interface (C# Reference)

Article • 12/08/2022

An interface defines a contract. Any class or struct that implements that contract must provide an implementation of the members defined in the interface. An interface may define a default implementation for members. It may also define static members in order to provide a single implementation for common functionality. Beginning with C# 11, an interface may define `static abstract` or `static virtual` members to declare that an implementing type must provide the declared members. Typically, `static virtual` methods declare that an implementation must define a set of overloaded operators.

In the following example, class `ImplementationClass` must implement a method named `SampleMethod` that has no parameters and returns `void`.

For more information and examples, see Interfaces.

Example interface

C#

```csharp
interface ISampleInterface
{
    void SampleMethod();
}

class ImplementationClass : ISampleInterface
{
    // Explicit interface member implementation:
    void ISampleInterface.SampleMethod()
    {
        // Method implementation.
    }

    static void Main()
    {
        // Declare an interface instance.
        ISampleInterface obj = new ImplementationClass();

        // Call the member.
        obj.SampleMethod();
    }
}
```

An interface can be a member of a namespace or a class. An interface declaration can contain declarations (signatures without any implementation) of the following members:

- Methods
- Properties
- Indexers
- Events

Default interface members

These preceding member declarations typically don't contain a body. An interface member may declare a body. Member bodies in an interface are the *default implementation*. Members with bodies permit the interface to provide a "default" implementation for classes and structs that don't provide an overriding implementation. An interface may include:

- Constants
- Operators
- Static constructor.
- Nested types
- Static fields, methods, properties, indexers, and events
- Member declarations using the explicit interface implementation syntax.
- Explicit access modifiers (the default access is public).

Static abstract and virtual members

Beginning with C# 11, an interface may declare `static abstract` and `static virtual` members for all member types except fields. Interfaces can declare that implementing types must define operators or other static members. This feature enables generic algorithms to specify number-like behavior. You can see examples in the numeric types in the .NET runtime, such as System.Numerics.INumber<TSelf>. These interfaces define common mathematical operators that are implemented by many numeric types. The compiler must resolve calls to `static virtual` and `static abstract` methods at compile time. The `static virtual` and `static abstract` methods declared in interfaces don't have a runtime dispatch mechanism analogous to `virtual` or `abstract` methods declared in classes. Instead, the compiler uses type information available at compile time. Therefore, `static virtual` methods are almost exclusively declared in generic interfaces. Furthermore, most interfaces that declare `static virtual` or `static abstract` methods declare that one of the type parameters must implement the declared interface. For example, the `INumber<T>` interface declares that

`T` must implement `INumber<T>`. The compiler uses the type argument to resolve calls to the methods and operators declared in the interface declaration. For example, the `int` type implements `INumber<int>`. When the type parameter `T` denotes the type argument `int`, the `static` members declared on `int` are invoked. Alternatively, when `double` is the type argument, the `static` members declared on the `double` type are invoked.

ⓘ **Important**

Method dispatch for `static abstract` and `static virtual` methods declared in interfaces is resolved using the compile time type of an expression. If the runtime type of an expression is derived from a different compile time type, the static methods on the base (compile time) type will be called.

You can try this feature by working with the tutorial on static abstract members in interfaces.

Interface inheritance

Interfaces may not contain instance state. While static fields are now permitted, instance fields aren't permitted in interfaces. Instance auto-properties aren't supported in interfaces, as they would implicitly declare a hidden field. This rule has a subtle effect on property declarations. In an interface declaration, the following code doesn't declare an auto-implemented property as it does in a `class` or `struct`. Instead, it declares a property that doesn't have a default implementation but must be implemented in any type that implements the interface:

```C#
public interface INamed
{
  public string Name {get; set;}
}
```

An interface can inherit from one or more base interfaces. When an interface overrides a method implemented in a base interface, it must use the explicit interface implementation syntax.

When a base type list contains a base class and interfaces, the base class must come first in the list.

A class that implements an interface can explicitly implement members of that interface. An explicitly implemented member can't be accessed through a class instance, but only through an instance of the interface. In addition, default interface members can only be accessed through an instance of the interface.

For more information about explicit interface implementation, see Explicit Interface Implementation.

Example interface implementation

The following example demonstrates interface implementation. In this example, the interface contains the property declaration and the class contains the implementation. Any instance of a class that implements `IPoint` has integer properties `x` and `y`.

C#

```csharp
interface IPoint
{
    // Property signatures:
    int X { get; set; }

    int Y { get; set; }

    double Distance { get; }
}

class Point : IPoint
{
    // Constructor:
    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }

    // Property implementation:
    public int X { get; set; }

    public int Y { get; set; }

    // Property implementation
    public double Distance =>
       Math.Sqrt(X * X + Y * Y);
}

class MainClass
{
    static void PrintPoint(IPoint p)
    {
        Console.WriteLine("x={0}, y={1}", p.X, p.Y);
    }

    static void Main()
    {
        IPoint p = new Point(2, 3);
        Console.Write("My Point: ");
        PrintPoint(p);
    }
}
// Output: My Point: x=2, y=3
```

C# language specification

For more information, see the Interfaces section of the C# language specification, the feature specification for C# 8 - Default interface members, and the feature spec for C# 11 - static abstract members in interfaces

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- Reference Types
- Interfaces
- Using Properties
- Using Indexers