CODING STANDARDS
Microsoft Coding Standards for .NET

Table of Contents

TABLE OF CONTENTS **2**

SCOPE **4**

NAMING GUIDELINES **4**

Capitalization Styles 4
Pascal case *4*
Camel case *4*
Uppercase *4*
Case Sensitivity 5
Abbreviations 6
Word Choice 6
Avoiding Type Name Confusion 6
Namespace Naming Guidelines 8
Class Naming Guidelines 8
Interface Naming Guidelines 9
Attribute Naming Guidelines 10
Enumeration Type Naming Guidelines 10
Static Field Naming Guidelines 11
Parameter Naming Guidelines 11
Method Naming Guidelines 11
Property Naming Guidelines 12
Event Naming Guidelines 14

GUIDELINES FOR EXPOSING FUNCTIONALITY TO COM **16**

ERROR RAISING AND HANDLING GUIDELINES **17**

ARRAY USAGE GUIDELINES **17**

Arrays vs. Collections 17
Using Indexed Properties in Collections 17
Array Valued Properties 17
Returning Empty Arrays 18

OPERATOR OVERLOADING USAGE GUIDELINES (C# ONLY) **18**

Guidelines for Implementing Equals and the Equality Operator (==) 19
Implementing the Equality Operator (==) on Value Types 19
Implementing the Equality Operator (==) on Reference Types 20

GUIDELINES FOR CASTING TYPES **20**

THREADING DESIGN GUIDELINES **21**

Guidelines for Asynchronous Programming 22

XML DOCUMENTATION **23**

EXAMPLE 23
DOCUMENTATION TAGS FOR OTHER .NET LANGUAGES 25
DESCRIBE THE ASSEMBLY INFORMATION IN THE FILE ASSEMBLYINFO.CS 25

CLASS MEMBER USAGE GUIDELINES 26

PROPERTY USAGE GUIDELINES 26
PROPERTY STATE ISSUES 26
RAISING PROPERTY-CHANGED EVENTS 26
PROPERTIES VS. METHODS 30
READ-ONLY AND WRITE-ONLY PROPERTIES 32
INDEXED PROPERTY USAGE 32
PARAMETER USAGE GUIDELINES 33
FIELD USAGE GUIDELINES 35
CONSTRUCTOR USAGE GUIDELINES 39
METHOD USAGE GUIDELINES 42
METHOD OVERLOADING GUIDELINES 42
METHODS WITH VARIABLE NUMBERS OF ARGUMENTS 46
EVENT USAGE GUIDELINES 47

TYPE USAGE GUIDELINES 51

BASE CLASS USAGE GUIDELINES 51
BASE CLASSES VS. INTERFACES 51
INTERFACES ARE APPROPRIATE IN THE FOLLOWING SITUATIONS: 52
PROTECTED METHODS AND CONSTRUCTORS 52
SEALED CLASS USAGE GUIDELINES 53
DELEGATE USAGE GUIDELINES 54
EVENT NOTIFICATIONS 54
CALLBACK FUNCTIONS 54
VALUE TYPE USAGE GUIDELINES 55
STRUCT USAGE GUIDELINES 55
ENUM USAGE GUIDELINES 57
NESTED TYPE USAGE GUIDELINES 59
ATTRIBUTE USAGE GUIDELINES 60

SETTING ENVIRONMENT OPTIONS (VB.NET) 62

REFERENCE: 62

Scope

This document describes the coding and design guidelines for .NET developers. Unless other wise stated explicitly, all the guidelines are common for all the .NET complaint languages.

Naming Guidelines

Capitalization Styles

Use the following three conventions for capitalizing identifiers.

Pascal case

The first letter in the identifier and the first letter of each subsequent concatenated word are capitalized. You can use Pascal case for identifiers of three or more characters.

For example: **B**ack**C**olor

Camel case

The first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized. For example: **b**ack**C**olor

Uppercase

All letters in the identifier are capitalized. Use this convention only for identifiers that consist of two or fewer letters. For example:

System.IO

System.Web.UI

The following table summarizes the capitalization rules and provides examples for the different types of identifiers.

Identifier	Case	Example
Class	Pascal	AppDomain
Enum type	Pascal	ErrorLevel
Enum values	Pascal	FatalError
Event	Pascal	ValueChangeEventHandler
Exception class	Pascal	WebException (Always ends with the suffix Exception.)
Read-only Static	Pascal	RedValue
Interface	Pascal	IDisposable (Always begins with the prefix I.)
Method	Pascal	ToString
Namespace	Pascal	System.Drawing
Parameter	Camel	typeName
Property	Pascal	BackColor
Protected instance	Camel	redValue (property is preferable to using a protected instance
Public instance field	Pascal	RedValue (property is preferable to using a public instance

Case Sensitivity

To avoid confusion and guarantee cross-language interoperation, follow these rules regarding the use of case sensitivity:

- Do not use names that require case sensitivity. Components must be fully usable from both case-sensitive and case-insensitive languages. Case-insensitive languages cannot distinguish between two names within the same context that differ only by case. Therefore, you must avoid this situation in the components or classes that you create.

- Do not create two namespaces with names that differ only by case. For example, a case insensitive language cannot distinguish between the following two-namespace declarations.

  namespace ee.cummings;

  namespace Ee.Cummings;

- Do not create a function with parameter names that differ only by case. The following example is incorrect.

  void MyFunction (string a, string A)

- Do not create a namespace with type names that differ only by case. In the following example, Point p and POINT p are inappropriate type names because they differ only by case.

  BreakFinder.Forms.Control.Text t

  BreakFinder.Forms.Control.TEXT T

- Do not create a type with property names that differ only by case. In the following example, int Color and int COLOR are inappropriate property names because they differ only by case.

  int Color {get, set}

  int COLOR {get, set}

- Do not create a type with method names that differ only by case. In the following example, calculate and Calculate are inappropriate method names because they differ only by case.

  void calculate ()

  void Calculate()

Abbreviations

Follow these rules regarding the use of abbreviations:

- Do not use abbreviations or contractions as parts of identifier names. For example, use GetWindow instead of GetWin.
- Where appropriate, use well-known acronyms to replace lengthy phrase names. For example, use UI for User Interface and OLAP for On-line Analytical Processing.
- When using acronyms, use Pascal case or camel case for acronyms more than two characters long. For example, use HtmlButton or HTMLButton. However, you should capitalize acronyms that consist of only two characters, such as System.IO instead of System.Io.
- Do not use abbreviations in identifiers or parameter names. If you must use abbreviations, use camel case for abbreviations that consist of more than two characters, even if this contradicts the standard abbreviation of the word.

Word Choice

Avoid using class names that duplicate commonly used .NET Framework namespaces. For example, do not use any of the following names as a class name: **System**, **Collections**, **Forms**, or **UI**. See the Class Library for a list of .NET Framework namespaces.

Avoiding Type Name Confusion

Different programming languages use different terms to identify the fundamental managed types. Class library designers must avoid using language-specific terminology. Follow the rules described in this section to avoid type name confusion.

Use names that describe a type's meaning rather than names that describe the type. In the rare case that a parameter has no semantic meaning beyond its type, use a generic name. For example, a class that supports writing a variety of data types into a stream might have the following methods.

[C#]

```
void Write (double value);
void Write (float value);
void Write (long value);
void Write (int value);
void Write (short value);
```

[VB.NET]

```
Sub Write(value As Double);
Sub Write(value As Single);
Sub Write(value As Long);
Sub Write(value As Integer);
Sub Write(value As Short);
```

Do not create language-specific method names, as in the following example.

[C#]
```
void Write (double doubleValue);
void Write (float floatValue);
void Write (long longValue);
void Write (int intValue);
void Write (short shortValue);
```

[VB.NET]
```
Sub Write(doubleValue As Double);
Sub Write(singleValue As Single);
Sub Write(longValue As Long);
Sub Write(integerValue As Integer);
Sub Write(shortValue As Short);
```

In the extremely rare case that it is necessary to create a uniquely named method for each fundamental data type, use a universal type name.

For example, a class that supports reading a variety of data types from a stream might have the following methods.

[C#]
```
double ReadDouble ();
float ReadSingle ();
long ReadInt64 ();
int ReadInt32 ();
short ReadInt16 ();
```

[VB.NET]
```
ReadDouble()As Double
ReadSingle()As Single
ReadInt64()As Long
ReadInt32()As Integer
ReadInt16()As Short
```

The preceding example is preferable to the following language-specific alternative.

[C#]
```
double ReadDouble ();
float ReadFloat ();
long ReadLong ();
int ReadInt ();
short ReadShort ();
```

[VB.NET]
```
ReadDouble()As Double
ReadSingle()As Single
ReadLong()As Long
ReadInteger()As Integer
ReadShort()As Short
```

Namespace Naming Guidelines

The general rule for naming namespaces is to use the company name followed by the technology name and optionally the feature and design as follows.

FOCAL3h.ModuleName [.Feature][.Design]

For example:

FOCAL3h.SAMPLE

FOCAL3h.SAMPLE.BreakFinder

Prefixing namespace names with a company name or other well-established brand avoids the possibility of two published namespaces having the same name. For example, Microsoft.Office is an appropriate prefix for the Office Automation Classes provided by Microsoft.

Use a stable, recognized technology name at the second level of a hierarchical name. Use organizational hierarchies as the basis for namespace hierarchies. Name a namespace that contains types that provide design-time functionality for a base namespace with the. Design suffix. For example, the System.Windows.Forms.Design Namespace contains designers and related classes used to design System.Windows.Forms based applications.

A nested namespace should have a dependency on types in the containing namespace. For example, the classes in the SAMPLE.Web.UI.Design depend on the classes in System.Web.UI. However, the classes in **SAMPLE.Web.UI** do not depend on the classes in **System.UI.Design**.

Use Pascal case for namespaces, and separate logical components with periods, as in Microsoft.Office.PowerPoint. If your brand employs nontraditional casing, follow the casing defined by your brand, even if it deviates from the prescribed Pascal case. For example, the namespaces NeXT.WebObjects and ee.cummings illustrate appropriate deviations from the Pascal case rule.

Use plural namespace names if it is semantically appropriate. For example, use System.Collections rather than System.Collection. Exceptions to this rule are brand names and abbreviations. For example, use System.IO rather than System.IOs.

Do not use the same name for a namespace and a class. For example, do not provide both a Debug namespace and a Debug class.

Class Naming Guidelines

The following rules outline the guidelines for naming classes:

- Use a noun or noun phrase to name a class.
- Use Pascal Case

- Use abbreviations sparingly.
- Do not use a type prefix, such as C for class, on a class name. For example, use the class name FileStream rather than CFileStream.
- Do not use the underscore character (_).
- Occasionally, it is necessary to provide a class name that begins with the letter I, even though the class is not an interface. This is appropriate as long as I is the first letter of an entire word that is a part of the class name. For example, the class name IdentityStore is appropriate.
- Where appropriate, use a compound word to name a derived class. The second part of the derived class's name should be the name of the base class. For example, ApplicationException is an appropriate name for a class derived from a class named Exception, because ApplicationException is a kind of Exception.

The following are examples of correctly named classes.

public class **FileStream**

public class **Button**

public class **String**

Interface Naming Guidelines

The following rules outline the naming guidelines for interfaces:

- Name interfaces with nouns or noun phrases, or adjectives that describe behavior. For example, the interface name **IComponent** uses a descriptive noun. The interface name **ICustomAttributeProvider** uses a noun phrase. The name **IPersistable** uses an adjective.
- Use Pascal Case
- Use abbreviations sparingly.
- Prefix interface names with the letter I, to indicate that the type is an interface.
- Use similar names when you define a class/interface pair where the class is a standard implementation of the interface. The names should differ only by the letter I prefix on the interface name.
- Do not use the underscore character (_).

The following are examples of correctly named interfaces.

[C#]

```
public interface IServiceProvider
public interface Iformatable
```

[VB.NET]

```
Public Interface IServiceProvider
Public Interface IFormatable
```

The following code example illustrates how to define the interface **IComponent** and its standard implementation, the class Component.

[C#]

```
public interface IComponent
{
  // Implementation code goes here.
}
public class Component: IComponent
{
  // Implementation code goes here.
}
```

[VB.NET]

```
Public Interface IComponent
  ' Implementation code goes here.
End Interface

Public Class Component
  Implements IComponent
  ' Implementation code goes here.
End Class
```

Attribute Naming Guidelines

You should always add the suffix Attribute to custom attribute classes. The following is an example of a correctly named attribute class.

public class **CMTSAttribute**

Enumeration Type Naming Guidelines

The enumeration (Enum) value type inherits from the Enum Class. The following rules outline the naming guidelines for enumerations:

- Use Pascal Case for **Enum** types and value names.
- Use abbreviations sparingly.
- Do not use an Enum suffix on **Enum** type names.
- Use a singular name for most **Enum** types, but use a plural name for **Enum** types that are bit fields.

- Always add the **FlagsAttribute** to a bit field **Enum** type.

Static Field Naming Guidelines

The following rules outline the naming guidelines for static fields:

- Use nouns, noun phrases, or abbreviations of nouns to name static fields.
- Use Pascal Case
- Use a Hungarian notation prefix on static field names.
- It is recommended that you use static properties instead of public static fields whenever possible.

Parameter Naming Guidelines

The following rules outline the naming guidelines for parameters:

- Use descriptive parameter names. Parameter names should be descriptive enough that the name of the parameter and its type can be used to determine its meaning in most scenarios.
- Use camel case for parameter names.
- Use names that describe a parameter's meaning rather than names that describe a parameter's type. Development tools should provide meaningful information about a parameter's type. Therefore, a parameter's name can be put to better use by describing meaning. Use type-based parameter names sparingly and only where it is appropriate.
- Do not use reserved parameters. Reserved parameters are private parameters that might be exposed in a future version if they are needed. Instead, if more data is needed in a future version of your class library, add a new overload for a method.
- Do not prefix parameter names with Hungarian type notation.

The following are examples of correctly named parameters.

[C#]
Type GetType (string **typeName**)
string Format (string **format**, **args ()** As object)

[VB.NET]
GetType(**typeName** As String)As Type
Format(**format** As String, object [] **args**)As String

Method Naming Guidelines

The following rules outline the naming guidelines for methods:

- Use verbs or verb phrases to name methods.
- Use Pascal Case

The following are examples of correctly named methods.

RemoveAll ()

GetCharArray ()

Invoke ()

Property Naming Guidelines

The following rules outline the naming guidelines for properties:

- Use a noun or noun phrase to name properties.
- Use Pascal Case.
- Do not use Hungarian notation.
- Consider creating a property with the same name as its underlying type. For example, if you declare a property named Color, the type of the property should likewise be Color. See the example later in this topic.

The following code example illustrates correct property naming.

[C#]

```
public class SampleClass
{
  public Color BackColor
  {
    // Code for Get and Set accessors goes here.
  }
}
```

[VB.NET]

```
Public Class SampleClass
  Public Property BackColor As Color
    ' Code for Get and Set accessors goes here.
  End Property
End Class
```

The following code example illustrates providing a property with the same name as a type.

[C#]

```
public enum Color
{
  // Insert code for Enum here.
}

public class Control
{
  public Color Color
  {
    get {// Insert code here.}
    set {// Insert code here.}
  }
}
```

[VB.NET]

```
Public Enum Color
  ' Insert code for Enum here.
End Enum
Public Class Control
  Public Property Color As Color
    Get
      ' Insert code here.
    End Get
    Set
      ' Insert code here.
    End Set
  End Property
End Class
```

The following code example is incorrect because the property Color is of type Integer.

[C#]

```
public enum Color {// Insert code for Enum here.}
public class Control
{
  public int Color
  {
    get {// Insert code here.}
    set {// Insert code here.}
```

```
  }
}
```

[VB.NET]

```
Public Enum Color
  ' Insert code for Enum here.
End Enum
Public Class Control
  Public Property Color As Integer
    Get
      ' Insert code here.
    End Get
    Set
      ' Insert code here.
    End Set
  End Property
End Class
```

In the incorrect example, it is not possible to refer to the members of the Color enumeration. Color.Xxx will be interpreted as accessing a member that first gets the value of the Color property (type Integer in Visual Basic or type int in C#) and then accesses a member of that value (which would have to be an instance member of System.Int32).

Event Naming Guidelines

The following rules outline the naming guidelines for events:

- Use an EventHandler suffix on event handler names.
- Specify two parameters named sender and e. The sender parameter represents the object that raised the event. The sender parameter is always of type **object**, even if it is possible to use a more specific type. The state associated with the event is encapsulated in an instance of an event class named e. Use an appropriate and specific event class for the e parameter type.
- Name an event argument class with the EventArgs suffix.
- Consider naming events with a verb.
- Use a gerund (the "ing" form of a verb) to create an event name that expresses the concept of pre-event, and a past-tense verb to represent post-event. For example, a **Close** event that can be canceled should have a Closing event and a Closed event. Do not use the BeforeXxx/AfterXxx naming pattern.
- Do not use a prefix or suffix on the event declaration on the type. For example, use Close instead of OnClose.
- In general, you should provide a protected method called OnXxx on types with events that can be overridden in a derived class. This method should only have the event parameter e, because the sender is always the instance of the type.

The following example illustrates an event handler with an appropriate name and parameters.

[C#]

```
public delegate void MouseEventHandler (object sender, MouseEventArgs e);
```

[VB.NET]

```
Public Delegate Sub MouseEventHandler(sender As Object, e As MouseEventArgs)
```

The following example illustrates a correctly named event argument class.

[C#]

```
public class MouseEventArgs : EventArgs
{
  int x;
  int y;
  public MouseEventArgs(int x, int y) {this.x = x; this.y = y;}
  public int X {get {return x; } }
  public int Y {get {return y; } }
}
```

[VB.NET]

```
Public Class MouseEventArgs
  Inherits EventArgs
  Dim x As Integer
  Dim y As Integer

  Public Sub New MouseEventArgs(x As Integer, y As Integer)
    me.x = x
    me.y = y
  End Sub

  Public Property X As Integer
    Get
      Return x
    End Get
  End Property

  Public Property Y As Integer
    Get
      Return y
    End Get
  End Property
End Class
```

Guidelines for Exposing Functionality to COM

The common language runtime provides rich support for interoperating with COM components. A COM component can be used from within a managed type and a managed instance can be used by a COM component. This support is the key to moving unmanaged code to managed code one piece at a time; however, it does present some issues for class library designers. In order to fully expose a managed type to COM clients, the type must expose functionality in a way that is supported by COM and abides by the COM versioning contract.

Mark managed class libraries with the ComVisibleAttribute attribute to indicate whether COM clients can use the library directly or whether they must use a wrapper that shapes the functionality so that they can use it.

Types and interfaces that must be used directly by COM clients, such as to host in an unmanaged container, should be marked with the **ComVisible**(true) attribute. The transitive closure of all types referenced by exposed types should be explicitly marked as **ComVisible**(true); if not, they will be exposed as **IUnknown**.

Note Members of a type can also be marked as **ComVisible** (false); this reduces exposure to COM and therefore reduces the restrictions on what a managed type can use.

Types marked with the **ComVisible** (true) attribute cannot expose functionality exclusively in a way that is not usable from COM. Specifically; COM does not support static methods or parameterized constructors. Test the type's functionality from COM clients to ensure correct behavior. Make sure that you understand the registry impact for making all types cocreateable.

Follow these guidelines when using marshal by reference:

- By default, instances should be marshal-by-value objects. This means that their types should be marked as **Serializable**.

- Component types should be marshal-by-reference objects. This should already be the case for most components, because the common base class, System.Component Class, is a marshal-by-reference class.

- If the type encapsulates an operating system resource, it should be a marshal-by-reference object. If the type implements the IDisposable Interface it will very likely have to be marshaled by reference. System.IO.Stream derives from MarshalByRefObject. Most streams, such as FileStreams and NetworkStreams, encapsulate external resources, so they should be marshal-by-reference objects.

- Instances that simply hold state should be marshal-by-value objects (such as a **DataSet**).

- Special types that cannot be called across an AppDomain (such as a holder of static utility methods) should not be marked as **Serializable**.

Error Raising and Handling Guidelines

Please refer SAMPLE Error Handling.doc in source safe

($\SAMPLE\Documents\General\)

Array Usage Guidelines

An array type is defined by specifying the element type of the array, the rank (number of dimensions) of the array, and the upper and lower bounds of each dimension of the array. All these are included in any signature of an array type, although they might be marked as dynamically (rather than statically) supplied. Exact array types are created automatically by the runtime as they are required, and no separate definition of the array type is needed. Arrays of a given type can only hold elements of that type

Arrays vs. Collections

Class library designers might need to make difficult decisions about when to use an array and when to return a collection. Although these types have similar usage models, they have different performance characteristics. You should use a collection in the following situations:

- When Add, Remove, or other methods for manipulating the collection are supported.

- To add read-only wrappers around internal arrays.

Using Indexed Properties in Collections

Use an indexed property only as a default member of a collection class or interface. Do not create families of functions in noncollection types. A pattern of methods, such as **Add**, **Item**, and **Count**, signal that the type should be a collection.

Array Valued Properties

Use collections to avoid code inefficiencies. In the following code example, each call to the myObj property creates a copy of the array. As a result, 2n+1 copies of the array will be created in the following loop.

[C#]

```
for (int i = 0; i < obj.myObj.Count; i++)
     DoSomething(obj.myObj[i]);
```

[VB.NET]

```
Dim i As Integer
For i = 0 To obj.myObj.Count - 1
   DoSomething(obj.myObj(i))
```

```
Next i
```

Returning Empty Arrays

String and Array properties should never return a null reference. Null can be difficult to understand in this context. For example, a user might assume that the following code will work.

[C#]

```
public void DoSomething()
{
  string s = SomeOtherFunc();
  if (s.Length > 0)
  {
    // Do something else.
  }
}
```

[VB.NET]

```
Public Sub DoSomething()
  Dim s As String = SomeOtherFunc()
  If s.Length > 0 Then
    ' Do something else.
  End If
End Sub
```

The general rule is that null, empty string (""), and empty (0 item) arrays should be treated the same way. Return an empty array instead of a null reference.

Operator Overloading Usage Guidelines (C# Only)

The following rules outline the guidelines for operator overloading:

- Define operators on value types that are logical built-in language types, such as the System.Decimal Structure.
- Provide operator-overloading methods only in the class in which the methods are defined.
- Use the names and signature conventions described in the Common Language Specification (CLS).
- Use operator overloading in cases where it is immediately obvious what the result of the operation will be. For example, it makes sense to be able to subtract one Time value from another Time value and get a TimeSpan. However, it is not appropriate to use the **or** operator to create the union of two database queries, or to use **shift** to write to a stream.
- Overload operators in a symmetric manner. For example, if you overload the equality operator (==), you should also overload the not equal operator(!=).

- Provide alternate signatures. Most languages do not support operator overloading. For this reason, always include a secondary method with an appropriate domain-specific name that has the equivalent functionality. It is a Common Language Specification (CLS) requirement to provide this secondary method. The following example is CLS-compliant.

  [C#]

```
class Time
{
  TimeSpan operator -(Time t1, Time t2) { }
  TimeSpan Difference(Time t1, Time t2) { }
}
```

Guidelines for Implementing Equals and the Equality Operator (==)

The following rules outline the guidelines for implementing the Equals method and the equality operator (==):

- Implement the **GetHashCode** method whenever you implement the Equals method. This keeps **Equals** and **GetHashCode** synchronized.

- Override the **Equals** method whenever you implement ==, and make them do the same thing. This allows infrastructure code such as Hashtable and ArrayList, which use the Equals method, to behave the same way as user code written using ==.

- Override the Equals method any time you implement the **IComparable** Interface.

- You should consider implementing operator overloading for the equality (==), not equal (!=), less than (<), and greater than (>) operators when you implement **IComparable**.

- Do not throw exceptions from the Equals or GetHashCode methods or the equality operator (==).

Implementing the Equality Operator (==) on Value Types

In most programming languages there is no default implementation of the equality operator (==) for value types. Therefore, you should overload == any time equality is meaningful.

You should consider implementing the **Equals** method on value types because the default implementation on System.ValueType will not perform as well as your custom implementation.

Implement == any time you override the **Equals** method.

Implementing the Equality Operator (==) on Reference Types

Most languages do provide a default implementation of the equality operator (==) for reference types. Therefore, you should use care when implementing == on reference types. Most reference types, even those that implement the Equals method, should not override ==.

Override == if your type is a base type such as a Point, String, BigNumber, and so on. Any time you consider overloading the addition (+) and subtraction (-) operators, you also should consider overloading ==.

Guidelines for Casting Types

The following rules outline the usage guidelines for casts:

- Do not allow implicit casts that will result in a loss of precision. For example, there should not be an implicit cast from **Double** to **Int32**, but there might be one from **Int32** to **Int64**.

- Do not throw exceptions from implicit casts because it is very difficult for the developer to understand what is happening.

- Provide casts that operate on an entire object. The value that is cast should represent the entire object, not a member of an object. For example, it is not appropriate for a Button to cast to a string by returning its caption.

- Do not generate a semantically different value. For example, it is appropriate to convert a Time or TimeSpan into an **Int32**. The **Int32** still represents the time or duration. It does not, however, make sense to convert a file name string such as "c:\mybitmap.gif" into a Bitmap object.

- Do not cast values from different domains. Casts operate within a particular domain of values. For example, numbers and strings are different domains. It makes sense that an **Int32** can cast to a **Double**. However, it does not make sense for an **Int32** to cast to a String, because they are in different domains.

Threading Design Guidelines

The following rules outline the design guidelines for implementing threading:

- Avoid providing static methods that alter static state. In common server scenarios, static state is shared across requests, which means multiple threads can execute that code at the same time. This opens up the possibility for threading bugs. Consider using a design pattern that encapsulates data into instances that are not shared across requests.

- Static state must be thread safe.

- Instance state does not need to be thread safe. By default, a library is not thread safe. Adding locks to create thread-safe code decreases performance, increases lock contention, and creates the possibility for deadlock bugs to occur. In common application models, only one thread at a time executes user code, which minimizes the need for thread safety. For this reason, the .NET Framework is not thread safe by default. In cases where you want to provide a thread-safe version, use a GetSynchronized method to return a thread-safe instance of a type. For examples, see the System.Collections Namespace.

- Design your library with consideration for the stress of running in a server scenario. Avoid taking locks whenever possible.

- Be aware of method calls in locked sections. Deadlocks can result when a static method in class A calls static methods in class B and vice versa. If A and B both synchronize their static methods, this will cause a deadlock. You might discover this deadlock only under heavy threading stress.

- Performance issues can result when a static method in class A calls a static method in class A. If these methods are not factored correctly, performance will suffer because there will be a large amount of redundant synchronization. Excessive use of fine-grained synchronization might negatively impact performance. In addition, it might have a significant negative impact on scalability.

- Be aware of issues with the lock statement (**SyncLock** in Visual Basic). It is tempting to use the lock statement to solve all threading problems. However, the System.Threading.Interlocked Class is superior for updates that must be made automatically. It executes a single **lock** prefix if there is no contention. In a code review, you should watch out for instances like the one shown in the following example.

  [C#]

```
lock(this)
{
  myField++;
}
```

  [VB.NET]

```
SyncLock Me
  myField += 1
```

```
End SyncLock
```

Alternatively, it might be better to use more elaborate code to create rhs outside of the lock, as in the following example. Then, you can use an interlocked compare exchange to update x only if it is still null. This assumes that creation of duplicate rhs values does not cause negative side effects.

[C#]

```
if (x == null)
{
  lock (this)
  {
    if (x == null)
    {
      // Perform some elaborate code to create rhs.
      x = rhs;
    }
  }
}
```

[VB.NET]

```
If x Is Nothing Then
  SyncLock Me
    If x Is Nothing Then
      ' Perform some elaborate code to create rhs.
      x = rhs
    End If
  End SyncLock
End If
```

- Avoid the need for synchronization if possible. For high traffic pathways, it is best to avoid synchronization. Sometimes the algorithm can be adjusted to tolerate race conditions rather than eliminate them.

Guidelines for Asynchronous Programming

Asynchronous programming is a feature supported by many areas of the common language runtime, such as Remoting, ASP.NET, and Windows Forms. Asynchronous programming is a core concept in the .NET Framework. This topic introduces the design pattern for asynchronous programming.

The philosophy behind these guidelines is as follows:

- The client should decide whether a particular call should be asynchronous.
- It is not necessary for a server to do additional programming in order to support its clients' asynchronous behavior. The runtime should be able to manage the difference between the client and server views. As a result, the situation where the server has to implement **IDispatch** and do a large amount of work to support dynamic invocation by clients is avoided.

- The server can choose to explicitly support asynchronous behavior either because it can implement asynchronous behavior more efficiently than a general architecture, or because it wants to support only asynchronous behavior by its clients. It is recommended that such servers follow the design pattern outlined in this document for exposing asynchronous operations.

- Type safety must be enforced.

- The runtime provides the necessary services to support the asynchronous programming model. These services include the following:

    - Synchronization primitives, such as critical sections and ReaderWriterLock instances.

    - Synchronization constructs such as containers that support the **WaitForMultipleObjects** method.

    - Thread pools.

    - Exposure to the underlying infrastructure, such as Message and ThreadPool objects.

XML Documentation

C# provides a mechanism for developers to document their code using XML. In source code files, lines that begin with /// and that precede a user-defined type such as a class, delegate, or interface; a member such as a field, event, property, or method; or a namespace declaration can be processed as comments and placed in a file. C# is the only language that supports the XML documentation.

Example

The following sample provides a basic overview of a type that has been documented. To compile the example, type the following command line:

[C#]

```
using System;

/// <summary>
/// Class level summary documentation goes here.</summary>
/// <remarks>
/// Longer comments can be associated with a type or member
/// through the remarks tag</remarks>
public class SomeClass
{
  /// <summary>
  /// Store for the name property</summary>
  private string myName = null;

  /// <summary>
  /// The class constructor. </summary>
```

```
  public SomeClass()
  {
     // TODO: Add Constructor Logic here
  }

  /// <summary>
  /// Name property </summary>
  /// <value>
  /// A value tag is used to describe the property value</value>
  public string Name
  {
    get
    {
      if ( myName == null )
      {
        throw new Exception("Name is null");
      }

      return myName;
    }
  }
  /// <summary>
  /// Description for SomeMethod.</summary>
  /// <param name="s"> Parameter description for s goes here</param>
  /// <seealso cref="String">
  /// You can use the cref attribute on any tag to reference a type or member
  /// and the compiler will check that the reference exists. </seealso>
  public void SomeMethod(string s)
  {
  }

  /// <summary>
  /// Some other method. </summary>
  /// <returns>
  /// Return results are described through the returns tag.</returns>
  /// <seealso cref="SomeMethod(string)">
  /// Notice the use of the cref attribute to reference a specific method </seealso>
  public int SomeOtherMethod()
  {
    return 0;
  }

  /// <summary>
  /// The entry point for the application.
  /// </summary>
  /// <param name="args"> A list of command line arguments</param>
  public static int Main(String[] args)
  {
    // TODO: Add code to start application here

     return 0;
  }
}
```

XML documentation starts with ///. When you create a new project, the wizards put some starter /// lines in for you. The processing of these comments has some restrictions:

The documentation must be well-formed XML. If the XML is not well-formed, a warning is generated and the documentation file will contain a comment saying that an error was encountered. For more information on well-formed XML, see XML Glossary.

Developers are free to create their own set of tags. There is a recommended set of tags. Some of the recommended tags have special meanings:

The <param> tag is used to describe parameters. If used, the compiler will verify that the parameter exists and that all parameters are described in the documentation. If the verification failed, the compiler issues a warning.

The cref attribute can be attached to any tag to provide a reference to a code element. The compiler will verify that this code element exists. If the verification failed, the compiler issues a warning. The compiler also respects any using statements when looking for a type described in the **cref** attribute.

The <summary> tag is used by IntelliSense inside Visual Studio to display additional information about a type or member.

Documentation Tags for other .NET Languages

[VB.NET]

```
‘ <doc>
‘ Summary: summary documentation goes here.
‘ Remarks: Longer comments can be written here.
‘ Parameter: ParamName-1: Description of ParamName-1
‘ Parameter: ParamName-2: Description of ParamName-2
‘ Returns: Description of the return value
‘ </doc>
```

Describe the assembly information in the file AssemblyInfo.cs

General Information about an assembly is controlled through the following set of attributes. Change these attribute values to modify the information associated with an assembly.

```
[assembly: AssemblyTitle("DataManager")]
[assembly: AssemblyDescription("Data manager component for BreakFinder application")]
[assembly: AssemblyConfiguration("Debug")]
[assembly: AssemblyCompany(" NeST Technologies")]
[assembly: AssemblyProduct("BreakFinder")]
[assembly: AssemblyCopyright("NeST Technologies 2002, All rights reserved.")]
[assembly: AssemblyTrademark("BreakFinder")]
[assembly: AssemblyCulture("")]
```

Add additional information's like Last Author and Last Modified Date in the same file as comments

Class Member Usage Guidelines

This topic provides guidelines for using class members in class libraries.

Property Usage Guidelines

Determine whether a property or a method is more appropriate for your needs. Choose a name for your property based on the recommended **Property Naming Guidelines**. Avoid creating a property with the same name as an existing type. Defining a property with the same name as a type causes ambiguity in some programming languages. For example, **System.Windows.Forms.Control** has a color property. Since a Color Structure also exists, the **System.Windows.Forms.Control** color property is named **BackColor**. It is a more meaningful name for the property and it does not conflict with the Color Structure name.

There might be situations where you have to violate this rule. For example, the System.Windows.Forms.Form Class contains an **Icon** property even though an **Icon** class also exists in the .NET Framework. This is because **Form.Icon** is a more straightforward and understandable name for the property than **Form.FormIcon** or **Form.DisplayIcon**.

When accessing a property using the **set** accessor, preserve the value of the property before you change it. This will ensure that data is not lost if the set accessor throws an exception.

Property State Issues

Allow properties to be set in any order. Properties should be stateless with respect to other properties. It is often the case that a particular feature of an object will not take effect until the developer specifies a particular set of properties, or until an object has a particular state. Until the object is in the correct state, the feature is not active. When the object is in the correct state, the feature automatically activates itself without requiring an explicit call. The semantics are the same regardless of the order in which the developer sets the property values or how the developer gets the object into the active state.

Raising Property-Changed Events

Components should raise property-changed events if they want to notify consumers when the component's property changes programmatically. The naming convention for a property-changed event is to add the Changed suffix to the property name, such as TextChanged. For example, a control might raise a TextChanged event when its text property changes. You can use a protected helper routine

Raise<Property>Changed, to raise this event. However, it is probably not worth the overhead to raise a property-changed event for a hash table item addition. The

following code example illustrates the implementation of a helper routine on a property-changed event.

[C#]

```
class Control: Component
{
  string text;
  public string Text
  {
    get
    {
      return text;
    }
    set
    {
      if (!text.Equals(value))
      {
        text = value;
        RaiseTextChangedEvent();
      }
    }
  }
}
```

[VB.NET]

```
Class Control
  Inherits Component
  Private text As String
  Public Property Text() As String
    Get
      Return text
    End Get
    Set
      If Not text.Equals(value) Then
        text = value
        RaiseTextChangedEvent()
      End If
    End Set
  End Property
End Class
```

Data binding uses this pattern to allow two-way binding of the property. Without **<Property>Changed** and **Raise<Property>Changed** events, data binding works in one direction; if the database changes, the property is updated. Each property that raises the **<Property>Changed** event should provide metadata to indicate that the property supports data binding.

It is recommended that you raise changing/changed events if the value of a property changes as a result of external forces. These events indicate to the developer that the value of a property is changing or has changed as a result of an operation, rather than by calling methods on the object.

A good example is the **Text** property of an **Edit** control. As a user types information into the control, the property value automatically changes. An event is raised before the value of the property has changed. It does not pass the old or new value, and the developer can cancel the event by throwing an exception. The name of the event is the name of the property followed by the suffix **Changing**. The following code example illustrates a changing event.

[C#]

```
class Edit : Control
{
  public string Text
  {
    get
    {
      return text;
    }
    set
    {
      if (text != value)
      {
        OnTextChanging(Event.Empty);
        text = value;
      }
    }
  }
}
```

[VB.NET]

```
Class Edit
  Inherits Control
  Public Property Text() As String
    Get
      Return text
    End Get
    Set
      If text <> value Then
        OnTextChanging(Event.Empty)
        text = value
      End If
    End Set
  End Property
End Class
```

An event is also raised after the value of the property has changed. This event cannot be canceled. The name of the event is the name of the property followed by the suffix Changed.

The generic **PropertyChanged** event should also be raised. The pattern for raising both of these events is to raise the specific event from the **OnPropertyChanged** method. The following example illustrates the use of the **OnPropertyChanged** method.

[C#]

```
class Edit : Control
{
  public string Text
  {
    get
    {
      return text;
    }
    set
    {
      if (text != value)
      {
        OnTextChanging(Event.Empty);
        text = value;
        RaisePropertyChangedEvent(Edit.ClassInfo.text);
      }
    }
  }

  protected void OnPropertyChanged(PropertyChangedEvent e)
  {
    if (e.PropertyChanged.Equals(Edit.ClassInfo.text))
      OnTextChanged(Event.Empty);
    if (onPropertyChangedHandler != null)
      onPropertyChangedHandler(this, e);
  }
}
```

[VB.NET]

```
Class Edit
  Inherits Control
  Public Property Text() As String
    Get
      Return text
    End Get
    Set
      If text <> value Then
        OnTextChanging(Event.Empty)
        text = value
        RaisePropertyChangedEvent(Edit.ClassInfo.text)
      End If
    End Set
  End Property
  Protected Sub OnPropertyChanged(e As PropertyChangedEvent)
    If e.PropertyChanged.Equals(Edit.ClassInfo.text) Then
      OnTextChanged(Event.Empty)
    End If
    If Not (onPropertyChangedHandler Is Nothing) Then
      onPropertyChangedHandler(Me, e)
    End If
  End Sub
End Class
```

There are cases when the underlying value of a property is not stored as a field, making it difficult to track changes to the value. When raising the changing event, find all the places that the property value can change and provide the ability to cancel the event. For example, the previous **Edit** control example is not entirely accurate because the **Text** value is actually stored in the window handle (**HWND**). In order to raise the **TextChanging** event, you must examine Windows messages to determine when the text might change, and allow for an exception thrown in **OnTextChanging** to cancel the event. If it is too difficult to provide a changing event, it is reasonable to support only the changed event.

Properties vs. Methods

Class library designers often must decide between implementing a class member as a property or a method. Use the following guidelines to help you choose between these options.

- Use a property when the member is a logical data member. In the following member declarations, Name is a property because it is a logical member of the class.

[C#]

```
public string Name
get
{
    return Name;
}
set
{
    Name = value;
}
```

[VB.NET]

```
Public Property Name As String
  Get
    Return Name
  End Get
  Set
    Name = value
  End Set
End Property
```

Use a method when:

- The operation is a conversion, such as **Object**.**ToString**.
- The operation is expensive enough that you want to communicate to the user that they should consider caching the result.
- Obtaining a property value using the get accessor would have an observable side effect.

- Calling the member twice in succession produces different results.
- The order of execution is important. Note that a type's properties should be able to be set and retrieved in any order.
- The member is static but returns a value that can be changed.
- The member returns an array. Properties that return arrays can be very misleading. Usually it is necessary to return a copy of the internal array so that the user cannot change internal state. This, coupled with the fact that a user can easily assume it is an indexed property, leads to inefficient code. In the following code example, each call to the Methods property creates a copy of the array. As a result, 2n+1 copies of the array will be created in the following loop.

[C#]

```
Type type = // Get a type.
for (int i = 0; i < type.Methods.Length; i++)
{
  if (type.Methods[i].Name.Equals ("text"))
  {
    // Perform some operation.
  }
}
```

[VB.NET]

```
Dim type As Type = ' Get a type.
Dim i As Integer
For i = 0 To type.Methods.Length - 1
  If type.Methods(i).Name.Equals("text") Then
    ' Perform some operation.
  End If
Next i
```

The following example illustrates the correct use of properties and methods.

[C#]

```
class Connection
{
  // The following three members should be properties
  // because they can be set in any order.
  string DNSName {get{};set{};}
  string UserName {get{};set{};}
  string Password {get{};set{};}

  // The following member should be a method
  // because the order of execution is important.
  // This method cannot be executed until after the
  // properties have been set.
  bool Execute ();
}
```

[VB.NET]

```
Class Connection
```

```
' The following three members should be properties
' because they can be set in any order.
Property DNSName() As String
   ' Code for get and set accessors goes here.
End Property
Property UserName() As String
   ' Code for get and set accessors goes here.
End Property
Property Password() As String
   'Code for get and set accessors goes here.
End Property
' The following member should be a method
' because the order of execution is important.
' This method cannot be executed until after the
' properties have been set.
Function Execute() As Boolean
```

Read-Only and Write-Only Properties

Use a read-only property when the user cannot change the property's logical data member. Do not use write-only properties.

Indexed Property Usage

The following rules outline guidelines for using indexed properties:

- Use only one indexed property per class, and make it the default indexed property for that class.
- Do not use nondefault indexed properties.
- Name an indexed property Item. For example, see the DataGrid.Item Property. Follow this rule, unless there is a name that is more obvious to users, such as the Chars property on the **String** class.
- Use an indexed property when the property's logical data member is an array.
- Do not provide an indexed property and a method that are semantically equivalent to two or more overloaded methods. In the following code example, the Method property should be changed to GetMethod(string) method.

  [C#]

```
// Change the MethodInfo Type.Method property to a method.
MethodInfo Type.Method[string name]
MethodInfo Type.GetMethod (string name, Boolean ignoreCase)
// The MethodInfo Type.Method property is changed to
// the MethodInfo Type.GetMethod method.
MethodInfo Type.GetMethod(string name)
MethodInfo Type.GetMethod (string name, Boolean ignoreCase)
```

[VB.NET]

```
' Change the MethodInfo Type.Method property to a method.
Property Type.Method(name As String) As MethodInfo
Function Type.GetMethod(name As String, ignoreCase As Boolean) As
MethodInfo
' The MethodInfo Type.Method property is changed to
' the MethodInfo Type.GetMethod method.
Function Type.GetMethod(name As String) As MethodInfo
Function Type.GetMethod(name As String, ignoreCase As Boolean) As
MethodInfo
```

Parameter Usage Guidelines

The following rules outline the usage guidelines for parameters:

Check for valid parameter arguments. Perform argument validation for every public or protected method and property **set** accessor. Throw meaningful exceptions to the developer for invalid parameter arguments. Use the System.ArgumentException Class, or a class derived from **System**.**ArgumentException**. The following example checks for valid parameter arguments and throws meaningful exceptions.

[C#]

```
class SampleClass
{
  public int Count
  {
    get
    {
      return count;
    }
    set
    {
        // Check for valid parameter.
        if (count < 0 || count >= MaxValue)
          throw newArgumentOutOfRangeException(
            Sys.GetString(
              "InvalidArgument","value",count.ToString()));
    }
  }

  public void Select(int start, int end)
  {
    // Check for valid parameter.
    if (start < 0)
      throw new ArgumentException(
          Sys.GetString("InvalidArgument","start",start.ToString()));
    // Check for valid parameter.
    if (end < 0)
      throw new ArgumentException(
          Sys.GetString("InvalidArgument","end",end.ToString()));
  }
```

```
}
```

[VB.NET]

```
Class SampleClass
  Private countValue As Integer
  Private maxValue As Integer = 100
  Public Property Count() As Integer
    Get
      Return countValue
    End Get
    Set
      ' Check for valid parameter.
      If value < 0 Or value >= maxValue Then
        Throw New ArgumentOutOfRangeException("value", value,
          "Value is invalid.")
      End If
      countValue = value
    End Set
  End Property
  Public Sub SelectItem(start As Integer, [end] As Integer)
    ' Check for valid parameter.
    If start < 0 Then
      Throw New ArgumentOutOfRangeException("start", start, "Start
        is invalid.")
    End If
    ' Check for valid parameter.
    If [end] < 0 Then
      Throw New ArgumentOutOfRangeException("end", [end], "End is
          invalid.")
    End If
    ' Insert code to do other work here.
    Console.WriteLine("Starting at {0}", start)
    Console.WriteLine("Ending at {0}", [end])
  End Sub
End Class
```

Note that the actual checking does not necessarily have to happen in the public or protected method itself. It could happen at a lower level in private routines. The main point is that the entire surface area that is exposed to the developer checks for valid arguments.

Field Usage Guidelines

The following rules outline the usage guidelines for fields:

- Do not use instance fields that are **public** or **protected** (**Public** or **Protected** in Visual Basic). If you avoid exposing fields directly to the developer, classes can be versioned more easily because a field cannot be changed to a property while maintaining binary compatibility. Consider providing **get** and **set** property accessors for fields instead of making them public. The presence of executable code in **get** and **set** property accessors allows later improvements, such as creation of an object on demand, upon usage of the property, or upon a property change notification. The following code example illustrates the correct use of private instance fields with **get** and **set** property accessors.

[C#]

```
public struct Point
{
  private int xValue;
  private int yValue;

  public Point(int x, int y)
  {
    this.xValue = x;
    this.yValue = y;
  }

  public int X
  {
    get
    {
      return xValue;
    }
    set
    {
      xValue = value;
    }
  }
  public int Y
  {
    get
    {
      return yValue;
    }
    set
    {
      yValue = value;
    }
  }
}
```

[VB.NET]

```
Public Structure Point
  Private xValue As Integer
  Private yValue As Integer

  Public Sub New(x As Integer, y As Integer)
    Me.xValue = x
    Me.yValue = y
  End Sub

  Public Property X() As Integer
    Get
      Return xValue
    End Get
    Set
      xValue = value
    End Set
  End Property
  Public Property Y() As Integer
    Get
      Return yValue
    End Get
    Set
      yValue = value
    End Set
  End Property
End Structure
```

- Expose a field to a derived class by using a **protected** property that returns the value of the field. This is illustrated in the following code example.

[C#]

```
public class Control: Component
{
  private int handle;
  protected int Handle
  {
    get
    {
      return handle;
    }
  }
}
```

[VB.NET]

```
Public Class Control
    Inherits Component
    Private handle As Integer

    Protected ReadOnly Property Handle() As Integer
      Get
        Return handle
```

```
        End Get
      End Property
    End Class
```

- It is recommended that you use read-only static fields instead of properties where the value is a global constant. This pattern is illustrated in the following code example.

[C#]

```
public struct Int32
{
  public static readonly int MaxValue = 2147483647;
  public static readonly int MinValue = -2147483648;
  // Insert other members here.
}
```

[VB.NET]

```
Public Structure Int32
  Public Const MaxValue As Integer = 2147483647
  Public Const MinValue As Integer = -2147483648
  ' Insert other members here.
End Structure
```

- Spell out all words used in a field name. Use abbreviations only if developers generally understand them. Do not use uppercase letters for field names. The following is an example of correctly named fields.

[C#]

```
class SampleClass
{
  string url;
  string destinationUrl;
}
```

[VB.NET]

```
Class SampleClass
  Private url As String
  Private destinationUrl As String
End Class
```

- Do not use Hungarian notation for field names. Good names describe semantics, not type.

- Do not apply a prefix to field names or static field names. Specifically, do not apply a prefix to a field name to distinguish between static and nonstatic fields. For example, applying a g_ or s_ prefix is incorrect.

- Use public static read-only fields for predefined object instances. If there are predefined instances of an object, declare them as public static read-only fields of the object itself. Use Pascal case because the fields are public. The following code example illustrates the correct use of public static read-only fields.

[C#]

```
public struct Color
{
  public static readonly Color Red = new Color(0x0000FF);
  public static readonly Color Green = new Color(0x00FF00);
  public static readonly Color Blue = new Color(0xFF0000);
  public static readonly Color Black = new Color(0x000000);
  public static readonly Color White = new Color(0xFFFFFF);

  public Color(int rgb)
  { // Insert code here.}
  public Color(byte r, byte g, byte b)
  { // Insert code here.}

  public byte RedValue
  {
    get
    {
      return Color;
    }
  }
  public byte GreenValue
  {
    get
    {
      return Color;
    }
  }
  public byte BlueValue
  {
    get
    {
      return Color;
    }
  }
}
```

[VB.NET]

```
Public Structure Color
  Public Shared Red As New Color(&HFF)
  Public Shared Green As New Color(&HFF00)
  Public Shared Blue As New Color(&HFF0000)
  Public Shared Black As New Color(&H0)
  Public Shared White As New Color(&HFFFFFF)

  Public Sub New(rgb As Integer)
  ' Insert code here.
  End Sub

  Public Sub New(r As Byte, g As Byte, b As Byte)
    ' Insert code here.
  End Sub

  Public ReadOnly Property RedValue() As Byte
    Get
      Return Color
    End Get
  End Property

  Public ReadOnly Property GreenValue() As Byte
    Get
      Return Color
    End Get
  End Property

  Public ReadOnly Property BlueValue() As Byte
    Get
      Return Color
    End Get
  End Property
End Structure
```

Constructor Usage Guidelines

The following rules outline the usage guidelines for constructors:

- Provide a default private constructor if there are only static methods and properties on a class. In the following example, the private constructor prevents the class from being created.

[C#]

```
public sealed class Environment
{
  // Private constructor prevents the class from being created.
  private Environment()
  {
    // Code for the constructor goes here.
  }
```

```
}
```

[VB.NET]

```
NotInheritable Public Class Environment
  ' Private constructor prevents the class from being created.
  Private Sub New()
    ' Code for the constructor goes here.
  End Sub
End Class
```

- Minimize the amount of work done in the constructor. Constructors should not do more than capture the constructor parameter or parameters. This delays the cost of performing further operations until the user uses a specific feature of the instance.

- Provide a **protected** (**Protected** in Visual Basic) constructor that can be used by types in a derived class.

- It is recommended that you not provide an empty constructor for a value type **struct**. If you do not supply a constructor, the runtime initializes all the fields of the **struct** to zero. This makes array and static field creation faster.

- Use parameters in constructors as shortcuts for setting properties. There should be no difference in semantics between using an empty constructor followed by property set accessors, and using a constructor with multiple arguments. The following three code examples are equivalent:

  [C#]

```
// Example #1.
Class SampleClass = new Class();
SampleClass.A = "a";
SampleClass.B = "b";

// Example #2.
Class SampleClass = new Class("a");
SampleClass.B = "b";

// Example #3.
Class SampleClass = new Class ("a", "b");
```

  [VB.NET]

```
' Example #1.
Dim SampleClass As New Class()
SampleClass.A = "a"
SampleClass.B = "b"

' Example #2.
Dim SampleClass As New Class("a")
SampleClass.B = "b"

' Example #3.
Dim SampleClass As New Class("a", "b")
```

- Use a consistent ordering and naming pattern for constructor parameters. A common pattern for constructor parameters is to provide an increasing number of parameters to allow the developer to specify a desired level of information. The more parameters that you specify, the more detail the developer can specify. In the following code example, there is a consistent order and naming of the parameters for all the SampleClass constructors.

[C#]

```
public class SampleClass
{
  private const string defaultForA = "default value for a";
  private const string defaultForB = "default value for b";
  private const string defaultForC = "default value for c";

  private string a;
  private string b;
  private string c;

  public MyClass():this(defaultForA, defaultForB, defaultForC) {}
  public MyClass (string a) : this(a, defaultForB, defaultForC) {}
  public MyClass (string a, string b) : this(a, b, defaultForC) {}
  public MyClass (string a, string b, string c)
  {
    this.a = a;
    this.b = b;
    this.c = c;
  }
}
```

[VB.NET]

```
Public Class SampleClass
  Private Const defaultForA As String = "default value for a"
  Private Const defaultForB As String = "default value for b"
  Private Const defaultForC As String = "default value for c"
  Private a As String
  Private b As String
  Private c As String

  Public Sub New()
    MyClass.New(defaultForA, defaultForB, defaultForC)
    Console.WriteLine("New()")
  End Sub

  Public Sub New(a As String)
    MyClass.New(a, defaultForB, defaultForC)
  End Sub

  Public Sub New(a As String, b As String)
    MyClass.New(a, b, defaultForC)
  End Sub
  Public Sub New(a As String, b As String, c As String)
    Me.a = a
```

```
        Me.b = b
        Me.c = c
    End Sub
End Class
```

Method Usage Guidelines

The following rules outline the usage guidelines for methods:

- Choose a name for your event based on the recommended Method Naming Guidelines.

- Do not use Hungarian notation.

- By default, methods are nonvirtual. Maintain this default in situations where it is not necessary to provide virtual methods. For more information about implementing inheritance, see Base Class Usage Guidelines.

Method Overloading Guidelines

Method overloading occurs when a class contains two methods with the same name, but different signatures. This section provides some guidelines for the use of overloaded methods.

- Use method overloading to provide different methods that do semantically the same thing.

- Use method overloading instead of allowing default arguments. Default arguments do not version well and therefore are not allowed in the Common Language Specification (CLS). The following code example illustrates an overloaded String.IndexOf method.

  [C#]
  ```
  int String.IndexOf (String name);
  int String.IndexOf (String name, int startIndex);
  ```

  [VB.NET]
  ```
  Function String.IndexOf(name As String) As Integer
  Function String.IndexOf(name As String, startIndex As Integer) As Integer
  ```

- Use default values correctly. In a family of overloaded methods, the complex method should use parameter names that indicate a change from the default state assumed in the simple method. For example, in the following code, the first method assumes the search will not be case-sensitive. The second method uses the name ignoreCase rather than caseSensitive to indicate how the default behavior is being changed.

  [C#]
  ```
  // Method #1: ignoreCase = false.
  MethodInfo Type.GetMethod(String name);
  ```

```
// Method #2: Indicates how the default behavior of method #1 is being //
changed.
 MethodInfo Type.GetMethod (String name, Boolean ignoreCase);
```

[VB.NET]

```
' Method #1: ignoreCase = false.
Function Type.GetMethod(name As String) As MethodInfo
' Method #2: Indicates how the default behavior of method #1
' is being changed.
Function Type.GetMethod(name As String, ignoreCase As Boolean) As
MethodInfo
```

- Use a consistent ordering and naming pattern for method parameters. It is common to provide a set of overloaded methods with an increasing number of parameters to allow the developer to specify a desired level of information. The more parameters that you specify, the more detail the developer can specify. In the following code example, the overloaded Execute method has a consistent parameter order and naming pattern variation. Each of the Execute method variations uses the same semantics for the shared set of parameters.

[C#]

```
public class SampleClass
{
  readonly string defaultForA = "default value for a";
  readonly string defaultForB = "default value for b";
  readonly string defaultForC = "default value for c";

  public void Execute()
  {
    Execute(defaultForA, defaultForB, defaultForC);
  }

  public void Execute (string a)
  {
    Execute(a, defaultForB, defaultForC);
  }

  public void Execute (string a, string b)
  {
    Execute (a, b, defaultForC);
  }

  public virtual void Execute (string a, string b, string c)
  {
    Console.WriteLine(a);
    Console.WriteLine(b);
    Console.WriteLine(c);
    Console.WriteLine();
  }
}
```

[VB.NET]

```
Public Class SampleClass
  Private defaultForA As String = "default value for a"
  Private defaultForB As String = "default value for b"
  Private defaultForC As String = "default value for c"

  Overloads Public Sub Execute()
    Execute(defaultForA, defaultForB, defaultForC)
  End Sub

  Overloads Public Sub Execute(a As String)
    Execute(a, defaultForB, defaultForC)
  End Sub

  Overloads Public Sub Execute(a As String, b As String)
    Execute(a, b, defaultForC)
  End Sub
  Overloads Public Overridable Sub Execute(a As String, b As String, c
        As String)
    Console.WriteLine(a)
    Console.WriteLine(b)
    Console.WriteLine(c)
    Console.WriteLine()
  End Sub
End Class
```

This consistent pattern applies if the parameters have different types. Note that the only method in the group that should be virtual is the one that has the most parameters.

- Use method overloading for variable numbers of parameters. Where it is appropriate to specify a variable number of parameters to a method, use the convention of declaring n methods with increasing numbers of parameters. Provide a method that takes an array of values for numbers greater than n. For example, n=3 or n=4 is appropriate in most cases. The following example illustrates this pattern.

  [C#]

```
public class SampleClass
{
  public void Execute(string a)
  {
    Execute(new string[] {a});
  }

  public void Execute(string a, string b)
  {
    Execute(new string[] {a, b});
  }

  public void Execute(string a, string b, string c)
  {
```

```
    Execute(new string[] {a, b, c});
  }

  public virtual void Execute(string[] args)
  {
    foreach (string s in args)
    {
      Console.WriteLine(s);
    }
  }
}
```

[VB.NET]

```
Public Class SampleClass

  Overloads Public Sub Execute(a As String)
    Execute(New String() {a})
  End Sub

  Overloads Public Sub Execute(a As String, b As String)
    Execute(New String() {a, b})
  End Sub

  Overloads Public Sub Execute(a As String, b As String, c As String)
    Execute(New String() {a, b, c})
  End Sub

  Overloads Public Overridable Sub Execute(args() As String)
    Dim s As String
    For Each s In args
      Console.WriteLine(s)
    Next s
  End Sub
End Class
```

- If you must provide the ability to override a method, make only the most complete overload virtual and define the other operations in terms of it. The following example illustrates this pattern.

[C#]

```
public class SampleClass
{
  private string myString;

  public MyClass(string str)
  {
    this.myString = str;
  }

  public int IndexOf(string s)
  {
    return IndexOf (s, 0);
  }
```

```
    public int IndexOf(string s, int startIndex)
    {
      return IndexOf(s, startIndex, myString.Length - startIndex );
    }

    public virtual int IndexOf(string s, int startIndex, int count)
    {
      return myString.IndexOf(s, startIndex, count);
    }
  }
```

[VB.NET]

```
Public Class SampleClass
  Private myString As String

  Public Sub New(str As String)
    Me.myString = str
  End Sub

  Overloads Public Function IndexOf(s As String) As Integer
    Return IndexOf(s, 0)
  End Function

  Overloads Public Function IndexOf(s As String, startIndex As
        Integer) As Integer
    Return IndexOf(s, startIndex, myString.Length - startIndex)
  End Function

  Overloads Public Overridable Function IndexOf(s As String,
        startIndex As Integer, count As Integer) As Integer
    Return myString.IndexOf(s, startIndex, count)
  End Function
End Class
```

Methods With Variable Numbers of Arguments

You might want to expose a method that takes a variable number of arguments. A classic example is the **printf** method in the C programming language. For managed class libraries, use the **params** (**ParamArray** in Visual Basic) keyword for this construct. For example, use the following code instead of several overloaded methods.

[C#]

```
void Format(string formatString, params object [] args)
```

[VB.NET]

```
Sub Format(formatString As String, ParamArray args() As Object)
```

You should not use the **VarArgs** calling convention exclusively because the Common Language Specification does not support it.

For extremely performance-sensitive code, you might want to provide special code paths for a small number of elements. You should only do this if you are going to special case the entire code path (not just create an array and call the more general method). In such cases, the following pattern is recommended as a balance between performance and the cost of specially cased code.

[C#]
```
void Format(string formatString, object arg1)
void Format(string formatString, object arg1, object arg2)
void Format(string formatString, params object [] args)
```

[VB.NET]
```
Sub Format(formatString As String, arg1 As Object)
Sub Format(formatString As String, arg1 As Object, arg2 As Object)
Sub Format(formatString As String, ParamArray args() As Object)
```

Event Usage Guidelines

The following rules outline the usage guidelines for events:

- Choose a name for your event based on the recommended Event Naming Guidelines.
- Do not use Hungarian notation.
- When you refer to events in documentation, use the phrase, "an event was raised" instead of "an event was fired" or "an event was triggered."
- In languages that support the **void** keyword, use a return type of void for event handlers, as shown in the following code example.

  [C#]
  ```
  public delegate void MouseEventHandler(object sender, MouseEventArgs e);
  ```

- Event classes should extend the System.EventArgs Class, as shown in the following example.

  [C#]
  ```
  public class MouseEvent: EventArgs { }
  ```

  [VB.NET]
  ```
  Public Class MouseEventArgs
  Inherits EventArgs
  ' Code for the class goes here.
  End Class
  ```

- Implement an event handler using the public EventHandler Click syntax. Provide an add and a remove accessor to add and remove event handlers. If your programming language does not support this syntax, name methods add_Click and remove_Click.

- If a class raises multiple events, the compiler generates one field per event delegate instance. If the number of events is large, the storage cost of one field per delegate might not be acceptable. For those situations, the .NET Framework provides a construct called event properties that you can use together with another data structure (of your choice) to store event delegates. The following code example illustrates how the **Component** class implements this space-efficient technique for storing handlers.

- Use a **protected** (**Protected** in Visual Basic) virtual method to raise each event. This technique is not appropriate for sealed classes, because classes cannot be derived from them. The purpose of the method is to provide a way for a derived class to handle the event using an override. This is more natural than using delegates in situations where the developer is creating a derived class. The name of the method takes the form OnEventName, where EventName is the name of the event being raised. For example:

[C#]

```
public class Button
{
  ButtonClickHandler onClickHandler;

  protected virtual void OnClick(ClickEvent e)
  {
    // Call the delegate if non-null.
    if (onClickHandler != null)
         onClickHandler(this, e);
  }
}
```

[VB.NET]

```
Public Class Button
  Private onClickHandler As ButtonClickHandler
  Protected Overridable Sub OnClick(e As ClickEvent)
    ' Call the delegate if non-null.
    If Not (onClickHandler Is Nothing) Then
      onClickHandler(Me, e)
    End If
  End Sub
End Class
```

The derived class can choose not to call the base class during the processing of OnEventName. Be prepared for this by not including any processing in the OnEventName method that is required for the base class to work correctly.

- You should assume that an event handler could contain any code. Classes should be ready for the event handler to perform almost any operation, and in all cases the object should be left in an appropriate state after the event has been raised. Consider using a try/finally block at the point in code where the event is raised. Since the developer can perform a callback function on the object to perform other actions, do not assume anything about the object state when control returns to the point at which the event was raised. For example:

[C#]

```
public class Button
{
  ButtonClickHandler onClickHandler;

  protected void DoClick()
  {
    // Paint button in indented state.
    PaintDown();
      try
    {
      // Call event handler.
      OnClick();
    }
    finally
    {
      // Window might be deleted in event handler.
      if (windowHandle != null)
        // Paint button in normal state.
        PaintUp();
    }
  }

  protected virtual void OnClick(ClickEvent e)
  {
    if (onClickHandler != null)
      onClickHandler(this, e);
  }
}
```

[VB.NET]

```
Public Class Button
  Private onClickHandler As ButtonClickHandler
  Protected Sub DoClick()
    ' Paint button in indented state.
    PaintDown()
    Try
      ' Call event handler.
      OnClick()
    Finally
      ' Window might be deleted in event handler.
      If Not (windowHandle Is Nothing) Then
        ' Paint button in normal state.
        PaintUp()
```

```
        End If
      End Try
    End Sub
    Protected Overridable Sub OnClick(e As ClickEvent)
      If Not (onClickHandler Is Nothing) Then
        onClickHandler(Me, e)
      End If
    End Sub
  End Class
```

- Use or extend the System.ComponentModel.CancelEventArgs Class to allow the developer to control the default behavior of an object. For example, the TreeView control raises a **CancelEvent** when the user is about to edit a node label. The following code example illustrates how a developer can use this event to prevent a node from being edited.

  [C#]

```
public class Form1: Form
{
  TreeView treeView1 = new TreeView();

  void treeView1_BeforeLabelEdit(object source,
    NodeLabelEditEvent e)
  {
    e.cancel = true;
  }
}
```

  [VB.NET]

```
Public Class Form1
  Inherits Form
  Private treeView1 As New TreeView()

  Sub treeView1_BeforeLabelEdit(source As Object, e As NodeLabelEditEvent)
    e.cancel = True
  End Sub
End Class
```

Note that in this case, no error is generated to the user. The label is read-only.

The **CancelEvent** is not appropriate in cases where the developer would cancel the operation and return an exception. In these cases, the event does not derive from **CancelEvent**. You should raise an exception inside of the event handler in order to cancel. For example, the user might want to write validation logic in an edit control as shown.

[C#]

```
public class Form1: Form
{
  Edit edit1 = new Edit();
```

```
  void edit1_TextChanging(object source, Event e)
  {
    throw new RuntimeException("Invalid edit");
  }
}
```

[VB.NET]

```
Public Class Form1
  Inherits Form
  Private edit1 As Edit = New Edit()

  Sub edit1_TextChanging(source As Object, e As Event)
    Throw New RuntimeException("Invalid edit")
  End Sub
End Class
```

Type Usage Guidelines

Types are the units of encapsulation in the common language runtime. This section provides usage guidelines for the basic kinds of types.

Base Class Usage Guidelines

A class is the most common kind of type. A class can be abstract or sealed. An abstract class requires a derived class to provide an implementation. A sealed class does not allow a derived class. It is recommended that you use classes over other types.

Base classes are a useful way to group objects that share a common set of functionality. Base classes can provide a default set of functionality, while allowing customization though extension.

You should add extensibility or polymorphism to your design only if you have a clear customer scenario for it. For example, providing an interface for data adapters is difficult and serves no real benefit. Developers will still have to program against each adapter specifically, so there is only marginal benefit from providing an interface. However, you do need to support consistency between all adapters. Although an interface or abstract class is not appropriate in this situation, providing a consistent pattern is very important. You can provide consistent patterns for developers in base classes. Follow these guidelines for creating base classes.

Base Classes vs. Interfaces

An interface type is a partial description of a value, potentially supported by many object types. Use base classes instead of interfaces whenever possible. From a versioning perspective, classes are more flexible than interfaces. With a class, you can ship Version 1.0 and then in Version 2.0 add a new method to the class. As long as the method is not abstract, any existing derived classes continue to function unchanged.

Because interfaces do not support implementation inheritance, the pattern that applies to classes does not apply to interfaces. Adding a method to an interface is equivalent to adding an abstract method to a base class; any class that implements the interface will break because the class does not implement the new method.

Interfaces are appropriate in the following situations:

- Several unrelated classes want to support the protocol.
- These classes already have established base classes (for example, some are user interface (UI) controls, and some are XML Web services).
- Aggregation is not appropriate or practical.

In all other situations, class inheritance is a better model.

Protected Methods and Constructors

Provide class customization through protected methods. The public interface of a base class should provide a rich set of functionality for the consumer of the class. However, users of the class often want to implement the fewest number of methods possible to provide that rich set of functionality to the consumer. To meet this goal, provide a set of nonvirtual or final public methods that call through to a single protected method that provides implementations for the methods. This method should be marked with the Impl suffix. Using this pattern is also referred to as providing a Template method. The following code example demonstrates this process.

[C#]

```
public class MyClass
{
  private int x;
  private int y;
  private int width;
  private int height;
  BoundsSpecified specified;

  public void SetBounds(int x, int y, int width, int height)
   {
     SetBoundsCore(x, y, width, height, this.specified);
   }

   public void SetBounds(int x, int y, int width, int height,
     BoundsSpecified specified)
   {
     SetBoundsCore(x, y, width, height, specified);
   }

   protected virtual void SetBoundsCore(int x, int y, int width, int
     height, BoundsSpecified specified)
   {
        // Add code to perform meaningful opertions here.
```

```
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
        this.specified = specified;
    }
}
```

[VB.NET]

```
Public Class SampleClass
  Private x As Integer
  Private y As Integer
  Private width As Integer
  Private height As Integer
  Private specified As BoundsSpecified

  Overloads Public Sub SetBounds(x As Integer, y As Integer, width As
      Integer, height As Integer)
    SetBoundsCore(x, y, width, height, Me.specified)
  End Sub

  Overloads Public Sub SetBounds(x As Integer, y As Integer, width As
      Integer, height As Integer, specified As BoundsSpecified)

    SetBoundsCore(x, y, width, height, specified)
  End Sub

  Protected Overridable Sub SetBoundsCore(x As Integer, y As Integer,
      width As Integer, height As Integer,
      specified As BoundsSpecified)
    ' Insert code to perform meaningful operations here.
    Me.x = x
    Me.y = y
    Me.width = width
    Me.height = height
    Me.specified = specified
    Console.WriteLine("x {0}, y {1}, width {2}, height {3}, bounds {4}",
      Me.x, Me.y, Me.width, Me.height, Me.specified)
  End Sub
End Class
```

Many compilers will insert a **public** or **protected** constructor if you do not. Therefore, for better documentation and readability of your source code, you should explicitly define a protected constructor on all abstract classes.

Sealed Class Usage Guidelines

The following rules outline the usage guidelines for sealed classes:

- Use sealed classes if it will not be necessary to create derived classes. A class cannot be derived from a sealed class.

- Use sealed classes if there are only static methods and properties on a class. The following code example shows a correctly defined sealed class.

[C#]

```
public sealed class Runtime
{
  // Private constructor prevents the class from being created.
  private Runtime();

  // Static method.
  public static string GetCommandLine()
  {
    // Implementation code goes here.
  }
}
```

[VB.NET]

```
NotInheritable Public Class Runtime
  ' Private constructor prevents the class from being created.
  Private Sub New()
  End Sub

  ' Static method.
  Public Shared Sub GetCommandLine() As String
    ' Implementation code goes here.
  End Sub
End Class
```

Delegate Usage Guidelines

A delegate is a powerful tool that allows the managed code object model designer to encapsulate method calls. Delegates are useful for event notifications and callback functions.

Event notifications

Use the appropriate event design pattern for events even if the event is not user interface-related. For more information on using events, see the Event Usage Guidelines.

Callback functions

Callback functions are passed to a method so that user code can be called multiple times during execution to provide customization. Passing a Compare callback function to a sort routine is a classic example of using a callback function. These methods should use the callback function conventions described in Callback Function Usage.

Name end callback functions with the suffix Callback.

Value Type Usage Guidelines

A value type describes a value that is represented as a sequence of bits stored on the stack. For a description of all the .NET Framework's built-in data types, see Value Types. This section provides guidelines for using the structure (**struct**) and enumeration (**enum**) value types.

Struct Usage Guidelines

It is recommended that you use a **struct** for types that meet any of the following criteria:

- Act like primitive types.
- Have an instance size under 16 bytes.
- Are immutable.
- Value semantics are desirable.

The following example shows a correctly defined structure.

[C#]

```
public struct Int32: IComparable, IFormattable
{
  public const int MinValue = -2147483648;
  public const int MaxValue = 2147483647;

  public static string ToString(int i)
  {
    // Insert code here.
  }

  public string ToString(string format, IFormatProvider formatProvider)
  {
    // Insert code here.
  }

  public override string ToString()
  {
    // Insert code here.
  }

  public static int Parse(string s)
  {
    // Insert code here.
    return 0;
  }

  public override int GetHashCode()
  {
    // Insert code here.
    return 0;
  }
```

```
  public override bool Equals(object obj)
  {
    // Insert code here.
    return false;
  }

  public int CompareTo(object obj)
  {
    // Insert code here.
    return 0;
  }

}
```

[VB.NET]

```
Public Structure Int32
  Implements IFormattable
  Implements IComparable
  Public Const MinValue As Integer = -2147483648
  Public Const MaxValue As Integer = 2147483647

  Private intValue As Integer

  Overloads Public Shared Function ToString(i As Integer) As String
    ' Insert code here.
  End Function

  Overloads Public Function ToString(ByVal format As String, ByVal
      formatProvider As IFormatProvider) As String Implements
      IFormattable.ToString
    ' Insert code here.
  End Function

  Overloads Public Overrides Function ToString() As String
    ' Insert code here.
  End Function
  Public Shared Function Parse(s As String) As Integer
    ' Insert code here.
    Return 0
  End Function

  Public Overrides Function GetHashCode() As Integer
    ' Insert code here.
    Return 0
  End Function

  Public Overrides Overloads Function Equals(obj As Object) As Boolean
    ' Insert code here.
    Return False
  End Function

  Public Function CompareTo(obj As Object) As Integer Implements
```

```
        IComparable.CompareTo
      ' Insert code here.
      Return 0
    End Function
End Structure
```

When using a **struct**, do not provide a default constructor. The runtime will insert a constructor that initializes all the values to a zero state. This allows arrays of structs to be created more efficiently. You should also allow a state where all instance data is set to zero, false, or null (as appropriate) to be valid without running the constructor.

Enum Usage Guidelines

The following rules outline the usage guidelines for enumerations:

- Use an **enum** to strongly type parameters, properties, and return types. Always define enumerated values using an **enum** if they are used in a parameter or property. This allows development tools to know the possible values for a property or parameter. The following example shows how to define an enum type.

  [C#]

```
public enum FileMode
{
  Append,
  Create,
  CreateNew,
  Open,
  OpenOrCreate,
  Truncate
}
```

  [VB.NET]

```
Public Enum FileMode
  Append
  Create
  CreateNew
  Open
  OpenOrCreate
  Truncate
End Enum
```

The following example shows the constructor for a **FileStream** object that uses the **FileMode** enum.

[C#]

```
public FileStream(string path, FileMode mode);
```

[VB.NET]

```
Public Sub New(ByVal path As String, ByVal mode As FileMode);
```

- Use the System.FlagsAttribute Class to create custom attribute for an enum if a bitwise OR operation is to be performed on the numeric values. This attribute is applied in the following code example.

  [C#]

```
[Flags]
public enum Bindings
{
  IgnoreCase = 0x01,
  NonPublic = 0x02,
  Static = 0x04,
  InvokeMethod = 0x0100,
  CreateInstance = 0x0200,
  GetField = 0x0400,
  SetField = 0x0800,
  GetProperty = 0x1000,
  SetProperty = 0x2000,
  DefaultBinding = 0x010000,
  DefaultChangeType = 0x020000,
  Default = DefaultBinding | DefaultChangeType,
  ExactBinding = 0x040000,
  ExactChangeType = 0x080000,
  BinderBinding = 0x100000,
  BinderChangeType = 0x200000
}
```

  [VB.NET]

```
<Flags> _
Public Enum Bindings
  IgnoreCase = &H1
  NonPublic = &H2
  Static = &H4
  InvokeMethod = &H100
  CreateInstance = &H200
  GetField = &H400
  SetField = &H800
  GetProperty = &H1000
  SetProperty = &H2000
  DefaultBinding = &H10000
  DefaultChangeType = &H20000
  [Default] = DefaultBinding Or DefaultChangeType
  ExactBinding = &H40000
  ExactChangeType = &H80000
  BinderBinding = &H100000
  BinderChangeType = &H200000
End Enum
```

Note An exception to this rule is when encapsulating a Win32 API. It is common to have internal definitions that come from a Win32 header. You can leave these with the Win32 casing, which is usually all capital letters.

- Use an **enum** with the flags attribute only if the value can be completely expressed as a set of bit flags. Do not use **an** enum for open sets (such as the operating system version).

- Do not assume that **enum** arguments will be in the defined range. Perform argument validation as illustrated in the following code example.

[C#]

```
public  void SetColor (Color color)
{
  if (!Enum.IsDefined (typeof(Color), color)
    throw new ArgumentOutOfRangeException();
}
```

[VB.NET]

```
Public Sub SetColor(newColor As Color)
  If Not [Enum].IsDefined(GetType(Color), newColor) Then
    Throw New ArgumentOutOfRangeException()
  End If
End Sub
```

- Use an **enum** instead of static final constants.

- Use type **Int32** as the underlying type of an **enum** unless either of the following is true:

    - The **enum** represents flags and there are currently more than 32 flags, or the **enum** might grow to many flags in the future.

    - The type needs to be different from **int** for backward compatibility.

- Do not use a nonintegral **enum** type. Use only **Byte**, **Int16**, **Int32**, or **Int64**.

- Do not define methods, properties, or events on an **enum**.

- Do not use an **Enum** suffix on **enum** types.

Nested Type Usage Guidelines

A nested type is a type defined within the scope of another type. Nested types are very useful for encapsulating implementation details of a type, such as an enumerator over a collection, because they can have access to private state.

Public nested types should be used rarely. Use them only in situations where both of the following are true:

- The nested type logically belongs to the containing type.

- The nested type is not used often, or at least not directly.

The following examples illustrates how to define types with and without nested types:

[C#]

```
// With nested types.
ListBox.SelectedObjectCollection
// Without nested types.
ListBoxSelectedObjectCollection

// With nested types.
RichTextBox.ScrollBars
// Without nested types.
RichTextBoxScrollBars
```

Do not use nested types if the following are true:

- The type is used in many different methods in different classes. The FileMode Enumeration is a good example of this kind of type.

- The type is commonly used in different APIs. The StringCollection Class is a good example of this kind of type.

Attribute Usage Guidelines

The .NET Framework enables developers to invent new kinds of declarative information, to specify declarative information for various program entities, and to retrieve attribute information in a run-time environment. For example, a framework might define a HelpAttribute attribute that can be placed on program elements such as classes and methods to provide a mapping from program elements to their documentation. New kinds of declarative information are defined through the declaration of attribute classes, which might have positional and named parameters. For more information about attributes, see Writing Custom Attributes.

The following rules outline the usage guidelines for attribute classes:

- Add the Attribute suffix to custom attribute classes, as shown in the following example.

  [C#]

  ```
  public class ObsoleteAttribute{}
  ```

  [VB.NET]

  ```
  Public Class ObsoleteAttribute{}
  ```

- Specify AttributeUsage on your attributes to define their usage precisely, as shown in the following example.

  [C#]

  ```
  [AttributeUsage(AttributeTargets.All, Inherited = false, AllowMultiple = true)]
  public class ObsoleteAttribute: Attribute {}
  ```

[VB.NET]

```
<AttributeUsage(AttributeTargets.All, Inherited := False, AllowMultiple :=
True)> _

Public Class ObsoleteAttribute
  Inherits Attribute
  ' Insert code here.
End Class
```

- Seal attribute classes whenever possible, so that classes cannot be derived from them.
- Use positional arguments for required parameters.
- Use named arguments for optional parameters.
- Do not name a parameter with both named and positional arguments.
- Provide a read-only property with the same name as each positional argument, but change the case to differentiate between them.
- Provide a read/write property with the same name as each named argument, but change the case to differentiate between them.

[C#]

```
public class NameAttribute: Attribute
{
  public NameAttribute (string username)
  {
     // Implement code here.
  }
  public string UserName
  {
    get
    {
      return UserName;
    }
  }
  public int Age
  {
    get
    {
      return Age;
    }
    set
    {
      Age = value;
    }
  }
  // Positional argument.
}
```

[VB.NET]

```
Public Class NameAttribute
  Inherits Attribute

  Public Sub New(username As String)
    ' Implement code here.
  End Sub

  Public ReadOnly Property UserName() As String
    Get
      Return UserName
    End Get
  End Property

  Public Property Age() As Integer
    Get
      Return Age
    End Get
    Set
      Age = value
    End Set
  End Property
  ' Positional argument.
End Class
```

Setting Environment Options (VB.NET)

- Use Option **Explicit**.
- Use Option **Strict**.

Reference:

Design Guidelines for Class Library Developers in MSDN.NET.

ms-help://MS.VSCC/MS.MSDNVS/cpgenref/html/cpconnetframeworkdesignguidelines.htm