C# Coding Conventions

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **TEMPLATE CHANGE LOG** | | FOR QA DEPARTMENT USE ONLY | | | |
| *This Template Change Log shall be deleted by user after downloaded and saved as a document* | | | |
| **Revision** | **Description** | **Author** | **Date** | **Approved** | **Date** |
| 1.0 | First revision | Hoan Ng | 28-Jan | Hoan Ng | 28-Jan |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

Table of Contents

[I. Naming Conventions 3](#_Toc410253620)

[II. Layout Conventions 8](#_Toc410253621)

[Tabs & Indenting 9](#_Toc410253622)

[Bracing 9](#_Toc410253623)

[Single line statements 10](#_Toc410253624)

[Spacing 10](#_Toc410253625)

[File Organization 10](#_Toc410253626)

[III. Commenting 11](#_Toc410253627)

[Copyright notice 12](#_Toc410253628)

[Documentation Comments 12](#_Toc410253629)

[Comment Style 12](#_Toc410253630)

[IV. Language Guidelines 13](#_Toc410253631)

[String Data Type 13](#_Toc410253632)

[Implicitly Typed Local Variables 13](#_Toc410253633)

[Unsigned Data Type 14](#_Toc410253634)

[Arrays 14](#_Toc410253635)

[Delegates 15](#_Toc410253636)

[try-catch and using Statements in Exception Handling 15](#_Toc410253637)

[&& and || Operators 16](#_Toc410253638)

[New Operator 16](#_Toc410253639)

[Event Handling 17](#_Toc410253640)

[Static Members 17](#_Toc410253641)

[LINQ Queries 18](#_Toc410253642)

[Reference 20](#_Toc410253643)

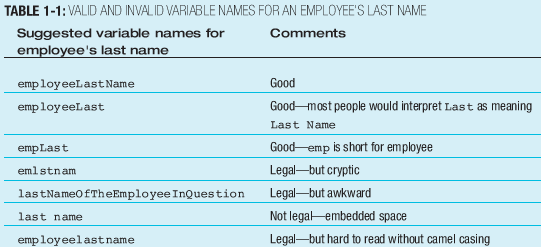
## [Naming Conventions](javascript:void(0))

In short examples that do not include [using directives](https://msdn.microsoft.com/en-us/library/sf0df423.aspx), use namespace qualifications. If you know that a namespace is imported by default in a project, you do not have to fully qualify the names from that namespace. Qualified names can be broken after a dot (.) if they are too long for a single line, as shown in the following example.

var currentPerformanceCounterCategory = new System.Diagnostics.

PerformanceCounterCategory();

Choose meaningful names for variables



You do not have to change the names of objects that were created by using the Visual Studio designer tools to make them fit other guidelines.

Do use **PascalCasing** for class names and method names.

public class ClientActivity

{

public void ClearStatistics()

{

//...

}

public void CalculateStatistics()

{

//...

}

}

do use **camelCasing** for method arguments and local variables.

public class UserLog

{

public void Add(LogEvent logEvent)

{

int itemCount = logEvent.Items.Count;

// ...

}

}

do not use **Hungarian** notation or any other type identification in identifiers

// Correct

int counter;

string name;

// Avoid

int iCounter;

string strName;

do not use **Screaming Caps** for constants or readonly variables

// Correct

public static const string ShippingType = "DropShip";

// Avoid

public static const string SHIPPINGTYPE = "DropShip";

avoid using **Abbreviations**. Exceptions: abbreviations commonly used as names,   
                 such as **Id, Xml, Ftp, Uri**

// Correct

UserGroup userGroup;

Assignment employeeAssignment;

// Avoid

UserGroup usrGrp;

Assignment empAssignment;

// Exceptions

CustomerId customerId;

XmlDocument xmlDocument;

FtpHelper ftpHelper;

UriPart uriPart;

do use **PascalCasing** for abbreviations 3 characters or more (2 chars are both uppercase)

HtmlHelper htmlHelper;

FtpTransfer ftpTransfer;

UIControl uiControl;

do not use **Underscores** in identifiers. Exception: you can prefix private static variables   
                    with an underscore.

// Correct

public DateTime clientAppointment;

public TimeSpan timeLeft;

// Avoid

public DateTime client\_Appointment;

public TimeSpan time\_Left;

// Exception

private DateTime \_registrationDate;

do use **predefined type names** instead of system type names like Int16, Single, UInt64, etc

// Correct

string firstName;

int lastIndex;

bool isSaved;

// Avoid

String firstName;

Int32 lastIndex;

Boolean isSaved;

. do use implicit type **var** for local variable declarations. Exception: primitive types (int, string,   
          double, etc) use predefined names.

var stream = File.Create(path);

var customers = new Dictionary();

// Exceptions

int index = 100;

string timeSheet;

bool isCompleted;

do use noun or noun phrases to name a class.

public class Employee

{

}

public class BusinessLocation

{

}

public class DocumentCollection

{

}

do prefix interfaces with the letter **I**.  Interface names are noun (phrases) or adjectives.

public interface IShape

{

}

public interface IShapeCollection

{

}

public interface IGroupable

{

}

do name source files according to their main classes. Exception: file names with partial classes  
          reflect their source or purpose, e.g. designer, generated, etc.

// Located in Task.cs

public partial class Task

{

//...

}

// Located in Task.generated.cs

public partial class Task

{

//...

}

do organize namespaces with a clearly defined structure

// Examples

namespace Company.Product.Module.SubModule

namespace Product.Module.Component

namespace Product.Layer.Module.Group

do vertically align curly brackets.

// Correct

class Program

{

static void Main(string[] args)

{

}

}

do declare all member variables at the top of a class, with static variables at the very top.

// Correct

public class Account

{

public static string BankName;

public static decimal Reserves;

public string Number {get; set;}

public DateTime DateOpened {get; set;}

public DateTime DateClosed {get; set;}

public decimal Balance {get; set;}

// Constructor

public Account()

{

// ...

}

}

do use singular names for enums. Exception: bit field enums.

// Correct

public enum Color

{

Red,

Green,

Blue,

Yellow,

Magenta,

Cyan

}

// Exception

[Flags]

public enum Dockings

{

None = 0,

Top = 1,

Right = 2,

Bottom = 4,

Left = 8

}

do notexplicitly specify a type of an enum or values of enums (except bit fields)

// Don't

public enum Direction : long

{

North = 1,

East = 2,

South = 3,

West = 4

}

// Correct

public enum Direction

{

North,

East,

South,

West

}

do not suffix enum names with Enum

// Don't

public enum CoinEnum

{

Penny,

Nickel,

Dime,

Quarter,

Dollar

}

// Correct

public enum Coin

{

Penny,

Nickel,

Dime,

Quarter,

Dollar

}

## [Layout Conventions](javascript:void(0))

Good layout uses formatting to emphasize the structure of your code and to make the code easier to read. Microsoft examples and samples conform to the following conventions:

* Use the default Code Editor settings (smart indenting, four-character indents, tabs saved as spaces). For more information, see [Options, Text Editor, C#, Formatting](https://msdn.microsoft.com/en-us/library/03864tbz.aspx).
* Write only one statement per line.
* Write only one declaration per line.
* If continuation lines are not indented automatically, indent them one tab stop (four spaces).
* Add at least one blank line between method definitions and property definitions.
* Use parentheses to make clauses in an expression apparent, as shown in the following code.

if ((val1 > val2) && (val1 > val3))

{

// Take appropriate action.

}

### Tabs & Indenting

Tab characters (\0x09) should not be used in code. All indentation should be done with 4 space characters.

### Bracing

Open braces should always be at the beginning of the line after the statement that begins the block. Contents of the brace should be indented by 4 spaces. For example:

if (someExpression)

{

DoSomething();

}

else

{

DoSomethingElse();

}

“case” statements should be indented from the switch statement like this:

switch (someExpression)

{

case 0:

DoSomething();

break;

case 1:

DoSomethingElse();

break;

case 2:

{

int n = 1;

DoAnotherThing(n);

}

break;

}

Braces should never be considered optional. Even for single statement blocks, you should always use braces. This increases code readability and maintainability.

for (int i=0; i<100; i++) { DoSomething(i); }

### Single line statements

Single line statements can have braces that begin and end on the same line.

public class Foo

{

int bar;

public int Bar

{

get { return bar; }

set { bar = value; }

}

}

It is *suggested* that all control structures (if, while, for, etc.) use braces, but it is not required.

### Spacing

Spaces improve readability by decreasing code density. Here are some guidelines 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 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 ];
* 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)

### 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

Directory names should follow the namespace for the class

For example, I would expect to find the public class “System.Windows.Forms.Control” in “System\Windows\Forms\Control.cs”…

Classes member should be **alphabetized**, and grouped into sections (Fields, Constructors, Properties, Events, Methods, Private interface implementations, Nested types)

Using statements should be inside the namespace declaration.

namespace MyNamespace

{

using System;

public class MyClass : IFoo

{

// fields

int foo;

// constructors

public MyClass() { … }

// properties

public int Foo { get { … } set { … } }

// events

public event EventHandler FooChanged { add { … } remove { … } }

// methods

void DoSomething() { … }

void FindSomethind() { … }

//private interface implementations

void IFoo.DoSomething() { DoSomething(); }

// nested types

class NestedType { … }

}

}

## Commenting

Comments should be used to describe intention, algorithmic overview, and/or logical flow. It would be ideal, if from reading the comments alone, someone other than the author could understand a function’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 programmer’s intent and approach.

* Place the comment on a separate line, not at the end of a line of code.
* Begin comment text with an uppercase letter.
* End comment text with a period.
* Insert one space between the comment delimiter (//) and the comment text, as shown in the following example.

C#

// The following declaration creates a query. It does not run

// the query.

* Do not create formatted blocks of asterisks around comments.

### Copyright notice

Each file should start with a copyright notice. To avoid errors in doc comment builds, you don’t want to use triple-slash doc comments, but using XML makes the comments easy to replace in the future. Final text will vary by product (you should contact legal for the exact text), but should be similar to:

//-----------------------------------------------------------------------  
// <copyright file="ContainerControl.cs" company="Microsoft">  
// Copyright (c) Microsoft Corporation. All rights reserved.  
// </copyright>  
//-----------------------------------------------------------------------

### Documentation Comments

All methods should use XML doc comments. For internal dev comments, the <devdoc> tag should be used.

public class Foo   
{

/// <summary>Public stuff about the method</summary>  
/// <param name=”bar”>What a neat parameter!</param>  
/// <devdoc>Cool internal stuff!</devdoc>  
///  
public void MyMethod(int bar) { … }

}

However, it is common that you would want to move the XML documentation to an external file – for that, use the <include> tag.

public class Foo   
{

/// <include file='doc\Foo.uex' path='docs/doc[@for="Foo.MyMethod"]/\*' />  
 ///  
 public void MyMethod(int bar) { … }

}

UNDONE§ there is a big doc with all the comment tags we should be using… where is that?

### Comment Style

The // (two slashes) style of comment tags should be used in most situations. Where ever possible, place comments above the code instead of beside it. Here are some examples:

// This is required for WebClient to work through the proxy  
GlobalProxySelection.Select = new WebProxy("http://itgproxy");

// Create object to access Internet resources  
//  
WebClient myClient = new WebClient();

Comments can be placed at the end of a line when space allows:

public class SomethingUseful   
{  
 private int itemHash; // instance member  
 private static bool hasDoneSomething; // static member  
}

## [Language Guidelines](javascript:void(0))

The following sections describe practices that the C# team follows to prepare code examples and samples.

### [String Data Type](javascript:void(0))

Use the + operator to concatenate short strings, as shown in the following code.

string displayName = nameList[n].LastName + ", " + nameList[n].FirstName;

To append strings in loops, especially when you are working with large amounts of text, use a [StringBuilder](https://msdn.microsoft.com/en-us/library/system.text.stringbuilder.aspx) object.

var phrase = "lalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";

var manyPhrases = new StringBuilder();

for (var i = 0; i < 10000; i++)

{

manyPhrases.Append(phrase);

}

//Console.WriteLine("tra" + manyPhrases);

### [Implicitly Typed Local Variables](javascript:void(0))

Use [implicit typing](https://msdn.microsoft.com/en-us/library/bb384061.aspx) for local variables when the type of the variable is obvious from the right side of the assignment, or when the precise type is not important.

// When the type of a variable is clear from the context, use var

// in the declaration.

var var1 = "This is clearly a string.";

var var2 = 27;

var var3 = Convert.ToInt32(Console.ReadLine());

Do not use [var](https://msdn.microsoft.com/en-us/library/bb383973.aspx) when the type is not apparent from the right side of the assignment.

// When the type of a variable is not clear from the context, use an

// explicit type.

int var4 = ExampleClass.ResultSoFar();

Do not rely on the variable name to specify the type of the variable. It might not be correct.

// Naming the following variable inputInt is misleading.

// It is a string.

var inputInt = Console.ReadLine();

Console.WriteLine(inputInt);

Avoid the use of var in place of [dynamic](https://msdn.microsoft.com/en-us/library/dd264741.aspx).

Use implicit typing to determine the type of the loop variable in [for](https://msdn.microsoft.com/en-us/library/ch45axte.aspx) and [foreach](https://msdn.microsoft.com/en-us/library/ttw7t8t6.aspx) loops.

The following example uses implicit typing in a for statement.

var syllable = "ha";

var laugh = "";

for (var i = 0; i < 10; i++)

{

laugh += syllable;

Console.WriteLine(laugh);

}

The following example uses implicit typing in a foreach statement.

foreach (var ch in laugh)

{

if (ch == 'h')

Console.Write("H");

else

Console.Write(ch);

}

Console.WriteLine();

### [Unsigned Data Type](javascript:void(0))

In general, use int rather than unsigned types. The use of int is common throughout C#, and it is easier to interact with other libraries when you use int.

### [Arrays](javascript:void(0))

Use the concise syntax when you initialize arrays on the declaration line.

// Preferred syntax. Note that you cannot use var here instead of string[].

string[] vowels1 = { "a", "e", "i", "o", "u" };

// If you use explicit instantiation, you can use var.

var vowels2 = new string[] { "a", "e", "i", "o", "u" };

// If you specify an array size, you must initialize the elements one at a time.

var vowels3 = new string[5];

vowels3[0] = "a";

vowels3[1] = "e";

// And so on.

### [Delegates](javascript:void(0))

Use the concise syntax to create instances of a delegate type.

// First, in class Program, define the delegate type and a method that

// has a matching signature.

// Define the type.

public delegate void Del(string message);

// Define a method that has a matching signature.

public static void DelMethod(string str)

{

Console.WriteLine("DelMethod argument: {0}", str);

}

// In the Main method, create an instance of Del.

// Preferred: Create an instance of Del by using condensed syntax.

Del exampleDel2 = DelMethod;

// The following declaration uses the full syntax.

Del exampleDel1 = new Del(DelMethod);

### [try-catch and using Statements in Exception Handling](javascript:void(0))

Use a [try-catch](https://msdn.microsoft.com/en-us/library/0yd65esw.aspx) statement for most exception handling.

static string GetValueFromArray(string[] array, int index)

{

try

{

return array[index];

}

catch (System.IndexOutOfRangeException ex)

{

Console.WriteLine("Index is out of range: {0}", index);

throw;

}

}

Simplify your code by using the C# [using statement](https://msdn.microsoft.com/en-us/library/yh598w02.aspx). If you have a [try-finally](https://msdn.microsoft.com/en-us/library/zwc8s4fz.aspx) statement in which the only code in the finally block is a call to the [Dispose](https://msdn.microsoft.com/en-us/library/system.idisposable.dispose.aspx) method, use a using statement instead.

// This try-finally statement only calls Dispose in the finally block.

Font font1 = new Font("Arial", 10.0f);

try

{

byte charset = font1.GdiCharSet;

}

finally

{

if (font1 != null)

{

((IDisposable)font1).Dispose();

}

}

// You can do the same thing with a using statement.

using (Font font2 = new Font("Arial", 10.0f))

{

byte charset = font2.GdiCharSet;

}

### [&& and || Operators](javascript:void(0))

To avoid exceptions and increase performance by skipping unnecessary comparisons, use [&&](https://msdn.microsoft.com/en-us/library/2a723cdk.aspx) instead of [&](https://msdn.microsoft.com/en-us/library/sbf85k1c.aspx) and [||](https://msdn.microsoft.com/en-us/library/6373h346.aspx) instead of [|](https://msdn.microsoft.com/en-us/library/kxszd0kx.aspx) when you perform comparisons, as shown in the following example.

Console.Write("Enter a dividend: ");

var dividend = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter a divisor: ");

var divisor = Convert.ToInt32(Console.ReadLine());

// If the divisor is 0, the second clause in the following condition

// causes a run-time error. The && operator short circuits when the

// first expression is false. That is, it does not evaluate the

// second expression. The & operator evaluates both, and causes

// a run-time error when divisor is 0.

if ((divisor != 0) && (dividend / divisor > 0))

{

Console.WriteLine("Quotient: {0}", dividend / divisor);

}

else

{

Console.WriteLine("Attempted division by 0 ends up here.");

}

### [New Operator](javascript:void(0))

Use the concise form of object instantiation, with implicit typing, as shown in the following declaration.

var instance1 = new ExampleClass();

The previous line is equivalent to the following declaration.

ExampleClass instance2 = new ExampleClass();

Use object initializers to simplify object creation.

// Object initializer.

var instance3 = new ExampleClass { Name = "Desktop", ID = 37414,

Location = "Redmond", Age = 2.3 };

// Default constructor and assignment statements.

var instance4 = new ExampleClass();

instance4.Name = "Desktop";

instance4.ID = 37414;

instance4.Location = "Redmond";

instance4.Age = 2.3;

### [Event Handling](javascript:void(0))

If you are defining an event handler that you do not need to remove later, use a lambda expression.

public Form2()

{

// You can use a lambda expression to define an event handler.

this.Click += (s, e) =>

{

MessageBox.Show(

((MouseEventArgs)e).Location.ToString());

};

}

// Using a lambda expression shortens the following traditional definition.

public Form1()

{

this.Click += new EventHandler(Form1\_Click);

}

void Form1\_Click(object sender, EventArgs e)

{

MessageBox.Show(((MouseEventArgs)e).Location.ToString());

}

### [Static Members](javascript:void(0))

Call [static](https://msdn.microsoft.com/en-us/library/98f28cdx.aspx) members by using the class name: ClassName.StaticMember. This practice makes code more readable by making static access clear. Do not qualify a static member defined in a base class with the name of a derived class. While that code compiles, the code readability is misleading, and the code may break in the future if you add a static member with the same name to the derived class.

### [LINQ Queries](javascript:void(0))

Use meaningful names for query variables. The following example uses seattleCustomers for customers who are located in Seattle.

var seattleCustomers = from cust in customers

where cust.City == "Seattle"

select cust.Name;

Use aliases to make sure that property names of anonymous types are correctly capitalized, using Pascal casing.

var localDistributors =

from customer in customers

join distributor in distributors on customer.City equals distributor.City

select new { Customer = customer, Distributor = distributor };

Rename properties when the property names in the result would be ambiguous. For example, if your query returns a customer name and a distributor ID, instead of leaving them as Name and ID in the result, rename them to clarify that Name is the name of a customer, and ID is the ID of a distributor.

var localDistributors2 =

from cust in customers

join dist in distributors on cust.City equals dist.City

select new { CustomerName = cust.Name, DistributorID = dist.ID };

Use implicit typing in the declaration of query variables and range variables.

var seattleCustomers = from cust in customers

where cust.City == "Seattle"

select cust.Name;

Align query clauses under the [from](https://msdn.microsoft.com/en-us/library/bb383978.aspx) clause, as shown in the previous examples.

Use [where](https://msdn.microsoft.com/en-us/library/bb311043.aspx) clauses before other query clauses to ensure that later query clauses operate on the reduced, filtered set of data.

var seattleCustomers2 = from cust in customers

where cust.City == "Seattle"

orderby cust.Name

select cust;

* Use multiple from clauses instead of a [join](https://msdn.microsoft.com/en-us/library/bb311040.aspx) clause to access inner collections. For example, a collection of Student objects might each contain a collection of test scores. When the following query is executed, it returns each score that is over 90, along with the last name of the student who received the score.

// Use a compound from to access the inner sequence within each element.

var scoreQuery = from student in students

from score in student.Scores

where score > 90

select new { Last = student.LastName, score };

## Reference

[1] <http://www.dofactory.com/reference/csharp-coding-standards>

[2] <http://blogs.msdn.com/b/brada/archive/2005/01/26/361363.aspx>

[3] <https://msdn.microsoft.com/en-us/library/ff926074.aspx>