**.NET Coding Conventions:**

**Recommended Coding Guidelines and Standards**

**Table of Contents**

[1. Introduction 3](#_Toc95205365)

[2. Style 3](#_Toc95205366)

[2.1 Tabs and Indents 3](#_Toc95205367)

[2.2 Regions 3](#_Toc95205368)

[2.3 Braces 4](#_Toc95205369)

[2.4 Single Line Statements 5](#_Toc95205370)

[2.5 Spacing 5](#_Toc95205371)

[3. Naming 6](#_Toc95205372)

[3.1 General Guidelines 6](#_Toc95205373)

[3.2 Conventions 6](#_Toc95205374)

[3.3 Interop Classes 7](#_Toc95205375)

[4. Commenting 8](#_Toc95205376)

[5. File Organization 9](#_Toc95205377)

# 1. Introduction

Before reading this document, please familiarize yourself with the [.NET Framework Design Guidelines](http://msdn.microsoft.com/en-us/library/ms229042.aspx) (MSDN) for the Microsoft recommended guidelines for working with .NET. Most recommended naming conventions, casing rules, and similar, are detailed in that document. The standards herein are intended to compliment and augment the framework design guidelines to provide a foundation for code consistency, whether it was authored by a single developer or many. Please remember, the principal focus should always be the production of code that is clean, legible, and easily maintainable. When performing maintenance on an existing code base, please be sure to adopt the style of the existing code.

These guidelines are based on, and borrow heavily from, the coding standards used by Microsoft’s CLR and .NET Framework development group. Those guidelines were posted by the group’s Program Manager, Brad Abrams, on this blog. <http://blogs.msdn.com/brada/articles/361363.aspx>

# Style

## 2.1 Tabs and Indents

Tab characters should not be used in code. All indentation should be done with 2 space characters.

## 2.2 Regions

Classes may contain region blocks that segment the code into functional areas. Some examples would be “Published Event Definitions,” “Class Data Definitions,” “Constructor Definitions,” “Event Handler Definitions,” and “IDataProvider Implementation.” All explicit interface implementations should be enclosed in a matching region block. Please avoid use of nested regions or region blocks that have semantics other than the description of functional areas. Also, be aware that regions have the potential to make it more difficult to realize that a class has grown too large and complex. Please use them judiciously.

## 2.3 Braces

Open braces should always be at the beginning of the line following the statement that begins the block. Contents of the brace should be indented by 2 spaces. Braces should never be considered optional. Even for single statement blocks, braces should always be used. This increases code readability and maintainability. All control structures (if, while, for, etc.) should use braces. Additionally, case statements should be indented from the switch statement. For example:

if (someExpression)

{

  DoSomething();

}

else

{

  DoSomethingElse();

}

switch (someExpression)

{

  case 0:

    DoSomething();

    break;

  case 1:

    DoSomethingElse();

    break;

  case 2:

    int n = 1;

    DoAnotherThing(n);

    break;

}

for (int i = 0; i < 100; i++)

{

  DoSomething(i);

}

## 2.4 Single Line Statements

In general, forcing statements onto a single line is discouraged, as it hinders the readability of the code, and increases code density. If a statement normally requiring braces is used on a single line, its braces should also begin and end on the same line. For example, the following is acceptable:

public class Foo

{

  public int Bar { get; set; }

}

if (someExpression) { DoSomething(); }

However, it is preferred that the statements be expanded for consistency and readability.

public class Foo

{

  public int Bar

{

get;

set;

}

}

if (someExpression)

{

  DoSomething();

}

## 2.5 Spacing

Spaces improve readability by decreasing code density. The following guidelines define the expectations for the use of space characters within code:

* **Do** use a single space after a comma between function arguments:  
  Right: Console.In.Read(myChar, 0, 1);  
  Wrong:       Console.In.Read(myChar,0,1);
* **Do** use a single space before flow control statements:  
  Right: while (x == y)  
  Wrong: while(x==y)
* **Do** use a single space before and after comparison operators:  
  Right: if (x == y)  
  Wrong: if (x==y)
* **Do not** use a space after the parenthesis and function arguments:  
  Right: CreateFoo(myChar, 0, 1)  
  Wrong: CreateFoo( myChar, 0, 1 )
* **Do not** use spaces between a function name and parenthesis:  
  Right: CreateFoo()  
  Wrong: CreateFoo ()
* **Do not** use spaces inside brackets:  
  Right: x = dataArray[index];  
  Wrong: x = dataArray[ index ];

# Naming

## 3.1 General Guidelines

Except where noted otherwise, follow all [.NET Framework Design Guidelines](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconnamingguidelines.asp) for both internal and external members. Applicable highlights of these include:

* **Do not** use Hungarian notation.
* **Do** use camelCasing for parameters.
* **Do** use camelCasing for local variables.
* **Do** use PascalCasing for function, property, event, and class names.
* **Do** prefix interfaces names with “I”.
* **Do not** prefix enums, classes, or delegates with any letter.

## 3.2 Conventions

* Property backing variables should use camelCasing and be prefixed with an underscore “\_“.
* Constants should be declared in uppercase and use an underscore “\_” to separate individual words.
* No prefixes should be used to decorate instance or static variables. If there is a need to explicitly reference the variable in a way that describes its scope, use the notations “this.variable” or “TypeName.variable”.
* Type name conflicts should be fully controlled through the use of namespaces. Just as Hungarian notation is discouraged for variable naming, it is also discouraged in type naming. For example, a tree view user control should be called “TreeView”, not “ucTreeView.”

// Property variable

private string \_firstName = null;

// Instance variable

private string baseCaption = "Base Caption";

// Static variable

private static int itemCount = 0;

// Constant

private const string ROOT\_NODE = "Root";

## 3.3 Interop Classes

Classes serving as interop wrappers (DllImport statements) should follow the following naming convention:

* **NativeMethods** – No suppress unmanaged code attribute, these methods can be used anywhere because a stack walk will be performed.
* **UnsafeNativeMethods** – Has suppress unmanaged code attribute. These methods are potentially dangerous and any *caller* of these methods must do a full security review to ensure that the usage is safe and protected as no stack walk will be performed.
* **SafeNativeMethods** – Has suppress unmanaged code attribute. These methods are safe and can be used fairly safely and the caller isn’t needed to do full security reviews even though no stack walk will be performed.

class NativeMethods

{

  private NativeMethods()

  {

  }

  [DllImport(“user32”)]

  internal static extern void FormatHardDrive(string driveName);

}

[SuppressUnmanagedCode()]

class UnsafeNativeMethods

{

  private UnsafeNativeMethods()

  {

  }

  [DllImport(“user32”)]

  internal static extern void CreateFile(string fileName);

}

[SuppressUnmanagedCode()]

class SafeNativeMethods

{

  private SafeNativeMethods() {}

  [DllImport(“user32”)]

  internal static extern void MessageBox(string text);

}

All interop classes **must** be private, and all methods must be **internal**. In addition a private constructor should be provided to prevent instantiation.

# Commenting

Comments should be used to describe intention, algorithmic overview, and/or logical flow.  Ideally, by reading the comments alone, someone other than the author could understand the code’s intended behavior and general operation. While there are no minimum comment requirements and certainly some very small routines need no commenting at all, it is hoped that most routines will have comments reflecting the developer’s intent and approach. The // (two slashes) style of comment tags should be used in most situations. Avoid using /// (three slashes) in your comments, except when using XML commenting structure, as doing so can confuse the XML parser/generator. Where ever possible, place comments above the code instead of beside it.

XML commenting should be employed, utilizing the [MSDN recommended](http://msdn.microsoft.com/en-us/library/5ast78ax.aspx) tags. The goal is to be able to produce MSDN style documentation from the comments, using Sandcastle, or another documentation generator. This will help to make the documentation more structured and readily available, and will ease the chore of maintenance for those unfamiliar with the code base. In addition to the XML commenting, all closing braces for namespaces, classes, methods, and properties should be decorated with a comment remarking that fact.

# File Organization

* Source files should contain only one public type, although multiple internal classes are allowed.
* Source files should be given the name of the public class in the file. Just as Hungarian notation is discouraged for variable and type naming, it is also discouraged in file naming. For example, a tree view web user control should be called “TreeView.ascx”, not “ucTreeView.ascx.”
* Using statements should be the first entry in the source file, preceding the namespace declaration.
* Enumerations that apply to the project space should all be defined within a dedicated file, not included within the definition of another type. It is permissible to group all enumerations together in a single file, such as one titled “Enumerations.cs”.