

ENLAB SOFTWARE .NET CODING STANDARDS

Abstract

This document describes coding style and standards that Enterprise Lab adopts for its all .NET projects developed. It is paramount to understand and strictly follow these styles and standards so to improve quality of work at company-wise level

Vinh Tran
vinh.tran@enterpriselab.net

Table of Contents

1. OVERVIEW

This document is directly copied from the “All In-One Code Framework Coding Standards” document with modification that suits our coding practices. The styles and standards are covered only for C# and VB.NET languages.

The goal of making and rolling out this document is to instruct all .NET developers inside Enterprise Lab to follow the same coding style and standards so that to minimize the time and effort to understand others’ codes. This will ultimately help us build a great team of developers who can work with their maximum quality and productivity.

Reference:
https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=2&cad=rja&uact=8&ved=0ahUKEwjG4Mvdk4vLAhVG2qYKHVDaDYoQFggnMAE&url=https%3A%2F%2Flwoodyiii.files.wordpress.com%2F2012%2F10%2Fall-in-one-code-framework-coding-standards.docx&usg=AFQjCNHkcdsEmhDn37dmsUh8J7QsDsWXwQ&sig2=LYThoKQ61gLA3PEchfwhFg

1.1 Terminology

Through-out this document there will be recommendations or suggestions for standards and practices. Some practices are very important and must be followed, others are guidelines that are beneficial in certain scenarios but are not applicable everywhere. In order to clearly state the intent of the standards and practices that are discussed we will use the following terminology.

Wording	Intent	Justification
☑ **Do...**	This standard or practice should be followed in all cases. If you think that your specific application is exempt, it probably isn't.	These standards are present to mitigate bugs.
☒ **Do Not...**	This standard or practice should never be applied.	
☑ **You should...**	This standard or practice should be followed in most cases.	These standards are typically stylistic and attempt to promote a consistent and clear style.
☒ **You should not...**	This standard or practice should not be followed, unless there's reasonable justification.	
☑ **You can…**	This standard or practice can be followed if you want to; it's not necessarily good or bad. There are probably implications to following the practice (dependencies, or constraints) that should be considered before adopting it.	These standards are typically stylistic, but are not ubiquitously adopted.

2. GENERAL CODING STANDARDS

These general coding standards can be applied to all languages - they provide high-level guidance to the style, formatting and structure of your source code.

2.1 Clarity and Consistency

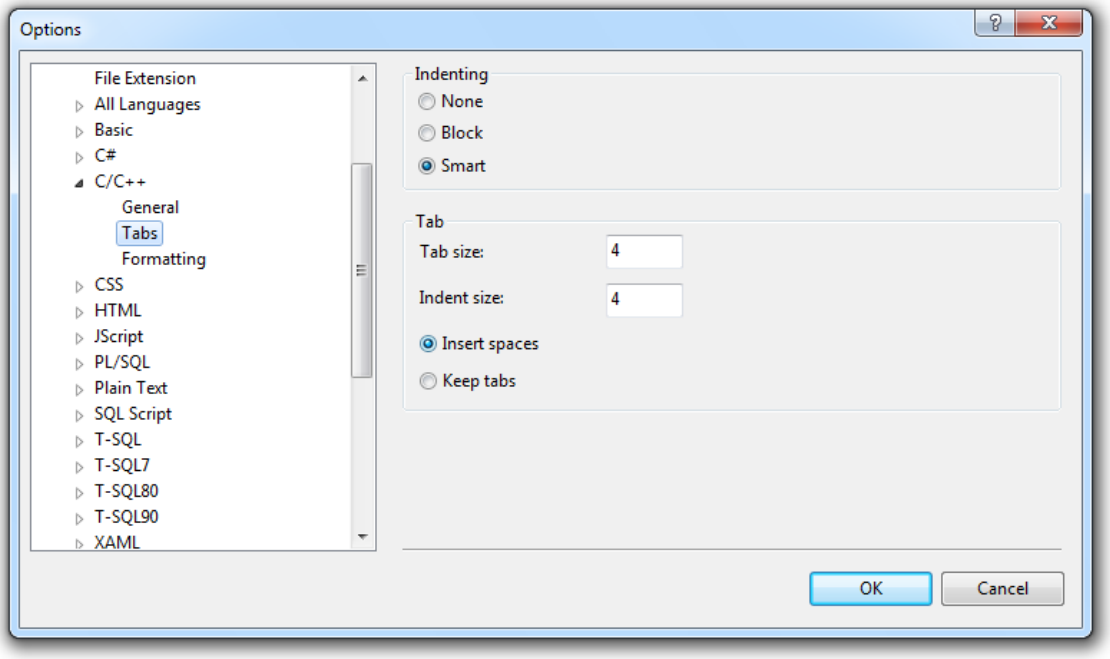
☑ **Do** ensure that clarity, readability and transparency are paramount. These coding standards strive to ensure that the resultant code is easy to understand and maintain, but nothing beats fundamentally clear, concise, self-documenting code.

☑ **Do** ensure that when applying these coding standards that they are applied consistently.

2.2 Formatting and Style

☒ **Do not** use tabs. It's generally accepted across Microsoft that tabs shouldn't be used in source files - different text editors use different spacing to render tabs, and this causes formatting confusion. All code should be written using four spaces for indentation.

Visual Studio text editor can be configured to insert spaces for tabs.



☑ **You should** limit the length of lines of code. Having overly long lines inhibits the readability of code. Break the code line when the line length is greater than around column 85 for readability.

Visual C# sample:


```
// Get and display whether the primary access token of the process belongs
// to user account that is a member of the local Administrators group even
// if it currently is not elevated (IsUserInAdminGroup).
try
{
    bool fInAdminGroup = IsUserInAdminGroup();
    this.lbInAdminGroup.Text = fInAdminGroup.ToString();
}
catch (Exception ex)
{
    this.lbInAdminGroup.Text = "N/A";
    MessageBox.Show(ex.Message, "An error occurred in IsUserInAdminGroup",
        MessageBoxButtons.OK, MessageBoxIcon.Error);
}
```

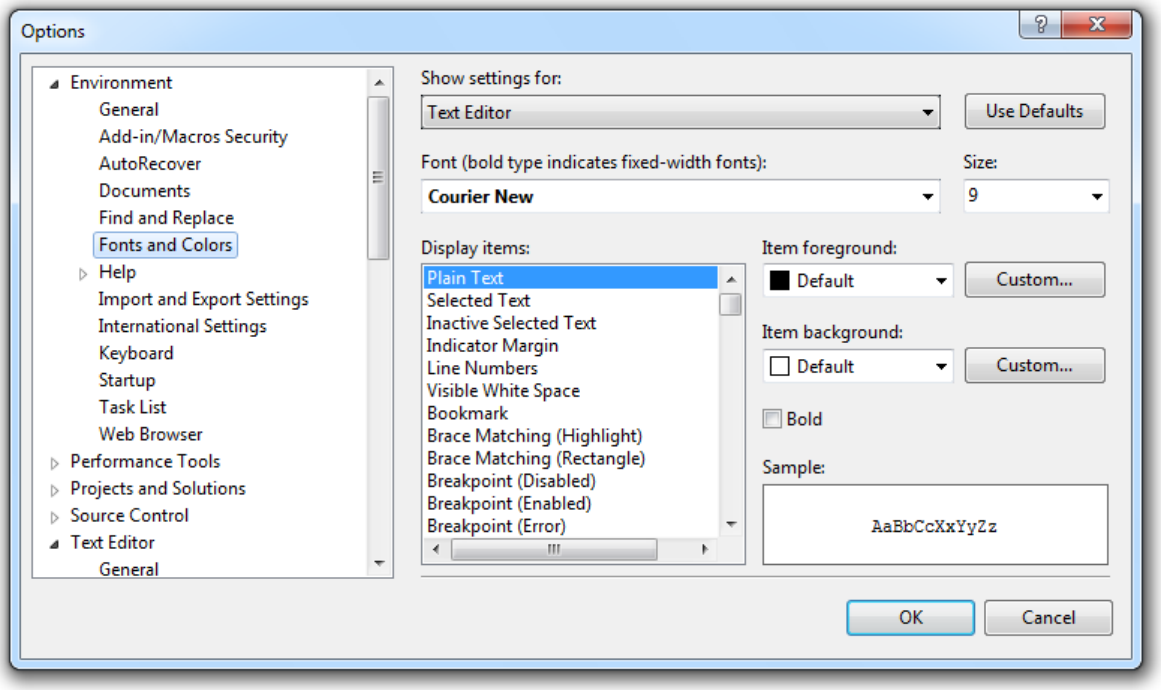
Column 86

Visual Basic sample:

```
' Get and display whether the primary access token of the process belongs
' to user account that is a member of the local Administrators group even
' if it currently is not elevated (IsUserInAdminGroup).
Try
    Dim fInAdminGroup As Boolean = Me.IsUserInAdminGroup
    Me.lbInAdminGroup.Text = fInAdminGroup.ToString
Catch ex As Exception
    Me.lbInAdminGroup.Text = "N/A"
    MessageBox.Show(ex.Message, "An error occurred in IsUserInAdminGroup",
        MessageBoxButtons.OK, MessageBoxIcon.Error)
End Try
```

Column 86

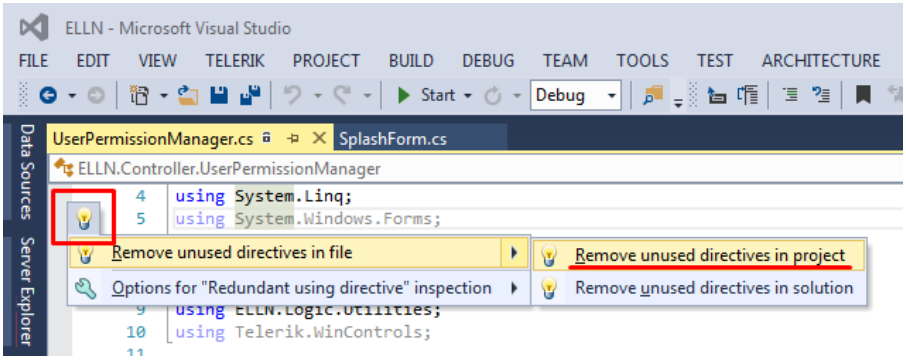
☑ **Do** use a fixed-width font, typically Courier New, in your code editor.



2.3 Using Libraries

☒ **Do not** reference unnecessary libraries, or reference unnecessary assemblies. Paying attention to small things like this can improve build times, minimize chances for mistakes, and give readers a good impression.

If you use ReSharper tool, ensure to let it does the cleanup like below



2.4 Global Variables

☑ **Do** minimize global variables. To use global variables properly, always pass them to functions through parameter values. Never reference them inside of functions or classes directly because doing so creates a side effect that alters the state of the global without the caller knowing. The same goes for static variables. If you need to modify a global variable, you should do so either as an output parameter or return a copy of the global.

Following code sample demonstrates this standard


```
class Program
{
    private static int _sharedInt;

    3 references
    public static int SharedInt...

    0 references
    static void Main(string[] args)
    {
        _sharedInt = 123;

        var myObject = new MyClass();

        // Ensure to pass the global variable to method to use instead of using it directly in the method
        myObject.UseGlobalVariables(Program.SharedInt);

        // Ensure to modify the global variable in method by passing it
        _sharedInt = myObject.CalculateGlobalVariable(Program.SharedInt);
    }
}
```

```
public class MyClass
{
    1 reference
    public void UseGlobalVariables(int sharedVariable)
    {
        // Use sharedVariable inside this block
        Console.Write(sharedVariable);

        // DO NOT: use global variable directly inside method like below
        Console.Write(Program.SharedInt);
    }

    1 reference
    public int CalculateGlobalVariable(int sharedVariable)
    {
        return (sharedVariable + 321);
    }
}
```

2.5 Variable Declarations and Initializations

☑ **Do** declare local variables in the minimum scope block that can contain them, just before use it.

```
1 reference
public void UseGlobalVariables(int sharedVariable)
{
    // Use sharedVariable inside this block
    Console.Write(sharedVariable);

    // DO NOT: use global variable directly inside method like below
    Console.Write(Program.SharedInt);

    if (sharedVariable > 0)
    {
        // Ensure to declare the variable in the block and near where you need to use it (JUST IN TIME declaration)
        var myVariable = sharedVariable * 123;
        Console.Write(myVariable);
    }
}
```


☑ **Do** declare and initialize/assign local variables on a single line where the language allows it. This reduces vertical space and makes sure that a variable does not exist in an un-initialized state or in a state that will immediately change.

```
// C# sample:
string name = myObject.Name;
int val = time.Hours;

' VB.NET sample:
Dim name As String = myObject.Name
Dim val As Integer = time.Hours
```

☒ **Do not** declare multiple variables in a single line. One declaration per line is recommended since it encourages commenting, and could avoid confusion. As a Visual C++ example,

Good:
```
CodeExample pFirst = null;
CodeExample pSecond = null;
```

Bad:
```
CodeExample pFirst, pSecond;
```

2.6 Function Declarations and Calls

The function/method name, return value and parameter list can take several forms. Ideally this can all fit on a single line. If there are many arguments that don't fit on a line those can be wrapped, many per line or one per line. Put the return type on the same line as the function/method name. For example,

```
/// <summary>
/// Demo method with many parameters
/// </summary>
/// <param name="firstInt">Should explain the intention of the parameter passing into this method if it is complecated and difficult to understand</param>
/// <param name="secondInt"></param>
/// <param name="thirdInt"></param>
/// <param name="firstString"></param>
/// <param name="secondString"></param>
/// <param name="thirdString"></param>
/// <param name="firstFloat"></param>
/// <param name="secondFloat"></param>
/// <param name="thirdFloat"></param>
/// <param name="firstBool"></param>
/// <param name="secondBool"></param>
/// <param name="errorMessage">To return error message while proccessing this method</param>
0 references
public void MethodWithManyParameters(int firstInt, int secondInt, int thirdInt, string firstString,
    string secondString, string thirdString, float firstFloat, float secondFloat, float thirdFloat,
    bool firstBool, bool secondBool, out string errorMessage)
{

}
```

When breaking up the parameter list into multiple lines, each type/parameter pair should line up under the preceding one, the first one being on a new line, indented one tab. Parameter lists for function/method *calls* should be formatted in the same manner.

☑ **Do** order parameters, grouping the in parameters first, the out parameters last. Within the group, order the parameters based on what will help programmers supply the right values. For example, if a function takes arguments named “left” and “right”, put “left” before “right” so that their place match their names. When designing a series of functions which take the same arguments, use a consistent order across the functions. For example, if one function takes an input handle as the first parameter, all of the related functions should also take the same input handle as the first parameter.

2.7 Statements

☒ **Do not** put more than one statement on a single line because it makes stepping through the code in a debugger much more difficult.

Good:

```
// C# sample:
a = 1;
b = 2;

' VB.NET sample:
If (IsAdministrator()) Then
    Console.WriteLine("YES")
End If
```

Bad:

```
// C# sample:
a = 1; b = 2;

' VB.NET sample:
If (IsAdministrator()) Then Console.WriteLine("YES")
```

2.8 Enums

☑ **Do** use an enum to strongly type parameters, properties, and return values that represent sets of values. Do use enums over static constants. An enum is a structure with a set of static constants. The reason to follow this guideline is because you will get some additional compiler and reflection support if you define an enum versus manually defining a structure with static constants.

```
// C# sample:
public enum Color
{
    Red,
    Green,
    Blue
}

' VB.NET sample:
Public Enum Color
    Red
    Green
    Blue
End Enum
```

Bad:

```
// C# sample:
public static class Color
{
    public const int Red = 0;
    public const int Green = 1;
    public const int Blue = 2;
```



```
}

' VB.NET sample:
Public Class Color
    Public Const Red As Integer = 0
    Public Const Green As Integer = 1
    Public Const Blue As Integer = 2
End Class
```

☒ **Do not** use an enum for open sets (such as the operating system version, names of your friends, etc.).

☑ **Do** provide a value of zero on simple enums. Consider calling the value something like “None.” If such value is not appropriate for this particular enum, the most common default value for the enum should be assigned the underlying value of zero.

```
// C# sample:
public enum Compression
{
    None = 0,
    GZip,
    Deflate
}

' VB.NET sample:
Public Enum Compression
    None = 0
    GZip
    Deflate
End Enum
```

☒ **Do not** use Enum.IsDefined for enum range checks in .NET. There are really two problems with Enum.IsDefined. First it loads reflection and a bunch of cold type metadata, making it a surprisingly expensive call. Second, there is a versioning issue here.

Good:

```
// C# sample:
if (c > Compression.Deflate || c < Compression.None)
{
    throw new ArgumentOutOfRangeException(...);
}

' VB.NET sample:
If (c > Compression.Deflate Or c < Compression.None) Then
    Throw New ArgumentOutOfRangeException(...);
End If
```


Bad:

```
// C# sample:
if (!Enum.IsDefined(typeof(Compression), c))
{
    throw new InvalidEnumArgumentException(...);
}

' VB.NET sample:
If Not [Enum].IsDefined(GetType(Compression), c) Then
    Throw New ArgumentOutOfRangeException(...);
End If
```

2.9 Whitespace

2.9.1 Blank Lines

☑ **You should** use blank lines to separate groups of related statements. Omit extra blank lines that do not make the code easier to read. For example, you can have a blank line between variable declarations and code.

Good:

```
// C# sample:
void ProcessItem(Item item)
{
    int counter = 0;

    if (...)
    {
    }
}
```

Bad:

```
// C# sample:
void ProcessItem(Item item)
{
    int counter = 0;


    // Implementation starts here
    //
    if(...)
    {
    }


}
```

In this example of bad usage of blank lines, there are multiple blank lines between the local variable declarations, and multiple blank likes after the ‘if’ block.

☑ **You should** use two blank lines to separate method implementations and class declarations.

2.9.2 Spaces

Spaces improve readability by decreasing code density. Here are some guidelines for the use of space characters within code:

☑ **You should** use spaces within a line as follows.

Good:

```
// C# sample:
CreateFoo();                // No space between function name and parenthesis
Method(myChar, 0, 1);       // Single space after a comma
x = array[index];           // No spaces inside brackets
while (x == y)              // Single space before flow control statements
if (x == y)                 // Single space separates operators

' VB.NET sample:
CreateFoo()                 ' No space between function name and parenthesis
Method(myChar, 0, 1)        ' Single space after a comma
x = array(index)            ' No spaces inside brackets
While (x = y)               ' Single space before flow control statements
If (x = y) Then             ' Single space separates operators
```

Bad:

```
// C++ / C# sample:
CreateFoo ();               // Space between function name and parenthesis
Method(myChar,0,1);         // No spaces after commas
CreateFoo( myChar, 0, 1 ); // Space before first arg, after last arg
x = array[ index ];         // Spaces inside brackets
while(x == y)               // No space before flow control statements
if (x==y)                   // No space separates operators

' VB.NET sample:
CreateFoo ()                ' Space between function name and parenthesis
Method(myChar,0,1)          ' No spaces after commas
CreateFoo( myChar, 0, 1 )   ' Space before first arg, after last arg
x = array( index )          ' Spaces inside brackets
While(x = y)                ' No space before flow control statements
If (x=y) Then               ' No space separates operators
```

2.10 Braces

☑ **Do** use Allman bracing style or "ANSI style".

The Allman style is named after Eric Allman. It is sometimes referred to as "ANSI style". The style puts the brace associated with a control statement on the next line, indented to the same level as the control statement. Statements within the braces are indented to the next level.

Good:

```
// C# sample:
if (x > 5)
{
    y = 0;
}

' VB.NET sample:
If (x > 5) Then
    y = 0
End If
```

Bad:

```
// C# sample:
if (x > 5) {
    y = 0;
}
```

☑ **You should** use braces around single line conditionals. Doing this makes it easier to add code to these conditionals in the future and avoids ambiguities should the tabbing of the file become disturbed.

Good:

```
// C# sample:
if (x > 5)
{
    y = 0;
}

' VB.NET sample:
If (x > 5) Then
    y = 0
End If
```

Bad:

```
// C# sample:
if (x > 5) y = 0;

' VB.NET sample:
If (x > 5) Then y = 0
```

2.11 Comments

☑ **You should** use comments that summarize what a piece of code is designed to do and why. **Do not** use comments to repeat the code.

Good:

```
// Determine whether system is running Windows Vista or later operating
// systems (major version >= 6) because they support linked tokens, but
// previous versions (major version < 6) do not.
```


Bad:

```
// The following code sets the variable i to the starting value of the
// array. Then it loops through each item in the array.
```

☑ **You should** use '//' comments instead of '/* */' for comments for C# code comments. The single-line syntax (// …) is preferred even when a comment spans multiple lines.

```
// Determine whether system is running Windows Vista or later operating
// systems (major version >= 6) because they support linked tokens, but
// previous versions (major version < 6) do not.
if (Environment.OSVersion.Version.Major >= 6)
{
}
```

```
' Get and display the process elevation information (IsProcessElevated)
' and integrity level (GetProcessIntegrityLevel). The information is not
' available on operating systems prior to Windows Vista.
If (Environment.OSVersion.Version.Major >= 6) Then
End If
```

☑ **You should** indent comments at the same level as the code they describe.

☑ **You should** use full sentences with initial caps, a terminating period and proper punctuation and spelling in comments.

Good:

```
// Intialize the components on the Windows Form.
InitializeComponent();
```

```
' Intialize the components on the Windows Form.
InitializeComponent()
```

Bad:

```
//intialize the components on the Windows Form.
InitializeComponent();
```

```
'intialize the components on the Windows Form
InitializeComponent()
```

2.11.1 Inline Code Comments

Inline comments should be included on their own line and should be indented at the same level as the code they are commenting on, with a blank line before, but none after. Comments describing a block of code should appear on a line by themselves, indented as the code they describe, with one blank line before it and one blank line after it. For example:


```
if (MAXVAL >= exampleLength)
{
    // Reprort the error.
    ReportError(GetLastError());

    // The value is out of range, we cannot continue.
    return E_INVALIDARG;
}
```

☒ **You should not** comment every line with obvious descriptions of what the code does actually hinders readability and comprehension. Single-line comments should be used when the code is doing something that might not be immediately obvious.

The following example contains many unnecessary comments:

Bad:

```
// Loop through each item in the wrinkles array
for (int i = 0; i <= nLastWrinkle; i++)
{
    Wrinkle wrinkle = apWrinkles[i];    // Get the next wrinkle
    if (wrinkle.IsNew() &&              // Process if it's a new wrinkle
        nMaxImpact < wrinkle.GetImpact()) // And it has the biggest impact
    {
        nMaxImpact = wrinkle.GetImpact(); // Save its impact for comparison
        bestWrinkle = wrinkle;        // Remember this wrinkle as well
    }
}
```

A better implementation would be:

Good:

```
// Loop through each item in the wrinkles array, find the Wrinkle with
// the largest impact that is new, and store it in 'bestWrinkle'.
for (int i = 0; i <= nLastWrinkle; i++)
{
    var wrinkle = apWrinkles[i];
    if (wrinkle.IsNew() && nMaxImpact < wrinkle.GetImpact())
    {
        nMaxImpact = wrinkle.GetImpact();
        bestWrinkle = wrinkle;
    }
}
```

☑ **You should** add comments to call out non-intuitive or behavior that is not obvious from reading the code.

2.11.2 Class Comments

☑ **You should** provide banner comments for all classes and structures that are non-trivial. The level of commenting should be appropriate based on the audience of the code.

C# and VB.NET use .NET descriptive XML Documentation comments. When you compile .NET projects with /doc the compiler will search for all XML tags in the source code and create an XML documentation file.

C# class comment template:

```
/// <summary>
/// <Class description>
/// </summary>
```

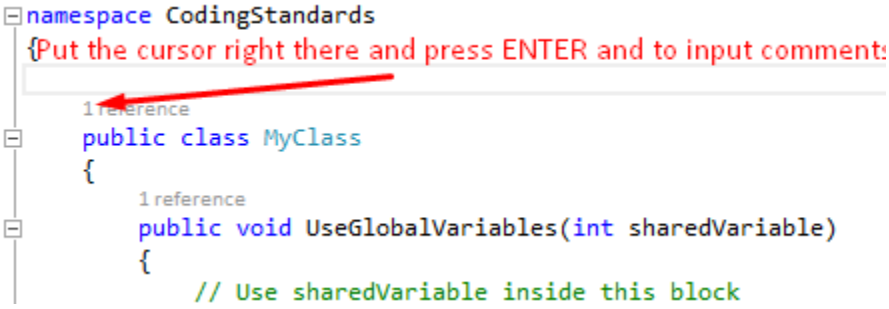
VB.NET class comment template:

```
''' <summary>
''' <Class description>
''' </summary>
```

For example,

```
/// <summary>
/// The CodeExample class represents an example of code, and tracks
/// the length and complexity of the example.
/// </summary>
public class CodeExample
{
    ...
}
```

In Visual Studio, to enter comments, for a class or method, follow this instruction:



2.11.3 Function Comments

☑ **You should** provide banner comments for all public and non-public functions that are not trivial. The level of commenting should be appropriate based on the audience of the code.

C# and VB.NET use descriptive XML Documentation comments. At least a <summary> element and also a <parameters> element and <returns> element, where applicable, are required. Methods that throw exceptions should make use of the <exception> element to indicate to consumers what exceptions may be thrown.

C# function comment template:


```
/// <summary>
/// <Function description>
/// </summary>
/// <param name="Parameter name">
/// <Parameter description>
/// </param>
/// <returns>
/// <Description of function return value>
/// </returns>
/// <exception cref="<Exception type>">
/// <Exception that may be thrown by the function>
/// </exception>
```

VB.NET function comment template:

```
''' <summary>
''' <Function description>
''' </summary>
''' <param name="Parameter name">
''' <Parameter description>
''' </param>
''' <returns>
''' <Description of function return value>
''' </returns>
''' <exception cref="<Exception type>">
''' <Exception that may be thrown by the function>
''' </exception>
```

For example,

```
/// <summary>
/// The function checks whether the primary access token of the process
/// belongs to user account that is a member of the local Administrators
/// group, even if it currently is not elevated.
/// </summary>
/// <param name="token">The handle to an access token</param>
/// <returns>
/// Returns true if the primary access token of the process belongs to
/// user account that is a member of the local Administrators group.
/// Returns false if the token does not.
/// </returns>
/// <exception cref="System.ComponentModel.Win32Exception">
/// When any native Windows API call fails, the function throws a
/// Win32Exception with the last error code.
/// </exception>
```

2.11.4 TODO Comments

☑ **You should** use TODO comments to mark unfinished/planned tasks so not to forget what to do.

2.12 Regions

☑ **Do** use region declarations where there is a large amount of code that would benefit from this organization. Grouping the large amount of code by scope or functionality improves readability and structure of the code.

C# regions:

```
#region Helper Functions for XX
...
#endregion
```

VB.NET regions:

```
#Region "Helper Functions for XX"
...
#End Region
```


3. .NET CODING STANDARDS

3.1 Files and Structure

☒ **Do not** have more than one public type (such as class, enum, etc.) in a source file, unless they differ only in the number of generic parameters or one is nested in the other. Multiple internal types in one file are allowed.

☑ **Do** name the source file with the name of the public type it contains. For example, MainForm class should be in MainForm.cs file and List<T> class should be in List.cs file.

3.2 Assembly Properties

The assembly should contain the appropriate property values describing its name, copyright, and so on.

Standard	Example
Set Copyright to Copyright © Microsoft Corporation 2016	`[assembly: AssemblyCopyright("Copyright © Microsoft Corporation 2016")]`
Set AssemblyCompany to Microsoft Corporation	`[assembly: AssemblyCompany("Microsoft Corporation")]`
Set both AssemblyTitle and AssemblyProduct to the current sample name	`[assembly: AssemblyTitle("CSNamedPipeClient")]` `[assembly: AssemblyProduct("CSNamedPipeClient")]`

3.3 Naming Conventions

3.3.1 General Naming Conventions

☑ **Do** use meaning names for various types, functions, variables, constructs and types.

☑ **Do** favor readability over brevity. For instance the property name CanScrollHorizontally is better than ScrollableX (an obscure reference to the X-axis)

☒ **Do not** use underscores, hyphens, or any other non-alphanumeric characters.

☒ **You should not** use of shortenings or contractions as parts of identifier names. For example, use “GetWindow” rather than “GetWin”. For functions of common types, thread procs, window procedures, dialog procedures use the common suffixes for these “ThreadProc”, “DialogProc”, “WndProc”.

3.3.2 Capitalization Naming Rules for Identifiers

The following table describes the capitalization and naming rules for different types of identifiers.

Identifier	Casing	Naming Structure	Example
Class, Structure	PascalCasing	Noun	`public class ComplexNumber {...}` `public struct ComplextStruct {...}`

Namespace	PascalCasing	Noun ☒ **Do not** use the same name for a namespace and a type in that namespace.	`namespace Microsoft.Sample.Windows7`

Enumeration	PascalCasing	Noun
		
		☑ **Do** name flag enums with plural nouns or noun phrases and simple enums with singular nouns or noun phrases.
			`[Flags]`
			`public enum ConsoleModifiers`
			`{ Alt, Control }`

Method	PascalCasing	Verb or Verb phrase	`public void Print() {...}` `public void ProcessItem() {...}`

Public Property	PascalCasing	Noun or Adjective
		
		☑ **Do** name collection proprieties with a plural phrase describing the items in the collection, as opposed to a singular phrase followed by “List” or “Collection”.
		
		☑ **Do** name Boolean proprieties with an affirmative phrase (CanSeek instead of CantSeek). Optionally, you can also prefix Boolean properties with “Is,” “Can,” or “Has” but only where it adds value.
			`public string CustomerName`
			`public ItemCollection Items`
			`public bool CanRead`

Non-public Field	_camelCasing	Noun or Adjective. ☑ **Do** be consistent in a code sample when you use the '_' prefix.	`private string _name;`

Event	PascalCasing	Verb or Verb phrase
		
		☑ **Do** give events names with a concept of before and after, using the present and past tense.
		
		☒ **Do not** use “Before” or “After” prefixes or postfixes to indicate pre and post events.
			`// A close event that is raised after the window is closed.`
			`public event WindowClosed`
			
			`// A close event that is raised before a window is closed.`

			`public event WindowClosing`
Delegate	PascalCasing	☑ **Do** add the suffix 'EventHandler' to names of delegates that are used in events. ☑ **Do** add the suffix 'Callback' to names of delegates other than those used as event handlers. ☒ **Do not** add the suffix "Delegate" to a delegate.	`public delegate WindowClosedEventHandler`
Interface	PascalCasing 'I' prefix	Noun	`public interface IDictionary`

Constant	PascalCasing for publicly visible;
	
	camelCasing for internally visible;
	
	All capital only for abbreviation of one or two chars long.
		Noun	`public const string MessageText = "A";`
			
			`private const string messageText = "B";`
			
			`public const double PI = 3.14159...;`

Parameter, Variable	camelCasing	Noun	`int customerID;`
Generic Type Parameter	PascalCasing 'T' prefix	Noun ☑ **Do** name generic type parameters with descriptive names, unless a single-letter name is completely self-explanatory and a descriptive name would not add value. ☑ **Do** prefix descriptive type parameter names with T. ☑ **You should** using T as the type parameter name for types	`T, TItem, TPolicy`

		with one single-letter type parameter.	
Resource	PascalCasing	Noun ☑ **Do** provide descriptive rather than short identifiers. Keep them concise where possible, but do not sacrifice readability for space. ☑ **Do** use only alphanumeric characters and underscores in naming resources.	`ArgumentExceptionInvalidName`

3.3.3 Hungarian Notation

☒ **Do not** encode the type of a variable in its name such as strQuery, intNumberOfRecords, etc. (Hungarian notation) in .NET.

3.3.4 UI Control Naming Conventions

UI controls would use the following prefixes. The primary purpose was to make code more readable.

Control Type	Prefix	Control Type	Prefix
Button	btn	NotificationIcon	nfy
CheckBox	chk	Panel	pnl
CheckedListBox	lst	PictureBox	pct
ComboBox	cmb	ProgressBar	prg
ContextMenu	mnu	RadioButton	rad
DataGrid	dg	Splitter	spl
DateTimePicker	dtp	StatusBar	sts
Form	suffix: XXXForm	TabControl	tab
GroupBox	grp	TabPage	tab
ImageList	iml	TextBox	tb
Label	lb	Timer	tmr
ListBox	lst	TreeView	tvw
ListView	lvw	NotificationIcon	nfy
Menu	mnu	Panel	pnl
MenuItem	mnu		
NotificationIcon	nfy		

For example, for the “File | Save” menu option, the “Save” MenuItem would be called “mnuFileSave”.

3.4 Constants

☑ **Do** use constant fields for constants that will never change. The compiler burns the values of const fields directly into calling code. Therefore const values can never be changed without the risk of breaking compatibility.

```
public class Int32
{
    public const int MaxValue = 0x7fffffff;
    public const int MinValue = unchecked((int)0x80000000);
}

Public Class Int32
    Public Const MaxValue As Integer = &H7FFFFFFF
    Public Const MinValue As Integer = &H80000000
End Class
```

☑ **Do** use public static (shared) readonly fields for predefined object instances. If there are predefined instances of the type, declare them as public readonly static fields of the type itself. For example,

```
public class ShellFolder
{
    public static readonly ShellFolder ProgramData = new
ShellFolder("ProgramData");
    public static readonly ShellFolder ProgramFiles = new
ShellFolder("ProgramData");
    ...
}

Public Class ShellFolder
    Public Shared ReadOnly ProgramData As New ShellFolder("ProgramData")
    Public Shared ReadOnly ProgramFiles As New ShellFolder("ProgramFiles")
    ...
End Class
```

3.5 Strings

☒ **Do not** use the '+' operator (or '&' in VB.NET) to concatenate many strings (big string). Instead, you should use StringBuilder for concatenation as it provides better performance. However, **do** use the '+' operator (or '&' in VB.NET) to concatenate small numbers of strings.

Good:

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 10; i++)
{
   sb.Append(i.ToString());
}
```

Bad:


```
string str = string.Empty;
for (int i = 0; i < 10; i++)
{
   str += i.ToString();
}
```

☑ **Do** use StringComparison.Ordinal or StringComparison.OrdinalIgnoreCase for comparisons as your safe default for culture-agnostic string matching, and for better performance.

☑ **Do** use an overload of the String.Equals method to test whether two strings are equal. For example, to test if two strings are equal ignoring the case,

```
if (str1.Equals(str2, StringComparison.OrdinalIgnoreCase))

If (str1.Equals(str2, StringComparison.OrdinalIgnoreCase)) Then
```

☒ **Do not** use an overload of the String.Compare or CompareTo method and test for a return value of zero to determine whether two strings are equal. They are used to sort strings, not to check for equality.

☑ **Do** use the String.ToUpperInvariant method instead of the String.ToLowerInvariant method when you normalize strings for comparison as the former is more reliable.

3.6 Arrays and Collections

☑ **You should** use arrays in low-level functions to minimize memory consumption and maximize performance. In public interfaces, do prefer collections over arrays.

Collections provide more control over contents, can evolve over time, and are more usable. In addition, using arrays for read-only scenarios is discouraged as the cost of cloning the array is prohibitive.

However, if you are targeting more skilled developers and usability is less of a concern, it might be better to use arrays for read-write scenarios. Arrays have a smaller memory footprint, which helps reduce the working set, and access to elements in an array is faster as it is optimized by the runtime.

☒ **Do not** use read-only array fields. The field itself is read-only and can't be changed, but elements in the array can be changed. This example demonstrates the pitfalls of using read-only array fields:

Bad:

```
public static readonly char[] InvalidPathChars = { '\"', '<', '>', '|'};
```


This allows callers to change the values in the array as follows:

```
InvalidPathChars[0] = 'A';
```

Instead, you can use either a read-only collection (only if the items are immutable) or clone the array before returning it. However, the cost of cloning the array may be prohibitive.

```
public static ReadOnlyCollection<char> GetInvalidPathChars()
{
    return Array.AsReadOnly(badChars);
}

public static char[] GetInvalidPathChars()
{
    return (char[])badChars.Clone();
}
```

☑ **You should** use jagged arrays instead of multidimensional arrays. A jagged array is an array with elements that are also arrays. The arrays that make up the elements can be of different sizes, leading to less wasted space for some sets of data (e.g., sparse matrix), as compared to multidimensional arrays. Furthermore, the CLR optimizes index operations on jagged arrays, so they might exhibit better runtime performance in some scenarios.

```
// Jagged arrays
int[][] jaggedArray =
{
    new int[] {1, 2, 3, 4},
    new int[] {5, 6, 7},
    new int[] {8},
    new int[] {9}
};

Dim jaggedArray As Integer()() = New Integer()() _
{ _
    New Integer() {1, 2, 3, 4}, _
    New Integer() {5, 6, 7}, _
    New Integer() {8}, _
    New Integer() {9} _
}

// Multidimensional arrays
int [,] multiDimArray =
{
    {1, 2, 3, 4},
    {5, 6, 7, 0},
    {8, 0, 0, 0},
    {9, 0, 0, 0}
```



```
};

Dim multiDimArray(,) As Integer = _
{ _
    {1, 2, 3, 4}, _
    {5, 6, 7, 0}, _
    {8, 0, 0, 0}, _
    {9, 0, 0, 0} _
}
```

☑ **Do** use Collection<T> or a subclass of Collection<T> for properties or return values representing read/write collections, and use ReadOnlyCollection<T> or a subclass of ReadOnlyCollection<T> for properties or return values representing read-only collections.

☑ **You should** reconsider the use of ArrayList because any objects added into the ArrayList are added as System.Object and when retrieving values back from the arraylist, these objects are to be unboxed to return the actual value type. So it is recommended to use the custom typed collections instead of ArrayList. For example, .NET provides a strongly typed collection class for String in System.Collection.Specialized, namely StringCollection.

☑ **You should** reconsider the use of Hashtable. Instead, try other dictionary such as StringDictionary, NameValueCollection, HybridCollection. Hashtable can be used if less number of values is stored.

☑ When you are creating a collection type, **you should** implement IEnumerable so that the collection can be used with LINQ to Objects.

☒ **Do not** return a null reference for Array or Collection but an empty array or collection. Null can be difficult to understand in this context. For example, a user might assume that the following code will work. Return an empty array or collection instead of a null reference.

```
int[] arr = SomeOtherFunc();
// if SomeOtherFunc() return null reference, foreach must be put inside a not-
null if statement and it's not preferred
foreach (int v in arr)
{
    ...
}
```

3.7 Classes

☑ **Do** use inheritance to express "is a" relationships such as "cat is an animal".

☑ **Do** use interfaces such as IDisposable to express “can do” relationships such as using “objects of this class can be disposed”.

3.7.1 Fields

☒ **Do not** provide instance fields that are public or protected. Public and protected fields do not version well and are not protected by code access security demands. Instead of using publicly visible fields, use private fields and expose them through properties.

☑ **Do** use public static read-only fields for predefined object instances.

☑ **Do** use constant fields for constants that will never change.

☒ **Do not** assign instances of mutable types to read-only fields.

3.7.2 Properties

☑ **Do** create read-only properties if the caller should not be able to change the value of the property.

☒ **Do not** provide set-only properties. If the property getter cannot be provided, use a method to implement the functionality instead. The method name should begin with Set followed by what would have been the property name.

☑ **Do** provide sensible default values for all properties, ensuring that the defaults do not result in a security hole or an extremely inefficient design.

☒ **You should not** throw exceptions from property getters. Property getters should be simple operations without any preconditions. If a getter might throw an exception, consider redesigning the property to be a method. This recommendation does not apply to indexers. Indexers can throw exceptions because of invalid arguments. It is valid and acceptable to throw exceptions from a property setter.

3.7.3 Constructors

☑ **Do** minimal work in the constructor. Constructors should not do much work other than to capture the constructor parameters and set main properties. The cost of any other processing should be delayed until required.

☑ **Do** throw exceptions from instance constructors if appropriate.

☑ **Do** explicitly declare the public default constructor in classes, if such a constructor is required. Even though some compilers automatically add a default constructor to your class, adding it explicitly makes code maintenance easier. It

also ensures the default constructor remains defined even if the compiler stops emitting it because you add a constructor that takes parameters.

3.7.4 Methods

☑ **Do** place all out parameters after all of the pass-by-value and ref parameters (excluding parameter arrays), even if this results in an inconsistency in parameter ordering between overloads.

☑ **Do** validate arguments passed to public, protected, or explicitly implemented members. Throw System.ArgumentException, or one of its subclasses, if the validation fails: If a null argument is passed and the member does not support null arguments, throw ArgumentNullException. If the value of an argument is outside the allowable range of values as defined by the invoked method, throw ArgumentOutOfRangeException.

3.7.5 Events

☑ **Do** be prepared for arbitrary code executing in the event-handling method. Consider placing the code where the event is raised in a try-catch block to prevent program termination due to unhandled exceptions thrown from the event handlers.

☒ **Do not** use events in performance sensitive APIs. While events are easier for many developers to understand and use, they are less desirable than Virtual Members from a performance and memory consumption perspective.

3.7.6 Member Overloading

☑ **Do** use member overloading rather than defining members with default arguments. Default arguments are not CLS-compliant (Common Language Specification) and cannot be used from some languages. There is also a versioning issue in members with default arguments. Imagine version 1 of a method that sets an optional parameter to 123. When compiling code that calls this method without specifying the optional parameter, the compiler will embed the default value (123) into the code at the call site. Now, if version 2 of the method changes the optional parameter to 863, then, if the calling code is not recompiled, it will call version 2 of the method passing in 123 (version 1's default, not version 2's default).

Good:

```
Public Overloads Sub Rotate(ByVal data As Matrix)
    Rotate(data, 180)
End Sub

Public Overloads Sub Rotate(ByVal data As Matrix, ByVal degrees As Integer)
    ' Do rotation here
End Sub
```

Bad:

```
Public Sub Rotate(ByVal data As Matrix, Optional ByVal degrees As Integer =
180)
    ' Do rotation here
End Sub
```


☒ **Do not** arbitrarily vary parameter names in overloads. If a parameter in one overload represents the same input as a parameter in another overload, the parameters should have the same name. Parameters with the same name should appear in the same position in all overloads.

☑ **Do** make only the longest overload virtual (if extensibility is required). Shorter overloads should simply call through to a longer overload.

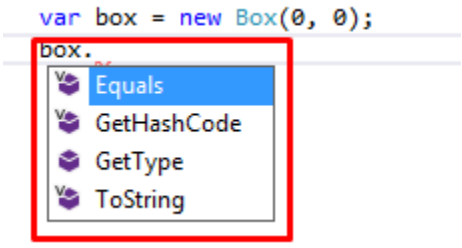
3.7.7 Interface Members

☒ **You should not** implement interface members explicitly without having a strong reason to do so. Explicitly implemented members can be confusing to developers because they don't appear in the list of public members and they can also cause unnecessary boxing of value types.

Bad

```
namespace CodingStandards
{
    3 references
    interface IDimensions
    {
        1 reference
        float Length();
        1 reference
        float Width();
    }

    1 reference
    class Box : IDimensions
    {
        readonly float _lengthInches;
        readonly float _widthInches;

        0 references
        public Box(float length, float width)
        {
            _lengthInches = length;
            _widthInches = width;
        }

        /// <summary>
        /// Explicit interface member implementation:
        /// </summary>
        /// <returns></returns>
        1 reference
        float IDimensions.Length()
        {
            return _lengthInches;
        }

        /// <summary>
        /// Explicit interface member implementation:
        /// </summary>
        /// <returns></returns>
        1 reference
        float IDimensions.Width()
        {
            return _widthInches;
        }

    }
}
```


It is bad because implemented methods can't be called through object initialization like below



☑ **You should** implement interface members explicitly, if the members are intended to be called only through the interface.

3.7.8 Virtual Members

Virtual members perform better than callbacks and events, but do not perform better than non-virtual methods.

☒ **Do not** make members virtual unless you have a good reason to do so and you are aware of all the costs related to designing, testing, and maintaining virtual members.

☑ **You should** prefer protected accessibility over public accessibility for virtual members. Public members should provide extensibility (if required) by calling into a protected virtual member.

3.7.9 Static Classes

☑ **Do** use static classes sparingly. Static classes should be used only as supporting classes for the object-oriented core of the framework.

3.7.10 Abstract Classes

☒ **Do not** define public or protected-internal constructors in abstract types.

☑ **Do** define a protected or an internal constructor on abstract classes.

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

```
public abstract class Claim
{
    protected Claim()
    {
        ...
```



```
        }
    }
```

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

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

3.8 Errors and Exceptions

3.8.1 Exception Throwing

☑ **Do** report execution failures by throwing exceptions. Exceptions are the primary means of reporting errors in frameworks. If a member cannot successfully do what it is designed to do, it should be considered an execution failure and an exception should be thrown. **Do not** return error codes.

☑ **Do** throw the most specific (the most derived) exception that makes sense. For example, throw ArgumentNullException and not its base type ArgumentException if a null argument is passed. Throwing System.Exception as well as catching System.Exception are nearly always the wrong thing to do.

☒ **Do not** use exceptions for the normal flow of control, if possible. Except for system failures and operations with potential race conditions, you should write code that does not throw exceptions. For example, you can check preconditions before calling a method that may fail and throw exceptions. For example,

```
// C# sample:
if (collection != null && !collection.IsReadOnly)
{
    collection.Add(additionalNumber);
}

' VB.NET sample:
If ((Not collection Is Nothing) And (Not collection.IsReadOnly)) Then
    collection.Add(additionalNumber)
End If
```

3.8.2 Exception Handling

☒ **You should not** swallow errors by catching nonspecific exceptions, such as System.Exception, System.SystemException, and so on in .NET code. Do catch only specific errors that the code knows how to handle.

You should catch a more specific exception, or re-throw the general exception as the last statement in the catch block. There are cases when swallowing errors in applications is acceptable, but such cases are rare.

Good:

```
// C# sample:
try
{
    ...
}
catch(System.NullReferenceException exc)
{
    ...
}
catch(System.ArgumentOutOfRangeException exc)
{
    ...
}
catch(System.InvalidCastException exc)
{
    ...
}

' VB.NET sample:
Try
    ...
Catch exc As System.NullReferenceException
    ...
Catch exc As System.ArgumentOutOfRangeException
    ...
Catch exc As System.InvalidCastException
    ...
End Try
```

Bad:

```
// C# sample:
try
{
    ...
}
catch (Exception ex)
{
    ...
}

' VB.NET sample:
Try
    ...
Catch ex As Exception
    ...
```



```
End Try
```

☑ **Do** prefer using an empty throw when catching and re-throwing an exception. This is the best way to preserve the exception call stack.

Good:

```
// C# sample:
try
{
    ... // Do some reading with the file
}
catch
{
    file.Position = position; // Unwind on failure
    throw; // Rethrow
}

' VB.NET sample:
Try
    ... ' Do some reading with the file
Catch ex As Exception
    file.Position = position ' Unwind on failure
    Throw ' Rethrow
End Try
```

Bad:

```
// C# sample:
try
{
    ... // Do some reading with the file
}
catch (Exception ex)
{
    file.Position = position; // Unwind on failure
    throw ex; // Rethrow
}

' VB.NET sample:
Try
    ... ' Do some reading with the file
Catch ex As Exception
    file.Position = position ' Unwind on failure
    Throw ex ' Rethrow
End Try
```

3.9 Resource Cleanup

☒ **Do not** force garbage collections with GC.Collect because it may creates performance issues.

3.9.1 Try-finally Block

☑ **Do** use try-finally blocks for cleanup code and try-catch blocks for error recovery code. **Do not** use catch blocks for cleanup code. Usually, the cleanup logic rolls back resource (particularly, native resource) allocations. For example,

```
// C# sample:
FileStream stream = null;
try
{
    stream = new FileStream(...);
    ...
}
finally
{
    if (stream != null)
    {
        stream.Close();
    }
}

' VB.NET sample:
Dim stream As FileStream = Nothing
Try
    stream = New FileStream(...)
    ...
Catch ex As Exception
    If (stream IsNot Nothing) Then
        stream.Close()
    End If
End Try
```

C# and VB.NET provide the using statement that can be used instead of plain try-finally to clean up objects implementing the IDisposable interface.

```
// C# sample:
using (FileStream stream = new FileStream(...))
{
    ...
}

' VB.NET sample:
Using stream As New FileStream(...)
    ...
End Using
```

Many language constructs emit try-finally blocks automatically for you. Examples are C#/VB's using statement, C#'s lock statement, VB's SyncLock statement, C#'s foreach statement, and VB's For Each statement.

3.9.2 Basic Dispose Pattern

The basic implementation of the pattern involves implementing the System.IDisposable interface and declaring the Dispose(bool) method that implements all resource cleanup logic to be shared between the Dispose method and the optional finalizer. Please note that this section does not discuss providing a finalizer. Finalizable types are extensions to this basic pattern and are discussed in the next section. The following example shows a simple implementation of the basic pattern:

```
// C# sample:
public class DisposableResourceHolder : IDisposable
{
    private bool disposed = false;
    private SafeHandle resource; // Handle to a resource

    public DisposableResourceHolder()
    {
        this.resource = ... // Allocates the native resource
    }

    public void DoSomething()
    {
        if (disposed)
        {
            throw new ObjectDisposedException(...);
        }

        // Now call some native methods using the resource
        ...
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        // Protect from being called multiple times.
        if (disposed)
        {
            return;
        }

        if (disposing)
        {
            // Clean up all managed resources.
            if (resource != null)
            {
                resource.Dispose();
```



```
            }
        }

        disposed = true;
    }
}

' VB.NET sample:
Public Class DisposableResourceHolder
    Implements IDisposable

    Private disposed As Boolean = False
    Private resource As SafeHandle ' Handle to a resource

    Public Sub New()
        resource = ... ' Allocates the native resource
    End Sub

    Public Sub DoSomething()
        If (disposed) Then
            Throw New ObjectDisposedException(...)
        End If

        ' Now call some native methods using the resource
        ...
    End Sub

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

    Protected Overridable Sub Dispose(ByVal disposing As Boolean)
        ' Protect from being called multiple times.
        If disposed Then
            Return
        End If

        If disposing Then
            ' Clean up all managed resources.
            If (resource IsNot Nothing) Then
                resource.Dispose()
            End If
        End If

        disposed = True
    End Sub

End Class
```


☑ **Do** implement the Basic Dispose Pattern on types containing instances of disposable types.

☑ **Do** extend the Basic Dispose Pattern to provide a finalizer on types holding resources that need to be freed explicitly and that do not have finalizers. For example, the pattern should be implemented on types storing unmanaged memory buffers.

☑ **You should** implement the Basic Dispose Pattern on classes that themselves don't hold unmanaged resources or disposable objects but are likely to have subtypes that do. A great example of this is the System.IO.Stream class. Although it is an abstract base class that doesn't hold any resources, most of its subclasses do and because of this, it implements this pattern.

☑ **Do** declare a protected virtual void Dispose(bool disposing) method to centralize all logic related to releasing unmanaged resources. All resource cleanup should occur in this method. The method is called from both the finalizer and the IDisposable.Dispose method. The parameter will be false if being invoked from inside a finalizer. It should be used to ensure any code running during finalization is not accessing other finalizable objects. Details of implementing finalizers are described in the next section.

```
// C# sample:
protected virtual void Dispose(bool disposing)
{
      // Protect from being called multiple times.
      if (disposed)
      {
          return;
      }

      if (disposing)
      {
          // Clean up all managed resources.
          if (resource != null)
          {
              resource.Dispose();
          }
      }

      disposed = true;
}

' VB.NET sample:
Protected Overridable Sub Dispose(ByVal disposing As Boolean)
    ' Protect from being called multiple times.
    If disposed Then
        Return
    End If
```



```
        If disposing Then
            ' Clean up all managed resources.
            If (resource IsNot Nothing) Then
                resource.Dispose()
            End If
        End If

        disposed = True
    End Sub
```

☑ **Do** implement the IDisposable interface by simply calling Dispose(true) followed by GC.SuppressFinalize(this). The call to SuppressFinalize should only occur if Dispose(true) executes successfully.

```
// C# sample:
public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}

' VB.NET sample:
Public Sub Dispose() Implements IDisposable.Dispose
    Dispose(True)
    GC.SuppressFinalize(Me)
End Sub
```

☒ **Do not** make the parameterless Dispose method virtual. The Dispose(bool) method is the one that should be overridden by subclasses.

☒ **You should not** throw an exception from within Dispose(bool) except under critical situations where the containing process has been corrupted (leaks, inconsistent shared state, etc.). Users expect that a call to Dispose would not raise an exception. For example, consider the manual try-finally in this C# snippet:

```
TextReader tr = new StreamReader(File.OpenRead("foo.txt"));
try
{
    // Do some stuff
}
finally
{
    tr.Dispose();
    // More stuff
}
```

If Dispose could raise an exception, further finally block cleanup logic will not execute. To work around this, the user would need to wrap every call to Dispose (within their finally block!) in a try block, which leads to very complex cleanup handlers.

☑ **Do** throw an ObjectDisposedException from any member that cannot be used after the object has been disposed.

```
// C# sample:
public class DisposableResourceHolder : IDisposable
{
    private bool disposed = false;
    private SafeHandle resource; // Handle to a resource

    public void DoSomething()
    {
        if (disposed)
        {
            throw new ObjectDisposedException(...);
        }

        // Now call some native methods using the resource
        ...
    }

    protected virtual void Dispose(bool disposing)
    {
        if (disposed)
        {
            return;
        }

        // Cleanup
        ...

        disposed = true;
    }
}

' VB.NET sample:
Public Class DisposableResourceHolder
    Implements IDisposable

    Private disposed As Boolean = False
    Private resource As SafeHandle ' Handle to a resource

    Public Sub DoSomething()
        If (disposed) Then
            Throw New ObjectDisposedException(...)
        End If

        ' Now call some native methods using the resource
        ...
    End Sub
```



```
    Protected Overridable Sub Dispose(ByVal disposing As Boolean)
        ' Protect from being called multiple times.
        If disposed Then
            Return
        End If

        ' Cleanup
        ...

        disposed = True
    End Sub

End Class
```

3.9.3 Finalizable Types

Finalizable types are types that extend the Basic Dispose Pattern by overriding the finalizer and providing finalization code path in the Dispose(bool) method. The following code shows an example of a finalizable type:

```
// C# sample:
public class ComplexResourceHolder : IDisposable
{
    bool disposed = false;
    private IntPtr buffer; // Unmanaged memory buffer
    private SafeHandle resource; // Disposable handle to a resource

    public ComplexResourceHolder()
    {
        this.buffer = ... // Allocates memory
        this.resource = ... // Allocates the resource
    }

    public void DoSomething()
    {
        if (disposed)
        {
            throw new ObjectDisposedException(...);
        }

        // Now call some native methods using the resource
        ...
    }

    ~ComplexResourceHolder()
    {
        Dispose(false);
    }

    public void Dispose()
```



```
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }

        protected virtual void Dispose(bool disposing)
        {
            // Protect from being called multiple times.
            if (disposed)
            {
                return;
            }

            if (disposing)
            {
                 // Clean up all managed resources.
                 if (resource != null)
                 {
                     resource.Dispose();
                 }
            }

            // Clean up all native resources.
            ReleaseBuffer(buffer);

            disposed = true;
        }
    }
```

☑ **Do** make a type finalizable, if the type is responsible for releasing an unmanaged resource that does not have its own finalizer. When implementing the finalizer, simply call Dispose(false) and place all resource cleanup logic inside the Dispose(bool disposing) method.

```
    // C# sample:
    public class ComplexResourceHolder : IDisposable
    {
        ...
        ~ComplexResourceHolder()
        {
            Dispose(false);
        }

        protected virtual void Dispose(bool disposing)
        {
            ...
        }
    }

    ' VB.NET sample:
```



```
Public Class DisposableResourceHolder
    Implements IDisposable

    ...
    Protected Overrides Sub Finalize()
        Dispose(False)
        MyBase.Finalize()
    End Sub

    Protected Overridable Sub Dispose(ByVal disposing As Boolean)
        ...
    End Sub

End Class
```

☑ **Do** be very careful to make type finalizable. Carefully consider any case in which you think a finalizer is needed. There is a real cost associated with instances with finalizers, from both a performance and code complexity standpoint.

☑ **Do** implement the Basic Dispose Pattern on every finalizable type. See the previous section for details on the basic pattern. This gives users of the type a means to explicitly perform deterministic cleanup of those same resources for which the finalizer is responsible.

☒ **Do not** let exceptions escape from the finalizer logic, except for system-critical failures. If an exception is thrown from a finalizer, the CLR may shut down the entire process preventing other finalizers from executing and resources from being released in a controlled manner.

3.9.4 Overriding Dispose

If you're inheriting from a base class that implements IDisposable, you must implement IDisposable also. Always call your base class's Dispose(bool) so it cleans up.

```
public class DisposableBase : IDisposable
{
    ~DisposableBase()
    {
        Dispose(false);
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
```



```
            // ...
        }
    }

    public class DisposableSubclass : DisposableBase
    {
        protected override void Dispose(bool disposing)
        {
            try
            {
                if (disposing)
                {
                    // Clean up managed resources.
                }

                // Clean up native resources.
            }
            finally
            {
                base.Dispose(disposing);
            }
        }
    }
```