Interfaces

An **interface** in C# is a contract that specifies a set of members (methods, properties, events, indexers) that a class or struct must implement. Unlike classes, interfaces primarily define *what* a type can do, rather than *how* it does it or *what data* it holds.

Key characteristics of interfaces:

- **Behavior Definition Only:** Interfaces can define only functions (methods, properties, events, indexers). They **cannot contain fields** (data) because their purpose is to specify behavior, not state.
- **Implicitly Abstract Members:** Interface members are implicitly `abstract` by default, meaning they do not provide an implementation within the interface itself. The implementing class or struct is responsible for providing the actual code for these members.
- **Multiple Implementation:** A class or struct can **implement multiple interfaces**. This is a key advantage over classes, as a class can only inherit from a single base class, and structs cannot inherit at all (beyond `System.ValueType`). This allows a single type to embody diverse functionalities from different contracts.
- **Implicitly `public`:** Interface members are always implicitly `public` and cannot declare an access modifier. This is because interfaces define a public contract.

Here's an example, using the `IEnumerator` interface from `System.Collections`:

```
public interface IEnumerator // Interface Declaration
{
    bool MoveNext();      // Method
    object Current { get; } // Read-only Property
    void Reset();         // Method
}
```

To implement an interface, a class or struct must provide a `public` implementation for all of its members:

```
internal class Countdown : IEnumerator // Implements the IEnumerator interface
{
    int count = 11;

    public bool MoveNext() => count-- > 0; // Implementation of MoveNext
    public object Current => count;        // Implementation of Current
    public void Reset() { throw new NotSupportedException(); } //
Implementation of Reset
}
```

You can implicitly cast an object to any interface it implements. This is a core aspect of polymorphism with interfaces.

```
IEnumerator e = new Countdown(); // Casts Countdown instance to IEnumerator
interface
while (e.MoveNext())
    Console.Write(e.Current); // Output: 109876543210
```

Even though `Countdown` is an `internal` class, its members that implement `IEnumerator` can be called publicly by casting an instance of `Countdown` to `IEnumerator`, provided the interface itself is accessible.

Extending an Interface

Interfaces can derive from other interfaces, allowing for the creation of interface hierarchies. The derived interface "inherits" all the members of its base interfaces.

```
public interface IUndoable             { void Undo(); }
public interface IRedoable : IUndoable { void Redo(); } // IRedoable extends
IUndoable
```

Any type that implements `IRedoable` must also implement all members of `IUndoable` (in this case, `Undo()`).

Explicit Interface Implementation

When a class implements multiple interfaces, sometimes there can be a **collision** between member signatures (e.g., two interfaces define a method with the same name but potentially different return types, or simply the same name and signature). To resolve such collisions, or to hide specialized interface members from the general public interface of the class, you can use **explicit interface implementation**.

```
interface I1 { void Foo(); }
interface I2 { int Foo(); }

public class Widget : I1, I2 // Implements both interfaces
{
    // Implicit implementation for I1.Foo()
    public void Foo()
    {
        Console.WriteLine("Widget's implementation of I1.Foo");
    }

    // Explicit implementation for I2.Foo()
    // Note the interface name preceding the member name, and no access
modifier.
    int I2.Foo()
    {
        Console.WriteLine("Widget's implementation of I2.Foo");
        return 42;
    }
}
```

When a member is explicitly implemented:

- It is not public by default; it is only accessible when the object is cast to the specific interface type.
- You cannot add an access modifier to an explicitly implemented member.

```
Widget w = new Widget();
w.Foo();       // Calls Widget's implicitly implemented Foo() (which is I1.Foo
here)
((I1)w).Foo(); // Calls Widget's implementation of I1.Foo()
((I2)w).Foo(); // Calls Widget's explicit implementation of I2.Foo()
```

Explicit implementation is also useful for hiding "boilerplate" or highly specialized interface members that are not part of the type's primary public API.

Implementing Interface Members Virtually

An implicitly implemented interface member (where you just write `public void Foo() { ... }` in the class) is, by default, `sealed`. To allow subclasses to override this implementation, you must mark it as `virtual` or `abstract` in the base class.

```
public interface IUndoable { void Undo(); }

public class TextBox : IUndoable
{
    public virtual void Undo() => Console.WriteLine("TextBox.Undo"); // Marked
virtual
}

public class RichTextBox : TextBox
{
    public override void Undo() => Console.WriteLine("RichTextBox.Undo"); //
Overrides
}
```

Calling the interface member through either the base class or the interface reference will correctly call the subclass's implementation due to polymorphism.

An explicitly implemented interface member (`void IUndoable.Undo()`) cannot be marked `virtual` and cannot be overridden in the usual manner. However, it can be **reimplemented** by a subclass.

Reimplementing an Interface in a Subclass

A subclass can **reimplement** an interface member that has already been implemented by a base class. This "hijacks" the member's implementation when called through the interface.

```
public interface IUndoable { void Undo(); }
```

```
public class TextBox : IUndoable
{
    void IUndoable.Undo() => Console.WriteLine("TextBox.Undo"); // Explicit
implementation
}

public class RichTextBox : TextBox, IUndoable // Re-implements IUndoable
{
    public void Undo() => Console.WriteLine("RichTextBox.Undo"); // New
implicit implementation
}
```

When called:

```
RichTextBox r = new RichTextBox();
r.Undo();                 // Output: RichTextBox.Undo (calls the new implicit
implementation)
((IUndoable)r).Undo();    // Output: RichTextBox.Undo (calls the new
implementation through the interface)
((TextBox)r).Undo();      // This would lead to a compile-time error if
TextBox's Undo was explicit.
                          // If TextBox.Undo was implicit and not virtual, this
would call TextBox.Undo.
```

Reimplementation is most effective and less confusing when dealing with explicitly implemented interface members in the base class. If the base class implements the member implicitly and not virtually, then calling through the base class type can lead to inconsistent behavior, which is generally undesirable.

Alternatives to Reimplementation:

It's generally better to design base classes to avoid the need for reimplementation. Strategies include:

- Marking implicitly implemented interface members as `virtual` when appropriate.

- For explicitly implemented members, create a `protected virtual` helper method that the explicit implementation calls. This allows subclasses to override the

logic via the `protected virtual` method.

```
public class TextBox : IUndoable
{
    void IUndoable.Undo() => Undo(); // Calls the protected virtual method
    protected virtual void Undo() => Console.WriteLine("TextBox.Undo");
}
public class RichTextBox : TextBox
{
    protected override void Undo() => Console.WriteLine("RichTextBox.Undo"); //
Overrides logic
}
```

-
- This pattern ensures that all paths (direct call, interface call) lead to the most derived implementation.

Interfaces and Boxing

When you convert a **struct to an interface**, **boxing** occurs. A copy of the struct is placed on the heap, and the interface reference points to this boxed copy.

```
interface I { void Foo(); }
struct S : I { public void Foo() { Console.WriteLine("S.Foo"); } }

S s = new S();
s.Foo(); // No boxing: Foo is called directly on the stack-allocated struct.

I i = s; // Boxing occurs here: 's' is copied to the heap, and 'i' refers to
the boxed copy.
i.Foo(); // Called on the boxed copy.
```

Calling an implicitly implemented member directly on a struct instance does not cause boxing. However, accessing it via an interface variable *does* cause boxing.

Default Interface Members (C# 8+)

From C# 8, you can provide a **default implementation** for an interface member. This makes the member optional for implementing classes or structs to implement.

```
interface ILogger
{
    void Log(string text) => Console.WriteLine(text); // Default implementation
}
```

This feature is highly advantageous for evolving interfaces in existing libraries without introducing breaking changes. If a class implements `ILogger` but does not provide its own `Log` method, it can still use the default.

Default implementations are always **explicit**. If a class implementing `ILogger` fails to define a `Log` method, the only way to call the default implementation is by casting the instance to the interface:

```
class Logger : ILogger { } // Does not implement Log()

((ILogger)new Logger()).Log("message"); // Calls the default interface implementation
```

This design prevents ambiguities when a class implements multiple interfaces that might provide the same default member.

Static Interface Members (C# 8+, C# 11+)

Interfaces can also declare **static members**. There are two main kinds:

Static Non-Virtual Interface Members

These exist primarily to support default interface members and are not implemented by classes or structs. Instead, they are consumed directly through the interface type. They can include fields, methods, properties, events, and indexers.

```
interface ILogger
```

```
{
    void Log(string text) => Console.WriteLine(Prefix + text); // Uses static
field
    static string Prefix = ""; // Static field in interface
}

ILogger.Prefix = "File log: "; // Accessing static member directly on the
interface
```

Static non-virtual interface members are `public` by default, but you can restrict their accessibility with modifiers (`private`, `protected`, `internal`). Instance fields are still prohibited in interfaces, maintaining the principle of defining behavior, not state.

Static Virtual/Abstract Interface Members (C# 11+)

From C# 11, **static virtual/abstract interface members** enable **static polymorphism**, an advanced feature.

- **`static abstract`**: Must be implemented by the implementing class or struct.
- **`static virtual`**: Can optionally be implemented by the implementing class or struct, providing a default.

```
interface ITypeDescribable
{
    static abstract string Description { get; } // Must be implemented
    static virtual string Category => null;      // Optional
implementation
}

class CustomerTest : ITypeDescribable
{
    public static string Description => "Customer tests"; // Mandatory
implementation
    public static string Category => "Unit testing";    // Optional
implementation
}
```

These are used with **constrained generic type parameters** to allow generic algorithms to operate on static members of types that implement the interface. This is a powerful feature that underpins concepts like "Generic Math."

Writing a Class Versus an Interface

Deciding whether to use a class or an interface is a fundamental design choice:

- **Use classes and subclasses for types that naturally share an implementation.** This aligns with the "is a" relationship (e.g., a `Dog` is an `Animal`). If types share common data and behavior, inheritance is appropriate.
- **Use interfaces for types that have independent implementations but share a common *contract* or *capability*.** This aligns with the "can do" or "has a" relationship (e.g., a `Car` can be `IDriveable`, a `Printer` can be `IPrintable`). Different types might implement the same interface in vastly different ways.

Consider the example of animals:

If Bird, Insect, FlyingCreature, and Carnivore were all classes, you couldn't represent an Eagle as a Bird, FlyingCreature, and Carnivore using class inheritance due to C#'s single-inheritance rule.

The solution is to convert capabilities that can be implemented in independent ways into interfaces:

```
abstract class Animal {}
abstract class Bird : Animal {}
abstract class Insect : Animal {}

interface IFlyingCreature {} // Different creatures fly differently
interface ICarnivore {}      // Different creatures hunt/eat differently

// Concrete classes can now implement multiple interfaces:
class Ostrich : Bird {}
class Eagle   : Bird, IFlyingCreature, ICarnivore {} // OK
class Bee     : Insect, IFlyingCreature {}           // OK
class Flea    : Insect, ICarnivore {}                // OK
```

This allows `Eagle` to be a `Bird` (sharing core animal and bird implementations) and also have the capabilities of `IFlyingCreature` and `ICarnivore` (implemented distinctly). Interfaces are essential for achieving flexibility, loose coupling, and plug-in architectures in your C# applications.