

Coding Guidelines - .NET

# List of contents

[List of contents 2](#_Toc103168308)

[Formatting and Style 3](#_Toc103168309)

[Using Libraries 3](#_Toc103168310)

[Variable Declaration and Initializations 3](#_Toc103168311)

[Implicit Typing 3](#_Toc103168312)

[Function Declarations and Calls 3](#_Toc103168313)

[Statements 4](#_Toc103168314)

[Enums 4](#_Toc103168315)

[Whitespace 4](#_Toc103168316)

[Braces 4](#_Toc103168317)

[Comments 6](#_Toc103168318)

[What Constitutes a "Bad" Comment 6](#_Toc103168319)

[Regions 8](#_Toc103168320)

[Files and Structure 8](#_Toc103168321)

[General Naming Conventions 8](#_Toc103168322)

[Arrays and Collections 10](#_Toc103168327)

[Fields 10](#_Toc103168328)

[Properties 10](#_Toc103168329)

[Methods 10](#_Toc103168330)

[Method Input arguments 10](#_Toc103168331)

[Short Methods 11](#_Toc103168332)

[Member Overloading 11](#_Toc103168333)

[Static Classes 11](#_Toc103168334)

[Exception Throwing 11](#_Toc103168335)

[Exception Handling 11](#_Toc103168336)

[Try-Finally Block 11](#_Toc103168337)

[Class structure 12](#_Toc103168338)

# Formatting and Style

1. Always use 4 spaces instead of a tab for indentation;
2. There is no limit on the length of a line of code. Breaking lines arbitrarily to meet a certain column count is unnecessary.

# Using Libraries

1. Never include unnecessary libraries/assemblies (whether at the project level or in source code);
2. Prefer NuGet Packages – avoid committing binaries/unnecessary artifacts to a project unless absolutely necessary.

# Variable Declaration and Initializations

1. Always declare local variables "in the minimum scope block than can contain them, typically just before use.";
2. Prefer to initialize variables when declared;
3. Prefer to initialize the variable on the same line it is declared;
4. Never declare more than one variable on a single line.

Good:

*int xAxis;*

*int yAxis;*

Bad:

*int xAxis, yAxis;*

# Implicit Typing

Prefer implicit typing (var)

Good:

*var message = new Message();*

Bad:

*Message message = new Message();*

# Function Declarations and Calls

Always order parameters in a function as follows:

1. incoming parameters first;
2. in or ref parameters next;
3. out parameters next;
4. optional parameters last.

# Statements

Never "put more than one statement on a single line."

Bad:

*xAxis = 5; yAxis = 10;*

# Enums

1. Always "use an enum to strongly type parameters, properties, and return values that represent sets of values";
2. Prefer using an enum over static constants.

# Whitespace

1. Never use more than one consecutive blank line within a method;
2. Always include exactly one blank line between methods and no more or less;
3. Never use a space between the function name and its parenthesis.

Good:

*Method(x, y, z)*

Bad:

*Method (x, y, z)*

1. Always use a single space after a comma

Good:

*Method(x, y, z)*

Bad:

*Method(x,y,z)*

# Braces

1. Always use Allman bracing 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:

*public int ExampleFunction(int index)*

*{*

*if (index < 0)*

*{*

*return index;*

*}*

*else*

*{*

*var result = 0;*

*for (var currentIndex = 0; currentIndex < index; currentIndex++)*

*{*

*result += ExampleFunction2(currentIndex);*

*}*

*return result;*

*}*

*}*

Bad:

*if (index < 0) { return index; }*

*if (index < 0) {*

*return index; }*

Exceptions:

1. Simple initializers may be placed on one line;

Good:

*new int[] {1, 2}*

1. If using the simple form of get/set, always place on the same line as follows:

Good:

*PropertyName { get; set; }*

1. Always use braces around single line conditionals

Good:

*if (condition) { body; }*

# Comments

1. Prefer "//" style comments over "/\* \*/" comments
2. Prefer "comments at the same level as the code they describe"

Good:

*// Example comment.*

*DoSomething();*

Bad:

*// Example comment.*

*DoSomething();*

1. Prefer comments with "full sentences with initial caps, a terminating period and proper punctuation and spelling in comments"
2. Never use trailing inline comments (comments on the same line as the actual code)

Good:

*// Example comment.*

*DoSomething();*

Bad:

*DoSomething(); // Example comment.*

1. Always include a preceding comment describing why a piece of code is commented out

Good:

*// Removing this line for testing - should not be checked in still*

*// commented*

*// DoSomething();*

1. Prefer deleting code over commenting it out
2. TODO comments always require a valid Git task number embedded within the comment (right after the word "TODO")

Good:

*// TODO Task-667 Re-enable the following line once Task-665*

*// completes*

*// DoSomething();*

# What Constitutes a "Bad" Comment

The following are guidelines for what constitutes a "bad" comment (Sourced from Code Complete and Clean Code books):

1. An empty comment (a comment where the only contents are whitespace or templated text) - whether in whole or in part. The following are examples of empty comments (with the portion to be removed/corrected underlined):

*//*

*///*

*/// <summary>*

*///*

*/// </summary>*

*/// <summary>*

*/// Calculates the mileage of the kilometer entry at the given index*

*/// </summary>*

*/// <param name="index"></param>*

1. A "noise" comment (As per Clean Code: A Handbook of Agile Software Craftmanship) - comments that "restate the obvious and provide no new information". Example (with the portion to be removed/corrected underlined):

*/// Default Constructor*

*public MyObject() {*

*/// The day of the week*

*public int dayOfTheWeek = 0*

*/// Implements a process for calculating the number of seconds between*

*/// two dates*

*public long CalculateSecondsBetween(Datetime from, Datetime to)*

1. A misleading or incorrect comment - a comment that is either wrong or inaccurate in a way that would lead to miscommunication.

*/// Returns true if the "vacationDate" is anywhere in between the "to" /// and the "from"*

*public bool IsWithinPeakSeason(DateTime vacationDate, DateTime from, DateTime to)*

*{*

*return vacationDate >= from && vacationDate < to;*

*}*

Note that it is true if it inclusive of the from but exclusive of the to (you could also make further arguments regarding the clarity of it...)

1. A journal comment - a comment that summarizes the successive changes to a file
2. Commented out code - code left in but commented out should almost always be removed except in rare circumstances where it should be preceded by an explanation of why it is commented out (e.g. must be justified).

*var a = b + c;*

*// a;*

*return a;*

1. Redundant or "repeat of the code" comment - a comment that merely restates what the code is doing in its entirety in other words.

*/// Sums a + b then multiplies by c*

*public int Calculate(a, b, c)*

*{*

*return (a + b) \* c;*

*}*

# Regions

1. Never use regions within a method (refactor the code to make it more readable if necessary)
2. Prefer not using regions

# Files and Structure

1. Never "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"
2. Always "name the source file with the name of the public type it contains"

# General Naming Conventions

## Casing

Always use PascalCasing EXCEPT for the following cases (where camelCasing should be used):

* Non-public Field
* Constants
* Parameters
* Variables

## Extensions

Extension class always ends with 'Extension'.

Good:

*public static DateTimeOffsetExtension*

*{*

*}*

Bad:

*public static DateTimeOffsetHelper*

*{*

*}*

## Additional Rules

1. Always use meaningful names for types, functions, etc.
2. Never use "underscores, hyphens, or any other non-alphanumeric characters"
3. Never use contractions or abbreviations in names except the following (to be extended):
4. IP - InternetProtocol
5. Always prefix an interface with "I"
6. Always prefix a generic type parameter with "T"
7. Never use "the same name for a namespace and a type in that namespace"
8. Always "name flag enums with plural nouns or noun phrases and simple enums with singular nouns or noun phrases."
9. Always "name collection properties with a plural phrase describing the items in the collection"
10. Prefer prefixing booleans with "Is", "Are", "Has" and "Can"
11. Prefer booleans that indicate the positive sense as opposed to negative (e.g. prefer "IsEmpty" instead of "IsNotEmpty")
12. Always add "the suffix 'EventHandler' to names of delegates that are used in events."
13. Always add "the suffix 'Callback' to names of delegates other than those used as event handlers."
14. Never add "the suffix "Delegate" to a delegate."
15. Never use Hungarian notation (or otherwise encoding a variable type within its name)
16. Never use single letter variable names except for the following:

Good:

*policy => policy.Id*

Bad:

*p => p.Id*

1. in for-loops, where you can use "i, j, k" for a counter/index so long as the use is within 5 lines of the declaration (excluding blank lines)
2. Prefer the format "MethodName\_StateUnderTest\_ExpectedBehavior" when naming test cases OR"Test[Feature being tested]" for tests that do not easily fit in (a) (e.g. integration tests)

Good:

*SendMessageToProcessor\_WhenMissingInformation\_ShouldThrowException*

*TestMessageSending*

## Constants

1. Always "use constant fields for constants that will never change";
2. Always "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".

# Arrays and Collections

1. Prefer collections over arrays in public interfaces
2. Prefer returning an empty array or collection instead of a null reference

# Fields

1. Never assign mutable types to read-only fields

Bad:

*private readonly StringBuilder buffer = new StringBuilder()*

# Properties

1. Always provide reasonable default values for properties
2. Disfavor throwing exceptions in property getters

# Methods

1. Always add the suffix "Async" "to the names of methods that have an Async or async modifier" (As specified in[https://msdn.microsoft.com/en-us/library /hh191443.aspx#BKMK\_NamingConvention](https://msdn.microsoft.com/en-us/library/hh191443.aspx#BKMK_NamingConvention))
2. Disfavor nulls as valid return values
3. Always validate arguments passed to public or protected members as follows:
4. Method should return concrete type (not abstract type or interface).
5. It's better to use abstract input arguments (not concrete implementation).

# Method Input arguments

Max count of input arguments is 5, otherwise developer should create a class/record with all parameters to path arguments into the method.

There is some differences in method parameters between DAL, BLL (Managers) and Presentation layer.

# Short Methods

1. Use => in case method contains one line of code. Enter method code on next line.

Good:

*public void MyMethod() =>*

*someService.DoSomething();*

Bad:

*public void MyMethod() => someService.DoSomething();*

1. Do not use => in controllers. It's hard to read with action attributes.

# Member Overloading

1. Never vary parameter names arbitrarily in different versions of the method
2. Always place parameter names in the same position in all overloads where possible

# Static Classes

1. Disfavor static classes (their use needs to be justified)

# Exception Throwing

1. Disfavor using exceptions for the normal flow of control
2. Never explicitly throw exceptions from finally blocks

# Exception Handling

1. Disfavor catching nonspecific exceptions, such as System.Exception
2. Prefer an empty "throw" when catching and re-throwing an exception (preserves call stack whereas "throw ex" does not)
3. Prefer logging the exception (with stack trace) on the first handling of an exception (and try to avoid multiple logging of the same exception)

# Try-Finally Block

1. Always "use try-finally blocks for cleanup code and try-catch blocks for error recovery code". Never "use catch blocks for cleanup code."
2. Always use "using" statement to cleanup IDisposable objects

# Class structure

1. List all imports ("using") at the top of each class file, with no blank lines between them, and before any namespace declaration.
2. Keep imports tidy:

* Remove imports not in use

1. When describing a class, use the following structure

* Declare all class members, including readonly or constants
* Declare all public properties, including auto-properties
* Declare all constructor(s)
* Declare the destructor, if any
* Declare all public methods
* Declare all protected methods
* Declare all private methods