Chapter 6

Types

A Microsoft .NET Framework type is a more general concept than a .NET class. More specifically, a class is what is also called a *reference type*, whereas a structure is known as a *value type*. The difference between these two flavors of the type concept is particularly important and is thoroughly explained in many articles and books. In this chapter, we focus on *when* you should render a type as a class or a structure and also cover many other guidelines related to types, including how to name them, how to group their members, how to apply scope qualifiers, and when to use nested types.

Note One or more code examples in this chapter assume that the following namespace has been imported by means of an *Imports* (Visual Basic) or *using* (C#) statement:

```
System.Diagnostics
```

6.1 Type names

Use the following guidelines for type names:

a. Use PascalCase for type names. Example: Customer, ForeignCustomer.

b. Don't use underscores inside type names.

c. Try to avoid class and structure names with a leading I character to minimize confusion with interface types; use casing if you can't help having a name beginning with the I character. Example: Invoice.

6.2 U.S. English for identifiers

Use American English for type and member names. For example, use Color (U.S. English) rather than Colour (U.K. English).

Why: Developers outside the United States might consider this rule as quite arbitrary and too U.S.-centric, but it undoubtedly has one important benefit: type and member U.S. English names can look like names used in the .NET Framework, thus many developers will feel at ease with these identifiers.

6.3 Abbreviations and acronyms

Follow these guidelines to decide how you should render abbreviations and acronyms:

a. Don't use abbreviations in type and member names.

b. Use acronyms only if they are well known among the developer community.

c. Use all-uppercase style for acronyms of two characters.

d. Use PascalCase for acronyms with three characters or more.

Example: UIThread, AsciiDocument, HtmlParser.

More details: Even though these guidelines come from Microsoft, notice that some class names in the .NET Framework incorrectly use uppercase for acronyms, for example, ASCIIEncoding and CLSCompliant.

6.4 Words to avoid

Avoid using the words listed in Table 6-1 as type or member names. The table includes all Visual Basic and C#, including keywords from their 2005 versions (in beta version as of this writing).

Table 6-1 Words to Avoid

abstract	AddHandler	AddressOf	Alias	And
AndAlso	Ansi	As	Assembly	Auto
Base	bool	Boolean	break	ByRef
Byte	ByVal	Call	Case	Catch
CBool	CByte	CChar	CDate	CDec
CDbl	Char	checked	CInt	Class
CLng	CObj	Const	continue	CSByte
CShort	CSng	CStr	CType	CUInteger
CULong	CUShort	Custom	Date	Decimal
Declare	Default	Delegate	Dim	Do
Double	Each	Else	ElseIf	End
Enum	Erase	Error	eval	Event
Exit	explicit	extends	extern	ExternalSource
False	Finalize	Finally	fixed	float
For	foreach	Friend	Function	Get
GetType	Global	Goto	Handles	If
Implements	implicit	Imports	In	Inherits
instanceof	int	Integer	Interface	internal
Is	IsFalse	IsNot	IsTrue	Let
Lib	Like	lock	Long	Loop
Me	Mod	Module	MustInherit	MustOverride
My	MyBase	MyClass	Namespace	Narrowing
New	Next	Not	Nothing	NotInheritable
NotOverridable	null	Object	Of	On

Table 6-1 Words to Avoid

operator	Option	Optional	Or	OrElse
out	Overloads	Overridable	override	Overrides
package	ParamArray	params	Partial	Preserve
Private	Property	Protected	Public	RaiseEvent
ReadOnly	ReDim	ref	Region	Rem
RemoveHandler	Resume	Return	sbyte	sealed
Select	Set	Shadows	Shared	Short
Single	sizeof	stackalloc	Static	Step
Stop	String	struct	Structure	Sub
switch	SyncLock	Then	this	Throw
To	True	Try	TryCast	TypeOf
uint	UInteger	ulong	ushort	unchecked
Unicode	unsafe	Until	using	var
virtual	void	volatile	When	While
Widening	With	WithEvents	WriteOnly	Xor
yield				

More details: You should also avoid names that match .NET Framework namespaces, such as System, Forms, Web, UI, Collections, Win32, and so on.

6.5 One type per source file

Each source file should contain only one type definition.

Exception: See rules 6.7, 8.2, and 10.1 for exceptions to this rule.

6.6 Type complexity

Consider splitting a type with many methods and properties into one main class and one or more dependent classes. A simple rule of thumb is to consider a type as a candidate for splitting when it exposes many properties or methods with similar names.

Why: Code in smaller types can be reused more easily.

Example: Let's say you have a Person class that exposes many similar properties such as HomeAddress, HomeCity, HomeZipCode, HomeState, and HomeCountry. Such an entity can be better represented by two distinct types, Person and Location, where Person exposes one property of type Location.

```
' [Visual Basic]
Public Class Person
   Public Property Home() As Location
      ...
   End Property
End Class
```

```
// [C#]
public class Person
{
   public Location Home
   {
      ...
   }
}
```

The benefit of having a separate Location type becomes apparent if you later decide to implement other similar properties, such as Work (for the office's address), Vacation, and so forth.

6.7 Dependency on external variables

Ensure that code inside a type doesn't reference any external variable, including global variables defined in a module (Visual Basic) or static fields or properties exposed by another type. All the values that a type needs should be assigned to the type's properties or passed as arguments to its constructors or methods.

Why: Self-containment is the key to code reuse: if the type doesn't depend on any other type, you can drop it in a different project with fewer or no side effects. Besides, the code in the type can't be broken by accidentally or purposely assigning an invalid value to the external variable, and you don't have to synchronize access to the global variable in a multithreaded environment.

Why not: In many complex object hierarchies, you must be prepared to relax this rule to an extent. For example, you might decide to have one class that contains all the configuration settings of the assembly, in which case many types in the assembly have to read values from static fields of the configuration class.

More details: If you can't help breaking this rule, you can at least ensure that external values are implemented as properties (as opposed to fields) so that you can validate them and ensure that they are never invalid. Also, if just two or three types depend on one another, you can mitigate this rule by having all of them stored in the same source file.

6.8 Module names [Visual Basic]

Don't use any special suffix or naming convention for Visual Basic modules.

Why: Modules are just types whose members are all static, so they shouldn't be dealt with in a different way.

6.9 App type or module

Use App name for the module (Visual Basic) or the class (C#) that contains the Main procedure. Don't place the Main procedure in a Windows Forms class.

6.10 Globals class

Use Globals for the type that contains all the global variables of the current application. Global variables are implemented as static fields of this type. (Visual Basic developers can also use a module.)

Why: References to those variables are in the form Globals.*VariableName* and are therefore more readable and easier to spot.

See also: See rule 6.7 about why you should avoid dependency on global variables. Visual Basic developers should also read rule 18.4 about referencing members of a module.

6.11 Member names from common interfaces

Avoid names used in common .NET interfaces if your property or method doesn't implement that interface.

Example: Examples of such methods are Count, Clone, Dispose, CompareTo, Compare, GetEnumerator, and GetObjectData.

6.12 Case sensitivity in member names [C#]

Never define public members using names that differ only in the casing of the characters. Private members using names that differ only in casing aren't recommended either.

Why: Names that differ only in their case make it harder to read the source code. In addition, if you have two or more public members that differ only in their casing, only one of them is visible to client applications written in Visual Basic or another case-insensitive .NET language.

Example: Examples of members that should be avoided are count and Count, UserName and userName.

6.13 Member ordering and grouping

Group members of the same kind (fields, properties, methods, etc.), and use a #region directive to collapse them easily. Always adopt the same order when defining members.

Example: Always define type members in this order:

1. Event and delegate definitions
2. Private and public fields, except those wrapped by properties (see rule 12.17)
3. Constructors, including static constructors
4. Instance public properties (and the private fields they wrap)
5. Instance public methods

6. Static public methods and properties
7. Methods in interfaces
8. Private (helper) methods

6.14 Language-specific member types

Use the language-specific keywords when defining the type of fields, properties, and methods. For example, use *Integer* (Visual Basic) or *int* (C#) rather than Int32 or System.Int32.

Why: Developers feel it is more natural to use keywords that they know well. Besides, the Microsoft Visual Studio .NET code editor renders language-specific types with a different color, thus increasing code readability.

```
' [Visual Basic]
' *** OK
Dim total As Int32
' *** Better
Dim total As Integer

// [C#]
// *** OK
private Int32 total;
// *** Better
private int total;
```

Alternative rule: Many .NET developers prefer using names specific to .NET (e.g., Int32) in the definition of a field, property, method, or parameter. The rationale behind this guideline is that, everything being an object in the .NET Framework, integer and string values shouldn't be dealt with in any special way and shouldn't be rendered with a different color in the code editor. Arguably, this style makes the source code more readable for developers who work in other languages. Another good point in favor of this style is that it works well with methods whose names contain the name of the .NET type they return, as in this example:

```
' [Visual Basic]
' *** OK, but return type doesn't match suffix in method name.
Function GetDataInt32() As Integer
   ...
End Function

' *** Better
Function GetDataInt32() As Int32
   ...
End Function

// [C#]
// *** OK, but return type doesn't match suffix in method name.
public int GetDataInt32()
{}

// *** Better
public Int32 GetDataInt32()
{}
```

More details: Both guidelines have their merits; therefore, we list both of them. In this book, we have used language-specific types because we believe that most readers are more familiar with this style.

6.15 Nested types

Use a private or internal (*Friend* in Visual Basic) scope qualifier for nested types.

Why: A type should be nested if it is used only by the type that encloses it; therefore, in most cases there is no reason for making the nested type public.

Exception: Nested enumerators and comparers can be given public scope.

6.16 Member scope

Don't make a field, a property, or a method public if that isn't necessary. Mark it with the *Friend* (Visual Basic) or *internal* (C#) keyword if it isn't meant to be invoked from other assemblies; mark it as private if it isn't meant to be invoked from other types in the current assembly.

6.17 Explicit scope qualifier

Always explicitly use a scope keyword for all types and members.

Why: The default scope for Visual Basic type members is *Public*, whereas the default scope for C# is *private*. Omitting the scope keyword might disorient developers who are more familiar with other languages.

```
' [Visual Basic]
' *** OK
Sub PerformTask()
   ...
End Sub

' *** Better
Public Sub PerformTask()
   ...
End Sub

// [C#]
// *** OK
void PerformTask()
{}

// *** Better
private void PerformTask()
{}
```

More details: Visual Basic developers shouldn't use the *Dim* keyword to define type-level fields because this keyword implies a private scope if used inside a class, but it implies a public scope if used inside a structure. Favor using explicit *Private* and *Public* keywords so that you can later change the class to a structure (and vice versa) with minimal impact on the remainder of the application.

```
' [Visual Basic]
' *** Wrong
Structure Person
   Dim FirstName As String      ' These are public fields.
   Dim LastName As String
End Function

' *** Better: explicit Public keyword
Structure Person
   Public FirstName As String
   Public LastName As String
End Function
```

6.18 Shadowed members

Avoid *Shadows* (Visual Basic) or *new* (C#) keywords in favor of *Overridable* and *virtual* keywords, respectively, to redefine methods and properties in derived types.

6.19 Non-CLS-compliant types [C#]

Try to avoid public methods or properties that take or return object types that aren't compliant with Common Language Specifications (CLS), such as unsigned integers. If you can't avoid these members, mark them with the CLSCompliant(false) attribute.

Why: Methods that take or return unsigned integers aren't callable from Visual Basic and some other .NET languages.

```
// [C#]
public class SampleClass
{
   [CLSCompliant(false)]
   public void PerformTask(uint x)
   {}
}
```

More details: The C# compiler checks the CLS compliance and honors the CLSCompliant attribute only for public members in public types and emits a compilation error if a member is incorrectly marked as CLS-compliant. You can't mark a type or a member as CLS-compliant unless the assembly is also marked with a CLSCompliant(true) attribute (see rule 4.5).

6.20 The *Me/this* keyword

Avoid the *Me* (Visual Basic) or *this* (C#) keyword to reference a field or a property unless it helps make the code less ambiguous.

Exception: Using the *Me* or *this* keyword is OK in a method or a constructor that has a parameter or a local variable whose name is the same or is similar to the name of a class-level field or property.

```
' [Visual Basic]
Class Person
   ' These would be properties in a real-world application.
   Public FirstName As String
   Public LastName As String

   Sub New(ByVal firstName As String, ByVal lastName As String)
      Me.FirstName = firstName
      Me.LastName = lastName
   End Sub
End Class

// [C#]
class Person
{
   // These would be properties in a real-world application.
   public string FirstName;
   public string LastName;

   public Person(string firstName, string lastName)
   {
      this.FirstName = firstName;
      this.LastName = lastName;
   }
}
```

6.21 "New" member [C#]

Don't use public member names that match Visual Basic keywords, more specifically the *New* keyword.

Why: Visual Basic developers would need to enclose the name in square brackets to access a method named *New*:

```
' [Visual Basic]
' MyType is a C# type that exposes a New void method.
Dim o As New MyType
' This is the syntax required to access that method from Visual Basic.
o.[New]()
```

6.22 The Conditional attribute

Use the Conditional attribute instead of the *#If* (Visual Basic) or *#if* (C#) compiler directive to exclude a method and all the statements that invoke it.

```
' [Visual Basic]
' *** Wrong
#If DEMOVERSION Then
   ShowNagScreen()
#End If

Sub ShowNagScreen()
   ...
End Sub
```

```
' *** Correct
ShowNagScreen()            ' No need for #If directive

<Conditional("DEMOVERSION")> _
Sub ShowNagScreen()
   ...
End Sub

// [C#]
// *** Wrong
#if DEMOVERSION
   ShowNagScreen();
#endif

void ShowNagScreen()
{
   ...
}

// *** Correct
ShowNagScreen();            // No need for #If directive

[Conditional("DEMOVERSION")]
void ShowNagScreen()
{
   ...
}
```

More details: The Conditional attribute can discard all the statements that invoke the method, but it doesn't discard the method definition itself, so you can still invoke it through reflection. The Visual Basic developer should also keep in mind that the Conditional attribute is ignored when applied to methods that return a value: in other words, *Function* methods are always included, even if they're marked with a Conditional attribute. (The C# compiler correctly flags these cases as compilation errors.) If you need to receive a value back from a method marked with the Conditional attribute, you must use an argument passed with the *ByRef* (Visual Basic) or *ref* (C#) keyword. You shouldn't use an *out* parameter because the compiler would flag the passed variable as unassigned if the Conditional attribute discards the method call that initializes the variable.

6.23 The Serializable attribute

As a rule, apply the Serializable attribute to all nonsealed classes, but mark all nonserializable fields with the NonSerialized attribute.

Why: This technique ensures that instances of this class (and of all types that inherit from it) can be passed as arguments to remote methods.

See also: See rule 12.26 about nonserializable fields and rule 17.23 about events in serializable types.

6.24 The Obsolete attribute

Mark deprecated types and members with the Obsolete attribute so that clients receive a compile warning when they use the deprecated member. The attribute's Message property should describe why the code element is obsolete and what should be used in its place. In later versions of your class library, consider passing True as the second argument of the Obsolete attribute to cause a compile error and therefore force clients to remove any reference to the member.

More details: You should never remove a type (or a member of a type) without marking it obsolete in at least one or two versions of your library.

```
' [Visual Basic]
' Cause a compilation warning.
<Obsolete("Call ShellSort instead")> _
Sub BubbleSort()
   ...
End Sub

' Cause a compilation error.
<Obsolete("Call ShellSort instead", True)> _
Sub BubbleSort()
   ...
End Sub

// [C#]
// Cause a compilation warning.
[Obsolete("Call ShellSort instead")]
void BubbleSort()
{
   ...
}

// Cause a compilation error.
[Obsolete("Call ShellSort instead", true)]
void BubbleSort()
{
   ...
}
```