Visual Studio LIVE! EXPERT SOLUTIONS FOR ENTERPRISE DEVELOPERS | Las Vegas

Catching Up with C# Interfaces

What you know might be wrong

Jeremy Clark

Developer Betterer

jeremybytes.com

Level: Intermediate

Resources

Code Samples & Resources

https://github.com/jeremybytes/vslive2023-lasvegas

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());
    }
}
```

MORE THAN JUST THIS

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
- Static Abstract Members (.NET 7 / C# 11)

Here There Be Dragons

You might choose not to use these features, but you may encounter them in the libraries you use.

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)
- **May not** contain static members

Abstract Class

- **May** contain implementation code
- A class may only descend from a **single** base class
- Members have **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)~~
- ~~**May not** contain static members~~

Abstract Class

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

Reasons for the Updates

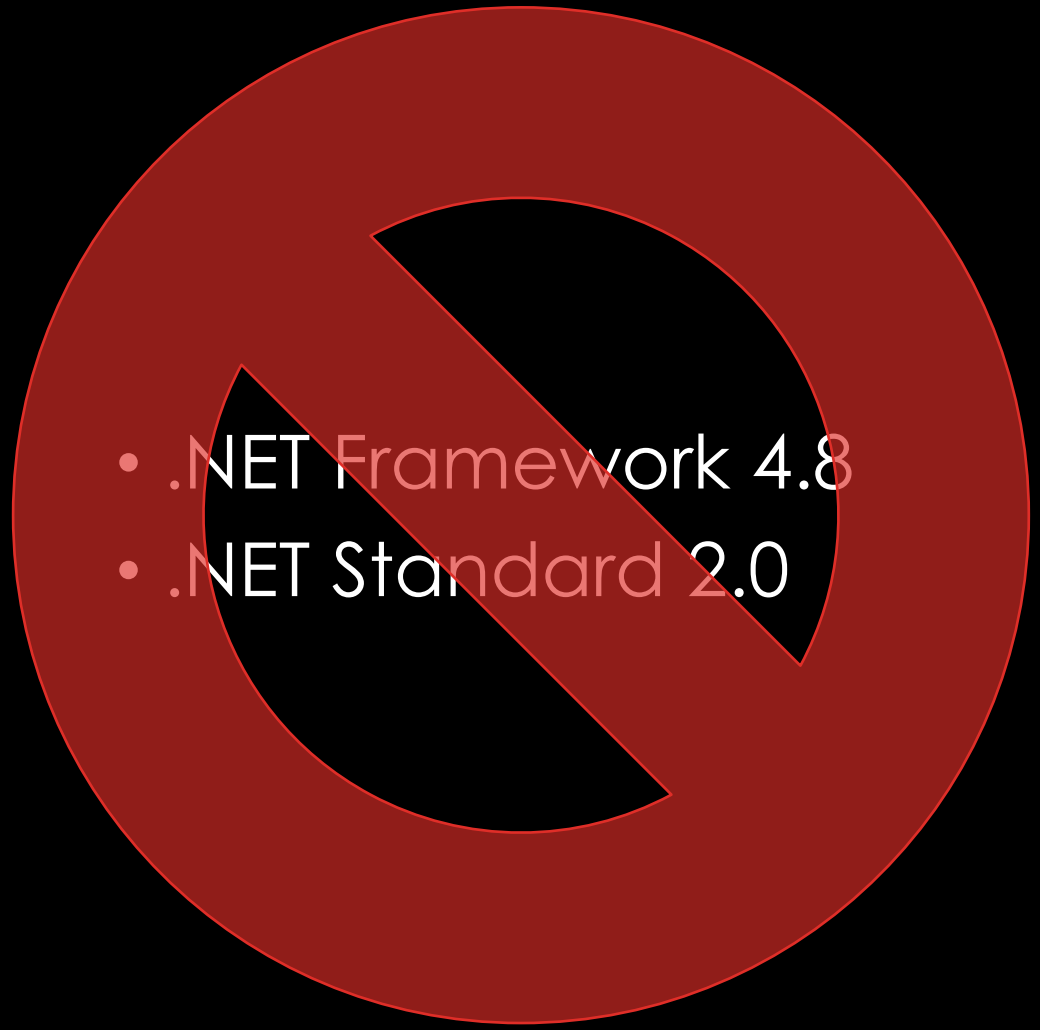
- Default Implementation
  - Compatibility with Java (Android) and Swift (iOS)
  - Extend existing APIs
  - Mix-ins
- Static Abstract
  - Operator overloading (ex. Math libraries)

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
- Static Abstract Members (.NET 7 / C# 11)

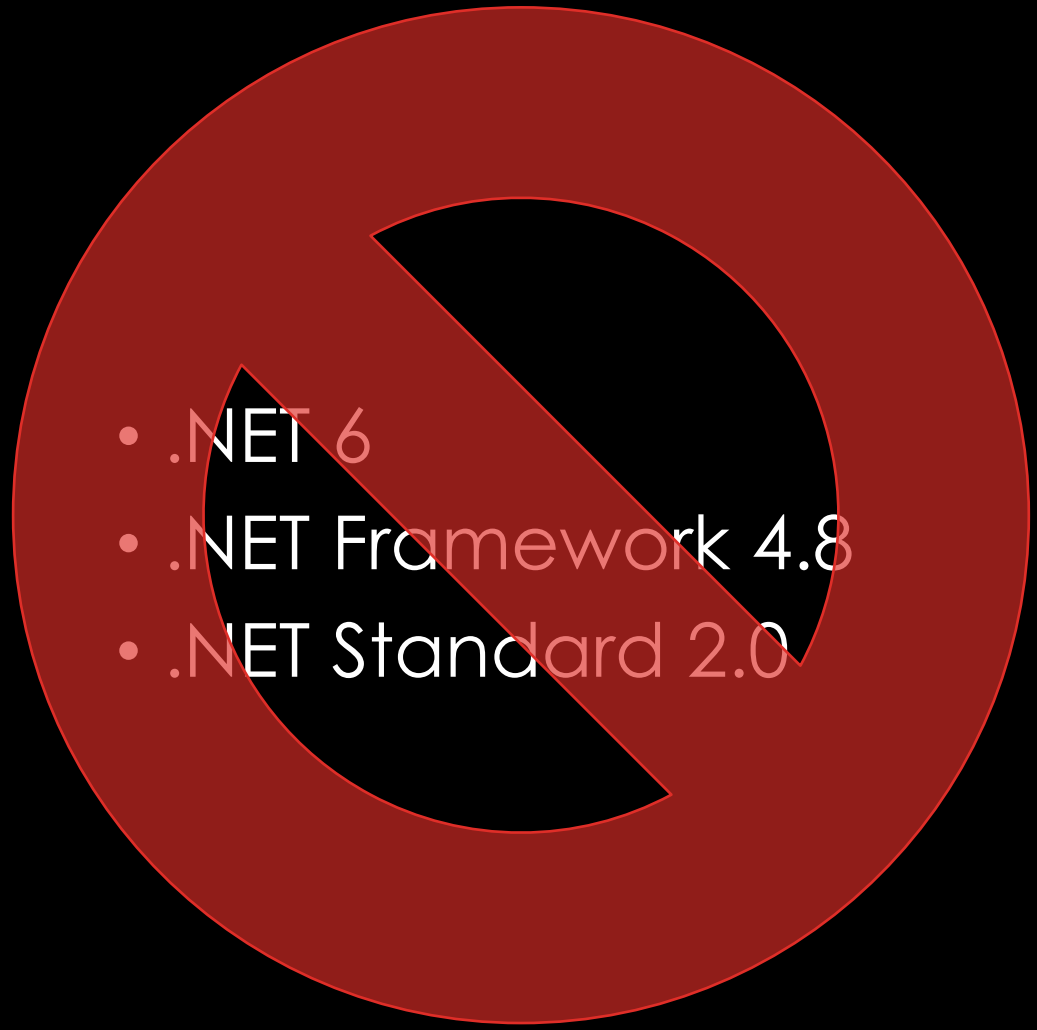
Default Implementation (and other features)

- .NET 6
- .NET 7



Static Abstract Members

- .NET 7



What is an Interface?

An interface describes a set of capabilities of an object.

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
- Static Abstract Members (.NET 7 / C# 11)

Default Method Implementation

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

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

/DefaultImplementation/ILogger.cs

Recommendation

When calling interface members, use the interface type.

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
```

Replacing a Default Implementation

- Implementing class can provide its own implementation

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

    public void LogException(Exception ex)
    {
        Log(LogLevel.Error, $"From JeremyLogger: {ex.Message}");
    }
}
```

Visual Studio Tooling Quirk

- Visual Studio 2022 does not include interface members with default implementation in the "Implement interface" shortcut.

Calling an Implemented 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 JeremyLogger();
logger1.LogException(ex);

JeremyLogger logger2 = new JeremyLogger();
logger2.LogException(ex);

var logger3 = new JeremyLogger();
logger3.LogException(ex);
```

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}");
    }
}
```

/DefaultImplementation/ExplicitLogger.cs

```
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
```

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
```

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.

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.

Be Careful of Assumptions

- What's wrong with the following interface?

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


}
```

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

/DangerousAssumptions/BadInterface/IFileHandler.cs

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.

/DangerousAssumptions/SlowPerformance/IReader.cs

Recommendation

Know the capabilities of your mocking framework.

Unit Testing & Mocks

Many mocking frameworks do not support testing default implementation.

- Supported
  - **Moq**
  - **Rocks**
- Not Supported
  - **FakeItEasy**
    (open issue: https://github.com/FakeItEasy/FakeItEasy/issues/1633)
  - **NSubstitute**

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

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

Example: FakeItEasy

```
[Test]
0 references
public void FakeItEasy_CheckDefaultImplementation()
{
    var mock = A.Fake<IRegularPolygon>();
    A.CallTo(() => mock.NumberOfSides).Returns(3);
    A.CallTo(() => mock.SideLength).Returns(5);

    double result = mock.GetPerimeter();

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

- Test Fails: Since "GetPerimeter" is not set up on the mock object, FakeItEasy uses its own default behavior for a mock object (which is to return "0.0" for a method returning a double).

/UnitTests.Tests/IRegularPolygonTests.cs

```
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 Succeeds: “CallBase = true” will call the “GetPerimeter” default implementation from the interface.

/UnitTests.Tests/IRegularPolygonTests.cs

Recommendation

Know the capabilities of your mocking framework.

Observation

Default implementation is good for calculated properties.

(And not much else for properties.)

Default Property Implementation

- Good for calculated properties (getters)

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

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

/InterfaceProperties/Interface/IRegularPolygon.cs

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

/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; }
}
```

/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; }
```

/InterfaceProperties/Interface/IRegularPolygon.cs

Observation

Default implementation is good for calculated properties.

(And not much else for properties.)

Access Modifiers

All access modifiers are (technically) allowed

- "public" is default
- "private" has limited usefulness

DANGER
- "protected" and "internal" are undefined

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);
}
```

/AccessModifiers/Public/ICustomerReader.cs

Access Modifiers - Private

- "private" members can only be accessed within the interface.
- "private" members must have a default implementation.

```
private int DirectDistance((int, int) point1, (int, int) point2)
{
    (int x1, int y1) = point1;
    (int x2, int y2) = point2;
    return (x1 - x2) + (y1 - y2);
}
1 reference
private int CubeIt(int x) => x * x * x;
1 reference
private int FlipSign(int x) => x * -1;
```

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

/AccessModifiers/Private/IDistanceCalculator.cs

Access Modifiers - Private

- "private" members can be used to break up larger default implementation methods.
- 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

- Behavior of protected / internal members is undefined in the language specification.
- Note: the compiler does not stop you from using protected / internal access modifiers.

My Opinion: Stick with "public" and "private" members.

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;
    }
}
```

/StaticMembers/Factories

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/whats-new/tutorials/default-interface-methods-versions#provide-parameterization

Abstract Members

- A member can be marked "abstract"
- “abstract” is the default.
- The following properties are both abstract:

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

Partial Interfaces

- Interfaces can 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):

```
namespace StaticMain;

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

/StaticMain/IHelloWorld.cs

Static Abstract Members

- Interfaces can have static abstract members.

```
public interface IGetNext<T>
    where T : IGetNext<T>
{
    static abstract T operator ++(T other);
}
```

- This means that interface implementations must have that static member with the member implementation.

Static Abstract Members

- Can be used for operator overloads with interfaces.

```
public interface IGetNext<T>
    where T : IGetNext<T>
{
    static abstract T operator ++(T other);
}
```

- Can also be used for static factories.

Operator Overload Example 1

```
public record RepeatSequence : IGetNext<RepeatSequence>
{
    private const char Ch = 'A';
    public string Text = new string(Ch, 1);

    public static RepeatSequence operator ++(RepeatSequence other)
    {
        return other with { Text = other.Text + Ch };
    }

    public override string ToString() => Text;
}
```

https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/tutorials/static-virtual-interface-members

Operator Overload Example 2

```
public record FibonacciNext : IGetNext<FibonacciNext>
{
    public int Previous = 0;
    public int Current = 1;

    public static FibonacciNext operator ++(FibonacciNext other)
    {
        checked
        {
            int next = other.Previous + other.Current;
            return other with { Previous = other.Current, Current = next };
        }
    }

    public override string ToString() => Current.ToString();
}
```

Using the Interface Operator

```
public class Nexter<T> : IEnumerable<T> where T : IGetNext<T>, new()
{
    public IEnumerator<T> GetEnumerator()
    {
        // first value
        T value = new();
        yield return value;

        // subsequent values
        while (true)
        {
            try { value++; }
            catch { break; }
            yield return value;
        }
    }
```

WARNING

- Interfaces with static abstract members cannot be used as generic type parameters.
- This can specifically cause problems with dependency injection.

Observation

Avoid using static abstract members unless you are building a framework that requires operator overloads.

(or in some other edge cases)

Reasons for the Updates

- Default Implementation
  - Compatibility with Java (Android) and Swift (iOS)
  - Extend existing APIs
  - Mix-ins
- Static Abstract
  - Generic Math / Operator overloading

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:

- Because default implementations should call existing members, it is difficult to come up with scenarios that are not trivial.
- Interface inheritance supports a more controlled way of extending an interface.

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/whats-new/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.

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
- Static Abstract Members (.NET 7 / C# 11)

Recommendations and Opinions 1

- When calling interface members, use the interface type.
- Default implementations should only reference other interface members.
- Know the capabilities of your mocking framework
- Default implementation is good for calculated properties. (And not much else for properties.)

Recommendations and Opinions 2

- If code inside an interface is complex enough that it requires this type of refactoring (private members), maybe it is not appropriate for it to be part of an interface.
- Stick with "public and "private" access modifiers.
- Compatability with libraries is a good reason to support default implementation (however, I would like to see the use limited as much as possible).

Recommendations and Opinions 3

- Because default implementations should call existing members, it is difficult to come up with scenarios that are not trivial.
- Interface inheritance supports a more controlled way of extending an interface.
- I like the idea of mix-ins, but I wish that it was implemented with a different construct.

Here There Be Dragons

You might choose not to use these features, but you may encounter them in the libraries you use.

Thank You!

Jeremy Clark

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

https://github.com/jeremybytes/vslive2023-lasvegas