CSharp Concepts III

**Two interfaces with same method name**

C# – How to implement two interfaces having same method names in a class

[DEEPAK SHARMA](http://deepak-sharma.net/author/deepak/)    February 8, 2015    [No Commentson C# – How to implement two interfaces having same method names in a class](http://deepak-sharma.net/2015/02/08/c-how-to-implement-two-interfaces-having-same-method-names-in-a-class/#respond)

We have two interfaces IMan and IBirds. Both the interfaces defines a method named Eat with same name and signature. A class MyClass is inheriting both the interfaces. How can we implement method of both the interfaces in the derived class MyClass ?

Here is the code:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public interface IMan  {  void Eat();  }    public interface IBirds  {  void Eat();  }    public class MyClass : IMan, IBirds  {  void IMan.Eat()  {  Console.WriteLine("Roti");  }  void IBirds.Eat()  {  Console.WriteLine("Earthworms");  }  } |

We use interface name to access its method to implement.

Similarly, we call a method by instantiating the desired interface and assigning object of the derived class to it.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | IMan man = new MyClass();  man.Eat();  IBirds birds = new MyClass();  birds.Eat();    //OR    ((IMan)new MyClass()).Eat();  ((IBirds)new MyClass()).Eat(); |

# Solution of problem having same method name in two interfaces inheriting a class

SheoNarayan

Posted by [Sheo Narayan](http://www.dotnetfunda.com/profile/sheonarayan.aspx) in [C#](http://www.dotnetfunda.com/articles/cat/6/csharp) category on 9/7/2008 for Advance level | Views : 29873 [red flag](javascript:ShowHide('dvFlg'))

Rating: 5 out of 5

 1 vote(s)

[[http://www.dotnetfunda.com/images/dnfnewpost.gif](http://www.dotnetfunda.com/articles/postarticle.aspx) Post Article](http://www.dotnetfunda.com/articles/postarticle.aspx) | [[http://www.dotnetfunda.com/images/dnfsearch.gif](http://www.dotnetfunda.com/articles/search.aspx) Search Articles](http://www.dotnetfunda.com/articles/search.aspx) | [[http://www.dotnetfunda.com/images/dnfarticle.gif](http://www.dotnetfunda.com/articles/) Articles Home](http://www.dotnetfunda.com/articles/) [[http://www.feedburner.com/fb/images/pub/feed-icon16x16.png](http://feeds2.feedburner.com/DotNetFundaArticles)](http://feeds2.feedburner.com/DotNetFundaArticles)

This articles describes the solution of having the same method name in two different interfaces inheriting a class.

###### Introduction

I think this question is very favourite to the interviewer these days. I have got many questions from visitors asking the solution of implementing the method and calling it if a class inherits two different interfaces and both interfaces have same method name and signature. This articles describes various scenario related with that and gives the solution.

###### Defining interfaces

To descibe this scenario, I have created a class called ExampleClass that is inheriting two interface IExampleClass1 and IExampleClass2. Both interfaces have same method name and Signature CurrentDateTime(). See the code snippet for them below.

First Interface: **IExampleClass1**

public interface IExampleClass1

{

string CurrentDateTime();

}

Second Interface: **IExampleClass2**

using System;

public interface IExampleClass2

{

string CurrentDateTime();

string CurrentDateTime(DateTime thisDateTime);

}

###### Implementing Methods

If we inherit these two interfaces into the same class then we will have to implement all the methods defined in these interfaces. Lets see the ways of implementing interface into our class. There are two ways we can implement interfaces into a class. Even when you inherit the interface and right click the interface name in the VWD or Visual Studio, you will get two options to implement interfaces.

public class ExampleClass : IExampleClass1, IExampleClass2

1. Implement Interface - When we choose this option of implementing interface, we need to define methods name that interface has and need to implement those methods. By default these methods modifiers will be public. This is also called Implicitely implementation of the interface.

#region IExampleClass2 Members

public string CurrentDateTime()

{

return "Interface 2 : " + DateTime.Now.ToString();

}

public string CurrentDateTime(DateTime thisDateTime)

{

return "Interface 2 : " + thisDateTime.ToLongDateString();

}

#endregion

2. Implement Interface Explicitely - When you choose this option, you need to define method names that interface has by prefixing the interface name. Notice that there is no access modifier for this method implementation so by default its scope will be **internal**. Internal is the access modifier for the method or class that is only accessible from the class in same assembly.

#region IExampleClass1 Members

string IExampleClass1.CurrentDateTime()

{

return "Interface 1 : " + DateTime.Now.ToString();

}

#endregion

If you try to change its access modifier to public in this scenario, you will get "CS0106: The modifier 'public' is not valid for this item" error while running your application. Because there is a public method with the same name and signature already implemented (for IExampleClass2 interface) in the class.

###### Using Methods

Lets see how to use methods of both interfaces implemented in the class.

ExampleClass ex = new ExampleClass();

lblMessage1.Text = ex.CurrentDateTime();

lblMessage1.Text += "<br />" + ex.CurrentDateTime(DateTime.Now);

IExampleClass1 ex1 = new ExampleClass();

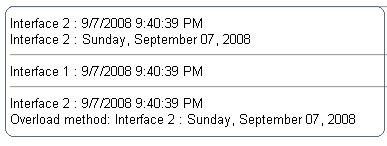
lblMessage2.Text = ex1.CurrentDateTime();

IExampleClass2 ex2 = new ExampleClass();

lblMessage3.Text = ex2.CurrentDateTime();

lblMessage3.Text += "<br />Overload method: " + ex2.CurrentDateTime(DateTime.Now);

**Output**



In the above code snippet, you can see that I have instantiated ExampleClass in three different ways. Lets see them one by one.

1. **ExampleClass** ex = new ExampleClass(); - When we are instantiating the class by specifying the object as the class name, we will be able to call methods for the interface that is implicitely implemented. In this case IExampleClass2.

2. **IExampleClass1** ex1 = new ExampleClass(); - If we want to call the method that is explicitely implemented, we need to specify the object as the interface name and instantiate the class then we will be able to call the method of this interface. In this case IExampleClass1.

3. **IExampleClass2**ex2 = new ExampleClass(); - You can call methods of the interface that is implicitely implemented by specifying the name of the Interface as the object name (The way we have done in the 2nd point). This will give the same result as the 1st point above.

<http://www.dotnetfunda.com/articles/show/133/solution-of-problem-having-same-method-name-in-two-interfaces-inheriti>

**== and .equals**

using System;

public class Test

{

static void Main()

{

// Create two equal but distinct strings

string a = new string(new char[] {'h', 'e', 'l', 'l', 'o'});

string b = new string(new char[] {'h', 'e', 'l', 'l', 'o'});

Console.WriteLine (a==b);

Console.WriteLine (a.Equals(b));

// Now let's see what happens with the same tests but

// with variables of type object

object c = a;

object d = b;

Console.WriteLine (c==d);

Console.WriteLine (c.Equals(d));

}

}

The results are:

True

True

False

True

The third line is False because the compiler can only call the non-overloaded version of ==  
as it doesn’t know that the contents of c and d are both string references. As they  
are references to different strings, the identity operator returns false.

So, when should you use which operator? My rule of thumb is that for almost all reference types, use Equals  
when you want to test equality rather than reference identity. The exception is for strings – comparing strings with  
== does make things an awful lot simpler and more readable *but* you need to remember that both  
sides of the operator must be expressions of type string in order to get the comparison to work properly.

For value types, I’d normally use == for easier-to-read code. Things would get tricky if a value type  
provided an overload for == which acted differently to Equals, but I’d consider such a type  
very badly designed to start with.

<https://blogs.msdn.microsoft.com/csharpfaq/2004/03/29/when-should-i-use-and-when-should-i-use-equals/>

**Private constructor - when we use - how we can use - can instantiate?**

Private constructors can be useful when using a factory pattern (in other words, a static function that's used to obtain an instance of the class rather than explicit instantiation).

public class MyClass

{

private static Dictionary<object, MyClass> cache =

new Dictionary<object, MyClass();

private MyClass() { }

public static MyClass GetInstance(object data)

{

MyClass output;

if(!cache.TryGetValue(data, out output))

cache.Add(data, output = new MyClass());

return output;

}

}

# [Why do we need the async keyword?](https://softwareengineering.stackexchange.com/questions/187492/why-do-we-need-the-async-keyword)

There are several answers here, and all of them talk about what async methods do, but none of them answer the question, which is why async is needed as a keyword that goes in the function declaration.

It's not "to direct the compiler to transform the function in a special way"; await alone could do that. Why? Because C# already has another mechanism where the presence of a special keyword in the method body causes the compiler to perform extreme (and very similar to async/await) transformations on the method body: yield.

Except that yield isn't its own keyword in C#, and understanding why will explain async as well. Unlike in most languages that support this mechanism, in C# you can't say yield value; You have to say yield return value; instead. Why? Because it was added in to the language after C# already existed, and it was quite reasonable to assume that someone, somewhere, might have used yield as the name of a variable. But because there was no pre-existing scenario in which <variable name> return was syntactically correct, yield return got added to the language to make it possible to introduce generators while maintaining 100% backwards compatibility with existing code.

And this is why async was added as a function modifier: to avoid breaking existing code that used await as a variable name. Since no async methods already existed, no old code is invalidated, and in new code, the compiler can use the presence of the async tag to know that await should be treated as a keyword and not an identifier.

# [How and When to use `async` and `await`](https://stackoverflow.com/questions/14455293/how-and-when-to-use-async-and-await)

When using async and await the compiler generates a state machine in the background.

Here's an example on which I hope I can explain some of the high-level details that are going on:

public async Task MyMethodAsync()

{

Task<int> longRunningTask = LongRunningOperationAsync();

// independent work which doesn't need the result of LongRunningOperationAsync can be done here

//and now we call await on the task

int result = await longRunningTask;

//use the result

Console.WriteLine(result);

}

public async Task<int> LongRunningOperationAsync() // assume we return an int from this long running operation

{

await Task.Delay(1000); //1 seconds delay

return 1;

}

OK, so what happens here:

1. Task<int> longRunningTask = LongRunningOperationAsync(); starts executing LongRunningOperation
2. Independent work is done on let's assume the Main Thread (Thread ID = 1) then await longRunningTask is reached.

Now, if the longRunningTask hasn't finished and it is still running, MyMethodAsync() will return to its calling method, thus the main thread doesn't get blocked. When the longRunningTask is done then a thread from the ThreadPool (can be any thread) will return to MyMethodAsync() in its previous context and continue execution (in this case printing the result to the console).

A second case would be that the longRunningTask has already finished its execution and the result is available. When reaching the await longRunningTask we already have the result so the code will continue executing on the very same thread. (in this case printing result to console). Of course this is not the case for the above example, where there's a Task.Delay(1000) involved.

# Asynchronous Programming with Async and Await (C# and Visual Basic)

**Visual Studio 2012**

[Other Versions](javascript:void(0))

https://i-msdn.sec.s-msft.com/Areas/Epx/Content/Images/ImageSprite.png?v=636311636105440369

You can avoid performance bottlenecks and enhance the overall responsiveness of your application by using asynchronous programming. However, traditional techniques for writing asynchronous applications can be complicated, making them difficult to write, debug, and maintain.

Visual Studio 2012 introduces a simplified approach, async programming, that leverages asynchronous support in the .NET Framework 4.5 and the Windows Runtime. The compiler does the difficult work that the developer used to do, and your application retains a logical structure that resembles synchronous code. As a result, you get all the advantages of asynchronous programming with a fraction of the effort.

This topic contains the following sections.

* [Async Improves Responsiveness](https://msdn.microsoft.com/library/hh191443(vs.110).aspx#BKMK_WhentoUseAsynchrony)
* [Async Methods Are Easier to Write](https://msdn.microsoft.com/library/hh191443(vs.110).aspx#BKMK_HowtoWriteanAsyncMethod)
* [What Happens in an Async Method](https://msdn.microsoft.com/library/hh191443(vs.110).aspx#BKMK_WhatHappensUnderstandinganAsyncMethod)
* [API Async Methods](https://msdn.microsoft.com/library/hh191443(vs.110).aspx#BKMK_APIAsyncMethods)
* [Threads](https://msdn.microsoft.com/library/hh191443(vs.110).aspx#BKMK_Threads)
* [Async and Await](https://msdn.microsoft.com/library/hh191443(vs.110).aspx#BKMK_AsyncandAwait)
* [Return Types and Parameters](https://msdn.microsoft.com/library/hh191443(vs.110).aspx#BKMK_ReturnTypesandParameters)
* [Naming Convention](https://msdn.microsoft.com/library/hh191443(vs.110).aspx#BKMK_NamingConvention)
* [Related Topics](https://msdn.microsoft.com/library/hh191443(vs.110).aspx#BKMK_RelatedTopics)
* [Complete Example](https://msdn.microsoft.com/library/hh191443(vs.110).aspx#BKMK_CompleteExample)
* [Related Topics](https://msdn.microsoft.com/library/hh191443(vs.110).aspx#seeAlsoToggle)

This topic provides an overview of when and how to use async programming and includes links to support topics that contain details and examples.

## Async Improves Responsiveness

Asynchrony is essential for activities that are potentially blocking, such as when your application accesses the web. Access to a web resource sometimes is slow or delayed. If such an activity is blocked within a synchronous process, the entire application must wait. In an asynchronous process, the application can continue with other work that doesn't depend on the web resource until the potentially blocking task finishes.

The following table shows typical areas where asynchronous programming improves responsiveness. The listed APIs from the .NET Framework 4.5 and the Windows Runtime contain methods that support async programming.

|  |  |
| --- | --- |
| **Application area** | **Supporting APIs that contain async methods** |
| Web access | [HttpClient](https://msdn.microsoft.com/en-us/library/system.net.http.httpclient(v=vs.110).aspx), [SyndicationClient](http://go.microsoft.com/fwlink/p/?LinkId=259441) |
| Working with files | [StorageFile](http://go.microsoft.com/fwlink/p/?LinkId=248220), [StreamWriter](https://msdn.microsoft.com/en-us/library/system.io.streamwriter(v=vs.110).aspx), [StreamReader](https://msdn.microsoft.com/en-us/library/system.io.streamreader(v=vs.110).aspx), [XmlReader](https://msdn.microsoft.com/en-us/library/system.xml.xmlreader(v=vs.110).aspx) |
| Working with images | [MediaCapture](http://go.microsoft.com/fwlink/p/?LinkId=261839), [BitmapEncoder](http://go.microsoft.com/fwlink/p/?LinkId=261840), [BitmapDecoder](http://go.microsoft.com/fwlink/p/?LinkId=261841) |
| WCF programming | [Synchronous and Asynchronous Operations](http://go.microsoft.com/fwlink/p/?LinkID=192382) |

Asynchrony proves especially valuable for applications that access the UI thread because all UI-related activity usually shares one thread. If any process is blocked in a synchronous application, all are blocked. Your application stops responding, and you might conclude that it has failed when instead it's just waiting.

When you use asynchronous methods, the application continues to respond to the UI. You can resize or minimize a window, for example, or you can close the application if you don't want to wait for it to finish.

The async-based approach adds the equivalent of an automatic transmission to the list of options that you can choose from when designing asynchronous operations. That is, you get all the benefits of traditional asynchronous programming but with much less effort from the developer.

## Async Methods Are Easier to Write

The [Async](https://msdn.microsoft.com/en-us/library/hh191564(v=vs.110).aspx) and [Await](https://msdn.microsoft.com/en-us/library/hh191564(v=vs.110).aspx) keywords in Visual Basic and the [async](https://msdn.microsoft.com/en-us/library/hh156513(v=vs.110).aspx) and [await](https://msdn.microsoft.com/en-us/library/hh156528(v=vs.110).aspx) keywords in C# are the heart of async programming. By using those two keywords, you can use resources in the .NET Framework 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 async and await are referred to as async methods.

The following example shows an async method. Almost everything in the code should look completely familiar to you. The comments call out the features that you add to create the asynchrony.

You can find the complete example file at the end of this topic, and you can download the sample from [Async Sample: Example from "Asynchronous Programming with Async and Await"](http://go.microsoft.com/fwlink/?LinkID=261549).

C#

[**VB**](https://msdn.microsoft.com/library/hh191443(v=vs.110).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-1)

// Three things to note in the signature:

// - The method has an async modifier.

// - The return type is Task or Task<T>. (See "Return Types" section.)

// Here, it is Task<int> because the return statement returns an integer.

// - The method name ends in "Async."

async Task<int> AccessTheWebAsync()

{

// You need to add a reference to System.Net.Http to declare client.

HttpClient client = new HttpClient();

// GetStringAsync returns a Task<string>. That means that when you await the

// task you'll get a string (urlContents).

Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");

// You can do work here that doesn't rely on the string from GetStringAsync.

DoIndependentWork();

// The await operator suspends AccessTheWebAsync.

// - AccessTheWebAsync can't continue until getStringTask is complete.

// - Meanwhile, control returns to the caller of AccessTheWebAsync.

// - Control resumes here when getStringTask is complete.

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

string urlContents = await getStringTask;

// The return statement specifies an integer result.

// Any methods that are awaiting AccessTheWebAsync retrieve the length value.

return urlContents.Length;

}

If AccessTheWebAsync 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#

[**VB**](https://msdn.microsoft.com/library/hh191443(v=vs.110).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-2)

string urlContents = await client.GetStringAsync();

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

* The method signature includes an **Async** or **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](https://msdn.microsoft.com/en-us/library/dd321424(v=vs.110).aspx) if your method has a return statement in which the operand has type TResult.
  + [Task](https://msdn.microsoft.com/en-us/library/system.threading.tasks.task(v=vs.110).aspx) if your method has no return statement or has a return statement with no operand.
  + Void (a [Sub](https://msdn.microsoft.com/en-us/library/831f9wka(v=vs.110).aspx) in Visual Basic) if you're writing an async event handler.

For more information, see "Return Types and Parameters" later in this topic.

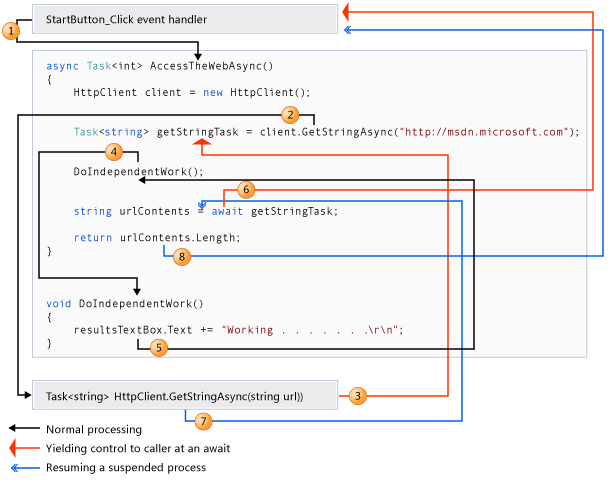
* The method usually includes at least one await expression, which marks a point where the method can't continue until the awaited asynchronous operation is complete. In the meantime, the method is suspended, and control returns to the method's caller. The next section of this topic illustrates what 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 the .NET Framework, see [TPL and Traditional .NET Framework Asynchronous Programming](https://msdn.microsoft.com/en-us/library/dd997423(v=vs.110).aspx).

## 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.



The numbers in the diagram correspond to the following steps.

1. An event handler calls and awaits the AccessTheWebAsync async method.
2. AccessTheWebAsync creates an [HttpClient](https://msdn.microsoft.com/en-us/library/system.net.http.httpclient(v=vs.110).aspx) instance and calls the [GetStringAsync](https://msdn.microsoft.com/en-us/library/hh551746(v=vs.110).aspx) 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, AccessTheWebAsync.

**GetStringAsync** returns a [Task](https://msdn.microsoft.com/en-us/library/dd321424(v=vs.110).aspx) where TResult is a string, and AccessTheWebAsync 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.

1. Because getStringTask hasn't been awaited yet, AccessTheWebAsync 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.
2. DoIndependentWork is a synchronous method that does its work and returns to its caller.
3. AccessTheWebAsync has run out of work that it can do without a result from getStringTask. AccessTheWebAsync 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, AccessTheWebAsync uses an await operator to suspend its progress and to yield control to the method that called AccessTheWebAsync. AccessTheWebAsync returns a **Task(Of Integer)** or **Task<int>** to the caller. The task represents a promise to produce an integer result that's the length of the downloaded string.

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

Inside the caller (the event handler in this example), the process is repeated. The caller might do other work that doesn't depend on the result from AccessTheWebAsync before awaiting that result, or the caller might await immediately. When the event handler reaches an await expression, the application focuses on the completion of **GetStringAsync**. The event handler is waiting for AccessTheWebAsync, and AccessTheWebAsync is waiting for **GetStringAsync**.

1. **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 urlContents.
2. When AccessTheWebAsync has the string result, the method can calculate the length of the string. Then the work of AccessTheWebAsync 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.