Coding standards – C#.

This document is meant as a guideline for code practices for using C#. It's not intended to tell you how to write everything but merely suggests a few best practices and techniques to use so that the code is easier to interpret and maintain.

One rule to keep in mind is to be consistent. If someone has adopted a slightly different standard for their parts of the code, try to keep to theirs instead of writing small parts / extra parts in your own version. Inconsistency is worse than not following rules!

The majority of the coding standards are taken from the following website:
http://msdn.microsoft.com/en-us/library/ff926074.aspx

Please refer to the above document for more detailed information.

Table of Contents

Basic Conventions.

Basic C# conventions are covered here.

Naming Conventions.

When the using directive is not being utilized, use namespace qualifications. If you know that a namespace is being imported by default then there is no need to qualify names from that namespace. A small example is below:

```
var currentPerformanceCounterCategory = new System.Diagnostics.
    PerformanceCounterCategory();
```

As you can see in the above example, if qualifying takes up too much space on a line, the . Operator can be used to make it fit on another line.

Layout Conventions.

Good formatting makes code more readable and understandable. Microsoft suggests the following standards to conform to:

- Use the default code editor settings (Smart indenting, four character indents, tabs saved as spaces) see this for more information.
- Write one statement per line.
- Write one declaration per line.
- Indent continuation lines one tab stop (four spaces).
- Add a blank line between method definitions and property definitions.

Commenting Conventions.

Some commenting conventions are as follows:

- Place the comment on a separate line, not at the end of code.
- Begin comments with upper case letters
- End with a full-stop (period).
- Leave a space between the comment delimiter and the text, i.e.:
  // See the space to my left?
  //Not here on this one!
- Do not create formatted blocks of asterisks (*) around comments.

Language Guidelines.

This section describes some practices that the MS C# team follows.

Strings.

Strings have a few standards:

- Use the + operator to concatenate short strings
- To append longer strings, use the StringBuilder object.

Local Variables.

Use implicit type for local variables when the type of the variable is obvious or where type is not important. For example:

- When the type of a variable is clear from the context or assignment, use var.

```
var var1 = "This is clearly a string.";
var var2 = 27;
var var3 = Convert.ToInt32(Console.ReadLine());
```

- When the type of variable is not clear from the context, use an explicit type
```
int var4 = ExampleClass.ResultSoFar();
```

- Do not rely upon the variable name to specify the correct type. It might be incorrect or misleading.
```
var inputInt = Console.ReadLine();
Console.WriteLine(inputInt);
```
- Avoid the use of Var in place of Dynamic.
- Use implicit typing to determine the type of a loop variable:

```
for (var i = 0; i < 10; i++)
foreach (var ch in laugh)
```
- In general, always using int rather than unsigned types. It makes it easier to interact with other C# libraries as they commonly use int rather than an unsigned int.

Arrays.

Use concise syntax for array initialisation 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.
```

Try-Catch / Exception Handling.

A few pointers about exception handling:

- Use try-catch statements for most exception handling.
- You can simply your code by implementing the using statement. It should be used if you have a try-finally block and if the finally block calls the dispose method. See below:

```
// 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;
}
```

Logical Operators.

Two simple rules / suggestions to avoid exceptions and increase performance:

1. use && instead of a single &.
2. use || instead of a single |.

New Operator.

- Use the concise form of object instantiation using implicit typing, see below:

```
var instance1 = new ExampleClass();
Is equivilent to
ExampleClass instance2 = new ExampleClass();
```

- Use object initialisers to simply 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;
```