Naming Conventions & Coding Standards in C#

Andreas Winther Moen
andreas@winthermoen.no

Abstract

This document is made as a personal guideline for my C# Unity projects. Feel free to use it yourself if you like. Although this document is intended for Unity projects, it may just as well be used for other C# projects. The last section is specific to Unity and may be ignored.

Above all, code should be easily readable and understandable. Write your code for the reader, not the writer. In general I try to avoid abbreviations, and prefer maximizing readability over typing speed.

Typing is quick.
Debugging is slow.

Contents

1 Naming

1.1 Variables

Variables must be descriptive. The variable's purpouse must be clear based entirely on its name. The only exception is loop counters (which may be named *i, j, k, ...*), although descriptive loop counters are encouraged for intricate and/or nested loops.

1.1.1 Private Member Variables

Variables which are inaccessible by external classes (access types *private*, *protected* or *private protected*) must be named using lower camel case with the prefix *m*.

```
private int mFooBarBaz = 0;

protected float mQuxQuuzCorge = 1f;
```

1.1.2 Public Member Variables

Variables with public scope (acceess types *internal*, *public* or *protected internal*) should generally be avoided, except for accessors. Such variables must be named using lower camel case.

```
// Private member variable and associated accessor.
private string mFooBarBaz = "Qux";
public string fooBarBaz
{
    get { return mFooBarBaz; }
    private set { mFooBarBaz = value; }
}

// Public variable, generally discouraged except for structs
public int quxQuuzCorge = 10;
```

1.1.3 Local Variables

Variables declared within the local scope of a function must be named using lower camel case. This also applies to function parameters.

```
private void Foo(int barBaz)
{
    int quxQuux = 0;

    //more code
}
```

1.1.4 Static & Const Variables

Static and const variables must be named using lower camel case, regardless of scope.

```
public static int fooBar;

private static int bazQux;
```

1.2 Functions

All functions must be named using upper camel case, regardless of scope.

```
public void FooBar()
{
    //code
}

private int BazQux()
{
    //more code
}
```

1.3 Custom Types

1.3.1 Classes and Structs

Custom classes and structs must be named using upper camel case. A class should resemble one, and only one, specific concept, and should be named accordingly. Structs are used to hold specific, small amounts of non-pointer data and must similarly be named according to its specific use.

```
public class FooBar
{
    //code
};

public struct BazQux
{
    //code
};
```

1.3.2 Interfaces

Interfaces are signatures of a class, and should be named accordingly. Every interface must start with *I*, followed by an upper camel case name.

```
public interface IFoo
{
    //code
};
```

1.3.3 Enums

Enums, or enumerations, are distinct types consisting of named constants. The name of an enum must use upper camel case. The values must be named using upper case letters with underscores as word separators. Enums may be declared on either one or multiple lines. If multiple lines are used, each value should be declared on a separate line.

```
// Correct
public enum Foo { BAR, BAZ_QUX };

// Also correct
public enum Quuz
{
    CORGE_GRAULT,
    GARPLY,
    WALDO
};
```

1.4 Other Data Types

- Delegates are treated like variables.
- Events are treated like variables.

1.5 Abbreviations

Abbreviations should be used with care. Word shortening may be used to make variables or functions more compact as long as the purpouse is still clear.

```
// Full name
private int numberOfColonizablePlanets;

// Acceptable abbreviation
private int numColonizablePlanets;

// Wrong. Not very descriptive.
private int numColPla;

// Wrong. If you do this you should reconsider your career.
private int ncp;
```

Common acronyms may be used, but must be typed as a regular word with only the first letter being upper case, (or lowercase if it's the first word of a local variable or public member variable). If the acronym is more commonly used than the full name (such as HTTP instead of HyperText Transfer Protocol), acronyms are preferred.

```
// Correct
private int mObjectId;
private SomeClass mXmlHttpRequest;

// Wrong
private int mObjectID;
private SomeClass mXMLHTTPRequest;

// Very wrong for obvious reasons
private SomeClass mEXtensibleMarkupLanguageHyperTextTransferProtocolRequest;
```

1.6 Files

Files and folders must be named using upper camel case. Special files, such as temporary files, init files or other files that should appear at the top of an alphabetically sorted directory, must start with a descriptive upper case prefix, preceded and followed by an underscore.

```
FooBar.txt

_TMP_SomeTemporaryFile.cs
```

2 Code Formatting

2.1 Indentation

Tabs must be used to indent code, and tabs must be set to four spaces.

2.2 Spacing

2.2.1 Binary Operators

A space must precede and follow binary operators (*+*, *-*, ***, *etc*) to enhance readability.

```
// Correct
foo += 1;

// Wrong
foo+=1;
```

2.2.2 Comments

Comments must always be declared with *//* as opposed to */* */*. The latter is undesirable for multiline comments because only the first and last lines are clearly marked as comments. Use *Ctrl+K+C* to comment multiple lines, and *Ctrl+K+U* to uncomment multiple lines.

```
// This
// is
// the
// correct
// way
// to
// comment
// multiple
// lines


/*
    This
    is
    not
    the
    correct
    way
*/
```

The only occasion on which /* */ should be used is when only a part of the line should be commented.

```
public void Foo (/* information about parameter */)
{
    //code
}
```

A comment which describes a function or variable must have one space separator after //. Comments which are used to (temporarily) disable certain code lines should not be separated from //. This is simply because the *Ctrl+K+C* hotkey to comment out code doesn't add a space, hence it is quicker this way.

```
// This is a comment.
//private int someUnusedVariable;
```

2.2.3 Line Spacing

In general there should always be one line between each line of text. The exceptions are:

- Code within a method must be separated based on tasks, not lines. That is, if one specific task requires multiple lines, the lines require no separating line. This includes, but is not limited to, updating only one axis of the position of a Transform component (since the Vector3 must be cached).
- Accessors must be placed directly below their associated member variable. If separate *Get* and *Set* methods are used as accessors, all three lines of code must be placed with no separation, starting with the variable declaration.
- Documentation of methods and variables must be placed directly above the method/variable.
- *Using* directives.
- The line of code directly following or preceding a curly bracket or preprocessor directive (such as #*if*, #*endif*, #*region*, *etc*).

2.3 Scope Brackets

Scope brackets must be placed on separate lines, and the content of the scope must be tabbed.

```
while (fooBar > 10)
{
    //code
}
```

The only exception is scopes with only one line of code, which may be typed either like above or as just one line. The former is generally preferred, but the latter may be used if there are many similarly declared functions (such as *Get* and *Set* functions of a database).

```
private int mFoo;
public void SetFoo(int foo) { this.mFoo = foo; }
public int GetFoo() { return mFoo; }
```

2.4 Boolean Checks

Negative boolean checks must be typed as below to increase readability. An exclamation mark is easy to miss, but == *false* is very clear. When checking for positives, only the statement should be typed.

```
// Correct
if (statement == false)
{
    //code
}

// Wrong
if (!statement)
{
    //code
}

// Correct
if (statement)
{
    //code
}

// Wrong
if (statement == true)
{
    //code
}
```

2.5 Text Width

All lines of code must be less than 120 characters wide. This makes the code appear similar on any monitor with a resolution of at least 720p. For big monitors it makes the text not cover the whole screen so you don't have to continuously move your head around like a hyperactive owl.
Some IDEs and text editors have built in text wrap and/or guidelines. For some reason Visual Studio doesn't. Here is a VS extension to add guidelines: `https://marketplace.visualstudio.com/items?itemName=PaulHarrington.EditorGuidelines`

3 Code Structure

3.1 Scope

Member variables and functions should generally be *private* (or *protected*). Member variables should generally be accessed externally through a public member function.

3.2 Access Types

The access type of a function or a non-local variable must always be specified, even if the default type is the desired one.

```
// Correct
private int mFoo = 1;

// Wrong
int mFoo = 1;
```

3.3 Comments

3.3.1 General

Comments should be used to describe in depth the purpouse of certain parts of code, and are highly encouraged. However, unnecessary usage of comments impedes efficiency and readability. Properly naming of variables is always preferred over redundant and excessive comments.

> *Good comments don't repeat the code or explain it. They clarify its intent. Comments should explain, at a higher level of abstraction than the code, what you're trying to do.* (Steven C. McConnell)

3.3.2 Documentation

All functions and variables accessible by external classes (access types *internal*, *public* or *protected internal*) must be documented using the <*summary*> tag.

Input parameters and the return value should be documented if their names may be ambiguous. Private functions may be documented in case of ambiguity.

```
/// <summary>
/// Description of function goes here
/// </summary>
/// <param name="baz">Description of the parameter</param>
public void FooBar(int baz)
{
    //code
}
```

3.4 Identifiers

Strive to use const and readonly whenever possible. The MSIL will get the constant value directly instead of loading it from a variable (we basically skip an ldloc.0 call, increasing performance by one CPU cycle per call). However, the main reason for const- and readonly-correctness is that it prevents dumb people (including yourself) from making dumb mistakes. Use them as much as possible!

3.5 Regions and Ordering

Every class must be separated into two main regions: *Variables* and *Methods* (in that order). Nested regions are encouraged for large documents. The correct order of access modifiers within these regions is:

- *public*
- *protected internal*
- *internal*
- *protected*
- *private protected*
- *private*

There are some exceptions to this rule:

- Static and const variables must always be placed at the top of the *Variables* region (in that order), regardless of scope.
- Accessors must be placed right after their associated member variable. Overloaded methods with different access modifiers must be placed together. The highest order access modifier of the methods in the group of overloaded methods determines the position of the group.
- Constructors must be placed within a *Constructors* region at the top of the *Methods* region.

4 Unity

4.1 Variable Scope and Inspector Accessibility

The Unity documentation, as well as Unity tutorials and documentation by third parties, often declare member variables as public to make them visible and editable directly in the game engine (the editor). However, this violates scope conventions and is generally considered bad programming practice. To make variables appear in the inspector, add the attribute *[SerializeField]*.

```
[SerializeField]
private int mPlayerHp;
```

4.2 File Structure

The asset folder must only contain subfolders. That is, no files may be placed directly in the asset folder. Temporary files may be placed in a folder dedicated for this specific purpouse.

The first layer (the children) of the asset folder must consist of subfolders separating files by their types. Some examples include *Scripts, Materials, Models*. The second layer (and further layers) separate files by their purpouse. For instance, a *Scripts* folder may be sub-divided into *WorldObjects, Camera, Database, etc.*

In general, files should be grouped in folders based on their purpouse as opposed to their types. For instance, an enum named *WeaponTypes* may be placed in a folder named *Items*, but should not be placed in a folder named *Enums*.

4.3 Namespaces and the *using* Directive

The usage of namespaces are encouraged similar to standard coding procedures. However, the *using* directive for namespaces is repeatedly used in Unity's documentation and by third party tutorials and documentation, even though the general coding practice is to avoid it. It's also worth noting that Unity's classes often exist in deeply nested namespaces. Thusly, the *using* directive should be used for namespaces that occur frequently in code. In case of naming conflicts, the full namespace tree must be specified.

4.4 Regions and Ordering

Usage of regions and ordering should follow the standards defined in subsection 3.5. The only exception is built-in Unity functions, such as *Update*, *Start* and *OnEnable*. These must be placed first within the *Methods* region. If subregions are used within *Methods*, these functions should be placed within a separate region.