# default interface methods

Article • 08/12/2022 • 27 minutes to read

## Summary

Add support for *virtual extension methods* - methods in interfaces with concrete implementations. A class or struct that implements such an interface is required to have a single *most specific* implementation for the interface method, either implemented by the class or struct, or inherited from its base classes or interfaces. Virtual extension methods enable an API author to add methods to an interface in future versions without breaking source or binary compatibility with existing implementations of that interface.

These are similar to Java's "Default Methods" .

(Based on the likely implementation technique) this feature requires corresponding support in the CLI/CLR. Programs that take advantage of this feature cannot run on earlier versions of the platform.

## Motivation

The principal motivations for this feature are

- Default interface methods enable an API author to add methods to an interface in future versions without breaking source or binary compatibility with existing implementations of that interface.
- The feature enables C# to interoperate with APIs targeting Android (Java) and iOS (Swift) , which support similar features.
- As it turns out, adding default interface implementations provides the elements of the "traits" language feature (https://en.wikipedia.org/wiki/Trait_(computer_programming) ). Traits have proven to be a powerful programming technique (http://scg.unibe.ch/archive/papers/Scha03aTraits.pdf ).

## Detailed design

The syntax for an interface is extended to permit

- member declarations that declare constants, operators, static constructors, and nested types;
- a *body* for a method or indexer, property, or event accessor (that is, a "default" implementation);
- member declarations that declare static fields, methods, properties, indexers, and events;
- member declarations using the explicit interface implementation syntax; and
- Explicit access modifiers (the default access is `public`).

Members with bodies permit the interface to provide a "default" implementation for the method in classes and structs that do not provide their own implementation.

Interfaces may not contain instance state. While static fields are now permitted, instance fields are not permitted in interfaces. Instance auto-properties are not supported in interfaces, as they would implicitly declare a hidden field.

Static and private methods permit useful refactoring and organization of code used to implement the interface's public API.

A method override in an interface must use the explicit interface implementation syntax.

It is an error to declare a class type, struct type, or enum type within the scope of a type parameter that was declared with a *variance_annotation*. For example, the declaration of `C` below is an error.

C#

```
interface IOuter<out T>
{
    class C { } // error: class declaration within the scope of variant type
parameter 'T'
}
```

## Concrete methods in interfaces

The simplest form of this feature is the ability to declare a *concrete method* in an interface, which is a method with a body.

C#

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
```

A class that implements this interface need not implement its concrete method.

C#

```
class C : IA { } // OK

IA i = new C();
i.M(); // prints "IA.M"
```

The final override for `IA.M` in class `C` is the concrete method `M` declared in `IA`. Note that a class does not inherit members from its interfaces; that is not changed by this feature:

C#

```
new C().M(); // error: class 'C' does not contain a member 'M'
```

Within an instance member of an interface, `this` has the type of the enclosing interface.

## Modifiers in interfaces

The syntax for an interface is relaxed to permit modifiers on its members. The following are permitted: `private`, `protected`, `internal`, `public`, `virtual`, `abstract`, `sealed`, `static`, `extern`, and `partial`.

> ***TODO***: check what other modifiers exist.

An interface member whose declaration includes a body is a `virtual` member unless the `sealed` or `private` modifier is used. The `virtual` modifier may be used on a function member that would otherwise be implicitly `virtual`. Similarly, although `abstract` is the default on interface members without bodies, that modifier may be given explicitly. A non-virtual member may be declared using the `sealed` keyword.

It is an error for a `private` or `sealed` function member of an interface to have no body. A `private` function member may not have the modifier `sealed`.

Access modifiers may be used on interface members of all kinds of members that are permitted. The access level `public` is the default but it may be given explicitly.

> ***Open Issue:*** We need to specify the precise meaning of the access modifiers such as `protected` and `internal`, and which declarations do and do not override them (in a derived interface) or implement them (in a class that implements the interface).

Interfaces may declare `static` members, including nested types, methods, indexers, properties, events, and static constructors. The default access level for all interface members is `public`.

Interfaces may not declare instance constructors, destructors, or fields.

> ***Closed Issue:*** Should operator declarations be permitted in an interface? Probably not conversion operators, but what about others? ***Decision***: Operators are permitted *except* for conversion, equality, and inequality operators.

> ***Closed Issue:*** Should `new` be permitted on interface member declarations that hide members from base interfaces? ***Decision***: Yes.

> ***Closed Issue:*** We do not currently permit `partial` on an interface or its members. That would require a separate proposal. ***Decision***: Yes. https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#permit-partial-in-interface

## Explicit implementation in interfaces

Explicit implementations allow the programmer to provide a most specific implementation of a virtual member in an interface where the compiler or runtime would not otherwise find one. An implementation declaration is permitted to *explicitly* implement a particular base interface method by qualifying the declaration with the interface name (no access modifier is permitted in this case). Implicit implementations are not permitted.

C#

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
```

```
interface IB : IA
{
    void IA.M() { WriteLine("IB.M"); } // Explicit implementation
}
interface IC : IA
{
    void M() { WriteLine("IC.M"); } // Creates a new M, unrelated to `IA.M`.
Warning
}
```

Explicit implementations in interfaces may not be declared `sealed`.

Public `virtual` function members in an interface may only be implemented in a derived interface explicitly (by qualifying the name in the declaration with the interface type that originally declared the method, and omitting an access modifier). The member must be *accessible* where it is implemented.

## Reabstraction

A virtual (concrete) method declared in an interface may be reabstracted in a derived interface

C#

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
interface IB : IA
{
    abstract void IA.M();
}
class C : IB { } // error: class 'C' does not implement 'IA.M'.
```

The `abstract` modifier is required in the declaration of `IB.M`, to indicate that `IA.M` is being reabstracted.

This is useful in derived interfaces where the default implementation of a method is inappropriate and a more appropriate implementation should be provided by implementing classes.

> ***Open Issue:*** Should reabstraction be permitted?

## The most specific implementation rule

We require that every interface and class have a *most specific implementation* for every virtual member among the implementations appearing in the type or its direct and indirect interfaces. The *most specific implementation* is a unique implementation that is more specific than every other implementation. If there is no implementation, the member itself is considered the most specific implementation.

One implementation `M1` is considered *more specific* than another implementation `M2` if `M1` is declared on type `T1`, `M2` is declared on type `T2`, and either

1. `T1` contains `T2` among its direct or indirect interfaces, or
2. `T2` is an interface type but `T1` is not an interface type.

For example:

C#

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
interface IB : IA
{
    void IA.M() { WriteLine("IB.M"); }
}
interface IC : IA
{
    void IA.M() { WriteLine("IC.M"); }
}
interface ID : IB, IC { } // error: no most specific implementation for 'IA.M'
abstract class C : IB, IC { } // error: no most specific implementation for
'IA.M'
abstract class D : IA, IB, IC // ok
{
    public abstract void M();
}
```

The most specific implementation rule ensures that a conflict (i.e. an ambiguity arising from diamond inheritance) is resolved explicitly by the programmer at the point where the conflict arises.

Because we support explicit reabstractions in interfaces, we could do so in classes as well

C#

```
abstract class E : IA, IB, IC // ok
{
    abstract void IA.M();
}
```

> ***Open issue***: should we support explicit interface abstract implementations in classes?

In addition, it is an error if in a class declaration the most specific implementation of some interface method is an abstract implementation that was declared in an interface. This is an existing rule restated using the new terminology.

C#

```
interface IF
{
    void M();
}
abstract class F : IF { } // error: 'F' does not implement 'IF.M'
```

It is possible for a virtual property declared in an interface to have a most specific implementation for its `get` accessor in one interface and a most specific implementation for its `set` accessor in a different interface. This is considered a violation of the *most specific implementation* rule.

## `static` and `private` methods

Because interfaces may now contain executable code, it is useful to abstract common code into private and static methods. We now permit these in interfaces.

> ***Closed issue***: Should we support private methods? Should we support static methods?
> **Decision: YES**

> ***Open issue***: should we permit interface methods to be `protected` or `internal` or other access? If so, what are the semantics? Are they `virtual` by default? If so, is there a way to make them non-virtual?

> ***Open issue***: If we support static methods, should we support (static) operators?

## Base interface invocations

Code in a type that derives from an interface with a default method can explicitly invoke that interface's "base" implementation.

C#

```
interface I0
{
   void M() { Console.WriteLine("I0"); }
}
interface I1 : I0
{
   override void M() { Console.WriteLine("I1"); }
}
interface I2 : I0
{
   override void M() { Console.WriteLine("I2"); }
}
interface I3 : I1, I2
{
   // an explicit override that invoke's a base interface's default method
   void I0.M() { I2.base.M(); }
}
```

An instance (nonstatic) method is permitted to invoke the implementation of an accessible instance method in a direct base interface nonvirtually by naming it using the syntax `base(Type).M`. This is useful when an override that is required to be provided due to diamond inheritance is resolved by delegating to one particular base implementation.

C#

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
interface IB : IA
{
    override void IA.M() { WriteLine("IB.M"); }
}
interface IC : IA
{
    override void IA.M() { WriteLine("IC.M"); }
}

class D : IA, IB, IC
```

```
{
    void IA.M() { base(IB).M(); }
}
```

When a `virtual` or `abstract` member is accessed using the syntax `base(Type).M`, it is required that `Type` contains a unique *most specific override* for `M`.

## Binding base clauses

Interfaces now contain types. These types may be used in the base clause as base interfaces. When binding a base clause, we may need to know the set of base interfaces to bind those types (e.g. to lookup in them and to resolve protected access). The meaning of an interface's base clause is thus circularly defined. To break the cycle, we add a new language rules corresponding to a similar rule already in place for classes.

While determining the meaning of the *interface_base* of an interface, the base interfaces are temporarily assumed to be empty. Intuitively this ensures that the meaning of a base clause cannot recursively depend on itself.

**We used to have the following rules:**

"When a class B derives from a class A, it is a compile-time error for A to depend on B. A class **directly depends on** its direct base class (if any) and **directly depends on** the ~~**class**~~ within which it is immediately nested (if any). Given this definition, the complete set of ~~**classes**~~ upon which a class depends is the reflexive and transitive closure of the **directly depends on** relationship."

It is a compile-time error for an interface to directly or indirectly inherit from itself. The **base interfaces** of an interface are the explicit base interfaces and their base interfaces. In other words, the set of base interfaces is the complete transitive closure of the explicit base interfaces, their explicit base interfaces, and so on.

**We are adjusting them as follows:**

When a class B derives from a class A, it is a compile-time error for A to depend on B. A class **directly depends on** its direct base class (if any) and **directly depends on** the ***type*** within which it is immediately nested (if any).

When an interface IB extends an interface IA, it is a compile-time error for IA to depend on IB. An interface **directly depends on** its direct base interfaces (if any) and **directly depends on** the type within which it is immediately nested (if any).

Given these definitions, the complete set of **types** upon which a type depends is the reflexive and transitive closure of the **directly depends on** relationship.

## Effect on existing programs

The rules presented here are intended to have no effect on the meaning of existing programs.

Example 1:

C#

```
interface IA
{
    void M();
}
class C: IA // Error: IA.M has no concrete most specific override in C
{
    public static void M() { } // method unrelated to 'IA.M' because static
}
```

Example 2:

C#

```
interface IA
{
    void M();
}
class Base: IA
{
    void IA.M() { }
}
class Derived: Base, IA // OK, all interface members have a concrete most spe-
cific override
{
    private void M() { } // method unrelated to 'IA.M' because private
}
```

The same rules give similar results to the analogous situation involving default interface methods:

C#

```
interface IA
{
    void M() { }
}
class Derived: IA // OK, all interface members have a concrete most specific
override
{
    private void M() { } // method unrelated to 'IA.M' because private
}
```

> ***Closed issue***: confirm that this is an intended consequence of the specification.
> **Decision: YES**

## Runtime method resolution

> ***Closed Issue:*** The spec should describe the runtime method resolution algorithm in the face of interface default methods. We need to ensure that the semantics are consistent with the language semantics, e.g. which declared methods do and do not override or implement an `internal` method.

## CLR support API

In order for compilers to detect when they are compiling for a runtime that supports this feature, libraries for such runtimes are modified to advertise that fact through the API discussed in https://github.com/dotnet/corefx/issues/17116 . We add

C#

```
namespace System.Runtime.CompilerServices
{
    public static class RuntimeFeature
    {
        // Presence of the field indicates runtime support
        public const string DefaultInterfaceImplementation =
nameof(DefaultInterfaceImplementation);
    }
}
```

> ***Open issue***: Is that the best name for the *CLR* feature? The CLR feature does much more than just that (e.g. relaxes protection constraints, supports overrides in interfaces,

> etc). Perhaps it should be called something like "concrete methods in interfaces", or "traits"?

### Further areas to be specified

- [ ] It would be useful to catalog the kinds of source and binary compatibility effects caused by adding default interface methods and overrides to existing interfaces.

## Drawbacks

This proposal requires a coordinated update to the CLR specification (to support concrete methods in interfaces and method resolution). It is therefore fairly "expensive" and it may be worth doing in combination with other features that we also anticipate would require CLR changes.

## Alternatives

None.

## Unresolved questions

- Open questions are called out throughout the proposal, above.
- See also https://github.com/dotnet/csharplang/issues/406 for a list of open questions.
- The detailed specification must describe the resolution mechanism used at runtime to select the precise method to be invoked.
- The interaction of metadata produced by new compilers and consumed by older compilers needs to be worked out in detail. For example, we need to ensure that the metadata representation that we use does not cause the addition of a default implementation in an interface to break an existing class that implements that interface when compiled by an older compiler. This may affect the metadata representation that we can use.
- The design must consider interoperation with other languages and existing compilers for other languages.

## Resolved Questions

## Abstract Override

The earlier draft spec contained the ability to "reabstract" an inherited method:

C#

```
interface IA
{
    void M();
}
interface IB : IA
{
    override void M() { }
}
interface IC : IB
{
    override void M(); // make it abstract again
}
```

My notes for 2017-03-20 showed that we decided not to allow this. However, there are at least two use cases for it:

1. The Java APIs, with which some users of this feature hope to interoperate, depend on this facility.
2. Programming with *traits* benefits from this. Reabstraction is one of the elements of the "traits" language feature (https://en.wikipedia.org/wiki/Trait_(computer_programming) ). The following is permitted with classes:

C#

```
public abstract class Base
{
    public abstract void M();
}
public abstract class A : Base
{
    public override void M() { }
}
public abstract class B : A
{
    public override abstract void M(); // reabstract Base.M
}
```

Unfortunately this code cannot be refactored as a set of interfaces (traits) unless this is permitted. By the *Jared principle of greed*, it should be permitted.

> ***Closed issue:*** Should reabstraction be permitted? [YES] My notes were wrong. The LDM notes say that reabstraction is permitted in an interface. Not in a class.

## Virtual Modifier vs Sealed Modifier

From Aleksey Tsingauz :

> We decided to allow modifiers explicitly stated on interface members, unless there is a reason to disallow some of them. This brings an interesting question around virtual modifier. Should it be required on members with default implementation?
>
> We could say that:
>
> - if there is no implementation and neither virtual, nor sealed are specified, we assume the member is abstract.
> - if there is an implementation and neither abstract, nor sealed are specified, we assume the member is virtual.
> - sealed modifier is required to make a method neither virtual, nor abstract.
>
> Alternatively, we could say that virtual modifier is required for a virtual member. I.e, if there is a member with implementation not explicitly marked with virtual modifier, it is neither virtual, nor abstract. This approach might provide better experience when a method is moved from a class to an interface:
>
> - an abstract method stays abstract.
> - a virtual method stays virtual.
> - a method without any modifier stays neither virtual, nor abstract.
> - sealed modifier cannot be applied to a method that is not an override.
>
> What do you think?

> ***Closed Issue:*** Should a concrete method (with implementation) be implicitly `virtual`? [YES]

***Decisions:*** Made in the LDM 2017-04-05:

1. non-virtual should be explicitly expressed through `sealed` or `private`.

2. `sealed` is the keyword to make interface instance members with bodies non-virtual
3. We want to allow all modifiers in interfaces
4. Default accessibility for interface members is public, including nested types
5. private function members in interfaces are implicitly sealed, and `sealed` is not permitted on them.
6. Private classes (in interfaces) are permitted and can be sealed, and that means sealed in the class sense of sealed.
7. Absent a good proposal, partial is still not allowed on interfaces or their members.

## Binary Compatibility 1

When a library provides a default implementation

C#

```
interface I1
{
    void M() { Impl1 }
}
interface I2 : I1
{
}
class C : I2
{
}
```

We understand that the implementation of `I1.M` in `C` is `I1.M`. What if the assembly containing `I2` is changed as follows and recompiled

C#

```
interface I2 : I1
{
    override void M() { Impl2 }
}
```

but `C` is not recompiled. What happens when the program is run? An invocation of `(C as I1).M()`

1. Runs `I1.M`
2. Runs `I2.M`

3. Throws some kind of runtime error

***Decision:*** Made 2017-04-11: Runs `I2.M`, which is the unambiguously most specific override at runtime.

## Event accessors (closed)

> ***Closed Issue:*** Can an event be overridden "piecewise"?

Consider this case:

C#

```
public interface I1
{
    event T e1;
}
public interface I2 : I1
{
    override event T
    {
        add { }
        // error: "remove" accessor missing
    }
}
```

This "partial" implementation of the event is not permitted because, as in a class, the syntax for an event declaration does not permit only one accessor; both (or neither) must be provided. You could accomplish the same thing by permitting the abstract remove accessor in the syntax to be implicitly abstract by the absence of a body:

C#

```
public interface I1
{
    event T e1;
}
public interface I2 : I1
{
    override event T
    {
        add { }
        remove; // implicitly abstract
```

```
    }
}
```

Note that *this is a new (proposed) syntax*. In the current grammar, event accessors have a mandatory body.

> ***Closed Issue:*** Can an event accessor be (implicitly) abstract by the omission of a body, similarly to the way that methods in interfaces and property accessors are (implicitly) abstract by the omission of a body?

***Decision:*** (2017-04-18) No, event declarations require both concrete accessors (or neither).

## Reabstraction in a Class (closed)

***Closed Issue:*** We should confirm that this is permitted (otherwise adding a default implementation would be a breaking change):

C#

```
interface I1
{
    void M() { }
}
abstract class C : I1
{
    public abstract void M(); // implement I1.M with an abstract method in C
}
```

***Decision:*** (2017-04-18) Yes, adding a body to an interface member declaration shouldn't break C.

## Sealed Override (closed)

The previous question implicitly assumes that the `sealed` modifier can be applied to an `override` in an interface. This contradicts the draft specification. Do we want to permit sealing an override? Source and binary compatibility effects of sealing should be considered.

> ***Closed Issue:*** Should we permit sealing an override?

***Decision:*** (2017-04-18) Let's not allowed `sealed` on overrides in interfaces. The only use of `sealed` on interface members is to make them non-virtual in their initial declaration.

## Diamond inheritance and classes (closed)

The draft of the proposal prefers class overrides to interface overrides in diamond inheritance scenarios:

> We require that every interface and class have a *most specific override* for every interface method among the overrides appearing in the type or its direct and indirect interfaces. The *most specific override* is a unique override that is more specific than every other override. If there is no override, the method itself is considered the most specific override.
>
> One override `M1` is considered *more specific* than another override `M2` if `M1` is declared on type `T1`, `M2` is declared on type `T2`, and either
>
> 1. `T1` contains `T2` among its direct or indirect interfaces, or
> 2. `T2` is an interface type but `T1` is not an interface type.

The scenario is this

C#

```
interface IA
{
    void M();
}
interface IB : IA
{
    override void M() { WriteLine("IB"); }
}
class Base : IA
{
    void IA.M() { WriteLine("Base"); }
}
class Derived : Base, IB // allowed?
{
    static void Main()
    {
        IA a = new Derived();
        a.M();           // what does it do?
    }
}
```

We should confirm this behavior (or decide otherwise)

> ***Closed Issue:*** Confirm the draft spec, above, for *most specific override* as it applies to mixed classes and interfaces (a class takes priority over an interface). See https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-19.md#diamonds-with-classes .

## Interface methods vs structs (closed)

There are some unfortunate interactions between default interface methods and structs.

C#

```
interface IA
{
    public void M() { }
}
struct S : IA
{
}
```

Note that interface members are not inherited:

C#

```
var s = default(S);
s.M(); // error: 'S' does not contain a member 'M'
```

Consequently, the client must box the struct to invoke interface methods

C#

```
IA s = default(S); // an S, boxed
s.M(); // ok
```

Boxing in this way defeats the principal benefits of a `struct` type. Moreover, any mutation methods will have no apparent effect, because they are operating on a *boxed copy* of the struct:

C#

```
interface IB
{
    public void Increment() { P += 1; }
    public int P { get; set; }
}
struct T : IB
{
    public int P { get; set; } // auto-property
}

T t = default(T);
Console.WriteLine(t.P); // prints 0
(t as IB).Increment();
Console.WriteLine(t.P); // prints 0
```

> ***Closed Issue:*** What can we do about this:
>
> 1. Forbid a `struct` from inheriting a default implementation. All interface methods would be treated as abstract in a `struct`. Then we may take time later to decide how to make it work better.
> 2. Come up with some kind of code generation strategy that avoids boxing. Inside a method like `IB.Increment`, the type of `this` would perhaps be akin to a type parameter constrained to `IB`. In conjunction with that, to avoid boxing in the caller, non-abstract methods would be inherited from interfaces. This may increase compiler and CLR implementation work substantially.
> 3. Not worry about it and just leave it as a wart.
> 4. Other ideas?

***Decision:*** Not worry about it and just leave it as a wart. See https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-19.md#structs-and-default-implementations .

## Base interface invocations (closed)

The draft spec suggests a syntax for base interface invocations inspired by Java: `Interface.base.M()`. We need to select a syntax, at least for the initial prototype. My favorite is `base<Interface>.M()`.

> ***Closed Issue:*** What is the syntax for a base member invocation?

***Decision:*** The syntax is `base(Interface).M()`. See https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-19.md#base-invocation . The interface so named must be a base interface, but does not need to be a direct base interface.

> ***Open Issue:*** Should base interface invocations be permitted in class members?

***Decision***: Yes. https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-19.md#base-invocation

## Overriding non-public interface members (closed)

In an interface, non-public members from base interfaces are overridden using the `override` modifier. If it is an "explicit" override that names the interface containing the member, the access modifier is omitted.

> ***Closed Issue:*** If it is an "implicit" override that does not name the interface, does the access modifier have to match?

***Decision:*** Only public members may be implicitly overridden, and the access must match. See https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-18.md#dim-implementing-a-non-public-interface-member-not-in-list .

> ***Open Issue:*** Is the access modifier required, optional, or omitted on an explicit override such as `override void IB.M() {}`?

> ***Open Issue:*** Is `override` required, optional, or omitted on an explicit override such as `void IB.M() {}`?

How does one implement a non-public interface member in a class? Perhaps it must be done explicitly?

C#

```
interface IA
{
    internal void MI();
    protected void MP();
}
```

```
class C : IA
{
    // are these implementations?
    internal void MI() {}
    protected void MP() {}
}
```

> ***Closed Issue:*** How does one implement a non-public interface member in a class?

***Decision:*** You can only implement non-public interface members explicitly. See https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-18.md#dim-implementing-a-non-public-interface-member-not-in-list .

***Decision***: No `override` keyword permitted on interface members. https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#does-an-override-in-an-interface-introduce-a-new-member

## Binary Compatibility 2 (closed)

Consider the following code in which each type is in a separate assembly

C#

```
interface I1
{
    void M() { Impl1 }
}
interface I2 : I1
{
    override void M() { Impl2 }
}
interface I3 : I1
{
}
class C : I2, I3
{
}
```

We understand that the implementation of `I1.M` in `C` is `I2.M`. What if the assembly containing `I3` is changed as follows and recompiled

C#

```
interface I3 : I1
{
    override void M() { Impl3 }
}
```

but `C` is not recompiled. What happens when the program is run? An invocation of `(C as I1).M()`

1. Runs `I1.M`
2. Runs `I2.M`
3. Runs `I3.M`
4. Either 2 or 3, deterministically
5. Throws some kind of runtime exception

***Decision***: Throw an exception (5). See https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#issues-in-default-interface-methods .

## Permit `partial` in interface? (closed)

Given that interfaces may be used in ways analogous to the way abstract classes are used, it may be useful to declare them `partial`. This would be particularly useful in the face of generators.

> ***Proposal:*** Remove the language restriction that interfaces and members of interfaces may not be declared `partial`.

***Decision***: Yes. See https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#permit-partial-in-interface .

## `Main` in an interface? (closed)

> ***Open Issue:*** Is a `static Main` method in an interface a candidate to be the program's entry point?

***Decision***: Yes. See https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#main-in-an-interface .

## Confirm intent to support public non-virtual methods (closed)

Can we please confirm (or reverse) our decision to permit non-virtual public methods in an interface?

```
C#
```

```
interface IA
{
    public sealed void M() { }
}
```

> ***Semi-Closed Issue:*** (2017-04-18) We think it is going to be useful, but will come back to it. This is a mental model tripping block.

***Decision***: Yes. https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#confirm-that-we-support-public-non-virtual-methods .

## Does an `override` in an interface introduce a new member? (closed)

There are a few ways to observe whether an override declaration introduces a new member or not.

```
C#
```

```
interface IA
{
    void M(int x) { }
}
interface IB : IA
{
    override void M(int y) { }
}
interface IC : IB
{
```

```
    static void M2()
    {
        M(y: 3); // permitted?
    }
    override void IB.M(int z) { } // permitted? What does it override?
}
```

> ***Open Issue:*** Does an override declaration in an interface introduce a new member? (closed)

In a class, an overriding method is "visible" in some senses. For example, the names of its parameters take precedence over the names of parameters in the overridden method. It may be possible to duplicate that behavior in interfaces, as there is always a most specific override. But do we want to duplicate that behavior?

Also, it is possible to "override" an override method? [Moot]

***Decision***: No `override` keyword permitted on interface members. https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#does-an-override-in-an-interface-introduce-a-new-member .

## Properties with a private accessor (closed)

We say that private members are not virtual, and the combination of virtual and private is disallowed. But what about a property with a private accessor?

C#

```
interface IA
{
    public virtual int P
    {
        get => 3;
        private set => { }
    }
}
```

Is this allowed? Is the `set` accessor here `virtual` or not? Can it be overridden where it is accessible? Does the following implicitly implement only the `get` accessor?

C#

```
class C : IA
{
    public int P
    {
        get => 4;
        set { }
    }
}
```

Is the following presumably an error because IA.P.set isn't virtual and also because it isn't accessible?

C#

```
class C : IA
{
    int IA.P
    {
        get => 4;
        set { }
    }
}
```

***Decision***: The first example looks valid, while the last does not. This is resolved analogously to how it already works in C#.
https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#properties-with-a-private-accessor

## Base Interface Invocations, round 2 (closed)

Our previous "resolution" to how to handle base invocations doesn't actually provide sufficient expressiveness. It turns out that in C# and the CLR, unlike Java, you need to specify both the interface containing the method declaration and the location of the implementation you want to invoke.

I propose the following syntax for base calls in interfaces. I'm not in love with it, but it illustrates what any syntax must be able to express:

C#

```
interface I1 { void M(); }
interface I2 { void M(); }
interface I3 : I1, I2 { void I1.M() { } void I2.M() { } }
```

```
interface I4 : I1, I2 { void I1.M() { } void I2.M() { } }
interface I5 : I3, I4
{
    void I1.M()
    {
        base<I3>(I1).M(); // calls I3's implementation of I1.M
        base<I4>(I1).M(); // calls I4's implementation of I1.M
    }
    void I2.M()
    {
        base<I3>(I2).M(); // calls I3's implementation of I2.M
        base<I4>(I2).M(); // calls I4's implementation of I2.M
    }
}
```

If there is no ambiguity, you can write it more simply

C#

```
interface I1 { void M(); }
interface I3 : I1 { void I1.M() { } }
interface I4 : I1 { void I1.M() { } }
interface I5 : I3, I4
{
    void I1.M()
    {
        base<I3>.M(); // calls I3's implementation of I1.M
        base<I4>.M(); // calls I4's implementation of I1.M
    }
}
```

Or

C#

```
interface I1 { void M(); }
interface I2 { void M(); }
interface I3 : I1, I2 { void I1.M() { } void I2.M() { } }
interface I5 : I3
{
    void I1.M()
    {
        base(I1).M(); // calls I3's implementation of I1.M
    }
    void I2.M()
    {
        base(I2).M(); // calls I3's implementation of I2.M
```

```
    }
}
```

Or

C#

```
interface I1 { void M(); }
interface I3 : I1 { void I1.M() { } }
interface I5 : I3
{
    void I1.M()
    {
        base.M(); // calls I3's implementation of I1.M
    }
}
```

***Decision***: Decided on `base(N.I1<T>).M(s)`, conceding that if we have an invocation binding there may be problem here later on.
https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-11-14.md#default-interface-implementations

## Warning for struct not implementing default method? (closed)

@vancem asserts that we should seriously consider producing a warning if a value type declaration fails to override some interface method, even if it would inherit an implementation of that method from an interface. Because it causes boxing and undermines constrained calls.

***Decision***: This seems like something more suited for an analyzer. It also seems like this warning could be noisy, since it would fire even if the default interface method is never called and no boxing will ever occur.
https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#warning-for-struct-not-implementing-default-method

## Interface static constructors (closed)

When are interface static constructors run? The current CLI draft proposes that it occurs when the first static method or field is accessed. If there are neither of those then it might

never be run??

[2018-10-09 The CLR team proposes "Going to mirror what we do for valuetypes (cctor check on access to each instance method)"]

***Decision***: Static constructors are also run on entry to instance methods, if the static constructor was not `beforefieldinit`, in which case static constructors are run before access to the first static field.
https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#when-are-interface-static-constructors-run

# Design meetings

2017-03-08 LDM Meeting Notes 2017-03-21 LDM Meeting Notes 2017-03-23 meeting "CLR Behavior for Default Interface Methods" 2017-04-05 LDM Meeting Notes 2017-04-11 LDM Meeting Notes 2017-04-18 LDM Meeting Notes 2017-04-19 LDM Meeting Notes 2017-05-17 LDM Meeting Notes 2017-05-31 LDM Meeting Notes 2017-06-14 LDM Meeting Notes 2018-10-17 LDM Meeting Notes 2018-11-14 LDM Meeting Notes