https://github.com/jeremybytes/sdd-2022

Catching up with C# Interfaces
What you know is probably wrong

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

```csharp
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
- Abstract Static Members (coming soon!)

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

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

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

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

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

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 Update

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

An interface describes
a set of capabilities
of an object.

Using Default Implementation (and other features)

- .NET Core 3.1
- .NET Standard 2.1
- .NET 5 (out of support)
- .NET 6
- .NET 7 (coming soon!)

- .NET Framework 4.8
- .NET Standard 2.0

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

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

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

https://github.com/jeremybytes/06-catching-up-with-csharp-interfaces/DefaultImplementation/ILogger.cs

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

```csharp
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}");
    }
}
```

https://github.com/jeremybytes/sdd-2022/06-catching-up-with-csharp-interfaces/
DefaultImplementation/OverrideLogger.cs

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).

```csharp
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
```

Explicit Implementation

```csharp
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

```csharp
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}");
    }
}
```

```csharp
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.

©Jeremy Clark 2022

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?

```csharp
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).

https://github.com/jeremybytes/sdd-2022/06-catching-up-with-csharp-interfaces/
DangerousAssumptions/BadInterface/IFileHandler.cs

Recommendation

Default implementations should only reference other interface members.

©Jeremy Clark 2022

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.

Recommendation

Know the capabilities of your mocking framework.

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**

Example: FakeItEasy

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

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

```csharp
[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" for a method returning an integer).

https://github.com/jeremybytes/sdd-2022/06-catching-up-with-csharp-interfaces/UnitTests.Tests/IRegularPolygonTests.cs

Example: Moq

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

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

```csharp
[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.

Recommendation

Know the capabilities of your mocking framework.

Default implementation is good for calculated properties.

(And not much else for properties.)

Default Property Implementation

- Good for calculated properties (getters)

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

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

https://github.com/jeremybytes/sdd-2022/06-catching-up-with-csharp-interfaces/
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

https://github.com/jeremybytes/sdd-2022/06-catching-up-with-csharp-interfaces/
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:

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

https://github.com/jeremybytes/sdd-2022/06-catching-up-with-csharp-interfaces/
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:

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

Access Modifiers

All access modifiers are allowed

- "public" is default
- "private" has limited usefulness
- "protected" is undefined **DANGER**

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.
- "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
https://github.com/jeremybytes/sdd-2022/06-catching-up-with-csharp-interfaces/
AccessModifiers/Private/ICalendarItem.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 members is undefined in the language specification.
- "protected" members are possible, but are a bit strange.
- "protected" members do *not* require a default implementation.

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

https://github.com/jeremybytes/sdd-2022/06-catching-up-with-csharp-interfaces/
AccessModifiers/Protected/IInventoryController.cs

Access Modifiers - Protected

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

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

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

Access Modifiers - Protected

- "protected" members can be implemented as "public" in the implementing class.

```csharp
public class TestInventoryController : IInventoryController
{
    public void PushInventoryItem(InventoryItem item)...

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

Access Modifiers - Protected

- However, there is **no way** to call a "protected" interface 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

- "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.

- 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/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 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):

```csharp
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]}!");
        }
    }
}
```

https://github.com/jeremybytes/sdd-2022/06-catching-up-with-csharp-interfaces/StaticMain/IHelloWorld.cs

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/advanced

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:

- 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
- Abstract Static Members (coming soon!)

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.

Code Samples & Resources

https://github.com/jeremybytes/
sdd-2022

Thank You!

Jeremy Clark

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