styleguide

C# at Google Style Guide

This style guide is for C# code developed internally at Google, and is the default style for C# code at Google. It makes stylistic choices that conform to other languages at Google, such as Google C++ style and Google Java style.

Formatting guidelines

Naming rules

Naming rules follow Microsoft's C# naming guidelines. Where Microsoft's naming guidelines are unspecified (e.g. private and local variables), rules are taken from the CoreFX C# coding guidelines

Rule summary:

Code

- Names of classes, methods, enumerations, public fields, public properties, namespaces: `PascalCase`.
- Names of local variables, parameters: `camelCase`.
- Names of private, protected, internal and protected internal fields and properties: `_camelCase`.
- Naming convention is unaffected by modifiers such as const, static, readonly, etc.
- For casing, a "word" is anything written without internal spaces, including acronyms. For example, `MyRpc` instead of ~~`MyRPC`~~.
- Names of interfaces start with `I`, e.g. `IInterface`.

Files

- Filenames and directory names are `PascalCase`, e.g. `MyFile.cs`.
- Where possible the file name should be the same as the name of the main class in the file, e.g. `MyClass.cs`.
- In general, prefer one core class per file.

Organization

- Modifiers occur in the following order: `public protected internal private new abstract virtual override sealed static readonly extern unsafe volatile async`.
- Namespace `using` declarations go at the top, before any namespaces. `using` import order is alphabetical, apart from `System` imports which always go first.
- Class member ordering:
  - Group class members in the following order:
    - Nested classes, enums, delegates and events.
    - Static, const and readonly fields.
    - Fields and properties.
    - Constructors and finalizers.
    - Methods.
  - Within each group, elements should be in the following order:
    - Public.
    - Internal.
    - Protected internal.
    - Protected.
    - Private.
  - Where possible, group interface implementations together.

Whitespace rules

Developed from Google Java style.

- A maximum of one statement per line.
- A maximum of one assignment per statement.
- Indentation of 2 spaces, no tabs.
- Column limit: 100.
- No line break before opening brace.
- No line break between closing brace and `else`.
- Braces used even when optional.
- Space after `if` / `for` / `while` etc., and after commas.
- No space after an opening parenthesis or before a closing parenthesis.
- No space between a unary operator and its operand. One space between the operator and each operand of all other operators.
- Line wrapping developed from Google C++ style guidelines, with minor modifications for compatibility with Microsoft's C# formatting tools:
  - In general, line continuations are indented 4 spaces.
  - Line breaks with braces (e.g. list initializers, lambdas, object initializers, etc) do not count as continuations.

- For function definitions and calls, if the arguments do not all fit on one line they should be broken up onto multiple lines, with each subsequent line aligned with the first argument. If there is not enough room for this, arguments may instead be placed on subsequent lines with a four space indent. The code example below illustrates this.

Example

```
using System;                                       // `using` goes at the top, ou
                                                    // namespace.

namespace MyNamespace {                             // Namespaces are PascalCase.
                                                    // Indent after namespace.
  public interface IMyInterface {                   // Interfaces start with 'I'
    public int Calculate(float value, float exp);   // Methods are PascalCase
                                                    // ...and space after comma.
  }

  public enum MyEnum {                              // Enumerations are PascalCase
    Yes,                                            // Enumerators are PascalCase.
    No,
  }

  public class MyClass {                            // Classes are PascalCase.
    public int Foo = 0;                             // Public member variables are
                                                    // PascalCase.
    public bool NoCounting = false;                 // Field initializers are enco
    private class Results {
      public int NumNegativeResults = 0;
      public int NumPositiveResults = 0;
    }
    private Results _results;                       // Private member variables ar
                                                    // _camelCase.
    public static int NumTimesCalled = 0;
    private const int _bar = 100;                   // const does not affect namin
                                                    // convention.
    private int[] _someTable = {                    // Container initializers use
      2, 3, 4,                                      // space indent.
    }

    public MyClass() {
      _results = new Results {
        NumNegativeResults = 1,                     // Object initializers use a 2
        NumPositiveResults = 1,                     // indent.
      };
    }

    public int CalculateValue(int mulNumber) {      // No line break before openin
      var resultValue = Foo * mulNumber;            // Local variables are camelCa
      NumTimesCalled++;
      Foo += _bar;
```

```
    if (!NoCounting) {                              // No space after unary operat
                                                    // space after 'if'.
      if (resultValue < 0) {                        // Braces used even when optio
                                                    // spaces around comparison op
        _results.NumNegativeResults++;
      } else if (resultValue > 0) {                 // No newline between brace an
        _results.NumPositiveResults++;
      }
    }

    return resultValue;
  }

  public void ExpressionBodies() {
    // For simple lambdas, fit on one line if possible, no brackets or braces re
    Func<int, int> increment = x => x + 1;

    // Closing brace aligns with first character on line that includes the openi
    Func<int, int, long> difference1 = (x, y) => {
      long diff = (long)x - y;
      return diff >= 0 ? diff : -diff;
    };

    // If defining after a continuation line break, indent the whole body.
    Func<int, int, long> difference2 =
        (x, y) => {
          long diff = (long)x - y;
          return diff >= 0 ? diff : -diff;
        };

    // Inline lambda arguments also follow these rules. Prefer a leading newline
    // groups of arguments if they include lambdas.
    CallWithDelegate(
        (x, y) => {
          long diff = (long)x - y;
          return diff >= 0 ? diff : -diff;
        });
  }

  void DoNothing() {}                               // Empty blocks may be concise

  // If possible, wrap arguments by aligning newlines with the first argument.
  void AVeryLongFunctionNameThatCausesLineWrappingProblems(int longArgumentName,
                                                           int p1, int p2) {}

  // If aligning argument lines with the first argument doesn't fit, or is diffi
  // read, wrap all arguments on new lines with a 4 space indent.
  void AnotherLongFunctionNameThatCausesLineWrappingProblems(
      int longArgumentName, int longArgumentName2, int longArgumentName3) {}

  void CallingLongFunctionName() {
    int veryLongArgumentName = 1234;
    int shortArg = 1;
```

```
        // If possible, wrap arguments by aligning newlines with the first argument.
        AnotherLongFunctionNameThatCausesLineWrappingProblems(shortArg, shortArg,
                                                              veryLongArgumentName);
        // If aligning argument lines with the first argument doesn't fit, or is dif
        // read, wrap all arguments on new lines with a 4 space indent.
        AnotherLongFunctionNameThatCausesLineWrappingProblems(
            veryLongArgumentName, veryLongArgumentName, veryLongArgumentName);
      }
    }
  }
```

C# coding guidelines

Constants

- Variables and fields that can be made `const` should always be made `const`.
- If `const` isn't possible, `readonly` can be a suitable alternative.
- Prefer named constants to magic numbers.

IEnumerable vs IList vs IReadOnlyList

- For inputs use the most restrictive collection type possible, for example `IReadOnlyCollection` / `IReadOnlyList` / `IEnumerable` as inputs to methods when the inputs should be immutable.
- For outputs, if passing ownership of the returned container to the owner, prefer `IList` over `IEnumerable`. If not transferring ownership, prefer the most restrictive option.

Generators vs containers

- Use your best judgement, bearing in mind:
  - Generator code is often less readable than filling in a container.
  - Generator code can be more performant if the results are going to be processed lazily, e.g. when not all the results are needed.
  - Generator code that is directly turned into a container via `ToList()` will be less performant than filling in a container directly.
  - Generator code that is called multiple times will be considerably slower than iterating over a container multiple times.

Property styles

- For single line read-only properties, prefer expression body properties (`=>`) when possible.
- For everything else, use the older `{ get; set; }` syntax.

Expression body syntax

For example:

```
int SomeProperty => _someProperty
```

- Judiciously use expression body syntax in lambdas and properties.
- Don’t use on method definitions. This will be reviewed when C# 7 is live, which uses this syntax heavily.
- As with methods and other scoped blocks of code, align the closing with the first character of the line that includes the opening brace. See sample code for examples.

Structs and classes:

- Structs are very different from classes:
  - Structs are always passed and returned by value.
  - Assigning a value to a member of a returned struct doesn’t modify the original - e.g. `transform.position.x = 10` doesn’t set the transform’s position.x to 10; `position` here is a property that returns a `Vector3` by value, so this just sets the x parameter of a copy of the original.
- Almost always use a class.
- Consider struct when the type can be treated like other value types - for example, if instances of the type are small and commonly short-lived or are commonly embedded in other objects. Good examples include Vector3, Quaternion and Bounds.
- Note that this guidance may vary from team to team where, for example, performance issues might force the use of structs.

Lambdas vs named methods

- If a lambda is non-trivial (e.g. more than a couple of statements, excluding declarations), or is reused in multiple places, it should probably be a named method.

Field initializers

- Field initializers are generally encouraged.

Extension methods

- Only use an extension method when the source of the original class is not available, or else when changing the source is not feasible.

- Only use an extension method if the functionality being added is a ‘core’ general feature that would be appropriate to add to the source of the original class.
  - Note - if we have the source to the class being extended, and the maintainer of the original class does not want to add the function, prefer not using an extension method.
- Only put extension methods into core libraries that are available everywhere - extensions that are only available in some code will become a readability issue.
- Be aware that using extension methods always obfuscates the code, so err on the side of not adding them.

ref and out

- Use `out` for returns that are not also inputs.
- Place `out` parameters after all other parameters in the method definition.
- `ref` should be used rarely, when mutating an input is necessary.
- Do not use `ref` as an optimisation for passing structs.
- Do not use `ref` to pass a modifiable container into a method. `ref` is only required when the supplied container needs be replaced with an entirely different container instance.

LINQ

- In general, prefer single line LINQ calls and imperative code, rather than long chains of LINQ. Mixing imperative code and heavily chained LINQ is often hard to read.
- Prefer member extension methods over SQL-style LINQ keywords - e.g. prefer `myList.Where(x)` to `myList where x`.
- Avoid `Container.ForEach(...)` for anything longer than a single statement.

Array vs List

- In general, prefer `List<>` over arrays for public variables, properties, and return types (keeping in mind the guidance on `IList` / `IEnumerable` / `IReadOnlyList` above).
- Prefer `List<>` when the size of the container can change.
- Prefer arrays when the size of the container is fixed and known at construction time.
- Prefer array for multidimensional arrays.
- Note:
  - array and `List<>` both represent linear, contiguous containers.
  - Similar to C++ arrays vs `std::vector`, arrays are of fixed capacity, whereas `List<>` can be added to.
  - In some cases arrays are more performant, but in general `List<>` is more flexible.

Folders and file locations

- Be consistent with the project.

- Prefer a flat structure where possible.

Use of tuple as a return type

- In general, prefer a named class type over `Tuple<>`, particularly when returning complex types.

String interpolation vs `String.Format()` vs `String.Concat` vs `operator+`

- In general, use whatever is easiest to read, particularly for logging and assert messages.
- Be aware that chained `operator+` concatenations will be slower and cause significant memory churn.
- If performance is a concern, `StringBuilder` will be faster for multiple string concatenations.

`using`

- Generally, don't alias long typenames with `using`. Often this is a sign that a `Tuple<>` needs to be turned into a class.
  - e.g. `using RecordList = List<Tuple<int, float>>` should probably be a named class instead.
- Be aware that `using` statements are only file scoped and so of limited use. Type aliases will not be available for external users.

Object Initializer syntax

For example:

```
var x = new SomeClass {
  Property1 = value1,
  Property2 = value2,
};
```

- Object Initializer Syntax is fine for 'plain old data' types.
- Avoid using this syntax for classes or structs with constructors.
- If splitting across multiple lines, indent one block level.

Namespace naming

- In general, namespaces should be no more than 2 levels deep.
- Don't force file/folder layout to match namespaces.

- For shared library/module code, use namespaces. For leaf ‘application’ code, such as `unity_app`, namespaces are not necessary.
- New top-level namespace names must be globally unique and recognizable.

Default values/null returns for structs

- Prefer returning a ‘success’ boolean value and a struct `out` value.
- Where performance isn’t a concern and the resulting code significantly more readable (e.g. chained null conditional operators vs deeply nested if statements) nullable structs are acceptable.

- Notes:

    - Nullable structs are convenient, but reinforce the general ‘null is failure’ pattern Google prefers to avoid. We will investigate a `StatusOr` equivalent in the future, if there is enough demand.

Removing from containers while iterating

C# (like many other languages) does not provide an obvious mechanism for removing items from containers while iterating. There are a couple of options:

- If all that is required is to remove items that satisfy some condition, `someList.RemoveAll(somePredicate)` is recommended.
- If other work needs to be done in the iteration, `RemoveAll` may not be sufficient. A common alternative pattern is to create a new container outside of the loop, insert items to keep in the new container, and swap the original container with the new one at the end of iteration.

Calling delegates

- When calling a delegate, use `Invoke()` and use the null conditional operator - e.g. `SomeDelegate?.Invoke()`. This clearly marks the call at the callsite as ‘a delegate that is being called’. The null check is concise and robust against threading race conditions.

The `var` keyword

- Use of `var` is encouraged if it aids readability by avoiding type names that are noisy, obvious, or unimportant.

- Encouraged:

    - When the type is obvious - e.g. `var apple = new Apple();`, or `var request = Factory.Create<HttpRequest>();`
    - For transient variables that are only passed directly to other methods - e.g. `var item = GetItem(); ProcessItem(item);`

- Discouraged:
  - When working with basic types - e.g. `var success = true;`
  - When working with compiler-resolved built-in numeric types - e.g. `var number = 12 * ReturnsFloat();`
  - When users would clearly benefit from knowing the type - e.g. `var listOfItems = GetList();`

Attributes

- Attributes should appear on the line above the field, property, or method they are associated with, separated from the member by a newline.
- Multiple attributes should be separated by newlines. This allows for easier adding and removing of attributes, and ensures each attribute is easy to search for.

Argument Naming

Derived from the Google C++ style guide.

When the meaning of a function argument is nonobvious, consider one of the following remedies:

- If the argument is a literal constant, and the same constant is used in multiple function calls in a way that tacitly assumes they're the same, use a named constant to make that constraint explicit, and to guarantee that it holds.
- Consider changing the function signature to replace a `bool` argument with an `enum` argument. This will make the argument values self-describing.
- Replace large or complex nested expressions with named variables.
- Consider using Named Arguments to clarify argument meanings at the call site.
- For functions that have several configuration options, consider defining a single class or struct to hold all the options and pass an instance of that. This approach has several advantages. Options are referenced by name at the call site, which clarifies their meaning. It also reduces function argument count, which makes function calls easier to read and write. As an added benefit, call sites don't need to be changed when another option is added.

Consider the following example:

```
// Bad - what are these arguments?
DecimalNumber product = CalculateProduct(values, 7, false, null);
```

versus:

```
// Good
ProductOptions options = new ProductOptions();
options.PrecisionDecimals = 7;
options.UseCache = CacheUsage.DontUseCache;
DecimalNumber product = CalculateProduct(values, options, completionDelegate: null
```