

# Making interface implementations async

Asked 11 years, 11 months ago   Modified 3 years, 5 months ago   Viewed 199k times



183

I'm currently trying to make my application using some Async methods. All my IO is done through explicit implementations of an interface and I am a bit confused about how to make the operations async.



As I see things I have two options in the implementation:



```
interface IIIO
{
    void DoOperation();
}
```

**OPTION1:** Do an implicit implementation async and await the result in the implicit implementation.

```
class IOImplementation : IIIO
{
    async void DoOperation()
    {
        await Task.Factory.StartNew(() =>
        {
            //WRITING A FILE OR SOME SUCH THINGAMAGIG
        });
    }

    #region IIIO Members

    void IIIO.DoOperation()
    {
        DoOperation();
    }

    #endregion
}
```

**OPTION2:** Do the explicit implementation async and await the task from the implicit implementation.

```
class IOAsyncImplementation : IIIO
{
    private Task DoOperationAsync()
    {
        return new Task(() =>
        {
            //DO ALL THE HEAVY LIFTING!!!
        });
    }
}
```

```

    }

    #region IIIOAsync Members

    async void IIIO.DoOperation()
    {
        await DoOperationAsync();
    }

    #endregion
}

```

Are one of these implementations better than the other or is there another way to go that I am not thinking of?

c# asynchronous async-await

Share Improve this question Follow

asked Jan 22, 2013 at 12:27



Moriya

7,896 ● 3 ● 37 ● 54

## 4 Answers

Sorted by: Highest score (default) ◆



329



Neither of these options is correct. You're trying to implement a synchronous interface asynchronously. Don't do that. The problem is that when `DoOperation()` returns, the operation won't be complete yet. Worse, if an exception happens during the operation (which is very common with IO operations), the user won't have a chance to deal with that exception.



What you need to do is to *modify the interface*, so that it is asynchronous:



```

interface IIIO
{
    Task DoOperationAsync(); // note: no async here
}

class IOImplementation : IIIO
{
    public async Task DoOperationAsync()
    {
        // perform the operation here
    }
}

```

This way, the user will see that the operation is `async` and they will be able to `await` it. This also pretty much forces the users of your code to switch to `async`, but that's unavoidable.

Also, I assume using `StartNew()` in your implementation is just an example, you shouldn't need that to implement asynchronous IO. (And `new Task()` is even worse, that won't even work, because you don't `start()` the `Task`.)

Share

edited Jan 24, 2013 at 18:34

answered Jan 22, 2013 at 13:00

Improve this answer



svick

244k ● 53 ● 403 ● 526

Follow

- 
- 2 How would this look with an explicit implementation? Also, Where do you await this implementation? – [Moriya](#) Jan 22, 2013 at 13:17
- 
- 2 @Animal Explicit implementation would look the same as always (just add `async`): `async Task II0.DoOperationAsync()`. And do you mean where do you `await` the returned `Task`? Wherever you call `DoOperationAsync()`. – [svick](#) Jan 22, 2013 at 13:19 ✎
- 
- 1 Basically I think I can condense my question into "Where do I await?" If I don't await inside the `async` method I get compilation warnings. – [Moriya](#) Jan 22, 2013 at 13:21
- 
- 1 Ideally, you shouldn't need to wrap the IO code in `Task.Run()`, that IO code should be asynchronous itself and you would `await` that directly. E.g. `line = await streamReader.ReadLineAsync()`. – [svick](#) Jan 22, 2013 at 13:58 ✎
- 
- 4 Then there is not much point in making your code `async`. See the article [Should I expose asynchronous wrappers for synchronous methods?](#) – [svick](#) Jan 22, 2013 at 14:05 ✎
- 

▲ Better solution is to introduce another interface for `async` operations. New interface must inherit from original interface.

33

▼ Example:

```
interface II0
{
    void DoOperation();
}

interface II0Async : II0
{
    Task DoOperationAsync();
}

class ClsAsync : II0Async
{
    public void DoOperation()
    {
        DoOperationAsync().GetAwaiter().GetResult();
    }

    public async Task DoOperationAsync()
    {
        //just an async code demo
    }
}
```

```

        await Task.Delay(1000);
    }
}

class Program
{
    static void Main(string[] args)
    {
        IIIOAsync asAsync = new ClsAsync();
        IIIO asSync = asAsync;

        Console.WriteLine(DateTime.Now.Second);

        asAsync.DoOperation();
        Console.WriteLine("After call to sync func using Async iface: {0}",
            DateTime.Now.Second);

        asAsync.DoOperationAsync().GetAwaiter().GetResult();
        Console.WriteLine("After call to async func using Async iface: {0}",
            DateTime.Now.Second);

        asSync.DoOperation();
        Console.WriteLine("After call to sync func using Sync iface: {0}",
            DateTime.Now.Second);

        Console.ReadKey(true);
    }
}

```

P.S. Redesign your async operations so they return Task instead of void, unless you really must return void.

Share

edited Oct 23, 2017 at 16:51

answered Apr 5, 2017 at 16:07

Improve this answer



Dima

1,761 ● 15 ● 34

Follow

- 5 Why not `GetAwaiter().GetResult()` instead of `wait()` ? That way you don't need to unpack an `AggregateException` to fetch the inner exception. – Tagc Oct 22, 2017 at 22:58
- 1 A variation is rely on the class implementing multiple (possibly explicit) interfaces: `class Impl : IIIO, IIIOAsync`. `IIIO` and `IIIOAsync` themselves, however, are different contracts which can avoid pulling in 'old contracts' into newer code. `var c = new Impl(); IIIOAsync asAsync = c; IIIO asSync = c`. – user2864740 Jun 19, 2019 at 23:27 ✎
- 1 Bad idea. If your operations are async, expose them as async, and let the consumer handle them as desired. The consumer can decide to "convert to sync" as your're doing in your example, or to keep being async. In fact, I'd recommend to also include always the Cancellation token, and use it, for example to time out the IO operations that are not done timely. Mosf of the async implementations in the .NET framework include it. – JotaBe Oct 7, 2021 at 11:36
- 3 You shouldn't use the `GetAwaiter` and `GetResult()` in your code. It's meant for the compiler only: [TaskAwaiter Struct](#) – Roeland Van Heddegem Oct 27, 2021 at 15:53



2



I created a sample app based on Svick's answer and found that calling `IImplementation.DoOperationAsync()` without the `async` keyword does not result in a compiler/Visual Studio warning. This was based on Visual Studio 2019 and .NET Core 3.1.

Sample code below.

```
public interface ISomething
{
    Task DoSomethingAsync();
}
```

```
public class Something : ISomething
{
    public async Task DoSomethingAsync()
    {
        await Task.Run(() => Thread.Sleep(2000));
        Console.WriteLine("Message from DoSomethingAsync");

        throw new Exception("Some exception");
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        ISomething something = new Something();

        Console.WriteLine("pre something.DoSomethingAsync() without await");
        something.DoSomethingAsync(); // No compiler warning for missing
"await" and exception is "swallowed"
        Console.WriteLine("post something.DoSomethingAsync() without await");

        Thread.Sleep(3000);

        // Output:
        // pre something.DoSomethingAsync() without await
        // post something.DoSomethingAsync() without await
        // Message from DoSomethingAsync
    }
}
```

Share Improve this answer Follow

answered Jul 9, 2021 at 21:01



datchung

4,572 ● 1 ● 36 ● 31

I believe that compiler have to warning about missing of "await" if your Main method is async. BTW, it's better to use Task.Delay instead of Thread.Sleep to implement waiting in async methods. – [valker](#) Feb 16, 2022 at 20:10

An abstract class can be used instead of an interface (in C# 7.3).

-5

```
// Like interface
abstract class IIO
{
    public virtual async Task<string> DoOperation(string Name)
    {
        throw new NotImplementedException(); // throwing exception
        // return await Task.Run(() => { return ""; }); // or empty do
    }
}

// Implementation
class IOImplementation : IIO
{
    public override async Task<string> DoOperation(string Name)
    {
        return await await Task.Run(() =>
        {
            if(Name == "Spiderman")
                return "ok";
            return "cancel";
        });
    }
}
```

Share

edited Aug 19, 2020 at 0:33

answered Aug 18, 2020 at 23:56

Improve this answer



[Alatey](#)

379 ● 2 ● 6

Follow

- 1 What is the benefit of doing this? Why does `IIO.DoOperation()` have a dummy implementation instead of being `abstract` itself? – [Lance U. Matthews](#) Aug 19, 2020 at 0:10

In C # 7.3, I cannot use an "abstract async" method with no body. – [Alatey](#) Aug 19, 2020 at 0:33

There is zero use in doing it this way. And there's no point in making the class `abstract` if there are no `abstract` members. The `DoOperation()` method could be made `abstract`, but then you've got something that is essentially just an interface. Which brings one back to, why not just use an interface? The code above is pointless. :( – [Peter Duniho](#) Aug 19, 2020 at 0:35

- 3 "I cannot use an "abstract async" method with no body" -- there is no reason to want to use `abstract async`. The only thing that adding `async` to the method declaration does, is to allow you to use `await` in the method. If your method doesn't use `await`, then you don't need `async`. That's why the interface approach works in the first place; **what's actually**

**important is the return type.** You can make the return type `Task<string>` without the `async` keyword. – [Peter Duniho](#) Aug 19, 2020 at 0:37

---