CHAPTER 13

Interfaces

In this chapter:

Many programmers are familiar with the concept of multiple inheritance: the ability to define a class that is derived from two or more base classes. For example, imagine a class named `TransmitData`, whose function is to transmit data, and another class named `ReceiveData`, whose function is to receive data. Now imagine that you want to create a class named `SocketPort`, whose function is to transmit and receive data. In order to accomplish this, you would want to derive `SocketPort` from both `TransmitData` and `ReceiveData`.

Some programming languages allow multiple inheritance, making it possible for the `SocketPort` class to be derived from the two base classes, `TransmitData` and `ReceiveData`. However, the common language runtime (CLR)—and therefore all managed programming languages—does not support multiple inheritance. Rather than not offer any kind of multiple inheritance at all, the CLR does offer scaled-down multiple inheritance via *interfaces*. This chapter will discuss how to define and use interfaces as well as provide some guidelines to help you determine when to use an interface rather than a base class.

Class and Interface Inheritance

In the Microsoft .NET Framework, there is a class called `System.Object` that defines four public instance methods: `ToString`, `Equals`, `GetHashCode`, and `GetType`. This class is the root or ultimate base class of all other classes—all classes will inherit `Object`'s four instance methods. This also means that code written to operate on an instance of the `Object` class can actually perform operations on an instance of any class.

Because someone at Microsoft has implemented `Object`'s methods, any class derived from `Object` is actually inheriting the following:

- **The method signatures** This allows code to think that it is operating on an instance of the `Object` class, when in fact, it could be operating on an instance of some other class.
- **The implementation of these methods** This allows the developer defining a class derived from `Object` not to be required to manually implement `Object`'s methods.

In the CLR, a class is always derived from one and only one class (that must ultimately be derived from `Object`). This base class provides a set of method signatures and implementations for these methods. And a cool thing about defining a new class is that it can become the base class for another class defined in the future by some other developer—all of the method signatures and their implementations will be inherited by the new derived class.

The CLR also allows developers to define an *interface*, which is really just a way to give a name to a set of method signatures. These methods do not come with any implementation at all. A class inherits an interface by specifying the interface's name, and the class must explicitly provide implementations of the interface's methods before the CLR will consider the type definition to be valid. Of course, implementing interface methods can be tedious, which is why I referred to interface inheritance as a scaled-down mechanism to achieve multiple inheritance. The C# compiler and the CLR actually allow a class to inherit several interfaces, and of course, the class must provide implementations for all of the inherited interface methods.

One of the great features of class inheritance is that it allows instances of a derived type to be substituted in all contexts that expect instances of a base type. Similarly, interface inheritance allows instances of a type that implements the interface to be substituted in all contexts that expect instances of the named interface type. We will now look at how to define interfaces to make our discussion more concrete.

Defining an Interface

As mentioned in the previous section, an interface is a named set of method signatures. Note that interfaces can also define events, parameterless properties, and parameterful properties (indexers in C#) because all of these are just syntax shorthands that map to methods anyway, as shown in previous chapters. However, an interface cannot define any constructor methods. In addition, an interface is not allowed to define any instance fields.

Although the CLR does allow an interface to define static methods, static fields, constants, and static constructors, a Common Language Specification (CLS)–compliant interface must not have any of these static members because some programming languages aren't able to define or access them. In fact, C# prevents an interface from defining any of these static members.

In C#, you use the `interface` keyword to define an interface, giving it a name and its set of instance method signatures. Here are the definitions of a few interfaces defined in the Framework Class Library (FCL).

```
public interface IDisposable {
   void Dispose();
}

public interface IEnumerable {
   IEnumerator GetEnumerator();
}

public interface IEnumerable<out T> : IEnumerable {
   new IEnumerator<T> GetEnumerator();
}

public interface ICollection<T> : IEnumerable<T>, IEnumerable {
   void    Add(T item);
   void    Clear();
   Boolean Contains(T item);
   void    CopyTo(T[] array, Int32 arrayIndex);
   Boolean Remove(T item);
   Int32   Count      { get; } // Read-only property
   Boolean IsReadOnly { get; } // Read-only property
}
```

To the CLR, an interface definition is just like a type definition. That is, the CLR will define an internal data structure for the interface type object, and reflection can be used to query features of the interface type. Like types, an interface can be defined at file scope or defined nested within another type. When defining the interface type, you can specify whatever visibility/accessibility (`public`, `protected`, `internal`, etc.) you want.

By convention, interface type names are prefixed with an uppercase `I`, making it easy to spot an interface type in source code. The CLR does support generic interfaces (as you can see from some of the previous examples) as well as generic methods in an interface. I will discuss some of the many features offered by generic interfaces later in this chapter and in Chapter 12, "Generics," in which I cover generics more broadly.

An interface definition can "inherit" other interfaces. However, I use the word *inherit* here rather loosely because interface inheritance doesn't work exactly like class inheritance. I prefer to think of interface inheritance as including the contract of other interfaces. For example, the `ICollection<T>` interface definition includes the contracts of the `IEnumerable<T>` and `IEnumerable` interfaces.

This means that:

- Any class that inherits the `ICollection<T>` interface must implement all of the methods defined by the `ICollection<T>`, `IEnumerable<T>`, and `IEnumerable` interfaces.
- Any code that expects an object whose type implements the `ICollection<T>` interface can assume that the object's type also implements the methods of the `IEnumerable<T>` and `IEnumerable` interfaces.

Inheriting an Interface

In this section, I'll show how to define a type that implements an interface, and then I'll show how to create an instance of this type and use the object to call the interface's methods. C# actually makes this pretty simple, but what happens behind the scenes is a bit more complicated. I'll explain what is happening behind the scenes later in this chapter.

The `System.IComparable<T>` interface is defined (in MSCorLib.dll) as follows.

```
public interface IComparable<in T> {
   Int32 CompareTo(T other);
}
```

The following code shows how to define a type that implements this interface and also shows code that compares two `Point` objects.

```
using System;


// Point is derived from System.Object and implements IComparable<T> for Point.
public sealed class Point : IComparable<Point> {
   private Int32 m_x, m_y;

   public Point(Int32 x, Int32 y) {
      m_x = x;
      m_y = y;
   }

   // This method implements IComparable<T>.CompareTo() for Point
   public Int32 CompareTo(Point other) {
      return Math.Sign(Math.Sqrt(m_x * m_x + m_y * m_y)
         - Math.Sqrt(other.m_x * other.m_x + other.m_y * other.m_y));
   }

   public override String ToString() {
      return String.Format("({0}, {1})", m_x, m_y);
   }
}
```

```
public static class Program {
   public static void Main() {
      Point[] points = new Point[] {
         new Point(3, 3),
         new Point(1, 2),
      };

      // Here is a call to Point's IComparable<T> CompareTo method
      if (points[0].CompareTo(points[1]) > 0) {
         Point tempPoint = points[0];
         points[0] = points[1];
         points[1] = tempPoint;
      }
      Console.WriteLine("Points from closest to (0, 0) to farthest:");
      foreach (Point p in points)
         Console.WriteLine(p);
   }
}
```

The C# compiler requires that a method that implements an interface be marked as public. The CLR requires that interface methods be marked as virtual. If you do not explicitly mark the method as virtual in your source code, the compiler marks the method as virtual and sealed; this prevents a derived class from overriding the interface method. If you explicitly mark the method as virtual, the compiler marks the method as virtual (and leaves it unsealed); this allows a derived class to override the interface method.

If an interface method is sealed, a derived class cannot override the method. However, a derived class can re-inherit the same interface and can provide its own implementation for the interface's methods. When calling an interface's method on an object, the implementation associated with the object's type is called. Here is an example that demonstrates this.

```
using System;

public static class Program {
   public static void Main() {
      /************************ First Example ************************/
      Base b = new Base();

      // Calls Dispose by using b's type: "Base's Dispose"
      b.Dispose();

      // Calls Dispose by using b's object's type: "Base's Dispose"
      ((IDisposable)b).Dispose();


      /************************ Second Example ************************/
      Derived d = new Derived();

      // Calls Dispose by using d's type: "Derived's Dispose"
      d.Dispose();

      // Calls Dispose by using d's object's type: "Derived's Dispose"
      ((IDisposable)d).Dispose();
```

```
      /************************ Third Example ************************/
      b = new Derived();

      // Calls Dispose by using b's type: "Base's Dispose"
      b.Dispose();

      // Calls Dispose by using b's object's type: "Derived's Dispose"
      ((IDisposable)b).Dispose();
   }
}

// This class is derived from Object and it implements IDisposable
internal class Base : IDisposable {
   // This method is implicitly sealed and cannot be overridden
   public void Dispose() {
      Console.WriteLine("Base's Dispose");
   }
}

// This class is derived from Base and it re-implements IDisposable
internal class Derived : Base, IDisposable {
   // This method cannot override Base's Dispose. 'new' is used to indicate
   // that this method re-implements IDisposable's Dispose method
   new public void Dispose() {
      Console.WriteLine("Derived's Dispose");

      // NOTE: The next line shows how to call a base class's implementation (if desired)
      // base.Dispose();
   }
}
```

More About Calling Interface Methods

The FCL's `System.String` type inherits `System.Object`'s method signatures and their implementations. In addition, the `String` type also implements several interfaces: `IComparable`, `ICloneable`, `IConvertible`, `IEnumerable`, `IComparable<String>`, `IEnumerable<Char>`, and `IEquatable<String>`. This means that the `String` type isn't required to implement (or override) the methods its `Object` base type offers. However, the `String` type must implement the methods declared in all of the interfaces.

The CLR allows you to define field, parameter, or local variables that are of an interface type. Using a variable of an interface type allows you to call methods defined by that interface. In addition, the CLR will allow you to call methods defined by `Object` because all classes inherit `Object`'s methods. The following code demonstrates this.

```
// The s variable refers to a String object.
String s = "Jeffrey";
// Using s, I can call any method defined in
// String, Object, IComparable, ICloneable, IConvertible, IEnumerable, etc.

// The cloneable variable refers to the same String object
ICloneable cloneable = s;
```

```
// Using cloneable, I can call any method declared by the
// ICloneable interface (or any method defined by Object) only.

// The comparable variable refers to the same String object
IComparable comparable = s;
// Using comparable, I can call any method declared by the
// IComparable interface (or any method defined by Object) only.

// The enumerable variable refers to the same String object
// At run time, you can cast a variable from one interface to another as
// long as the object's type implements both interfaces.
IEnumerable enumerable = (IEnumerable) comparable;
// Using enumerable, I can call any method declared by the
// IEnumerable interface (or any method defined by Object) only.
```

In this code, all of the variables refer to the same "Jeffrey" `String` object that is in the managed heap, and therefore, any method that I call while using any of these variables applies to the one "Jeffrey" `String` object. However, the type of the variable indicates the action that I can perform on the object. The `s` variable is of type `String`, and therefore, I can use `s` to call any members defined by the `String` type (such as the `Length` property). I can also use the variable `s` to call any methods inherited from `Object` (such as `GetType`).

The `cloneable` variable is of the `ICloneable` interface type, and therefore, using the `cloneable` variable, I can call the `Clone` method defined by this interface. In addition, I can call any method defined by `Object` (such as `GetType`) because the CLR knows that all types derive from `Object`. However, using the `cloneable` variable, I cannot call public methods defined by `String` itself or any methods defined by any other interface that `String` implements. Similarly, using the `comparable` variable, I can call `CompareTo` or any method defined by `Object`, but no other methods are callable using this variable.

Important Like a reference type, a value type can implement zero or more interfaces. However, when you cast an instance of a value type to an interface type, the value type instance must be boxed. This is because an interface variable is a reference that must point to an object on the heap so that the CLR can examine the object's type object pointer to determine the exact type of the object. Then, when calling an interface method with a boxed value type, the CLR will follow the object's type object pointer to find the type object's method table in order to call the proper method.

Implicit and Explicit Interface Method Implementations (What's Happening Behind the Scenes)

When a type is loaded into the CLR, a method table is created and initialized for the type (as discussed in Chapter 1, "The CLR's Execution Model"). This method table contains one entry for every new method introduced by the type as well as entries for any virtual methods inherited by the type.

Inherited virtual methods include methods defined by the base types in the inheritance hierarchy as well as any methods defined by the interface types. So if you have a simple type defined like this:

```
internal sealed class SimpleType : IDisposable {
   public void Dispose() { Console.WriteLine("Dispose"); }
}
```

the type's method table contains entries for the following:

- All the virtual instance methods defined by `Object`, the implicitly inherited base class.
- All the interface methods defined by `IDisposable`, the inherited interface. In this example, there is only one method, `Dispose`, because the `IDisposable` interface defines just one method.
- The new method, `Dispose`, introduced by `SimpleType`.

To make things simple for the programmer, the C# compiler assumes that the `Dispose` method introduced by `SimpleType` is the implementation for `IDisposable`'s `Dispose` method. The C# compiler makes this assumption because the method is `public`, and the signatures of the interface method and the newly introduced method are identical. That is, the methods have the same parameter and return types. By the way, if the new `Dispose` method were marked as `virtual`, the C# compiler would still consider this method to be a match for the interface method.

When the C# compiler matches a new method to an interface method, it emits metadata indicating that both entries in `SimpleType`'s method table should refer to the same implementation. To help make this clearer, here is some code that demonstrates how to call the class's public `Dispose` method as well as how to call the class's implementation of `IDisposable`'s `Dispose` method.

```
public sealed class Program {
   public static void Main() {
      SimpleType st = new SimpleType();

      // This calls the public Dispose method implementation
      st.Dispose();

      // This calls IDisposable's Dispose method implementation
      IDisposable d = st;
      d.Dispose();
   }
}
```

In the first call to `Dispose`, the `Dispose` method defined by `SimpleType` is called. Then I define a variable, `d`, which is of the `IDisposable` interface type. I initialize the `d` variable to refer to the `SimpleType` object. Now when I call `d.Dispose()`, I am calling the `IDisposable` interface's `Dispose` method. Because C# requires the public `Dispose` method to also be the implementation for `IDisposable`'s `Dispose` method, the same code will execute, and, in this example, you can't see any observable difference. The output is as follows.

```
Dispose
Dispose
```

Now, let me rewrite the preceding `SimpleType` so that you can see an observable difference.

```
internal sealed class SimpleType : IDisposable {
   public void Dispose() { Console.WriteLine("public Dispose"); }
   void IDisposable.Dispose() { Console.WriteLine("IDisposable Dispose"); }
}
```

Without changing the `Main` method shown earlier, if we just recompile and rerun the program, the output will be the following.

```
public Dispose
IDisposable Dispose
```

In C#, when you prefix the name of a method with the name of the interface that defines the method (`IDisposable.Dispose` as in this example), you are creating an *explicit interface method implementation* (EIMI). Note that when you define an explicit interface method in C#, you are not allowed to specify any accessibility (such as `public` or `private`). However, when the compiler generates the metadata for the method, its accessibility is set to private, preventing any code using an instance of the class from simply calling the interface method. The only way to call the interface method is through a variable of the interface's type.

Also note that an EIMI method cannot be marked as `virtual` and therefore cannot be overridden. This is because the EIMI method is not really part of the type's object model; it's a way of attaching an interface (set of behaviors or methods) onto a type without making the behaviors/methods obvious. If all of this seems a bit kludgy to you, you *are* understanding it correctly—this is all a bit kludgy. Later in this chapter, I'll show some valid reasons for using EIMIs.

Generic Interfaces

C#'s and the CLR's support of generic interfaces offers many great features for developers. In this section, I'd like to discuss the benefits offered when using generic interfaces.

First, generic interfaces offer great compile-time type safety. Some interfaces (such as the nongeneric `IComparable` interface) define methods that have `Object` parameters or return types. When code calls these interface methods, a reference to an instance of any type can be passed. But this is usually not desired. The following code demonstrates.

```
private void SomeMethod1() {
   Int32 x = 1, y = 2;
   IComparable c = x;

   // CompareTo expects an Object; passing y (an Int32) is OK
   c.CompareTo(y);     // y is boxed here

   // CompareTo expects an Object; passing "2" (a String) compiles
   // but an ArgumentException is thrown at runtime
   c.CompareTo("2");
}
```

Obviously, it is preferable to have the interface method strongly typed, and this is why the FCL includes a generic `IComparable<in T>` interface.

Here is the new version of the code revised by using the generic interface.

```
private void SomeMethod2() {
   Int32 x = 1, y = 2;
   IComparable<Int32> c = x;

   // CompareTo expects an Int32; passing y (an Int32) is OK
   c.CompareTo(y);     // y is not boxed here

   // CompareTo expects an Int32; passing "2" (a String) results
   // in a compiler error indicating that String cannot be cast to an Int32
   c.CompareTo("2");   // Error
}
```

The second benefit of generic interfaces is that much less boxing will occur when working with value types. Notice in `SomeMethod1` that the non-generic `IComparable` interface's `CompareTo` method expects an `Object`; passing y (an `Int32` value type) causes the value in y to be boxed. However, in `SomeMethod2`, the generic `IComparable<in T>` interface's `CompareTo` method expects an `Int32`; passing y causes it to be passed by value, and no boxing is necessary.

Note The FCL defines non-generic and generic versions of the `IComparable`, `ICollection`, `IList`, and `IDictionary` interfaces, as well as some others. If you are defining a type, and you want to implement any of these interfaces, you should typically implement the generic versions of these interfaces. The non-generic versions are in the FCL for backward compatibility to work with code written before the .NET Framework supported generics. The non-generic versions also provide users a way of manipulating the data in a more general, less type-safe fashion.

Some of the generic interfaces inherit the non-generic versions, so your class will have to implement both the generic and non-generic versions of the interfaces. For example, the generic `IEnumerable<out T>` interface inherits the non-generic `IEnumerable` interface. So if your class implements `IEnumerable<out T>`, your class must also implement `IEnumerable`.

Sometimes when integrating with other code, you may have to implement a non-generic interface because a generic version of the interface simply doesn't exist. In this case, if any of the interface's methods take or return `Object`, you will lose compile-time type safety, and you will get boxing with value types. You can alleviate this situation to some extent by using a technique I describe in the "Improving Compile-Time Type Safety with Explicit Interface Method Implementations" section near the end of this chapter.

The third benefit of generic interfaces is that a class can implement the same interface multiple times as long as different type parameters are used.

The following code shows an example of how useful this could be.

```
using System;

// This class implements the generic IComparable<T> interface twice
public sealed class Number: IComparable<Int32>, IComparable<String> {
   private Int32 m_val = 5;

   // This method implements IComparable<Int32>'s CompareTo
   public Int32 CompareTo(Int32 n) {
      return m_val.CompareTo(n);
   }

   // This method implements IComparable<String>'s CompareTo
   public Int32 CompareTo(String s) {
      return m_val.CompareTo(Int32.Parse(s));
   }
}

public static class Program {
   public static void Main() {
      Number n = new Number();

      // Here, I compare the value in n with an Int32 (5)
      IComparable<Int32> cInt32 = n;
      Int32 result = cInt32.CompareTo(5);

      // Here, I compare the value in n with a String ("5")
      IComparable<String> cString = n;
      result = cString.CompareTo("5");
   }
}
```

An interface's generic type parameters can also be marked as contra-variant and covariant, which allows even more flexibility for using generic interfaces. For more about contra-variance and covariance, see the "Delegate and Interface Contra-variant and Covariant Generic Type Arguments" section in Chapter 12.

Generics and Interface Constraints

In the previous section, I discussed the benefits of using generic interfaces. In this section, I'll discuss the benefits of constraining generic type parameters to interfaces.

The first benefit is that you can constrain a single generic type parameter to multiple interfaces. When you do this, the type of parameter you are passing in must implement *all* of the interface constraints.

Here is an example.

```
public static class SomeType {
   private static void Test() {
      Int32 x = 5;
      Guid g = new Guid();

      // This call to M compiles fine because
      // Int32 implements IComparable AND IConvertible
      M(x);

      // This call to M causes a compiler error because
      // Guid implements IComparable but it does not implement IConvertible
      M(g);
   }

   // M's type parameter, T, is constrained to work only with types that
   // implement both the IComparable AND IConvertible interfaces
   private static Int32 M<T>(T t) where T : IComparable, IConvertible {
      ...
   }
}
```

This is actually quite cool! When you define a method's parameters, each parameter's type indicates that the argument passed must be of the parameter's type or be derived from it. If the parameter type is an interface, this indicates that the argument can be of any class type as long as the class implements the interface. Using multiple interface constraints actually lets the method indicate that the passed argument must implement multiple interfaces.

In fact, if we constrained T to a class and two interfaces, we are saying that the type of argument passed must be of the specified base class (or derived from it), and it must also implement the two interfaces. This flexibility allows the method to really dictate what callers can pass, and compiler errors will be generated if callers do not meet these constraints.

The second benefit of interface constraints is reduced boxing when passing instances of value types. In the previous code fragment, the `M` method was passed `x` (an instance of an `Int32`, which is a value type). No boxing will occur when `x` is passed to `M`. If code inside `M` does call `t.CompareTo(...)`, still no boxing occurs to make the call (boxing may still happen for arguments passed to `CompareTo`).

On the other hand, if `M` had been declared like this:

```
private static Int32 M(IComparable t) {
   ...
}
```

then in order to pass `x` to `M`, `x` would have to be boxed.

For interface constraints, the C# compiler emits certain Intermediate Language (IL) instructions that result in calling the interface method on the value type directly without boxing it. Aside from using interface constraints, there is no other way to get the C# compiler to emit these IL instructions, and therefore, calling an interface method on a value type always causes boxing.

Implementing Multiple Interfaces That Have the Same Method Name and Signature

Occasionally, you might find yourself defining a type that implements multiple interfaces that define methods with the same name and signature. For example, imagine that there are two interfaces defined as follows.

```
public interface IWindow {
   Object GetMenu();
}

public interface IRestaurant {
   Object GetMenu();
}
```

Let's say that you want to define a type that implements both of these interfaces. You'd have to implement the type's members by using explicit interface method implementations as follows.

```
// This type is derived from System.Object and
// implements the IWindow and IRestaurant interfaces.
public sealed class MarioPizzeria : IWindow, IRestaurant {

   // This is the implementation for IWindow's GetMenu method.
   Object IWindow.GetMenu() { ... }

   // This is the implementation for IRestaurant's GetMenu method.
   Object IRestaurant.GetMenu() { ... }

   // This (optional method) is a GetMenu method that has nothing
   // to do with an interface.
   public Object GetMenu() { ... }
}
```

Because this type must implement multiple and separate `GetMenu` methods, you need to tell the C# compiler which `GetMenu` method contains the implementation for a particular interface.

Code that uses a `MarioPizzeria` object must cast to the specific interface to call the desired method. The following code demonstrates.

```
MarioPizzeria mp = new MarioPizzeria();

// This line calls MarioPizzeria's public GetMenu method
mp.GetMenu();

// These lines call MarioPizzeria's IWindow.GetMenu method
IWindow window = mp;
window.GetMenu();

// These lines call MarioPizzeria's IRestaurant.GetMenu method
IRestaurant restaurant = mp;
restaurant.GetMenu();
```

Improving Compile-Time Type Safety with Explicit Interface Method Implementations

Interfaces are great because they define a standard way for types to communicate with each other. Earlier, I talked about generic interfaces and how they improve compile-time type safety and reduce boxing. Unfortunately, there may be times when you need to implement a non-generic interface because a generic version doesn't exist. If any of the interface's method(s) accept parameters of type `System.Object` or return a value whose type is `System.Object`, you will lose compile-time type safety, and you will get boxing. In this section, I'll show you how you can use EIMI to improve this situation somewhat.

Look at the very common `IComparable` interface.

```
public interface IComparable {
   Int32 CompareTo(Object other);
}
```

This interface defines one method that accepts a parameter of type `System.Object`. If I define my own type that implements this interface, the type definition might look like the following.

```
internal struct SomeValueType : IComparable {
   private Int32 m_x;
   public SomeValueType(Int32 x) { m_x = x; }
   public Int32 CompareTo(Object other) {
      return(m_x - ((SomeValueType) other).m_x);
   }
}
```

Using `SomeValueType`, I can now write the following code.

```
public static void Main() {
   SomeValueType v = new SomeValueType(0);
   Object o = new Object();
   Int32 n = v.CompareTo(v); // Undesired boxing
   n = v.CompareTo(o);       // InvalidCastException
}
```

There are two characteristics of this code that are not ideal.

- **Undesired boxing** When v is passed as an argument to the `CompareTo` method, it must be boxed because `CompareTo` expects an `Object`.
- **The lack of type safety** This code compiles, but an `InvalidCastException` is thrown inside the `CompareTo` method when it attempts to cast `o` to `SomeValueType`.

Both of these issues can be fixed by using EIMIs. Here's a modified version of `SomeValueType` that has an EIMI added to it.

```
internal struct SomeValueType : IComparable {
   private Int32 m_x;
   public SomeValueType(Int32 x) { m_x = x; }
```

```
   public Int32 CompareTo(SomeValueType other) {
      return(m_x - other.m_x);
   }

   // NOTE: No public/private used on the next line
   Int32 IComparable.CompareTo(Object other) {
      return CompareTo((SomeValueType) other);
   }
}
```

Notice several changes in this new version. First, it now has two `CompareTo` methods. The first `CompareTo` method no longer takes an `Object` as a parameter; it now takes a `SomeValueType` instead. Because this parameter has changed, the code that casts `other` to `SomeValueType` is no longer necessary and has been removed. Second, changing the first `CompareTo` method to make it type-safe means that `SomeValueType` no longer adheres to the contract placed on it by implementing the `IComparable` interface. So `SomeValueType` must implement a `CompareTo` method that satisfies the `IComparable` contract. This is the job of the second `IComparable.CompareTo` method, which is an EIMI.

Having made these two changes means that we now get compile-time type safety and no boxing.

```
public static void Main() {
   SomeValueType v = new SomeValueType(0);
   Object o = new Object();
   Int32  n = v.CompareTo(v); // No boxing
   n = v.CompareTo(o);        // compile-time error
}
```

If, however, we define a variable of the interface type, we will lose compile-time type safety and experience undesired boxing again.

```
public static void Main() {
   SomeValueType v = new SomeValueType(0);
   IComparable c = v;         // Boxing!

   Object o = new Object();
   Int32  n = c.CompareTo(v); // Undesired boxing
   n = c.CompareTo(o);        // InvalidCastException
}
```

In fact, as mentioned earlier in this chapter, when casting a value type instance to an interface type, the CLR must box the value type instance. Because of this fact, two boxings will occur in the previous `Main` method.

EIMIs are frequently used when implementing interfaces such as `IConvertible`, `ICollection`, `IList`, and `IDictionary`. They let you create type-safe versions of these interfaces' methods, and they enable you to reduce boxing operations for value types.

Be Careful with Explicit Interface Method Implementations

It is critically important for you to understand some ramifications that exist when using EIMIs. And because of these ramifications, you should try to avoid EIMIs as much as possible. Fortunately, generic interfaces help you avoid EIMIs quite a bit. But there may still be times when you will need to use them (such as implementing two interface methods with the same name and signature). Here are the big problems with EIMIs:

- There is no documentation explaining how a type specifically implements an EIMI method, and there is no Microsoft Visual Studio IntelliSense support.
- Value type instances are boxed when cast to an interface.
- An EIMI cannot be called by a derived type.

Let's take a closer look at these problems.

When examining the methods for a type in the .NET Framework reference documentation, explicit interface method implementations are listed, but no type-specific help exists; you can just read the general help about the interface methods. For example, the documentation for the `Int32` type shows that it implements all of `IConvertible` interface's methods. This is good because developers know that these methods exist; however, this has been very confusing to developers because you can't call an `IConvertible` method on an `Int32` directly. For example, the following method won't compile.

```
public static void Main() {
   Int32 x = 5;
   Single s = x.ToSingle(null); // Trying to call an IConvertible method
}
```

When compiling this method, the C# compiler produces the following message: `messagepil17: 'int' does not contain a definition for 'ToSingle'.` This error message confuses the developer because it's clearly stating that the `Int32` type doesn't define a `ToSingle` method when, in fact, it does.

To call `ToSingle` on an `Int32`, you must first cast the `Int32` to an `IConvertible`, as shown in the following method.

```
public static void Main() {
   Int32 x = 5;
   Single s = ((IConvertible) x).ToSingle(null);
}
```

Requiring this cast isn't obvious at all, and many developers won't figure this out on their own. But an even more troublesome problem exists: casting the `Int32` value type to an `IConvertible` also boxes the value type, wasting memory and hurting performance. This is the second of the big problems I mentioned at the beginning of this section.

The third and perhaps the biggest problem with EIMIs is that they cannot be called by a derived class. Here is an example.

```
internal class Base : IComparable {

   // Explicit Interface Method Implementation
   Int32 IComparable.CompareTo(Object o) {
      Console.WriteLine("Base's CompareTo");
      return 0;
   }
}

internal sealed class Derived : Base, IComparable {

   // A public method that is also the interface implementation
   public Int32 CompareTo(Object o) {
      Console.WriteLine("Derived's CompareTo");

      // This attempt to call the base class's EIMI causes a compiler error:
      // error CS0117: 'Base' does not contain a definition for 'CompareTo'
      base.CompareTo(o);
      return 0;
   }
}
```

In `Derived`'s `CompareTo` method, I try to call `base.CompareTo`, but this causes the C# compiler to issue an error. The problem is that the `Base` class doesn't offer a public or protected `CompareTo` method that can be called; it offers a `CompareTo` method that can be called only by using a variable that is of the `IComparable` type. I could modify `Derived`'s `CompareTo` method so that it looks like the following.

```
// A public method that is also the interface implementation
public Int32 CompareTo(Object o) {
   Console.WriteLine("Derived's CompareTo");

   // This attempt to call the base class's EIMI causes infinite recursion
   IComparable c = this;
   c.CompareTo(o);

   return 0;
}
```

In this version, I am casting `this` to an `IComparable` variable, `c`. And then, I use `c` to call `CompareTo`. However, the `Derived`'s public `CompareTo` method serves as the implementation for `Derived`'s `IComparableCompareTo` method, and therefore, infinite recursion occurs. This could be fixed by declaring the `Derived` class without the `IComparable` interface, like the following.

```
internal sealed class Derived : Base /*, IComparable */ { ... }
```

Now the previous `CompareTo` method will call the `CompareTo` method in `Base`. But sometimes you cannot simply remove the interface from the type because you want the derived type to implement an interface method. The best way to fix this is for the base class to provide a virtual method in addition to the interface method that it has chosen to implement explicitly. Then the `Derived` class can override the virtual method. Here is the correct way to define the `Base` and `Derived` classes.

```
internal class Base : IComparable {

   // Explicit Interface Method Implementation
   Int32 IComparable.CompareTo(Object o) {
      Console.WriteLine("Base's IComparable CompareTo");
      return CompareTo(o);   // This now calls the virtual method
   }

   // Virtual method for derived classes (this method could have any name)
   public virtual Int32 CompareTo(Object o) {
      Console.WriteLine("Base's virtual CompareTo");
      return 0;
   }
}

internal sealed class Derived : Base, IComparable {

   // A public method that is also the interface implementation
   public override Int32 CompareTo(Object o) {
      Console.WriteLine("Derived's CompareTo");

      // Now, we can call Base's virtual method
      return base.CompareTo(o);
   }
}
```

Note that I have defined the virtual method above as a public method, but in some cases, you will prefer to make the method protected instead. It is fine to make this method protected instead of public, but that will necessitate other minor changes. This discussion clearly shows you that EIMIs should be used with great care. When many developers first learn about EIMIs, they think that they're cool and they start using them whenever possible. Don't do this! EIMIs are useful in some circumstances, but you should avoid them whenever possible because they make using a type much more difficult.

Design: Base Class or Interface?

I often hear the question, "Should I design a base type or an interface?" The answer isn't always clear-cut. Here are some guidelines that might help you:

- **IS-A vs. CAN-DO relationship** A type can inherit only one implementation. If the derived type can't claim an IS-A relationship with the base type, don't use a base type; use an interface. Interfaces imply a CAN-DO relationship. If the CAN-DO functionality appears to belong with various object types, use an interface. For example, a type can convert instances of itself

to another type (IConvertible), a type can serialize an instance of itself (ISerializable), etc. Note that value types must be derived from System.ValueType, and therefore, they cannot be derived from an arbitrary base class. In this case, you must use a CAN-DO relationship and define an interface.

- **Ease of use** It's generally easier for you as a developer to define a new type derived from a base type than to implement all of the methods of an interface. The base type can provide a lot of functionality, so the derived type probably needs only relatively small modifications to its behavior. If you supply an interface, the new type must implement all of the members.
- **Consistent implementation** No matter how well an interface contract is documented, it's very unlikely that everyone will implement the contract 100 percent correctly. In fact, COM suffers from this very problem, which is why some COM objects work correctly only with Microsoft Word or with Windows Internet Explorer. By providing a base type with a good default implementation, you start off using a type that works and is well tested; you can then modify parts that need modification.
- **Versioning** If you add a method to the base type, the derived type inherits the new method, you start off using a type that works, and the user's source code doesn't even have to be recompiled. Adding a new member to an interface forces the inheritor of the interface to change its source code and recompile.

In the FCL, the classes related to streaming data use an implementation inheritance design. The System.IO.Stream class is the abstract base class. It provides a bunch of methods, such as Read and Write. Other classes—System.IO.FileStream, System.IO.MemoryStream, and System.Net.Sockets.NetworkStream—are derived from Stream. Microsoft chose an IS-A relationship between each of these three classes and the Stream class because it made implementing the concrete classes easier. For example, the derived classes need to implement only synchronous I/O operations; they inherit the ability to perform asynchronous I/O operations from the Stream base class.

Admittedly, choosing to use inheritance for the stream classes isn't entirely clear-cut; the Stream base class actually provides very little implementation. However, if you consider the Windows Forms control classes, in which Button, CheckBox, ListBox, and all of the other controls are derived from System.Windows.Forms.Control, it's easy to imagine all of the code that Control implements, which the various control classes simply inherit to function correctly.

By contrast, Microsoft designed the FCL collections to be interface based. The System.Collections.Generic namespace defines several collection-related interfaces: IEnumerable<out T>, ICollection<T>, IList<T>, and IDictionary<TKey, TValue>. Then Microsoft provided a number of classes, such as List<T>, Dictionary<TKey, TValue>, Queue<T>, Stack<T>, and so on, that implement combinations of these interfaces. Here the designers chose a CAN-DO relationship between the classes and the interfaces because the implementations of these various collection classes are radically different from one another. In other words, there isn't a lot of sharable code between a List<T>, a Dictionary<TKey, TValue>, and a Queue<T>.

The operations these collection classes offer are, nevertheless, pretty consistent. For example, they all maintain a set of elements that can be enumerated, and they all allow adding and removing of elements. If you have a reference to an object whose type implements the `IList<T>` interface, you can write code to insert elements, remove elements, and search for an element without having to know exactly what type of collection you're working with. This is a very powerful mechanism.

Finally, it should be pointed out that you can actually do both: define an interface *and* provide a base class that implements the interface. For example, the FCL defines the `IComparer<in T>` interface and any type can choose to implement this interface. In addition, the FCL provides an abstract base class, `Comparer<T>`, which implements this interface and provides a default implementation for the non-generic `IComparer`'s `Compare` method. Having both an interface definition and a base class offers great flexibility because developers can now choose whichever they prefer.