4 Type Design Guidelines

FROM THE CLR PERSPECTIVE, there are only two categories of types—reference types and value types—but for the purpose of framework design discussion we divide types into more logical groups, each with its own specific design rules. Figure 4-1 shows these logical groups.

Classes are the general case of reference types. They make up the bulk of types in the majority of frameworks. Classes owe their popularity to the rich set of object-oriented features they support and to their general applicability. Base classes and abstract classes are special logical groups related to extensibility. Extensibility and base classes are covered in Chapter 6.

Interfaces are types that can be implemented both by reference types and value types. This allows them to serve as roots of polymorphic hierarchies of reference types and value types. In addition, interfaces can be used to simulate multiple inheritance, which is not natively supported by the CLR.

Structs are the general case of value types and should be reserved for small, simple types, similar to language primitives.

Enums are a special case of value types used to define short sets of values, such as days of the week, console colors, and so on.

Static classes are types intended as containers for static members. They are commonly used to provide shortcuts to other operations.

Delegates, exceptions, attributes, arrays, and collections are all special cases of reference types intended for specific uses, and guidelines for their design and usage are discussed elsewhere in this book.

Abrams_ch04.fm Page 68 Friday, August 19, 2005 7:46 PM

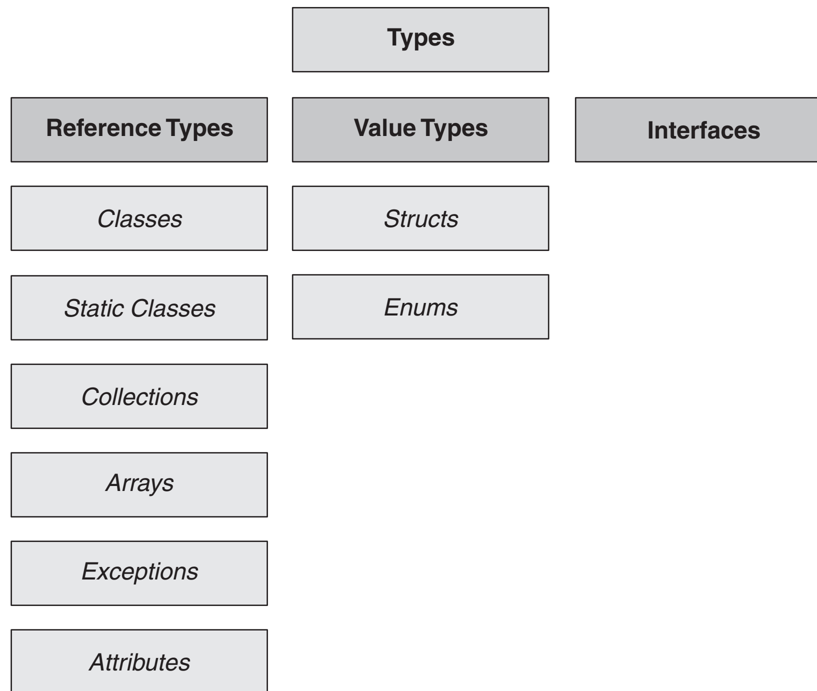


FIGURE 4-1: The logical grouping of types

✓ **DO** ensure that each type is a well-defined set of related members, not just a random collection of unrelated functionality.

It is important that a type can be described in one simple sentence. A good definition should also rule out functionality that is only tangentially related.

> **BRAD ABRAMS** If you have ever managed a team of people you know that they don't do well without a crisp set of responsibilities. Well, types work the same way. I have noticed that types without a firm and focused scope tend to be magnets for more random functionality, which, over time, make a small problem a lot worse. It becomes more difficult to justify why the next member with even more random functionality does not belong in the type. As the focus of the members in a type blurs, the developer's ability to predict where to find a given functionality is impaired, and therefore so is productivity.

Abrams_ch04.fm Page 69 Friday, August 19, 2005 3:35 PM

RICO MARIANI Good types are like good diagrams: What has been omitted is as important to clarity and usability as what has been included. Every additional member you add to a type starts at a net negative value and only by proven usefulness does it go from there to positive. If you add too much in an attempt to make the type more useful to some, you are just as likely to make the type useless to everyone.

JEFFREY RICHTER When I was learning OOP back in the early 1980s, I was taught a mantra that I still honor today: If things get too complicated, make more types. Sometimes, I find that I am thinking really hard trying to define a good set of methods for a type. When I start to feel that I'm spending too much time on this or when things just don't seem to fit together well, I remember my mantra and I define more, smaller types where each type has well-defined functionality. This has worked extremely well for me over the years. On the flip side, sometimes types do end up being dumping grounds for various loosely related functions. The .NET Framework offers several types like this, such as `Marshal`, `GC`, `Console`, `Math`, and `Application`. You will note that all members of these types are static and so it is not possible to create any instances of these types. Programmers seem to be OK with this. Fortunately, these types' methods are separated a bit by types. It would be awful if all these methods were defined in just one type!

4.1 Types and Namespaces

Before designing a large framework you should decide how to factor your functionality into a set of functional areas represented by namespaces. This kind of top-down architectural design is important to ensure a coherent set of namespaces containing types that are well integrated, don't collide, and are not repetitive. Of course the namespace design process is iterative and it should be expected that the design will have to be tweaked as types are added to the namespaces over the course of several releases. This leads to the following guidelines.

✓ **DO** use namespaces to organize types into a hierarchy of related feature areas.

The hierarchy should be optimized for developers browsing the framework for desired APIs.

> **KRZYSZTOF CWALINA** This is an important guideline. Contrary to popular belief, the main purpose of namespaces is not to help in resolving naming conflicts between types with the same name. As the guideline states, the main purpose of namespaces is to organize types in a hierarchy that is coherent, easy to navigate, and easy to understand.
>
> I consider type name conflicts in a single framework to indicate sloppy design. Types with identical names should either be merged to allow for better integration between parts of the library or should be renamed to improve code readability and searchability.

✗ **AVOID** very deep namespace hierarchies. Such hierarchies are difficult to browse because the user has to backtrack often.

✗ **AVOID** having too many namespaces.

Users of a framework should not have to import many namespaces in most common scenarios. Types that are used together in common scenarios should reside in a single namespace if at all possible.

> **JEFFREY RICHTER** As an example of a problem, the runtime serializer types are defined under the `System.Runtime.Serialization` namespace and its subnamespaces. However, the `Serializable` and `NonSerialized` attributes are incorrectly defined in the `System` namespace. Because these types are not in the same namespace, developers don't realize that they are closely related. In fact, I have run into many developers who apply the `Serializable` attribute to a class that they are serializing with the `System.Xml.Serialization`'s `XmlSerializer` type. However, the `XmlSerializer` completely ignores the `Serializable` attribute; applying the attribute gives no value and just bloats your assembly's metadata.

✗ **AVOID** having types designed for advanced scenarios in the same namespace as types intended for common programming tasks.

This makes it easier to understand the basics of the framework and to use the framework in the common scenarios.

Abrams_ch04.fm Page 71 Friday, August 19, 2005 3:35 PM

BRAD ABRAMS One of the best features of Visual Studio is Intellisense, which provides a drop-down for your likely next type or member usage. The benefit of this feature is inversely proportional to the number of options. That is, if there are too many items in the list it takes longer to find the one you are looking for. Following this guideline to split out advanced functionality into a separate namespace enables developers to see the smallest number of types possible in the common case.

BRIAN PEPIN One thing we've learned is that most programmers live or die by Intellisense. If something isn't listed in the drop-down, most programmers won't believe it exists. But, as Brad says above, too much of a good thing can be bad and having too much stuff in the drop-down list dilutes its value. If you have functionality that should be in the same namespace, but you don't think it needs to be shown all the time to users, you can use the `EditorBrowsable` attribute. Put this attribute on a class or member and you can instruct Intellisense to only show the class or member for advanced scenarios.

RICO MARIANI Don't go crazy adding members for every exotic thing someone might want to do with your type. You'll make fatter, uglier assemblies that are hard to grasp. Provide good primitives with understandable limitations. A great example of this is the urge people get to duplicate functionality that is already easy to use via Interop to native. Interop is there for a reason—it's not an unwanted stepchild. When wrapping anything, be sure you are adding plenty of value. Otherwise, the value added by being smaller would have made your assembly more helpful to more people.

JEFFREY RICHTER I agree with this guideline but I'd like to further add that the more advanced classes should be in a namespace that is under the namespace that contains the simple types. For example, the simple types might be in `System.Mail` and the more advanced types should be in `System.Mail.Advanced`.

✗ **DO NOT** define types without specifying their namespaces.

This organizes related types in a hierarchy, and can help to resolve potential type name collisions. Please note that the fact that namespaces

Abrams_ch04.fm Page 72 Friday, August 19, 2005 3:35 PM

can help to resolve name collisions does not mean that such collisions should be introduced. See section 3.3.1 for details.

BRAD ABRAMS It is important to realize that namespaces cannot actually prevent naming collisions but they can significantly reduce them. I could define a class called `MyNamespace.MyType` in an assembly called `MyAssembly`, and define a class with precisely the same name in another assembly. I could then build an application that uses both of these assemblies and types. The CLR would not get confused because the type identity in the CLR is based on strong name (which includes fully qualified assembly name) rather than just the namespace and type name. This can be seen by looking at the C# and ILASM of code creating an instance of `MyType`.

```
C#:
new MyType();

IL:
IL_0000: newobj instance void
[MyAssembly]MyNamespace.MyType::.ctor()
```

Notice that the C# compiler adds a reference to the assembly that defines the type, of the form `[MyAssembly]`, so the runtime always has a disambiguated, fully qualified name to work with.

JEFFREY RICHTER Although what Brad says is true, the C# compiler doesn't let you specify in source code which assembly to pull a type out of, so if you have code that wants to use a type called `MyNamespace.MyType` that exists in two or more assemblies, there is no easy way to do this in C# source code. Prior to C# 2.0, distinguishing between the two types was impossible. However, with C# 2.0, it is now possible using the new extern aliases and namespace qualifier features.

RICO MARIANI Namespaces are a language thing. The CLR doesn't know anything about them really. As far as the CLR is concerned the name of the class really is something like `MyNameSpace.MyOtherNameSpace.MyAmazingType`. The compilers give you syntax (e.g., "using") so that you don't have to type those long class names all the time. So the CLR is never confused about class names because everything is always fully qualified.

Abrams_ch04.fm Page 73 Friday, August 19, 2005 3:35 PM

4.1.1 Standard Subnamespace Names

Types that are rarely used should be placed in subnamespaces to avoid cluttering the main namespaces. We have identified several groups of types that should be separated from their main namespaces.

4.1.1.1 The `.Design` Subnamespace

Design-time-only types should reside in a subnamespace named `.Design`. For example, `System.Windows.Forms.Design` contains `Designers` and related classes used to do design of applications based on `System.Windows.Forms`.

```
System.Windows.Forms.Design
System.Messaging.Design
System.Diagnostics.Design
```

✓ **DO** use a namespace with the ".Design" suffix to contain types that provide design-time functionality for a base namespace.

4.1.1.2 The `.Permissions` Subnamespace

Permission types should reside in a subnamespace named `.Permissions`.

✓ **DO** use a namespace with the ".Permissions" suffix to contain types that provide custom permissions for a base namespace.

> **KRZYSZTOF CWALINA** In the initial design of the .NET Framework namespaces, all types related to a given feature area were in the same namespace. Prior to the first release, we moved design-related types to subnamespaces with the ".Design" suffix. Unfortunately, we did not have time to do it for the `Permission` types. This is a problem in several parts of the Framework. For example, a large portion of the types in the `System.Diagnostics` namespace are types needed for the security infrastructure and very rarely used by the end users of the API.

4.1.1.3 The `.Interop` Subnamespace

Many frameworks need to support interoperability with legacy components. Due diligence should be used in designing interoperability from the

Abrams_ch04.fm Page 74 Friday, August 19, 2005 3:35 PM

ground up. However, the nature of the problem often requires that the shape and style of such interoperability APIs is often quite different from good managed framework design. Thus, it makes sense to put functionality related to interoperation with legacy components in a subnamespace.

You should not put types that completely abstract unmanaged concepts and expose them as managed into the `Interop` subnamespace. It is often the case that managed APIs are implemented by calling out to unmanaged code. For example the `System.IO.FileStream` class calls out to Win32 `CreateFile`. This is perfectly acceptable and does not imply that the `FileStream` class needs to be in `System.IO.Interop` namespace as `FileStream` completely abstracts the Win32 concepts and publicly exposes a nice managed abstraction.

✓ **DO** use a namespace with the ".Interop" suffix to contain types that provide interop functionality for a base namespace.

✓ **DO** use a namespace with the ".Interop" suffix for all code in a Primary Interop Assembly (PIA).

4.2 Choosing Between Class and Struct

One of the basic design decisions every framework designer faces is whether to design a type as a class (a reference type) or as a struct (a value type). Good understanding of the differences in the behavior of reference types and value types is crucial in making this choice.

Reference types are allocated on the heap, and garbage-collected, whereas value types are allocated either on the stack or inline in containing types and deallocated when the stack unwinds or when their containing type gets deallocated. Therefore, allocations and deallocations of value types are in general cheaper than allocations and deallocations of reference types.

Arrays of reference types are allocated out-of-line, meaning the array elements are just references to instances of the reference type residing on the heap. Value type arrays are allocated in-line, meaning that the array elements are the actual instances of the value type. Therefore, allocations and deallocations of value type arrays are much cheaper than allocations and deallocations of reference type arrays. In addition, in a majority of cases value type arrays exhibit much better locality of reference.

Abrams_ch04.fm Page 75 Friday, August 19, 2005 3:35 PM

> **RICO MARIANI** The preceding is often true but it's a very broad generalization that I would be very careful about. Whether or not you get better locality of reference when value types get boxed when cast to an array of value types will depend on how much of the value type you use, how much searching you have to do, how much data reuse there could have been with equivalent array members (sharing a pointer), the typical array access patterns, and probably other factors I can't think of at the moment. Your mileage might vary but value type arrays are a great tool for your toolbox.

Value types get boxed when cast to object or one of the interfaces they implement. They get unboxed when cast back to the value type. Because boxes are objects that are allocated on the heap and are garbage collected, too much boxing and unboxing can have a negative impact on the heap, the garbage collector, and ultimately the performance of the application.

Reference type assignments copy the reference, whereas value type assignments copy the entire value. Therefore assignments of large reference types are cheaper than assignments of large value types.

Finally, reference types are passed by reference, whereas value types are passed by value. Changes to an instance of a reference type affect all references pointing to the instance. Value type instances are copied when they are passed by value. When an instance of a value type is changed, it of course does not affect any of its copies. Because the copies are not created explicitly by the user, but rather implicitly when arguments are passed or return values are returned, value types that can be changed can be confusing to many users. Therefore value types should be immutable.[1]

> **RICO MARIANI** If you make your value type mutable you will find that you end up having to pass it by reference a lot to get the semantics you want (using, e.g., "out" syntax in C#). This might be important in cases in which the value type is expected to be embedded in a variety of other

1. Immutable types are types that don't have any public members that can modify this instance. For example, `System.String` is immutable. Its members, such as `ToUpper`, do not modify the sting on which they are called, but rather return a new modified string and leave the original string unchanged.

Abrams_ch04.fm Page 76 Friday, August 19, 2005 3:35 PM

objects that are themselves reference types or embedded in arrays. The biggest trouble from having mutable value types is where they look like independent entities like, for example, a complex number. Value types that have a mission in life of being an accumulator of sorts or a piece of a reference type have fewer pitfalls for mutability.

As a rule of thumb, the majority of types in a framework should be classes. There are, however, some situations in which the characteristics of a value type make it more appropriate to use structs.

✓ **CONSIDER** defining a struct instead of a class if instances of the type are small and commonly short-lived or are commonly embedded in other objects.

✗ **DO NOT** define a struct unless the type has all of the following characteristics:

- It logically represents a single value, similar to primitive types (`int`, `double`, etc.).
- It has an instance size under 16 bytes.
- It is immutable.
- It will not have to be boxed frequently.

In all other cases, you should define your types as classes.

JEFFREY RICHTER In my opinion, a value type should be defined for types that have approximately 16 bytes or less. Value types can be more than 16 bytes if you don't intend to pass them to other methods or copy them to and from a collection class (like an array). I would also define a value type if you expect instances of the type to be used for short periods of time (usually they are created in a method and no longer needed after a method returns). I used to discourage defining value types if you thought that instances of them would be placed in a collection due to all the boxing that would have to be done. But, fortunately, newer versions of the CLR, C#, and other languages support generics so that boxing is no longer necessary when putting value type instances in a collection.

Abrams_ch04.fm Page 77 Friday, August 19, 2005 3:35 PM

4.3 Choosing Between Class and Interface

In general, classes are the preferred construct for exposing abstractions.

The main drawback of interfaces is that they are much less flexible than classes when it comes to allowing for evolution of APIs. Once you ship an interface, the set of its members is fixed forever. Any additions to the interface would break existing types implementing the interface.

A class offers much more flexibility. You can add members to classes that have already shipped. As long as the method is not abstract (i.e., as long as you provide a default implementation of the method), any existing derived classes continue to function unchanged.

Let's illustrate the concept with a real example from the .NET Framework. The `System.IO.Stream` abstract class shipped in version 1.0 of the framework without any support for timing out pending I/O operations. In version 2.0, several members were added to `Stream` to allow subclasses to support timeout-related operations, even when accessed through their base class APIs.

```
public abstract class Stream {
    public virtual bool CanTimeout {
        get { return false; }
    }
    public virtual int ReadTimeout{
        get{
            throw new NotSupportedException(…);
        {
        set {
            throw new NotSupportedException(…);
        }
    }
}

public class FileStream : Stream {
    public override bool CanTimeout {
        get { return true; }
    }
    public override int ReadTimeout{
        get{
            …
        {
        set {
            …
        }
    }
}
```

Abrams_ch04.fm Page 78 Friday, August 19, 2005 3:35 PM

The only way to evolve interface-based APIs is to add a new interface with the additional members. This might seem like a good option, but it suffers from several problems. Let's illustrate this on a hypothetical IStream interface. Let's assume we had shipped the following APIs in version 1.0 of the Framework.

```
public interface IStream {
    …
}

public class FileStream : IStream {
    …
}
```

If we wanted to add support for timeouts to streams in version 2.0, we would have to do something like the following:

```
public interface ITimeoutEnabledStream : IStream  {
    int ReadTimeout{ get; set; }
}

public class FileStream : ITimeoutEnabledStream {
    public int ReadTimeout{
        get{
            …
        {
        set {
            …
        }
    }
}
```

But now we would have a problem with all the existing APIs that consume and return IStream. For example StreamReader has several constructor overloads and a property typed as Stream.

```
public class StreamReader {
    public StreamReader(IStream stream){ … }
    public IStream BaseStream { get { … } }
}
```

How would we add support for ITimeoutEnabledStream to StreamReader? We would have several options, each with substantial development cost and usability issues:

Abrams_ch04.fm Page 79 Friday, August 19, 2005 3:35 PM

Leave the `StreamReader` as is, and ask users who want to access the timeout-related APIs on the instance returned from `BaseStream` property to use a dynamic cast and query for the `ITimeoutEnabledStream` interface.

```
StreamReader reader = GetSomeReader();
ITimeoutEnabledStream stream = reader.BaseStream as
ITimeoutEnabledStream;
if(stream != null){
    stream.ReadTimeout = 100;
}
```

This option unfortunately does not perform well in usability studies. The fact that some streams can now support the new operations is not immediately visible to the users of `StreamReader` APIs. Also, some developers have difficulties understanding and using dynamic casts.

Add a new property to `StreamReader` that would return `ITimeoutEnabledStream` if one was passed to the constructor or `null` if `IStream` was passed.

```
StreamReader reader = GetSomeReader();
ITimeoutEnabledStream stream = reader.TimeoutEnabledBaseStream;
if(stream!= null){
    stream.ReadTimeout = 100;
}
```

Such APIs are only marginally better in terms of usability. It's really not obvious to the user that the `TimeoutEnabledBaseStream` property getter might return null, which results in confusing and often unexpected `NullReferenceExceptions`.

Add a new type called `TimeoutEnabledStreamReader` that would take `ITimeoutEnabledStream` parameters to the constructor overloads and return `ITimeoutEnabledStream` from the `BaseStream` property. The problem with this approach is that every additional type in the framework adds complexity for the users. What's worse, the solution usually creates more problems like the one it is trying to solve. `StreamReader` itself is used in other APIs. These other APIs will now need new versions that can operate on the new `TimeoutEnabledStreamReader`.

The Framework streaming APIs are based on an abstract class. This allowed for an addition of timeout functionality in version 2.0 of the Framework. The addition is straightforward, discoverable, and had little impact on other parts of the framework.

```
StreamReader reader = GetSomeReader();
if(reader.BaseStream.CanTimeout){
    reader.BaseStream.ReadTimeout = 100;
}
```

One of the most common arguments in favor of interfaces is that they allow separating contract from the implementation. However, the argument incorrectly assumes that you cannot separate contracts from implementation using classes. Abstract classes residing in a separate assembly from their concrete implementations are a great way to achieve such separation. For example, the contract of `IList<T>` says that when an item is added to a collection, the `Count` property is incremented by one. Such a simple contract can be expressed and, what's more important, locked for all subtypes, using the following abstract class:

```
public abstract class CollectionContract<T> : IList<T> {

    public void Add(T item){
        AddCore(item);
        this.count++;
    }
    public int Count {
        get { return this.count; }
    }
    protected abstract void AddCore(T item);
    private int count;
}
```

KRZYSZTOF CWALINA I often hear people saying that interfaces specify contracts. I believe this is a dangerous myth. Interfaces, by themselves, do not specify much beyond the syntax required to use an object. The interface-as-contract myth causes people to do the wrong thing when trying to separate contracts from implementation, which is a great engineering practice. Interfaces separate syntax from implementation, which is not that useful, and the myth provides a false sense of doing the right engineering. In reality, the contract is semantics, and these can actually be nicely expressed with some implementation.

Abrams_ch04.fm Page 81 Friday, August 19, 2005 3:35 PM

COM exposed APIs exclusively through interfaces, but you should not assume that COM did this because interfaces were superior. COM did it because COM is an interface standard that was intended to be supported on many execution environments. CLR is an execution standard and it provides a great benefit for libraries that rely on portable implementation.

✓ **DO** favor defining classes over interfaces.

Class-based APIs can be evolved with much greater ease than interface-based APIs because it is possible to add members to a class without breaking existing code.

KRZYSZTOF CWALINA Over the course of the three versions of the .NET Framework, I have talked about this guideline with quite a few developers on our team. Many of them, including those who initially disagreed with the guideline, have said that they regret having shipped some API as an interface. I have not heard of even one case in which somebody regretted that they shipped a class.

JEFFREY RICHTER I agree with Krzysztof in general. However, you do need to think about some other things. There are some special base classes, such as `MarshalByRefObject`. If your library type provides an abstract base class that isn't itself derived from `MarshalByRefObject`, then types that derive from your abstract base class cannot live in a different `AppDomain`. My recommendation to people is this: Define an interface first and then define an abstract base class that implements the interface. Use the interface to communicate to the object and let end-user developers decide for themselves whether they can just define their own type based on your abstract base class (for convenience) or define their own type based on whatever base class they desire and implement the interface (for flexibility). A good example of this is the `IComponent` interface and the `Component` base class that implements `IComponent`.

✓ **DO** use abstract classes instead of interfaces to decouple the contract from implementations.

Abstract classes, if designed correctly, allow for the same degree of decoupling between contract and implementation.

Abrams_ch04.fm Page 82 Friday, August 19, 2005 3:35 PM

CHRIS ANDERSON Here is one instance in which the design guideline, if followed too strictly, can paint you into a corner. Abstract types do version much better, and allow for future extensibility, but they also burn your one and only one base type. Interfaces are appropriate when you are really defining a contract between two objects that is invariant over time. Abstract base types are better for defining a common base for a family of types. When we did .NET there was somewhat of a backlash against the complexity and strictness of COM—interfaces, Guids, variants, and IDL, were all seen as bad things. I believe today that we have a more balanced view of this. All of these COMisms have their place, and in fact you can see interfaces coming back as a core concept in Indigo.

BRIAN PEPIN One thing I've started doing is to actually bake as much contract into my abstract class as possible. For example, I might want to have four overloads to a method where each overload offers an increasingly complex set of parameters. The best way to do this is to provide a nonvirtual implementation of these methods on the abstract class, and have the implementations all route to a protected abstract method that provides the actual implementation. By doing this, you can write all the boring argument-checking logic once. Developers who want to implement your class will thank you.

✓ **DO** define an interface if you need to provide a polymorphic hierarchy of value types.

Value types cannot inherit from other types, but they can implement interfaces. For example, `IComparable`, `IFormattable`, and `IConvertible` are all interfaces so value types such as `Int32`, `Int64`, and other primitives can all be comparable, formattable, and convertible.

```
public struct Int32 : IComparable, IFormattable, IConvertible {
    …
}
public struct Int64 : IComparable, IFormattable, IConvertible {
    …
}
```

✓ **CONSIDER** defining interfaces to achieve a similar effect to that of multiple inheritance.

Abrams_ch04.fm Page 83 Friday, August 19, 2005 3:35 PM

RICO MARIANI Good interface candidates often have this "mix in" feel to them. All sorts of objects can be `IFormattable`—it isn't restricted to a certain subtype. It's more like a type attribute. Other times we have interfaces that look more like they should be classes—`IFormatProvider` springs to mind. The fact that the best interface name ended in "er" speaks volumes.

BRIAN PEPIN Another sign that you've got a well-defined interface is that the interface does exactly one thing. If you have an interface that has a grab bag of functionality, that's a warning sign. You'll end up regretting it because in the next version of your product you'll want to add new functionality to this rich interface, but you can't.

For example, `System.IDisposable` and `System.ICloneable` are both interfaces so types, like `System.Drawing.Image`, can be both disposable, cloneable, and still inherit from `System.MarshalByRefObject` class.

```
public class Image : MarshalByRefObject, IDisposable, ICloneable {
…
}
```

JEFFREY RICHTER When a class is derived from a base class, I say that the derived class has an IS-A relationship with the base. For example, a `FileStream` IS-A `Stream`. However, when a class implements an interface, I say that the implementing class has a CAN-DO relationship with the interface. For example, a `FileStream` CAN-DO disposing.

4.4 Abstract Class Design

✗ **DO NOT** define public or protected-internal constructors in abstract types.

Constructors should be public only if users will need to create instances of the type. Because you cannot create instances of an abstract type, an

Abrams_ch04.fm Page 84 Friday, August 19, 2005 3:35 PM

abstract type with a public constructor is incorrectly designed and misleading to the users.[2]

```
// bad design
public abstract class Claim {
    public Claim (int number) {
    }
}
// good design
public abstract class Claim {
    // incorrect Design
    protected Claim (int number) {
    }
}
```

✓ **DO** define a protected or an internal constructor on abstract classes.

A protected constructor is more common and simply allows the base class to do its own initialization when subtypes are created.

```
public abstract class Claim {
    protected Claim() {
        …
    }
}
```

An internal constructor can be used to limit concrete implementations of the abstract class to the assembly defining the class.

```
public abstract class Claim {
    internal Claim() {
        …
    }
}
```

> **BRAD ABRAMS** Many languages (such as C#) will insert a protected constructor if you do not. It is a good practice to define the constructor explicitly in the source so that it can be more easily documented and maintained over time.

✓ **DO** provide at least one concrete type that inherits from each abstract class that you ship.

2. This also applies to protected-internal constructors.

Abrams_ch04.fm Page 85 Friday, August 19, 2005 3:35 PM

This helps to validate the design of the abstract class. For example, System.IO.FileStream is an implementation of the System.IO.Stream abstract class.

> **BRAD ABRAMS** I have seen countless examples of a "well-designed" base class or interface where the designers spent hundreds of hours debating and tweaking the design only to have it blown out of the water when the first real-world client came to use the design. Far too often these real-world clients come too late in the product cycle to allow time for the correct fix. Forcing yourself to provide at least one concrete implementation reduces the chances of finding a new problem late in the product cycle.

4.5 Static Class Design

A static class is defined as a class that contains only static members (of course besides the instance members inherited from System.Object and possibly a private constructor). Some languages provide built-in support for static classes. In C# 2.0, when a class is declared to be static, it is sealed, abstract, and no instance members can be overridden or declared.

```
public static class File {
    …
}
```

If your language does not have built-in support for static classes, you can declare such classes manually as in the following C++ example:

```
public class File abstract sealed {
    …
}
```

Static classes are a compromise between pure object-oriented design and simplicity. They are commonly used to provide shortcuts to other operations (such as System.IO.File), or functionality for which a full object-oriented wrapper is unwarranted (such as System.Environment).

✓ **DO** use static classes sparingly.

Abrams_ch04.fm Page 86 Friday, August 19, 2005 3:35 PM

Static classes should be used only as supporting classes for the object-oriented core of the framework.

✗ **DO NOT** treat static classes as a miscellaneous bucket.

There should be a clear charter for the class.

✗ **DO NOT** declare or override instance members in static classes.

✓ **DO** declare static classes as sealed, abstract, and add a private instance constructor, if your programming language does not have built-in support for static classes.

BRIAN GRUNKEMEYER In the .NET Framework 1.0, I wrote the code for the `System.Environment` class, which is an excellent example of a static class. I messed up and accidentally added a property to this class that wasn't static (`HasShutdownStarted`). Because it was an instance method on a class that could never be instantiated, no one could call it. We didn't discover the problem early enough in the product cycle to fix it before releasing version 1.0.

If I were inventing a new language, I would explicitly add the concept of a static class into the language to help people avoid falling into this trap. And in fact, in C# 2.0 did add support for static classes!

JEFFREY RICHTER Make sure that you do not attempt to define a static structure, because structures (value types) can always be instantiated no matter what. Only classes can be static.

4.6 Interface Design

Although most APIs are best modeled using classes and structs, there are cases in which interfaces are more appropriate or are the only option.

The CLR does not support multiple inheritance (i.e., CLR classes cannot inherit from more than one base class), but it does allow types to implement one or more interfaces in addition to inheriting from a base class. Therefore interfaces are often used to achieve the effect of multiple inheritance. For example, `IDisposable` is an interface that allows types to sup-

Abrams_ch04.fm Page 87 Friday, August 19, 2005 3:35 PM

port disposability independent of any other inheritance hierarchy in which they want to participate.

```
public class Component : MarshalByRefObject, IDisposable, IComponent {
…
}
```

The other situation in which defining an interface is appropriate is in creating a common interface that can be supported by several types including some value types. Value types cannot inherit from types other than `System.ValueType`, but they can implement interfaces, so using an interface is the only option to provide a common base type.

```
public struct Boolean : IComparable {
    …
}
public class String: IComparable {
    …
}
```

✓ **DO** define an interface if you need some common API to be supported by a set of types that includes value types.

✓ **CONSIDER** defining an interface if you need to support its functionality on types that already inherit from some other type.

✓ **AVOID** using marker interfaces (interfaces with no members).

If you need to mark a class as having a specific characteristic (marker), in general, use a custom attribute rather than an interface.

```
// Avoid
public interface IImmutable {} // empty interface

public class Key: IImmutable {
    …
}

//Do
[Immutable]
public class Key {
    …
}
```

Abrams_ch04.fm Page 88 Friday, August 19, 2005 3:35 PM

Methods can be implemented to reject parameters that are not marked with a specific attribute as follows:

```
public void Add(Key key, object value){
    if(!key.GetType().IsDefined(typeof(ImmutableAttribute),
false)){
        throw new ArgumentException("The parameter must be
immutable","key");
    }
    …
}
```

RICO MARIANI Of course any kind of marking like this has a cost. Attribute testing is a lot more costly than type checking. You might find that it's necessary to use the marker interface approach for performance reasons—measure and see. My own experience is that true markers (with no members) don't come up very often. Most of the time, you need a no-kidding-around interface with actual functionality to do the job, in which case there is no choice to make.

The problem with this approach is that the check for the custom attribute can occur only at runtime. Sometimes, it is very important that the check for the marker be done at compile-time. For example, a method that can serialize objects of any type might be more concerned with verifying the presence of the marker than with type verification at compile-time. Using marker interfaces might be acceptable in such situations. The following example illustrates this design approach:

```
public interface ITextSerializable {} // empty interface
public void Serialize(ITextSerializable item){
    // use reflection to serialize all public properties
    …
}
```

✓ **DO** provide at least one type that is an implementation of an interface.

This helps to validate the design of the interface. For example, `System.Collections.ArrayList` is an implementation of the `System.Collections.IList` interface.

✓ **DO** provide at least one API consuming each interface you define (a method taking the interface as a parameter or a property typed as the interface).

This helps to validate the interface design. For example, `List<T>.Sort` consumes `IComparer<T>` interface.

✗ **DO NOT** add members to an interface that has previously shipped.

Doing so would break implementations of the interface. You should create a new interface to avoid versioning problems.

Except for the situations described in these guidelines, you should, in general, choose classes rather than interfaces in designing managed code reusable libraries.

4.7 Struct Design

The general-purpose value type is most often referred to as a struct, its C# keyword. This section provides some guidelines for general struct design. Section 4.8 presents guidelines for the design of a special case of value type, the enum.

✗ **DO NOT** provide a default constructor for a struct.

This allows arrays of structs to be created without having to run the constructor on each item of the array. Notice that C# does not allow structs to have default constructors.

✓ **DO** ensure that a state where all instance data is set to zero, false, or null (as appropriate) is valid.

This prevents accidental creation of invalid instances when an array of the structs is created. For example, the following struct is incorrectly designed. The parameterized constructor is meant to ensure valid state, but the constructor is not executed when an array of the struct is created and so the instance filed value gets initialized to 0, which is not a valid value for this type.

```
// bad design
public struct PositiveInteger {
    int value;
```

Abrams_ch04.fm Page 90 Friday, August 19, 2005 3:35 PM


```
    public PositiveInteger(int value) {
        if (value <= 0) throw new ArgumentException(…);
        this.value = value;
    }

    public override string ToString() {
        return value.ToString();
    }
}
```

The problem can be fixed by ensuring that the default state (in this case the value field equal to 0) is a valid logical state for the type.

```
// good design
public struct PositiveInteger {
    int value; // the logical value is value+1

    public PositiveInteger(int value) {
        if (value <= 0) throw new ArgumentException(…);
        this.value = value-1;
    }

    public override string ToString() {
        return (value+1).ToString();
    }
}
```

✓ **DO** implement `IEquatable<T>` on value types.

The `Object.Equals` method on value types causes boxing and its default implementation is not very efficient, as it uses reflection. `IEquatable<T>.Equals` can have much better performance and can be implemented such that it will not cause boxing. See Chapter 8, section 8.5, on implementing `IEquatable<T>`.

✗ **DO NOT** explicitly extend `System.ValueType`. In fact, most languages prevent THIS.

In general, structs can be very useful, but should only be used for small, single, immutable values that will not be boxed frequently. Next are guidelines for enum design, a more complex matter.

Abrams_ch04.fm Page 91 Friday, August 19, 2005 3:35 PM

4.8 Enum Design

Enums are a special kind of value type. There are two kinds of enums: simple enums and flag enums.

Simple enums represent small, closed sets of choices. A common example of the simple enum is a set of colors. For example,

```
public enum Color {
    Red,
    Green,
    Blue,
    …
}
```

Flag enums are designed to support bitwise operations on the enum values. A common example of the flags enum is a list of options. For example,

```
[Flags]
public enum AttributeTargets {
    Assembly= 0x0001,
    Module  = 0x0002,
    Cass = 0x0004,
    Struct = 0x0008,
    …
}
```

BRAD ABRAMS We had some debates about what to call enums that are designed to be bitwise ORed together. We considered bitfields, bitflags, and even bitmasks, but ultimately decided to use flag enums as it was clear, simple, and approachable.

STEVEN CLARKE I'm sure that less experienced developers will be able to understand bitwise operations on flags. The real question, though, is whether they would expect to have to do this. Most of the APIs that I have run through the labs don't require them to perform such operations so I have a feeling that they would have the same experience that we observed during a recent study—it's just not something that they are used to doing so they might not even think about it.

Abrams_ch04.fm Page 92 Friday, August 19, 2005 3:35 PM

Where it could get worse, I think, is that if less advanced developers don't realize they are working with a set of flags that can be combined with one another, they might just look at the list available and think that is all the functionality they can access. As we've seen in other studies, if an API makes it look to them as though a specific scenario or requirement isn't immediately possible, it's likely that they will change the requirement and do what does appear to be possible, rather than being motivated to spend time investigating what they need to do to achieve the original goal.

Historically, many APIs (e.g., Win32 APIs) represented sets of values using integer constants. Enums make such sets more strongly typed, and thus improve compile-time error checking, usability, and readability. For example, use of enums allows development tools to know the possible choices for a property or a parameter.

✓ **DO** use an enum to strongly type parameters, properties, and return values that represent sets of values.

✓ **DO** favor using an enum over static constants.

```
// Avoid the following
public static class Color {
    public static int Red    = 0;
    public static int Green  = 1;
    public static int Blue   = 2;
    …
}

// Favor the following
public enum Color {
    Red,
    Green,
    Blue,
    …
}
```

JEFFREY RICHTER An enum is a structure with a set of static constants. The reason to follow this guideline is because you will get some additional compiler and reflection support if you define an enum versus manually defining a structure with static constants.

Abrams_ch04.fm Page 93 Friday, August 19, 2005 3:35 PM

✗ **DO NOT** use an enum for open sets (such as the operating system version, names of your friends, etc.).

✗ **DO NOT** provide reserved enum values that are intended for future use.

You can always simply add values to the existing enum at a later stage. See section 4.8.2 for more details on adding values to enums. Reserved values just pollute the set of real values and tend to lead to user errors.

```
public enum DeskType {
    Circular,
    Oblong,
    Rectangular,

    // the following two  values should not be here
    ReservedForFutureUse1,
    ReservedForFutureUse2,
}
```

✗ **AVOID** publicly exposing enums with only one value.

A common practice for ensuring future extensibility of C APIs is to add reserved parameters to method signatures. Such reserved parameters can be expressed as enums with a single default value. This should not be done in managed APIs. Method overloading allows adding parameters in future releases.

```
// Bad Design
public enum SomeOption {
   DefaultOption
   // we will add more options in the future
}

…

// The option parameter is not needed.
// It can always be added in the future
// to an overload of SomeMethod().
public void SomeMethod(SomeOption option) {
    …
}
```

✗ **DO NOT** include sentinel values in enums.

Although they are sometimes helpful to framework developers, they are confusing to users of the framework. Sentinel values are values

Abrams_ch04.fm Page 94 Friday, August 19, 2005 3:35 PM

used to track the state of the enum, rather than being one of the values from the set represented by the enum. The following example shows an enum with an additional sentinel value used to identify the last value of the enum, and intended for use in range checks. This is bad practice in framework design.

```
public enum DeskType {
   Circular    = 1,
   Oblong      = 2,
   Rectangular = 3,

   LastValue   = 3 // this sentinel value should not be here
}

public void OrderDesk(DeskType desk){
    if((desk > DeskType.LastValue){
        throw new ArgumentOutOfRangeException(…);
    }
    …
}
```

Rather than relying on sentinel values, framework developers should perform the check using one of the real enum values.

```
public void OrderDesk(DeskType desk){
    if(desk > DeskType.Rectangular || desk < DeskType.Circular){
        throw new ArgumentOutOfRangeException(…);
    }
    …
}
```

RICO MARIANI You can get yourself into a lot of trouble by trying to be too clever with enums. Sentinel values are a great example of this: People write code like the above but using the sentinel value `LastValue` instead of `Rectangular` as recommended. When a new value comes along and `LastValue` is updated, their program "automatically" does the right thing and accepts the new input value without giving an `ArgumentOutOfRangeException`. That sounds grand except for all that we didn't show, the part that's doing the actual work, and might not yet expect or even handle the new value. The less clever tests will force you to revisit all the right places to ensure that the new value really is going to work. The few minutes you spend visiting those call sites will be more than repaid in time you save avoiding bugs.

✓ **DO** provide a value of zero on simple enums.

Consider calling the value something like "None." If such value is not appropriate for this particular enum, the most common default value for the enum should be assigned the underlying value of zero.

```
public enum Compression {
    None = 0,
    GZip,
    Deflate,
}
public enum EventType {
    Error = 0,
    Warning,
    Information,
    …
}
```

✓ **CONSIDER** using `Int32` (the default in most programming languages) as the underlying type of an enum unless any of the following is true:

- The enum is a flags enum and you have more than 32 flags, or expect to have more in the future.

> **BRAD ABRAMS** This might not be as uncommon a concern as you first expect. We are only in version 2.0 of the .NET Framework and we are already running out of values in the `CodeDom GeneratorSupport` enum. In retrospect, we should have used a different mechanism for communicating the generator support options than an enum.

> **RICO MARIANI** Did you know that the CLR supports enums with an underlying type of `float` or `double` even though most languages don't choose to expose it? This is very handy for strongly typed constants that happen to be floating point (e.g., a set of canonical conversion factors for different measuring systems). It's in the ECMA standard.

- The underlying type needs to be different than `Int32` for easier interoperability with unmanaged code expecting different size enums.

Abrams_ch04.fm Page 96 Friday, August 19, 2005 3:35 PM

- A smaller underlying type would result in substantial savings in space. If you expect for enum to be used mainly as an argument for flow of control, the size makes little difference. The size savings might be significant if:
  - You expect the enum to be used as a field in a very frequently instantiated structure or class.
  - You expect users to create large arrays or collections of the enum instances.
  - You expect a large number of instances of the enum to be serialized.

For in-memory usage, be aware that managed objects are always DWORD aligned so you effectively need multiple enums or other small structures in an instance to pack a smaller enum with to make a difference, as the total instance size is always going to be rounded up to a DWORD.

> **BRAD ABRAMS** Keep in mind that it is a binary breaking change to change the size of the enum type once you have shipped, so choose wisely with an eye on the future. Our experience is that `Int32` is usually the right choice and thus we made it the default.

✓ **DO** name flag enums with plural nouns or noun phrases and simple enums with singular nouns or noun phrases.

See Chapter 3, section 3.5.3 for details.

✗ **DO NOT** extend `System.Enum` directly.

`System.Enum` is a special type used by the CLR to create user-defined enumerations. Most programming languages provide a programming element that gives you access to this functionality. For example, in C# the *enum* keyword is used to define an enumeration.

Abrams_ch04.fm Page 97 Friday, August 19, 2005 3:35 PM

4.8.1 Designing Flag Enums

> **JEFFREY RICHTER** I use flag enums quite frequently in my own programming. They store very efficiently in memory and manipulation is very fast. In addition, they can be used with interlocked operations, making them ideal for solving thread synchronization problems. I'd love to see the `System.Enum` type offer a bunch of additional methods that could be easily inlined by the JIT compiler that would make source code easier to read and maintain. Here are some of the methods I'd like to see added to Enum: `IsExactlyOneBitSet`, `CountOnBits`, `AreAllBitsOn`, `AreAnyBitsOn`, and `TurnBitsOnOff`.

✓ **DO** apply the `System.FlagsAttribute` to flag enums. Do not apply this attribute to simple enums.

```
[Flags]
public enum AttributeTargets {
    …
}
```

✓ **DO** use powers of two for the flags enum values so they can be freely combined using the bitwise OR operation.

```
[Flags]
public enum WatcherChangeTypes {
    Created = 0x0002,
    Deleted = 0x0004,
    Changed = 0x0008,
    Renamed = 0x0010,
}
```

✓ **CONSIDER** providing special enum values for commonly used combinations of flags.

Bitwise operations are an advanced concept and should not be required for simple tasks. `FileAccess.ReadWrite` is an example of such a special value.

```
[Flags]
public enum FileAccess {
    Read = 1,
    Write = 2,
    ReadWrite = Read | Write
}
```

✗ **AVOID** creating flag enums where certain combinations of values are invalid.

The `System.Reflection.BindingFlags` enum is an example of an incorrect design of this kind. The enum tries to represent many different concepts, such as visibility, staticness, member kind, and so on.

```
[Flags]
public enum BindingFlags {
    Instance,
    Static,

    NonPublic,
    Public,

    CreateInstance,
    GetField,
    SetField,
    GetProperty,
    SetProperty,
    InvokeMethod,
    …
}
```

Certain combinations of the values are not valid. For example, the `Type.GetMembers` method accepts this enum as a parameter but the documentation for the method warns users, "You must specify either `BindingFlags.Instance` or `BindingFlags.Static` in order to get a return." Similar warnings apply to several other values of the enum.

If you have an enum with this problem, you should separate the values of the enum into two or more enums or other types. For example, the Reflection APIs could have been designed as follows:

```
[Flags]
public enum Visibilities {
    Public,
    NonPublic
}

[Flags]
public enum MemberScopes {
    Instance,
    Static
}

[Flags]
public enum MemberKinds {
```

Abrams_ch04.fm Page 99 Friday, August 19, 2005 3:35 PM


```
    Constructor,
    Field,
    PropertyGetter,
    PropertySetter,
    Method,
}

public class Type {
    public MemberInfo[] GetMembers(MemberKinds members,
                                   Visibilities visibility,
                                   MemberScopes scope);
}
```

✗ **AVOID** using flag enum values of zero, unless the value represents "all flags are cleared" and is named appropriately as prescribed by the following guideline.

The following example shows a common implementation of a check that programmers use to determine if a flag is set (see the if-statement below). The check works as expected for all flag enum values except the value of zero, where the Boolean expression always evaluates to true.

```
[Flags]
public enum SomeFlag {
    ValueA = 0,  // this might be confusing to users
    ValueB = 1,
    ValueC = 2,
    ValueBAndC = ValueB | ValueC,
}

SomeFlag flags = GetValue();
if ((flags & SomeFlag.ValueA) === SomeFlag.ValueA) {
    …
}
```

ANDERS HEJLSBERG Note that in C# the literal constant 0 implicitly converts to any enum type, so you could just write:

```
if (Foo.SomeFlag == 0)…
```

We support this special conversion to provide programmers with a consistent way of writing the default value of an enum type, which by CLR decree is "all bits zero" for any value type.

✓ **DO** name the zero-value of flag enums `None`. For a flag enum, the value must always mean "all flags are cleared."

```
[Flags]
public enum BorderStyle {
    Fixed3D          = 0x1,
    FixedSingle      = 0x2,
    None             = 0x0
}
if (foo.BorderStyle == BorderStyle.None)....
```

VANCE MORRISON The rationale for avoiding zero in a flag enumeration for an actual flag (only the special enum member `None` should have the value zero) is that you can't OR it in with other flags as expected.

However, notice that this rule only applies to flag enumerations; in the case where enumeration is not a flag enumeration, there is a real disadvantage to avoiding zero that we have discovered. All enumerations begin their life with this value (memory is zeroed by default). Thus if you avoid zero, every enumeration has an illegal value in it when it first starts its life in the run time (we can't even pretty print it properly). This seems bad.

In my own coding, I do one of two things for nonflag enumerations.

If there is an obvious default that is highly unlikely to cause grief if programmers forget to set it (program invariants do not depend on it), I make this the zero case. Usually this is the most common value, which makes the code a bit more efficient (it is easier to set memory to 0 than to any other value).

If no such default exists, I make zero my "error" (none-of-the-above) enumeration value. That way when people forget to set it, some assert will fire later and we will find the problem.

In either case, however, from the compiler (and runtime), point of view, every enumeration has a value for 0 (which means we can pretty print it).

4.8.2 Adding Values to Enums

It is very common to discover that you need to add values to an enum after you have already shipped it. There is a potential application compatibility problem when the newly added value is returned from an existing API, because poorly written applications might not handle the new value correctly. Documentation, samples, and FxCop rules encourage application developers to write robust code that can help applications deal with unexpected values. Therefore, it is generally acceptable to add values to enums,

Abrams_ch04.fm Page 101 Friday, August 19, 2005 3:35 PM

but as with most guidelines there might be exceptions to the rule based on the specifics of the framework.

✓ **CONSIDER** adding values to enums, despite a small compatibility risk.

If you have real data about application incompatibilities caused by additions to an enum, consider adding a new API that returns the new and old values, and deprecate the old API, which should continue returning just the old values. This will ensure that your existing applications remain compatible.

CLEMENS SZYPERSKI Adding a value to an enum presents a very real possibility of breaking a client. Before the addition of the new enum value, a client who was throwing unconditionally in the default case presumably never actually threw the exception, and the corresponding catch path is likely untested. Now that the new enum value can pop up, the client will throw and likely fold.

The biggest concern with adding values to enums is that you don't know whether clients perform an exhaustive switch over an enum or a progressive case analysis across wider-spread code. Even with the FxCop rules above in place and even when it is assumed that client apps pass FxCop without warnings, we still would not know about code that performs things like `if (myEnum == someValue) ...` in various places.

Clients might instead perform point-wise case analyses across their code, resulting in fragility under enum versioning. It is important to provide specific guidelines to developers of enum client code detailing what they need to do to survive the addition of new elements to enums they use. Developing with the suspected future versioning of an enum in mind is the required attitude.

4.9 Nested Types

A nested type is a type defined within the scope of another type, which is called the enclosing type. A nested type has access to all members of its enclosing type. For example, it has access to private fields defined in the enclosing type and to protected fields defined in all ascendants of the enclosing type.

Abrams_ch04.fm Page 102 Friday, August 19, 2005 3:35 PM


```
// enclosing type
public class OuterType {
    private string name;

    // nested type
    public class InnerType {
        public InnerType(OuterType outer){
            // the name field is private, but it works just fine
            Console.WriteLine(outer.name);
        }
    }
}
```

In general, nested types should be used sparingly. There are several reasons for this. Some developers are not fully familiar with the concept. These developers might, for example, have problems with the syntax of declaring variables of nested types. Nested types are also very tightly coupled with their enclosing types, and as such are not suited to be general-purpose types.

Nested types are best suited for modeling implementation details of their enclosing types. The end user should rarely have to declare variables of a nested type and almost never explicitly instantiate nested types. For example, the enumerator of a collection can be a nested type of that collection. Enumerators are usually instantiated by their enclosing type and because many languages support the foreach statement, enumerator variables rarely have to be declared by the end user.

✓ **DO** use nested types when the relationship between the nested type and its outer type is such that member-accessibility semantics are desirable.

For example, the nested type needs to have access to private members of the outer-type.

```
public OrderCollection : IEnumerable<Order> {
    Order[] data = …;

    public IEnumerator<Order> GetEnumerator(){
        return new OrderEnumerator(this);
    }
```

Abrams_ch04.fm Page 103 Friday, August 19, 2005 3:35 PM


```
        // This nested type will have access to the data array
        // of its outer type.
        class OrderEnumerator : IEnumerator<Order> {
        }
    }
```

✗ **DO NOT** use public nested types as a logical grouping construct; use namespaces for this.

✗ **AVOID** publicly exposed nested types. The only exception to this is if variables of the nested type need to be declared only in rare scenarios such as subclassing or other advanced customization scenarios.

> **KRZYSZTOF CWALINA** The main motivation for this guideline is that many less skilled developers don't understand why some type names have dots in them and some don't. As long as they don't have to type in the type name, they don't care. But the moment you ask them to declare a variable of a nested type, they get lost. Therefore, we, in general, avoid nested types and use them only in places where developers almost never have to declare variables of that type (e.g., collection enumerators).

✗ **DO NOT** use nested types if the type is likely to be referenced outside of the containing type.

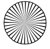
For example, an enum passed to a method defined on a class should not be defined as a nested type in the class.

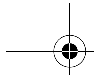
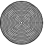
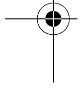
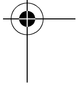
✗ **DO NOT** use nested types if they need to be instantiated by client code. If a type has a public constructor, it should probably not be nested.

If a type can be instantiated, it seems to indicate that the type has a place in the framework on its own (you can create it, work with it, and destroy it, without ever using the outer type), and thus should not be nested. Inner types should not be widely reused outside of the outer type without any relationship whatsoever to the outer type.

✗ **DO NOT** define a nested type as a member of an interface. Many languages do not support such a construct.

In general, nested types should be used sparingly, and exposure as public types should be avoided.



Abrams_ch04.fm Page 104 Friday, August 19, 2005 3:35 PM





4.10 Summary

This chapter presented guidelines that describe when and how to design classes, structs, and interfaces.

The next chapter goes to the next level in type design—the design of members.