

EUR AND IO FRAMEWORK CODING STANDARDS

This document outlines the EUR and IO coding style guideline for .NET (C#), JavaScript and database programming to be followed when doing application development. This document will periodically be reviewed for additional content and revised. Make sure to verify you are referencing the most recent version.

*C#, JavaScript,
Database*

Publish Date:
October 18, 2019

Author:
Sailer, Francis C

Contents

1. ACKNOWLEDGEMENT	3
2. OVERVIEW	3
3. PRINCIPLES & THEMES	3
4. TERMINOLOGY.....	4
5. GENERAL CODING STANDARDS	4
5.1. CLARITY AND CONSISTENCY	4
5.2. FORMATTING AND STYLE.....	4
5.3. USING LIBRARIES	5
5.4. GLOBAL VARIABLES.....	6
5.5. VARIABLE DECLARATIONS AND INITIALIZATIONS.....	6
5.6. FUNCTION DECLARATIONS AND CALLS	7
5.7. STATEMENTS	7
5.8. ENUMS.....	8
5.9. WHITESPACE	10
5.10. BRACES	12
5.11. COMMENTS	12
5.12. FILE NAMING.....	17
6. .NET CODING STANDARDS	18
6.1. DESIGN GUIDELINES FOR DEVELOPING CLASS LIBRARIES	18
6.2. FILES AND STRUCTURE.....	18
6.3. ASSEMBLY PROPERTIES.....	18
6.4. NAMING CONVENTIONS.....	18
6.5. CONSTANTS.....	22
6.6. STRINGS	23
1.1. ARRAYS AND COLLECTIONS	24
6.7. STRUCTURES.....	25
6.8. CLASSES	26
1.2. NAMESPACES	29
6.9. ERRORS AND EXCEPTIONS	29
1.3. RESOURCE CLEANUP	31
6.10. INTEROP	ERROR! BOOKMARK NOT DEFINED.
7. JAVASCRIPT CODING STANDARDS.....	38
7.1. FILES AND STRUCTURE	38
7.2. NAMING CONVENTIONS	39
7.3. STRINGS	39
7.4. ARRAYS AND COLLECTIONS.....	40
7.5. CLASSES	40
7.6. ERRORS AND EXCEPTIONS	40
8. SQL SERVER DATABASE CODING STANDARDS	41

8.1.	GENERAL	41
8.2.	NAMING	41
8.3.	STRUCTURE	42
8.4.	FORMATTING.....	42
8.5.	CODING.....	43
9.	APPENDIX	45
9.1.	REFERENCES	45

1. Acknowledgement

This document borrows heavily from a number of documents that are referenced in the appendix of this document.

2. Overview

This document defines the native .NET, JavaScript and database coding standard for the Application Developer Group (ADG) project team. This standard derives from the experience of previous and ongoing product development efforts and therefore is continuously evolving. If you discover a new best practice or a topic that is not covered, please bring that to the attention of the Application Development Group (EUR-IO Application Development Corps) for inclusion in this document.

While no set of guidelines will satisfy everyone, the goal of a standard is to create efficiencies across a community of developers. Applying a set of well-defined coding standards will result in code with fewer bugs, and better maintainability.

3. Principles & Themes

High-quality samples exhibit the following characteristics because customers use them as examples of best practices:

1. **Understandable**. Samples must be clearly readable and straightforward. They must showcase the key things they're designed to demonstrate. The relevant parts of a sample should be easy to reuse. Samples should not contain unnecessary code. They must include appropriate documentation.
2. **Correct**. Samples must demonstrate properly how to perform the key things they are designed to teach. They must compile cleanly, run correctly as documented, and be tested.
3. **Consistent**. Samples should follow consistent coding style and layout to make the code easier to read. Likewise, samples should be consistent with each other to make them easier to use together. Consistency shows craftsmanship and attention to detail.
4. **Modern**. Samples should demonstrate current practices such as use of Unicode, error handling, defensive programming, and portability. They should use current recommendations for runtime library and API functions. They should use recommended project & build settings.
5. **Safe**. Samples must comply with legal, privacy, and policy standards. They must not demonstrate hacks or poor programming practices. They must not permanently alter machine state. All installation and execution steps must be reversible.
6. **Secure**. The samples should demonstrate how to use secure programming practices such as least privilege, secure versions of runtime library functions, and SDL-recommended project settings.

The proper use of programming practices, design, and language features determines how well samples can achieve these. This code standard is designed to help you create samples that serve as "best practices" for others to emulate.

4. 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
<input checked="" type="checkbox"/> 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.
<input checked="" type="checkbox"/> Do Not...	This standard or practice should never be applied.	
<input checked="" type="checkbox"/> 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.
<input checked="" type="checkbox"/> You should not...	This standard or practice should not be followed, unless there's reasonable justification.	
<input checked="" type="checkbox"/> 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.

5. 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.

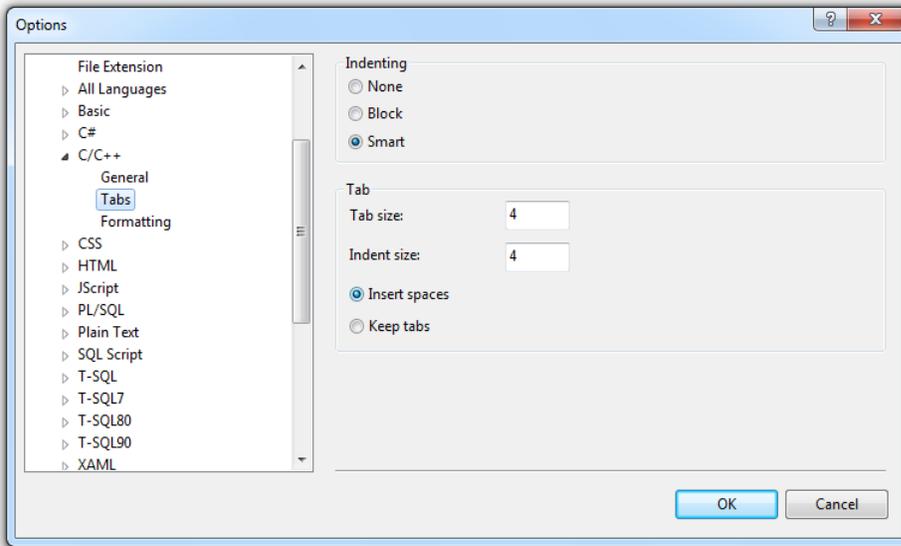
5.1. Clarity and Consistency

- 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**.

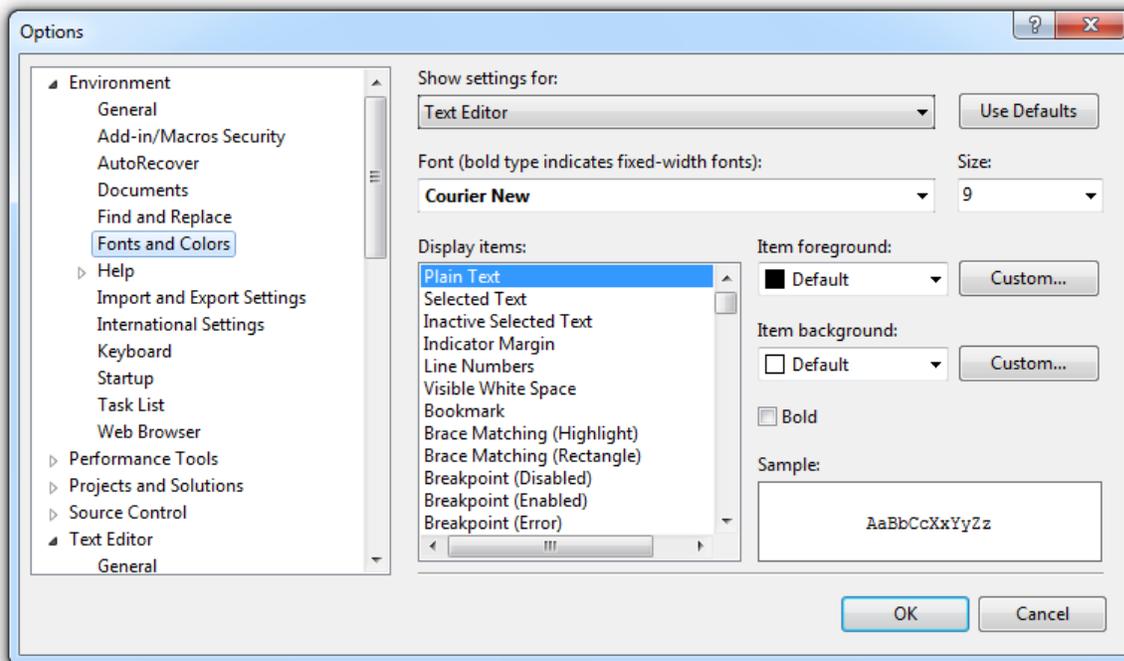
5.2. Formatting and Style

- 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 to 120. Having overly long lines inhibits the readability of code. Break the code line when the line length is greater than column 120 for readability.
- ✓ **Do** use a fixed-width font, typically Courier New, in your code editor.



5.3. Using Libraries

- ✗ **Do not** reference unnecessary libraries, include unnecessary header files, 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.

5.4. 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. Minimize the use of Session and Application objects and instead use Singleton objects and models.

5.5. Variable Declarations and Initializations

✓ **Do** use C# predefined types.

✗ **Do not** use aliases in the System namespace.

✓ With generics **Do** use capital letters for types. Reserve suffixing `Type` when dealing with the .NET type `Type`.

✗ **Do not** use fully qualified type names. Use the using statement instead.

✓ **Do** declare local variables in the minimum scope block that can contain them, typically just before use if the language allows; otherwise, at the top of that scope block.

✓ **Do** initialize variables when they are declared.

✓ **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;

// JavaScript sample:
var name = myObject["name"];
```

✗ **Do not** declare multiple variables in a single line. One declaration per line is recommended since it encourages commenting, and could avoid confusion.

Good:

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

// JavaScript sample:
var name = myObject["name"];
var val = d.getHours();
```

Bad:

```
// C# sample:  
string name = myObject.Name.ToString(), string val = time.Hours;  
  
// JavaScript sample:  
var name = myObject["name"], val = d.getHours().toString();
```

5.6. 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 if it will fit. For example,

Single Line Format:

```
// C# / JavaScript function call sample:  
hr = DoSomeFunctionCall(param1, param2, param3);
```

Multiple Line Formats:

```
// C# / JavaScript function declaration sample:  
HRESULT DoSomeFunctionCall(int param1, int param2, int *param3,  
                           int param4, int param5);  
  
// C# function call sample:  
hr = DoSomeFunctionCall(param1, param2, param3,  
                        param4, param5);
```

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.

```
// C# / JavaScript function call sample:  
hr = DoSomeFunctionCall(  
    hwnd,  
    param1,  
    param2,  
    param3,  
    param4,  
    param5);
```

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.

Do not include a space between a function name and its parentheses.

5.7. 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# / JavaScript sample:  
a = 1;  
b = 2;
```

Bad:

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

✅ **Do** add a space between control statements and its parenthesis.

5.8. Enums

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

✅ **Do** favor using an enum over static constants or “#define” values. 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.

Good:

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

Bad:

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

❌ **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  
    }
```

❌ **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 > Color.Black || c < Color.White)  
{  
    throw new ArgumentOutOfRangeException(...);  
}
```

Bad:

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

5.8.1. Flag Enums

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

✅ **Do** apply the `System.FlagsAttribute` to flag enums in .NET. **Do not** apply this attribute to simple enums.

✅ **Do** use powers of two for the flags enum values so they can be freely combined using the bitwise OR operation. For example,

```
// C# sample:  
[Flags]  
public enum AttributeTargets  
{  
    Assembly = 0x0001,  
    Class   = 0x0002,  
    Struct  = 0x0004,  
    ...  
}
```

✅ **You should** provide special enum values for commonly used combinations of flags. Bitwise operations are an advanced concept and should not be required for simple tasks. `FileAccess.ReadWrite` is an example of such a special value. However, **you should not** create flag enums where certain combinations of values are invalid.

```
// C# sample:  
[Flags]  
public enum FileAccess  
{  
    Read = 0x1,  
    Write = 0x2,  
    ReadWrite = Read | Write  
}
```

```
}
```

❌ **You should not** use flag enum values of zero, unless the value represents “all flags are cleared” and is named appropriately as “None”. The following C# example shows a common implementation of a check that programmers use to determine if a flag is set (see the if-statement below). The check works as expected for all flag enum values except the value of zero, where the Boolean expression always evaluates to true.

Bad:

```
[Flags]
public enum SomeFlag
{
    ValueA = 0, // This might be confusing to users
    ValueB = 1,
    ValueC = 2,
    ValueBAndC = ValueB | ValueC,
}
SomeFlag flags = GetValue();
if ((flags & SomeFlag.ValueA) == SomeFlag.ValueA)
{
    ...
}
```

Good:

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

5.9. Whitespace

5.9.1. 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(const Item& item)
{
    int counter = 0;

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

Bad:

```
// C++ sample:
void ProcessItem(const 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 lines after the 'if' block.

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

5.9.2. 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

// JavaScript 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
```

Bad:

```
// 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

// JavaScript 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
```

5.10. Braces

Do put starting braces on their own line, except in JavaScript. Additionally, always use braces in all frameworks.

Good:

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

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

Bad:

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

5.11. Comments

You should describe code that is not obvious at first glance.

You should not over comment your code. If your comments meet or exceed your code size, you are over commenting. Comments should be short and to the point.

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. Also all old code should be removed rather than being commented out.

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 `///
The single-line syntax (///`

```
// 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)
{
}
```

☑ **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:

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

Bad:

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

5.11.1.1. 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)
{
    // Report the error.
    ReportError(GetLastError());

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

Inline comments are permissible on the same line as the actual code only when giving a brief description of a structure member, class member variable, parameter, or a short statement. In this case it is a good idea to align the comments for all variables. For example:

```
class Example
{
public:
    ...

    void TestFunction
    {
        ...
        do
        {
            ...
        }
    }
}
```

```
    }  
    while (!ifFinished); // Continue if not finished.  
}  
  
private:  
    int m_length; // The length of the example  
    float m_accuracy; // The accuracy of the example  
};
```

You should not drown your code in comments. Commenting 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 *pWrinkle = apWrinkles[i]; // Get the next wrinkle  
    if (pWrinkle->IsNew() && // Process if it's a new wrinkle  
        nMaxImpact < pWrinkle->GetImpact()) // And it has the biggest impact  
    {  
        nMaxImpact = pWrinkle->GetImpact(); // Save its impact for comparison  
        pBestWrinkle = pWrinkle; // 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 'pBestWrinkle'.  
for (int i = 0; i <= nLastWrinkle; i++)  
{  
    Wrinkle *pWrinkle = apWrinkles[i];  
    if (pWrinkle->IsNew() && nMaxImpact < pWrinkle->GetImpact())  
    {  
        nMaxImpact = pWrinkle->GetImpact();  
        pBestWrinkle = pWrinkle;  
    }  
}
```

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

5.11.1.2. File Header Comments

Do have a file header comment at the start of every human-created code file. The header comment templates are as follows:

C# file header comment template:

```
/* ***** Module Header *****\  
Module Name: <File Name>  
Project: <Sample Name>
```

```
Copyright (c) Microsoft Corporation.
```

```
<Description of the file>
```

```
This source is subject to the Microsoft Public License.  
See http://www.microsoft.com/opensource/licenses.msp#Ms-PL.  
All other rights reserved.
```

```
THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND,  
EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED  
WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE.  
\\*****
```

JavaScript file header comment template:

```
/*!  
 * jQuery JavaScript Library v2.1.4  
 * http://jquery.com/  
 *  
 * Includes Sizzle.js  
 * http://sizzlejs.com/  
 *  
 * Copyright 2005, 2014 jQuery Foundation, Inc. and other contributors  
 * Released under the MIT license  
 * http://jquery.org/license  
 *  
 * Date: 2015-04-28T16:01Z  
 */
```

5.11.1.3. 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.

Use XML Documentation comments in C# classes. 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>  
/// The CodeExample class represents an example of code, and tracks  
/// the length and complexity of the example.  
/// </summary>  
public class CodeExample  
{  
    ...  
}
```

5.11.1.4. 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 JS 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 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>
```

Javascript XML comment:

```
function getArea(radius)  
{  
    /// <summary>Determines the area of a circle that has the specified radius  
    parameter.</summary>  
    /// <param name="radius" type="Number">The radius of the circle.</param>  
    /// <returns type="Number">The area.</returns>  
    var areaVal;  
    areaVal = Math.PI * radius * radius;  
    return areaVal;  
}
```

Any method or function which can fail with side-effects should have those side-effects clearly communicated in the function comment. As a general rule, code should be written so that it has no side-effects in error or failure cases; the presence of such side-effects should have some clear justification when the code is written. (Such justification is not necessary for routines which zero-out or otherwise overwrite some output-only parameter.)

5.11.1.5. Commenting Out Code

Commenting out code is necessary when you demonstrate multiple ways of doing something. The ways except the first one are commented out. Use [-or-] to separate the multiple ways. For example,

```
// C# / JavaScript sample:  
// Demo the first solution.  
DemoSolution1();  
  
// [-or-]  
  
// Demo the second solution.  
//DemoSolution2();
```

5.11.2. TODO Comments

☒ **Do not** use TODO comments in any released samples. Every sample must be complete and not require a list of unfinished tasks sprinkled throughout the code.

5.11.3. 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
```

5.12. File Naming

See Files and Structure sections in this document associated to a specific technology.

6. .NET Coding Standards

These coding standards can be applied to C#.

6.1. Design Guidelines for Developing Class Libraries

The [Design Guidelines for Developing Class Libraries](#) document on MSDN is a fairly thorough discussion of how to write managed code. The information in this section highlights some important standards and lists the EUR-IO Code Framework code samples' exceptions to the guidelines. Therefore, you should read the two documents side by side.

6.2. Files and Structure

Do not have more than one public type 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.

6.3. Assembly Properties

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

Standard	Example
Set Copyright to Copyright © Department of State 2019	<code>[assembly: AssemblyCopyright("Copyright © Department of State 2019")]</code>
Set AssemblyCompany to Department of State	<code>[assembly: AssemblyCompany("Department of State")]</code>
Set both AssemblyTitle and AssemblyProduct to the current sample name	<code>[assembly: AssemblyTitle("CSNamedPipeClient")] [assembly: AssemblyProduct("CSNamedPipeClient")]</code>

6.4. Naming Conventions

6.4.1. General Naming Conventions

Do use meaningful names for various types, functions, variables, constructs and types.

You should not use 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".

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

6.4.2. 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	<pre>public class ComplexNumber {...} public struct ComplexStruct {...}</pre>
Namespace	PascalCasing	Noun <input type="checkbox"/> Do not use the same name for a namespace and a type in that namespace.	<pre>namespace Microsoft.Sample.Windows7</pre>
Enumeration	PascalCasing	Noun <input checked="" type="checkbox"/> Do name flag enums with plural nouns or noun phrases and simple enums with singular nouns or noun phrases.	<pre>[Flags] public enum ConsoleModifiers { Alt, Control }</pre>
Method	PascalCasing	Verb or Verb object pair <input checked="" type="checkbox"/> Methods with return values Should have a name describing the value returned.	<pre>public void Print() {...} public void ProcessItem() {...} public string GetObjectState(){...}</pre>
Public Property	PascalCasing	Noun or Adjective <input checked="" type="checkbox"/> 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”. <input checked="" type="checkbox"/> 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.	<pre>public string CustomerName public ItemCollection Items public bool CanRead</pre>
Non-public Field	camelCasing or _camelCasing	Noun or Adjective. <input checked="" type="checkbox"/> Do be consistent in a code sample when you use the '_' prefix.	<pre>private string name; private string _name;</pre>

Event	PascalCasing	<p>Verb or Verb phrase</p> <ul style="list-style-type: none"> <input checked="" type="checkbox"/> Do give events names with a concept of before and after, using the present and past tense. <input type="checkbox"/> Do not use “Before” or “After” prefixes or postfixes to indicate pre and post events. 	<pre>// 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</pre>
Delegate	PascalCasing	<ul style="list-style-type: none"> <input checked="" type="checkbox"/> Do add the suffix ‘EventHandler’ to names of delegates that are used in events. <input checked="" type="checkbox"/> Do add the suffix ‘Callback’ to names of delegates other than those used as event handlers. <input checked="" type="checkbox"/> Do suffix custom attributes classes with <code>Attribute</code>. <input checked="" type="checkbox"/> Do suffix custom exception classes with <code>Exception</code>. <input type="checkbox"/> Do not add the suffix “Delegate” to a delegate. 	<pre>public delegate WindowClosedEventHandler</pre>
Interface	PascalCasing ‘I’ prefix	Noun	<pre>public interface IDictionary</pre>
Constant	PascalCasing for publicly visible; camelCasing for internally visible; All capital only for abbreviation of one or two chars long.	Noun	<pre>public const string MessageText = "A"; private const string messageText = "B"; public const double PI = 3.14159...;</pre>
Parameter, Variable	camelCasing	Noun	<pre>int customerID;</pre>

Generic Type Parameter	PascalCasing 'T' prefix	Noun <input checked="" type="checkbox"/> 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. <input checked="" type="checkbox"/> Do prefix descriptive type parameter names with T. <input checked="" type="checkbox"/> You should use T as the type parameter name for types with one single-letter type parameter.	T, TItem, TPolicy
Resource	PascalCasing	Noun <input checked="" type="checkbox"/> Do provide descriptive rather than short identifiers. Keep them concise where possible, but do not sacrifice readability for space. <input checked="" type="checkbox"/> Do use only alphanumeric characters and underscores in naming resources.	ArgumentExceptionInvalidName

6.4.3. Hungarian Notation

Do not use Hungarian notation (i.e., do not encode the type of a variable in its name) in .NET.

6.4.4. UI Control Naming Conventions

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

Control Type	Prefix
Button	btn
CheckBox	chk
CheckedListBox	lst
ComboBox	cmb
ContextMenu	mnu
DataGrid	dg
DateTimePicker	dtp
Form	suffix: XXXForm
GroupBox	grp

ImageList	iml
Label	lbl
ListBox	lst
ListView	lvw
Menu	mnu
MenuItem	mnu
NotificationIcon	nfy
Panel	pnl
PictureBox	pct
ProgressBar	prg
RadioButton	rb
Splitter	spl
StatusBar	sts
TabControl	tab
TabPage	tab
TextBox	tb
TreeView	tvw

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

6.5. 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);
}
```

☑ **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");  
...  
}
```

6.6. Strings

❌ **Do not** use the '+' operator to concatenate many strings. Instead, you should use `StringBuilder` for concatenation. However, **do** use the '+' operator 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 overloads that explicitly specify the string comparison rules for string operations. Typically, this involves calling a method overload that has a parameter of type `StringComparison`.
- ✅ **Do** use `StringComparison.Ordinal` or `StringComparison.OrdinalIgnoreCase` for comparisons as your safe default for culture-agnostic string matching, and for better performance.
- ✅ **Do** use string operations that are based on `StringComparison.CurrentCulture` when you display output to the user.
- ✅ **Do** use the non-linguistic `StringComparison.Ordinal` or `StringComparison.OrdinalIgnoreCase` values instead of string operations based on `CultureInfo.InvariantCulture` when the comparison is linguistically irrelevant (symbolic, for example). Do not use string operations based on `StringComparison.InvariantCulture` in most cases. One of the few exceptions is when you are persisting linguistically meaningful but culturally agnostic data.
- ✅ **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))
```
- ❌ **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.

6.1.1 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}  
};  
  
// Multidimensional arrays
```

```
int [,] multiDimArray =  
{  
    {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 types such as `StringDictionary`, `NameValueCollection`, and `HybridCollection`. `Hashtables` can be used if lesser 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** implement both `IEnumerator<T>` and `IEnumerable<T>` on the same type. The same applies to the nongeneric interfaces `IEnumerator` and `IEnumerable`. In other words, a type should be either a collection or an enumerator, but not both.
- ✗ **Do not** return a null reference for `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();  
foreach (int v in arr)  
{  
    ...  
}
```

6.7. Structures

- ✓ **Do** ensure that a state where all instance data is set to zero, false, or null (as appropriate) is valid. This prevents accidental creation of invalid instances when an array of the structs is created.
- ✓ **Do** implement `IEquatable<T>` on value types. The `Object.Equals` method on value types causes boxing and its default implementation is not very efficient, as it uses reflection. `IEquatable<T>.Equals` can have much better performance and can be implemented such that it will not cause boxing.

6.7.1. Structures vs. Classes

- ☒ **Do not** define a struct unless the type has all of the following characteristics:
 - It logically represents a single value, similar to primitive types (int, double, etc.).
 - It has an instance size fewer than 16 bytes.
 - It is immutable.
 - It will not have to be boxed frequently.

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

6.8. 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”.

6.8.1. 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.

6.8.2. 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.

6.8.3. 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.
- ✗ **Do not** call virtual members on an object inside its constructors. Calling a virtual member causes the most-derived override to be called regardless of whether the constructor for the type that defines the most-derived override has been called.

6.8.4. Methods

- ✓ **Do** place all **out** parameters after all 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`.

6.8.5. Events & Exceptions

- ✓ **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.

6.8.6. Member Overloading

- ✓ **Do** use member overloading rather than defining members with default arguments. Default arguments are not CLS-compliant 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 void Rotate(Matrix data) {  
    Rotate(data, 180);  
}  
  
public void Rotate(Matrix data, int degrees) {  
    // Do rotation here  
}
```

Bad:

```
public void Rotate(Matrix data, int degrees, void =, void 180) {  
    // Do rotation here  
    // Warning!!! Optional parameters not supported  
}
```

❌ **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.

6.8.7. 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.

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

6.8.8. 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.

6.8.9. Static Classes

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

6.8.10. 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()
    {
        ...
    }
}
```

1.1.Namespaces

- ✅ **Do** use the default namespaces of projects created by Visual Studio in EUR-IO Code Framework code samples.

6.9. Errors and Exceptions

6.9.1. 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. 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);
}
```

☒ **Do not** throw exceptions from exception filter blocks. When an exception filter raises an exception, the exception is caught by the CLR, and the filter returns false. This behavior is indistinguishable from the filter executing and returning false explicitly and is therefore very difficult to debug.

```
// C# sample
// This is bad design. The exception filter (when clause)
// may throw an exception when the InnerException property
// returns null
try {
}
catch (ArgumentException e) {
    e.InnerException.Message.StartsWith("File")...End;
    try {
    }
}
```

☒ **Do not** explicitly throw exceptions from finally blocks. Implicitly thrown exceptions resulting from calling methods that throw are acceptable.

6.9.2. 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)
{
    ...
}
```

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

☑ **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  
}
```

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

1.2.Resource Cleanup

☒ **Do not** force garbage collections with GC.Collect.

6.9.3. 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();  
    }  
}
```

C# provides the using statement that can be leveraged instead of plain try-finally to clean up objects implementing the IDisposable interface.

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

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

1.1.1. 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;  
  }  
}
```

- ✓ **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;  
}
```

- ✓ **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);  
}
```

❌ **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. If executing a Dispose(bool disposing) method, never throw an exception if disposing is false. Doing so will terminate the process if executing inside a finalizer context.

✅ **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;  
    }  
}
```

6.9.4. 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)  
    {  
        ...  
    }  
}
```

☑ **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.

☑ **You should** create and use a critical finalizable object (a type with a type hierarchy that contains `CriticalFinalizerObject`) for situations in which a finalizer absolutely must execute even in the face of forced application domain unloads and thread aborts.

☑ **Do** prefer resource wrappers based on `SafeHandle` or `SafeHandleZeroOrMinusOneIsInvalid` (for Win32 resource handle whose value of either 0 or -1 indicates an invalid handle) to writing finalizer by yourself to encapsulate unmanaged resources where possible, in which case a finalizer becomes unnecessary because the wrapper is responsible for its own resource cleanup. Safe handles implement the `IDisposable` interface, and inherit from `CriticalFinalizerObject` so the finalizer logic will absolutely execute even in the face of forced application domain unloads and thread aborts.

```
/// <summary>  
/// Represents a wrapper class for a pipe handle.  
/// </summary>  
[SecurityCritical(SecurityCriticalScope.Everything),  
HostProtection(SecurityAction.LinkDemand, MayLeakOnAbort = true),  
SecurityPermission(SecurityAction.LinkDemand, UnmanagedCode = true)]  
internal sealed class SafePipeHandle : SafeHandleZeroOrMinusOneIsInvalid
```

```
{
    private SafePipeHandle()
        : base(true)
    {
    }

    public SafePipeHandle(IntPtr preexistingHandle, bool ownsHandle)
        : base(ownsHandle)
    {
        base.SetHandle(preexistingHandle);
    }

    [ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success),
    DllImport("kernel32.dll", CharSet = CharSet.Auto, SetLastError = true)]
    [return: MarshalAs(UnmanagedType.Bool)]
    private static extern bool CloseHandle(IntPtr handle);

    protected override bool ReleaseHandle()
    {
        return CloseHandle(base.handle);
    }
}

/// <summary>
/// Represents a wrapper class for a local memory pointer.
/// </summary>
[SuppressUnmanagedCodeSecurity,
HostProtection(SecurityAction.LinkDemand, MayLeakOnAbort = true)]
internal sealed class SafeLocalMemHandle : SafeHandleZeroOrMinusOneIsInvalid
{
    public SafeLocalMemHandle()
        : base(true)
    {
    }

    public SafeLocalMemHandle(IntPtr preexistingHandle, bool ownsHandle)
        : base(ownsHandle)
    {
        base.SetHandle(preexistingHandle);
    }

    [ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success),
    DllImport("kernel32.dll", CharSet = CharSet.Auto, SetLastError = true)]
    private static extern IntPtr LocalFree(IntPtr hMem);

    protected override bool ReleaseHandle()
    {
        return (LocalFree(base.handle) == IntPtr.Zero);
    }
}
```

☒ **Do not** access any finalizable objects in the finalizer code path, as there is significant risk that they will have already been finalized. For example, a finalizable object A that has a reference to another finalizable object B cannot reliably use B in A's finalizer, or vice versa. Finalizers are called in a random order (short of a weak ordering guarantee for critical finalization).

It is OK to touch unboxed value type fields.

Also, be aware that objects stored in static variables will get collected at certain points during an application domain unload or while exiting the process. Accessing a static variable that refers to a finalizable object (or calling a static method that might use values stored in static variables) might not be safe if `Environment.HasShutdownStarted` returns true.

6.9.5 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);
        }
    }
}
```

7. JavaScript Coding Standards

These coding standards apply to general JavaScript code. There may be variations acceptable when using JavaScript frameworks such as JQuery.

7.1. Files and Structure

- ✔ Do link to external JavaScript files using `<script>` tags.
- ✘ Do not include JavaScript directly in an HTML file.

7.2. Naming Conventions

7.2.1. General Naming Conventions

- ✔ Do use meaningful names for various types, functions, variables, constructs and types.
- ✘ **You should not** use 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”.
- ✘ **Do not** use underscores, hyphens, or any other non-alphanumeric characters.

7.2.2. Capitalization Naming Rules for Identifiers

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

Identifier	Casing	Naming Structure	Example
Variables	camelCasing	Noun or Adjective	<code>var customerID;</code>
Constant Variables	CAPS_WITH_UNDERLINES	Noun or Adjective	<code>var CUSTOMER_ID;</code>
File Names	Lowercase-with-dashes	Noun, Adjective or Verb combined as necessary to describe the feature or features.	cache-logger.js
Functions / Methods	camelCasing	Verb or Verb object pair ✔ Methods with return values Should have a name describing the value returned.	<code>function print() {...}</code> <code>function processItem() {...}</code> <code>function getObjectState() {...}</code>

7.3. Strings

- ✔ Do use single quotes instead of doubles. This is especially useful for strings that include HTML.

```
// sample:
var message = '<div class="className">Hello World!</div>'
```

7.4. Arrays and Collections

- ✓ **Do** use object literals instead of constructors. Constructors are prone to errors from their arguments, and are generally slower than literals.

Good:

```
var array = [];  
var hash = {};
```

Bad:

```
var array = new Array();  
var hash = new Hash();
```

7.5. Classes

7.5.1. Naming

- ✗ **You should not** use numbers in a class name unless absolutely unavoidable.
- ✓ **Do** add an underscore before any private method's or property's name.

7.5.2. Properties

- ✓ **Do** provide sensible default values for properties that will not result in a security hole or inefficiency.

7.5.3. Methods

- ✓ **Do** start the method name with a verb that describes its behavior.
- ✓ **Do** validate data types of passed arguments.
- ✓ **Do** include 'has' or 'is' or 'can' in the name if it returns a Boolean value.

7.6. Errors and Exceptions

- ✓ **Do** report execution failures by throwing exceptions.
- ✗ **Do not** return error codes.
- ✓ **Do** throw the most specific exception that makes sense.

8. SQL Server Database Coding Standards

8.1. General

☒ **You should not** use triggers when application logic will complete the same result. Triggers tend to hide functionality and are hard to debug.

8.2. Naming

☒ **Do not** use spaces in the name of database objects.

☒ **Do not** use SQL keywords as the name of database objects. If this is unavoidable, surround the object name with brackets such as [Year].

☒ **Do not** prefix stored procedures with 'sp_'.

☑ **Do** prefix table names with the owner name.

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

Identifier	Casing	Naming Structure	Example
Tables	PascalCasing	Noun (plural) ☑ Do end table names with an 's'	Products Customers
Stored Procedures	camelCasing	sp<App Name>_[<Group Name>]<Action><table or logical instance>	spOrders_GetNewOrders spProducts_UpdateProduct
Triggers	Modified PascalCasing	TR_<TableName>_<Action>	TR_Orders_UpdateProducts
Indexes	Modified PascalCasing	IX_<TableName>_<Columns separated by _>	IX_Products_ProductID
Primary Keys	Modified PascalCasing	PK_<TableName>	PK_Products
Foreign Keys	Modified PascalCasing	FK_<TableName1>_<TableName2>	FK_Products_Orders

Defaults	Modified PascalCasing	DF_<TableName1>_<ColumnName>	DF_Products_Quantity
Columns	PascalCasing	<input checked="" type="checkbox"/> Do precede the column name with <TableName> if a column references another table's column.	ID For another table's column CustomerID
Keywords	UpperCasing		SELECT INSERT UPDATE WHERE AND OR LIKE

8.3. Structure

- Do** add a primary key to each table.
- Do** normalize data to third normal form if reasonable and prudent. Do not compromise on performance to reach third normal form. Sometimes, a little de-normalization results in better performance.
- Do not** use TEXT as a data type; use the maximum allowed characters of VARCHAR instead.
- Do not** default VARCHAR columns to NULL. Default to an empty string instead.
- Do not** allow NULLs in columns with default values.
- Do** create stored procedures on the same database as the main tables they will be accessing.

8.4. Formatting

- Do** use upper case for all SQL keywords.
- Do** indent code to improve readability.
- Do** use single quote characters to delimit strings. Nest single quotes to express a single quote within a string such as,

```
SET @sExample = 'SQL''s Authority'
```

- Do** use parentheses to increase readability.
- Do** use BEGIN...END blocks only when multiple statements are present within a conditional code segment.
- Do** use one blank line to separate code sections.

- ✓ **Do** use spaces so that expressions read like sentences.
- ✓ **Do** format JOIN operations using indents. Use ANSI Joins instead of old style joins.
- ✓ **Do** place SET statements before any executing code in the procedure.

8.5. Coding

- ✓ **Do** optimize queries using the tools provided by SQL Server.
- ✗ **Do not** use SELECT *.
- ✓ **Do** return multiple result sets from one stored procedure to avoid trips from the application server to SQL Server.
- ✗ **You should not** use temporary tables. Use derived tables or common table expressions (CTE) wherever possible as they perform better.
- ✗ **You should not** <> as a comparison operator. Use IN(...) instead.

Good

ID IN(1,3,4,5)

Bad

ID <> 2

- ✓ **Do** use SET NOCOUNT ON at the beginning of a stored procedure.
- ✗ **Do not** use cursors or application loops to do inserts. Use INSERT INTO instead.
- ✓ **Do** fully qualify all stored procedure and table references in stored procedures.
- ✗ **Do not** define default values for parameters. The front end should supply the value.
- ✗ **Do not** use the RECOMPILE option for stored procedures.
- ✗ **Do not** use column numbers in the ORDER BY clause.
- ✗ **Do not** use GOTO.
- ✓ **Do** check the global variable @@ERROR immediately after executing a data manipulation statement such as INSERT/UPDATE/DELETE so that you can roll-back the transaction if an error occurs.
- ✓ **Do** basic validations in the front-end during data entry.

- ✓ **Do** off-load tasks, like string manipulations, concatenations or row numbering to the front-end application if these operations are going to consume more CPU cycles on the database server.
- ✓ **Do** use a column list in an INSERT statement. This will help avoid problems when the table structure changes.
- ✓ **Do** minimize the use of NULLs, as they often confuse front-end applications, unless the applications are coded intelligently to eliminate NULLs or convert the NULLs into some other form. Make sure any expression that deals with NULL results in a NULL output. The ISNULL and COALESCE functions are helpful in dealing with NULL values.
- ✗ **Do not** use the identitycol or rowguidcol.
- ✗ **You should not** use cross joins, if possible.
- ✓ **Do** use the primary key in the WHERE condition when executing an UPDATE or DELETE statement, if possible. This reduces error possibilities.
- ✗ **You should not** use TEXT or NTEXT datatypes for storing large textual data. Use the maximum allowed characters of VARCHAR instead.
- ✗ **You should not** use dynamic SQL statements.
- ✓ **Do** access tables in the same order in your stored procedures and triggers consistently. Remember to avoid the use of Triggers.
- ✗ **Do not** call functions repeatedly within your stored procedures, triggers, functions and batches.
- ✓ **Do** define default constraints at the column level.
- ✗ **You should not** use wild-card characters at the beginning of a word while searching using the LIKE keyword, as these results in an index scan, which defeats the purpose of an index.
- ✓ **Do** define all constraints, other than defaults, at the table level.
- ✗ **Do not** return a result set when a result set is not needed.
- ✓ **Do** avoid rules, database level defaults that must be bound or user-defined data types. While these are legitimate database constructs, opt for constraints and column defaults to hold the database consistent for development and conversion coding.
- ✓ **Do** define constraints that apply to more than one column at the table level.
- ✓ **Do** use the CHAR data type for a column only when the column is non-nullable.
- ✗ **Do not** use white space in identifiers.

☒ **You should not** use the RETURN statement to return data. It is meant for returning the execution status only.

9. Appendix

9.1. References

All-In-One Code Framework Coding Standards guide for the .NET coding standards, Dan Ruder, Jialiang Ge, Microsoft, 2011.

C# Coding Standard, Juval Lowy, iDesign, 2011.

Google Javascript Coding Standards, Google, 2014.

SQL Server Coding Standards and Guidelines v1.0, SQL Authority.