Code conventions - Programming in C#

👋 Welcome peeps.

GitHub rules for programming.

When you’re assigned a task in GitHub in “Issues”, you can create a branch from there since it’s easier than naming your own branches.

Steps:

1. Click “create a branch”
2. Create branch

Since you don’t have to create your own issues (maybe you have to) you will be assigned tasks from GitHub for this.

Definition of done

When working on your research, tell the rest what will you have at the end of it. It’s also for yourself so you can see what you’ve done.

Example: Player Pick-up object

- [ ] Play pick-up animation
- [ ] Add weight to the object
- [ ] Create a scriptable object to store data

Every issue will have a definition of done.

Working in GitHub

Work with labels in GitHub. Add the appropriate label to the issue.

Code conventions

Variables

Var

Using var instead of the datatype name.

```
//This

Dictionary<String, Vector3> locations = new Dictionary<String, Vector3>()

//Can be this
var locations = new Dictionary<String, Vector3>();
```

This way we can read the code more clearly. Not for ambiguous or primitive cases:

```
//This
int count = GetCount();   // clearer than var
```

private global variables

```
//This
private float movementSpeed;

//Can be this
private float _movementSpeed;

// Do not use this for [SerializeFields]
```

Naming of methodes / classes. Defining [Serializedfields]

In C# methode names start with UpperCaseLetters, also very basic stuff but give your methode names a proper and fitting name. Also don't forget to add modifiers to your methodes: public e.g.

```
//This
private float getMovementSpeed() {}

//Must be this
private float GetMovementSpeed() {}
```

When your using [SerializeField], try using [Headers(”something”)]

```
//This

[SerializeField] private float movementSpeed = 5f;

//Must be this, also do not define base value in the IDE.
//Let gameplay define the movementspeed for example, don't overwork urs
elf 0_-

[Header("Player movement setting")]
[SerializeField] private float movementSpeed;
```

Spacing and Indentation

```
//One space after commas and around operators.
//No extra blank lines between logically grouped statements.

//This
void MovePlayer()
{
    float horizontal = Input.GetAxis("Horizontal");

    float vertical = Input.GetAxis("Vertical");

    Vector3 direction = new(horizontal, 0, vertical);



    _rigidbody.MovePosition(transform.position + direction * _speed * Time.
deltaTime);
```

```
    _animator.SetFloat("Speed", direction.magnitude);
}

//Must be this
private void MovePlayer()
{
    // Input
    float horizontal = Input.GetAxis("Horizontal");
    float vertical = Input.GetAxis("Vertical");
    Vector3 direction = new(horizontal, 0, vertical);

    // Applying the input
    _rigidbody.MovePosition(transform.position + direction * _speed * Time.
deltaTime);

    // Play animations
    _animator.SetFloat("Speed", direction.magnitude);
}
```

Unity MonoBehaviour lifecycle.

```
//Serialized fields
[SerializeField] private float speed;

//Components
private Rigidbody _rigidbody;
private Animator _animate;

//Unity lifecycle methods
private void Awake() { ... }
private void OnEnable() { ... }
private void Start() { ... }
private void Update() { ... }
private void FixedUpdate() { ... }
private void OnDisable() { ... }
private void OnDestroy() { ... }
```

```
//Public methods
public void Move(Vector3 direction)
{
//Call the private helpers
}

//Private helpers
private void HandleMovement()
{
//Handle movement
}
```

Expression bodied member

Lambda

Read-only properties

```
public bool IsAlive => _health > 0f;
public float Distance => Vector3.Distance(_a, _b);
```

Methods: for short return statements

```
public int GetScore() => _kills * 100;
public string GetNameTag() => $"{_playerName}#{_id}";
```

Constructors and Finalizers

```
public Player(string name) => _playerName = name;
~Player() => Debug.Log("Destroyed");
```

Indexers

```
public int this[int i] => _items[i];
```

Expression bodied version

```
//This
public float Speed
{
    get { return _speed; }
}

//Can be this
public float Speed => _speed;
```

Use it when:

- The member is simple, usually a one-liner
- It improves readability by reducing clutter

```
//Too much going on, not readable
public void Move() => transform.Translate(Vector3.forward * _speed * Time.
deltaTime);
public void Heal(float amount) => _health = Mathf.Min(_health + amount, 10
0f);
public void Damage(float dmg) => _health -= dmg;

//Must be this
public void Move()
{
    transform.Translate(Vector3.forward * _speed * Time.deltaTime);
}

public void Heal(float amount)
{
    _health = Mathf.Min(_health + amount, 100f);
}

public void Damage(float dmg)
{
    _health -= dmg;
}
```

Magic numbers in code

So basically don’t use magic numbers meaning:

```
//This

public float SetMovementSpeed(float speed)
{
   speed = 5;
}

//Must be this

private const float MovementSpeed = 5; //Also PascalCase

private readonly

public void SetMovementSpeed(float speed) ⇒ _speed = speed;
public float GetMovementSpeed() ⇒ _speed;
```

Naming conventions

Cases

```
//Classes and structs
public class PlayerController : MonoBehaviour //class

//struct
 struct PlayerData {
    int health;
    int level;
    int experience;
    string name;

    public void LogPlayer() {
      //.....
    }
    }
```

```
//PascalCase with I (Interface)
public interface IMoveable

//Enums
public enum GameState
{
    MainMenu,
    Playing,
    Paused,
    GameOver
}

//Methodes
private void TakeDamage() {}

//Public fields / Properties
public float MoveSpeed
{
    get;
    set;
}

//Constants
//Unlike Java the constants or definitive values arent all caps.

public const float CurrentSpeed = 5f;

//For local variables / parameteres

float deltaTime;
```

Naming of packages

Script packages are all lowercase!! We will be using folders by feature not type

So basically player folder: Scripts → player → PlayerController

Using namespaces

```
/* Use of name spaces in code lets say we have a package called controlle
r inside
there is another package called player, we are saying this controller is for pl
ayer.
So in the PlayerController script we do the following.
We give the namespace the name of the game, as per Unity clearer guideli
nes.
*/

namespace WhileYourHere.player
// WhileYourHere can be capitilized since it's the name of the game.
// Also important do not create a folder in script called "WhileYourHere"
{
        public class PlayerController : MonoBehaviour
        {

        [SerilizedField] private float speedMovement;

        private void SomeMethode()
        {
            // Some logic
        }
    }
}
```

Why do we use it?

- It's mostly used for an internal organized system, inside our project folder.
- A way of presenting program elements that are exposed to other programs.
- Makes auto-completion cleaner.
- Mirrors folder structure = easy to understand.
- Prevents naming conflicts

New stuff

```
// Use can use [range(10,100)] for objects for example

```

```
[Header("Object information")]
[Range(10,100)]
[SerilizedField] private float objectWeight;
```

Unity Guideline

Install latest version of Unity from Unity Hub one with no security issues.
6000.2.8f1
Just clone the repo from GitHub and it will tell you that your missing a version.

Scriptable objects

Store most of your programmed events in a scriptable object, that way people who are depended on your task, can easily access it.

Creating a scriptable object

```
//Give your scriptable objects proper names!

[CreateAssetMenu(fileName="MyScriptableObject")]

public class MyScript : ScriptableObject
{
   public int somevalue;
   public Actions/UnityEvents blsl;

//Calling the scriptableobject

[SerializeField] private MyScript myScript;
```

The most common use for ScriptableObjects is as data containers for shared static data, particularly for game configuration data that

In scriptable objects you can store the following:

— Store them in variables
— Dynamically create or destroy them at runtime
— Pass them as arguments
— Return them from a

doesn’t change at runtime.		method — Include them in data structures — Serialize/deserialize them

Typical use for scriptable objects

— Inventories like item type, icon, rarity, effects
— Enemy, player, or item statistics like health, damage, speed, AI parameters
— Audio collections like audio clip groups for footsteps, UI sounds, or ambient loops
— Config Files like difficulty settings, progression curves, spawn tables
— Dialogue data

— Keep data-only classes separate from logic classes.

Testing ScriptableObjects events in Unity inspector

I got this from the lecture from week 8, I’ll try my best to explain it.

```
// We begin with defining a ScriptableObject class

[CreateAssetMenu(fileName = "NewParameterEventChannelInstance",
 menuName = "Events/Parameter Event Channel")]
public class ParameterEventChannel<T> : ScriptableObject
// The <T> in ParameterEventChannel stand for type of parameter.
// We use <T> Because we want the same event system, but for different ty
pes of events.

{
    public UnityAction<T> OnEventRaised;
    //or
    public Action OnEventRaised;

    public void RaiseEvent(T parameter)
    {
        Debug.Log(name + " was raised");
        if (OnEventRaised == null)
            return;

        OnEventRaised.Invoke(parameter);
    }
}
```

After we created the main ScriptableObject class we can start defining what will happen if we raise the events.

```
[CreateAssetMenu(fileName = "NewIntParameterEventChannelInstance",
menuName = "Events/Int Parameter Event Channel")]
public class IntParameterEventChannel : ParameterEventChannel<int>
{
    // Here we call the previous ScriptableObject class and define it with an i
nt.
}
```

Using the modern UIElements System in Unity.

Tell Unity this **Editor** class replaces the default inspector for **IntParameterEventChannel** instances.

```
[CustomEditor(typeof(IntParameterEventChannel))] // → Telling Unity
public class IntParameterEventChannelEditor : Editor
```

Class fields

```
private IntParameterEventChannel m_EventChannel; // Refrence to object
private Label m_ListenersLabel;
private ListView m_ListenersListView;
private IntegerField m_IntParameterValueField;
private Button m_RaiseEventButton;

// The other fields are UI elements the inpector creates and reuses.
```

Enabling the ScriptableObject

```
private void OnEnable()
{
    if (m_EventChannel == null)
        m_EventChannel = target as IntParameterEventChannel;
        // Telling the target that it's suppose to call the IntParameterEventChan
nel.
}
```

CreateInspectorGUI

This is the UIElements entry point for a custom inspector

```
public override VisualElement CreateInspectorGUI()
{
    var root = new VisualElement();

    // Draw default inspector fields:
    InspectorElement.FillDefaultInspector(root, serializedObject, this);
    // Also adds the default inspector for SerializedField

    // Add space, a label "Listeners:" and a ListView bound to GetListeners()
    // Add IntegerField and a Button which calls RaiseEvent when clicked
    return root;
}
```

MakeItem and BindItem

```
// Each row is a VisualElement with a single label child
private VisualElement MakeItem()
{
    var element = new VisualElement();
    var label = new Label();
    element.Add(label);
    return element;
}
```

```
// Fetch the current listeners list and sets the label text to the name & type
// It also registers a mouse callback to ping the object.
```

```
private void BindItem(VisualElement element, int index)
{
    List<MonoBehaviour> listeners = GetListeners();
    var item = listeners[index];
    Label label = (Label)element.ElementAt(0);
    label.text = GetListenerName(item);

    label.RegisterCallback<MouseDownEvent>(evt =>
    {
        EditorGUIUtility.PingObject(item.gameObject);
    });
}
```

Important

Registering callbacks inside `BindItem` can add multiple callbacks over time because `ListView` reuses item elements. Each time `BindItem` runs, it will re-register another callback on the same UI element.

GetListenerName

```
// Nicely formats GameObjectName (ComponentType)
private string GetListenerName(MonoBehaviour listener)
{
    if (listener == null)
        return "<null>";

    return listener.gameObject.name + " (" + listener.GetType().Name + ")";
}
```

How to find subscribers to the event

```
/*
Casting Target to MonoBehaviour because listeners are expected to be co
mponent
instances
*/
private List<MonoBehaviour> GetListeners()
{
    List<MonoBehaviour> listeners = new List<MonoBehaviour>();
```

```
        // m_EventChannel is assumed to be a multicast delegate.
    if (m_EventChannel == null || m_EventChannel.OnEventRaised == null)
        return listeners;

        // GetInvocationList returns an array of Delegate representing each su
bscribed methode
    var delegateSubscribers = m_EventChannel.OnEventRaised.GetInvocatio
nList();

    foreach (var subscriber in delegateSubscribers)
    {
            // subscriber.Target give the object instance target for instance met
hodes.
            // If the subscriber is a static methode, target is null
        var componentListener = subscriber.Target as MonoBehaviour;
        // Filter duplicates
        if (!listeners.Contains(componentListener))
        {
            listeners.Add(componentListener);
        }
    }

    return listeners;
}
```

Important notes from ScriptableObjects

Performance → *GetInvocationLis()* and building lists are cheap in editor UI, but if you refresh every frame or very often it could be wasteful.

Static subscribers → Decide wether you want to show static methode subscribers

(they have Target == null)

Destroyed objects → If a subscribed component was destroyed but the delegate still references it, you may need to detect *target == null* or *taget.Equals(null)* (Unity's fake null pattern)

**Editor-only → ** Keep this in a an Editor folder. Don’t ship editor in scripts in builds

Threading → Editor UI runs on main thread. Do not attempt to raise events from background threads via this inspector.

Final version

Defensive null checks unlike the architect lecture

```
using System;
using System.Collections.Generic;
using UnityEngine;
using UnityEditor;
using UnityEngine.UIElements;
using UnityEditor.UIElements;

[CustomEditor(typeof(IntParameterEventChannel))]
public class IntParameterEventChannelEditor : Editor
{
    private IntParameterEventChannel m_EventChannel;

    private Label m_ListenersLabel;
    private ListView m_ListenersListView;
    private IntegerField m_IntParameterValueField;
    private Button m_RaiseEventButton;
    private List<MonoBehaviour> m_ListenersCache = new();

    private void OnEnable()
    {
        m_EventChannel = target as IntParameterEventChannel;
    }

    public override VisualElement CreateInspectorGUI()
    {
        var root = new VisualElement();

        InspectorElement.FillDefaultInspector(root, serializedObject, this);

        root.Add(new VisualElement { style = { marginBottom = 20 } });
        // Header
        m_ListenersLabel = new Label("Listeners:");
        m_ListenersLabel.style.borderBottomWidth = 1;
        m_ListenersLabel.style.borderBottomColor = Color.grey;
        m_ListenersLabel.style.marginBottom = 2;
        root.Add(m_ListenersLabel);

        // Initial fill of cache
        UpdateListenersCache();

        // ListView: use the cached list so we can update it in-place
        m_ListenersListView = new ListView(m_ListenersCache, 20, MakeListe
nerItem, BindListenerItem)
        {
            selectionType = SelectionType.None,
            style = { marginBottom = 8 }
        };
        root.Add(m_ListenersListView);

        // Integer field
        m_IntParameterValueField = new IntegerField("Parameter Value") { val
ue = 0, style = { marginTop = 8 } };
        root.Add(m_IntParameterValueField);

        // Raise button
        m_RaiseEventButton = new Button(() => RaiseEventFromInspector()) { t
ext = "Raise Event" };
        m_RaiseEventButton.style.marginTop = 5;
        root.Add(m_RaiseEventButton);

        // Optional: Refresh button to update listeners
        // Update the list on demand via this button.
        var refreshButton = new Button(() => { UpdateListenersCache(); m_List
enersListView.Refresh(); }) { text = "Refresh List" };
        root.Add(refreshButton);

        return root;
```

```
    }

    private VisualElement MakeListenerItem()
    {
        var container = new VisualElement();
        var label = new Label();
        label.name = "listenerLabel";
        label.style.unityTextAlign = TextAnchor.MiddleLeft;

        // Use Clickable manipulator and store the label on the container.
        container.Add(label);
        container.userData = label; // store label for easy retrieval
        container.AddManipulator(new Clickable(() ⇒
        {
            // Click action uses the userData index set in BindListenerItem
            var lbl = (Label)container.userData;
            var listener = lbl.userData as MonoBehaviour;
            if (listener != null)
                EditorGUIUtility.PingObject(listener.gameObject);
        }));

        return container;
    }

    private void BindListenerItem(VisualElement element, int index)
    {
        if (index < 0) return;

        // Ensure cache is up-to-date (defensive)
        UpdateListenersCache();

        var label = (Label)element.Q<Label>("listenerLabel");
        var listener = (index < m_ListenersCache.Count) ? m_ListenersCache[i
ndex] : null;

        label.userData = listener; // store for clickable action
        label.text = FormatListenerName(listener);
    }
    private string FormatListenerName(MonoBehaviour listener)
    {
        if (listener == null) return "<null>";
        return $"{listener.gameObject.name} ({listener.GetType().Name})";
    }

        // Filters out non-MonoBehaviour and duplicates explicity.
    private void UpdateListenersCache()
    {
        m_ListenersCache.Clear();

        if (m_EventChannel == null) return;
        var del = m_EventChannel.OnEventRaised;
        if (del == null) return;

        var invocationList = del.GetInvocationList();
        foreach (var d in invocationList)
        {
            var target = d.Target as MonoBehaviour;
            if (target != null && !m_ListenersCache.Contains(target))
            {
                m_ListenersCache.Add(target);
            }
        }
    }

    private void RaiseEventFromInspector()
    {
        if (m_EventChannel == null) return;
        m_EventChannel.RaiseEvent(m_IntParameterValueField.value);
    }
}
```

LINQ (Language Intergrated Query

With LINQ you can query, filter and transform data (like array's, lists, dictionaries, etc.) in a very clean and readable way. Kind of like SQL query's.

```
// To use LINQ
using System.Linq;
```

Basic example

```
using System.Linq;
using UnityEngine;
using System.Collections.Generic;

public class LinqExample : MonoBehaviour
{
    private void Start()
    {
        List<int> numbers = new() { 1, 2, 3, 4, 5, 6 };

        // Filter: get even numbers
        var evens = numbers.Where(n => n % 2 == 0);

        // Print them
        foreach (var n in evens)
            Debug.Log(n);
    }
}

/*
Output will be:
2
4
6
*/
```

Common LINQ Methodes

Where() → Filter elements → players.Where(p => p.Health > 0)

Select() → Transform each element → names.Select(n => n.ToUpper())

OrderBy/OrderByDecending → Sort data → scores.OrderByDescending(s => s)

First() / FirstOrDefault() → Get first element → players.First(p => p.IsAlive)

Any() / All() → Check conditions → enemies.Any(e ⇒ e.IsBoss)

Count() → Count elements → coins.Count(c ⇒ c.Collected)

Distinct → Remove duplicates → items.Distinct()

Sum() / Average() / Max() / Min() → Aggregations → scores.Max()

ToList() / ToArray() → Materialize results → alivePlayers.ToList()

Unity example

```
using UnityEngine;
using System.Linq;

public class LinqUnityExample : MonoBehaviour
{
    private void Start()
    {
        GameObject[] allObjects = FindObjectsOfType<GameObject>();

        // Find all active enemies with tag "Enemy"
        var enemies = allObjects
            .Where(obj ⇒ obj.CompareTag("Enemy") && obj.activeInHierarchy)
            .ToList();

        Debug.Log($"Active enemies: {enemies.Count}");
    }
}
```

Rules of LINQ

> LINQ is slower than loops for huge datasets (GC allocations).
>
> Use it for clarity, not heavy per-frame operations.

> Don’t use LINQ in Update() — prefer caching or pre-processing.

> You can mix LINQ with ScriptableObjects, lists of components, and data-driven systems.

> Remember to using System.Linq; — otherwise, methods like .Where() won’t appear.

Finding closest enemy

```
using UnityEngine;
using System.Linq;

public class EnemyFinder : MonoBehaviour
{
    void Start()
    {
        var enemies = GameObject.FindGameObjectsWithTag("Enemy");

        var closest = enemies
            .OrderBy(e => Vector3.Distance(transform.position, e.transform.posi
tion))
            .FirstOrDefault();

        if (closest != null)
            Debug.Log($"Closest enemy: {closest.name}");
    }
}
```

Scenes

Work in separate scenes for everything. Do not use the **MainScene** NEVER!!