Coding guidelines

# Contents

[Contents 1](#_Toc253579822)

[1.1 Goal 3](#_Toc253579823)

[1.2 C# Rules 3](#_Toc253579824)

[1.2.1 Spacing 3](#_Toc253579825)

[1.2.2 Line Spacing 5](#_Toc253579826)

[1.2.3 Curly Brackets 6](#_Toc253579827)

[1.2.4 Documentation 8](#_Toc253579828)

[1.2.5 File Headers 8](#_Toc253579829)

[1.2.6 Element Order 9](#_Toc253579830)

[1.2.7 Keyword Order 10](#_Toc253579831)

[1.2.8 Class Members 11](#_Toc253579832)

[1.2.9 Naming 12](#_Toc253579833)

[1.2.10 Access Modifiers 13](#_Toc253579834)

[1.2.11 Comments 14](#_Toc253579835)

[1.2.12 Tabs 14](#_Toc253579836)

[1.2.13 Parenthesis 14](#_Toc253579837)

[1.2.14 Statements 15](#_Toc253579838)

[1.2.15 Method Parameter Placement 15](#_Toc253579839)

[1.2.16 Variables and Types 17](#_Toc253579840)

[1.2.17 Strings 17](#_Toc253579841)

[1.2.18 Locks 18](#_Toc253579842)

[1.2.19 Resource Management 18](#_Toc253579843)

[1.2.20 Tracing/Asserts 18](#_Toc253579844)

[1.2.21 Error Handling 20](#_Toc253579845)

[1.2.22 Platform Invoke (Interop) 21](#_Toc253579846)

[1.3 References 21](#_Toc253579847)

# Goal

* Create internal coding style for cooperation with team members.
* Improve code readability for easily understanding logic and design.
* Improve code maintenance for easily add code to implement new functionality.

# C# Rules

### Spacing

* **Do** place a space after the following keywords when used to begin a statement:

*if*

*for*

*foreach*

*while*

*do*

*switch*

*catch*

*throw*

*lock*

*using*

*return*

*yield*

*new*

*stackalloc*

*fixed*

Example: if (true) ...

* **Do not** place a space after the following keywords when used to begin a statement:

*checked*

*unchecked*

*sizeof*

*typeof*

*default*

Example: Type type = typeof(int);

* **Do not** place a space after the name of a *method* when performing a method-call.

Example: int i = this.MyMethod(true);

* **Do** place a single space after a comma or semicolon, but no space before.

Example: for (int i = 0; i < 2; i++) ...

* **Do** place a single space before and after all arithmetic and relational symbols.

Example: if (i < 0 && i + 2 > 3) ...

* **Do** place a single space before, but no space after, all unary logical symbols.

Example: bool x = !y;

Example: int x = -y;

* **Do not** place a space after an opening parenthesis, square bracket, or generic bracket.
* **Do not** place a space before a closing parenthesis, square bracket, or generic bracket.

Example: int i = ((int)value);

Example: int i = value[2];

Example: int i = new Dictionary<string, List<int>>();

* **Do** place a single space after an opening curly bracket.
* **Do** place a single space before a closing curly bracket.

Example: int i = new int[] { 1, 2, 3 };

* **Do** place a single space before and after a colon base-clause colon.

Example: public class MyClass : BaseClass

* **Do** place a single space before and after a where-clause colon.

Example: public class MyClass<T> : BaseClass where T : int

* **Do** place a single space after, but no space before, a label or case-colon.

Example: case 2: return false;

* **Do** place a single space after, but not before, a dereference or address-of symbol when used in a type declaration.

Example: int\* x;

* **Do** place a single space before, but not after, a dereference or address-of symbol when used outside of a type declaration.

Example: object x = \*y;

* **Do not** place spaces between dereference and address-of symbols when more than one is specified in a row.

Example: int\*\*\* x;

* **Do not** place a space before a nullable-type symbol.

Example: int? x;

* **Do not** place spaces on either side of a member access symbol.

Example: this.data.Value.Count

* **Do** place a single space after a single-line comment symbol or an Xml documentation symbol.

Example: // This is a comment.

Example: /// <summary>

* **Do not** place a space after a preprocessor symbol.

Example: #region

* **Do not** place more than one space in a row outside of comments and strings.

### Line Spacing

* **Do not** place a blank line after an opening curly bracket or before a closing curly bracket.
* **Do not** place a blank line before an opening curly bracket.

Example: if (value)

{

x += 1;

}

* **Do** place a blank line after a closing curly bracket when used to close a statement or element which spans multiple lines.

Example: if (value)

{

x += 1;

}

y = 2;

* **Do not** place a blank line before an *else, catch,* or *finally* keyword, or before the *while* keyword in a *do/while* statement.

Example: ...

}

else

{

x += 2;

}

* **Do not** place a blank line after a single-line comment or an Xml header.

Example: // Add the screen buffer.

width += buffer;

* **Do** place a blank line before an Xml header, unless the header is the first item in its scope.
* **Do** place a blank line before a single-line comment, unless it is immediately preceded by another single-line comment, or it is the first item in its scope.
* **Do not** place two or more blank lines back-to-back.

### Curly Brackets

* **Do not** omit curly brackets, even if the language allows it.

C# allows *if, while, for, foreach,* and *do* statements to be written with or without curly brackets. However, Require brackets to be used in all cases. This increases code readability and maintainability. For example:

if (value)

{

this.DoSomething();

}

* **Do not** place bracketed statements all on a single line.

Statements which contain bodies wrapped in curly-brackets should never be placed completely on a single line. For example, the following is not allowed:

if (value) { this.DoSomething(); }

* **Do** place accessors completely on a single line if the other accessors in the same property, indexer, or event are also completely on a single line.

The following code is not allowed, since one accessor is written completely on a single line, and the other is not. In this case, both accessors should be written across multiple lines:

public int Value

{

get { return this.value; }

set

{

if (this.check == true)

{

return 1;

}

else

{

return 2;

}

}

}

* **Do** place opening and closing curly brackets on a line by themselves.

Opening and closing curly brackets must not share a line with other code unless the entire statement is written on a single line. The following bracket style must be used:

if (value)

{

this.DoSomething();

}

The following are not allowed:

if (value) {

this.DoSomething();

}

if (value)

{

this.DoSomething(); }

This rule also applies to *do/while* statements, and *try/catch* statements, which must be written as:

do

{

this.DoSomething();

}

while (value != null);

and:

try

{

this.DoSomething();

}

catch (Exception)

{

this.DoSomethingElse();

}

### Documentation

* **Do** place an Xml documentation header above every element in the file and write descriptive help text for all parts of the header.

Visual Studio provides built-in mechanisms to convert Xml headers into SDK documentation. These headers are also used at design-time to provide Intellisense information. For example, consider the following static method, which writes a value to the registry:

public class RegistryUtils

{

/// <summary>

/// Adds or overwrites a value under the HKCU key.

/// </summary>

/// <param name="name">The path to the value.</param>

/// <param name="valueobj">The object to write.</param>

/// <returns>Returns true if the value was written, false if not.</returns>

public static bool SetValue(string name, object valueobj)

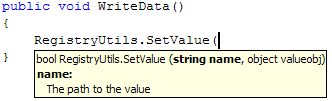
{

// write the value

}

}

When this method is called from elsewhere in the code, Visual Studio displays information in the form of a tooltip. The data in the tooltip is taken directly from the Xml header above the element:



### File Headers

* **Do** include a header at the top of every file C# file.

//---------------------------------------------------------------------------

//

// Copyright (C) Microsoft Corporation, 2007

//

// File: DatabaseHelpers.cs

//

// Contents: Components for wrapping common database access to the Roots

// table.

//

// Created by: waits 2007-11-13

//

//---------------------------------------------------------------------------

### Element Order

* **Do** place elements in a specific order, as defined below:

If the file contains a *namespace* element, this should be the first and only element at the root of the file. A file must not contain more than one *namespace* element.

The order of elements within a *namespace* element must be as follows. If the file does not contain a namespace element, this order applies at the root-level:

*Using directives*

*Enums*

*Interfaces*

*Structs*

*Classes*

Within a *class, interface,* or *struct*, elements must be written in the following order:

*Constants*

*Fields*

*Constructors*

*Destructors*

*Delegates*

*Events*

*Enums*

*Interfaces*

*Properties*

*Indexers*

*Methods*

*Structs*

*Sub-classes*

* **Do** order elements of the same type by their access, as follows:

*public*

*internal*

*protected internal*

*protected*

*private*

* **Do** place const and readonly elements before non-const, non-readonly elements of the same type.
* **Do** place static elements before non-static elements of the same type and access.
* **Do not** place more than one *class* within a file. Multiple *partial classes* of the same type may be placed in the same file. Multiple sub-classes may be placed within a parent class.
* **Consider** implementing interfaces within a separate *partial class*. This allows the interface members to be placed next to one another. For example:

/// <summary>

/// My special class.

/// </summary>

public partial class MyClass

{

// This class contains all of the custom code for MyClass.

}

/// <content>

/// Implements the IEnumerator interface.

/// </content>

public partial class MyClass : IEnumerator

{

public object Current

{

get { return null; }

}

public bool MoveNext()

{

return false;

}

public void Reset()

{

}

}

### Keyword Order

* **Do** place declaration keywords in a specific order, as defined below:

In an element declaration, the access modifier should always appear first. If the element is static, const, etc., these keywords should appear next, followed by the element type or name. For example, the following element declarations are correct:

private int customerAge;

private static int customerAge;

private static readonly int customerAge;

But the following is incorrect because the access modifier should come at the beginning of the declaration.

static private int customerAge;

* **Do** place the keyword *protected* before the keyword *internal* for elements with protected internal access.

### Class Members

* **Do** prefix calls to class members with *‘this.*’.

All calls made to other members of the same class, including *fields, properties, methods, indexers, events,* etc., must be prefixed by the token *this* to indicate that the referenced item lives within the same class. This also applies to members contained within base classes of the current class.

* **Do not** prefix calls to base class members with ‘*base.*’, unless a base class member is being called from an override of the same member in the local class.

Prefixing base-class members with ‘*base.*’ can potentially lead to maintainability problems. If the base-class member is later made virtual, and an override is created in the local class, the local override will not be called since the local call is explicitly prefixed by ‘*base.*’. For example:

public class BaseClass

{

protected void BaseMethod()

{

}

}

public class DerivedClass : BaseClass

{

public void MyMethod()

{

base.BaseMethod();

}

}

If the code is changed to:

public class BaseClass

{

protected virtual void BaseMethod()

{

}

}

public class DerivedClass : BaseClass

{

public void MyMethod()

{

base.BaseMethod();

}

protected override void BaseMethod()

{

}

}

The local override of BaseMethod will not be called. Instead, the call to BaseMethod() should be prefixed with ‘*this.*’ to indicate that it part of the local instance.

The ‘*base.*’ prefix may only be used within an overriden method or property, when calling the base implementation of that same method or property. For example:

public class BaseClass

{

protected virtual void BaseMethod()

{

}

}

public class DerivedClass : BaseClass

{

protected override void BaseMethod()

{

base.BaseMethod();

}

}

* **Do not** use fully qualified type names. Use the using statement instead.
* **Do not** put a using statement inside a namespace.
* **Do not** have multiple namespaces in the same file.
* **Do not** use extension methods unless there is no other way to achieve what you want to do. Notes:If you want to know what is extentsion methods, please refer to <http://msdn.microsoft.com/en-us/library/bb383977.aspx>
* **Do** generic methods should use T as the parameterized type.

### Naming

* **Do** use Pascal casing when naming constant *fields* and *variables*.

Example: string correctText = string.Empty;

* **Do** use Pascal casing when naming readonly *fields* and *variables* which are not declared *private*. Private readonly *fields* and *variables* may be cased with Pascal or camel casing.

Example:

public readonly string HostName;

private string sourceTypeField;

* **Do** use camel casing when naming non-constant, non-readonly *fields* and *variables*.
* **Do** use litter-case for proper noun when name should be camel casing.

Prop noun: URL, HTML, UI, FTP, IP,

Example:

private string uiText;

private string urlText;

* **Do** use Pascal casing when naming *public* or *internal* fields for proper noun except 2 letter proper noun.

Example:

public string UrlText;

public string UIText;

* **Do** use Pascal casing when naming *public* or *internal* fields.
* **Do** use pascal casing (beginning with an upper-case letter) when naming the following element types: *namespaces, classes, structs, enums, delegates, events, properties, methods.*
* **Do** suffix custom attribute classes with Attribute.
* **Do** suffix custom exception classes with Exception.
* **Do** name methods using verb-object pair, e.g. ShowDialog().
* **Do** method names with return values should have a name describing the value returned, e.g. GetObjectState().
* **Do** use descriptive variable names. Avoid single character variable names.
* **Do not** prefix *fields* in a *class* or *struct* with ‘*m\_*’, ‘s\_’ for private fields.

private statics\_sync;

private stringm\_field;

* **Do** begin the names of all *interfaces* with the capital letter ‘I’.
* **Do not** use Hungarian notation when naming *fields* and variables.

Example:

// Don't use Hungarian notation.

private string sSourceTypeField;

### Access Modifiers

* **Do** explicitly declare access modifiers on all elements.

Every element declaration should begin with an access modifier keyword; either *public, private, protected*, or *internal*. In C# it is legal to declare an element without an access modifier, in which case the protection of the element defaults to *private*. However, to maintain consistency and transparency in the code base, require access modifiers to be explicitly declared on all elements.

* **Do** set the access of all class fields to *private.* Use properties to expose fields outside of the *class*. Within a struct, fields may have any access level. Constant fields may have any access type.

### Comments

* **Do not** write “empty” single-line comments unless they fall within the middle of a block of comments.

Invalid: //

// The empty line above is invalid, but the empty line below is ok

// since it falls in the middle of this block of comments.

Valid: //

// The empty line below is also invalid.

Invalid: //

int x = 0;

* **Do not** write empty comments using the multi-line comment style. For example:

/\*

\*/

* **Do not** use the triple-slash comment style for single-line comments. This style is reserved for Xml documentation headers.
* **Do** use four slashes in a row when commenting out lines of code. This distinguishes out-commented code from descriptive comments. Four slashes must be used since three slashes are reserved for Xml documentation headers.

// This is a normal comment.

CsDocument csdocument = document as CsDocument;

Debug.Assert(csdocument != null);

if (csdocument != null)

{

////if (csdocument.RootElement != null)

////{

//// this.ProcessComments(csdocument.RootElement);

////}

}

### Tabs

* **Do not** use tabs for indentation, or anywhere else within a code document. Each level of indentation should consist of four spaces.

### Parenthesis

* **Do not** use parenthesis in places where they are unnecessary and do not increase readability.

In the following example, the parenthesis are unnecessary and should be removed:

return ((this.data).Value).Count;

### Statements

* **Do not** place more than one statement on a single line.
* **Do not** add an extra semicolon to the file, resulting in an empty statement.
* **Do**  format WQL query as below :

SELECT <columna>, <columnb>, ...

FROM <tablea> INNER JOIN

[indent]<tableb> ON <join conditiona> AND

[indent][intent] <join conditionb> OR

...

WHERE <conditiona> AND

[indent] <conditionb> OR

[indent] …

In this format, the parts of select, from and where are very clear.

### Method Parameter Placement

* **Do** place the opening parenthesis of a method call on the same line as the method name.
* **Do** place the opening square bracket of an array access on the same line as the item name.
* **Do** place the closing parenthesis of a method call on the same line as the last parameter. If there are no parameters, place the closing parenthesis on the same line as the opening parenthesis.
* **Do** place the closing square bracket of an array access on the same line as the last array index parameter.
* **Do** place a comma separating two method parameters on the same line as the previous method parameter.

* **Do** place all method parameters on the same line, or else place each parameter on its own line.
* **Do** place each parameter on the same line as the previous parameter (if all parameters are on the same line), or else on the line immediately following the previous parameter.
* **Do** place the first parameter on the line after the method name when the parameters are on separate lines.

allows method parameters to be laid out in one of three ways:

MyMethod(item1, item2, item3, item4);

MyMethod(

item1, item2, item3, item4);

MyMethod(

item1,

item2,

item3,

item4);

The following styles violate the above rule and are not allowed:

MyMethod(item1, item2,

item3, item4);

MyMethod(

item1, item2,

item3, item4);

MyMethod(item1,

item2,

item3,

item4);

MyMethod(

item1, item2, item3, item4);

MyMethod(

item1,

item2,

item3,

item4);

* **Do not** allow any parameter other than the first parameter to span across multiple lines.

The following method calls are legal because only the first parameter spans across multiple lines:

MyMethod(string.Format(

“Fill {0} the {1}”,

item1,

item2));

MyMethod(new int[]

{

1, 2, 3

});

However, the following call is illegal because the second and third parameter spans across multiple lines. Because of this, this method call is difficult to read.

MyMethod(

item1,

"This is a string " +

"which continues onto a second line",

new int[]

{

1, 2, 3

},

5);

The second and third parameters should be stored in local variables and then passed into the method call. This makes the call much more readable.

string text = "This is a string " +

"which continues onto a second line";

int[] values = new int[]

{

1, 2, 3

};

MyMethod(item1, text, values, 5);

### Variables and Types

* **Do** use C# predefined types rather than the aliases in the System namespace and the framework’s String type.

Example:

// Use C# predefined types.

string text = string.Empty;

// Don't use aliases in the System namespace.

String aliase = String.Empty;

* **Do** declare a local variable as close to possible to its first use.
* **Do** use the const directive only on natural constants such as the number of days of the week.
* **Do not** use const on read-only variables. For that, use the readonly directive.

Example: private const readonly int number;

* **Do not** provide explicit values for enums unless they are integer powers of 2.
* **Do not** specifying a specific type for an enum.

### Strings

* **Do** use String.Empty instead of “”.

Example:

// Use string.Empty.

string correctText = string.Empty;

// Don't use "".

string incorrectText = "";

* **Do** use StringBuilder, not string, When building a long string.

Example:

StringBuilder builder = new System.Text.StringBuilder();

foreach (char ch in stringInput)

{

char chThis = ch;

if (Char.IsLetter(chThis) == true)

{

chThis = Char.ToUpper(chThis);

}

builder.Append(chThis);

}

return builder.ToString();

### Locks

* **Do** only lock on a private or private static object.

Example: lock(m\_myVariable);

* **Do not** lock on the current object instance.

Example: lock(this);

### Resource Management

* **Do** implement the IDisposable interface on classes referencing external resources.
* **Do** call Close() or Dispose() on classes that offer it.
* **Do** wrap instantiation of IDisposable objects with a using statement to ensure that Dispose() is automatically called.

### Tracing/Asserts

* **Do** use TraceAssert when asserting.

Example:

public static BaseException Create(

Exception ex,

string operationString,

string objectString)

{

TraceAssert(Tracing.Source, ex != null, "Passed null exception");

…

}

* **Do** trace argument values when using TraceMethod.

Example:

public static string GetParameterValue(

string parameterName,

string defaultValue)

{

TraceAssert(Tracing.Source, !String.IsNullOrEmpty(parameterName));

TraceAssert(Tracing.Source, !String.IsNullOrEmpty(defaultValue));

using (TraceMethod(

Tracing.Source,

"parameterName={0}, defaultValue={1}",

parameterName,

defaultValue))

{

…

}

}

* **Do** trace all non-trivial methods.
* **Do** trace all exception throws and catches.

Example:

try

{

// load all the provider paths into the list for processing

foreach (string path in Properties.Settings.Default.DataProviderPaths)

{

m\_ProviderPaths.Add(path);

}

// initialization

ValidateConfiguration();

// process the files

ProcessInput();

}

catch (Exception ex)

{

// All errors in the process will come here and will be logged before the utility

// throws back to the command line

Logger.Write(

"Fatal exception in FeedbackGenerator. Program terminated." +

Environment.NewLine +

ex.ToString(),

LoggingCategory,

1,

(int) FeedBackGeneratorEvent.GeneralError,

TraceEventType.Critical);

throw;

}

* **Do** trace all locking operations.

Example:

lock (s\_initializationSyncRoot)

{

TraceAssert(

Tracing.Source,

s\_isInitialized,

"Shutdown called for unitiliazied TelemetryService");

…

}

### Error Handling

* **Do** catch only exceptions for which you have explicit handling.

Example:

try

{

PrefixMode = ( CanonUrlPrefixMode ) nativeUrl.PrefixMode;

}

catch( InvalidCastException ex )

{

Debug.WriteLine( "Invalid prefix reason found: " + ex.Message );

PrefixMode = CanonUrlPrefixMode.PREFIXMODE\_NO\_PREFIX\_MODE;

}

* **Do** throw the original exception (or another exception constructed from the original) to maintain the stack location of the original error in a catch statement that throws an exception.

Example:

catch (InvalidOperationException ex)

{

TraceUtility.TraceCatchException(Tracing.Source, ex);

string opStr = String.Format(

CultureInfo.InvariantCulture,

"Cannot connect to database server - connection string: {0}",

Adapter.Connection.ConnectionString);

// construct another exception from the original exception.

BaseException baseEx = BaseException.Create(

ex,

opStr,

"Database server: " + server);

TraceUtility.TraceThrowException(Tracing.Source, baseEx);

throw baseEx;

}

* **Do** provide a way to check preconditions before calling a member so that users can write code that does not throw exceptions.
* **Do** check preconditions before calling a member to avoid throwing exceptions.
* **Do** specify the inner exception when wrapping exceptions.
* **Do not** use error codes as return values.
* **Do not** define custom exception classes.
* **Do not** catch Exception class.

### Platform Invoke (Interop)

* **Do** use the original names for structs, constants, functions, enums, etc. Removal of Hungarian prefix is optional.
* **Do** specify CharSet when declaring a native entry point. Should always be Unicode unless the function is ansi-only.
* **Do** Specify SetLastError when declaring a native entry point if the API returns error information via GetLastError (use Marshal.GetLastWin32Error).
* **Do** Specify CallingConvention if the API does not use stdcall.
* **Do** declare native entities in a separate class, e.g. MsiNative, PeNative,...
* **Do** provide managed code wrappers for native APIs. These wrappers should encapsulate error handling semantics and resource management.
* **Do** correctly handle native resources such as file handles, network sockets, etc. These resources should be wrapped in native disposable objects, e.g. SafeHandle.

# References

If you want to know more, you can refer to link-address as below:

[\\mrs-ws-file\share\v-yapa\code guideline\ Reputation Services CSharp Coding Guidelines.docx](file://mrs-ws-file/share/v-yapa/code%20guideline/%20Reputation%20Services%20CSharp%20Coding%20Guidelines.docx)

[\\mrs-ws-file\share\v-yapa\code guideline\ MSCom Code Review Guidelines.doc](file://mrs-ws-file/share/v-yapa/code%20guideline/%20MSCom%20Code%20Review%20Guidelines.doc)

[\\mrs-ws-file\share\v-yapa\code guideline\ Microsoft Source Analysis User Guide.docx](file://mrs-ws-file/share/v-yapa/code%20guideline/%20Microsoft%20Source%20Analysis%20User%20Guide.docx)

[\\mrs-ws-file\share\v-yapa\code guideline\ DesignGuidelines.doc](file://mrs-ws-file/share/v-yapa/code%20guideline/%20DesignGuidelines.doc)