C# Coding Conventions

Introduction

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.

I. Naming Conventions & Styles

1.1 PascalCasing for Class/Method/Constant, camelCasing for variable

- **PascalCasing -** This convention capitalizes the first character of each word.

- **camelCasing -** This convention capitalizes the first character of each word except the first one.

- Use a noun or noun phrase to name a class.
- Use _ to disinguish private and internal variables (_localVariable).
- Avoid the this keyword.

1.2 Use descriptive naming.

Always give a descriptive name to your classes, variables and methods. It is **OK** if the name is rather long. Also, avoid abbreviations except if they are well known. Abbreviations with two characters should be upper case, otherwise just the first character should be upper case.

- databaseID - **OK**
- CreateHtmlString() - **OK**
- CreateHTMLString() - **not OK**
- sanatizedHtmlString - **Ok rather long variable name**
- txtName - **Not OK, use full name (textName)**

1.3 Using C# predefined types (int, string, double,..) instead of System classes

(System.Int32, System.String,..)

- Use implicit typing 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 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.
- Use implicit typing to determine the type of the loop variable in for and foreach 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();
```

1.4 Prefix IMyInterface, Suffix MyAttribute/MyException

Prefix interface names with the letter "I", to indicate that the type is an interface. Do not use the underscore character (_). Use Pascal case. e.g., IServiceProvider.

1.5 Implicitly Typed Local Variables

- Use implicit typing 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 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.
- Use implicit typing to determine the type of the loop variable in for and foreach 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();
```

1.6 Commenting

- 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.

  */

  This should only be done for temporary measures, for instance when a code section needs to be temporary commented out.
  */
- All if statements should be laid out with the braces on separate lines and the condition starting on the same line as the if statement. When an else statement is required, it should be shown on its own line. When there is an else if then they should be on one line together with the condition. By default, blank lines should not separate the else statements, but if a large if/else statement is built up, optional blank lines are permitted. Any comments for the else/if statements should be placed inside the else/if block. e.g.,

  // Overall comment regarding the if block.

  if (j < 0)

  {

  // Some meaningful description in here.

  ...

  }

  else if (j > 0)

  {

  // Some meaningful comment goes here to describe

  // the condition.

  ...

  }

  else

  {

  // Some meaningful comment goes here to describe

  // the default condition.

  ...

```
}
```

1.7 Layout conventions

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.
- 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.
}
```

- Break after a comma;
- Break after an operator;
- Align the new line with the beginning of the expression at the same level.

Indentation: Tabs should not be used for indentation. The standard indentation level to be used is four spaces.

Inter-term spacing should include a single space after a comma or semicolon e.g., CurrencyConvert(amount, fromCurrency, toCurrency);

1.8 New Operator

- 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;
```

II. Language Guidelines

1. ## Usage of strings

   - 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 object. var phrase = "lalalalalalalalalalalalalalalalalalalalalalalalalala"; var manyPhrases = new StringBuilder(); for (var i = 0; i < 10000; i++) { manyPhrases.Append(phrase); }
   - As a rule of thumb, adding three or less strings is faster using concatination. When adding more than three strings, use a StringBuilder.

2. ## Arrays

   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";
```

LINQ Queries

- 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;
```
- Split Linq queries into multiple lines of statement for easier readability.

III. C#6 Improvements & must do

(source: MSDN)

2.1 String Interpolation: $"{LastName}, {FirstName}"

2.2 Auto-property initializes: public int MyValue { get; } = 10;

2.3 Null-conditional operator: nullableParent**?.**Children.First()**?.**School**?.**Name ?? "n/a"

2.4 Expression-bodied function with: string ToString() => $"{LastName}, {FirstName}"

2.5 using static System.Math: to not import the whole namespace extensions

IIII. Best Practices

3.1 ALWAYS release resources "using" block or PROPERLY use try/finally block

```
// ALWAYS: use using code block if the resource supports IDisposable
using (var firstResource = Resources.CreateFirst())
{
 using (var otherResource = Resources.CreateOther())
 {
 }
}

// DOT NOT: miss use of resources creation inside of try block
```

```
Resource firstResource = null, otherResource = null; // BAD: have to
initialize
try
{
 firstResource = Resources.CreateFirst(); // BAD: Still go to finally block
if exception occurs
 // Doing something with firstResource

 otherResource = Resources.CreateOther(); // BAD: Still go to finally block
if exception occurs
 // Doing something with both resources
}
finally
{
 if (firstResource != null) Resources.Release(firstResource); // BAD: have
to check null before release
 if (otherResource != null) Resources.Release(otherResource); // BAD: have
to check null before release
}

// ALWAYS: resource creation must be right before the try block
var firstResource = Resources.CreateFirst(); // GOOD: initialize at
creation (save CPU cycles)
try
{
 // Doing something with firstResource


 var otherResource = Resources.CreateOther(); // GOOD: initialize at
creation (save CPU cycles)
 try
 {
  // Doing something with both resources
 }
 finally
 {
  Resources.Release(otherResource); // GOOD: always not null (simple is
better)
 }
}
finally
{
```

```
 Resources.Release(firstResource); // GOOD: always not null (simple is
better)
}
```

3.2 Using chaining Lambda (data pipeline thinking) instead of LinQ (a SQL language optimized for RDBMS)

```
// NO: do not use this code & this thinking
from bullShit in theHell where bullShit.Owner == "??" and (much more
filtering...) select bullShit // again - make no sense

// YES: thinking like a pipe of data from input (the 1st argument) to
output (the return value of the expressing chain)
manyThings.Where(thing => filterSmaller(thing))
 .Select(thing => convertToOther(thing))
 .Where(..filter otherThings..).OrderBy(...).Take(..10 first items..)
 .Select(..from the 10..).OrderBy(..difference..).First(..found thing..
```

3.4 Only using C# dynamic for COM, Javascript Interop or complex reflection

```
// C# dynamic is a late-type binding or run-time type checking mechanism by
.NET, which will
// seriously degrade application performance. By just debugging the
application then
// try to inspect the dynamic variable in Visual Studio, we could see how
slow is it

// Example1: migrate VBA code to C#
-----------------------------------------------------------------------
// Original VBA code
excel.GetCalculator().Add("B1", "C2")

// C# with reflection
object excel = GetExcel();
var excelType = excel.GetType();
object calc = excelType.InvokeMember("GetCalculator",
BindingFlags.InvokeMethod, null, new object[] { });
var calcType = calc.GetType();
object res = calcType.InvokeMember("Add", BindingFlags.InvokeMethod, null,
new object[] { 10, 20 });
var sum = Convert.ToDouble(res);

// C# dynamic
dynamic excel = GetExcel();
var sum = Convert.ToDouble(excel.GetCalculator().Add("B1", "C2"));

// Example2: dynamic Scripting Interop with C# dynamic (javascript,
python,..)
-----------------------------------------------------------------------
var items = new int[] { 1, 2, 3, 4, 5, 6, 7 };
var py = Python.CreateRuntime();
dynamic random = py.UseFile("random.py");
random.shuffle(items);
```