Please prefer the original source:
https://avangarde-software.com/unity-coding-guidelines-basic-best-practices/

This is just a backup offline copy-paste, in case the URL goes down

[Unity] Coding guidelines & Basic Best Practices

Have you ever wondered how a company organizes its code? Look no further, because here lie all your answers (probably not quite *all your answers*, but at least you'll take a look at how we do it and maybe learn from it). This is by no means a full comprehensive guide or a "this is the only way to do it" type of guide, it's more like an internal guideline that we use to stay organized and to be able to easily pass code from one person to another. This is **our** way of doing it and you are not required to follow these, unless you are an Avangarde Software employee, in which case it's mandatory.

Each style of writing code is personal and it feels like a customization option in a video game. Each person might do it differently to express themselves in a certain way (all the way from clean, organized, easily readable to messy but gets the job done). Nevertheless, one rule is a must for everyone and it will be the first topic of our discussion :

CONSISTENCY

There are primarily two types of consistency you should be concerned with : individual and collective consistency.

Individual consistency refers to your own set of rules that you use when you write code. It is important to use the same set of rules every time because it helps you read and understand your own code better, in case you ever need to come back to it at a later date. It also speeds up the process of writing code if you do it in the same way every time.

Collective consistency refers to when you need to write code in groups (2 or more people). Look at it like a common language between the members of the same team / company. It helps a lot when you can easily read and understand others code like it’s your own. In some cases it’s OK to use your own preferred style where you know others won’t interfere, but make sure you use the common styles in zones where you know others will work on. It’s considered rude to rewrite others code while they work on it without their permission, it’s better to just ask them to refactor or redo the code you consider unfit.

With this out of the way, let’s look at some examples of coding guidelines we like to use in different situations.

Naming conventions

Here is a basic set of rules for writing code that we use and is consistent within our company. This way, when we read others code, we don’t have to first translate it from their way or writing code to our way and only after that start understand what the in the world they tried to achieve there. Basically, it makes life easier, don’t you like it when it’s easy ? We will mostly use C# standards since we use C# in Unity (don’t use JavaScript/UnityScript, please ^_^). This means we will almost never use hyphens (–) or underscores (_).

Namespaces are all PascalCase

Bad :

```
avangarde.currentgame.ui.mainmenu
```

Good :

```
Avangarde.CurrentGame.UI.MainMenu
```

Classes & Methods are all PascalCase

Bad :

```
public class myclass
{
    public void somemethod{}
}
```

Good :

```
public class MyClass
{
    public void SomeMethod{}
  }
```

Fields are camelCase

(including **public** fields as per Unity convention, we'll talk more about these later in the post)

Static Fields should be PascalCase

Bad :

```
public static int _someStaticValue = 10;
```

Good :

```
public static int SomeStaticValue = 10;
```

Properties are PascalCase

Bad :

```
private string _name;
public string name
{
    get
    {
        return _name;
    }
    set
    {
        _name = value;
    }
  }
```

Good :

```
private string name;
public string Name
{
    get
    {
        return name;
    }
    set
    {
        name = value;
    }
```

```
}
```

Parameters should be camelCase

Bad :

```
public void HighlightElement(bool SomeCondition)
```

Good :

```
public void HighlightElement(bool someCondition)
```

Add Callback suffix to delegates

```
// Declare a delegate type for processing a user:
public delegate void ProcessUserCallback(User u);
```

Add On prefix to events and actions

```
public UnityAction OnDeath;
```

Basic Best Practices

Declarations

Always use access level modifiers. Do not omit them. If you don’t write any access modifier, the compiler will assume it’s private, but that doesn’t mean you shouldn’t write the keyword private.

Bad :

```
int privateVariable;
```

Good:

```
private int privateVariable;
```

Remember that we talked about **consistency**? Well, when everything that needs to be public is marked with its appropriate access modifier (public), then everything that should be private should also be marked as private. This brings us straight to the following topic.

Use single declaration per line. It helps with keeping everything clear.

Bad :

```
private int firstVariable, secondVariable;
```

Good :

```
private int firstVariable;
  private int secondVariable;
```

Declare only one **class** / **interface** per source file, with the exception of **internal** classes. You may end up with more files, but it's clear and you can easily find what you are searching for.

Preface interfaces with an **I**

Bad :

```
PointerDown
```

Good :

```
IPointerDown
```

Spacing

Keeping everything spaced out properly will not only increase the speed at which you're reading the code, but also increase readability and helps with clear understanding of code logic. When we are talking about spacing, multiple cases can arise.

Unity comes with Visual Studio Community edition and the standard used in Visual Studio is generally good enough, so you won't have to do a lot here. After you open a project with Visual Studio, go to **Edit -> Advanced -> View White Spaces**, this will help you visualize spacing. One space is marked with a dot and tabs are marked with an arrow. By default, Visual Studio will treat tabs as 4 spaces. You can change this option to treat tabs as actual tabs and change the length of tabs, but the default value of 4 is what we're looking for. Some people prefer using only spaces, others will maybe use tools that automatically analyze the code and require for you to use tabs. This is again a personal preference, so as long as the code looks clean and correctly indented you should be good to go.

Use **Vertical Spacing** to organize code within a class. Keep one blank line between methods to better visualize the structure of the class and keep a blank line inside methods to separate functionality (for example : a method that creates an object should have its creation part and initialization part separated). If you have many logical parts inside the same method, you might consider refactoring and creating multiple methods, each doing its own job.

Brace style

This is a topic that many people have different opinions on. As much as we would like to do it our own way, the C# convention states : all braces get their own line.

Bad :

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

Good :

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

The “case” statements should be indented from the switch statement like this:

```
switch (someExpression)
{

   case 0:
      {
          DoSomething();
      }
      break;

   case 1:
      {
           DoSomethingElse();
      }
      break;
```

```
   case 2:
      {
         int n = 1;
         DoAnotherThing(n);
      }
      break;
}
```

Also, **always** use braces. Yes, the compiler will work if you only have 1 line of code, but then again, we're talking about improving code readability and this one helps a lot.

Bad :

```
for(int i=0; i<10; i++)
    ExecuteSomething();
```

Good :

```
for(int i=0; i<10; i++)
{
    ExecuteSomething();
}
```

Public fields

Generally, when people discuss about OOP (Object Oriented Programming) principles, public fields are not something recommended.

```
public int publicVariable;
```

These can create a lot of problems due to their access level. Since code from outside of the class can access this, you can easily make the mistake of modifying certain parts from who knows where and then it will be hard to debug if a problem arises. On the other hand, Unity team recommends using public fields in their documentation since they are automatically serialized and displayed in the inspector. This is more of a preference, as long as you are careful when writing code, the fact that some fields are public shouldn't be a problem. The proper OOP way of doing this would be to make everything private. If you need something to be accessed from another class, make a property with a get and a set method.

```
private string name;
public string Name
{
    get
    {
        return this.name;
    }
    set
    {
        this.name = value;
    }
}
```

This method requires a private field and it creates a public property available from other portions of your code. Inside these **get** and **set** methods, you can write additional code that is called when this property is read or wrote. By removing one of these methods, you can either make certain properties **read-only** or **write-only**. Also you can set one of them as private if you want (for example) to be able to access a property from outside, but change its value only from inside.

There is another type of property and it's called **auto-implemented property**. Creating a proeprty in this way it makes it behave exactly like a field.

```
public string Name { get; set; }
```

However, if you need something private to be displayed in the **Inspector**, there is a way of doing it and it's by using the **[SerializeField]** attribute

```
[SerializeField]
private bool hasHealthPotion = true;
```

This attribute can serialize private fields but it cannot serialize properties or static fields.

Coding Best Practices

General Coding

Put all your code in a **namespace**. This avoids code clashes among your own libraries and third-party code. But don't rely on namespaces to avoid clashes with important classes. Even if you use different namespaces, don't use "Object" or "Action" or "Event" as class names.

Use **assertions**. Assertions are useful to test invariants in code and help flush out logic bugs. Assertions are available in the **Unity.Assertions.Assert** class. They all test some condition, and write an error message in the console if the condition is not met.

DO NOT use **strings** for anything other than displayed text**.** In particular, do not use strings for identifying objects or prefabs. There are exceptions (there are still a few things that can only be accessed by name in Unity). In such cases, define those strings as constants in files such as AnimationNames or AudioModuleNames.

DO NOT use **Invoke** and **SendMessage.** These methods of MonoBehaviour call other methods by name. Methods called by name are hard to track in code (you cannot find “Usages”, and SendMessage has a wide scope that is even harder to track). Use **Coroutines** instead.

```
private static IEnumerator InvokeImpl(Action action, float time)
{
    yield return new WaitForSeconds(time);

    action();
}
```

DO NOT let **spawned objects** clutter your hierarchy when the game runs**.** Set their parents to a scene object to make it easier to find stuff when the game is running. You could use an empty game object as their parent.

If you have a lot of text, put it in a file. **Don’t** put it in fields for editing in the inspector. Make it easy to change without having to open the Unity editor, and especially without having to save the scene.

Unity Best Practices

Now it’s time to list some of the things that are specific to Unity and help boost either application performance or code readability. For further information you can either check the documentation of the methods, the references list or try them for yourself and see the benefits.

DO NOT use `GameObject.Find();`. As much as you can, develop your project without the use of this method as it's very slow. If you really need to use it, try using it in methods that are called when necessary (for example *Start()* for initialization or maybe in an event that you define, but NEVER in *Update()*)

DO NOT use `someObject.GetComponent()` in **Update**. It's very inefficient to use it in this manner. Cache the components you want to access in private properties and use them like that.

Avoid using public index-coupled arrays. For instance, do not define an array of weapons, an array of bullets, and an array of particles, so that your code looks like this:

```
public void SelectWeapon(int index)
{
    currentWeaponIndex = index;
    Player.SwitchWeapon(weapons[currentWeaponIndex]);
}

public void Shoot()
{
    Fire(bullets[currentWeaponIndex]);
    FireParticles(particles[currentWeaponIndex]);
}
```

The problem for this is not so much in the code, but rather setting it up in the inspector without making mistakes.

Rather, define a class that encapsulates the three variables, and make an array of that:

```
[Serializable]
public class Weapon
{
    public GameObject prefab;
    public ParticleSystem particles;
    public Bullet bullet;
}
```

The code looks neater, but most importantly, it is harder to make mistakes in setting up the data in the inspector.

Avoid using arrays for structure other than sequences. For example, a player may have three types of attacks. Each uses the current weapon, but generates different bullets and different behaviour.You may be tempted to dump the three bullets in an array, and then use this kind of logic:

```
public void FireAttack()
{
    Fire(bullets[0]);
}
```

```
public void IceAttack()
{
    Fire(bullets[1]);
}

public void WindAttack()
{
    Fire(bullets[2]);
}
```

Enums can make things look better in code…

```
public void WindAttack()
{
    Fire(bullets[WeaponType.Wind]);
}
```

…but not in the inspector.

It's better to use separate variables so that the names help show which content to put in. Use a class to make it neat.

```
[Serializable]
public class Bullets
{
    public Bullet fireBullet;
    public Bullet iceBullet;
    public Bullet windBullet;
}
```

Use singletons for convenience. Singletons are useful for managers, such as `AudioManager` or `GameManager`.

Avoid using singletons for unique instances of prefabs that are not managers (such as the Player). Not adhering to this principle complicates inheritance hierarchies, and makes certain types of changes harder. Rather keep references to these in your GameManager

Define **static properties** and **methods** for public variables and methods that are used often from outside the class. This allows you to write GameManager.Player instead of GameManager.Instance.player.

References :

http://wiki.unity3d.com/index.php/Csharp_Coding_Guidelines

https://github.com/raywenderlich/c-sharp-style-guide

https://teamtreehouse.com/community/what-is-the-difference-between-public-vs-serializefield-and-when-to-use-it

https://trello.com/b/Z6cDRyis/good-coding-practices-in-unity