Coding Style Guide For Unity

Contents

1. Introduction

This is a condensed version of C# Style Guide Unity 6 edition. That guide is then a subset of Microsoft Framework Design Guidelines. As you go to the superset there are more rules defined for clarification.

Consistency is king. When this style guide conflicts with the any superset, the ruling in the lowest subset (generally this one) takes precedence.

2. Naming Conventions

2.1 Fields and Variables

Examples to avoid	Use instead	Notes
`int d`	`int elapsedTimeInDays`	Avoid single letter abbreviations unless a counter or expression. Be specific about the measurement unit.
`int hp, string tName, int mvmtSpeed`	`int healthPoints, string teamName, int movementSpeed`	Variable names reveal intent. Make names searchable and pronouncable.
`int getMovementSpeed`	`int movementSpeed`	Use nouns. Reserve verbs for methods unless it's a bool (below).
`bool dead`	`bool isDead, bool isPlayerDead`	Booleans ask a question that can be answered true or false.

— **Use pascal case (MyPropertyName) for public fields. Use camel case (myPropertyname) for private variables.**

```
// EXAMPLE: private and public field
public float DamageMultiplier = 1.5f;
public float MaxHealth;
private bool isDead;
private float currentHealth;
```

— **Use the `this` keyword** to distinguish between member and local variables.

```
// EXAMPLE: this keyword
public class Employee
{
    private string nickname;
    private string name;

    public Employee(string name, string nickname)
    {
        this.name = name;
        this.nickname = nickname;
    }
}
```

— **Use capital snake case (MY_PROPERTY_NAME) for constant variables.** This helps distinguish constants from regular variables or properties.

```
// EXAMPLE: constants
public class MathConstants
{
    public const int MAX_ITEMS = 100;
}
```

— **Use one variable declaration per line:** It's less compact, but enhances readability.

— **Avoid Redundant names:** If your class is called `Player`, you don't need to create member variables called `PlayerScore` or `PlayerTarget`. Trim them down to `Score` or `Target`

2.2 Enums

Use Pascal case for enum names and values. You can place public enums outside of a class to make them global. Use a singular noun for the enum name as it represents a single value from a set of possible values. They should have no prefix or suffix.

```
// EXAMPLE: enums use singular nouns
public enum FireMode
{
    None = 0,
    Single = 5,
    Burst = 7,
    Auto = 8,
}

// EXAMPLE: but a bitwise enum is plural (you can also use the 1 << bitnum
↪  style)
[Flags] public enum AttackModes
{
    // Decimal                          // Binary
    None = 0,                           // 000000
    Melee = 1,                          // 000001
    Ranged = 2,                         // 000010
    Special = 4,                        // 000100

    MeleeAndSpecial = Melee | Special   // 000101
}
```

2.3 Classes and interfaces

— **Use Pascal case nouns or noun phrases for class names:** This distinguishes type names from methods, which are named with verb phrases.

— **If you have a Monobehaviour in a file, the source file name must match:** You may have other internal classes in the file, but only one Monobehaviour should exist per file.

— **Prefix interface names with a capital I:** Follow this with an adjective that describes the functionality.

```
// EXAMPLE: Class formatting
public class ExampleClass : MonoBehaviour
{

}

// EXAMPLE: Interfaces
public interface IKillable
{
    void Kill();
}
public interface IDamageable<T>
{
    void Damage(T damageTaken);
}
```

2.4 Methods

— **Start the name with a verb or verb phrases:** Add context if necessary. e.g. `GetDirection`, `FindTarget`, etc.

— **Use camel case for parameters:** Format parameters passed into the method like local variables.

— **Methods returning bool should ask questions:** Much like Boolean variables themselves, prefix methods with a verb if they return a true-false condition This phrases them in the form of a question, e.g. `IsGameOver`, `HasStartedTurn`.

```
// EXAMPLE: Methods start with a verb
public void SetInitialPosition(float x, float y, float z)
{
    transform.position = new Vector3(x, y, z);
}

// EXAMPLE: Methods ask a question when they return bool
public bool IsNewPosition(Vector3 currentPosition)
{
    return (transform.position == newPosition);
}
```

2.5 Events and event handlers

Events in C# implement the observer pattern. This software design pattern defines a relationship in which one object, the subject (or publisher), can notify a list of dependent objects called observers (or subscribers). Thus, the subject can broadcast state changes to its observers without tightly coupling the objects involved.

— **Name the event with a verb phrase:** Choose a name that communicates the state change accurately. use the present or past participle to indicate events "before" or "after." For example, specify "OpeningDoor" for an event before opening a door or "DoorOpened" for an event afterward.

— **use the EventHandler delgate for events:** In most cases the EventHandler or EventHandler<TEventArgs> delegate can handle the events needed for gameplay. Using the predefined delegate saves code.

```
// EXAMPLE: Events using EventHandler
public event EventHandler OpeningDoor; // event before
public event EventHandler DoorOpened;  // event after

public event EventHandler<PointsScoredEventArgs> PointsScored;
public event EventHandler<CustomEventArgs> ThingHappened;
```

— **Prefix the event raising method (in the subject) with "On":** The subject that invokes the event typically does so from a method prefixed with "On", e.g. "OnOpeningDoor" or "OnDoorOpened."

```
// raises the Event if you have subscribers
public void OnDoorOpened()
{
    DoorOpened?.Invoke();
}

public void OnPointsScored(int points)
{
    PointsScored?.Invoke(points);
}
```

3. Formatting

3.1 Properties

A property provides a flexible mechanism to read, write, or compute class values. Properties behave as if they were public member variables, but are actually accessors. Each property has a get and set method to access a private field, called a backing field.

In this way, the property encapsulates the data, hiding it from unwanted changes by the user or external objects. The getter and setter each have their own access modifier, allowing your property to be read-write, read-only, or write-only.

You can also use the accessors to validate or convert the data (e.g., verify that the data fits your preferred format or change a value to a particular unit).

— **Use expression-bodies properties for single line read-only properties** (=>): This returns the private backing field.

```
// EXAMPLE: expression bodied properties

// the private backing field
private int maxHealth;

// read-only, returns backing field
// equivalent to: public int MaxHealth { get; private set; }
public int MaxHealth => maxHealth;
```

— **Everything uses the older** `{ get; set; }` **syntax:** Remember to make the setter private if you don't want to give write access.

```
// EXAMPLE: expression bodied properties

// backing field
private int maxHealth;

// explicitly implementing getter and setter
public int MaxHealth
{
    get => maxHealth;
    set => maxHealth = value;
}
// write-only (not using backing field)
public int Health { private get; set; }

// write-only, without an explicit setter
public SetMaxHealth(int newMaxValue) => maxHealth = newMaxValue;
```

— While you can also use functions to expose private data as in our example below, it's generally recommended to use properties for simple get/set operations. For operations involving complex logic or computation, methods are generally recommended.

```
// EXAMPLE: expression bodied properties

// backing field
private int maxHealth;

public int GetMaxHealth
{
    return maxHealth;
}
```

3.2 Serialization

Script serialization is the automatic process of transforming data structures or object states into a format that Unity can store and reconstruct later.

— **use the `[SerializeField]` attribute:** The SerializeField attribute can work with private or protected variables to make them appear in the Inspector. This encapsulates the data better than marking the variable public and prevents an external object from overwriting it's values.

— **Use the Range attribute to set minimum and maximum values:** The [Range(min, max)] attribute is handy if you want to limit what the user can assign to a numeric field. It also conveniently represents the field as a slider in the Inspector.

— **Group data in serializable classes or structs to clean up the Inspector:** Define a public class or struct and mark it with the `[Serializable]` attribute. Define public variables for each type you want to expose in the Inspector.

```
// EXAMPLE: a serializable class for PlayerStats
using System;
using UnityEngine;

public class Player : MonoBehaviour
{
    [Serializable]
    public struct PlayerStats
    {
        public int MovementSpeed;
        public int HitPoints;
        public bool HasHealthPotion;
    }

    // EXAMPLE: The private field is visible in the Inspector
    [SerializeField] private PlayerStats stats;
}
```

3.3 Brace and indentation style

— **Use the Allman style for curly braces.**

```
// EXAMPLE: Allman or BSD style puts opening brace on a new line.
void DisplayMouseCursor(bool showMouse)
{
    if (!showMouse)
    {
        Cursor.visible = false;
    }
    else
    {
        Cursor.visible = true;
    }
}
```

— **Use four spaces for indentation:** Make sure that your IDE converts tabs to spaces.

— **Where possible, don't omit braces, even for single line statements:** this increases consistency and if later you need to add a Debug line or run `DoSomethingElse` the braces will already be in place.

```
// EXAMPLE: keep braces for clarity and the clause on a separate line.
for (int i = 0; i < 100; i++)
{
    DoSomething(i);
}

// AVOID: omitting braces
for (int i = 0; i < 100; i++) DoSomething(i);
```

— **Switch statements:** It's generally advisable to replace longer if-else statements with a switch statement for better readability. It's generally recommended to include a default case as well. Even if the default case is not needed, including one ensures that the code is prepared to handle unexpected values.

```
// EXAMPLE: indent cases from the switch statement

switch (someExpression)
{
    case 0:
        DoSomething();
        break;
    case 1:
        DoSomethingElse();
        break;
    case 2:
        int n = 1;
        DoAnotherThing(n);
        break;
    default:
        // Handle unexpected or default case
        break;
}
```

3.4 Horizontal Spacing

— **Add spaces to decrease code density:** The extra whitespace can give a sense of visual separation between parts of a line improving readability.

```
// EXAMPLE: add spaces to make lines easier to read
for (int i = 0; i < 100; i++) { DoSomething(i); }

// AVOID: no spaces
for(inti=0;i<100;i++){DoSomething(i);}
```

— **Use a single space after a comma between function arguments.**

```
// EXAMPLE: single space after comma between arguments
CollectItem(myObject, 0, 1);

// AVOID: leaving out spacing
CollectItem(myObject,0,1);
```

— **Don't add a space after the parenthesis and function arguments.**

```
// EXAMPLE: no space after the parenthesis and function arguments
DropPowerUp(myPrefab, 0, 1);

//AVOID:
DropPowerUp( myPrefab, 0, 1 );
```

— **Don't use spaces between a function name and parenthesis.**

```
// EXAMPLE: omit spaces between a function name and parenthesis.
DoSomething()

// AVOID
DoSomething ()
```

— **Avoid spaces inside brackets.**

```
// EXAMPLE: omit spaces inside brackets
x = dataArray[index];

// AVOID
x = dataArray[ index ];
```

— **Use a single space before flow control conditions:** Add a space between the flow comparison operator and the parentheses.

— **Use a single space before and after comparison operators**

```
// EXAMPLE: space before condition; separate parentheses with a space.
while (x == y)

// AVOID
while(x==y)
```

— **Keep lines short. Consider horizontal whitespace:** Use the standard line width of 120 characters. Break a long line into smaller statements rather than letting it overflow.

3.5 Vertical Spacing

— **Group dependent and/or similar methods together:** Code needs to be logical and coherent. Keep methods that do the same thing next one another, so someone reading your logic doesn't have to jump around the file.

— **Use the vertical whitespace to your advantage to separate distinct parts of your class.** For example, you can add two blank lines between: variable declarations and methods, classes and interfaces, if-then-else block (if it helps readability).

— **Don't use regions:** If your follow the general advice for Classes, your class size should be manageable and the `#region` directive superfluous.

4. Classes

Limiting the size of each class makes it more focused and cohesive. It's easy to keep adding on top of an existing class until it overextends with functionality. Instead make a conscious effort to keep the classes short. Big, bloated classes become difficult to read and troubleshoot.

4.1 The newspaper metaphor

Imagine the source code of a class as a news article. You start reading from the top, where the headline and byline catch your eye. The lead-in paragraph gives you a rough summary. Then you glean more details as you continue downward.

Your class should also follow this basic pattern. Organize top-down and think of your functions as forming a hierarchy. Some methods serve a higher-level and lay the groundwork for the big picture. Put these first. Then, place lower-level functions with implementation details later.

For example, you might make a method called `ThrowBall` that references other methods, `SetInitialVelocity` and `CalculateTrajectory`. Keep `ThrowBall` first, since that describes the main action. Then, add the supporting methods below it.

Though each news article is short, a newspaper or news website will have many such collected stories. When taken together, the articles comprise a unified, functional whole. Think of your Unity project in the same way. It has numerous classes that must come together to form a larger, yet coherent, application.

4.2 Class organization

Each class will need some standardization. Group class members into sections to organize them:

— Fields

— Properties

— Events / Delegates

— Monobehaviour Methods (Awake, Start, OnEnable, OnDisable, OnDestroy, etc.)

— Public Methods

— Private Methods

Recall the recommended class naming rules in Unity: the source file name must match the name of the Monobehaviour in the file. You might have other internal classes in the file, but only one Monobehaviour should exist per file unless separated by namespaces.

4.3 Single-responsibility principle

Remember the goal is to keep each class short. In software design, the single-responsibility principle guides you toward simplicity.

The idea is that each module, class, or function is responsible for one thing. Suppose you want to build a game of *Pong*. You might start with classes for a paddle, a ball, and a wall.

For example, a Paddle class might need to:

— Store basic data about how fast it can move

— Check keyboard input

— Move the paddle in response

— Play a sound when colliding with a ball

Because the game design is simple, you could incorporate all of these thing into a basic paddle class. In fact, it's entirely possible to create on Monobehaviour that does everything you need.

Instead, break your Paddle class into smaller classes, each with a single responsibility. Separate data into its own PaddleData class or use a ScriptableObject. Then refractor everything else into a PaddleInput class, a PaddleMovement class, and a PaddleAudio class.

A PaddleLogic class can process the input from the PaddleInput. Applying the speed information from the PaddleData, it can shift the paddle using the PaddleMovement. Finally, the PaddleLogic can notify the PaddleAudio to play a sound when the ball collides with the paddle.

Each class does one thing in this redesign and fits into small, digestible pieces. You don't need to scroll through several screen to follow the code.

You'll still require a Paddle script but its sole job is to tie these other classes together. The bulk of the functionality is split into the other classes.

Note that clean code is not always the most compact code. Even when you use shorter classes, the total number of lines may increase during refactoring. However, each individual class becomes easier to read. When the time comes to debug or add new features, this simplified structure helps keep everything in its place.

5. Methods

Like classes, methods should be small with a single responsibility. Each method should describe one action or answer one question. It shouldn't do both.

A good name for a method reflects what it does. For example, `GetDistanceToTarget` is a name that clarifies its intended purpose. Try the following suggestions when you create methods for your custom classes:

— **Use fewer arguments:** Arguments can increase the complexity of your method. Reduce their number to make your methods easier to read and test.

— **Avoid excessive overloading:** You can generate an endless permutation of method overloads. Select the few that reflect how you will call the method and implement those. If you do overload a method, prevent confusion by making sure each method signature has a distinct number of arguments.

— **Avoid side effects:** A method only needs to do what its name advertises. Avoid modifying anything outside of its scope. Pass in arguments by value instead of by reference when possible. If sending back results via the `out` or `ref` keyword, make sure that's the one thing you intend the method to accomplish. Though side effects are useful for certain tasks, they can lead to unintended consequences. Write a method without side effects to cut down on unexpected behavior.

— **Instead of passing in a flag, make another method:** Don't set up your method to work in two different modes based on a flag. Make two methods with distinct names. For example, don't make a `GetAngle` method that returns degrees or radians based on a flag setting. Instead make methods for `GetAngleInDegrees` and `GetAngleInRadians`

5.1 The DRY principle: Don't repeat yourself

This oft-spoken mantra in software engineering advises programmers to avoid duplicate or repetitious logic. In doing so. If you follow the single-responsibility principle, you shouldn't need to change an unrelated piece of code whenever you modify a class or a method. Quashing a logical bug in a DRY program stops it everywhere.

The opposite of DRY is WET ("write everything twice"). Programming is WET when there are unnecessary repetitions in the code. Imagine two ParticleSystems (explosionA and explosionB) and two AudioClips (soundA and soundB). Each ParticleSystem needs to play with its own sound, which you can achieve with simple methods like this.

```
// EXAMPLE: WRITE EVERYTHING TWICE

private void PlayExplosionA(Vector3 hitPosition)
{
    explosionA.transform.position = hitPosition;
    explosionA.Stop();
    explosionA.Play();
    AudioSource.PlayClipAtPoint(soundA, hitPosition);
}

private void PlayExplosionB(Vector3 hitPosition)
{
    explosionB.transform.position = hitPosition;
    explosionB.Stop();
    explosionB.Play();
    AudioSource.PlayClipAtPoint(soundB, hitPosition);
}
```

One method is a cut-and-paste version of the other with a little text replacement. Though this works, you need to make a new method – with duplicate logic – every time you want to create an explosion.

Instead, refactor it into one PlayFXWithSound method like this:

```
// EXAMPLE: Refactored DRY version

private void PlayFXWithSound(ParticleSystem particle, AudioClip clip,
Vector3 hitPosition)
{
    particle.transform.position = hitPosition;
    particle.Stop();
    particle.Play();
    AudioSource.PlayClipAtPoint(clip, hitPosition);
}
```

Add more ParticleSystems and AudioClips and you can continue using this same method to play them in concert.

Note that it's possible to duplicate code without violating the DRY principle. It's more important that you don't duplicate logic.

Here, we've extracted the core functionality into the PlayFXWithSound method. If you need to adjust the logic, you only need to change it in one method rather than in both PlayExplosionA and PlayExplosionB

6. Comments

Well-placed comments enhance the readability of your code. Excessive or frivolous comments can have the opposite effect. Like all things, strike a balance when using them.

Most of your code won't need comments if you follow KISS principles and break your code into easy-to-digest logical parts. Well-named variables and functions will explain themselves.

Rather than answering "what," useful comments fill in the gaps and tell you "why." Did you make specific decisions that are not immediately obvious? Is there a tricky bit of logic that needs clarification? Useful comments reveal information not gleaned from the code itself.

Here are some dos and don'ts for comments:

— **Don't add comments to replace bad code:** If you need to add a comment to explain a convoluted tangle of logic, restructure your code to be more obvious. Then you won't need the comment.

— **A properly named class, variable, or method serves in place of a comment:** Is the code self-explanatory? Then reduce the noise and skip the comment.

```
// AVOID: noisy, redundant comments

// the target to shoot
Transform targetToShoot;
```

— **Place the comment on a separate line when possible, not at the end of a line of code:** In most cases, keep each one on its own line for clarity.

— **use the double slash (//) comment tag in most situations:** Keep the comment near the code that explains it rather than using a large mulit-line at the beginning. Keeping it close helps the reader connect the explanation with the logic.

— **Use a tooltip instead of a comment for serialized fields:** If your field in the Inspector need explanation add a tooltip attribute and skip the separate comment. The tooltip will do double duty.

```
// EXAMPLE: Tooltip replaces comment

[Tooltip("The amount of side-to-side friction.")]
[SerializeField] private float Grip;
```

— **You can also use a `summary XML tag` in front of public methods.**

```
// EXAMPLES:

// This is a common comment.
// Use them to show intent, logical flow, and approach.

// You can also use a summary XML tag.
//
/// <summary>
/// Controls the weapon system including firing, reloading, and dealing
 ↪  damage
/// </summary>
public void Fire()
{
    ...
}
```

— **Style your comments properly:** Insert one space between the comment delimiter (`//`) and the comment text. Begin each comment with an uppercase letter and end with a period.

— **Don't create formatted block of asterisks or special characters around comments:** This reduces readability and contributes to the general malaise of code clutter.

— **Remove commented out code:** Though commented out statements may be normal during testing and development, don't leave commented code lying around. Rely on the source control for previous versions of the code.

— **Keep your TODO comment up-to-date:** As you complete tasks, make sure you scrub the TODO comments you've left as a reminder. Outdated comments are distractions. You can add a name and date to a TODO for more accountability and context.

— **Avoid attributions:** You don't need to add bylines, e.g. `// added by devA or devB.` Let the source control take care of that.

— **Avoid journals:** The comments are not a place for your dev diary. There's no need to log everything you're doing in a comment when you start a new class. Proper use of a source control makes this redundant.