C#代码规范

可以直接看下面这两个网站内容

（<http://www.dofactory.com/reference/csharp-coding-standards>）

（<https://msdn.microsoft.com/en-us/library/ff926074.aspx>）

1. douse **PascalCasing** for class names and method names.
2. public class ClientActivity
3. {
4. public void ClearStatistics()
5. {
6. //...
7. }
8. public void CalculateStatistics()
9. {
10. //...
11. }
12. }

**Why**: consistent with the Microsoft's .NET Framework and easy to read.

1. douse **PascalCasing** for abbreviations 3 characters or more (2 chars are both uppercase)
2. HtmlHelper htmlHelper;
3. FtpTransfer ftpTransfer;
4. UIControl uiControl;

**Why**: consistent with the Microsoft's .NET Framework. Caps would grap visually too much attention.

1. douse **camelCasing** for method arguments and local variables.
2. public class UserLog
3. {
4. public void Add(LogEvent logEvent)
5. {
6. int itemCount = logEvent.Items.Count;
7. // ...
8. }
9. }

**Why**: consistent with the Microsoft's .NET Framework and easy to read.

1. douse **predefined type names** instead of system type names like Int16, Single, UInt64, etc
2. // Correct
3. string firstName;
4. int lastIndex;
5. bool isSaved;
7. // Avoid
8. String firstName;
9. Int32 lastIndex;
10. Boolean isSaved;

**Why**: consistent with the Microsoft's .NET Framework and makes code more natural to read.

1. douse implicit type **var** for local variable declarations. Exception: primitive types (int, string, double, etc) use predefined names.
2. var stream = File.Create(path);
3. var customers = new Dictionary();
5. // Exceptions
6. int index = 100;
7. string timeSheet;
8. bool isCompleted;

**Why**: removes clutter, particularly with complex generic types. Type is easily detected with Visual Studio tooltips.

1. douse noun or noun phrases to name a class.
2. public class Employee
3. {
4. }
5. public class BusinessLocation
6. {
7. }
8. public class DocumentCollection
9. {
10. }

**Why**: consistent with the Microsoft's .NET Framework and easy to remember.

1. doprefix interfaces with the letter **I**.  Interface names are noun (phrases) or adjectives.
2. public interface IShape
3. {
4. }
5. public interface IShapeCollection
6. {
7. }
8. public interface IGroupable
9. {
10. }

**Why**: consistent with the Microsoft's .NET Framework.

1. doname source files according to their main classes. Exception: file names with partial classes reflect their source or purpose, e.g. designer, generated, etc.
2. // Located in Task.cs
3. public partial class Task
4. {
5. //...
6. }
7. // Located in Task.generated.cs
8. public partial class Task
9. {
10. //...
11. }

**Why**: consistent with the Microsoft practices. Files are alphabetically sorted and partial classes remain adjacent.

1. doorganize namespaces with a clearly defined structure
2. // Examples
3. namespace Company.Product.Module.SubModule
4. namespace Product.Module.Component
5. namespace Product.Layer.Module.Group

**Why**: consistent with the Microsoft's .NET Framework. Maintains good organization of your code base.

1. dovertically align curly brackets.
2. // Correct
3. class Program
4. {
5. static void Main(string[] args)
6. {
7. }
8. }

**Why**: Microsoft has a different standard, but developers have overwhelmingly preferred vertically aligned brackets.

1. dodeclare all member variables at the top of a class, with static variables at the very top.
2. // Correct
3. public class Account
4. {
5. public static string BankName;
6. public static decimal Reserves;
8. public string Number {get; set;}
9. public DateTime DateOpened {get; set;}
10. public DateTime DateClosed {get; set;}
11. public decimal Balance {get; set;}
13. // Constructor
14. public Account()
15. {
16. // ...
17. }
18. }

**Why**: generally accepted practice that prevents the need to hunt for variable declarations.

1. douse singular names for enums. Exception: bit field enums.
2. // Correct
3. public enum Color
4. {
5. Red,
6. Green,
7. Blue,
8. Yellow,
9. Magenta,
10. Cyan
11. }
13. // Exception
14. [Flags]
15. public enum Dockings
16. {
17. None = 0,
18. Top = 1,
19. Right = 2,
20. Bottom = 4,
21. Left = 8
22. }

**Why**: consistent with the Microsoft's .NET Framework and makes the code more natural to read. Plural flags because enum can hold multiple values (using bitwise 'OR').

1. do notuse **Hungarian** notation or any other type identification in identifiers
2. // Correct
3. int counter;
4. string name;
6. // Avoid
7. int iCounter;
8. string strName;

**Why**: consistent with the Microsoft's .NET Framework and Visual Studio IDE makes determining types very easy (via tooltips). In general you want to avoid type indicators in any identifier.

1. do notuse **Screaming Caps** for constants or readonly variables
2. // Correct
3. public static const string ShippingType = "DropShip";
5. // Avoid
6. public static const string SHIPPINGTYPE = "DropShip";

**Why**: consistent with the Microsoft's .NET Framework. Caps grap too much attention.

1. do notuse **Underscores** in identifiers. Exception: you can prefix private static variables with an underscore.
2. // Correct
3. public DateTime clientAppointment;
4. public TimeSpan timeLeft;
6. // Avoid
7. public DateTime client\_Appointment;
8. public TimeSpan time\_Left;
10. // Exception
11. private DateTime \_registrationDate;

**Why**: consistent with the Microsoft's .NET Framework and makes code more natural to read (without 'slur'). Also avoids underline stress (inability to see underline).

1. avoidusing **Abbreviations**. Exceptions: abbreviations commonly used as names, such as**Id, Xml, Ftp, Uri**
2. // Correct
3. UserGroup userGroup;
4. Assignment employeeAssignment;
6. // Avoid
7. UserGroup usrGrp;
8. Assignment empAssignment;
10. // Exceptions
11. CustomerId customerId;
12. XmlDocument xmlDocument;
13. FtpHelper ftpHelper;
14. UriPart uriPart;

**Why**: consistent with the Microsoft's .NET Framework and prevents inconsistent abbreviations.

1. do notexplicitly specify a type of an enum or values of enums (except bit fields)
2. // Don't
3. public enum Direction : long
4. {
5. North = 1,
6. East = 2,
7. South = 3,
8. West = 4
9. }
11. // Correct
12. public enum Direction
13. {
14. North,
15. East,
16. South,
17. West
18. }

**Why**: can create confusion when relying on actual types and values.

1. do notsuffix enum names with Enum
2. // Don't
3. public enum CoinEnum
4. {
5. Penny,
6. Nickel,
7. Dime,
8. Quarter,
9. Dollar
10. }
12. // Correct
13. public enum Coin
14. {
15. Penny,
16. Nickel,
17. Dime,
18. Quarter,
19. Dollar
20. }

**Why**: consistent with the Microsoft's .NET Framework and consistent with prior rule of no type indicators in identifiers.

C# Coding Conventions (C# Programming Guide)

（<https://msdn.microsoft.com/en-us/library/ff926074.aspx>）

The [C# Language Specification](http://go.microsoft.com/fwlink/?LinkId=199552)  does not define a coding standard. However, the guidelines in this topic are used by Microsoft to develop samples and documentation.

Coding conventions serve the following purposes:

* They create a consistent look to the code, so that readers can focus on content, not layout.
* They enable readers to understand the code more quickly by making assumptions based on previous experience.
* They facilitate copying, changing, and maintaining the code.
* They demonstrate C# best practices.

[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();

* 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.

[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.

}

[Commenting Conventions](javascript:void(0))

* 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.

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

// the query.

* Do not create formatted blocks of asterisks around comments.

[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)) (这个。。值得商榷啊)

* 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 };