Naming Convention Unity

Created by	Marion
Created time	@April 18, 2024 1:54 PM
Tags	Principles Unity

Naming Convention

Public variable/Member of a class	PascalCase
Private variable/Member of a class	_camelCase
Local variable/Method parameter	camelCase
Function name	PascalCase
Const variable	CONST_VARIABLE
Static variable	s_PascalCase
Enum name & value	PascalCase
Class name	PascalCase
Events name & handlers	OnExampleEvent & OnExample

Fields and variables

- **Use nouns for variable names**: Variable names should be clear and descriptive because they represent a specific thing or state. Boolean are an exception (see below).
- **Prefix booleans with a verb**: These variables indicate a true or false value. Often they are the answer to a question, such as: Is the player running? Is the game over? Prefix them with a verb to make their meaning more apparent. Often this is paired with a description or condition, e.g., isDead, isWalking, hasDamageMultiplier**...**
- **Use meaningful names. Don't abbreviate (unless it's math):** Your variable names will reveal their intent. Choose names that are easy to pronounce and search for. While single-letter variables are fine for loops and math expressions, don't abbreviate in other situations. Clarity is more important than any time saved from omitting a few vowels.

Const & Static Variables

Const: A *const* variable is one whose value cannot be changed.

```
public const int MaxLives = 3;
```

Static: A *static* variable in Unity is a variable that is shared by all instances of a class. Meaning that, even if there are multiple instances of a script, on many different objects, they will all share the same static variable value. To mark a variable as static, simply add the static keyword when declaring it.

```
public class PlayerHealth : MonoBehaviour
{
    public static float s_Health=100;
}

public class PlayerStats : MonoBehaviour
{
        private float hp = PlayerHealth.s_Health;
}
```

The static keyword can be used to create singletons: components that have only one instance in a given time.

```
public class AudioManager : Monobehaviour
{
        public static AudioManager s_Instance;
}

public class Car : Monobehaviour
{
        private Start()
        {
                AudioManager.s_Instance.Play("Honk.mp3");
```

```
        }
}
```

Access Level Modifiers

Always specify if your variable is *private*, *protected* or *public*.

If the access modifier is left off, the compiler assumes the variable is private.

- *public*: In this, access is not restricted.
- *protected*: For this access is limited to the containing class or derived from the containing class within the current assembly
- *internal*: In this case, access is limited to the current assembly.
- *private*: Access is limited to the containing class.

Enums

Enums are special value types defined by a set of named constants. By default, the constants are integers, counting up from zero.

You can place public enums outside of a class to make them global. Use a singular noun for the enum name.

```
public enum WeaponType
{
    Knife,
    Gun,
    RocketLauncher,
    BFG
}

public enum FireMode
{
    None = 0,
    Single = 5,
    Burst = 7,
    Auto = 8,
}
```

Methods

Methods perform action.

- **Start the name with a verb:** Add context if necessary (e.g., GetDirection, FindTarget, etc.).
- **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);
}
```

Events, Event Raisers and Event Handlers

Name the event with a verb phrase. Be sure to choose one that communicates the state change accurately.

Use the present or past participle to indicate the state of events as before or after: For example, specify "*OnOpeningDoorEvent*" for an event before opening a door and "*OnDoorOpenedEvent*" for an event afterward.

The *RaiseMethod()* methods are used when we want to raise the event from another class. If the event is only raised in its declaring class, there is no need for such method.

```
//EXAMPLE: Events
//Using System.Action delegate

//Class that declares the events
public class ExampleClass1 : MonoBehaviour
```

```
{
        //Declaring the events
        public event Action OnDoorOpenedEvent;
        public event Action<int> OnPointsScoredEvent;

        //Methods that call the events
        public void RaiseDoorOpenedEvent()
        {
            OnDoorOpenedEvent?.Invoke();
        }

        public void RaisePointsScoredEvent(int points)
        {
            OnPointsScoredEven?.Invoke(points);
        }

}

//Class that subscribes to the events
public class ExampleClass2 : Monobehaviour
{
        //Subscribing to the events
        private void Awake()
        {
                ClassContainingTheEvent.OnDoorOpenedEvent +=
                OnPointsScoredEvent += IncrementScore;
        }

        //Methods subscribed to the events
        public void OnDoorOpened()
        {
                AudioManager.s_Instance.Play("door-creaking.m
        }

        public void OnPointsScored(int points)
        {
            score += points;
        }
```

```
        //Unsubscribing to the events
        private void Destroy()
        {
                OnDoorOpenedEvent -= PlayDoorCreak;
                OnPointsScoredEvent -= IncrementScore;
        }
}
```

Custom EventArgs

Create custom EventArgs only if necessary. If you need to pass custom data to your Event, create a new type of EventArgs, either inherited from **System.EventArgs** or from a custom struct.

```
// EXAMPLE: Read-only, custom struct used to pass an ID and C
public struct CustomEventArgs
{
    public int ObjectID { get; }
    public Color Color { get; }

    public CustomEventArgs(int objectId, Color color)
    {
        this.ObjectID = objectId;
        this.Color = color;
    }
}

//Declaring the event
public event Action<CustomEventArgs> OnEvent;
```

This document draws inspiration from the *Naming anf Code Style Tips for C# Scripting in Unity*:

Naming and Code Style Tips for C# Scripting in Unity

Pick up best practices for styling your C# code in Unity. These tips can help you and your team create a cleaner, more readable and scalable codebase.

https://unity.com/how-to/naming-and-code-style-tips-c-scripting-unity