C# Coding Style

The general rule we follow is "use Visual Studio defaults".

1. We use Allman style braces, where each brace begins on a new line. A single line statement block can go without braces but the block must be properly indented on its own line and must not be nested in other statement blocks that use braces (See rule 18 for more details). One exception is that a `using` statement is permitted to be nested within another `using` statement by starting on the following line at the same indentation level, even if the nested `using` contains a controlled block.

```
while (x == y)
{
    something();
    something_else();
}
```

```
using FileStream fileStream = new FileStream(filePath, FileMode.Open);
using StreamReader reader = new StreamReader(fileStream);
```

2. We use four spaces of indentation (no tabs).
3. We use `_camelCase` for internal and private fields and use `readonly` where possible. Prefix internal and private instance fields with `_`, static fields with `s_` and thread static fields with `t_`. When used on static fields, `readonly` should come after `static` (e.g. `static readonly` not `readonly static`). Public fields should be used sparingly and should use PascalCasing with no prefix when used.

```
private static readonly double _defaultSalary = 30000.0;

// Static field example (use s_ prefix for static fields)
private static int s_employeeCount;

        // Thread static field example (use t_ prefix for thread static
fields)
[ThreadStatic]
private static int t_threadSpecificData;

// Instance fields (private, use _ prefix)
private string _firstName;
private string _lastName;
private int _employeeId;
private double _salary;

// Public property for first name
public string FirstName
{
    get { return _firstName; }
    set { _firstName = value; }
}
```

4. We avoid `this.` unless absolutely necessary.
5. We always specify the visibility, even if it's the default (e.g. `private string _foo` not `string _foo`). Visibility should be the first modifier (e.g. `public abstract` not `abstract public`).
6. Namespace imports should be specified at the top of the file, *outside* of `namespace` declarations, and should be sorted alphabetically, with the exception of `System.*` namespaces, which are to be placed on top of all others.
7. Avoid more than one empty line at any time. For example, do not have two blank lines between members of a type.
8. Avoid spurious free spaces. For example avoid `if (someVar == 0)...`, where the dots mark the spurious free spaces. Consider enabling "View White Space (Ctrl+R, Ctrl+W)" or "Edit -> Advanced -> View White Space" if using Visual Studio to aid detection.

```
if ( someVar == 0 )
{
   Console.WriteLine("Variable is zero.");
}

for ( int i = 0; i < anotherVar; i++ ) {
   Console.WriteLine($"Current count: {i}");
}

TestFunction (someVar);
```

9. If a file happens to differ in style from these guidelines (e.g. private members are named `m_member` rather than `_member`), the existing style in that file takes precedence.
10. We only use `var` when the type is explicitly named on the right-hand side, typically due to either `new` or an explicit cast, e.g. `var stream = new FileStream(...)` not `var stream = OpenStandardInput()`.

- Similarly, target-typed `new()` can only be used when the type is explicitly named on the left-hand side, in a variable definition statement or a field definition statement. e.g. `FileStream stream = new(...);`, but not `stream = new(...);` (where the type was specified on a previous line).

```
[BAD] var users = GetUsers().Method1()...Methodn();
```

11. We use language keywords instead of BCL types (e.g. `int, string, float` instead of `Int32, String, Single`, etc) for both type references as well as method calls (e.g. `int.Parse` instead of `Int32.Parse`). See issue #13976 for examples.
12. We use PascalCasing to name all our constant local variables and fields. The only exception is for interop code where the constant value should exactly match the name and value of the code you are calling via interop.

13. We use PascalCasing for all method names, including local functions.
14. We use `nameof(...)` instead of `"..."` whenever possible and relevant.
15. Fields should be specified at the top within type declarations.
16. When including non-ASCII characters in the source code use Unicode escape sequences (\uXXXX) instead of literal characters. Literal non-ASCII characters occasionally get garbled by a tool or editor.

```
[BAD] private string _helloArabic = "مرحبا";
[GOOD] private string _helloArabic = "\u0645\u0631\u062D\u0628\u0627";
```

17. When using labels (for goto), indent the label one less than the current indentation.
18. When using a single-statement if, we follow these conventions:
    - Never use single-line form (for example: `if (source == null) throw new ArgumentNullException("source");`)
    - Using braces is always accepted, and required if any block of an `if`/`else if`/.../`else` compound statement uses braces or if a single statement body spans multiple lines.
    - Braces may be omitted only if the body of *every* block associated with an `if`/`else if`/.../`else` compound statement is placed on a single line.
19. Make all internal and private types static or sealed unless derivation from them is required. As with any implementation detail, they can be changed if/when derivation is required in the future.

Ours

20. If there is some Result pattern which contains IsSuccess and IsFailure properties, checking if(IsSuccess is true), true is redundant not need to have to use it.
21. Variable names should be self-describing without any prefixes or suffixes determining accessibility or type.

```
[BAD] var usersReadOnly = GetUsers();
[BAD] var usersDict = GetUsers();

[GOOD] IReadOnlyList<User> users = GetUsers();
[GOOD] IDictionary<string, User> = GetUsers();
```

An EditorConfig file (`.editorconfig`) has been provided at the root of the runtime repository, enabling C# auto-formatting conforming to the above guidelines.

We also use the .NET Codeformatter Tool to ensure the code base maintains a consistent style over time, the tool automatically fixes the code base to conform to the guidelines outlined above.