|  |
| --- |
| **All-In-One Code Framework Coding Standards** |
| by **Dan Ruder**, **Jialiang Ge** |
|  |
| This document describes the coding style guideline for native .NET (C#) programming used by the Microsoft All-In-One Code Framework project team. |
|  |
|  |
|  |
|  |

Acknowledgement

Each chapter in this document has to acknowledge **Dan Ruder**, a Principal Escalation Engineer of Microsoft. Dan carefully reviewed every word, and contributed review comments on significant portions of the book based on his more than 20 years’ programming experience. Working with the nice man has been a special treat to me.

I also thank four managers at Microsoft for continuously supporting and sponsoring the work: **Vivian Luo**, **Allen Ding**, **Felix Wu** and **Mei Liang**.

This document wouldn’t contain the depth of technical details or the level of completeness it has without the input of the following people.

**Alexei Levenkov , Hongye Sun, Jie Wang, Ji Zhou, Michael Sun, Kira Qian, Allen Chen, Yi-Lun Luo, Steven Cheng, Wen-Jun Zhang, Linda Liu**

Some chapters derive from several Microsoft product teams’ coding standards. I appreciate their sharing.

The coding standards are continuously evolving. If you discover a new best practice or a topic that is not covered, please bring that to the attention of the All-In-One Code Framework Project Group ([onecode@microsoft.com](mailto:onecode@microsoft.com)). I look forward to appreciating your contributions. ☺

Disclaimer

This coding-standard document 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.

Please feel free to use the coding standards when you are writing VC# code. It would be nice, however, if you could inform us that you are using the document, or send us your feedback. You may contact us at our email address: [onecode@microsoft.com](mailto:onecode@microsoft.com).

Sage 300 ERP Disclaimer

This document has had the C++ and VB.NET code samples and sections removed. The Sage 300 ERP Columbus NA 2.0 project is only concerned with the C# standards. Therefore, removing the other sections and samples has made this document smaller and more concise.

****Table of Contents****

1 Overview 1

1.1 Principles & Themes 1

1.2 Terminology 1

2 General Coding Standards 3

2.1 Clarity and Consistency 3

2.2 Formatting and Style 3

2.3 Using Libraries 4

2.4 Global Variables 4

2.5 Variable Declarations and Initalizations 5

2.6 Function Declarations and Calls 5

2.7 Statements 6

2.8 Enums 6

2.9 Whitespace 9

2.10 Braces 10

2.11 Comments 10

2.12 Regions 15

3 .NET Coding Standards 15

3.1 Design Guidelines for Developing Class Libraries 15

3.2 Files and Structure 15

3.3 Assembly Properties 15

3.4 Naming Convensions 15

3.5 Constants 19

3.6 Strings 19

3.7 Arrays and Collections 20

3.8 Structures 22

3.9 Classes 22

3.10 Namespaces 26

3.11 Errors and Exceptions 26

3.12 Resource Cleanup 28

3.13 Interop 37

# Overview

This document defines the native .NET coding standard for the [All-In-One Code Framework](http://1code.codeplex.com) project team. This standard derives from the experience of product development efforts and 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 [All-In-One Code Framework Project Group](mailto:onecode@microsoft.com) and have the conclusion added to this document.

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. Adopting an unfamiliar standard may be awkward initially, but the pain fades quickly and the benefits are quickly realized, especially when you inherit ownership of others' code.

## 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 customers to emulate.

## 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. |

# 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.

## Clarity and Consistency

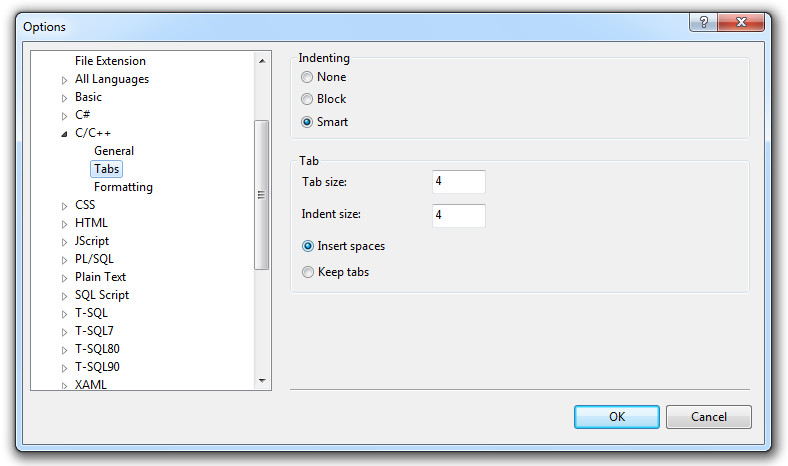
🗹 **Do** ensure that clarity, readability and transparency are paramount. These coding standards strive to ensure that the resultant code is easy to understand and maintain, but nothing beats fundamentally clear, concise, self-documenting code.

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

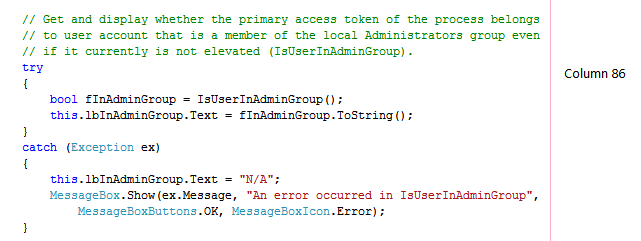
## Formatting and Style

🗷 **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.

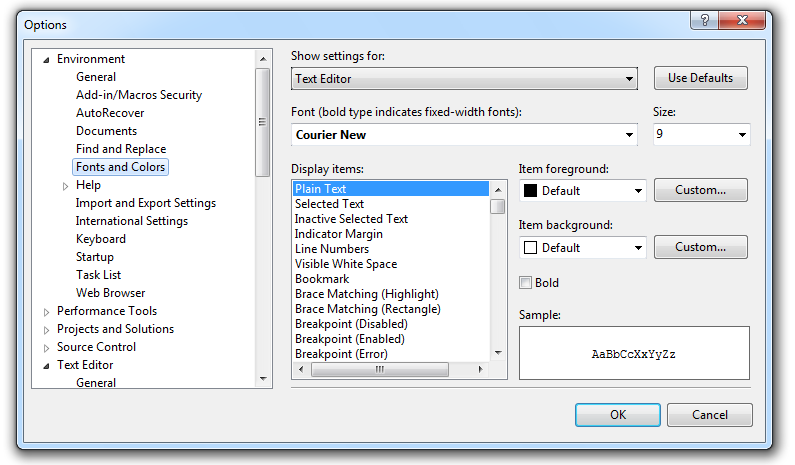
Visual Studio text editor can be configured to insert spaces for tabs.



🗹 **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 column 78 for readability. If column 78 looks too narrow, use column 86 or 90.



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



## 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.

## 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.

## Variable Declarations and Initalizations

🗹 **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.

string name = myObject.Name;

int val = 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.

## 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,

Single Line Format:

hr = DoSomeFunctionCall(param1, param2, param3);

Multiple Line Formats:

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.

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.

## 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:

a = 1;

b = 2;

Bad:

a = 1; b = 2;

## 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 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:

public enum Color

{

Red,

Green,

Blue

}

Bad:

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.

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:

if (c > Color.Black || c < Color.White)

{

throw new ArgumentOutOfRangeException(...);

}

Bad:

if (!Enum.IsDefined(typeof(Color), c))

{

throw new InvalidEnumArgumentException(...);

}

### 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,

[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.

[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 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)

{

...

}

## Whitespace

### 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:

void ProcessItem(Item item)

{

int counter = 0;

if(...)

{

}

}

Bad:

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.

### 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:

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:

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

## Braces

🗹 **Do** use Allman bracing style in [All-In-One Code Framework](http://1code.codeplex.com) code samples.

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:

if (x > 5)

{

y = 0;

}

Bad:

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:

if (x > 5)

{

y = 0;

}

Bad:

if (x > 5) y = 0;

## 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. 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)

{

}

🗹 **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();

Bad:

//intialize the components on the Windows Form.

InitializeComponent();

### 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;

}

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 (!fFinished); // 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.

### 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:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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.mspx#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.

\\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

### 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.

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.

Class comment template:

/// <summary>

/// <Class description>

/// </summary>

### 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.

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.

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>

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.)

### 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,

// Demo the first solution.

DemoSolution1();

// [-or-]

// Demo the second solution.

//DemoSolution2();

### 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.

## 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.

#region Helper Functions for XX

...

#endregion

# .NET Coding Standards

## Design Guidelines for Developing Class Libraries

The [Design Guidelines for Developing Class Libraries](http://msdn.microsoft.com/en-us/library/ms229042.aspx) 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 All-In-One Code Framework code samples’ exceptions to the guidelines. Therefore, you had better read the two documents side by side.

## 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.

## 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 2010 | [assembly: AssemblyCopyright("Copyright © Microsoft Corporation 2010")] |
| 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")] |

## Naming Convensions

### General Naming Conventions

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

🗷 **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”.

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

### 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 or \_camelCasing | Noun or Adjective.  🗹 **Do** be consistent in a code sample when you use the '\_' prefix. | private string name;  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 with one single-letter type parameter. | T, TItem, TPolicy |
| **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 |

### Hungarian Notation

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

### UI Control Naming Conventions

UI controls would use the following prefixes. 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 | lb |
| ListBox | lst |
| ListView | lvw |
| Menu | mnu |
| MenuItem | mnu |
| NotificationIcon | nfy |
| Panel | pnl |
| PictureBox | pct |
| ProgressBar | prg |
| RadioButton | rad |
| Splitter | spl |
| StatusBar | sts |
| TabControl | tab |
| TabPage | tab |
| TextBox | tb |
| Timer | tmr |
| TreeView | tvw |

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

## 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");

...

}

## 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](http://msdn.microsoft.com/en-us/library/system.stringcomparison.aspx).

🗹 **Do** use [StringComparison.Ordinal](http://msdn.microsoft.com/en-us/library/system.stringcomparison.ordinal.aspx) or [StringComparison.OrdinalIgnoreCase](http://msdn.microsoft.com/en-us/library/system.stringcomparison.ordinalignorecase.aspx) 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](http://msdn.microsoft.com/en-us/library/system.stringcomparison.currentculture.aspx) when you display output to the user.

🗹 **Do** use the non-linguistic [StringComparison.Ordinal](http://msdn.microsoft.com/en-us/library/system.stringcomparison.ordinal.aspx) or [StringComparison.OrdinalIgnoreCase](http://msdn.microsoft.com/en-us/library/system.stringcomparison.ordinalignorecase.aspx) values instead of string operations based on [CultureInfo.InvariantCulture](http://msdn.microsoft.com/en-us/library/system.globalization.cultureinfo.invariantculture.aspx) 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](http://msdn.microsoft.com/en-us/library/system.string.equals.aspx) 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](http://msdn.microsoft.com/en-us/library/system.string.compare.aspx) or [CompareTo](http://msdn.microsoft.com/en-us/library/system.string.compareto.aspx) 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](http://msdn.microsoft.com/en-us/library/system.string.toupperinvariant.aspx) method instead of the [String.ToLowerInvariant](http://msdn.microsoft.com/en-us/library/system.string.tolowerinvariant.aspx) method when you normalize strings for comparison.

## 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 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** 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)

{

...

}

## 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.

### 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.

## 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”.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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 = 180)

{

// Do rotation here

}

🗷 **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.

### 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.

### 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.

### Static Classes

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

### 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()

{

...

}

}

## Namespaces

🗹 **Do** use the default namespaces of projects created by Visual Studio in All-In-One Code Framework code samples. It is not necessary to rename the namespace to the form of Microsoft.Sample.TechnologyName.

## Errors and Exceptions

### 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,

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.

// This is bad design. The exception filter (When clause)

// may throw an exception when the InnerException property

// returns null

try

{

...

}

catch (ArgumentException e)

when (e.InnerException.Message.StartsWith("File"))

{

...

}

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

### 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:

try

{

...

}

catch(System.NullReferenceException exc)

{

...

}

catch(System.ArgumentOutOfRangeException exc)

{

...

}

catch(System.InvalidCastException exc)

{

...

}

Bad:

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:

try

{

... // Do some reading with the file

}

catch

{

file.Position = position; // Unwind on failure

throw; // Rethrow

}

Bad:

try

{

... // Do some reading with the file

}

catch (Exception ex)

{

file.Position = position; // Unwind on failure

throw ex; // Rethrow

}

## Resource Cleanup

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

### 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,

FileStream stream = null;

try

{

stream = new FileStream(...);

...

}

finally

{

if (stream != null)

{

stream.Close();

}

}

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

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.

### 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:

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.

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.

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.

🗷 Y**ou 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.

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;

}

}

### 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:

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.

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.

🗷 **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.

### 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);

}

}

}

## Interop

### P/Invoke

🗹 **Do** consult [P/Invoke Interop Assistant](http://clrinterop.codeplex.com/) and <http://pinvoke.net> to write P/Invoke signatures.

🗹 **You can** use IntPtr for manual marshaling. By declaring parameters and fields as IntPtr, you can boost performance, albeit at the expense of ease of use, type safety, and maintainability. Sometimes it is faster to perform manual marshaling by using methods available on the Marshal class rather than to rely on default interop marshaling. For example, if large arrays of strings need to be passed across an interop boundary, but the managed code needs only a few of those elements, you can declare the array as IntPtr and manually access only those few elements that are required.

🗷 **Do not** aggressively pin short-lived objects. Pinning short-lived objects unnecessarily extends the life of a memory buffer beyond the duration of the P/Invoke call. Pinning prevents the garbage collector from relocating the bytes of the object in the managed heap, or relocating the address of a managed delegate. However, it is acceptable to pin long-lived objects, which are ideally created during application initialization, because they are not moved relative to short-lived objects. It is costly to pin short-lived objects for a long period of time, because compacting occurs most in Generation 0 and the garbage collector cannot relocate pinned objects. This results in inefficient memory management that can adversely affect performance. For more information about copying and pinning, see <http://msdn.microsoft.com/en-us/library/23acw07k.aspx>.

🗹 **Do** set CharSet = CharSet.Auto and SetLastError = true in the P/Invoke signature. For example,

[DllImport("kernel32.dll", CharSet = CharSet.Auto, SetLastError = true)]

public static extern SafeFileMappingHandle OpenFileMapping(

FileMapAccess dwDesiredAccess, bool bInheritHandle, string lpName);

🗹 **You should** wrap unmanaged resources in SafeHandle classes. The SafeHandle class is discussed in the [Finalizable Types](#_Finalizable_Types) section. For example, the handle of file mapping is wrapped as follows.

/// <summary>

/// Represents a wrapper class for a file mapping handle.

/// </summary>

[SuppressUnmanagedCodeSecurity,

HostProtection(SecurityAction.LinkDemand, MayLeakOnAbort = true)]

internal sealed class SafeFileMappingHandle : SafeHandleZeroOrMinusOneIsInvalid

{

[SecurityPermission(SecurityAction.LinkDemand, UnmanagedCode = true)]

private SafeFileMappingHandle()

: base(true)

{

}

[SecurityPermission(SecurityAction.LinkDemand, UnmanagedCode = true)]

public SafeFileMappingHandle(IntPtr handle, bool ownsHandle)

: base(ownsHandle)

{

base.SetHandle(handle);

}

[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);

}

}

🗹 **You should** throw Win32Exception on the failure of P/Invoked functions that set the Win32 last error. If the function uses some unmanaged resources, free the resource in the finally block.

SafeFileMappingHandle hMapFile = null;

try

{

// Try to open the named file mapping.

hMapFile = NativeMethod.OpenFileMapping(

FileMapAccess.FILE\_MAP\_READ, // Read access

false, // Do not inherit the name

FULL\_MAP\_NAME // File mapping name

);

if (hMapFile.IsInvalid)

{

throw new Win32Exception();

}

...

}

finally

{

if (hMapFile != null)

{

// Close the file mapping object.

hMapFile.Close();

hMapFile = null;

}

}

### COM Interop

🗷 **Do not** force garbage collections with GC.Collect to release COM objects in performance sensitive APIs. A common approach for releasing COM objects is to set the RCW reference to null, and call System.GC.Collect followed by System.GC.WaitForPendingFinalizers. This is not recommended for performance reasons, because in many situations it can trigger the garbage collector to run too often. Code written by using this approach significantly compromises the performance and scalability of server applications. You should let the garbage collector determine the appropriate time to perform a collection.

🗹 **You should** use Marshal.FinalReleaseComObject or Marshal.ReleaseComObject to manage the lifetime of an RCW manually. It has much better performance than forcing garbage collections with GC.Collect.

🗷 **Do not** make cross-apartment calls. When you call a COM object from a managed application, make sure that the managed code's apartment matches the COM object's apartment type. By using matching apartments, you avoid the thread switch associated with cross-apartment calls.