Contents

Interfaces

- Explicit Interface Implementation
- How to: Explicitly Implement Interface Members
- How to: Explicitly Implement Members of Two Interfaces

Interfaces (C# Programming Guide)

3/20/2019 • 4 minutes to read • Edit Online

An interface contains definitions for a group of related functionalities that a class or a struct can implement.

By using interfaces, you can, for example, include behavior from multiple sources in a class. That capability is important in C# because the language doesn't support multiple inheritance of classes. In addition, you must use an interface if you want to simulate inheritance for structs, because they can't actually inherit from another struct or class.

You define an interface by using the interface keyword. as the following example shows.

```
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

The name of the struct must be a valid C# identifier name. By convention, interface names begin with a capital `I`.

Any class or struct that implements the IEquatable<T> interface must contain a definition for an Equals method that matches the signature that the interface specifies. As a result, you can count on a class that implements `IEquatable<T>` to contain an `Equals` method with which an instance of the class can determine whether it's equal to another instance of the same class.

The definition of `IEquatable<T>` doesn't provide an implementation for `Equals`. The interface defines only the signature. In that way, an interface in C# is similar to an abstract class in which all the methods are abstract. However, a class or struct can implement multiple interfaces, but a class can inherit only a single class, abstract or not.

For more information about abstract classes, see Abstract and Sealed Classes and Class Members.

Interfaces can contain methods, properties, events, indexers, or any combination of those four member types. For links to examples, see Related Sections. An interface can't contain constants, fields, operators, instance constructors, finalizers, or types. Interface members are automatically public, and they can't include any access modifiers. Members also can't be static.

To implement an interface member, the corresponding member of the implementing class must be public, non-static, and have the same name and signature as the interface member.

When a class or struct implements an interface, the class or struct must provide an implementation for all of the members that the interface defines. The interface itself provides no functionality that a class or struct can inherit in the way that it can inherit base class functionality. However, if a base class implements an interface, any class that's derived from the base class inherits that implementation.

The following example shows an implementation of the IEquatable<T> interface. The implementing class, `Car`, must provide an implementation of the Equals method.

```
public class Car : IEquatable<Car>
{
    public string Make {get; set;}
    public string Model { get; set; }
    public string Year { get; set; }

    // Implementation of IEquatable<T> interface
    public bool Equals(Car car)
    {
        return this.Make == car.Make &&
               this.Model == car.Model &&
               this.Year == car.Year;
    }
}
```

Properties and indexers of a class can define extra accessors for a property or indexer that's defined in an interface. For example, an interface might declare a property that has a get accessor. The class that implements the interface can declare the same property with both a `get` and set accessor. However, if the property or indexer uses explicit implementation, the accessors must match. For more information about explicit implementation, see Explicit Interface Implementation and Interface Properties.

Interfaces can inherit from other interfaces. A class might include an interface multiple times through base classes that it inherits or through interfaces that other interfaces inherit. However, the class can provide an implementation of an interface only one time and only if the class declares the interface as part of the definition of the class (`class ClassName : InterfaceName`). If the interface is inherited because you inherited a base class that implements the interface, the base class provides the implementation of the members of the interface. However, the derived class can reimplement any virtual interface members instead of using the inherited implementation.

A base class can also implement interface members by using virtual members. In that case, a derived class can change the interface behavior by overriding the virtual members. For more information about virtual members, see Polymorphism.

Interfaces summary

An interface has the following properties:

- An interface is like an abstract base class with only abstract members. Any class or struct that implements the interface must implement all its members.
- An interface can't be instantiated directly. Its members are implemented by any class or struct that implements the interface.
- Interfaces can contain events, indexers, methods, and properties.
- Interfaces contain no implementation of methods.
- A class or struct can implement multiple interfaces. A class can inherit a base class and also implement one or more interfaces.

In this section

Explicit Interface Implementation
Explains how to create a class member that's specific to an interface.

How to: Explicitly Implement Interface Members
Provides an example of how to explicitly implement members of interfaces.

How to: Explicitly Implement Members of Two Interfaces
Provides an example of how to explicitly implement members of interfaces with inheritance.

Related Sections

- Interface Properties
- Indexers in Interfaces
- How to: Implement Interface Events
- Classes and Structs
- Inheritance
- Methods
- Polymorphism
- Abstract and Sealed Classes and Class Members
- Properties
- Events
- Indexers

featured book chapter

Interfaces in Learning C# 3.0: Master the Fundamentals of C# 3.0

See also

- C# Programming Guide
- Inheritance
- Identifier names

Explicit Interface Implementation (C# Programming Guide)

3/1/2019 • 2 minutes to read • Edit Online

If a class implements two interfaces that contain a member with the same signature, then implementing that member on the class will cause both interfaces to use that member as their implementation. In the following example, all the calls to `Paint` invoke the same method.

```
class Test
{
    static void Main()
    {
        SampleClass sc = new SampleClass();
        IControl ctrl = sc;
        ISurface srfc = sc;

        // The following lines all call the same method.
        sc.Paint();
        ctrl.Paint();
        srfc.Paint();
    }
}


interface IControl
{
    void Paint();
}
interface ISurface
{
    void Paint();
}
class SampleClass : IControl, ISurface
{
    // Both ISurface.Paint and IControl.Paint call this method.
    public void Paint()
    {
        Console.WriteLine("Paint method in SampleClass");
    }
}

// Output:
// Paint method in SampleClass
// Paint method in SampleClass
// Paint method in SampleClass
```

If the two interface members do not perform the same function, however, this can lead to an incorrect implementation of one or both of the interfaces. It is possible to implement an interface member explicitly—creating a class member that is only called through the interface, and is specific to that interface. This is accomplished by naming the class member with the name of the interface and a period. For example:

```
public class SampleClass : IControl, ISurface
{
    void IControl.Paint()
    {
        System.Console.WriteLine("IControl.Paint");
    }
    void ISurface.Paint()
    {
        System.Console.WriteLine("ISurface.Paint");
    }
}
```

The class member `IControl.Paint` is only available through the `IControl` interface, and `ISurface.Paint` is only available through `ISurface`. Both method implementations are separate, and neither is available directly on the class. For example:

```
// Call the Paint methods from Main.

SampleClass obj = new SampleClass();
//obj.Paint();  // Compiler error.

IControl c = obj;
c.Paint();  // Calls IControl.Paint on SampleClass.

ISurface s = obj;
s.Paint(); // Calls ISurface.Paint on SampleClass.

// Output:
// IControl.Paint
// ISurface.Paint
```

Explicit implementation is also used to resolve cases where two interfaces each declare different members of the same name such as a property and a method:

```
interface ILeft
{
    int P { get;}
}
interface IRight
{
    int P();
}
```

To implement both interfaces, a class has to use explicit implementation either for the property P, or the method P, or both, to avoid a compiler error. For example:

```
class Middle : ILeft, IRight
{
    public int P() { return 0; }
    int ILeft.P { get { return 0; } }
}
```

See also

- C# Programming Guide
- Classes and Structs
- Interfaces

- Inheritance

How to: Explicitly Implement Interface Members (C# Programming Guide)

4/28/2019 • 2 minutes to read • Edit Online

This example declares an interface, `IDimensions`, and a class, `Box`, which explicitly implements the interface members `getLength` and `getWidth`. The members are accessed through the interface instance `dimensions`.

Example

```
interface IDimensions
{
    float getLength();
    float getWidth();
}

class Box : IDimensions
{
    float lengthInches;
    float widthInches;

    Box(float length, float width)
    {
        lengthInches = length;
        widthInches = width;
    }
    // Explicit interface member implementation:
    float IDimensions.getLength()
    {
        return lengthInches;
    }
    // Explicit interface member implementation:
    float IDimensions.getWidth()
    {
        return widthInches;
    }

    static void Main()
    {
        // Declare a class instance box1:
        Box box1 = new Box(30.0f, 20.0f);

        // Declare an interface instance dimensions:
        IDimensions dimensions = box1;

        // The following commented lines would produce compilation
        // errors because they try to access an explicitly implemented
        // interface member from a class instance:
        //System.Console.WriteLine("Length: {0}", box1.getLength());
        //System.Console.WriteLine("Width: {0}", box1.getWidth());

        // Print out the dimensions of the box by calling the methods
        // from an instance of the interface:
        System.Console.WriteLine("Length: {0}", dimensions.getLength());
        System.Console.WriteLine("Width: {0}", dimensions.getWidth());
    }
}
/* Output:
    Length: 30
    Width: 20
*/
```

Robust Programming

- Notice that the following lines, in the `Main` method, are commented out because they would produce compilation errors. An interface member that is explicitly implemented cannot be accessed from a class instance:

  ```
  //System.Console.WriteLine("Length: {0}", box1.getLength());
  //System.Console.WriteLine("Width: {0}", box1.getWidth());
  ```

- Notice also that the following lines, in the `Main` method, successfully print out the dimensions of the box because the methods are being called from an instance of the interface:

```
System.Console.WriteLine("Length: {0}", dimensions.getLength());
System.Console.WriteLine("Width: {0}", dimensions.getWidth());
```

See also

- C# Programming Guide
- Classes and Structs
- Interfaces
- How to: Explicitly Implement Members of Two Interfaces

How to: Explicitly Implement Members of Two Interfaces (C# Programming Guide)

3/1/2019 • 2 minutes to read • Edit Online

Explicit interface implementation also allows the programmer to implement two interfaces that have the same member names and give each interface member a separate implementation. This example displays the dimensions of a box in both metric and English units. The Box class implements two interfaces IEnglishDimensions and IMetricDimensions, which represent the different measurement systems. Both interfaces have identical member names, Length and Width.

Example

```
// Declare the English units interface:
interface IEnglishDimensions
{
    float Length();
    float Width();
}

// Declare the metric units interface:
interface IMetricDimensions
{
    float Length();
    float Width();
}

// Declare the Box class that implements the two interfaces:
// IEnglishDimensions and IMetricDimensions:
class Box : IEnglishDimensions, IMetricDimensions
{
    float lengthInches;
    float widthInches;

    public Box(float lengthInches, float widthInches)
    {
        this.lengthInches = lengthInches;
        this.widthInches = widthInches;
    }

    // Explicitly implement the members of IEnglishDimensions:
    float IEnglishDimensions.Length() => lengthInches;

    float IEnglishDimensions.Width() => widthInches;

    // Explicitly implement the members of IMetricDimensions:
    float IMetricDimensions.Length() => lengthInches * 2.54f;

    float IMetricDimensions.Width() => widthInches * 2.54f;

    static void Main()
    {
        // Declare a class instance box1:
        Box box1 = new Box(30.0f, 20.0f);

        // Declare an instance of the English units interface:
        IEnglishDimensions eDimensions = box1;

        // Declare an instance of the metric units interface:
        IMetricDimensions mDimensions = box1;

        // Print dimensions in English units:
        System.Console.WriteLine("Length(in): {0}", eDimensions.Length());
        System.Console.WriteLine("Width (in): {0}", eDimensions.Width());

        // Print dimensions in metric units:
        System.Console.WriteLine("Length(cm): {0}", mDimensions.Length());
        System.Console.WriteLine("Width (cm): {0}", mDimensions.Width());
    }
}
/* Output:
    Length(in): 30
    Width (in): 20
    Length(cm): 76.2
    Width (cm): 50.8
*/
```

Robust Programming

If you want to make the default measurements in English units, implement the methods Length and Width normally, and explicitly implement the Length and Width methods from the IMetricDimensions interface:

```
// Normal implementation:
public float Length() => lengthInches;
public float Width() => widthInches;

// Explicit implementation:
float IMetricDimensions.Length() => lengthInches * 2.54f;
float IMetricDimensions.Width() => widthInches * 2.54f;
```

In this case, you can access the English units from the class instance and access the metric units from the interface instance:

```
public static void Test()
{
    Box box1 = new Box(30.0f, 20.0f);
    IMetricDimensions mDimensions = box1;

    System.Console.WriteLine("Length(in): {0}", box1.Length());
    System.Console.WriteLine("Width (in): {0}", box1.Width());
    System.Console.WriteLine("Length(cm): {0}", mDimensions.Length());
    System.Console.WriteLine("Width (cm): {0}", mDimensions.Width());
}
```

See also

- C# Programming Guide
- Classes and Structs
- Interfaces
- How to: Explicitly Implement Interface Members