CODING STYLE GUIDE

Table of Contents

1. Naming Conventions

- Do not include using directives, 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.

2. Layout Conventions

- Use the default Code Editor settings (smart indenting, four-character indents, tabs saved as spaces).
- 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.
}
```

3. Commenting Conventions

- 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.

4. Language Guidelines Conventions

4.1.String Data Type

- Use the + operator to concatenate short strings
- To append strings in loops, especially when you are working with large amounts of text, use a StringBuilder object.

```
var phrase = "weeeeeeeeeeeeeeeeeeeeee";
var manyPhrases = new StringBuilder();
for (var i = 0; i < 10000; i++)
{
  manyPhrases.Append(phrase);
}
```

4.2. 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.

```
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.

```
int var4 = ExampleClass.ResultSoFar();
```

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

```
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.

```
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();
```

4.3. Unsigned Data Type

- 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**.

4.4. 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";
// And so on.
```

4.5. Delegates

- 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);
}
```

4.6. try-catch and using Statements in Exception Handling

- Use a try-catch statement for most exception handling.

- Simplify your code by using the C# using statement. If you have a try-finally statement in which the only code in the **finally** block is a call to the Dispose 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;
}
```

4.7. New Operator

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

  ```
  var instance1 = new ExampleClass();
  ```

- Use object initializers to simplify object creation.

4.8. Static Members

- Call static 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.