C#/VB .NET Coding Guidelines

Steve Sartain
Iridium Software, Senior Architect

October 2006

Contents

1 Overview

This document is a working document - it is not designed to meet the requirement that we have "a" coding standard but instead it is an acknowledgment that we can make our lives much easier in the long term if we all agree to a common set of conventions when writing code.

Inevitably, there are many places in this document where I have simply had to make a choice between two or more equally valid alternatives. I have tried to actually think about the relative merits of each alternative but inevitably some of my personal preferences have come into play.

This document is not fixed in stone; but it is not a suggestion, either. The only thing worse than no coding standard is multiple coding standards so these coding standards are mandatory where they apply (see the section When Does This Document Apply below).

However, if you think that something could be improved, or even if you think that I've made a wrong call somewhere, then let me know so we can review it.

I hope you find that this document is actually readable. I hate standards documents that are so dry as to be about as interesting as reading the Yellow Pages. However, do not assume that this document is any less important than those drier Yellow Pages. Iridium Software takes these standards very seriously.

2 When Does This Document Apply

It is the intention that all code written for or by Iridium Software adheres to this standard. However, there are some cases where it is impractical or impossible to apply these conventions.

This document applies to **all code** except the following:

2.1 Code changes made to existing systems not written to this standard

In general, it is a good idea to make your changes conform to the surrounding code style wherever possible. You might choose to adopt this standard for major additions to existing systems or when you are adding code that you think will become part of the Iridium Software code library.

2.2 Code written for customers that require that their standards should be adopted

Iridium Software may, from time to time work with customers that have their own coding standards. Most coding standards applicable to a Microsoft development language derive at least some of their content from a Microsoft white paper that documented a set of suggested naming standards. For this reason many coding standards are broadly compatible with each other. This document goes a little further than most in some areas; however it is likely that these extensions will not conflict with most other coding standards. We must be absolutely clear on this point: if there is a conflict, the customer's coding standards are to apply - **always**.

3 Naming Guidelines

Of all the components that make up a coding standard, naming standards are the most visible and arguably the most important.

Having a consistent standard for naming the various objects in your program will save you an enormous amount of time both during the development process itself and also during any later maintenance work.

3.1 Overview

For those of you coding in VB.NET, first things first, **always** use Option Explicit. The reasons are so obvious that I won't discuss it any further. If you don't agree, see me and we'll discuss it amicably. Secondly, you must set **Option Strict** on. Again, I shouldn't need to explain the advantages of this.

Remove the Visual Basic reference from your project. Making use of the original Visual Basic functions has been proven to be up to 1000 times slower than the .NET counterparts; you have been warned.

3.2 Capitalisation Styles

Use the following three conventions for capitalising identifiers.

3.2.1 Pascal Case

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

```
BackColor
```

3.2.2 Camel Case

The first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized. For example:

```
backColor
```

3.2.3 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.IO
```

You might also have to capitalize identifiers to maintain compatibility with existing, unmanaged symbol schemes, where all uppercase characters are often used for enumerations and constant values. In general, these symbols should not be visible outside of the assembly that uses them.

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	`ValueChange`
Exception class	Pascal	`WebException` Note: Always ends with the suffix Exception.
Read-only Static field	Pascal	`RedValue`
Interface	Pascal	`IDisposable` Note: Interfaces always begin with the prefix I.
Method	Pascal	`ToString`
Namespace	Pascal	`System.Drawing`
Parameter	Camel	`typeName`
Property	Pascal	`BackColor`
Protected instance field	Camel	`redValue` Note: Rarely used. A property is preferable to using a protected instance field.
Public instance field	Pascal	`RedValue` Note: Rarely used. A property is preferable to using a public instance field.

3.3 Case Sensitivity (not applicable to VB)

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

1. 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.

2. 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 IridiumSoftware
Namespace iridiumsoftware
```

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

```
void MyFunction(string a, string A)
```

4. 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.

```
System.Windows.Forms.Point p
System.Windows.Forms.POINT p
```

5. 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}
```

6. 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()
```

3.4 Abbreviations

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

7. Do not use abbreviations or contractions as parts of identifier names. For example, use `GetWindow` instead of `GetWin`.

8. 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.

9. Do not use acronyms that are not generally accepted in the computing field. (For example, `XML`, `TTL`, `DNS`, `UI`, `IP` and `IO` are all OK.)

10. 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`.

11. 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.

3.5 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 MSDN topic class library for a list of .NET Framework namespaces.

In addition, avoid using identifiers that conflict with the following keywords.

As	Assembly	Auto	Base	Boolean
ByRef	Byte	ByVal	Call	Case
Catch	CBool	CByte	CChar	CDate
CDec	CDbl	Char	CInt	Class
CLng	CObj	Const	CShort	CSng
CStr	CType	Date	Decimal	Declare
Default	Delegate	Dim	Do	Double
Each	Else	ElseIf	End	Enum
Erase	Error	Event	Exit	ExternalSource
False	Finalize	Finally	Float	For
Friend	Function	Get	GetType	Goto
Handles	If	Implements	Imports	In
Inherits	Integer	Interface	Is	Let
Lib	Like	Long	Loop	Me
Mod	Module	MustInherit	MustOverride	MyBase
MyClass	Namespace	New	Next	Not
Nothing	NotInheritable	NotOverridable	Object	On
Option	Optional	Or	Overloads	Overridable
Overrides	ParamArray	Preserve	Private	Property
Protected	Public	RaiseEvent	ReadOnly	ReDim
Region	REM	RemoveHandler	Resume	Return
Select	Set	Shadows	Shared	Short
Single	Static	Step	Stop	String

```
Structure       Sub             SyncLock        Then            Throw
To              True            Try             TypeOf          Unicode
Until           volatile        When            While           With
WithEvents      WriteOnly       Xor             eval            extends
instanceof      package         var
```

3.6 Avoid 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.

Visual Basic

```
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)
```

C#

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

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

Visual Basic

```
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)
```

C#

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

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. The following table lists fundamental data type names and their universal substitutions.

Type Name					
C#	**Visual Basic**	**Jscript**	**VC++**	**Ilasm**	**Universal**
sbyte	SByte	sByte	char	int8	SByte
byte	Byte	byte	unsigned char	unsigned int8	Byte
short	Short	short	short	int16	Int16
ushort	UInt16	ushort	unsigned short	unsigned int16	UInt16
int	Integer	int	Int	int32	Int32
uint	UInt32	uint	unsigned int	unsigned int32	UInt32
long	Long	long	__int64	int64	Int64
ulong	UInt64	ulong	unsigned __int64	unsigned int64	UInt64
float	Single	float	float	float32	Single
double	Double	double	double	float64	Double
bool	Boolean	boolean	bool	bool	Boolean
char	Char	char	wchar_t	char	Char
string	String	string	String	string	String
object	Object	object	Object	object	Object

For example, a class that supports reading a variety of data types from a stream might have the following methods. As should be noted, it is generally better practice to use .NET native data types rather than the language specific ones: in VB `Integer` would be `Int32` and `Long` would be `Int64`. This both aids cross platform development, but also inform the developer at a glance of the size of the data type.

Visual Basic

```
ReadDouble() As Double
ReadSingle() As Single
ReadInt64() As Long
ReadInt32() As Integer
ReadInt16() As Short
```

Visual Basic - preferred

```
ReadDouble() As Double
ReadSingle() As Single
ReadInt64() As Int64
ReadInt32() As Int32
ReadInt16() As Int16
```

C#

```
double ReadDouble();
float ReadSingle();
long ReadInt64();
int ReadInt32();
short ReadInt16();
```

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

Visual Basic

```
ReadDouble() As Double
ReadSingle() As Single
ReadLong() As Long
ReadInteger() As Integer
ReadShort() As Short
```

C#

```
double ReadDouble();
float ReadFloat();
long ReadLong();
int ReadInt();
short ReadShort();
```

3.7 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.

```
CompanyName.TechnologyName[.Feature][.Design]
```

For example:

```
IridiumSoftware.IridiumX
IridiumSoftware.IridiumX.Design
```

Prefixing namespace names with a company name or other well-established brands 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, recognised technology name at the second level of a hierarchical name. Use organisational 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 System.Web.UI.Design depend on the classes in System.Web.UI. However, the classes in System.Web.UI do not depend on the classes in System.Web.UI.Design.

You should use Pascal Case for namespaces, and separate logical components with periods, as in `Microsoft.Office.PowerPoint`. If your brand employs non-traditional 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.IridiumSoftware` 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.

Finally, note that a namespace name does not have to parallel an assembly name. For example, if you name an assembly `MyCompany.MyTechnology.dll`, it does not have to contain a `MyCompany.MyTechnology` namespace.

3.8 Class Naming Guidelines

The following rules outline the guidelines for naming classes:

1. Use a noun or noun phrase to name a class.
2. Use Pascal Case.
3. Use abbreviations sparingly.
4. Do not use a type prefix, such as `C` or class, on a class name. For example, use the class name `FileStream` rather than `CFileStream`.
5. Do not use the underscore character (_).
6. 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.
7. 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`. Use reasonable judgment in applying this rule. For example, `Button` is an appropriate name

for a class derived from `Control`. Although a button is a kind of control, making `Control` a part of the class name would lengthen the name unnecessarily.

The following are examples of correctly named classes.

Visual Basic

```
Public Class FileStream
Public Class Button
Public Class String
```

C#

```
public class FileStream
public class Button
public class String
```

3.9 Interface Naming Guidelines

The following rules outline the naming guidelines for interfaces:

1. Name interfaces with nouns or noun phrases, or adjectives that describe behaviour. For example, the interface name **`IComponent`** uses a descriptive noun. The interface name `ICustomAttributeProvider` uses a noun phrase. The name `IPersistable` uses an adjective.

2. Use Pascal Case.

3. Use abbreviations sparingly.

4. Prefix interface names with the letter **`I`**, to indicate that the type is an interface.

5. 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.

6. Do not use the underscore character (_).

The following are examples of correctly named interfaces.

Visual Basic

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

C#

```
public interface IServiceProvider
public interface IFormatable
```

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

Visual Basic

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

Public Class Component
    Implements IComponent

    ' Implementation goes here.
End Class
```

C#

```
public interface IComponent
{
    // Implementation goes here.
}

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

3.10 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.

Visual Basic

```
Public Class ObsoleteAttribute
```

C#

```
public class ObsoleteAttribute{}
```

3.11 Enumeration Type Naming Guidelines

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

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

5. Always add the `FlagsAttribute` to a bit field `Enum` type.

3.12 Static Field Naming Guidelines

The following rules outline the naming guidelines for static fields:

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

3.13 Parameter Naming Guidelines

The following rules outline the naming guidelines for parameters:

1. 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.
2. Use Camel Case for parameter names.
3. 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.
4. 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.
5. Do not prefix parameter names with Hungarian type notation.

The following are examples of correctly named parameters.

Visual Basic

```
GetType(typeName As String) As Type
Format(format As String, args As Object()) As String
```

C#

```
Type GetType(string typeName)
string Format(string format, object[] args)
```

3.14 Method Naming Guidelines

The following rules outline the naming guidelines for methods:

1. Use verbs or verb phrases to name methods.

2. Use Pascal Case.

The following are examples of correctly named methods.

```
RemoveAll()
GetCharArray()
Invoke()
```

3.15 Property Naming Guidelines

The following rules outline the naming guidelines for properties:

1. Use a noun or noun phrase to name properties.

2. Use Pascal Case.

3. Do not use Hungarian notation.

4. 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.

Visual Basic

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

C#

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

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

Visual Basic

```
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(ByVal Value As Color)
            ' Insert Code Here
        End Set
    End Property
End Class
```

C#

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

public class Control
{
    public Color Color
    {
        get {// Insert Code Here}
        set {// Insert Code Here}
    }
}
```

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

Visual Basic

```
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(ByVal Value As Integer)
            ' Insert Code Here
        End Set
    End Property
End Class
```

C#

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

public class Control
{
    public int Color
    {
        get {// Insert Code Here}
        set {// Insert Code Here}
    }
}
```

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).

3.16 Event Naming Guidelines

The following rules outline the naming guidelines for events:

1. Use an `EventHandler` suffix on event handler names.

2. 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.

3. Name an event argument class with the `EventArgs` suffix.

4. Consider naming events with a verb.

5. 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 cancelled should have a `Closing` event and a `Closed` event. Do not use the `BeforeXX/AfterXXX` naming pattern.

6. Do not use a prefix or suffix on the event declaration on the type. For example, use `Close` instead of `OnClose`.

7. 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.

Visual Basic

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

C#

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

The following example illustrates a correctly named event argument class.

Visual Basic

```
Public Class MouseEventArgs
    Inherits EventArgs
```

```
    Public Sub New MouseEventArgs(x As Int32, y As Int32)
        Me.x = x
        Me.y = y
    End Sub

    Public ReadOnly Property X As Int32
        Get
            Return x
        End Get
    End Property
    Private x As Int32

    Public ReadOnly Property Y As Int32
        Get
            Return y
        End Get
    End Property
    Private y As Int32
End Class
```

C#

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

    public int X
    {
        get { return x; }
    }
    int x;

    public int Y
    {
        get { return y; }
    }
    int y;
}
```

3.17 Control Naming Guidelines

Although not considered entirely correct, the choice to prefix design-time controls with a predetermined string is a sound one. It allows the developer to distinguish easily between design-time controls and other object kinds.

All controls **must** be changed from their default name to an appropriate replacement value. This will assist future development; and simply looks better. This must be done regardless of how insignificant the control appears.

You should consider taking the time to rename all controls prior to starting development of the form, this will likely speed up you work. In addition, if you select a control and choose to write code within an event of this control – whilst it will still work, the default name would be the original control name. If you do rename controls after adding event code, you **must** also modify the default event method names.

Controls have their own set of prefixes. They are used to identify the type of control so that code can be visually checked for correctness. They also assist in making it easy to know the name of a control without continually needing to look it up.

3.17.1 Specifying Particular Control Variants

In general, it is NOT a good idea to use specific identifiers for variations on a theme. For example, whether you are using a third-party button or a standard button generally is invisible to your code - you may have more properties to play with at design time to visually enhance the control, but your code usually traps the `Click()` event and maybe manipulates the `Enabled` and `Caption` properties, which will be common to all button-like controls.

Using generic prefixes means that your code is less dependent on the particular control variant that is used in an application and therefore makes code re-use simpler. Only differentiate between variants of a fundamental control if your code is totally dependent on some unique attribute of that particular control. Otherwise, use the generic prefixes where possible.

3.17.2 Table of Standard Control Prefixes

The following table is a list of the common types of controls you will encounter together with their prefixes:

Prefix	Control
`lbl`	`Label`
`llbl`	`LinkLabel`
`but`	`Button`
`txt`	`Textbox`
`mnu`	`MainMenu`
`chk`	`CheckBox`
`rdo`	`RadioButton`
`grp`	`GroupBox`
`pic`	`PictureBox`
`grd`	`Grid`
`lst`	`ListBox`
`cbo`	`ComboBox`
`lstv`	`ListView`
`tre`	`TreeView`
`tab`	`TabControl`
`dtm`	`DateTimePicker`
`mon`	`MonthCalendar`

sbr	ScrollBar
tmr	Timer
spl	Splitter
dud	DomainUpDown
nud	NumericUpDown
trk	TrackBar
pro	ProgressBar
rtxt	RichTextBox
img	ImageList
hlp	HelpProvider
tip	ToolTip
cmnu	ContextMenu
tbr	ToolBar
frm	Form
bar	StatusBar
nico	NotifyIcon
ofd	OpenFileDialog
sfd	SaveFileDialog
fd	FontDialog
cd	ColorDialog
pd	PrintDialog
ppd	PrintPreviewDialog
ppc	PrintPreviewControl
err	ErrorProvider
pdoc	PrintDocument
psd	PageSetupDialog
crv	CrystalReportViewer
pd	PrintDialog
fsw	FileSystemWatcher
log	EventLog
dire	DirectoryEntry

```
dirs      DirectorySearcher
msq       MessageQueue
pco       PerformanceCounter
pro       Process
ser       ServiceController
rpt       ReportDocument
ds        DataSet
olea      OleDbDataAdapter
olec      OleDbConnection
oled      OleDbCommand
sqla      SqlDbDataAdapter
sqlc      SqlDbConnection
sqld      SqlDbCommand
dvw       DataView
```

- Menu controls are subject to additional rules as defined below.
- There is no need to distinguish orientation.
- There is often a single Timer control in legacy projects, and it is used for a number of things. That makes it difficult to come up with a meaningful name. In this situation, it is acceptable to call the control simply "Timer".

3.17.3 Menu Controls

Menu controls should be named using the tag "`mnu`" followed by the complete path down the menu tree. This has the additional benefit of encouraging shallow menu hierarchies, which are generally considered to be "A Good Thing" in user interface design.

Here are some examples of menu control names:

```
mnuFileNew
mnuEditCopy
mnuInsertIndexAndTables
mnuTableCellHeightAndWidth
```

3.18 Data Naming Guidelines

As important as all the preceding rules are, the rewards you get for all the extra time thinking about the naming of objects will be small without this final step, something I call "data naming". The concept is simple but it is amazingly difficult to discipline yourself to do it without fail.

Essentially, the concept is simply an acknowledgment that any given data item has exactly one name. If something is called a `CustomerCode`, then that is what it is called **EVERYWHERE**. Not

`CustCode`. Not `CustomerID`. Not `CustID`. Not `CustomerCde`. No name except `CustomerCode` is acceptable.

Now, let's assume a customer code is a numeric item. If I need a customer code, I would use the name `CustomerCode`. If I want to display it in a text box, that text box must be called `txtCustomerCode`. If I want a combo box to display customer codes, that control would be called `cboCustomerCode`. If you need to store a customer code in a module level variable then it would be called `mCustomerCode`. If you want to convert it into a string (say to print it out later) you might use a statement like:

```
Dim mCustomerCode As String = CustomerCode.ToString()
```

I think you get the idea. It's really very simple. It's also incredibly difficult to do **EVERY** time, and it's only by doing it **EVERY** time that you get the real payoffs. It is just **SO** tempting to code the above line like this:

```
Dim mCustCode As String = CustomerCode.ToString()
```

3.18.1 Fields in Databases

As a general rule, the usual Property Naming Guidelines when naming fields in a database.

This may not always be practical or even possible. If the database already exists (either because the new program is referencing an existing database or because the database structure has been created as part of the database design phase) then it is not practical to apply these tags to every column or field. Even for new tables in existing databases, do not deviate from the conventions (hopefully) already in use in that database.

4 Class Member Usage Guidelines

4.1 Property Usage Guidelines

Determine whether a property or a method is more appropriate for your needs. For details on choosing between properties and methods, see Properties vs. Methods.

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.

4.1.1 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.

For example, a `TextBox` control might have two related properties: `DataSource` and `DataField`. `DataSource` specifies the table name, and `DataField` specifies the column name. Once both properties are specified, the control can automatically bind data from the table into the `Text` property of the control.

The following code example illustrates properties that can be set in any order.

Visual Basic

```
Dim txtData As New TextBox()
txtData.DataSource = "Publishers"
txtData.DataField = "AuthorID"
' The data-binding feature is now active.
```

C#

```
TextBox txtData = new TextBox();
txtData.DataSource = "Publishers";
txtData.DataField = "AuthorID";
// The data-binding feature is now active.
```

You can set the `DataSource` and `DataField` properties in any order. Therefore, the preceding code is equivalent to the following.

Visual Basic

```
Dim txtData As New TextBox()
txtData.DataField = "AuthorID"
txtData.DataSource = "Publishers"
' The data-binding feature is now active.
```

C#

```
TextBox txtData = new TextBox();
txtData.DataField = "AuthorID";
txtData.DataSource = "Publishers";
// The data-binding feature is now active.
```

You can also set a property to `null` (`Nothing` in Visual Basic) to indicate that the value is not specified.

Visual Basic

```
Dim txtData As New TextBox()
txtData.DataField = "AuthorID"
txtData.DataSource = "Publishers"
' The data-binding feature is now active.
txtData.DataSource = Nothing
' The data-binding feature is now inactive.
```

C#

```
TextBox txtData = new TextBox();
txtData.DataField = "AuthorID";
txtData.DataSource = "Publishers";
// The data-binding feature is now active.
txtData.DataSource = null;
// The data-binding feature is now inactive.
```

The following code example illustrates how to track the state of the "data binding" feature and automatically activate or deactivate it at the appropriate times.

Visual Basic

```
Public Class TextBox

    Public Property DataSource() As String
        Get
            Return mDataSource
        End Get
        Set(ByVal value As String)
            If value <> mDataSource Then
                ' Set the property value first,
                ' in case activate fails.
                mDataSource = value
```

```
                ' Update active state.
                SetActive((Not (mDataSource Is Nothing) And _
                           Not (mDataField Is Nothing)))
            End If
        End Set
    End Property
    Private mDataSource As String

    Public Property DataField() As String
        Get
            Return mDataField
        End Get
        Set(ByVal value As String)
            If value <> mDataField Then
                ' Set the property value first,
                ' in case activate fails.
            mDataField = value
            ' Update active state.
            SetActive(( Not (mDataSource Is Nothing) AndAlso _
                        Not (mDataField Is Nothing)))
            End If
        End Set
    End Property
    Private mDataField As String

    Sub SetActive(value As Boolean)
        If value <> mActive Then
            If value Then
                Activate()
                Text = mDataBase.Value(dataField)
            Else
                Deactivate()
                Text = ""
            End If
            ' Set active only if successful.
            mActive = value
        End If
    End Sub
    Private mActive As Boolean

    Sub Activate()
        ' Open database.
    End Sub

    Sub Deactivate()
        ' Close database.
    End Sub
End Class
```

C#

```
public class TextBox

    public string DataSource
    {
        get
        {
            return mDataSource;
        }
        set
        {
            if (value != mDataSource)
            {
                // Set the property value first,
                // in case activate fails.
                mDataSource = value;
```

```
                // Update active state.
                SetActive(mDataSource != null && mDataField != null);
            }
        }
    }
    string mDataSource;

    public string DataField
    {
        get
        {
            return m DataField;
        }
        set
        {
            if (value != m DataField)
            {
                // Set the property value first,
                // in case activate fails.
                mDataField = value;
                // Update active state.
                SetActive(mDataSource != null && mDataField != null);
            }
        }
    }
    string m DataField;

    void SetActive(Boolean value)
    {
        if (value != mActive)
        {
            if (value)
            {
                Activate();
                Text = mDataBase.Value(dataField);
            }
            else
            {
                Deactivate();
                Text = "";
            }
            // Set active only if successful.
            mActive = value;
        }
    }
    bool mActive;

    void Activate()
    {
        // Open database.
    }

    void Deactivate()
    {
        // Close database.
    }
}
```

In the preceding example, the following expression determines whether the object is in a state in which the data-binding feature can activate itself.

Visual Basic

```
(Not (mDataSource Is Nothing) AndAlso Not (mDataField Is Nothing))
```

C#

```
mDataSource != null && mDataField != null
```

You make activation automatic by creating a method that determines whether the object can be activated given its current state, and then activates it as necessary.

Visual Basic

```
Sub UpdateActive()
    SetActive(( Not (mDataSource Is Nothing) AndAlso _
                Not (mDataField Is Nothing)))
End Sub
```

C#

```
void UpdateActive()
{
    SetActive mDataSource != null && mDataField != null);
}
```

If you do have related properties, such as `DataSource` and `DataMember`, you should consider implementing the `ISupportInitialize` interface. This will allow the designer (or user) to call the `ISupportInitialize.BeginInit` and `ISupportInitialize.EndInit` methods when setting multiple properties to allow the component to provide optimizations. In the above example, `ISupportInitialize` could prevent unnecessary attempts to access the database until setup is correctly completed.

The expression that appears in this method indicates the parts of the object model that need to be examined in order to enforce these state transitions. In this case, the `DataSource` and `DataField` properties are affected. For more information on choosing between properties and methods, see Properties vs. Methods.

4.1.2 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.

Visual Basic

```
Class Control
    Inherits Component

    Public Property Text() As String
        Get
            Return mText
        End Get
        Set(ByVal value As String)
```

```
            If Not mText.Equals(value) Then
                mText = value
                RaiseTextChangedEvent()
            End If
        End Set
    End Property
    Private mText As String
End Class
```

C#

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

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, then 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.

Visual Basic

```
Class Edit
    Inherits Control

    Public Property Text() As String
        Get
            Return mText
        End Get
        Set(ByVal value As String)
            If Not mText.Equals(value) Then
                OnTextChanging(Event.Empty)
```

```
                mText = value
            End If
        End Set
    End Property
    Private mText As String
End Class
```

C#

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

An event is also raised after the value of the property has changed. This event cannot be cancelled. 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.

Visual Basic

```
Class Edit
    Inherits Control

    Public Property Text() As String
        Get
            Return mText
        End Get
        Set(ByVal value As String)
            If Not mText.Equals(value) Then
                OnTextChanging(Event.Empty)
                mText = value
                RaisePropertyChangedEvent(Edit.ClassInfo.text)
            End If
        End Set
    End Property
    Private mText As String

    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
```

C#

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

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

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. 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.

4.1.3 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.

1. 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.

Visual Basic

```
Public Property Name As String
    Get
        Return mName
    End Get
    Set(ByVal value As String)
        mName = value
```

```
    End Set
End Property
```

C#

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

2. 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, 2^n+1 copies of the array will be created in the following loop.

Visual Basic

```
Dim MyType As Type = ' Get a type.
Dim LoopCounter As Int32

For LoopCounter = 0 To MyType.Methods.Length - 1
    If MyType.Methods(LoopCounter).Name.Equals("text") Then
        ' Perform some operation.
    End If
Next LoopCounter
```

C#

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

```
    }
}
```

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

Visual Basic

```
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
    End Function
End Class
```

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();
}
```

4.1.4 Read-Only and Write-Only Properties

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

4.1.5 Indexed Property Usage

The following rules outline guidelines for using indexed properties:

1. Use only one indexed property per class, and make it the default-indexed property for that class.

2. Do not use nondefault-indexed properties.

3. 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.

4. Use an indexed property when the property's logical data member is an array.

5. 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.

Visual Basic

```
' 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
```

C#

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

Visual Basic

```
' 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
```

C#

```
// The MethodInfo Type.Method property is changed to
// the MethodInfo Type.GetMethod method.
MethodInfo Type.GetMethod (string name)
MethodInfo Type.GetMethod (string name, bool ignoreCase)
```

4.2 Event Usage Guidelines

The following rules outline the usage guidelines for events:

1. Choose a name for your event based on the recommended Event Naming Guidelines.

2. Do not use Hungarian notation.

3. 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."

4. In languages that support the `void` keyword, use a return type of `void` for event handlers, as shown in the following C# code example.

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

5. Event classes should extend the System.EventArgs class, as shown in the following example.

Visual Basic

```
Public Class MouseEventArgs
    Inherits EventArgs
    ' Code for the class goes here.
End Class
```

C#

```
Public Class MouseEventArgs : EventArgs
{
    // Code for the class goes here.
}
```

6. 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`.

7. 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. Unfortunately this feature is not available with Visual Basic .NET. The following code example illustrates how the `Component` class implements this space-efficient technique for storing handlers.

```
public class MyComponent : Component
{
    static readonly object EventClick = new object();
    static readonly object EventMouseDown = new object();
    static readonly object EventMouseUp = new object();
    static readonly object EventMouseMove = new object();

    public event MouseEventHandler Click
    {
        add
        {
            Events.AddHandler(EventClick, value);
        }
        remove
        {
            Events.RemoveHandler(EventClick, value);
        }
    }
    // Code for the EventMouseDown, EventMouseUp, and
    // EventMouseMove events goes here.

    // Define a private data structure to store the
    // event delegates
}
```

8. 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:

Visual Basic

```
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
```

C#

```
public class Button
{
    ButtonClickHandler onClickHandler;

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

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.

9. 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:

Visual Basic

```
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
```

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(Me, e);
    }
}
```

10. Use or extend the System.ComponentModel.CancelEventArgs class to allow the developer to control the default behaviour 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.

Visual Basic

```
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
```

C#

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

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

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.

Visual Basic

```
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
```

C#

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

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

4.3 Method Usage 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.

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

2. 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.

Visual Basic

```
Function String.IndexOf(name As String) As Int32
Function String.IndexOf(name As String, _
                        startIndex As Integer) As Int32
```

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

3. 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 behaviour is being changed.

Visual Basic

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

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

4. 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.

Visual Basic

```
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)
        With Console
            .WriteLine(a)
            .WriteLine(b)
            .WriteLine(c)
            .WriteLine()
        End With
    End Sub
End Class
```

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();
   }
}
```

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.

5. 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.

Visual Basic

```
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
```

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);
        }
    }
}
```

6. 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.

Visual Basic

```
Public Class SampleClass
    Private mTestString As String

    Public Sub New(testString As String)
        Me.mTestString = 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 Int32) _
                                      As Int32
        Return IndexOf(s, startIndex, _
                       mTestString.Length - startIndex)
    End Function

    Overloads Public Overridable Function IndexOf( _
                                      s As String, _
                                      startIndex As Int32, _
                                      count As Int32) _
                                      As Int32
        Return mTestString.IndexOf(s, startIndex, count)
    End Function
End Class
```

C#

```
public class SampleClass
{
    private string mTestString;

    public MyClass(string testString)
    {
        this. mTestString = testString;
    }

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

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

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

4.3.1 Methods With Variable Number 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.

Visual Basic

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

C#

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

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.

Visual Basic

```
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)
```

C#

```
void Format(string formatString, object arg1)
void Format(string formatString, object arg1, object arg2)

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

4.4 Constructor Usage Guidelines

The following rules outline the usage guidelines for constructors:

1. 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.

   Visual Basic

   ```
   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
   ```

   C#

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

2. 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.

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

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

5. 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:

Visual Basic

```
' 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")
```

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");
```

6. 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.

Visual Basic

```
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


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;
    }
}
```

4.5 Field Usage Guidelines

The following rules outline the usage guidelines for fields:

1. Do not use instance fields that are public, internal, protected or protected internal (Public, Friend 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 without creating a breaking change. 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.

Visual Basic

```
Public Structure Point

    Public Sub New(x As Int32, y As Int32)
        Me.mXValue = x
        Me.mYValue = y
    End Sub

    Public Property X() As Int32
        Get
            Return mXValue
        End Get
        Set(ByVal value As Int32)
            mXValue = value
        End Set
    End Property
    Private mXValue As Integer

    Public Property Y() As Int32
        Get
            Return mYValue
        End Get
        Set(ByVal value As Int32)
            mYValue = value
        End Set
    End Property
    Private mYValue As Int32
End Structure
```

C#

```
public struct Point
{
    public Point(int x, int y)
    {
        this.mXValue = x;
        this.mYValue = y;
    }

    public int X
    {
        get
        {
            return mXValue;
        }
        set
        {
            mXValue = value;
        }
    }
    private int mXValue;

    public int Y
    {
        get
        {
            return mYValue;
        }
        set
        {
            mYValue = value;
        }
```

```
    }
    private int mYValue;
}
```

2. 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.

Visual Basic

```
Public Class Control
    Inherits Component

    Protected ReadOnly Property Handle() As Int32
        Get
            Return mHandle
        End Get
    End Property
    Private mHandle As Int32
End Class
```

C#

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

3. 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.

Visual Basic

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

C#

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

4. 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.

Visual Basic

```
Class SampleClass
    Private mURL As String
    Private mDestinationURL As String
End Class
```

C#

```
Class SampleClass
{
    string mURL;
    string mDestinationURL;
}
```

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

6. 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 non-static fields. For example, applying a `g_` or `s_` prefix is incorrect.

7. 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.

Visual Basic

```
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 Int32)
        ' 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 Structure
```

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;
        }
    }
}
```

4.6 Parameter Usage Guidelines

The following rules outline the usage guidelines for parameters:

1. 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.

   Visual Basic

```
Class SampleClass
    Private maxValue As Integer = 100

    Public Property Count() As Int32
        Get
            Return mCount
        End Get
        Set(ByVal Value As Int32)
            ' Check for valid parameter.
            If Value < 0 OrElse Value >= maxValue Then
                Throw New ArgumentOutOfRangeException( _
                          "Value", _
                          Value, _
                          "Value is invalid.")
            End If
            mCount = Value
        End Set
    End Property
    Private mCount As Int32

   Public Sub SelectItem(start As Int32, [end] As Int32)
       ' 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
```

C#

```
class SampleClass
{
    int MaxValue = 100;

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

    public void Select(int start, int end)
```

```
    {
        // Check for valid parameter.
        if (start < 0)
            throw newArgumentOutOfRangeException(
                        Sys.GetString(
                          "InvalidArgument",
                          "start",
                          start.ToString()));
        // Check for valid parameter.
        if (end < 0)
            throw newArgumentOutOfRangeException(
                        Sys.GetString(
                          "InvalidArgument",
                          "end",
                          end.ToString()));
    }
}
```

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.

4.7 Type Usage Guidelines

Types are the units of encapsulation in the common language runtime. For a detailed description of the complete list of data types supported by the runtime, see the Common Type System. This section provides usage guidelines for the basic kinds of types.

4.8 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.

4.9 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:

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

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

4.9.1 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 non-virtual 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.

Visual Basic

```
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 Int32, _
                                   y As Int32, _
                                   width As Int32,
                                   height As Int32)
        SetBoundsCore(x, y, width, height, Me.specified)
    End Sub

    Overloads Public Sub SetBounds(x As Int32, _
                                   y As Int32, _
                                   width As Int32, _
                                   height As Int32, _
                                   specified As BoundsSpecified)
        SetBoundsCore(x, y, width, height, specified)
    End Sub

    Protected Overridable Sub SetBoundsCore(x As Int32, _
                                   y As Int32, _
                                   width As Int32, _
                                   height As Int32, _
                                   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
```

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;
    }
}
```

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.

4.10 Sealed Class Usage Guidelines

The following rules outline the usage guidelines for sealed classes:

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

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

Visual Basic

```
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
```

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.
    }
}
```

4.11 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.

4.12 Structure Usage Guidelines

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

1. Act like primitive types.
2. Have an instance size under 16 bytes.
3. Are immutable.
4. Value semantics are desirable.

The following example shows a correctly defined structure.

Visual Basic

```
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
```

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;
    }
}
```

When using a `structure`, 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 structures 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.

4.13 Enum Usage Guidelines

The following rules outline the usage guidelines for enumerations:

1. 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.

Visual Basic

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

C#

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

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

Visual Basic

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

C#

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

2. 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.

Visual Basic

```
<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
```

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
}
```

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.

3. 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).

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

Visual Basic

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

C#

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

5. Use an enum instead of static final constants.

6. 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.

7. Do not use a non-integral `enum` type. Use only `Byte`, `Int16`, `Int32`, or `Int64`.

8. Do not use an `Enum` suffix on `enum` types.

4.14 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.

4.14.1 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.

4.14.2 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`.

4.15 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:

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

   Visual Basic

```
<AttributeUsage(AttributeTargets.All, Inherited := False, _
 AllowMultiple := True)> _
Public Class ObsoleteAttribute
    Inherits Attribute
    ' Insert code here.
End Class
```

   C#

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

2. Seal attribute classes whenever possible, so that classes cannot be derived from them. (This improves performance.)

3. Use positional arguments for required parameters.

4. Use named arguments for optional parameters.

5. Do not name a parameter with both named and positional arguments.

6. Provide a read-only property with the same name as each positional argument, but change the case to differentiate between them.

7. Provide a read/write property with the same name as each named argument, but change the case to differentiate between them.

   Visual Basic

```
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 Int32
        Get
            Return Age
        End Get
        Set (value As Int32)
            Age = value
        End Set
    End Property
    ' Positional argument.
End Class
```

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.
}
```

4.16 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:

1. The nested type logically belongs to the containing type.

2. 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:

Visual Basic

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

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

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:

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

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

5 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 behaviour. Make sure that you understand the registry impact for making all types cocreateable.

5.1 Marshal By Reference

Marshal-by-reference objects are Remotable Objects. Object remoting applies to three kinds of types:

1. Types whose instances are copied when they are marshalled across an AppDomain boundary (on the same computer or a different computer). These types must be marked with the `Serializable` attribute.

2. Types for which the runtime creates a transparent proxy when they are marshalled across an AppDomain boundary (on the same computer or a different computer). These types must ultimately be derived from System.MarshalByRefObject class.

3. Types that are not marshalled across `AppDomains` at all. This is the default.

5.1.1 Marshal By Reference Guidelines

Follow these guidelines when using marshal by reference:

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

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

3. 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 marshalled by reference. System.IO.Stream derives from System.MarshalByRefObject. Most streams, such as `FileStreams` and `NetworkStreams`, encapsulate external resources, so they should be marshal-by-reference objects.

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

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

6 Error Raising & Handling Guidelines

The following rules outline the guidelines for raising and handling errors:

1. All code paths that result in an exception should provide a method to check for success without throwing an exception. For example, to avoid a `FileNotFoundException` you can call `File.Exists`. This might not always be possible, but the goal is that under normal execution no exceptions should be thrown.

2. Exceptions classes should always be serializable.

3. End `Exception` class names with the `Exception` suffix as in the following code example.

Visual Basic
```
<Serializable()> _
Public Class FileNotFoundException
    Inherits Exception
    ' Implementation code goes here.
End Class
```

C#
```
[Serializable()]
public class FileNotFoundException : Exception
{
    // Implementation code goes here.
}
```

4. Use the three common constructors shown in the following code example when creating exception classes in C# and the Managed Extensions for C++.

Visual Basic
```
Public Class XXXException
    Inherits ApplicationException

    Public Sub New()
        ' Implementation code goes here.
    End Sub

    Public Sub New(message As String)
        My.Base.New(message)
    End Sub

    Public Sub New(message As String, inner As Exception)
        My.Base.New(message, inner)
    End Sub
End Class
```

C#
```
public class XXXException : ApplicationException
{
    XxxException() {... }
    XxxException(string message) {... }
    XxxException(string message, Exception inner) {... }
}
```

5. In most cases, use the predefined exception types. Only define new exception types for programmatic scenarios, where you expect users of your class library to catch exceptions of this new type and perform a programmatic action based on the exception type itself. This is

in lieu of parsing the exception string, which would negatively impact performance and maintenance. For example, it makes sense to define a `FileNotFoundException` because the developer might decide to create the missing file. However, a `FileIOException` is not something that would typically be handled specifically in code.

6. Do not derive new exceptions directly from the base class Exception. Instead, derive from `ApplicationException`.

7. Group new exceptions derived from the base class `Exception` by namespace. For example, there will be derived classes for XML, IO, Collections, and so on. Each of these areas will have their own derived classes of exceptions as appropriate. Any exceptions that other library or application writers want to add will extend the `Exception` class directly. You should create a single name for all related exceptions, and extend all exceptions related to that application or library from that group.

8. Use a localized description string in every exception. When the user sees an error message, it will be derived from the description string of the exception that was thrown, and never from the exception class.

9. Create grammatically correct error messages with punctuation. Each sentence in the description string of an exception should end in a period. Code that generically displays an exception message to the user does not have to handle the case where a developer forgot the final period.

10. Provide exception properties for programmatic access. Include extra information (other than the description string) in an exception only when there is a programmatic scenario where that additional information is useful. You should rarely need to include additional information in an exception.

11. Do not use exceptions for normal or expected errors, or for normal flow of control. (Throwing an exception is an extremely costly operation.)

12. You should return null for extremely common error cases. For example, a `File.Open` command returns a null reference if the file is not found, but throws an exception if the file is locked.

13. Design classes so that in the normal course of use an exception will never be thrown. In the following code example, a `FileStream` class exposes another way of determining if the end of the file has been reached to avoid the exception that will be thrown if the developer reads past the end of the file.

Visual Basic

```
Class FileRead
    Sub Open()
        Dim stream As FileStream = _
               File.Open("myfile.txt",FileMode.Open)
        Dim b As Byte

        ' ReadByte returns -1 at end of file.
        While b = stream.ReadByte() <> true
            ' Do something.
        End While
    End Sub
End Class
```

```
C#
class FileRead
{
   void Open()
    {
        FileStream stream =
                File.Open("myfile.txt", FileMode.Open);
        byte b;

        // ReadByte returns -1 at end of file.
        while ((b = stream.ReadByte()) != true)
        {
            // Do something.
        }
    }
}
```

14. Throw the InvalidOperationException exception if a call to a property `set` accessor or method is not appropriate given the object's current state.

15. Throw an ArgumentException or create an exception derived from this class if invalid parameters are passed or detected.

16. Be aware that the stack trace starts at the point where an exception is thrown, not where it is created with the `new` operator. Consider this when deciding where to throw an exception.

17. Use the exception builder methods. It is common for a class to throw the same exception from different places in its implementation. To avoid repetitive code, use helper methods that create the exception using the `new` operator and return it. The following code example shows how to implement a helper method.

```
Visual Basic
Class File
    Private mFileName As String
    Public Function Read(bytes As Int32) As Byte()
        If Not ReadFile(handle, bytes) Then
            Throw New FileIOException()
        End If
    End Function

    Private Function NewFileIOException() As FileException
        Dim Descroption As String = _
           ' Build localized string, include fileName.
        Return New FileException(Descroption)
    End Sub
End Class
```

```
C#
class File
{
    string mFileName;
    public byte[] Read(int bytes)
    {
        if (!ReadFile(handle, bytes))
              throw NewFileIOException();
    }

    FileException NewFileIOException()
    {
        string description =
           // Build localized string, include fileName.
        return new FileException(description);
```

```
        }
    }
```

18. Throw exceptions instead of returning an error code or HRESULT.

19. Throw the most specific exception possible.

20. Create meaningful message text for exceptions, targeted at the developer.

21. Set all fields on the exception you use.

22. Use Inner exceptions (chained exceptions). However, do not catch and re-throw exceptions unless you are adding additional information and/or changing the type of the exception.

23. Do not create methods that throw NullReferenceException or IndexOutOfRangeException. (These are hard debug when the application is in a production environment as they do not give enough information about the runtime situation.)

24. Perform argument checking on protected (Family) and internal (Assembly) members. Clearly state in the documentation if the protected method does not do argument checking. Unless otherwise stated, assume that argument checking is performed. There might, however, be performance gains in not performing argument checking.

25. Clean up any side effects when throwing an exception. Callers should be able to assume that there are no side effects when an exception is thrown from a function. For example, if a `Hashtable.Insert` method throws an exception, the caller can assume that the specified item was not added to the `Hashtable`.

26. Very rarely "absorb" exceptions by having a `try…catch` block that does nothing.

27. Consider using `try…finally` without a `catch` block as a "last chance" option to restore state. This is a powerful method of ensuring that locks are unlocked or that state is returned to it's original values when an error occurs.

28. Consider that if you overload `ToString` on an exception class, ASP.NET does not use `ToString` to present the exception to the developer, hence any additional state information you dump in the `ToString` method will not be shown.

6.1 Standard Exception Types

The following table lists the standard exceptions provided by the runtime and the conditions for which you should create a derived class.

Exception type	Base type	Description	Example
Exception	Object	Base class for all exceptions.	None (use a derived class of this exception).
SystemException	Exception	Base class for all runtime-generated errors.	None (use a derived class of this exception).

IndexOutOfRangeException	SystemException	Thrown by the runtime only when an array is indexed improperly.	Indexing an array outside of its valid range: `arr[arr.Length+1]`
NullReferenceException	SystemException	Thrown by the runtime only when a null object is referenced.	`object o = null;` `o.ToString();`
InvalidOperationException	SystemException	Thrown by methods when in an invalid state.	Calling `Enumerator.GetNext()` after removing an `Item` from the underlying collection.
ArgumentException	SystemException	Base class for all argument exceptions.	None (use a derived class of this exception).
ArgumentNullException	ArgumentException	Thrown by methods that do not allow an argument to be null.	`String s = null;` `"Calculate".IndexOf(s);`
ArgumentOutOfRangeException	ArgumentException	Thrown by methods that verify that arguments are in a given range.	`String s = "string";` `s.Chars[9];`
ExternalException	SystemException	Base class for exceptions that occur or are targeted at environments outside of the runtime.	None (use a derived class of this exception).
COMException	ExternalException	Exception encapsulating COM Hresult information.	Used in COM interop.
SEHException	ExternalException	Exception encapsulating Win32 structured Exception Handling information.	Used in unmanaged code Interop.

6.2 Wrapping Exceptions

Errors that occur at the same layer as a component should throw an exception that is meaningful to target users. In the following code example, the error message is targeted at users of the `TextReader` class, attempting to read from a stream.

Visual Basic

```
Public Class TextReader
    Public Function ReadLine() As String
        Try
            ' Read a line from the stream.
        Catch e As Exception
            Throw New IOException("Could not read from stream", e)
        End Try
    End Function
```

```
End Class
```

C#

```
public class TextReader
{
    public string ReadLine()
    {
        try
        {
            // Read a line from the stream.
        }
        catch (Exception e)
        {
            throw new IOException ("Could not read from stream", e);
        }
    }
}
```

7 Array Usage Guidelines

For a general description of arrays and array usage see Arrays, and the System.Array class.

7.1 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:

1. When `Add`, `Remove`, or other methods for manipulating the collection are supported.
2. To add read-only wrappers around internal arrays.

For more information on using collections, see Grouping Data in Collections.

7.2 Using Indexed Properties in Collections

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

7.3 Array Valued Properties

You should 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, 2^n+1 copies of the array will be created in the following loop.

Visual Basic
```
Dim i As Int32
For i = 0 To obj.myObj.Count - 1
    DoSomething(obj.myObj(i))
Next i
```

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

For more information, see the Properties vs. Methods topic.

7.4 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.

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

```
    End If
End Sub
```

C#

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

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.

8 Operator Overloading Usage Guidelines

The following rules outline the guidelines for operator overloading:

1. Define operators on value types that are logical built-in language types, such as the System.Decimal structure.

2. Provide operator-overloading methods only in the class in which the methods are defined.

3. Use the names and signature conventions described in the Common Language Specification (CLS).

4. 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.

5. Overload operators in a symmetric manner. For example, if you overload the equality operator (==), you should also overload the not equal operator (!=).

6. 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) { }
}
```

The following table contains a list of operator symbols and the corresponding alternative methods and operator names.

C++ operator symbol	Name of alternative method	Name of operator
Not defined	ToXXX or FromXXX	op_Implicit
Not defined	ToXXX or FromXXX	op_Explicit
+ (binary)	Add	op_Addition
- (binary)	Subtract	op_Subtraction
* (binary)	Multiply	op_Multiply
/	Divide	op_Division
%	Mod	op_Modulus
^	Xor	op_ExclusiveOr
& (binary)	And	op_BitwiseAnd

|	Or	op_BitwiseOr
&&	AndAlso	op_LogicalAnd
||	OrElse	op_LogicalOr
=	Assign	op_Assign
<<	LeftShift	op_LeftShift
>>	RightShift	op_RightShift
Not defined	LeftShift	op_SignedRightShift
Not defined	RightShift	op_UnsignedRightShift
==	Equals	op_Equality
>	Compare	op_GreaterThan
<	Compare	op_LessThan
!=	Compare	op_Inequality
>=	Compare	op_GreaterThanOrEqual
<=	Compare	op_LessThanOrEqual
*=	Multiply	op_MultiplicationAssignment
-=	Subtract	op_SubtractionAssignment
^=	Xor	op_ExclusiveOrAssignment
<<=	LeftShift	op_LeftShiftAssignment
%=	Mod	op_ModulusAssignment
+=	Add	op_AdditionAssignment
&=	BitwiseAnd	op_BitwiseAndAssignment
|=	BitwiseOr	op_BitwiseOrAssignment
,	None assigned	op_Comma
/=	Divide	op_DivisionAssignment
–	Decrement	op_Decrement
++	Increment	op_Increment
- (unary)	Negate	op_UnaryNegation
+ (unary)	Plus	op_UnaryPlus
~	OnesComplement	op_OnesComplement

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

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

1. Implement the `GetHashCode` method whenever you implement the `Equals` method. This keeps `Equals` and `GetHashCode` synchronized.

2. 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 ==.

3. Override the `Equals` method any time you implement the IComparable interface.

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

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

For related information on the **Equals** method, see Implementing the Equals Method.

8.1.1 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.

8.1.2 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 ==.

8.1.3 Implementing the Equals Method

1. Override the `GetHashCode` method to allow a type to work correctly in a hash table.

2. Do not throw an exception in the implementation of an `Equals` method. Instead, return `false` for a null argument.

3. Follow the contract defined on the Object.Equals method as follows:

- `x.Equals(x)` returns `true`.
- `x.Equals(y)` returns the same value as `y.Equals(x)`.
- `(x.Equals(y) && y.Equals(z))` returns `true` if and only if `x.Equals(z)` returns `true`.
- Successive invocations of `x.Equals(y)` return the same value as long as the objects referenced by `x` and `y` are not modified.
- `x.Equals(null)` returns `false`.

4. For some kinds of objects, it is desirable to have `Equals` test for value equality instead of referential equality. Such implementations of `Equals` return `true` if the two objects have the same value, even if they are not the same instance. The definition of what constitutes an object's value is up to the implementer of the type, but it is typically some or all of the data stored in the instance variables of the object. For example, the value of a string is based on the characters of the string; the `Equals` method of the `String` class returns `true` for any two instances of a string that contain exactly the same characters in the same order.

5. When the `Equals` method of a base class provides value equality, an override of `Equals` in a derived class should call the inherited implementation of `Equals`.

6. If you are programming in a language that supports operator overloading, and you choose to overload the equality operator (==) for a specified type, that type should override the `Equals` method. Such implementations of the `Equals` method should return the same results as the equality operator. Following this guideline will help ensure that class library code using `Equals` (such as Hashtable and ArrayList) works in a manner that is consistent with the way the equality operator is used by application code.

7. If you are implementing a value type, you should consider overriding the `Equals` method to gain increased performance over the default implementation of the `Equals` method on System.ValueType. If you override `Equals` and the language supports operator overloading, you should overload the equality operator for your value type.

8. If you are implementing reference types, you should consider overriding the `Equals` method on a reference type if your type looks like a base type such as a Point, String, BigNumber, and so on. Most reference types should not overload the equality operator, even if they override `Equals`. However, if you are implementing a reference type that is intended to have value semantics, such as a complex number type, you should override the equality operator.

9. If you implement the IComparable Interface on a given type, you should override `Equals` on that type.

9 Guidelines for Casting Types

The following rules outline the usage guidelines for casts:

1. 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.

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

3. 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.

4. Do not generate a semantically different value. For example, it is appropriate to convert a 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.

5. 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 Double. However, it does not make sense for an Int32 to cast to a String, because they are in different domains.

10 Common Design Patterns

10.1 Implementing Finalize and Dispose to Clean Up Unmanaged Resources

Class instances often encapsulate control over resources that are not managed by the runtime, such as window handles (HWND), database connections, and so on. Therefore, you should provide both an explicit and an implicit way to free those resources. Provide implicit control by implementing the protected Finalize method on an object (destructor syntax in C# and the Managed Extensions for C++). The garbage collector calls this method at some point after there are no longer any valid references to the object.

In some cases, you might want to provide programmers using an object with the ability to explicitly release these external resources before the garbage collector frees the object. If an external resource is scarce or expensive, better performance can be achieved if the programmer explicitly releases resources when they are no longer being used. To provide explicit control, implement the Dispose method provided by the IDisposable Interface. The consumer of the object should call this method when it is done using the object. `Dispose` can be called even if other references to the object are alive.

Note that even when you provide explicit control by way of `Dispose`, you should provide implicit cleanup using the `Finalize` method. `Finalize` provides a backup to prevent resources from permanently leaking if the programmer fails to call `Dispose`.

For more information about implementing `Finalize` and `Dispose` to clean up unmanaged resources, see Programming for Garbage Collection. The following code example illustrates the basic design patter for implementing Dispose.

Visual Basic

```
' Design pattern for a base class.
Public Class Base
    Implements IDisposable

    ' Implement IDisposable.
    Public Sub Dispose()
        Dispose(True)
        GC.SuppressFinalize(Me)
     End Sub

    Protected Overloads Overridable Sub Dispose(disposing As Boolean)
        If disposing Then
            ' Free other state (managed objects).
        End If
        ' Free your own state (unmanaged objects).
        ' Set large fields to null.
    End Sub

    Protected Overrides Sub Finalize()
        ' Simply call Dispose(False).
        Dispose (False)
    End Sub
End Class

' Design pattern for a derived class.
Public Class Derived
```

```
    Inherits Base

    Protected Overloads Overrides Sub Dispose(disposing As Boolean)
        If disposing Then
            ' Release managed resources.
        End If
        ' Release unmanaged resources.
        ' Set large fields to null.
        ' Call Dispose on your base class.
        Mybase.Dispose(disposing)
    End Sub
    ' The derived class does not have a Finalize method
    ' or a Dispose method with parameters because it inherits
    ' them from the base class.
End Class
```

C#

```
// Design pattern for a base class.
public class Base: IDisposable
{
    //Implement IDisposable.
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (disposing)
        {
            // Free other state (managed objects).
        }
        // Free your own state (unmanaged objects).
        // Set large fields to null.
    }

    // Use C# destructor syntax for finalization code.
    ~Base()
    {
        // Simply call Dispose(false).
        Dispose (false);
    }

// Design pattern for a derived class.
public class Derived: Base
{
    protected override void Dispose(bool disposing)
    {
        if (disposing)
        {
            // Release managed resources.
        }
        // Release unmanaged resources.
        // Set large fields to null.
        // Call Dispose on your base class.
        base.Dispose(disposing);
    }
    // The derived class does not have a Finalize method
    // or a Dispose method with parameters because it inherits
    // them from the base class.
}
```

For a more detailed code example illustrating the design pattern for implementing `Finalize` and `Dispose`, see Implementing a Dispose Method.

10.2 Customizing a Dispose Method Name

Occasionally a domain-specific name is more appropriate than `Dispose`. For example, a file encapsulation might want to use the method name `Close`. In this case, implement `Dispose` privately and create a public `Close` method that calls `Dispose`. The following code example illustrates this pattern. You can replace `Close` with a method name appropriate to your domain.

Visual Basic

```
' Do not make this method overridable.
' A derived class should not be allowed
' to override this method.
Public Sub Close()
    ' Call the Dispose method with no parameters.
    Dispose()
End Sub
```

C#

```
// Do not make this method virtual.
// A derived class should not be allowed
// to override this method.
public void Close()
{
    // Call the Dispose method with no parameters.
    Dispose();
}
```

10.2.1 Finalize

The following rules outline the usage guidelines for the `Finalize` method:

1. Only implement `Finalize` on objects that require finalization. There are performance costs associated with `Finalize` methods.

2. If you require a `Finalize` method, you should consider implementing `IDisposable` to allow users of your class to avoid the cost of invoking the `Finalize` method.

3. Do not make the `Finalize` method more visible. It should be `protected`, not `public`.

4. An object's `Finalize` method should free any external resources that the object owns. Moreover, a `Finalize` method should release only resources that are held onto by the object. The `Finalize` method should not reference any other objects.

5. Do not directly call a `Finalize` method on an object other than the object's base class.

6. Call the `base.Finalize` method from an object's `Finalize` method.

Note

The base class's `Finalize` method is called automatically with the C# and the Managed Extensions for C++ destructor syntax.

10.2.2 Dispose

The following rules outline the usage guidelines for the `Dispose` method:

1. Implement the dispose design pattern on a type that encapsulates resources that explicitly need to be freed. Users can free external resources by calling the public `Dispose` method.

2. Implement the dispose design pattern on a base type that commonly has derived types that hold on to resources, even if the base type does not. If the base type has a close method, often this indicates the need to implement `Dispose`. In such cases, do not implement a `Finalize` method on the base type. `Finalize` should be implemented in any derived types that introduce resources that require cleanup.

3. Free any disposable resources a type owns in its `Dispose` method.

4. After `Dispose` has been called on an instance, prevent the `Finalize` method from running by calling the GC.SuppressFinalize method. The exception to this rule is the rare situation in which work must be done in `Finalize` that is not covered by `Dispose`.

5. Call the base class's `Dispose` method if it implements `IDisposable`.

6. Do not assume that `Dispose` will be called. Unmanaged resources owned by a type should also be released in a `Finalize` method in the event that `Dispose` is not called.

7. Throw an ObjectDisposedException when resources are already disposed. If you choose to reallocate resources after an object has been disposed, ensure that you call the GC.ReRegisterForFinalize method.

8. Propagate the calls to `Dispose` through the hierarchy of base types. The `Dispose` method should free all resources held by this object and any object owned by this object. For example, you can create an object like a `TextReader` that holds onto a `Stream` and an `Encoding`, both of which are created by the `TextReader` without the user's knowledge. Furthermore, both the `Stream` and the `Encoding` can acquire external resources. When you call the `Dispose` method on the `TextReader`, it should in turn call `Dispose` on the `Stream` and the `Encoding`, causing them to release their external resources.

9. You should consider not allowing an object to be usable after its `Dispose` method has been called. Recreating an object that has already been disposed is a difficult pattern to implement.

10. Allow a `Dispose` method to be called more than once without throwing an exception. The method should do nothing after the first call.

11 Callback Function Usage

Delegates, Interfaces and Events allow you to provide callback functionality. Each type has its own specific usage characteristics that make it better suited to particular situations.

11.1 Events

Use an event if the following are true:

1. A method signs up for the callback function up front, typically through separate `Add` and `Remove` methods.
2. Typically, more than one object will want notification of the event.
3. You want end users to be able to easily add a listener to the notification in the visual designer.

11.2 Delegates

Use a delegate if the following are true:

1. You want a C language style function pointer.
2. You want a single callback function.
3. You want registration to occur in the call or at construction time, not through a separate `Add` method.
4. The additional requirements of the event (separate event arguments class, virtual `On` method, etc.) would be too heavy for the particular implementation.

11.3 Interfaces

Use an interface if the callback function requires complex behaviour.

12 Time-Out Usage

Use time-outs to specify the maximum time a caller is willing to wait for completion of a method call.

Time-outs might take the form of a parameter to the method call as follows.

Visual Basic

```
server.PerformOperation(timeout)
```

C#

```
server.PerformOperation(timeout);
```

Alternately, time-outs can be used as a property on the server class as follows.

Visual Basic

```
server.Timeout = timeout
server.PerformOperation()
```

C#

```
server.Timeout = timeout;
server.PerformOperation();
```

You should favour the second approach, because the association between the operation and the time-out is clearer. The property-based approach might be better if the server class is designed to be a component used with visual designers.

Historically, time-outs have been represented by integers. Integer time-outs can be hard to use because it is not obvious what the unit of the time-out is, and it is difficult to translate units of time into the commonly used millisecond.

A better approach is to use the TimeSpan structure as the time-out type. TimeSpan solves the problems with integer time-outs mentioned above. The following code example shows how to use a time-out of type TimeSpan.

Visual Basic

```
Public Class Server
    Public Sub PerformOperation(timeout As TimeSpan)
        ' Insert code for the method here.
    End Sub
End Class

Public Class TestClass
    Dim server As New Server();
    server.PerformOperation(New TimeSpan(0,15,0))
End Class
```

C#

```
public class Server
{
    void PerformOperation(TimeSpan timeout)
    {
        // Insert code for the method here.
```

```
    }
}

public class TestClass
{
    public Server server = new Server();
    server.PerformOperation(new TimeSpan(0,15,0));
}
```

If the time-out is set to TimeSpan(0), the method should throw an exception if the operation is not immediately completed. If the time-out is TimeSpan.MaxValue, the operation should wait forever without timing out, as if there were no time-out set. A server class is not required to support either of these values, but it should throw an `InvalidArgumentException` if an unsupported time-out value is specified.

If a time-out expires and an exception is thrown, the server class should cancel the underlying operation.

If a default time-out is used, the server class should include a static `defaultTimeout` property to be used if the user does not specify one. The following code example includes a static `OperationTimeout` property of type TimeSpan that returns `defaultTimeout`.

Visual Basic

```
Class Server
    Private defaultTimeout As New TimeSpan(1000)

    Overloads Sub PerformOperation()
        Me.PerformOperation(OperationTimeout)
    End Sub

    Overloads Sub PerformOperation(timeout As TimeSpan)
        ' Insert code here.
    End Sub

    ReadOnly Property OperationTimeout() As TimeSpan
        Get
            Return defaultTimeout
        End Get
    End Property
End Class
```

C#

```
class Server
{
    TimeSpan defaultTimeout = new TimeSpan(1000);

    void PerformOperation()
    {
        this.PerformOperation(OperationTimeout);
    }

    void PerformOperation(TimeSpan timeout)
    {
        // Insert code here.
    }

    TimeSpan OperationTimeout
    {
        get
        {
            return defaultTimeout;
```

```
        }
    }
}
```

Types that are not able to resolve time-outs to the resolution of a TimeSpan should round the time-out to the nearest interval that can be accommodated. For example, a type that can only wait in one-second increments should round to the nearest second. An exception to this rule is when a value is rounded down to zero. In this case, the time-out should be rounded up to the minimum time-out possible. Rounding away from zero prevents "busy-wait" loops where a zero time-out value causes 100 percent processor utilisation.

In addition, it is recommended that you throw an exception when a time-out expires instead of returning an error code. Expiration of a time-out means that the operation could not complete successfully and therefore should be treated and handled as any other run-time error. For more information, see Error Raising and Handling Guidelines.

In the case of an asynchronous operation with a time-out, the callback function should be called and an exception thrown when the results of the operation are first accessed. This is illustrated in the following code example.

Visual Basic

```
Sub OnReceiveCompleted(ByVal sender As System.Object, _
                       ByVal asyncResult As ReceiveAsyncEventArgs)
    Dim queue As MessageQueue = CType(sender, MessageQueue)
    ' The following code will throw an exception
    ' if BeginReceive has timed out.
    Dim message As Message = _
                     queue.EndReceive(asyncResult.AsyncResult)
    Console.WriteLine(("Message: " + CStr(message.Body)))
    queue.BeginReceive(New TimeSpan(1, 0, 0))
End Sub
```

C#

```
void OnReceiveCompleted(Object sender,
                        ReceiveAsyncEventArgs asyncResult)
{
    MessageQueue queue = (MessageQueue) sender;
    // The following code will throw an exception
    // if BeginReceive has timed out.
    Message message = queue.EndReceive(asyncResult.AsyncResult);
    Console.WriteLine("Message: " + (string)message.Body);
    queue.BeginReceive(new TimeSpan(1,0,0));
}
```

For related information, see Guidelines for Asynchronous Programming.

13 Security in Class Libraries

Class library designers must understand code access security in order to write secure class libraries. When writing a class library, be aware of two security principles: protect objects with permissions, and write fully trusted code. The degree to which these principles apply will depend upon the class you are writing. Some classes, such as the System.IO.FileStream class, represent objects that need protection with permissions. The implementation of these classes is responsible for checking the permissions of callers and allowing only authorised callers to perform operations for which they have permission. The System.Security namespace contains classes that can help you perform these checks in the class libraries that you write. Class library code often is fully trusted or at least highly trusted code. Because class library code often accesses protected resources and unmanaged code, any flaws in the code represent a serious threat to the integrity of the entire security system. To minimise security threats, follow the guidelines described in this topic when writing class library code. For more information, see Writing Secure Class Libraries.

13.1 Protecting Objects with Permissions

Permissions are defined to protect specific resources. A class library that performs operations on protected resources must be responsible for enforcing this protection. Before acting on any request on a protected resource, such as deleting a file, class library code first must check that the caller (and usually all callers, by means of a stack walk) has the appropriate delete permission for the resource. If the caller has the permission, the action should be allowed to complete. If the caller does not have the permission, the action should not be allowed to complete and a security exception should be raised. Protection is typically implemented in code with either a declarative or an imperative check of the appropriate permissions.

It is important that classes protect resources, not only from direct access, but also from all possible kinds of exposure. For example, a cached file object is responsible for checking for file read permissions, even if the actual data is retrieved from a cache in memory and no actual file operation occurs. This is because the effect of handing the data to the caller is the same as if the caller had performed an actual read operation.

13.2 Fully Trusted Class Library Code

Many class libraries are implemented as fully trusted code that encapsulates platform-specific functionality as managed objects, such as COM or system APIs. Fully trusted code can expose a weakness to the security of the entire system. However, if class libraries are written correctly with respect to security, placing a heavy security burden on a relatively small set of class libraries and the core runtime security allows the larger body of managed code to acquire the security benefits of these core class libraries.

In a common class library security scenario, a fully trusted class exposes a resource that is protected by a permission; the resource is accessed by a native code API. A typical example of this type of resource is a file. The File class uses a native API to perform file operations, such as a deletion. The following steps are taken to protect the resource.

1. A caller requests the deletion of file `c:\test.txt` by calling the File.Delete method.

2. The `Delete` method creates a permission object representing the `delete c:\test.txt` permission.

3. The `File` class's code checks all callers on the stack to see if they have been granted the demanded permission; if not, a security exception is raised.

4. The `File` class asserts FullTrust in order to call native code, because its callers might not have this permission.

5. The `File` class uses a native API to perform the file delete operation.

6. The `File` class returns to its caller, and the file delete request is completed successfully.

13.3 Precautions for Highly Trusted Code

Code in a trusted class library is granted permissions that are not available to most application code. In addition, an assembly might contain classes that do not need special permissions but are granted these permissions because the assembly contains other classes that do require them. These situations can expose a security weakness to the system. Therefore, you must be take special care when writing highly or fully trusted code.

Design trusted code so that it can be called by any semi-trusted code on the system without exposing security holes. A stack walk of all callers normally protects resources. If a caller has insufficient permissions, attempted access is blocked. However, any time trusted code asserts a permission, the code takes responsibility for checking for required permissions. Normally, an assert should follow a permission check of the caller as described earlier in this topic. In addition, the number of higher permission asserts should be minimized to reduce the risk of unintended exposure.

Fully trusted code is implicitly granted all other permissions. In addition, it is allowed to violate rules of type safety and object usage. Independent of the protection of resources, any aspect of the programmatic interface that might break type safety or allow access to data not normally available to the caller can lead to a security problem.

13.4 Performance

Security checks involve checking the stack for the permissions of all callers. Depending upon the depth of the stack, these operations have the potential to be very expensive. If one operation actually consists of a number of actions at a lower level that require security checks, it might greatly improve performance to check caller permissions once and then assert the necessary permission before performing the actions. The assert will stop the stack walk from propagating further up the stack so that the check will stop there and succeed. This technique typically results in a performance improvement if three or more permission checks can be covered at once.

13.4.1 Summary of Class Security Issues

1. Any class library that uses protected resources must ensure that it does so only within the permissions of its callers.

2. Assertion of permissions should be done only when necessary, and should be preceded by the necessary permission checks.

3. To improve performance, aggregate operations that will involve security checks and consider the use of assert to limit stack walks without compromising security.

4. Be aware of how a semi-trusted malicious caller might potentially use a class to bypass security.

5. Do not assume that only callers with certain permissions will call the code.

6. Do not define non-type-safe interfaces that might be used to bypass security elsewhere.

7. Do not expose functionality in a class that allows a semi-trusted caller to take advantage of the higher trust of the class.

14 Threading Design Guidelines

The following rules outline the design guidelines for implementing threading:

1. 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.

2. Static state must be thread safe.

3. 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.

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

5. 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 synchronise their static methods, this will cause a deadlock. You might discover this deadlock only under heavy threading stress.

6. 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 synchronisation. Excessive use of fine-grained synchronisation might negatively impact performance . In addition, it might have a significant negative impact on scalability.

7. 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.

Visual Basic
```
SyncLock Me
    myField += 1
End SyncLock
```

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

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.

Visual Basic

```
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
```

C#

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

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

9. For performance, use `System.Threading.ReaderWriterLock` whenever usage warrants.

10. The `System.Threading.Interlocked` when developing synchronised code.

15 Formatting Standards

The physical layout of code is very important in determining how readable and maintainable it is. We need to come up with a common set of conventions for code layout to ensure that programs incorporating code from various sources is both maintainable and aesthetically pleasing.

These guidelines are not hard and fast rules, less so than the variable naming conventions already covered. As always, use your own judgement and remember that you are trying to create the best possible code, not slavishly follow rules.

That said, you'd better have a good reason before deviating from the standard.

15.1 White Space and Indentation

Indent four spaces at a time. Four is best for a couple of reasons. You don't want to indent too much, and I think three spaces is actually more conservative of screen real estate in the editor. The main reason that I chose four is that this is the default number.

When writing a `For` statement, the natural inclination is to indent four characters:

```
For CurrentValue = ValueMin To ValueMax
    MsgBox ...
Next CurrentValue
```

For the `Select Case` statement, there are two commonly used techniques, both of which are valid.

In the first technique, the `Case` statements are not indented at all but the code that is controlled by each statement is indented by the standard amount of four spaces, as in:

```
Select Case CurrentMonth
Case 1,3,5,7,8,10,12
    DaysInMonth = 31
Case 4,6,9,11
    DaysInMonth = 30
Case 2
    If IsLeapYear(CurrentYear) Then
        DaysInMonth = 29
    Else
        DaysInMonth = 28
    End If
Case Else
    DisplayError "Invalid Month"
End Select
```

In the second technique, the `Case` statements themselves are indented as well and the statements they control are super-indented, essentially suspending the rules if indentation:

```
Select Case CurrentMonth
    Case 1,3,5,7,8,10,12:  DaysInMonth = 31
    Case 4,6,9,11:         DaysInMonth = 30
    Case 2
        If IsLeapYear(CurrentYear) Then
            DaysInMonth = 29
        Else
```

```
            DaysInMonth = 28
        End If
    Case Else:              DisplayError "Invalid Month"
End Select
```

Notice how the colon is used to allow the statements to appear right beside the conditions. Notice also how you cannot do this for `If` statements.

Both techniques are valid and acceptable. In some sections of a program, one or the other will be clearer and more maintainable, so use common sense when deciding.

Let's not get too hung up on indentation. Most of us understand what is acceptable and what is not when it comes to indenting code. In .NET we can have out code indented for us by the environment, in this case – let it do what it feels is most appropriate.

16 Commenting Code

This one will be really controversial. Don't comment more than you need to. Now, here's a list of where you definitely need to; there may be others too.

16.1 XML Comments

Every type and member should have an XML comment. This includes VB .NET code, but because of limitations in the VS .NET 2002/2003 environment, these must be manually constructed.

16.2 In-line Comments

In-line comments are comments that appear by themselves on a line, whether they are indented or not.

In-line comments are the Post-It notes of programming. This is where you make annotations to help yourself or another programmer who needs to work with the code later. Use In-line comments to make notes in your code about:

- What you are doing.
- Where you are up to.
- Why you have chosen a particular option.
- Any external factors that need to be known.

Here are some examples of appropriate uses of In-line comments:

1. What we are doing:

```
' Now update the control totals file to keep everything in sync
```

2. Where we are up to:

```
' At this point, everything has been validated.
' If anything was invalid, we would have exited the procedure.
```

3. Why we chose a particular option:

```
' Use a sequential search for now because it's simple to code
' We can replace with a binary search later if it's not fast
' enough
' We are using a file-based approach rather than doing it all
' in memory because testing showed that the latter approach
' used too many resources under Win2000.  That's why the code
' is here rather than in XXX.VB where it belongs.
```

4. External factors that need to be kept in mind:

```
' This assumes that the INI file settings have been checked by
' the calling routine
```

Notice that we are not documenting what is self-evident from the code. Here are some examples of **inappropriate** In-line comments:

```
' Declare local variables
Dim CurrentEmployee As Int32
' Increment the array index
CurrentEmployee += 1
' Call the update routine
UpdateRecord
```

Comment what is not readily discernible from the code. Do not re-write the code in English, otherwise you will almost certainly not keep the code and the comments synchronised and *that is very dangerous.* The reverse side of the same coin is that when you are looking at someone else's code *you should totally ignore any comments that relate directly to the code statements.* In fact, do everyone a favour and remove them.

16.3 End of Line Comments

End of Line (EOL) comments are small annotations tacked on the end of a line of code. The perceptual difference between EOL comments and In-Line comments is that EOL comments are very much focused on one or a very few lines of code whereas In-Line comments refer to larger sections of code (sometimes the whole procedure).

Think of EOL comments like margin notes in a document. Their purpose is to explain why something needs to be done or why it needs to be done now. They may also be used to document a change to the code. Here are some appropriate EOL comments:

```
CurrentEmployee += 1                          ' Keep the module level
                                              ' pointer synchronised
                                              ' for external clients

mCurrentEmployee = CurrentEmployee            ' Do this first so that
UpdateProgress                                ' the meter ends at 100%

If CurrentEmployee < m CurrentEmployee Then   ' BUG FIX 1/8/2001 SCS
```

Notice that EOL comments may be continued onto additional lines if necessary as shown in the first example.

Here are some examples of inappropriate EOL comments:

```
CurrentEmployee += 1 ' Add 1 to loop counter
UpdateRecord         ' Call the update routine
```

Do you really want to write every program twice?

17 Code Reviews

Although the primary purpose for conducting code reviews throughout the development life cycle is to identify defects in the code, the reviews can also be used to enforce coding standards in a uniform manner. Adherence to a coding standard can only be feasible when followed throughout the software project from inception to completion. It is not practical, nor is it prudent, to impose a coding standard after the fact.

To this end, code reviews can provide a benefit to all systems developed by Iridium Software.

Code Reviews should be conducted under the following guidelines:

1. A section of a developers code (one or two random procedures) should be reviewed once every fortnight. The reviewer should be a development peer. The reviewer should note the issues and maybe address them at a coming company meeting.

2. A similar section of code should be reviewed once a month by a senior developer (or if no senior developer exists, two peers). Again, the reviewer should make notes of issues and bring them to the attention of the group if necessary.

3. The Architect or Analyst responsible for the current project should review general code structure on a monthly basis.

The issues that the reviewer should be looking for could be:

- Is the code implemented appropriately for the chosen language?
- Is the chosen language appropriate for the task?
- Does the code adhere, in principle, to the guidelines laid down in this document?

18 Additional Notes for VB .NET Developers

The rest of this document addresses issues relating to coding practices. We all know that there is no set of rules that can always be applied blindly and that will result in good code. Programming is not an art form but neither is it engineering. It is more like a craft: there are accepted norms and standards and a body of knowledge that is slowly being codified and formalised. Programmers become better by learning from their previous experience and by looking at good and bad code written by others. Especially by maintaining bad code.

Rather than creating a set of rules that must be slavishly followed, I have tried to create a set of principles and guidelines that will identify the issues you need to think about and where possible indicate the good, the bad and the ugly alternatives.

Ultimately, I want each programmer at Iridium Software to take responsibility for creating good code, not simply code that adheres to some rigid standard. That is a far nobler goal - one that is both more fulfilling to the programmer and more useful to the organisation.

The underlying principle is to keep it simple.

18.1 Procedure Length

There has been an urban myth in programming academia that short procedures of no more than "a page" (whatever that is) are better. Actual research has shown that this is simply not true. There have been several studies that suggest the exact opposite. For a review of these studies (and pointers to the studies themselves) see the book "Code Complete" by Steve McConnell which is a book well worth reading….three times.

To summarise, hard empirical data suggests that error rates and development costs for routines decreases as you move from small (<32 lines) routines to larger routines (up to about 200 lines). Comprehensibility of code (as measured on computer-science students) was no better for code super-modularised to routines about 10 lines long than one with no routines at all. In contrast, on the same code modularised to routines of around 25 lines, students scored 65% better.

What this means is that there is no sin in writing long routines. Let the requirements of the process determine the length of the routine. If you feel that this routine should be 200 lines long, just do it. Be careful how far you go, of course. There *is* an upper limit beyond which it is almost impossible to comprehend a routine. Studies on really **BIG** software, like IBM's OS/360 operating system, showed that the most error prone routines were those over 500 lines, with the rate being roughly proportional to length above this figure.

Of course, a procedure should do **ONE** thing. If you see an `And` or an `Or` in a procedure name, you are probably doing something wrong. Make sure that each procedure has high cohesion an low coupling, the standard aims of good structured design.

18.2 "If"

18.2.1 Write the nominal path through the code first, then write the exceptions

Write your code so that the normal path through the code is clear. Make sure that the exceptions don't obscure the normal path of execution. This is important for both maintainability and efficiency.

18.2.2 Make sure that you branch correctly on equality

A very common mistake is to use `>` instead of `>=` or vice versa.

18.2.3 Put the normal case after the If rather than after the Else

Contrary to popular thought, programmers really do not have a problem with negated conditions. Create the condition so that the `Then` clause corresponds to normal processing.

18.2.4 Follow the If with a meaningful statement

This is somewhat related to the previous point. Don't code null `Then` clauses just for the sake of avoiding a `Not`. Which one is easier to read:

```
If EOF(ThisFile) Then
    ' do nothing
Else
    ProcessRecord
End If


If Not EOF(ThisFile)
    ProcessRecord
End If
```

18.2.5 Always at least consider using the Else clause

A study of code by GM showed that only 17% of `If` statements had an `Else` clause. Later research showed that 50 to 80% should have had one. Admittedly, this was PL/1 code and 1976, but the message is ominous. Are you *absolutely* sure you don't need that `Else` clause?

18.2.6 Simplify complicated conditions with Boolean function calls

Rather than test twelve things in an `If` statement, create a function that returns `True` or `False`. If you give it a meaningful name, it can make the `If` statement very readable and significantly improve the program.

18.2.7 Don't use chains of If statements if a Select Case statement will do

The `Select Case` statement is often a better choice than a whole series of `If` statements. The one exception is when using `GetType`, which does not work with `Select Case` statements.

18.3 "Select Case"

18.3.1 Put the normal case first

This is both more readable and more efficient.

18.3.2 Order cases by frequency

Cases are evaluated in the order that they appear in the code, so if one case is going to be selected 99% of the time, put it first.

18.3.3 Keep the actions of each case simple

Code no more than a few lines for each case. If necessary, create procedures called from each case.

18.3.4 Use the Case Else only for legitimate defaults

Don't ever use `Case Else` simply to avoid coding a specific test.

18.3.5 Use Case Else to detect errors.

Unless you really do have a default, trap the `Case Else` condition and display or log an error message.

18.3.6 Exceptions to the rule

When writing any construct, the rules may be broken if the code becomes more readable and maintainable. The rule about putting the normal case first is a good example. While it is good advice in general, there are some where it would detract from the quality of the code. For example, if you were grading scores, so that less than 50 was a fail, 50 to 60 was an E and so on, then the "normal" and more frequent case would be in the 60-80 bracket, then alternating like this:

```
Select Case CurrentScore
    Case 70 To 79: sGrade = "C"

    Case 80 To 89: sGrade = "B"

    Case 60 To 69: sGrade = "D"

    Case Is < 50: sGrade = "F"

    Case 90 To 100: sGrade = "A"

    Case 50 To 59: sGrade = "E"

    Case Else: ' they cheated

End Select
```

However, the natural way to code this would be to follow the natural order of the scores:

```
Select Case CurrentScore
    Case Is < 50: sGrade = "F"

    Case Is < 60: sGrade = "E"

    Case Is < 70: sGrade = "D"

    Case Is < 80: sGrade = "C"

    Case Is < 90: sGrade = "B"

    Case Is <= 100: sGrade = "A"

    Case Else: ' they cheated
```

```
End Select
```

Not only is this easier to understand, it has the added advantage of being more robust - if the scores are later changed from integers to allow for fractional points, then the first version would allow 89.1 as an A which is probably not what was expected.

On the other hand, if this statement was identified as being the bottleneck in a program that was not performing quickly enough, then it would be quite appropriate to order the cases by their statistical probability of occurring, in which case you would document why you did it that way in comments. This discussion was included to reinforce the fact that we are not seeking to blindly obey rules - we are trying to write good code. The rules must be followed unless they result in bad code, in which case they must **not** be followed.

18.4 "Do"

18.4.1 Keep the body of a loop visible on the screen at once

If it is too long to see, chances are it is too long to understand buried inside that loop and should be taken out as a procedure.

18.4.2 Limit nesting to three levels

Studies have shown that the ability of programmers to understand a loop deteriorates significantly beyond three levels of nesting.

18.5 "For"

See Do Above

18.5.1 Never omit the loop variable from the Next statement

It is very hard to unravel loops if their end points do not identify themselves properly.

18.5.2 Try not to use *i*, *j* and *k* as the loop variables

Surely you can come up with a more meaningful name. Even if it's something generic like `LoopCounter` it is better than `i`.

18.6 "Goto"

Do not use `Goto` statements unless they make the code simpler. The general consensus of opinion is that `Goto` statements tend to make code difficult to follow but that in some cases the exact opposite is true.

You may want to use `Goto` to exit from a very complex nested control structure. Be careful here; if you really feel that a `Goto` is warranted, perhaps the control structure is just too complex and you should decompose the code into smaller routines.

That is not to say that there are no cases where the best approach is to use a `Goto`. If you really feel that it is necessary then go ahead and use one. Just make sure that you have thought it through and are convinced that it really is a good thing and not a hack.

18.7 "Exit Sub" / "Exit Function" And "Return"

Related to the use of a Goto is an `Exit Sub` (or `Exit Function`) statement. There are basically three ways to make some trailing part of the code not execute:

1. Make it part of a conditional (`If`) statement:

```
Sub DoSomething()
    If CanProceed() Then
        ...
        ...
    End If
End Sub
```

2. Jump over it with a `Goto`.

```
Sub DoSomething()
    If Not CanProceed() Then
        Goto DoSomething_Exit
    End If
    ...
    ...
DoSomething_Exit:

End Sub
```

3. Exit prematurely with an `Exit Sub/Function` or `Return`.

```
Sub DoSomething()
    If Not CanProceed() Then
        Exit Sub
        ' or even Return
    End If
    ...
    ...
End Sub
```

The one that seems to be the clearest in these simple, skeletal examples is the first one and it is in general a good approach to coding simple procedures. This structure becomes unwieldy when the main body of the procedure (denoted by the "…" above) is nested deep inside a series of control structures required to determine whether that body should be executed. It is not at all difficult to end up with the main body indented half way across the screen. If that main body is itself complex, the code looks quite messy, not to mention the fact that you then need to unwind all those nested control structures after the main body of the code.

When you find this happening in your code, adopt a different approach: determine whether you should proceed and, if not, exit prematurely. Both of the other techniques shown work. Although I think that the Exit statement is more 'elegant', I am forced to mandate the use of the `Goto ExitLabel` as the Iridium Software standard. The reason that this was chosen is that sometimes you need to clean up before exiting a procedure. Using the Goto `ExitLabel` construct means

that you can code that cleanup code just once (after the label) instead of many times (before each `Exit` statement).

If you need to prematurely exit a procedure, prefer the `Goto ExitLabel` construct to the `Exit Sub/Exit Function` or `Return` statements unless there is no chance that any cleanup will be needed before those statements.

One case where the `Exit` statement is very much OK is in conjunction with gatekeeper variables to avoid unwanted recursion. Eg:

```
Sub txtSomething_Change()
    Static IsBusy As Integer
    If IsBusy Then Exit Sub

    IsBusy = True
    ... ' some code that may re-trigger the Change() event
    IsBusy = False
Exit Sub
```

18.8 "Exit Do"

These statements prematurely bail out of the enclosing `Do` or `For` loop. Do use these when appropriate but be careful because they can make it difficult to understand the flow of execution in the program.

On the other hand, use these statements to avoid unnecessary processing. We always code for correctness and maintainability rather than efficiency, but there is no point doing totally unnecessary processing. In particular, do **NOT** do this:

```
For LoopIndex = _
  Items.GetLowerBound(LoopIndex) To Items.GetUpperBound(LoopIndex)
    If Items(LoopIndex) = SearchValue Then
        Found = True
        FoundIndex = LoopIndex
    End If
Next LoopIndex

If Found Then ...
```

This will always loop through all elements of the array, even if the item is found in the first element. Placing an `Exit For` statement inside the `If` block would improve performance with no loss of clarity in the code.

Do avoid the need to use these statements in deeply nested loops. (Indeed, avoid deeply nested loops in the first place.) Sometimes there really is no option, so this is not a hard and fast rule, but in general it is difficult to determine where an `Exit For` or `Exit Do` will branch to in a deeply nested loop.

19 Disclaimer

This document is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.