Interface type

To begin this chapter, we provide a formal definition of the interface type. An interface is nothing more than a named set of abstract members. Recall that abstract methods are pure protocol in that they do not provide a default implementation. The specific members defined by an interface depend on the exact behavior it is modeling. Said another way, an ***interface*** *expresses a behavior that a given class or structure may choose to support*. Furthermore, as you will, *a class or structure can support as many interfaces as necessary, thereby supporting (in essence) multiple behaviors.*

By convention, .net interfaces are prefixed with a capital letter I. When you are creating your own custom interfaces, it is considered a best practice to do the same. At a syntactic level, an interface is defined using the C# interface keyword.

So, in another words interface is set of method signature. Interface is a certificate.

Interface Types vs. Abstract Base Classes

When a class is marked as abstract, it may define any number of abstract members to provide a polymorphic interface to all derived types. However, even when a class does define a set of abstract members, it is also free to define any number of constructors, field data, nonabstract members (with implementation), and so on. *Interfaces, on the other hand, contain only member definitions.*

The polymorphic interface established by an abstract parent class suffers from one major limitation in that only derived types support the members defined by the abstract parent. However, in larger software systems, it is common to develop multiple class hierarchies that have no common parent beyond System. Object. Given that abstract members in an abstract base class apply only to derived types, you have no way to configure types in different hierarchies to support the same polymorphic interface. *Interfaces can be implemented by any class or structure, in any hierarchy, and within any namespace or any assembly (written in any .NET programming language). As you see, interfaces are highly polymorphic. For example, c*onsider the standard .NET interface named ICloneable, defined in the System namespace. This interface defines a single method named Clone().

```
public interface ICloneable
{
        object Clone();
}
```

So, a large number of seemingly unrelated types (System.Array, System.Data.SqlClient.SqlConnection, System. OperatingSystem, System.String, etc.) all implement this interface. Although these types have no common parent (other than System.Object), you can treat them polymorphically via the ICloneable interface type. Another example, if you had a method named CloneMe() that took an ICloneable interface parameter, you could pass this method any object that implements said interface.

Another limitation of abstract base classes is that each derived type must contend with the set of abstract members and provide an implementation. Unlike a class, *interfaces never specify a base class (not even System.Object; however, an interface can specify base interfaces).* Remember that when you define interface members, you do not define an implementation scope for the members in question.

Interfaces are pure protocol and, therefore, never define an implementation (that is up to the supporting class or structure).

.NET interface types are able to define any number of properties, methods, events and indexers.

Implementation of interfaces

When a class (or structure) chooses to extend its functionality by supporting interfaces, it does so using a comma-delimited list in the type definition. Be aware that the direct base class must be the first item listed after the colon operator. When your class type derives directly from System.Object, you are free to simply list the interface (or interfaces) supported by the class, as the C# compiler will extend your types from System.Object if you do not say otherwise. On a related note, given that structures always derive from System.ValueType, simply list each interface directly after the structure definition.

```
// This class also derives from System.Object
// and implements a single interface.
public class SwitchBlade : object, IPointy
{...}

// This class derives from a custom base class
// and implements a single interface.
public class Fork : Utensil, IPointy
{...}

// This struct implicitly derives from System.ValueType and
// implements two interfaces.
public struct PitchFork : ICloneable, IPointy {...}
```

We should understand that implementing an interface is an all-or-nothing proposition. The supporting type is not able to selectively choose which members it will implement. So, it should implement all the members the interface has.

When we have some classes that support the current interface, the next question is how we interact with the new functionality. The most straightforward way to interact with functionality supplied by a given interface is to invoke the members directly from the object level, but how can we dynamically determine whether a class or structure supports the correct interface? We can do this by as or is keywords without using try/catch blocks. In case of as keyword, if the object can be treated as the specified interface, you are returned a reference to the interface in question. If not, you receive a null reference. Therefore, be sure to check against a null value before proceeding. In case of is keyword, if the object in question is not compatible with the specified interface, you are returned the value false. On the other hand, if the type is compatible with the interface in question, you can safely call the members without needing to use try/catch logic.

Related to implementation of interfaces

Interfaces can be used as methods' *input parameters* and also be used as *method return values*.

Recall that the same interface can be implemented by numerous types, even if they are not within the same class hierarchy and do not have a common parent class beyond System.Object.

We just should remember this: *when you have an array of a given interface, the array can contain any class or structure that implements that interface.*

Given that interfaces are a named set of abstract members, you are required to type in the definition and implementation for each interface method on each type that supports the behavior. Therefore, if you want to support an interface that defines a total of five methods and three properties, you need to account for all eight members (or else you will receive compiler errors).

Explicit Interface Implementation

There is always the possibility you might implement interfaces that contain identical members. If you want to support each of these interfaces on a single class type named Octagon, the compiler would allow the following definition and the code will invoke the same method, regardless of which interface you obtain, also the code will compile cleanly.

```
public interface IDrawToForm
{
        void Draw();
}
public interface IDrawToMemory
{
        void Draw();
}
public interface IDrawToPrinter
{
        void Draw();
}
class Octagon : IDrawToForm, IDrawToMemory, IDrawToPrinter
{
        public void Draw()
        {
                // Shared drawing logic.
                Console.WriteLine("Drawing the Octagon...");
        }
}
```

```
class Program
{
        static void Main(string[] args)
        {
                Octagon o = new Octagon();
                o.Draw();

                IDrawToForm fo = new Octagon();
                fo.Draw();

                IDrawToMemory mo = (IDrawToMemory)o;
                mo.Draw();

                IDrawToPrinter po = (IDrawToPrinter)o;
                po.Draw();
        }
}
//  Drawing the Octagon...
//  Drawing the Octagon...
//  Drawing the Octagon...
//  Drawing the Octagon...
```

When you implement several interfaces that have identical members, you can resolve this sort of name clash using *explicit interface implementation* syntax.

```
class Octagon : IDrawToForm, IDrawToMemory, IDrawToPrinter
{
        // Explicitly bind Draw() implementations
        // to a given interface.
        void IDrawToForm.Draw()
        {
                Console.WriteLine("Drawing to form...");
        }
        void IDrawToMemory.Draw()
        {
                Console.WriteLine("Drawing to memory...");
        }
        void IDrawToPrinter.Draw()
        {
                Console.WriteLine("Drawing to a printer...");
        }
}
```

Note that when using this syntax, you do not supply an access modifier; explicitly implemented members are automatically private. ***Because explicitly implemented members are always implicitly private, these members are no longer available from the object level.***

In fact, if you were to apply the dot operator to an Octagon type, you would find that IntelliSense does not show you any of the Draw() members. As expected, you must use explicit casting to access the required functionality. Here's an example:

```
static void Main(string[] args)
{
        Octagon oct = new Octagon();

        // We now must use casting to access the Draw() members.
        IDrawToForm itfForm = (IDrawToForm)oct;
        itfForm.Draw();

        // Shorthand notation if you don't need the interface variable for later use.
        ((IDrawToPrinter)oct).Draw();

        // Could also use the "is" keyword.
        if (oct is IDrawToMemory dtm)
                dtm.Draw();

        IDrawToPrinter pr = oct;
        pr.Draw();
}
// Drawing to form...
// Drawing to a printer...
// Drawing to memory...
// Drawing to a printer...
```

While this syntax is quite helpful when you need to resolve name clashes, you can use explicit interface implementation simply to hide more “advanced” members from the object level. In this way, when the object user applies the dot operator, the user will see only a subset of the type’s overall functionality.

Interface Hierarchies

Like a class hierarchy, when an interface extends an existing interface, it inherits the abstract members defined by the parent (or parents). Of course, unlike class-based inheritance, derived interfaces never inherit true implementation. Rather, a derived interface simply extends its own definition with additional abstract members. Interface hierarchies can be useful when we want to extend the functionality of an existing interface without breaking existing codebases. When the class derives from the last interface in the hierarchy, it would be required to implement every member defined up the chain of inheritance. After this, we are able to invoke each method at the object level.

However, interfaces are a fundamental aspect of the .NET Framework. Regardless of the type of application you are developing (web-based, desktop GUIs, data-access libraries, etc.), working with interfaces will be part of the process. To summarize the story thus far, remember that interfaces can be extremely useful in the following cases:

- We have a single hierarchy where only a subset of the derived types supports a common behavior.
- We need to model a common behavior that is found across multiple hierarchies with no common parent class beyond System.Object.

IEnumerable and IEnumerator Interfaces

IEnumerable is the base interface for all non-generic collections that can be enumerated. This interface exposes an enumerator, which supports a simple iteration over a non-generic collection. IEnumerable contains a single method, *GetEnumerator*, which returns an IEnumerator.

```
public interface IEnumerable
{
        IEnumerator GetEnumerator();
}
```

IEnumerator is the base interface for all non-generic enumerators. **IEnumerator** provides the ability to iterate through the collection by exposing a *Current* property and *MoveNext* and *Reset* methods.

The foreach statement of the C# language hides the complexity of the enumerators. Therefore*, using foreach is recommended instead of directly manipulating the enumerator*. Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

The **Reset** method is provided for COM interoperability and does not need to be fully implemented; instead, the implementer can throw a NotSupportedException.

Initially, the enumerator is positioned before the first element in the collection. You must call the **MoveNext** method to advance the enumerator to the first element of the collection before reading the value of Current; otherwise, Current is undefined.

Current returns the same object until either MoveNext or Reset is called. MoveNext sets Current to the next element.

If MoveNext passes the end of the collection, the enumerator is positioned after the last element in the collection and MoveNext returns false. When the enumerator is at this position, subsequent calls to MoveNext also return false. If the last call to MoveNextreturned false, Current is undefined. To set Current to the first element of the collection again, you can call Reset, if it's implemented, followed by MoveNext. If Reset is not implemented, you must create a new enumerator instance to return to the first element of the collection. If changes are made to the collection, such as adding, modifying, or deleting elements, the behavior of the enumerator is undefined.

```
public interface IEnumerator
{
        object Current { get; }
        bool MoveNext();
        void Reset();
}
```

It is a best practice to implement IEnumerable and IEnumerator on our collection classes to enable the foreach syntax, however implementing IEnumerable is not required. If our collection does not implement IEnumerable, we must still follow the iterator pattern to support this syntax by providing a GetEnumerator method that returns an interface, class or struct. So, any type supporting a method named GetEnumerator() can be evaluated by the foreach construct. When developing with C# we must provide a class that contains a *Current* property, and *MoveNext* and *Reset* methods as described by IEnumerator, but the class does not have to implement IEnumerator.

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. Even when a collection is synchronized, other threads can still modify the collection, which causes the enumerator to throw an exception. To guarantee

thread safety during enumeration, you can either lock the collection during the entire enumeration or catch the exceptions resulting from changes made by other threads.

By this link you can find MyCollection class where I have implemented IEnumerable and IEnumerator interfaces: https://github.com/HakobyanAni/C-Sharp-Advanced/tree/master/MyCollection