior

In the precedent WPF execution there is no pool thread involved because there is no real asynchronous processing: remember we use `await Task.Delay(100);` to simulate it. Here is the output if we do some real processing instead:

```

1 reference
static async Task<int> MethodAAsync() {
    for (int i = 0; i < 5; i++) {
        Console.WriteLine($" A{i}");
        //await Task.Delay(100)
        await Task.Run(() => {
            Console.WriteLine($" A{i} real processing");
        });
    }
}

```

The screenshot shows a Visual Studio IDE with a console window and a WPF application window. The console window displays the output of the C# code, showing the execution of MethodAAsync. The WPF application window shows a button labeled 'Button'.

Output	Thread
Start Program	Thread 1
A0	Thread 1
A0 real processing	Thread 4
B0	Thread 1
B1	Thread 1
B2	Thread 1
B3	Thread 1
B4	Thread 1
Wait for taskA termination	Thread 1
A1	Thread 1
A1 real processing	Thread 4
A2	Thread 1
A2 real processing	Thread 4
A3	Thread 1
A3 real processing	Thread 4
A4	Thread 1
A4 real processing	Thread 4
A returns result 123	Thread 1
The result of taskA is 123	Thread 1

Why do we need a SynchronizationContext in WPF and Winforms scenarios? : In WPF there is a main UI thread that manages the UI (and a hidden thread that does the rendering) and in Winforms there is also a UI thread that does both the managing of controls and the rendering. When the UI thread gets too busy, the UI becomes unresponsive and the user gets nervous. This is why in both cases it is essential to run computation intensive task on a pool thread and not on the UI thread. This is why both WPF and Winforms have their own synchronization contexts, in order to resume by default on the UI thread to harness the result of an asynchronous operations that just terminated. Typically the result is used to refresh some controls. To do so, these synchronization contexts are relying on the internal infrastructure of the WPF and the Winforms platforms.

What is the runtime workflow in both WPF examples above? : In both WPF results above, we can see that first A0 is displayed and then task B is ran entirely (B0 ... B4) until task A can resume with (A1 ... A4). Remember that in task B we have `Task.Delay(50).Wait();` that first simulates a task and then wait for its termination. This is a blocking call equivalent to `Thread.Sleep(50);` unlike `await Task.Delay(100)` in task A that is not blocking. This means that the UI thread is kept busy with the task B until it finishes. Only upon task B termination, the UI thread gets available again and the WPF synchronization context can resume task A on it.

[Disabling the WPF and Winforms SynchronizationContext behavior with `task.ConfigureAwait\(false\)`](#)

This WPF and Winforms asynchronous contexts' default behavior of resuming on the main UI

thread after an asynchronous call can be discarded by calling the method `ConfigureAwait(false)` on the task in the await call. The value `false` is set to the parameter `ConfigureAwait(bool continueOnCapturedContext)`. By default this well-named parameter is set to `true`. With `ConfigureAwait(false)` called in a WPF or Winforms context, we go back to the console behavior where a random thread from the pool is used to resume after the await call. In a UI application you might wish to avoid preempting the UI thread when harnessing the result of an asynchronous operation to preserve the UI responsiveness that is conditioned by the amount of work done on the UI thread. Of course this only makes sense if the UI is not refreshed from the result.

```

1 reference
static async Task<int> MethodAAsync() {
    for (int i = 0; i < 5; i++) {
        Console.WriteLine($"A{i}");
        await Task.Delay(100).ConfigureAwait(false);
    }
}

```

C:\Users\pat\source\repos\WpfApp16\bin\Debug\net6.0-windows\WpfApp16.exe

Operation	Thread
Start Program	Thread 1
A0	Thread 1
B0	Thread 1
B1	Thread 1
A1	Thread 4
B2	Thread 1
B3	Thread 1
A2	Thread 4
B4	Thread 1
Wait for taskA termination	Thread 1
A3	Thread 7
A4	Thread 7
A returns result 123	Thread 7
The result of taskA is 123	Thread 1

In the execution result above, only the await usage in the `MethodAAsync()` method is done with `ConfigureAwait(false)`, not the await usage in the `Button_Click()` method. This is why the main thread is used to print "The result of taskA is 123", because of the WPF synchronization context behavior still enabled here.

No SynchronizationContext in ASP.NET Core

Let's notice that there is no synchronization context within an ASP.NET Core application. This was an important change because ASP.NET had an `AspNetSynchronizationContext` [as discussed in this stackoverflow Q/A](#). On [his blog](#), [Stephen Cleary](#) explains that the decision to discard `AspNetSynchronizationContext` was taken to obtain more simplicity and performance.

Finally let's note that you can create [custom synchronization context](#) as explained on this [github page](#), although you won't likely do so.

The Task Parallel Library (TPL)

In the precedent sections we mentioned the TPL. The TPL is an extensive library proposing everything one could need to address any asynchronous scenario, from the basic to the most

advanced ones. The classes `Task<TResult>` and `Task` are the central classes of the TPL. In the example below we start several tasks and wait for their terminations, one by one, with the TPL method `Task.WhenAny<TResult>(IEnumerable<Task<TResult>>)`:

```
C#
1 class Program {
2     static async Task Main(string[] args) {
3         Console.WriteLine($"Start Program");
4
5         var tasks = new List<Task<string>> {
6             MethodAsync("A", 50),
7             MethodAsync("B", 100),
8             MethodAsync("C", 20)
9         };
10
11         while (tasks.Any()) {
12             Task<string> taskTerminated = await Task.WhenAny(tasks);
13             Console.WriteLine($"Task terminated result {taskTerminated.Result}");
14             tasks.Remove(taskTerminated);
15         }
16
17         Console.WriteLine($"End Program");
18         Console.ReadKey();
19     }
20
21     static async Task<string> MethodAsync(string x, int delay) {
22         for (int i = 0; i < 3; i++) {
23             Console.WriteLine($" {x}{i}");
24             await Task.Delay(delay);
25         }
26         string result = new string(x[0], 4);
27         Console.WriteLine($" {x} returns result {result}");
28         return result;
29     }
30
31     // Convenient helper to print colorful threadId on console
32     static void Console.WriteLine(string str) {
33         int threadId = Thread.CurrentThread.ManagedThreadId;
34         Console.ForegroundColor = threadId == 1 ? ConsoleColor.White : ConsoleColor.Cyan;
35         Console.WriteLine(
36             $"{str}{new string(' ', 29 - str.Length)} Thread {threadId}");
37     }
38 }
```

Here is the result:

```

Start Program                                Thread 1
A0                                           Thread 1
B0                                           Thread 1
C0                                           Thread 1
C1                                           Thread 7
A1                                           Thread 4
C2                                           Thread 7
C returns result CCCC                       Thread 7
Task terminated result CCCC                 Thread 7
B1                                           Thread 7
A2                                           Thread 7
A returns result AAAA                       Thread 7
Task terminated result AAAA                 Thread 7
B2                                           Thread 7
B returns result BBBB                       Thread 7
Task terminated result BBBB                 Thread 7
End Program                                Thread 7

```

await and Exception Handling

Let's underline that the keyword `await` works as expected when an exception is thrown from an asynchronous processing.

```

C#
1  static async Task Main(string[] args) {
2      Console.WriteLine($"Start Program");
3      ...
4
5      Console.WriteLine("Wait for taskA termination");
6      try {
7          await taskA;
8          Console.WriteLine($"The result of taskA is {taskA.Result}");
9      }
10     catch (ApplicationException ex) {
11         Console.WriteLine($"{ex.GetType().ToString()} Msg:{ex.Message}");
12     }
13     Console.ReadKey();
14 }
15
16 static async Task<int> MethodAAsync() {
17     for (int i = 0; i < 5; i++) {
18         Console.WriteLine($" A{i}");
19         await Task.Delay(100);
20         Console.WriteLine($" A throws exception");
21         throw new ApplicationException("Boum");
22     }
23     int result = 123;
24     Console.WriteLine($" A returns result {result}");
25     return result;
26 }
27 ...

```

Here is the output of this program:

```

Start Program                                Thread 1
A0                                           Thread 1
B0                                           Thread 1
B1                                           Thread 1
A throws exception                          Thread 4
B2                                           Thread 1
B3                                           Thread 1
B4                                           Thread 1
Wait for taskA termination                  Thread 1
System.ApplicationException Msg:Boum        Thread 1

```

On the other hand if the line `await taskA;` within the `try { ... }` catch scope is replaced with the line `taskA.Wait();`, the exception is not handled by the catch clause. This unexpected behavior illustrates well that when doing asynchronous programming, the keyword `await` should be the preferred way to await asynchronous methods.

The plethora of asynchronous .NET APIs

The introduction explained that `async` and `await` keywords make asynchronous programming easier, especially when it comes to run computationally expensive tasks executed simultaneously on multiple CPU cores. The keywords `async` and `await` are also especially useful to run asynchronous I/O tasks. The .NET library offers hundreds of asynchronous methods to achieve all sorts of I/O tasks including network access, database access, JSON XML binary... file access, data compression and more.

Here is a small example where we gather 3 website home pages in order to print their sizes in bytes:

```

C#
1 var tasks = new Task<string>[] {
2     new HttpClient().GetStringAsync("https://www.google.com/"),
3     new HttpClient().GetStringAsync("https://www.microsoft.com/"),
4     new HttpClient().GetStringAsync("https://www.ndepend.com/")
5 };
6
7 await Task.WhenAll(tasks);
8
9 // Print the size of the webpages
10 Console.WriteLine(
11     $"Home page sizes: {tasks.Select(t => t.Result.Length.ToString()).Aggregate((str1,str2) =
12     Console.ReadKey());

```

This program prints:

```

C#
1 Home page sizes: 52891,193871,39755

```

Notice that this program relies on C#9 top level statement that works fine with the `await` keyword. It would be easy to modify this programs for example to read asynchronously some files content.

```

C#
1 var tasks = new Task<string>[] {
2     File.ReadAllTextAsync(@"C:\Program Files\dotnet\dotnet.exe"),
3     File.ReadAllTextAsync(@"C:\Windows\explorer.exe"),
4     File.ReadAllTextAsync(@"C:\Windows\py.exe"),

```

```
5 };  
6 ...
```

Conclusion

Hopefully now the asynchronous control flow obtained through the keyword `await` is less mysterious to you. The `await` keyword leads to a lot of code generated by the C# compiler while the `async` keyword just decorates asynchronous method but doesn't lead to anything tricky as `await` does.

In this article we only focused on the C# `async` and `await` keywords and things that can influence their behavior like the synchronization context or exception. If you need to implement more advanced asynchronous scenarios – like cancelling a task for example – it is time to learn more about the TPL.

Patrick Smacchia

Website

My dad being an early programmer in the 70's, I have been fortunate to switch from playing with Lego, to program my own micro-games, when I was still a kid. Since then I never stop programming.



I graduated in Mathematics and Software engineering. After a decade of C++ programming and consultancy, I got interested in the brand new .NET platform in 2002. I had the chance to write the best-seller book (in French) on .NET and C#, published by O'Reilly and also did manage some academic and professional courses on the platform and C#.

Over my consulting years I built an expertise about the architecture, the evolution and the maintenance challenges of large & complex real-world applications. It seemed like the spaghetti & entangled monolithic legacy concerned every sufficiently large team. As a consequence, I got interested in static code analysis and started the project NDepend in 2004.

Nowadays NDepend is a full-fledged Independent Software Vendor (ISV). With more than 12.000 client companies, including many of the Fortune 500 ones, NDepend offers deeper insight and full control on their application to a wide range of professional users around the world.

I live with my wife and our twin kids Léna and Paul in the beautiful island of Mauritius in the Indian Ocean.

Make your .NET code beautiful with NDepend

Measure quality with metrics, generate diagrams and enforce decisions with