



What's New in C# 8 Interfaces

and how to use them effectively

Jeremy Clark

www.jeremybytes.com

@jeremybytes

Default Implementation

```
public interface ILogger
{
    void Log(LogLevel level, string message);

    void LogException(Exception ex)
    {
        Log(LogLevel.Error, ex.ToString());
    }
}
```

Default Implementation

```
public interface ILogger
{
    void Log(LogLevel level, string message);

    void LogException(Exception ex)
    {
        Log(LogLevel.Error, ex.ToString());
    }
}
```

New Features

- Default Implementation
 - Methods, Properties, Events, and Indexers
- Access Modifiers
 - public, private, protected, internal, etc.
- Static Members
 - Methods, Fields, Constructors
- Abstract Members
- Partial Interfaces
- Static Main

Reasons for the Update

- Compatibility with Java (Android) and Swift (iOS)
- Extend existing APIs
- Mix-ins

Before C# 8

- No implementation
- Everything "public"
- No statics
- No fields

```
public interface IRegularPolygon
{
    int NumberOfSides { get; }
    int SideLength { get; set; }

    double GetPerimeter();
    double GetArea();
}
```

Before C# 8

Interface

- May not contain implementation code
- A class may implement any number of interfaces
- Members are always public
- May contain properties, methods, events, and indexers (not fields, constructors or destructors)
- No static members

Abstract Class

- May contain implementation code
- A class may only descend from a single base class
- Members contain access modifiers
- May contain fields, properties, constructors, destructors, methods, events and indexers
- May contain static members

After C# 8

Interface

- ~~May not contain implementation code~~
- A class may implement any number of interfaces
- ~~Members are always public~~
- ~~May contain properties, methods, events, and indexers (not fields, constructors or destructors)~~
- ~~No static members~~

Abstract Class

- May contain implementation code
- A class may only descend from a single base class
- Members contain access modifiers
- May contain fields, properties, constructors, destructors, methods, events and indexers
- May contain static members

Using C# 8 (and Default Implementation)

- .NET Core 3.1
- .NET Standard 2.1
- .NET 5 (coming soon!)

- 
- .NET Framework 4.8
 - .NET Standard 2.0

New Features

- Default Implementation
 - Methods, Properties, Events, and Indexers
- Access Modifiers
 - public, private, protected, internal, etc.
- Static Members
 - Methods, Fields, Constructors
- Abstract Members
- Partial Interfaces
- Static Main

Default Method Implementation

```
public interface ILogger
{
    void Log(LogLevel level, string message);

    void LogException(Exception ex)
    {
        Log(LogLevel.Error, ex.ToString());
    }
}
```

Calling a Default Method Implementation

- Must be called using the interface type
- The class type will not work
- "var" will not work

```
var ex = new InvalidOperationException();  
ILogger logger1 = new InitialLogger();  
logger1.LogException(ex); // calls default  
  
InitialLogger logger2 = new InitialLogger();  
logger2.LogException(ex); // compiler error  
  
var logger3 = new InitialLogger();  
logger3.LogException(ex); // compiler error
```

Explicit Implementation

```
public class ExplicitLogger : ILogger
{
    void ILogger.Log(LogLevel level, string message)
    {
        Console.WriteLine($"Level - {level:F}: {message}");
    }

    void ILogger.LogException(Exception ex)
    {
        Console.WriteLine($"Level - {LogLevel.Error}: {ex.Message}");
    }
}
```

```
public class ExplicitLogger : ILogger
{
    void ILogger.Log(LogLevel level, string message)
    {
        Console.WriteLine($"Level - {level:F}: {message}");
    }

    void ILogger.LogException(Exception ex)
    {
        Console.WriteLine($"Level - {LogLevel.Error}: {ex.Message}");
    }
}
```

Explicit Implementation

```
ILogger logger1 = new ExplicitLogger();
logger1.Log(LogLevel.Info, "Hello"); // calls method

ExplicitLogger logger2 = new ExplicitLogger();
logger2.Log(LogLevel.Info, "Hello"); // compiler error

var logger3 = new ExplicitLogger();
logger3.Log(LogLevel.Info, "Hello"); // compiler error
```

What is an Interface?

- An interface describes a set of capabilities of an object.
- The capabilities might not be directly exposed (example: explicitly implemented interface members).
- An object may only have the capabilities if it is referenced the right way (i.e., by the interface type rather than the class type).
- A default implementation is called the same way as an explicit implementation.

Default implementation is called the same way as explicit implementation

```
var ex = new InvalidOperationException();  
ILogger logger1 = new InitialLogger();  
logger1.LogException(ex); // calls default
```

```
InitialLogger logger2 = new InitialLogger();  
logger2.LogException(ex); // compiler error
```

```
var logger3 = new InitialLogger();  
logger3.LogException(ex); // compiler error
```

Overriding a Default

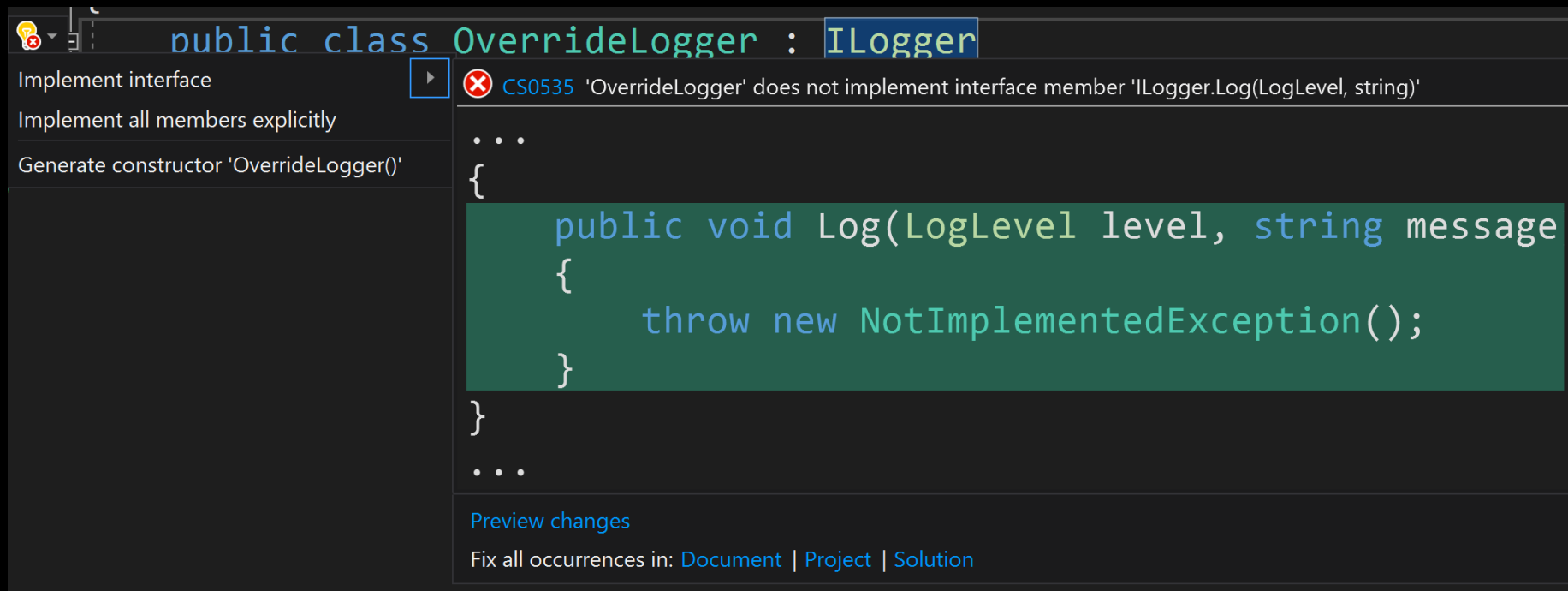
- Implementing class can provide an implementation

```
public class OverrideLogger : ILogger
{
    public void Log(LogLevel level, string message)
    {
        Console.WriteLine($"Level - {level:F}: {message}");
    }

    public void LogException(Exception ex)
    {
        Console.WriteLine($"Level - {LogLevel.Error}: {ex.Message}");
    }
}
```

Visual Studio Tooling Quirk

- Visual Studio 2019 (16.6.2) does not include interface members with default implementation in the "Implement interface" shortcut.



Calling an Overridden Interface Member

- The most specific method is always called (meaning, the default implementation is ignored if a class has a public implementation).

```
ILogger logger1 = new OverrideLogger();  
logger1.Log(LogLevel.Info, "Hello"); // calls class method  
  
OverrideLogger logger2 = new OverrideLogger();  
logger2.Log(LogLevel.Info, "Hello"); // calls class method  
  
var logger3 = new OverrideLogger();  
logger3.Log(LogLevel.Info, "Hello"); // calls class method
```

Confused Yet?

If a class does **NOT** provide an implementation...

- Using the interface type, the default is called.
- Using the class type, the code does not compile.
- Using "var", the code does not compile.

If the class **DOES** provide an implementation...

- Using the interface type, the class method is called.
- Using the class type, the class method is called.
- Using "var", the class method is called.



Recommendation

When calling interface members, use the interface type.

"dynamic" and Default Implementation

- "dynamic" relies on runtime type information.
- The result is that "dynamic" cannot see default implementations.

```
public interface IRegularPolygon
{
    int NumberOfSides { get; }
    int SideLength { get; set; }

    double GetPerimeter() => NumberOfSides * SideLength;

    double GetArea();
}
```


A Tale of Two Objects

- Square relies on the default implementation for "GetPerimeter()".

```
public class Square : IRegularPolygon
{
    public int NumberOfSides { get; }
    public int SideLength { get; set; }

    public Square(int sideLength)
    {
        NumberOfSides = 4;
        SideLength = sideLength;
    }

    public double GetArea()
    {
        return SideLength * SideLength;
    }
}
```

<https://github.com/jeremybytes/csharp-8-interfaces/blob/main/CodeSamples/DynamicAndDefaultImplementation/Polygon/Square.cs>

A Tale of Two Objects

- Triangle provides its own implementation for "GetPerimeter()".

```
public class Triangle : IRegularPolygon
{
    public int NumberOfSides { get; }
    public int SideLength { get; set; }

    public Triangle(int sideLength)
    {
        NumberOfSides = 3;
        SideLength = sideLength;
    }

    public double GetPerimeter() => NumberOfSides * SideLength;

    public double GetArea() =>
        SideLength * SideLength * Math.Sqrt(3) / 4;
}
```

Methods on a "dynamic" object

```
private static void ShowDynamicPolygonPerimeter(dynamic polygon)
{
    Console.WriteLine($"Perimeter: {polygon.GetPerimeter()}");
}
```

- With a Triangle object, this method runs fine. Triangle has a "GetPerimeter" method.
- With a Square object, this method throws a runtime exception. Square does not directly have a "GetPerimeter" method. "dynamic" does not see the interface.

Unit Testing & Mocks

- Mocking frameworks do not call default implementations by default.
- There are open issues on Moq, NSubstitute, and FakeItEasy regarding default implementation.

```
public interface IRegularPolygon
{
    int NumberOfSides { get; }
    int SideLength { get; set; }

    double GetPerimeter() => NumberOfSides * SideLength;
    double GetArea();
}
```

Example: Moq

```
[Test]
public void Moq_CheckDefaultImplementation()
{
    var mock = new Mock<IRegularPolygon>();
    mock.CallBase = true;
    mock.SetupGet(m => m.NumberOfSides).Returns(3);
    mock.SetupGet(m => m.SideLength).Returns(5);

    double result = mock.Object.GetPerimeter();

    Assert.AreEqual(15.0, result);
}
```

- Test Fails: Since "GetPerimeter" is not set up on the mock object, Moq uses its own default behavior (which is to return "0" for a method returning an integer).

Be Careful of Assumptions

- What's wrong with the following interface?

```
public interface IFileHandler
{
    void Delete(string filename);

    void Rename(string filename, string newfilename) =>
        System.IO.File.Move(filename, newfilename);
}
```

- Answer: It makes assumptions about the IFileHandler implementers (specifically that they use System.IO.File objects).



Recommendation

Default implementations
should only reference other
interface members.

Be Careful of Assumptions

- What's wrong with the following interface?

```
public interface IReader<T>
{
    IReadOnlyCollection<T> GetItems();
    T GetItemAt(int index) => GetItems().ElementAt(index);
}
```

- Answer: It assumes that using an iterator is okay. If "GetItems" returns a large collection, this could cause memory pressure. If the "Get" operation is slow (for example a large calculation), then getting the entire list to pull out a single item is inefficient.

Default Property Implementation

- Good for calculated properties (getters)

```
public interface IRegularPolygon
{
    int NumberOfSides { get; }
    int SideLength { get; set; }

    double Perimeter { get => NumberOfSides * SideLength; }
```

Default Property Implementation

- Read/Write Properties must have default implementation for both "get" and "set"... *

```
int NotAllowed
{
    get => 20;
    set;
}
```

```
int AlsoNotAllowed
{
    get;
    set { }
}
```

*See caveat on next slide

<https://github.com/jeremybytes/csharp-8-interfaces/blob/main/CodeSamples/InterfaceProperties/BadInterface/IBadInterface.cs>

Default Property Implementation

- Default implementation doesn't really make sense for "set" (there's no way to have a backing field).
- Things like this cause a StackOverflow:

```
int BadMember
{
    get => 1;
    set { BadMember = value; }
}
```

<https://github.com/jeremybytes/csharp-8-interfaces/blob/main/CodeSamples/InterfaceProperties/BadInterface/IBadInterface.cs>

Default Property Implementation

- Default implementation cannot be used to specify an automatic property.
- The following interface properties are normal (abstract) interface members:

```
public interface IRegularPolygon
{
    int NumberOfSides { get; }
    int SideLength { get; set; }
```

Access Modifiers

- All access modifiers are allowed
 - public
 - private
 - protected
 - internal
 - etc.
- Default access modifier is "public"
- "privates" must be implemented

Access Modifiers - Public

- Default access modifier is "public" for interfaces.
- The following interface members are both "public":

```
public interface ICustomerReader
{
    IReadOnlyCollection<Customer> GetCustomers();
    public Customer GetCustomer(int Id);
}
```


Access Modifiers - Private

- "private" members can only be accessed within the interface.

```
public interface ICalendarItem
{
    private CalendarItemType DefaultItemType { get => CalendarItemType.Unspecified; }
    public CalendarItemType ItemType { get => DefaultItemType; }
```

- "private" members must have a default implementation.

```
public interface ICalendarItem
{
    private CalendarItemType DefaultItemType { get; } // ERROR!
    public CalendarItemType ItemType { get => DefaultItemType; }
```

*Note: This sample is a bit contrived to show that "private" members must have implementation

Access Modifiers - Private

- "private" members can be used to break up larger default implementation methods.
- For example, if a "public" method has a complex default implementation, it can be split up into smaller "private" methods inside the interface.
- My Opinion: If code inside an interface is complex enough that it requires this type of factoring, maybe it is not appropriate for it to be part of an interface.

Access Modifiers - Protected

- "protected" members are possible, but are a bit strange.
- "protected" members do **not** require a default implementation.

```
public interface IInventoryController
{
    public void PushInventoryItem(InventoryItem item);
    protected InventoryItem PullInventoryItem(int id);
}
```

Access Modifiers - Protected

- "protected" members must be implemented explicitly by the implementing class.

```
public class TestInventoryController : IInventoryController
{
    public void PushInventoryItem(InventoryItem item)
    {
        throw new NotImplementedException();
    }

    InventoryItem IInventoryController.PullInventoryItem(int id)
    {
        throw new NotImplementedException();
    }
}
```

<https://github.com/jeremybytes/csharp-8-interfaces/blob/main/CodeSamples/AccessModifiers/Protected/TestInventoryController.cs>

Access Modifiers - Protected

- HOWEVER, there is no way to call the "protected" member.

```
IInventoryController controller = new TestInventoryController();  
var item = new InventoryItem(7);  
controller.PushInventoryItem(item);  
controller.PullInventoryItem(7);
```



★ `InventoryItem IInventoryController.PullInventoryItem(int id)`

'IInventoryController.PullInventoryItem(int)' is inaccessible due to its protection level

[Show potential fixes](#) (Alt+Enter or Ctrl+.)

Access Modifiers - Protected

```
public interface IInventoryController
{
    public void PushInventoryItem(InventoryItem item);
    protected InventoryItem PullInventoryItem(int id);
}
```

- "protected" members may be useful in an interface hierarchy (an interface that derives from this one can reference the protected member). However, I have not seen a good use case for this.
- My opinion: Stick with "public" and "private" members.

<https://github.com/jeremybytes/csharp-8-interfaces/blob/main/CodeSamples/AccessModifiers/Protected/IInventoryController.cs>

Static Methods

- Interface "static" methods are just like "static" methods on a class.
- The following are equivalent:

```
public class ReaderFactory
{
    private static IPeopleReader savedReader;
    public static Type readerType =
        typeof(HardCodedPeopleReader);

    public static IPeopleReader GetReader()
    {
        Implementation details
        return savedReader;
    }
}
```

```
public interface IReaderFactory
{
    private static IPeopleReader savedReader;
    public static Type readerType =
        typeof(HardCodedPeopleReader);

    public static IPeopleReader GetReader()
    {
        Implementation details
        return savedReader;
    }
}
```


Static Fields

- "static" fields on an interface are just like "static" fields on a class.
- A static field is a shared value; it is associated with the interface rather than any particular instance.

Static Fields as Parameters

- "static" fields can be used to parameterize a default implementation.
- See Microsoft Docs example:
<https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/default-interface-methods-versions#provide-parameterization>

Abstract Members

- A member can be marked "abstract" (which is the default).
- The following properties are both abstract:

```
public interface ICustomerReader
{
    IReadOnlyCollection<Customer> GetCustomers();
    abstract Customer GetCustomer(int Id);
}
```

- In complex hierarchies a member with default implementation can be re-abstracted. (Note: If you're doing this, your code may be too complex.)

Partial Interfaces

- Interfaces can now be marked "partial" (just like classes).
- This allows them to be extended in a separate file (and at a later time).

```
public partial interface ICustomerReader
{
    IReadOnlyCollection<Customer> GetCustomers();
    Customer GetCustomer(int Id);
}
```

Static Main

"static Main()" is the entry point to an application.

- This is a valid console application (just this file):

```
using System;

namespace StaticMain
{
    public interface IHelloWorld
    {
        static void Main(string[] args)
        {
            if (args?.Length == 0)
                Console.WriteLine("Hello World!");
            else
                Console.WriteLine($"Hello {args[0]}!");
        }
    }
}
```

Static Main

- A few notes about "static Main()":
 - "static Main()" is the entry point to an application.
 - "static Main()" can be in any class (it doesn't have to be in "Program").
 - There can only be one "static Main()" method per application.*
 - "static Main()" can now be in an interface.

* If you have more than one "static Main()", you *must* use the "-main" compiler directive or the "<StartupObject>" element in the project file to specify the startup object.

More info: <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/compiler-options/main-compiler-option>

Reasons for the Update

- Compatibility with Java (Android) and Swift (iOS)
- Extend existing APIs
- Mix-ins

Compatibility with Android and iOS

- Both Java (Android) and Swift (iOS) offer default implementation.
- For mobile & cross-platform developers, this provides C# with compatibility.
- My Opinion: Compatibility with libraries is a good reason to support default implementation (however, I would like to see the use limited as much as possible).

Extending APIs

- Default implementation allows API maintainers to extend an API after it is released.
- My Opinion:
 - Strays from the Interface Segregation Principle (ISP).
 - Interface inheritance supports a more controlled way of extending an interface.
 - Because default implementations should call existing members, it is difficult to come up with scenarios that are not trivial.

Changing an API

- When we have control over both the interface and the implementers, default implementation can give us a way to change the API without breaking the build.
- Sample Scenario:
 - Add new member (with default implementation) to an existing API.
 - Application continues to build and run.
 - API implementers can add the new member and functionality at their leisure (hopefully quickly).
 - As soon as the implementers are updated, the default implementation can be removed from the interface.

Mix-Ins

- Mix-ins are a cool idea.
- Create interfaces with *only* default implementations.
- Results in a fairly safe way to do multiple inheritance.
- IoT example:
- <https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/mixins-with-default-interface-methods>
- My Opinion: I like the idea of mix-ins, but I wish that it was implemented with a different construct.

New Features

- Default Implementation
 - Methods, Properties, Events, and Indexers
- Access Modifiers
 - public, private, protected, internal, etc.
- Static Members
 - Methods, Fields, Constructors
- Abstract Members
- Partial Interfaces
- Static Main



Thank You!

Jeremy Clark

- <http://www.jeremybytes.com>
- jeremy@jeremybytes.com
- @jeremybytes