Chapter 6. Inheritance

C# classes support *inheritance*, a popular object-oriented code reuse mechanism. When you write a class, you can optionally specify a base class. Your class will derive from this, meaning that everything in the base class will be present in your class, as well as any members you add.

Classes support only single inheritance. Interfaces offer a form of multiple inheritance. Value types do not support inheritance at all. One reason for this is that value types are not normally used by reference, which removes one of the main benefits of inheritance: runtime polymorphism. Inheritance is not necessarily incompatible with value-like behavior—some languages manage it—but it often has problems. For example, assigning a value of some derived type into a variable of its base type ends up losing all of the fields that the derived type added, a problem known as *slicing*. C# sidesteps this by restricting inheritance to reference types. When you assign a variable of some derived type into a variable of a base type, you're copying a reference, not the object itself, so the object remains intact. Slicing is an issue only if the base class offers a method that clones the object, and doesn't provide a way for derived classes to extend that (or it does, but some derived class fails to extend it).

Classes specify a base class using the syntax shown in Example 6-1—the base type appears after a colon that follows the class name. This example assumes that a class called `SomeClass` has been defined elsewhere in the project.

Example 6-1. Specifying a base class

```
public class Derived : SomeClass
{
}

public class AlsoDerived : SomeClass, IDisposable
{
    public void Dispose() { }
}
```

As Example 6-1 illustrates, if the class implements any interfaces, these are also listed after the colon. If you want to derive from a class, and you want to implement interfaces as well, the base class must appear first, as the second class shown in Example 6-1 illustrates.

You can derive from a class that in turn derives from another class. The `MoreDerived` class in Example 6-2 derives from `Derived`, which in turn derived from `Base`.

Example 6-2. Inheritance chain

```
public class Base
{
}

public class Derived : Base
{
}

public class MoreDerived : Derived
{
}
```

This means that `MoreDerived` technically has multiple base classes: it derives from both `Derived` (directly) and `Base` (indirectly, via `Derived`). This is not multiple inheritance because there is only a single chain of inheritance—any single class derives directly from at most one base class.

Since a derived class inherits everything the base class has—all its fields, methods, and other members, both public and private—an instance of the derived class can do anything an instance of the base class could do. This is the classic *is a* relationship that inheritance implies in many languages. Any instance of `MoreDerived` is a `Derived`, and also a `Base`. C#'s type system recognizes this relationship.

Inheritance and Conversions

C# provides various built-in implicit conversions. In Chapter 2, we saw the conversions for numeric types, but there are also ones for reference types. If some type `D` derives from `B` (either directly or indirectly), then a reference of type `D` can be converted implicitly to a reference of type `B`. This follows from the "is a" relationship I described in the preceding section—any instance of `D` is a `B`. This implicit conversion enables polymorphism: any code written to work in terms of `B` will be able to work with any type derived from `B`.

Obviously, there is no implicit conversion in the opposite direction—although a variable of type `B` could refer to an object of type `D`, there's no guarantee that it will. There could be any number of types derived from `B`, and a `B` variable could refer to an instance of any of them. Nevertheless, you will sometimes want to attempt to convert a reference from a base type to a derived type, an operation sometimes referred to as a *downcast*. Perhaps you know for a fact that a particular variable holds a reference of a certain type. Or perhaps you're not sure, and would like your code to provide additional services for specific types. C# offers three ways to do this.

The most obvious way to attempt a downcast is to use the cast syntax, the same syntax we use for performing nonimplicit numeric conversions, as Example 6-3 shows.

Example 6-3. Feeling downcast

```
public static void UseAsDerived(Base baseArg)
{
    var d = (Derived) baseArg;

    ... go on to do something with d
}
```

This conversion is not guaranteed to succeed—that's why it's not an implicit conversion. If you try this when the `baseArg` argument refers to something that's not an instance of `Derived`, nor something derived from `Derived`, the conversion will fail, throwing an `InvalidCastException`.

A cast is therefore appropriate only if you're confident that the object really is of the type you expect, and you would consider it to be an error if it turned out not to be. This is useful when an API accepts an object that it will later give back to you. Many asynchronous APIs do this, because in cases where you launch multiple operations concurrently, you need some way of working out which particular one finished when you get a completion notification (although, as we'll see in later chapters, there are various ways to tackle that problem). Since these APIs don't know what sort of data you'll want to associate with an operation, they usually just take a reference of type `object`, and you would typically use a cast to turn it back into a reference of the required type when the reference is eventually handed back to you.

Sometimes, you will not know for certain whether an object is of a particular type. In this case, you can use the `as` operator, as shown in Example 6-4, instead, which allows you to attempt a conversion without risking an exception. If the conversion fails, this operator just returns `null`.

Example 6-4. The as operator

```
public static void MightUseAsDerived(Base b)
{
    var d = b as Derived;

    if (d != null)
    {
        ... go on to do something with d
    }
}
```

Finally, it can occasionally be useful to know whether a reference refers to an object of a particular type, without actually wanting to use any members specific to that type. For example, you might want to skip some particular piece of processing for a certain derived class. The `is` operator, shown in Example 6-5, tests whether an object is of a particular type, returning `true` if it is, and `false` otherwise.

Example 6-5. The is operator

```
if (!(b is WeirdType))
{
    ... do the processing that everything except WeirdType requires
}
```

When converting with a cast or the `as` operator, or when using the `is` operator, you don't necessarily need to specify the exact type. These operations will succeed as long as a reference of the object's real type could be implicitly converted to the type you're looking for. For example, given the `Base`, `Derived`, and `MoreDerived` types that Example 6-2 defines, suppose you have a variable of type `Base` that currently contains a reference to an instance of `MoreDerived`. Obviously, you could cast the reference to `MoreDerived` (and both `as` and `is` would also succeed for that type), but as you'd probably expect, converting to `Derived` would work too.

These three mechanisms also work for interfaces. When you try to convert a reference to an interface type reference, conversion will succeed if the object referred to implements the relevant interface.

Interface Inheritance

Interfaces support inheritance, but it's not quite the same as class inheritance. The syntax is similar, but as Example 6-6 shows, an interface can specify multiple base interfaces, because C# supports multiple inheritance for interfaces. The reason .NET supports this despite offering only single implementation inheritance

is that most of the complications and potential ambiguities that can arise with multiple inheritance do not apply to purely abstract types.

Example 6-6. Interface inheritance

```
interface IBase1
{
    void Base1Method();
}

interface IBase2
{
    void Base2Method();
}

interface IBoth : IBase1, IBase2
{
    void Method3();
}
```

As with class inheritance, interfaces inherit all of their bases' members, so the `IBoth` interface here includes `Base1Method` and `Base2Method`, as well as its own `Method3`. Implicit conversions exist from derived interface types to their bases. For example, a reference of type `IBoth` can be assigned to a variable of type `IBase1` and also `IBase2`. Likewise, any class that implements a derived interface also implements that interface's base interfaces, although as Example 6-7 shows, the class needs to state only that it implements the derived interface, but the compiler will act as though `IBase1` and `IBase2` were in the interface list.

Example 6-7. Implementing a derived interface

```
public class Impl : IBoth
{
    public void Base1Method()
    {
    }

    public void Base2Method()
    {
    }

    public void Method3()
    {
    }
}
```

Generics

If you derive from a generic class, you must supply the type arguments it requires. You must provide concrete types unless your derived class is generic, in which case it can use its own type parameters as arguments. Example 6-8 shows both techniques, and also illustrates that when deriving from a class with multiple type parameters, you can use a mixture of techniques, specifying one type argument directly and punting on the other.

Example 6-8. Deriving from a generic base class

```
public class GenericBase1<T>
{
    public T Item { get; set; }
}

public class GenericBase2<TKey, TValue>
{
    public TKey Key { get; set; }
    public TValue Value { get; set; }
}

public class NonGenericDerived : GenericBase1<string>
```

```
{
}


public class GenericDerived<T> : GenericBase1<T>
{
}


public class MixedDerived<T> : GenericBase2<string, T>
{
}
```

Although you are free to use any of your type parameters as type arguments for a base class, you cannot derive from a type parameter. This is slightly disappointing if you are used to languages that permit such things, but the C# language specification simply forbids it.

Covariance and Contravariance

In Chapter 4, I mentioned that generic types have special rules for type compatibility, referred to as *covariance* and *contravariance*. These rules determine whether references of certain generic types are implicitly convertible to one another when implicit conversions exist between their type arguments.

> **NOTE**
>
> Covariance and contravariance are applicable only to the generic type arguments of interfaces and delegates. (Delegates are described in Chapter 9.) You cannot define a covariant class or struct.

Consider the simple `Base` and `Derived` classes shown earlier in Example 6-2, and look at the method in Example 6-9, which accepts any `Base`. (It does nothing with it, but that's not relevant here—what matters is what its signature says it can use.)

Example 6-9. A method accepting any Base

```
public static void UseBase(Base b)
{
}
```

We already know that as well as accepting a reference to any `Base`, this can also accept a reference to an instance of any type derived from `Base`, such as `Derived`. Bearing that in mind, consider the method in Example 6-10.

Example 6-10. A method accepting any IEnumerable<Base>

```
public static void AllYourBase(IEnumerable<Base> bases)
{
}
```

This requires an object that implements the `IEnumerable<T>` generic interface described in Chapter 5, where `T` is `Base`. What would you expect to happen if we attempted to pass an object that did not implement `IEnumerable<Base>`, but did implement `IEnumerable<Derived>`? Example 6-11 does this, and it compiles just fine.

Example 6-11. Passing an IEnumerable<T> of a derived type

```
IEnumerable<Derived> derivedBases =
    new Derived[] { new Derived(), new Derived() };
AllYourBase(derivedBases);
```

Intuitively, this makes sense. The `AllYourBase` method is expecting an object that can supply a sequence of objects that are all of type `Base`. An `IEnumerable<Derived>` fits the bill because it supplies a sequence of `Derived` objects, and any `Derived` object is also a `Base`. However, what about the code in Example 6-12?

Example 6-12. A method accepting any ICollection<Base>

```
public static void AddBase(ICollection<Base> bases)
{
    bases.Add(new Base());
}
```

Recall from Chapter 5 that `ICollection<T>` derives from `IEnumerable<T>`, and it adds the ability to modify the collection in certain ways. This particular method exploits that by adding a new `Base` object to the collection. That would mean trouble for the code in Example 6-13.

Example 6-13. Error: trying to pass an ICollection<T> with a derived type

```
ICollection<Derived> derivedList = new List<Derived>();
AddBase(derivedList);  // Will not compile
```

Any code that uses the `derivedList` variable will expect every object in that list to be of type `Derived` (or something derived from it, such as the `MoreDerived` class from Example 6-2). But the `AddBase` method in Example 6-12 attempts to add a plain `Base` instance. That can't be correct, and the compiler doesn't allow it. The call to `AddBase` will produce a compiler error complaining that references of type `ICollection<Derived>` cannot be converted implicitly to references of type `ICollection<Base>`.

How does the compiler know that it's not OK to do this, while the very similar-looking conversion from `IEnumerable<Derived>` to `IEnumerable<Base>` is allowed? It's not because Example 6-12 contains code that would cause a problem, by the way. You'd get the same compiler error even if the `AddBase` method were completely empty. The reason we don't get an error is that the `IEnumerable<T>` interface declares its type argument `T` as covariant. You saw the syntax for this in Chapter 5, but I didn't draw attention to it, so Example 6-14 shows the relevant part from that interface's definition again.

Example 6-14. Covariant type parameter

```
public interface IEnumerable<out T> : IEnumerable
```

That `out` keyword does the job. (Again, C# keeps up the C-family tradition of giving each keyword multiple, unrelated jobs—we last saw this keyword in the context of method parameters that can return information to the caller.) Intuitively, describing the type argument `T` as "out" makes sense, in that the `IEnumerable<T>` interface only ever *provides* a `T`—it does not define any members that *accept* a `T`. (The interface uses this type parameter in just one place: its read-only `Current` property.)

Compare that with `ICollection<T>`. This derives from `IEnumerable<T>`, so clearly it's possible to get a `T` out of it, but it's also possible to pass a `T` into its `Add` method. So `ICollection<T>` cannot annotate its type argument with `out`. (If you were to try to write your own similar interface, the compiler would produce an error if you declared the type argument as being covariant. Rather than just taking your word for it, it checks to make sure you really can't pass a `T` in anywhere.) The compiler rejects the code in Example 6-13 because `T` is not covariant in `ICollection<T>`.

The terms *covariant* and *contravariant* come from a branch of mathematics called *category theory*. The parameters that behave like `IEnumerable<T>`'s `T` are called covariant (as opposed to contravariant) because implicit reference conversions for the generic type work in the same direction as conversions for the type argument: `Derived` is implicitly convertible to `Base`, and since `T` is covariant in `IEnumerable<T>`, `IEnumerable<Derived>` is implicitly convertible to `IEnumerable<Base>`.

Predictably, contravariance works the other way around, and as you've probably guessed, we denote it with the `in` keyword. It's easiest to see this in action with code that uses members of types, so Example 6-15 shows a marginally more interesting pair of classes than the earlier examples.

Example 6-15. Class hierarchy with actual members

```
public class Shape
{
    public Rect BoundingBox { get; set; }
}

public class RoundedRectangle : Shape
{
    public double CornerRadius { get; set; }
}
```

Example 6-16 defines two classes that use these shape types. Both implement `IComparer<T>`, which I introduced in Chapter 4. The `BoxAreaComparer` compares two shapes based on the area of their bounding box—whichever shape covers the larger area will be deemed the "larger" by this comparison. The `CornerSharpnessComparer`, on the other hand, compares rounded rectangles by looking at how pointy their corners are.

Example 6-16. Comparing shapes

```
public class BoxAreaComparer : IComparer<Shape>
{
    public int Compare(Shape x, Shape y)
    {
        double xArea = x.BoundingBox.Width * x.BoundingBox.Height;
        double yArea = y.BoundingBox.Width * y.BoundingBox.Height;

        return Math.Sign(xArea - yArea);
    }
```

```
}

public class CornerSharpnessComparer : IComparer<RoundedRectangle>
{
    public int Compare(RoundedRectangle x, RoundedRectangle y)
    {
        // Smaller corners are sharper, so smaller radius is "greater" for
        // the purpose of this comparison, hence the backward subtraction.
        return Math.Sign(y.CornerRadius - x.CornerRadius);
    }
}
```

References of type `RoundedRectangle` are implicitly convertible to `Shape`, so what about `IComparer<T>`? Our `BoxAreaComparer` can compare any shapes, and declares this by implementing `IComparer<Shape>`. The comparer's type argument `T` is only ever used in the `Compare` method, and that is happy to be passed any `Shape`. It will not be fazed if we pass it a pair of `RoundedRectangle` references, so our class is a perfectly adequate `IComparer<RoundedRectangle>`. An implicit conversion from `IComparer<Shape>` to `IComparer<RoundedRectangle>` therefore makes sense, and is allowed. However, the `CornerSharpnessComparer` is fussier. It uses the `CornerRadius` property, which is available only on rounded rectangles, not on any old `Shape`. Therefore, no implicit conversion exists from `IComparer<RoundedRectangle>` to `IComparer<Shape>`.

This is the reverse of what we saw with `IEnumerable<T>`. Implicit conversion is available between `IEnumerable<T1>` and `IEnumerable<T2>` when an implicit reference conversion from `T1` to `T2` exists. But implicit conversion between `IComparer<T1>` and `IComparer<T2>` is available when an implicit reference conversion exists in the other direction: from `T2` to `T1`. That reversed relationship is called contravariance. Example 6-17 is an excerpt of the definition for `IComparer<T>` showing this contravariant type parameter.

Example 6-17. Contravariant type parameter

```
public interface IComparer<in T>
```

Most generic type parameters are neither covariant nor contravariant. `ICollection<T>` cannot be variant, because it contains some members that accept a `T` and some that return one. An `ICollection<Shape>` might contain shapes that are not `RoundedRectangles`, so you cannot pass it to a method expecting an `ICollection<RoundedRectangle>`, because such a method would expect every object it retrieves from the collection to be a rounded rectangle. Conversely, an `ICollection<RoundedRectangle>` cannot be expected to allow shapes other than rounded rectangles to be added, and so you cannot pass an `ICollection<RoundedRectangle>` to a method that expects an `ICollection<Shape>` because that method may try to add other kinds of shapes.

> **NOTE**
>
> Sometimes, generics do not support covariance or contravariance even in situations where they would make sense. One reason for this is that although the CLR has supported variance since generics were introduced in .NET 2.0, C# did not fully support it until version 4.0. Before that release (in 2010), it was not possible to write a covariant or contravariant generic in C#, and you would have gotten an error if you had tried to apply the `in` and `out` keywords to type parameters in earlier versions. The .NET Framework class library was modified in version 4.0: various classes that didn't previously support variance, but for which it made sense, were changed to offer it. However, there are plenty of other class libraries out there, and if these were written before .NET 4.0, there's a good chance that they won't define any kind of variance.

Arrays are covariant, just like `IEnumerable<T>`. This is rather odd, because we can write methods like the one in Example 6-18.

Example 6-18. Changing an element in an array

```
public static void UseBaseArray(Base[] bases)
{
    bases[0] = new Base();
}
```

If I were to call this with the code in Example 6-19, I would be making the same mistake as I did in Example 6-13, where I attempted to pass an `ICollection<Derived>` to a method that wanted to put something that was not `Derived` into the collection. But while Example 6-13 does not compile, Example 6-19 does, due to the surprising covariance of arrays.

Example 6-19. Passing an array with derived element type

```
Derived[] derivedBases = { new Derived(), new Derived() };
UseBaseArray(derivedBases);
```

This makes it look as though we could sneak a reference into an array to an object that is not an instance of the array's element type—in this case, putting a reference to a non-`Derived` object, `Base`, in `Derived[]`. But that would be a violation of the type system. Does this mean the sky is falling?

In fact, C# correctly forbids such a violation, but it does so at runtime. Although a reference to an array of type `Derived[]` can be implicitly converted to a reference of type `Base[]`, any attempt to set an array element in a way that is inconsistent with the type system will throw an `ArrayTypeMismatchException`. So Example 6-18 would throw that exception when it tried to assign a reference to a `Base` into the `Derived[]` array.

Type safety is maintained, and rather conveniently, if we write a method that takes an array and only reads from it, we can pass arrays of some derived element type and it will work. The downside is that the CLR has to do extra work at runtime when you modify array elements to ensure that there is no type mismatch. It may be able to optimize the code to avoid having to check every single assignment, but there is still some overhead, meaning that arrays are not quite as efficient as they might be.

This somewhat peculiar arrangement dates back to the time before .NET had formalized concepts of covariance and contravariance—these came in with generics, which were introduced in .NET 2.0. Perhaps if generics had been around from the start, arrays would be less odd, although having said that, their peculiar form of covariance was for many years the only mechanism built into the framework that provided a way to pass a collection covariantly to a method that wants to read from it using indexing. Until .NET 4.5 introduced `IReadOnlyList<T>` (for which `T` is covariant), there was no read-only indexed collection interface in the framework, and therefore no standard indexed collection interface with a covariant type parameter. (`IList<T>` is read/write, so just like `ICollection<T>`, it cannot offer variance.)

While we're on the subject of type compatibility and the implicit reference conversions that inheritance makes available, there's one more type we should look at: `object`.

System.Object

The `System.Object` type, or `object` as we usually call it in C#, is useful because it can act as a sort of universal container: a variable of this type can hold a reference to almost anything. I've mentioned this before, but I haven't yet explained why it's true. The reason this works is that almost everything derives from `object`.

If you do not specify a base class when writing a class, the C# compiler automatically uses `object` as the base. As we'll see shortly, it chooses different bases for certain kinds of types such as structs, but even those derive from `object` indirectly. (As ever, pointer types are an exception—these do not derive from `object`.)

The relationship between interfaces and objects is slightly more subtle. Interfaces do not derive from `object`, because an interface can specify only other interfaces as its bases. However, a reference of any interface type is implicitly convertible to a reference of type `object`. This conversion will always be valid, because all types that are capable of implementing interfaces ultimately derive from `object`. Moreover, C# chooses to make the `object` class's members available through interface references even though they are not, strictly speaking, members of the interface. This means that any references of any kind always offer the following methods defined by `object`: `ToString`, `Equals`, `GetHashCode`, and `GetType`.

The Ubiquitous Methods of object

I've used `ToString` in numerous examples already. The default implementation returns the object's type name, but many types provide their own implementation of `ToString`, returning a more useful textual representation of the object's current value. The numeric types return a decimal representation of their value, for example, while `bool` returns either `"True"` or `"False"`.

I discussed `Equals` and `GetHashCode` in Chapter 3, but I'll provide a quick recap here. `Equals` allows an object to be compared with any other object. The default implementation just performs an identity comparison—that is, it returns `true` only when an object is compared with itself. Many types provide an `Equals` method that performs value-like comparison—for example, two distinct `string` objects may contain identical text, in which case they will report being equal to each other. (Should you need it, the identity-based comparison is always available through the `object` class's static `ReferenceEquals` method.) Incidentally, `object` also defines a static version of `Equals` that takes two arguments. This checks whether the arguments are `null`, returning `true` if both are `null` and `false` if only one is `null`, and otherwise, it defers to the first argument's `Equals` method. And, as discussed in Chapter 3, `GetHashCode` returns an integer that is a reduced representation of the object's value, which is used by hash-based mechanisms such as the `Dictionary<TKey, TValue>` collection class. Any pair of objects for which `Equals` returns `true` must return the same hash codes.

The `GetType` method provides a way to discover things about the object's type. It returns a reference of type `Type`. That's part of the reflection API, which is the subject of Chapter 13.

Besides these public members, available through any reference, `object` defines two more members that are not universally accessible. An object has access to these members only on itself. They are `Finalize` and `MemberwiseClone`. The CLR calls the `Finalize` method for you to notify you that your object is no longer in use and the memory it uses is about to be reclaimed. In C# we do not normally work directly with the `Finalize` method, because C# presents this mechanism through destructors, as I'll show in Chapter 7. `MemberwiseClone` creates a new instance of the same type as your object, initialized with copies of all of your object's fields. If you need a way to create a clone of an object, this may be easier than writing code that copies all the contents across by hand.

The reason these last two methods are available only from inside the object is that you might not want other people cloning your object, and it would be unhelpful if external code could call the `Finalize` method, fooling your object into thinking that it was about to be freed if in fact it wasn't. The `object` class limits the accessibility of these members. But they're not private—that would mean that only the `object` class itself could access them, because private members are not visible even to derived classes. Instead, `object` makes theses members *protected*, an accessibility specifier designed for inheritance scenarios.

Accessibility and Inheritance

By now, you will already be familiar with most of the accessibility levels available for types and their members. Elements marked as `public` are available to all, `private` members are accessible only from within the type that declared them, and `internal` members are available to any code defined in the same component.[26] But with inheritance, we get two other accessibility options.

A member marked as `protected` is available inside the type that defined it, and also inside any derived types. But for code using an instance of your type, `protected` members are not accessible, just like `private` members.

There's another protection level for type members: `protected internal`. (You can write `internal protected` if you prefer; the order makes no difference.) This makes the member more accessible than either `protected` or `internal` on its own: the member will be accessible to all derived types *and* to all code that shares an assembly.

> **NOTE**
>
> You may be wondering about the obvious conceptual counterpart: members that are available only to types that are both derived from *and* defined in the same component as the defining type. The CLR does support such a protection level, but C# does not provide any way to specify it.

You can specify `protected` or `protected internal` for any member of a type, not just methods. Even nested types can use these accessibility specifiers.

Although `protected` (and `protected internal`) members are not available through an ordinary variable of the defining type, they are still part of the type's public API, in the sense that anyone who has access to your classes will be able to use these members. As with most languages that support a similar mechanism, `protected` members in C# are typically used to provide services that derived classes might find useful. If you write a `public` class that supports inheritance, then anyone can derive from it and gain access to its `protected` members. Removing or changing `protected` members would therefore risk breaking code that depends on your class just as surely as removing or changing `public` members would.

When you derive from a class, you cannot make your class more visible than its base. If you derive from an `internal` class, for example, you cannot declare your class to be `public`. Your base class forms part of your class's API, so anyone wishing to use your class will also in effect be using its base class; this means that if the base is inaccessible, your class will also be inaccessible, which is why C# does not permit a class to be more visible than its base. For example, if you derive from a `protected` nested class, your derived class could be `protected` or `private`, but not `public`, `internal`, or `protected internal`.

> **NOTE**
>
> This restriction does not apply to the interfaces you implement. A `public` class is free to implement `internal` or `private` interfaces. However, it does apply to an interface's bases: a `public` interface cannot derive from an `internal` interface.

When defining methods, there's another keyword you can add for the benefit of derived types: `virtual`.

Virtual Methods

A *virtual method* is one that a derived type can replace. Several of the methods defined by `object` are virtual: the `ToString`, `Equals`, `GetHashCode`, and `Finalize` methods are all designed to be replaced. The code required to produce a useful textual representation of an object's value will differ considerably from one type to another, as will the logic required to determine equality and produce a hash code. Types typically define a finalizer only if they need to do some specialized cleanup work when they go out of use.

Not all methods are virtual. In fact, C# makes methods nonvirtual by default. The `object` class's `GetType` method is not virtual, so you can always trust the information it returns to you because you know that you're calling the `GetType` method supplied by the .NET Framework, and not some type-specific substitute designed to fool you. To declare that a method should be virtual, use the `virtual` keyword as Example 6-20 shows.

Example 6-20. A class with a virtual method

```
public class BaseWithVirtual
{
    public virtual void ShowMessage()
    {
        Console.WriteLine("Hello from BaseWithVirtual");
    }
}
```

There's nothing unusual about the syntax for invoking a virtual method. As Example 6-21 shows, it looks just like calling any other method.

Example 6-21. Using a virtual method

```
public static void CallVirtualMethod(BaseWithVirtual o)
{
```

```
    o.ShowMessage();
}
```

The difference between virtual and nonvirtual method invocations is that a virtual method call decides at runtime which method to invoke. The code in Example 6-21 will, in effect, inspect the object passed in, and if the object's type supplies its own implementation of `ShowMessage`, it will call that instead of the one defined in `BaseWithVirtual`. The method is chosen based on the actual type the target object turns out to have at runtime, and not the static type (determined at compile time) of the expression that refers to the target object.

> **NOTE**
>
> Since virtual method invocation selects the method based on the type of the object on which you invoke the method, static methods cannot be virtual.

Derived types are not obliged to replace virtual methods, of course. Example 6-22 shows two classes that derive from the one in Example 6-20. The first leaves the base class's implementation of `ShowMessage` in place. The second overrides it. Note the `override` keyword—C# requires us to state explicitly that we are intending to override a virtual method.

Example 6-22. Overriding virtual methods

```
public class DeriveWithoutOverride : BaseWithVirtual
{
}


public class DeriveAndOverride : BaseWithVirtual
{
    public override void ShowMessage()
    {
        Console.WriteLine("This is an override");
    }
}
```

We can use these types with the method in Example 6-21. Example 6-23 calls it three times, passing in a different type of object each time.

Example 6-23. Exploiting virtual methods

```
CallVirtualMethod(new BaseWithVirtual());
CallVirtualMethod(new DeriveWithoutOverride());
CallVirtualMethod(new DeriveAndOverride());
```

This produces the following output:

```
Hello from BaseWithVirtual
Hello from BaseWithVirtual
This is an override
```

Obviously, when we pass an instance of the base class, we get the output that the base class's `ShowMessage` method prints. We also get that with the derived class that has not supplied an override. It's only the final class, which overrides the method, that produces different output.

Overriding is very similar to implementing methods in interfaces—virtual methods provide another way to write polymorphic code. Example 6-21 can use a variety of types, which can modify the behavior if necessary. The big difference is that the base class can supply a default implementation for each virtual method, something that interfaces cannot do.

Abstract Methods

You can define a virtual method without providing a default implementation. C# calls this an *abstract method*. If a class contains one or more abstract methods, the class is incomplete, because it doesn't provide all of the methods it defines. Classes of this kind are also described as being abstract, and it is not possible to construct instances of an abstract class; attempting to use the `new` operator with an abstract class will cause a compiler error. Sometimes when discussing classes, it's useful to make clear that some particular class is *not* abstract, for which we normally use the term *concrete class*.

If you derive from an abstract class, then unless you provide implementations for all the abstract methods, your derived class will also be abstract. You must state your intention to write an abstract class with the `abstract` keyword; if this is absent from a class that has unimplemented abstract methods (either ones it has defined itself, or ones it has inherited from its base class), the C# compiler will report an error. Example 6-24 shows an abstract class that defines a single abstract method. Abstract methods are virtual by definition; there wouldn't be much use in defining a method that has no body, and didn't provide a way for derived classes to supply a body.

Example 6-24. An abstract class

```
public abstract class AbstractBase
{
```

```
    public abstract void ShowMessage();
}
```

As with interface members, abstract method declarations just define the signature, and do not contain a body. Unlike with interfaces, each abstract member has its own accessibility—you can declare abstract methods as `public`, `internal`, `protected internal`, or `protected`. (It makes no sense to make an abstract or virtual method `private`, because the method will be inaccessible to derived types and therefore impossible to override.)

> **NOTE**
>
> Although classes that contain abstract methods are required to be abstract, the converse is not true. It is legal, albeit unusual, to define a class as abstract even if it would be a viable nonabstract class. This prevents the class from being constructed. A class that derives from this will be concrete without needing to override any abstract methods.

Abstract classes have the option to declare that they implement an interface without needing to provide a full implementation. You can't just declare the interface and omit members, though. You must explicitly declare all of its members, marking any that you want to leave unimplemented as being abstract, as Example 6-25 shows. This forces derived types to supply the implementation.

Example 6-25. Abstract interface implementation

```
public abstract class MustBeComparable : IComparable<string>
{
    public abstract int CompareTo(string other);
}
```

There's clearly some overlap between abstract classes and interfaces. Both provide a way to define an abstract type that code can use without needing to know the exact type that will be supplied at runtime. Each option has its pros and cons. Interfaces have the advantage that a single type can implement multiple interfaces; a class gets to specify only a single base class. But abstract classes can provide default implementations for some or even all methods. This makes abstract classes more amenable to evolution as you release new versions of your code.

Imagine what would happen if you had written and released a library that defined some public interfaces, and in the second release of the library, you decided that you wanted to add some new members to some of these interfaces. This might not cause a problem for customers using your code; any place where they use a reference of that interface type will be unaffected by the addition of new features. However, what if some of your customers have written implementations of your interfaces? Suppose, for example, that in a future version of .NET, Microsoft decided to add a new member to the `IEnumerable<T>` interface.

That would be a disaster. This interface is widely used, but also widely implemented. Classes that already implement `IEnumerable<T>` would become invalid because they would not provide this new member, so old code would fail to compile, and code already compiled would throw `MissingMethodException` errors at runtime. Or worse, some classes might by chance already have a member with the same name and signature as the newly added method. The compiler would treat that existing member as part of the implementation of the interface, even though the developer who wrote the method did not write it with that intention. So unless the existing code coincidentally happens to do exactly what the new member requires, we'd have a problem, and we wouldn't get a compiler error.

Consequently, the widely accepted rule is that you do not alter interfaces once they have been published. If you have complete control over all of the code that uses an interface, you can get away with modifying the interface, because you can make any necessary modifications to code that consumes it. But once the interface has become available for use in codebases you do not control—that is, once it has been published—it's no longer possible to change it without being likely to break someone else's code.

Abstract base classes do not have to suffer from this problem. Obviously, introducing new abstract members would cause exactly the same sorts of issues, but introducing new virtual methods is considerably less problematic. With a nonabstract virtual method, you supply a default implementation, so it doesn't matter if a derived class does not implement it.

But what if, after releasing version 1.0 of a component, you add a new virtual method in v1.1 that turns out to have the same name and signature as a method that one of your customers happens to have added in a derived class? Perhaps in version 1.0, your component defines the rather uninteresting base class shown in Example 6-26.

Example 6-26. Base type version 1.0

```
public class LibraryBase
{
}
```

If you release this library, perhaps as a product in its own right, or maybe as part of some software development kit (SDK) for your application, a customer might write a derived type such as the one in Example 6-27. She has written a `Start` method that is clearly not meant to override anything in the base class.

Example 6-27. Class derived from version 1.0 base

```
public class CustomerDerived : LibraryBase
{
    public void Start()
    {
        Console.WriteLine("Derived type's Start method");
    }
}
```

Of course, you won't necessarily get to see every line of code that your customers write, so you might be unaware of that `Start` method. So in version 1.1 of your component, you might decide to add a new virtual method, also called `Start`, as Example 6-28 shows.

Example 6-28. Base type version 1.1

```
public class LibraryBase
{
    public virtual void Start() { }
}
```

Imagine that your system calls this method as part of some initialization procedure. You've defined a default empty implementation so that types derived from `LibraryBase` that don't need to take part in that procedure don't have to do anything. Types that wish to participate will override this method. But what happens with the class in Example 6-27? Clearly the developer who wrote that did not intend to participate in your new initialization mechanism, because that didn't even exist at the time at which the code was written. It could be bad if your code calls the `CustomerDerived` class's `Start` method, because the developer presumably expects it to be called only when her code decides to call it. Fortunately, the compiler will detect this problem. If the customer attempts to compile Example 6-27 against version 1.1 of your library (Example 6-28), the compiler will warn her that something is not right:

```
warning CS0114: 'CustomerDerived.Start()' hides inherited member
'LibraryBase.Start()'. To make the current member override that implementation,
add the override keyword. Otherwise add the new keyword.
```

This is why the C# compiler requires the `override` keyword when we replace virtual methods. It wants to know whether we were intending to override an existing method, so that if we weren't, it can warn us about collisions.

It reports a *warning* rather than an *error*, because it provides a behavior that is likely to be safe when this situation has arisen due to the release of a new version of a library. The compiler guesses—correctly, in this case—that the developer who wrote the `CustomerDerived` type didn't mean to override the `LibraryBase` class's `Start` method. So rather than having the `CustomerDerived` type's `Start` method override the base class's virtual method, it *hides* it. A derived type is said to hide a member of a base class when it introduces a new member with the same name.

Hiding methods is quite different than overriding them. When hiding occurs, the base method is not replaced. Example 6-29 shows how the hidden `Start` method remains available. It creates a `CustomerDerived` object and places a reference to that object in two variables of different types: one of type `CustomerDerived`, and one of type `LibraryBase`. It then calls `Start` through each of these.

Example 6-29. Hidden versus virtual method

```
var d = new CustomerDerived();
LibraryBase b = d;

d.Start();
b.Start();
```

When we use the `d` variable, the call to `Start` ends up calling the derived type's `Start` method, the one that has hidden the base member. But the `b` variable's type is `LibraryBase`, so that invokes the base `Start` method. If `CustomerDerived` had overridden the base class's `Start` method instead of hiding it, both of those method calls would have invoked the override.

When name collisions occur because of a new library version, this hiding behavior is usually the right thing to do. If the customer's code has a variable of type `CustomerDerived`, then that code will want to invoke the `Start` method specific to that derived type. However, the compiler produces a warning, because it doesn't know for certain that this is the reason for the problem. It might be that you *did* mean to override the method, and you just forgot to write the `override` keyword.

Like many developers, I don't like to see compiler warnings, and I try to avoid committing code that produces them. But what should you do if a new library version puts you in this situation? The best long-term solution is probably to change the name of the method in your derived class so that it doesn't clash with the method in the new version of the library. However, if you're up against a deadline, you may want a more expedient solution. So C# lets you declare that you know that there's a name clash, and that you definitely want to hide the base member, not override it. As Example 6-30 shows, you can use the `new` keyword to state that you're aware of the issue, and you definitely want to hide the base class member. The code will still behave in the same way, but you'll no longer get the warning, because you've assured the compiler that you know what's going on. But this is an issue you should fix at some point, because sooner or later the existence of two methods with the same name on the same type that mean different things is likely to cause confusion.

Example 6-30. Avoiding warnings when hiding members

```
public class CustomerDerived : LibraryBase
{
    public new void Start()
    {
        Console.WriteLine("Derived type's Start method");
    }
}
```

Just occasionally, you may see the `new` keyword used in this way for reasons other than handling library versioning issues. For example, the `ISet<T>` interface that I showed in Chapter 5 uses it to introduce a new `Add` method. `ISet<T>` derives from `ICollection<T>`, an interface that already provides an `Add` method, which takes an instance of `T` and has a `void` return type. `ISet<T>` makes a subtle change to this, shown in Example 6-31.

Example 6-31. Hiding to change the signature

```
public interface ISet<T> : ICollection<T>
{
    new bool Add(T item);
    ... other members omitted for clarity
}
```

The `ISet<T>` interface's `Add` method tells you whether the item you just added was already in the set, something the base `ICollection<T>` interface's `Add` method doesn't support. `ISet<T>` needs its `Add` to have a different return type—`bool` instead of `void`—so it defines `Add` with the `new` keyword to indicate that it should hide the `ICollection<T>` one. Both methods are still available—if you have two variables, one of type `ICollection<T>` and the other of type `ISet<T>`, both referring to the same object, you'll be able to access the `void Add` through the former, and the `bool Add` through the latter. (Microsoft didn't have to do this. It could have called the new `Add` method something else—`AddIfNotPresent`, for example. But it's arguably less confusing just to have the one method name for adding things to a collection, particularly since you're free to ignore the return value, at which point the new `Add` looks indistinguishable from the old one. And most `ISet<T>` implementations will implement the `ICollection<T>.Add` method by calling straight through to the `ISet<T>.Add` method, so it makes sense that they have the same name.)

So far, I've discussed method hiding only in the context of compiling old code against a new version of a library. What happens if you have old code compiled against an old library but that ends up running against a new version? That's a scenario you are highly likely to run into when the library in question is the .NET Framework class library. Suppose you are using third-party components that you have only in binary form (e.g., ones you've bought from a company that does not supply source code). The supplier will have built these to use some particular version of .NET. If you upgrade your application to run with a new version of .NET, you might not be able to get hold of newer versions of the third-party components—maybe the vendor hasn't released them yet, or perhaps it's gone out of business.

If the components you're using were compiled for, say, .NET 4.0, and you use them in a project built for .NET 4.5, all of those older components will end up using the .NET 4.5 versions of the framework class library. The .NET Framework has a versioning policy that arranges for all the components that a particular program uses to get the same version of the framework class library, regardless of which version any individual component may have been built for. So it's entirely possible that some component, *OldControls.dll*, contains classes that derive from classes in the .NET 4.0 Framework, and that define members that collide with the names of members newly added in .NET 4.5.

This is more or less the same scenario as I described earlier, except that the code that was written for an older version of a library is not going to be recompiled. We're not going to get a compiler warning about hiding a method, because that would involve running the compiler, and we have only the binary for the relevant component. What happens now?

Fortunately, we don't need the old component to be recompiled. The C# compiler sets various flags in the compiled output for each method it compiles, indicating things like whether the method is virtual or not, and whether the method was intended to override some method in the base class. When you put the `new` keyword on a method, the compiler sets a flag indicating that the method is not meant to override anything. The CLR calls this the *newslot* flag. When C# compiles a method such as the one in Example 6-27, which does not specify either `override` or `new`, it also sets this same *newslot* flag for that method, because at the time the method was compiled, there was no method of the same name on the base class. As far as both the developer and the compiler were concerned, the `CustomerDerived` class's `Start` was written as a brand-new method that was not connected to anything on the base class.

So when this old component gets loaded in conjunction with a new version of the framework library defining the base class, the CLR can see what was intended—it can see that, as far as the author of the `CustomerDerived` class was concerned, `Start` is not meant to override anything. It therefore treats `CustomerDerived.Start` as a distinct method from `LibraryBase.Start`—it hides the base method just like it did when we were able to recompile.

By the way, everything I've said about virtual methods can also apply to properties, because a property's accessors are just methods. So you can define virtual properties, and derived classes can override or hide these in exactly the same way as with methods. I won't be getting to events until Chapter 9, but those are also methods in disguise, so they can also be virtual.

Just occasionally, you may want to write a class that overrides a virtual method, and then prevents derived classes from overriding it again. For this, C# defines the `sealed` keyword, and in fact, it's not just methods that can be sealed.

Sealed Methods and Classes

Virtual methods are deliberately open to modification through inheritance. A sealed method is the opposite—it is one that cannot be overridden. Methods are sealed by default in C#: methods cannot be overridden unless declared virtual. But when you override a virtual method, you can seal it, closing it off for further modification. Example 6-32 uses this technique to provide a custom `ToString` implementation that cannot be further overridden by derived classes.

Example 6-32. A sealed method

```
public class FixedToString
{
    public sealed override string ToString()
    {
        return "Arf arf!";
    }
```

```
}
```

You can also seal an entire class, preventing anyone from deriving from it. Example 6-33 shows a class that not only does nothing, but also prevents anyone from extending it to do something useful. (You'd normally seal only a class that does something. This example is just to illustrate where the keyword goes.)

Example 6-33. A sealed class

```
public sealed class EndOfTheLine
{
}
```

Some types are inherently sealed. Value types, for example, do not support inheritance, so structs and enums are effectively sealed. The built-in `string` class is also sealed.

There are two normal reasons for sealing either classes or methods. One is that you want to guarantee some particular invariant, and if you leave your type open to modification, you will not be able to guarantee that invariant. For example, instances of the `string` type are immutable. The `string` type itself does not provide any way to modify an instance's value, and because nobody can derive from `string`, you can guarantee that if you have a reference of type `string`, you have a reference to an immutable object. This makes it safe to use in scenarios where you don't want the value to change—for example, when you use an object as a key to a dictionary (or anything else that relies on a hash code), you need the value not to change, because if the hash code changes while the item is in use as a key, the container will malfunction.

The other usual reason for leaving things sealed is that designing types that can successfully be modified through inheritance is hard, particularly if your type will be used outside of your own organization. Simply opening things up for modification is not sufficient—if you decide to make all your methods virtual, it might make it easy for people using your type to modify its behavior, but you will have made a rod for your back when it comes to maintaining the base class. Unless you control all of the code that derives from your class, it will be almost impossible to change anything in the base, because you will never know which methods may have been overridden in derived classes, making it very hard to ensure that your class's internal state is consistent at all times. Developers writing derived types will doubtless do their best not to break things, but they will inevitably rely on aspects of your class's behavior that are undocumented. So in opening up every aspect of your class for modification through inheritance, you rob yourself of the freedom to change your class.

You should typically be very selective about which methods, if any, you make virtual. And you should also document whether callers are allowed to replace the method completely, or whether they are required to call the base implementation as part of their override. Speaking of which, how do you do that?

Accessing Base Members

Everything that is in scope in a base class and is not private will also be in scope and accessible in a derived type. So, for the most part, if you want to access some member of the base class, you just access it as if it were a normal member of your class. You can either access members through the `this` reference, or just refer to them by name without qualification.

However, there are some situations in which it is useful to be able to state that you are explicitly referring to a base class member. In particular, if you have overridden a method, calling that method by name will invoke your override. If you want to call back to the original method that you overrode, there's a special keyword for that, shown in Example 6-34.

Example 6-34. Calling the base method after overriding

```
public class CustomerDerived : LibraryBase
{
    public override void Start()
    {
        Console.WriteLine("Derived type's Start method");
        base.Start();
    }
}
```

By using the `base` keyword, we are opting out of the normal virtual method dispatch mechanism. If we had written just `Start()`, that would have been a recursive call, which would be undesirable here. By writing `base.Start()`, we get the method that would have been available on an instance of the base class, the method that we overrode.

In this example, I've called the base class's implementation after completing my own work. C# doesn't care when you call the base—you could call it as the first thing the method does, as the last, or halfway through the method. You could even call it several times, or not at all. It is up to the author of the base class to document whether and when the base class implementation of the method should be called by an override.

You can use the `base` keyword for other members too, such as properties and events. However, access to base constructors works slightly differently.

Inheritance and Construction

Although a derived class inherits all the members of its base class, this does not mean the same thing for constructors as it does for everything else. With other members, if they are public in the base class, they will be public members of the derived class too, accessible to anyone who uses your derived class. But constructors are special, because someone using your class cannot construct it by using one of the constructors defined by the base class.

It's obvious enough why that should be: if you want an instance of some type `D`, then you'll want it to be a fully fledged `D` with everything in it properly initialized. Suppose that `D` derives from `B`. If you were able to use one of `B`'s constructors directly, it wouldn't do anything to the parts specific to `D`. A base class's constructor won't know about any of the fields defined by a derived class, so it cannot initialize them. If you want a `D`, you'll need a constructor that

knows how to initialize a `D`. So with a derived class, you can use only the constructors offered by that derived class, regardless of what constructors the base class might provide.

In the examples I've shown so far in this chapter, I've been able to ignore this because of the default constructor that C# provides. As you saw in Chapter 3, if you don't write a constructor, C# writes one for you that takes no arguments. It does this for derived classes too, and the generated constructor will invoke the no-arguments constructor of the base class. But this changes if I start writing my own constructors. Example 6-35 defines a pair of classes, where the base defines an explicit no-arguments constructor, and the derived class defines one that requires an argument.

Example 6-35. No default constructor in derived class

```
public class BaseWithZeroArgCtor
{
    public BaseWithZeroArgCtor()
    {
        Console.WriteLine("Base constructor");
    }
}

public class DerivedNoDefaultCtor : BaseWithZeroArgCtor
{
    public DerivedNoDefaultCtor(int i)
    {
        Console.WriteLine("Derived constructor");
    }
}
```

Because the base class has a zero-argument constructor, I can construct it with `new BaseWithZeroArgCtor()`. But I cannot do this with the derived type: I can construct that only by passing an argument—for example, `new DerivedNoDefaultCtor(123)`. So as far as the publicly visible API of `DerivedNoDefaultCtor` is concerned, the derived class appears not to have inherited its base class's constructor.

However, it has in fact inherited it, as you can see by looking at the output you get if you construct an instance of the derived type:

```
Base constructor
Derived constructor
```

When constructing an instance of `DerivedNoDefaultCtor`, the base class's constructor runs immediately before the derived class's constructor. Since the base constructor ran, clearly it was available. All of the base class's constructors are available to a derived type, but they can be invoked only by constructors in the derived class. In Example 6-35, the base constructor was invoked implicitly: all constructors are required to invoke a constructor on their base class, and if you don't specify which to invoke, the compiler invokes the base's zero-argument constructor for you.

What if the base doesn't define a parameterless constructor? In that case, you'll get a compiler error if you derive a class that does not specify which constructor to call. Example 6-36 shows a base class with no zero-argument constructor. (The presence of any explicit constructors disables the compiler's normal generation of a default constructor, and since this base class supplies only a constructor that takes arguments, this means there is no zero-argument constructor.) It also shows a derived class with two constructors, both of which call into the base constructor explicitly, using the `base` keyword.

Example 6-36. Invoking a base constructor explicitly

```
public class BaseNoDefaultCtor
{
    public BaseNoDefaultCtor(int i)
    {
        Console.WriteLine("Base constructor: " + i);
    }
}

public class DerivedCallingBaseCtor : BaseNoDefaultCtor
{
    public DerivedCallingBaseCtor()
        : base(123)
    {
        Console.WriteLine("Derived constructor (default)");
    }

    public DerivedCallingBaseCtor(int i)
```

```
        : base(i)
    {
        Console.WriteLine("Derived constructor: " + i);
    }
}
```

The derived class here decides to supply a parameterless constructor even though the base class doesn't have one—it supplies a fixed value for the argument the base requires. The second just passes its argument through to the base.

> **NOTE**
>
> Here's a frequently asked question: *how do I provide all the same constructors as my base class, just passing all the arguments straight through?* The answer is: *write all the constructors by hand.* There is no way to get C# to generate a set of constructors in a derived class that look identical to the ones that the base class offers. You need to do it the long-winded way.

As Chapter 3 showed, a class's field initializers run before its constructor. The picture is slightly more complicated once inheritance is involved, because there are multiple classes and multiple constructors. The easiest way to predict what will happen is to understand that although instance field initializers and constructors have separate syntax, C# ends up compiling all the initialization code for a particular class into the constructor. This code performs the following steps: first, it runs any field initializers specific to this class (so this step does not include base field initializers—the base class will take care of itself); next, it calls the base class constructor; and finally, it runs the body of the constructor. The upshot of this is that in a derived class, your instance field initializers will run before any base class construction has occurred—not just before the base constructor body, but even before the base's instance fields have been initialized. Example 6-37 illustrates this.

Example 6-37. Exploring construction order

```
public class BaseInit
{
    protected static int Init(string message)
    {
        Console.WriteLine(message);
        return 1;
    }

    private int b1 = Init("Base field b1");

    public BaseInit()
    {
        Init("Base constructor");
    }

    private int b2 = Init("Base field b2");
}

public class DerivedInit : BaseInit
{
    private int d1 = Init("Derived field d1");

    public DerivedInit()
    {
        Init("Derived constructor");
    }

    private int d2 = Init("Derived field d2");
}
```

I've put the field initializers on either side of the constructor just to prove that their position relative to nonfield members is irrelevant. The order of the fields matters, but only with respect to one another. Constructing an instance of the `DerivedInit` class produces this output:

```
 Derived field d1
 Derived field d2
```

```
Base field b1

Base field b2

Base constructor

Derived constructor
```

This verifies that the derived type's field initializers run first, and then the base field initializers, followed by the base constructor, and then finally the derived constructor. In other words, although constructor bodies start with the base class, instance field initialization happens in reverse.

That's why you don't get to invoke instance methods in field initializers. Static methods are available, but instance methods are not, because the class is a long way from being ready. It could be problematic if one of the derived type's field initializers were able to invoke a method on the base class, because the base class has performed no initialization at all at that point—not only has its constructor body not run, but its field initializers haven't run either. If instance methods were available during this phase, we'd have to write all of our code to be very defensive, because we could not assume that our fields contain anything useful.

As you can see, the constructor bodies run relatively late in the process, which is why we are allowed to invoke methods from them. But there's still potential danger here. What if the base class defines a virtual method and invokes that method on itself in its constructor? If the derived type overrides that, we'll be invoking the method before the derived type's constructor body has run. (Its field initializers will have run at that point, though. In fact, this is the main benefit of the fact that field initializers run in what seems to be reverse order—it means that derived classes have a way of performing some initialization before the base class's constructor has a chance to invoke a virtual method.) If you're familiar with C++, you might hazard a guess that when the base constructor invokes a virtual method, it'll run the base implementation. But C# does it differently: a base class's constructor will invoke the derived class's override in that case. This is not necessarily a problem, and it can occasionally be useful, but it means you need to think carefully and document your assumptions clearly if you want your object to invoke virtual methods on itself during construction.

Special Base Types

The .NET Framework class library defines a few base types that have special significance in C#. The most obvious is `System.Object`, which I've already described in some detail.

There's also `System.ValueType`. This is the abstract base type of all value types, so any `struct` you define—and also all of the built-in value types, such as `int` and `bool`—derive from `ValueType`. Ironically, `ValueType` itself is a reference type; only types that derive from `ValueType` are value types. Like most types, `ValueType` derives from `System.Object`. There is an obvious conceptual difficulty here: in general, derived classes are everything their base class is, plus whatever functionality they add. So, given that `object` and `ValueType` are both reference types, it may seem odd that types derived from `ValueType` are not. And for that matter, it's not obvious how an `object` variable can hold a reference to an instance of something that's not a reference type. I will resolve all of these issues in Chapter 7.

C# does not permit you to derive explicitly from `ValueType`. If you want to write a type that derives from `ValueType`, that's what the `struct` keyword is for. You can declare a variable of type `ValueType`, but since the type doesn't define any public members, a `ValueType` reference doesn't enable anything you can't do with an `object` reference. The only observable difference is that with a variable of that type, you can assign instances of any value type into it but not instances of a reference type. Aside from that, it's identical to `object`. Consequently, it's fairly rare to see `ValueType` mentioned explicitly in C# code.

Enumeration types also all derive from a common abstract base type: `System.Enum`. Since enums are value types, you won't be surprised to find out that `Enum` derives from `ValueType`. As with `ValueType`, you would never derive from `Enum` explicitly—you use the `enum` keyword for that. Unlike `ValueType`, `Enum` does add some useful members. For example, its static `GetValues` method returns an array of all the enumeration's values, while `GetNames` returns an array with all those values converted to strings. It also offers `Parse`, which converts from the string representation back to the enumeration value.

As Chapter 5 described, arrays all derive from a common base class, `System.Array`, and you've already seen the features that offers.

The `System.Exception` base class is special: when you throw an exception, C# requires that the object you throw be of this type or a type that derives from it. (Exceptions are the topic of Chapter 8.)

Delegate types all derive from a common base type, `System.MulticastDelegate`, which in turn derives from `System.Delegate`. I'll discuss these in Chapter 9.

Those are all the base types that the CTS treats as being special. There's one more base type to which the C# compiler assigns special significance, and that's `System.Attribute`. In Chapter 1, I applied certain annotations to methods and classes to tell the unit test framework to treat them specially. These attributes all correspond to types, so I applied the `[TestClass]` attribute to a class, and in doing so, I was using a type called `TestClassAttribute`. Types designed to be used as attributes are all required to derive from `System.Attribute`. Some of them are recognized by the compiler—for example, there are some that control the version numbers that the compiler puts into the file headers of the EXE and DLL files it produces. I'll show all of this in Chapter 15.

Summary

C# supports single implementation inheritance, and only with classes—you cannot derive from a struct at all. However, interfaces can declare multiple bases, and a class can implement multiple interfaces. Implicit reference conversions exist from derived types to base types, and generic types can choose to offer additional implicit reference conversions using either covariance or contravariance. All types derive from `System.Object`, guaranteeing that certain standard members are available on all variables. We saw how virtual methods allow derived classes to modify selected members of their bases, and how sealing can disable that. We also looked at the relationship between a derived type and its base when it comes to accessing members, and constructors in particular.

Our exploration of inheritance is complete, but it has raised some new issues, such as the relationship between value types and references, and the role of finalizers. So, in the next chapter, I'll talk about the relationship between references and an object's life cycle, along with the way the CLR bridges the gap between references and value types.

[26] More precisely, the same assembly. Chapter 12 describes assemblies.