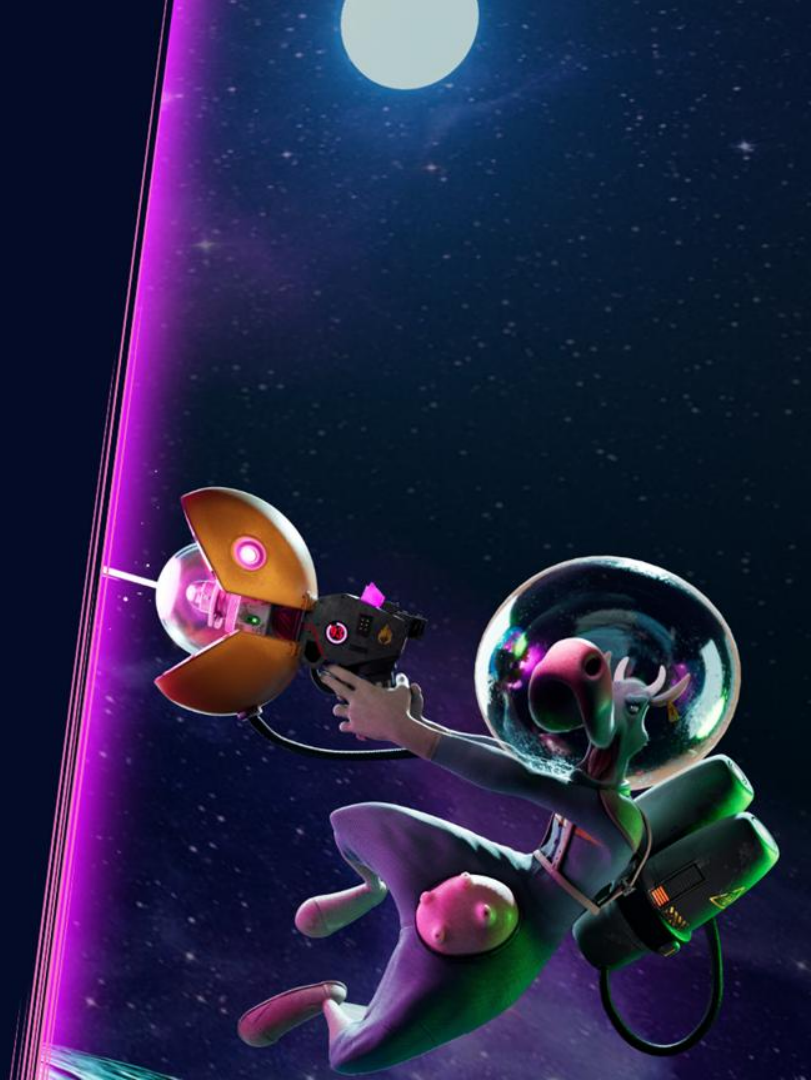
Advanced Programming

Good practices

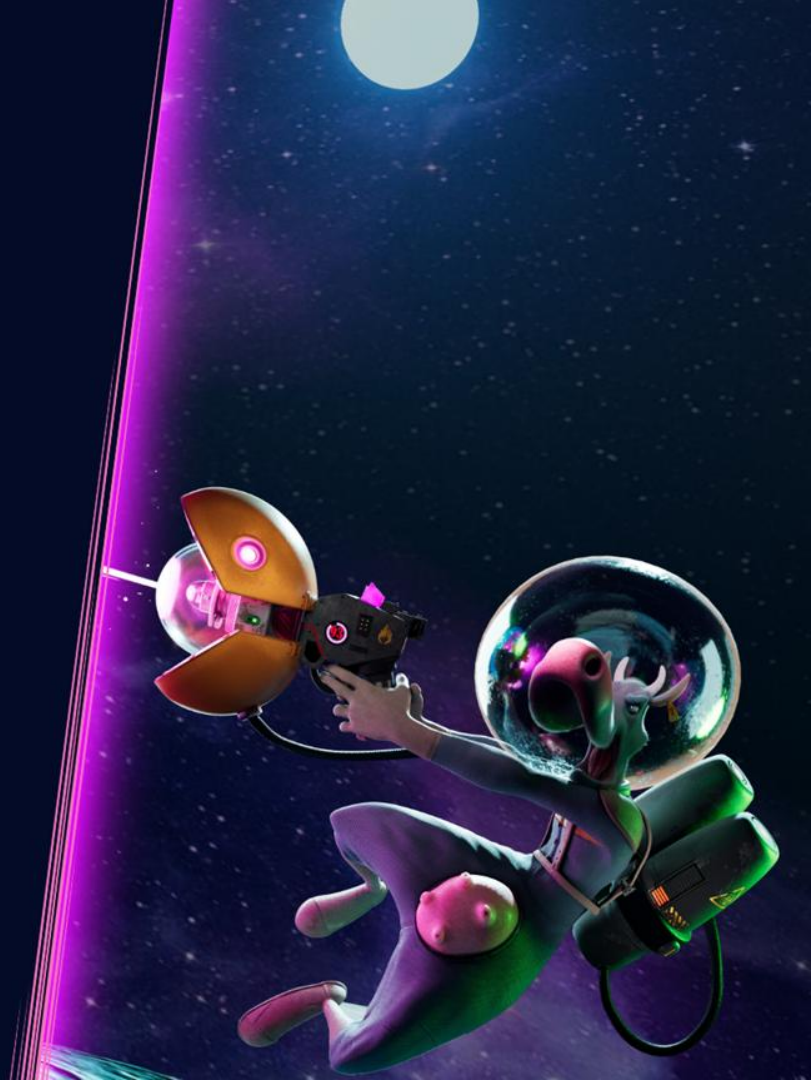
Samuel Styles - 05.08.2025



- Board member of the SGA (Swiss game academy)
- Gold member of the SGC (Swiss game center)

[SStyles93 (Samuel Styles) (github.com)]

[Samuel Styles Blogposts – Blogposts about my studies (sstyles93.github.io)]



Advanced Programming

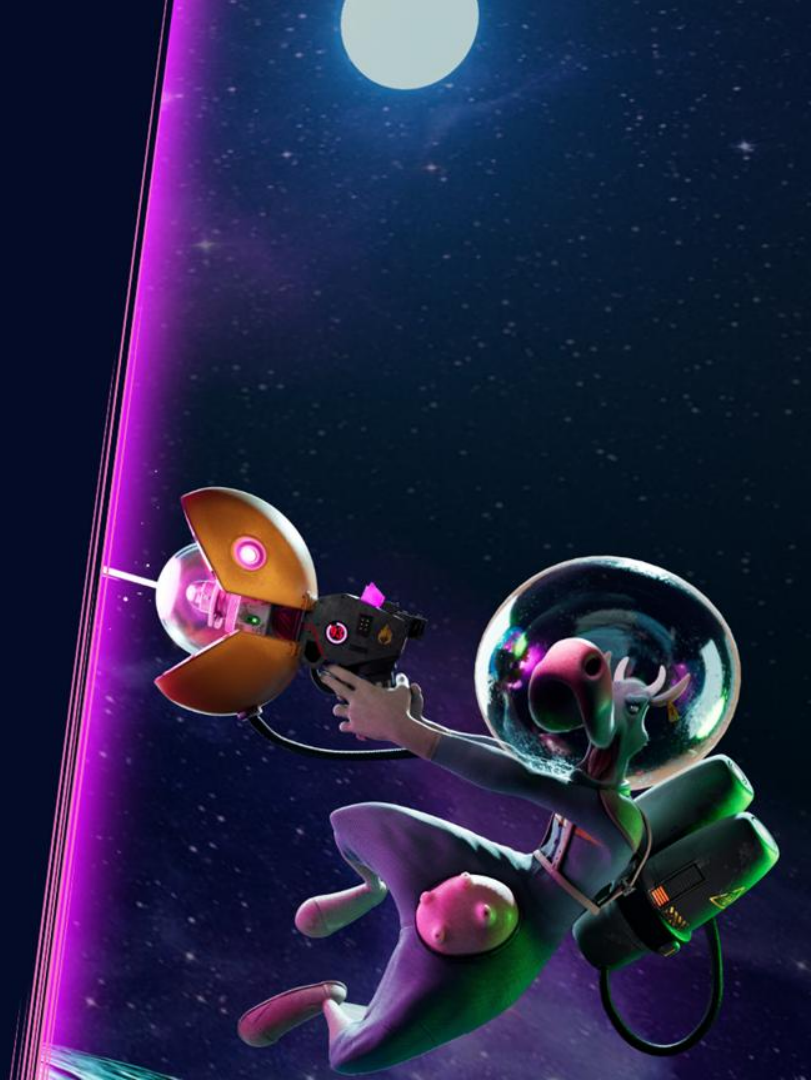
Chapter I
Attributes, Enums, Heritage, ScriptableObjects

Chapter II
Delegates

Chapter III
Coroutines, Gizmos, Raycast, Interface Segregation

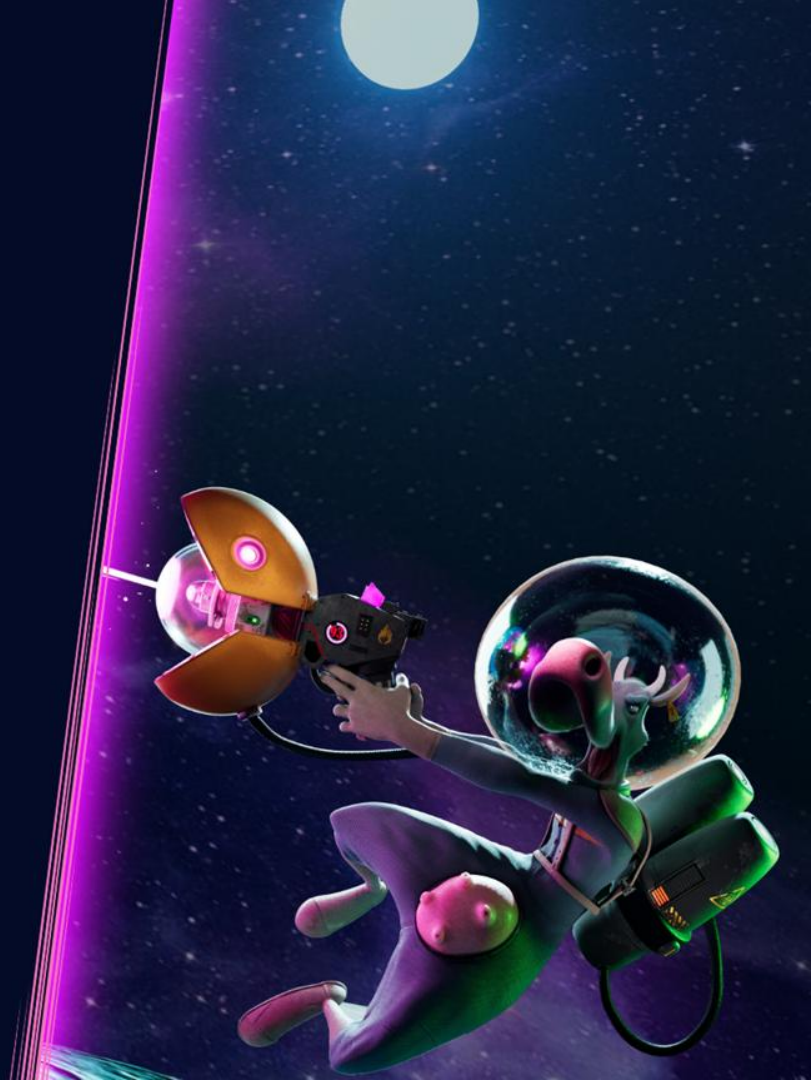
Chapter IV
Delegates (again), Saving data, Singleton, Observer Pattern

+ S.O.L.I.D all along



Let’s start !

First we are going to **install** the project !

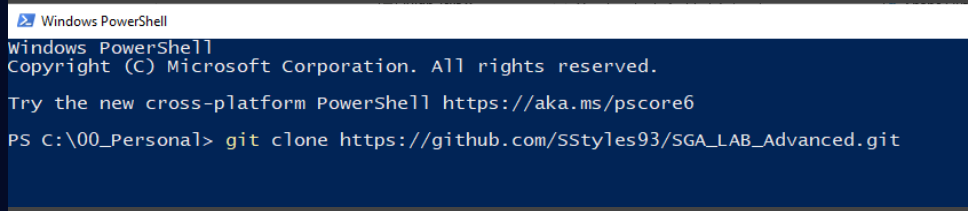


Project Instalation

Use:

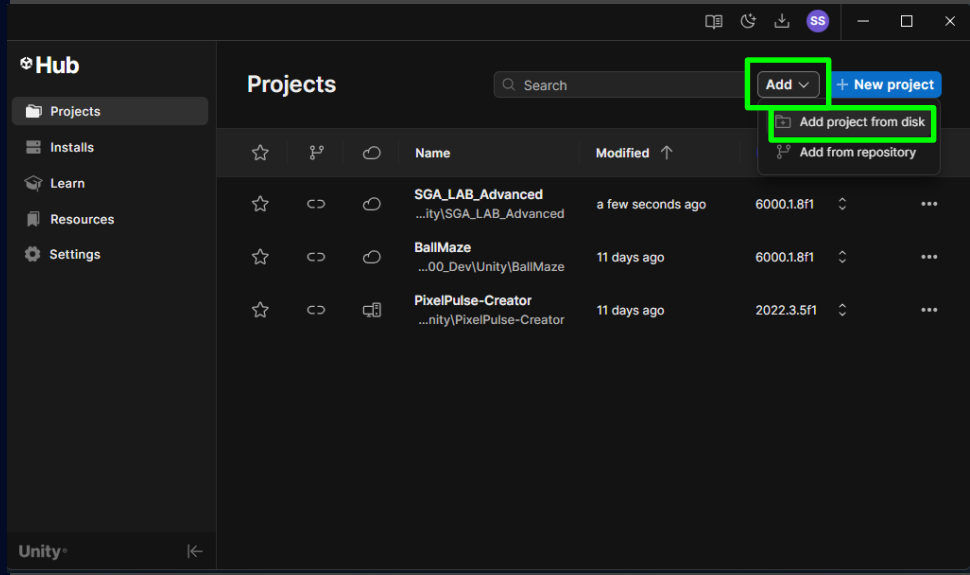
git clone
https://github.com/SStyles93/SGA_LAB_Advanced.git

In a terminal at the desired location.



Project Instalation

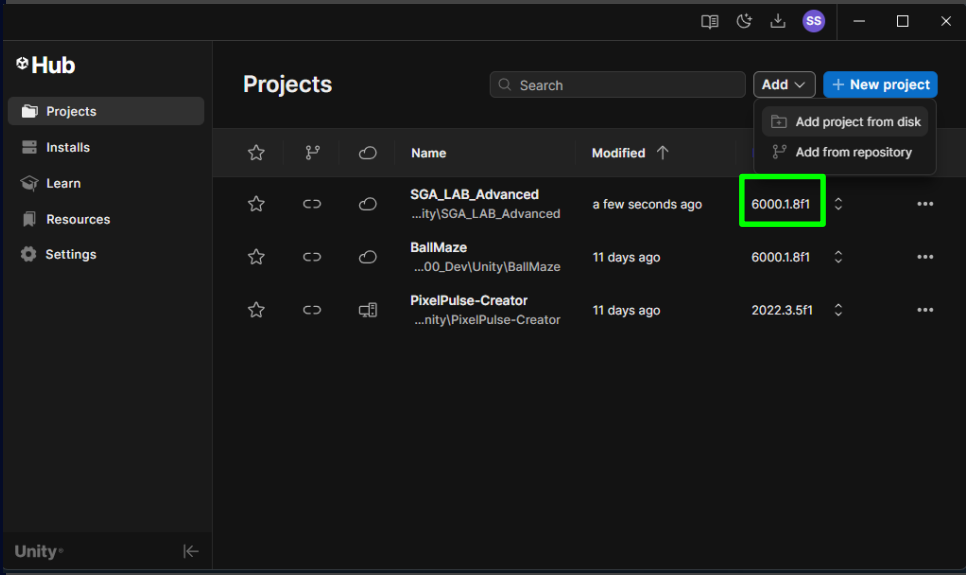
Add the project from disk with the clone’s path



Project Instalation

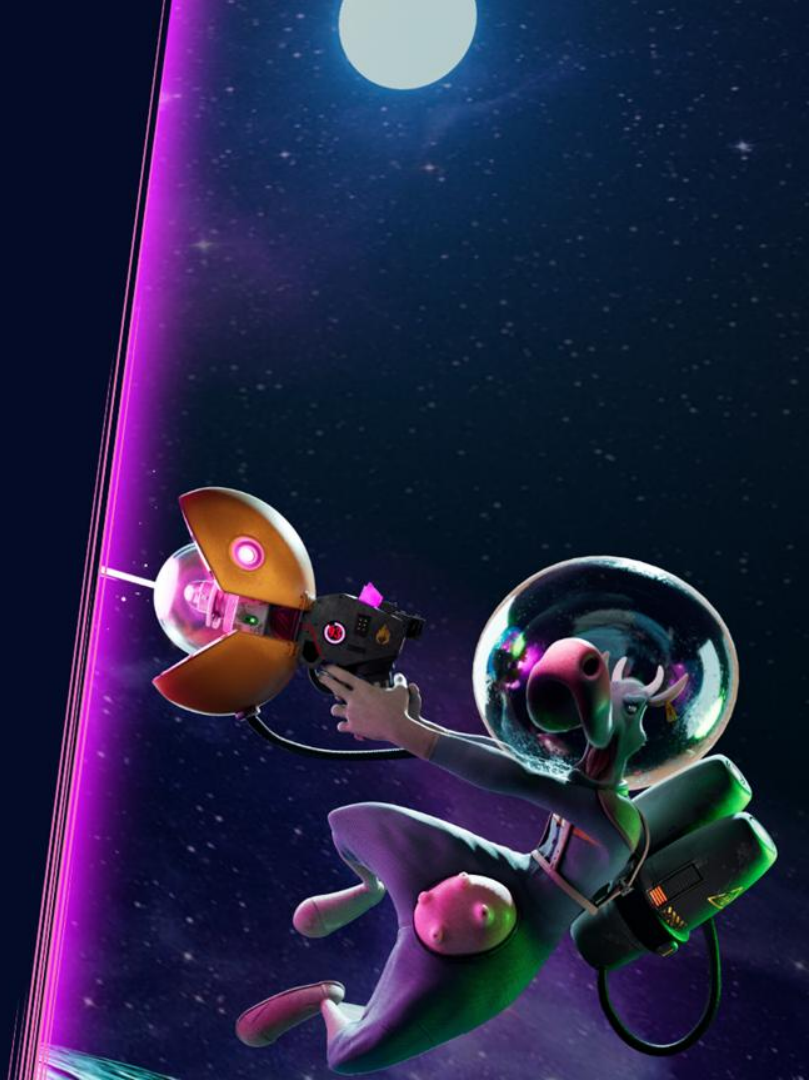
Open the project with the Unity Hub

Install the Unity version **6000.1.8f1**
If necessary



Chapter I

Attributes, ScriptableObjects, Enums and Heritage



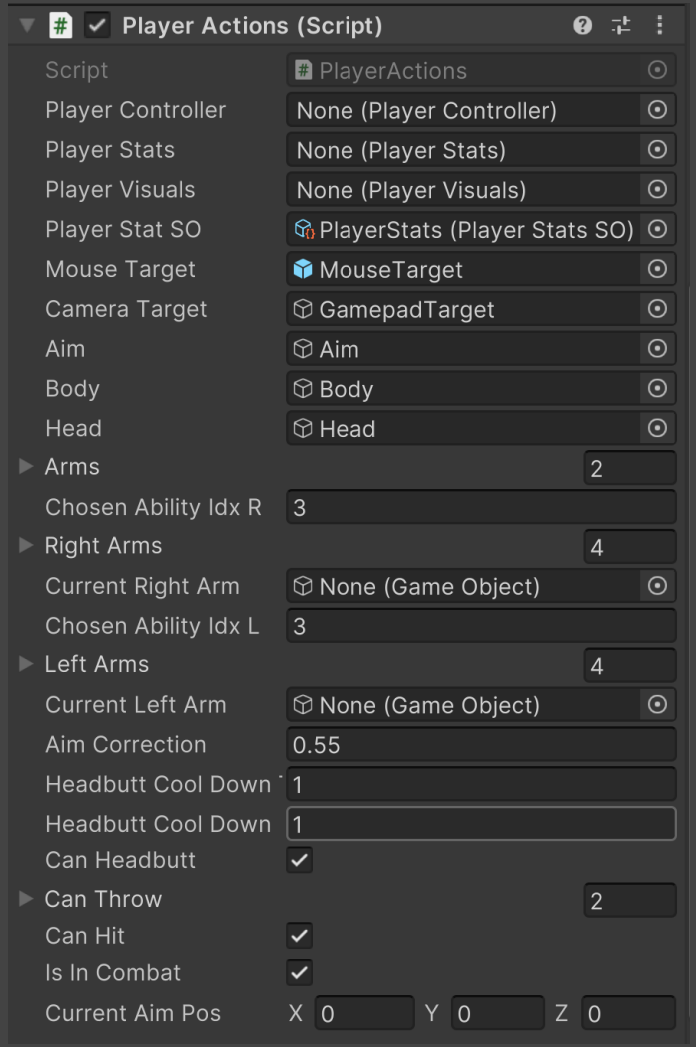
Chapter I - Unity attributes

If your code looks like this…the course is for you !

```
Unity Script (2 asset references) | 18 references
public class PlayerActions : MonoBehaviour
{
    public PlayerController _playerController;
    public PlayerStats _playerStats;
    public PlayerVisuals _playerVisuals;
    public PlayerStatsSO _playerStatSO;
    public GameObject mouseTarget;
    public GameObject _cameraTarget;
    public GameObject _aim;
    public GameObject _body;
    public GameObject _head;
    public List<GameObject> _arms;
    public int _chosenAbilityIdxR = 0;
    public List<GameObject> _rightArms;
    public GameObject currentRightArm;
    public int _chosenAbilityIdxL = 0;
    public List<GameObject> _leftArms;
    public GameObject currentLeftArm;
    public float _aimCorrection;
    public float _headbuttCoolDownTime = 1.0f;
    public float _headbuttCoolDown = 1.0f;
    public bool _canHeadbutt = true;
    public bool[] _canThrow = new bool[2] { true, true };
    public bool _canHit = true;
    public bool _isInCombat = true;
    public Vector3 currentAimPos;
```

Chapter I - Unity attributes

This is what it looks like in the editor :(



Chapter I - Unity attributes

Things you can use to make things **more user friendly** **AND** **look better**:

[teebarjunk/Unity-Built-In-Attributes: A list of built in Unity Attributes. (github.com)]

For nice Unity attributes

Chapter I - Unity attributes

Most usefull in my experience:

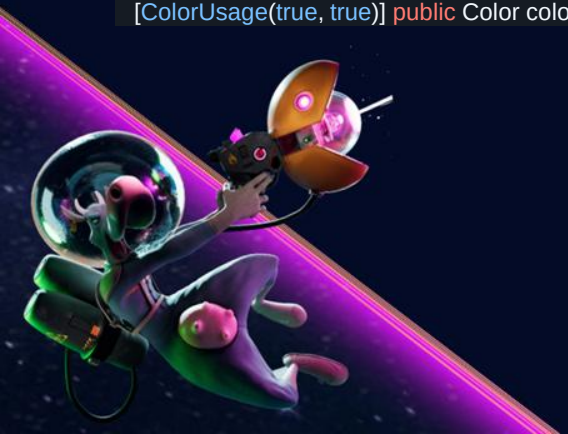
```
[Header("Stats")]
[Space(10)]
[Tooltip("The games score.")] public int score = 0;
```

For code

```
[SerializeField] private int score;
[System.Serializable]
[Range(0, 100)] public float speed = 2f;
[RequireComponent(typeof(RigidBody))]
```

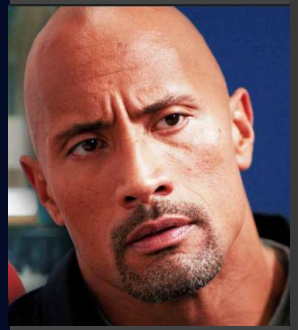
For Visuals :

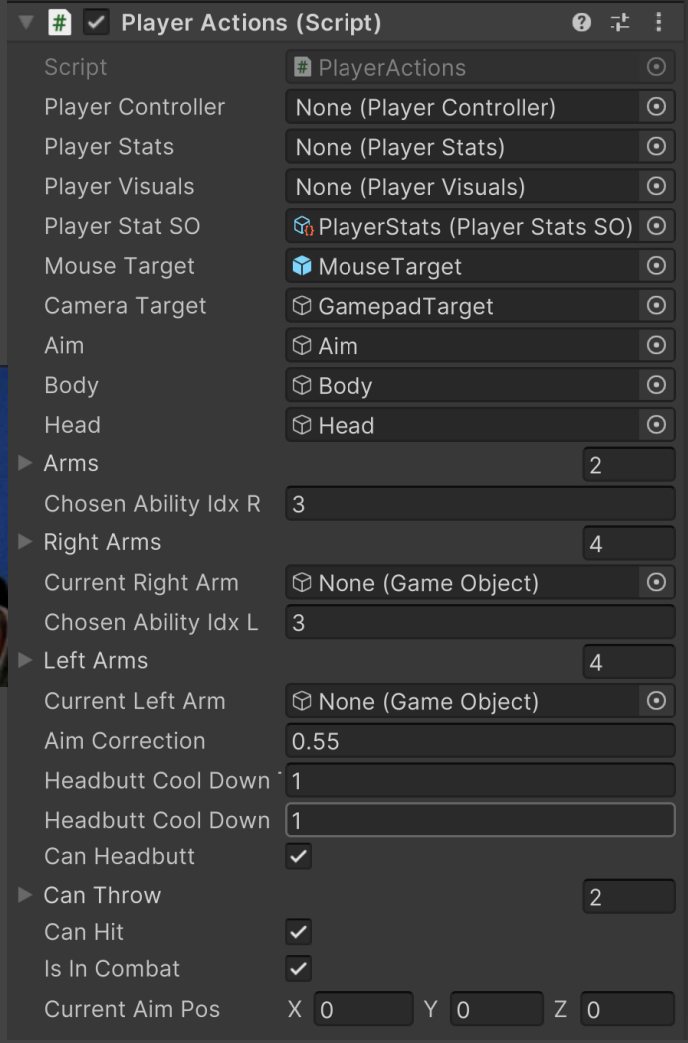
```
[GradientUsage(true)] public Gradient gradient;
[ColorUsage(true, true)] public Color color = Color.white;
```



Chapter I - SOLID

SOLID !





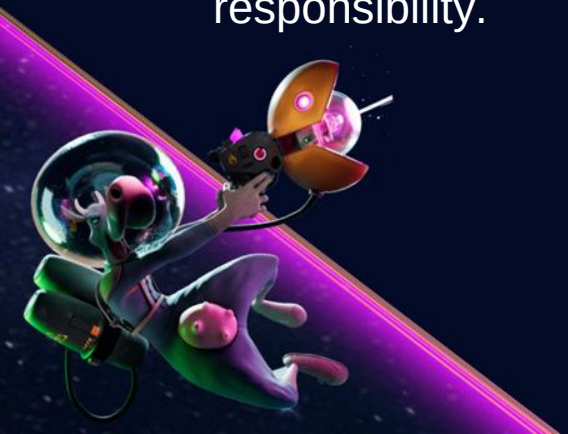
Chapter I – S.O.L.I.D

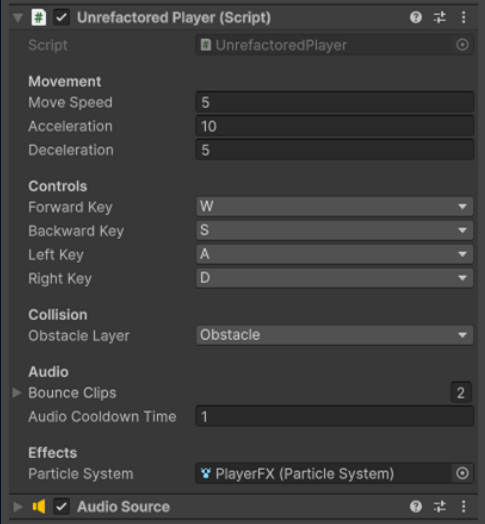
S = Single Responsibility Principle

“There should never be more than one reason for a class to change."

In other words:

Every class should have only one responsibility.





Chapter I – S.O.L.I.D

Lets PRACTICE !

Chapter I - Unity attributes

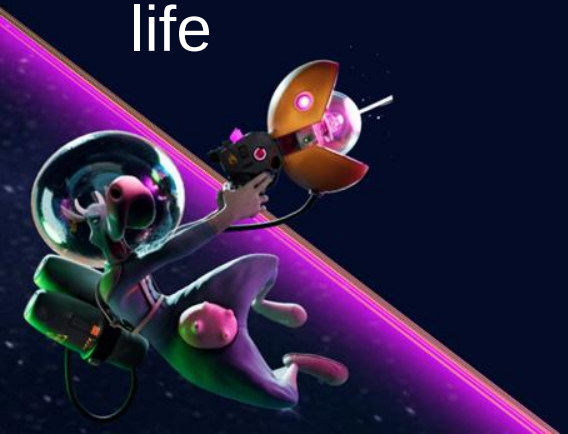
Say Hi to

AlchemisTeddy

Chapter I - Unity attributes

AlchemisTeddy did a mistake and Poisonned himself..

Code to help him regain his life

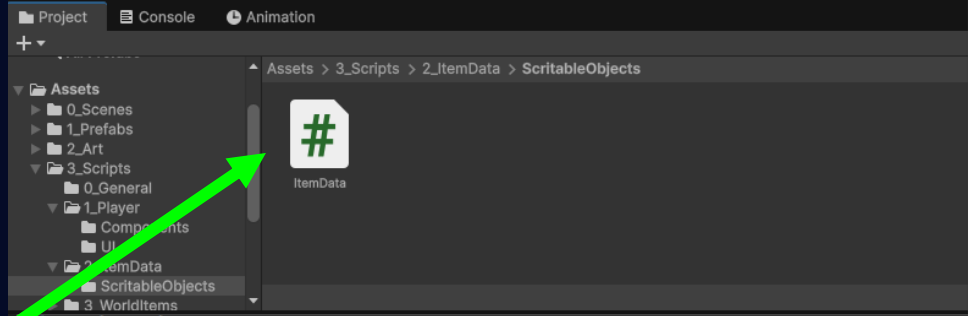


Chapter I - Unity attributes

PRACTICE Attributes & ScriptableObjects!

Go to the folder:
Assets/3_Scripts/2_ItemData/ScritableObjects

And open the « ItemData » script

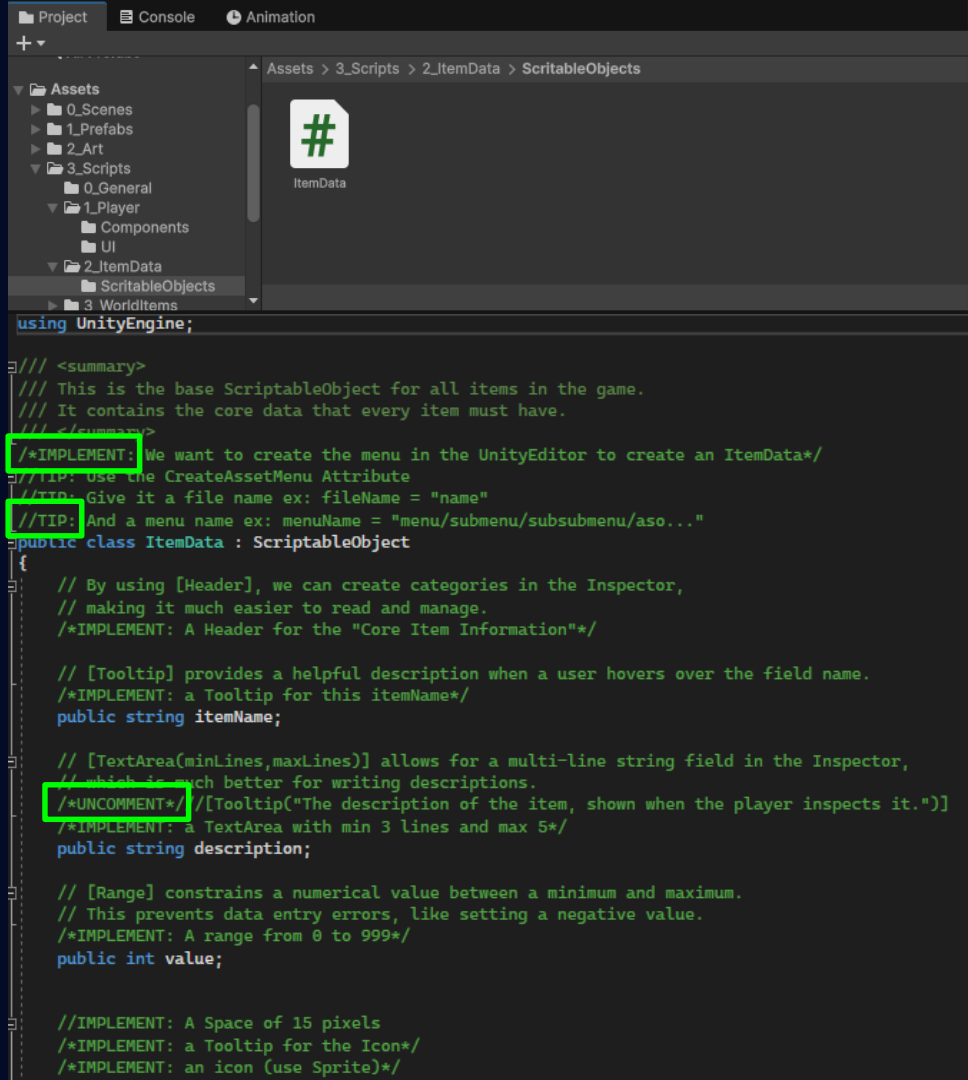



```
using UnityEngine;

/// <summary>
/// This is the base ScriptableObject for all items in the game.
/// It contains the core data that every item must have.
/// </summary>
/*IMPLEMENT: We want to create the menu in the UnityEditor to create an ItemData*/
//TIP: Use the CreateAssetMenu Attribute
//TIP: Give it a file name ex: fileName = "name"
//TIP: And a menu name ex: menuName = "menu/submenu/subsubmenu/aso..."
public class ItemData : ScriptableObject
{
    // By using [Header], we can create categories in the Inspector,
    // making it much easier to read and manage.
    /*IMPLEMENT: A Header for the "Core Item Information"*/

    // [Tooltip] provides a helpful description when a user hovers over the field name.
    /*IMPLEMENT: a Tooltip for this itemName*/
    public string itemName;

    // [TextArea(minLines,maxLines)] allows for a multi-line string field in the Inspector,
    // which is much better for writing descriptions.
    /*UNCOMMENT*///[Tooltip("The description of the item, shown when the player inspects it.")]
    /*IMPLEMENT: a TextArea with min 3 lines and max 5*/
    public string description;

    // [Range] constrains a numerical value between a minimum and maximum.
    // This prevents data entry errors, like setting a negative value.
    /*IMPLEMENT: A range from 0 to 999*/
    public int value;


    //IMPLEMENT: A Space of 15 pixels
    /*IMPLEMENT: a Tooltip for the Icon*/
    /*IMPLEMENT: an icon (use Sprite)*/
```

Chapter I - Unity attributes

We are going to implement the code we need.

/*IMPLEMENT: …*/
- > has to be REPLACED by your code.
//TIP
-> can be deleted but it is mainly for you.
/*UNCOMMENT*/
-> is to be uncommented.
/*REFACTOR*/
-> has to be improved

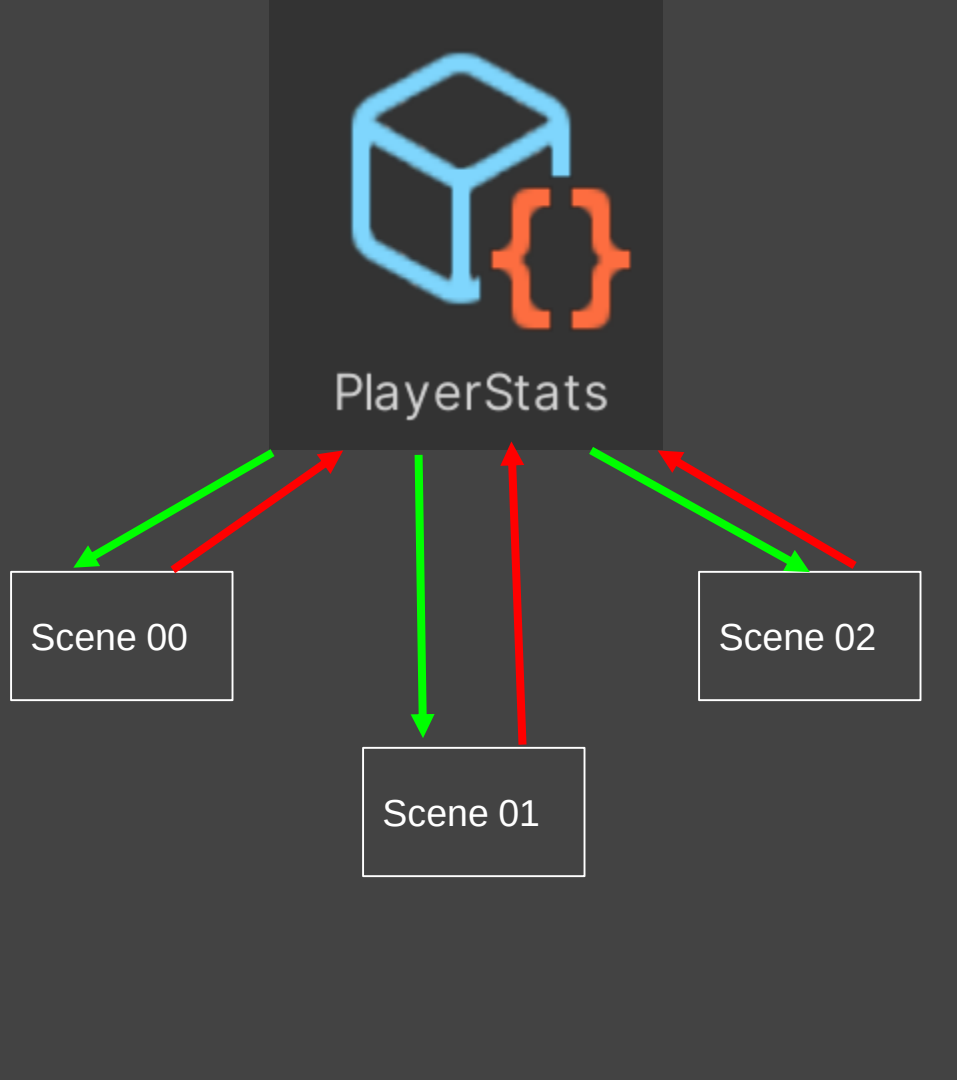


Chapter I – Scriptable Objs.

What is a scriptable object ?

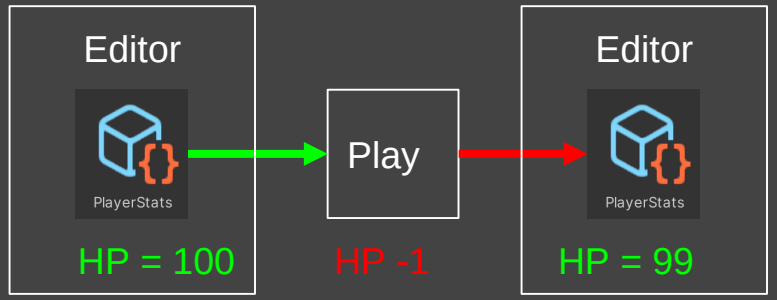
Chapter I – Scriptable Objs.

- It is a data container that can be used to READ and WRITE between scenes



Chapter I – Scriptable Objs.

- It is a data container that can be used to READ and WRITE between scenes
- Data stays modified even when exiting play mode ! (NO RESET!)



SOLID

Coding best practices

L = Liskov substitution Principle

“Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it."

In other words:
Super class objects should be substitutable by Sub class objects

```
/// <summary>
/// This class shows a violation of Liskov Substitution. The subclass adds a time-based duration, not
/// present in the base class. Though the logic is functional, "duration" is not a concept from the base class.
/// Thus, the UnrefactoredSpeedBoost cannot be substituted for other PowerUps that do not support duration.
/// </summary>
Unity Script | 0 references
public class UnrefactoredSpeedBoost : UnrefactoredPowerUp
{
    public float m_SpeedMultiplier = 2f;
    public float m_Duration = 5f; // Duration not supported by the base class

    2 references
    public override void ApplyEffect(GameObject player)
    {
        if (m_Duration > 0)
        {
            SpeedModifier playerMovement = player.GetComponent<SpeedModifier>();
            if (playerMovement != null)
            {
                playerMovement.ModifySpeed(m_SpeedMultiplier, m_Duration);
            }
        }
        else
        {
            // This branch or logic might be confusing for someone who only expects to "ApplyEffect"
            // without a duration.  Not every PowerUp is interchangeable if we use this logic.
        }
    }
}
```

```
/// <summary>
/// Each PowerUp subclass can have its own unique behavior.
/// </summary>
Unity Script (1 asset reference) | 0 references
public class SpeedBoost : PowerUp
{
    [Header("Speed parameters")]
    [Tooltip("Factor used to multiply speed")] [SerializeField]
    float m_SpeedMultiplier = 2f;

    // Override this method in the subclass
    2 references
    public override void ApplyEffect(GameObject player)
    {
        // Add SpeedBoost logic here
        SpeedModifier speedModifier = player.GetComponent<SpeedModifier>();

        if (speedModifier != null)
        {
            speedModifier.ModifySpeed(m_SpeedMultiplier, m_Duration);
        }
    }
}
```

Chapter I – Unity scripts

Now our variables **look good** and we coded our first Scriptable Object but what about **the code** ?

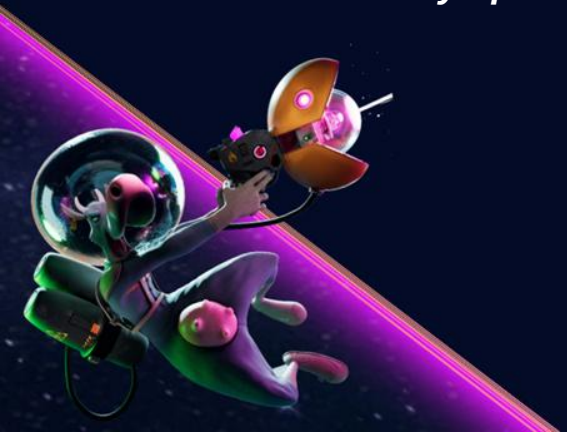
Chapter I – Unity scripts

Open PlayerInventoryManager

Go to UpdateStatus()

We are going to Use

Switches
and
Ternary operators



Chapter I – Unity scripts (TIP)

In C# we can use some **useful features**:

Instead of this:

Multiple variants of method

You can do this:

Optional parameters

public void Func(x,y, z = default Value)

```
4 references
public void EnablePlayersArm(BODYPART armSide, bool enable)
{
    _arms[(int)armSide].SetActive(enable);
    _canThrow[(int)armSide] = true;

}
0 references
public void EnablePlayersArmTwo(BODYPART armSide, bool enable, bool secondaryCondition)
{
    if (secondaryCondition == true)
    {
        _arms[(int)armSide].SetActive(enable);
        _canThrow[(int)armSide] = true;

    }
}
```

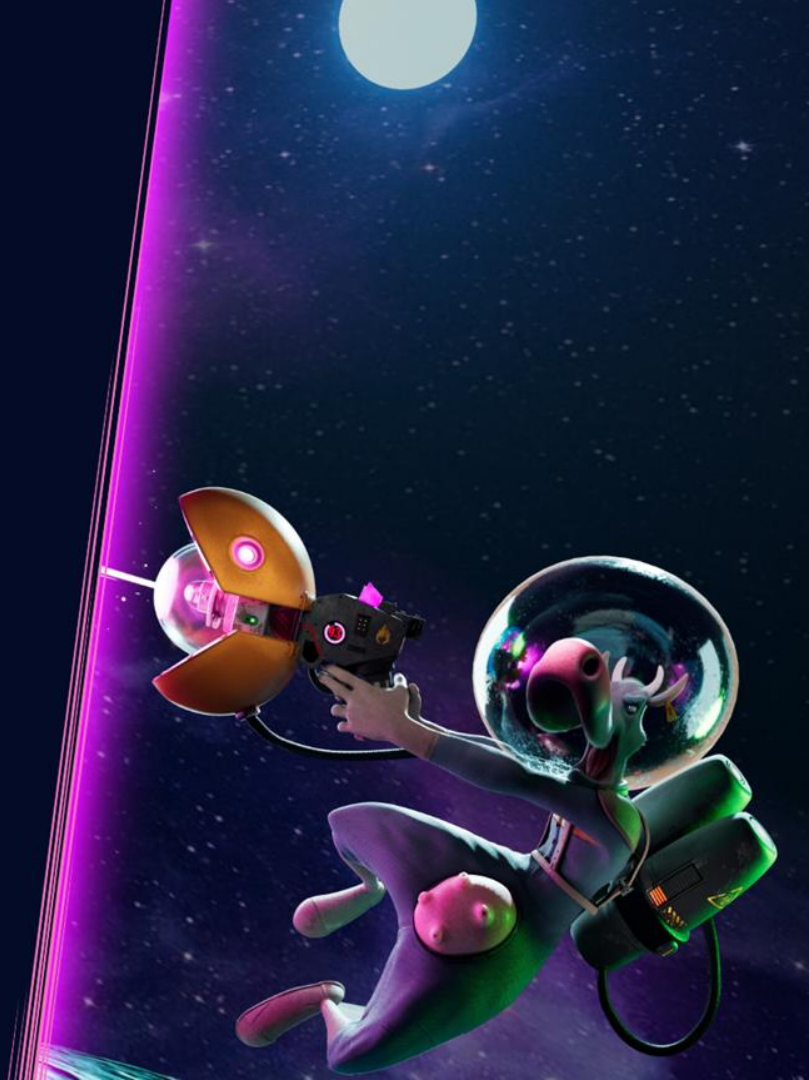
```
4 references
public void EnablePlayersArm(BODYPART armSide, bool enable, bool secondaryCondition = true)
{
    if (secondaryCondition)
    {
        _arms[(int)armSide].SetActive(enable);
        _canThrow[(int)armSide] = true;
    }
}
```

Tip: Undefined number of parameters

Params collections - C# feature specifications | Microsoft Learn

Chapter II

Delegates
(and Actions)



Chapter II - Delegates

What is a delegates !

```
2 references
public abstract class Enemy
{
    public int Health;
    public int AttackPower;
    2 references
    public abstract void Init();

    public delegate void PauseDelegate();
```

Chapter II - Delegates

It is a method with a SIGNATURE:
ReturnType Name (Parameters)
Here
void PauseDelegate(No parameters);

```
2 references
public abstract class Enemy
{
    public int Health;
    public int AttackPower;
    2 references
    public abstract void Init();

    public delegate void PauseDelegate();
```

Chapter II - Delegates

We then declare a delegate "PauseDelegate" called PauseMethod;

And with that we can subscribe methods with **THE SAME SIGNATURE** to the delegate method using:

DelegateMethod += Method;

```
2 references
public abstract class Enemy
{
    public int Health;
    public int AttackPower;
    2 references
    public abstract void Init();

    public delegate void PauseDelegate();
    public PauseDelegate PauseMethod;
}
0 references
public class SpecialEnemy : Enemy
{
    1 reference
    public void Pause()
    {
        //Pause specific elements
    }
    1 reference
    public override void Init()
    {
        PauseMethod += Pause;
    }
```

Chapter II - Delegates

This means that when the PauseMethod delegate is called
EVERY SUBSCRIBED method will be also called ! :D

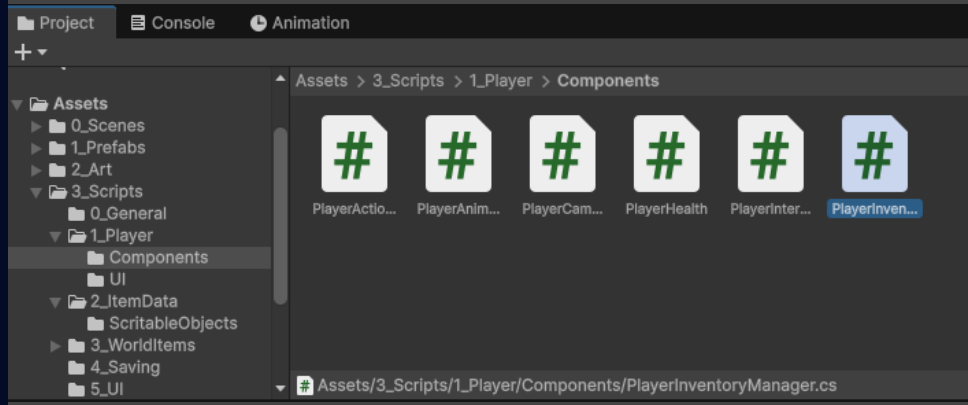
```
2 references
public abstract class Enemy
{
    public int Health;
    public int AttackPower;
    2 references
    public abstract void Init();

    public delegate void PauseDelegate();
    public PauseDelegate PauseMethod;
}
0 references
public class SpecialEnemy : Enemy
{
    1 reference
    public void Pause()
    {
        //Pause specific elements
    }
    1 reference
    public override void Init()
    {
        PauseMethod += Pause;
    }
```

Chapter II - Delegates

PRACTICE Attributes & ScriptableObjects!

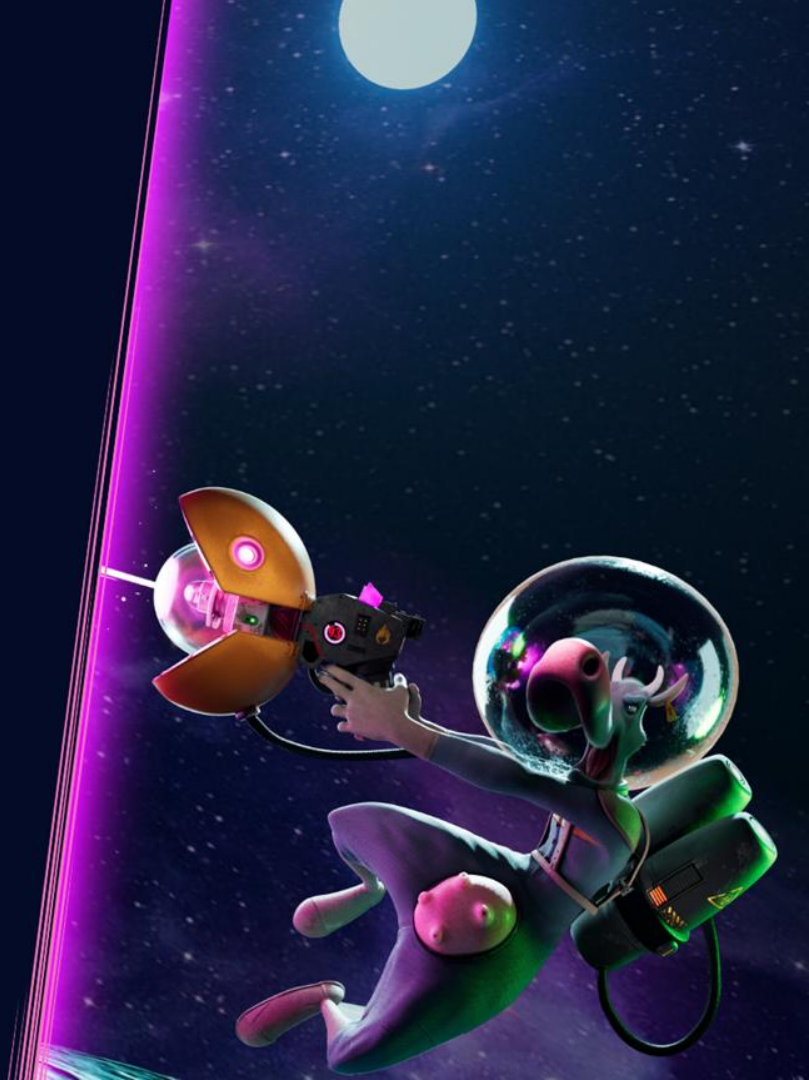
Go to the folder:
Assets/3_Scripts/1_Player/Components/

And open the «PlayerInventoryManager»



Chapter III

Coroutines, Gizmos, Raycast, Interfaces



Chapter III - Coroutines

A coroutine is a method that can suspend execution and resume at a later time.

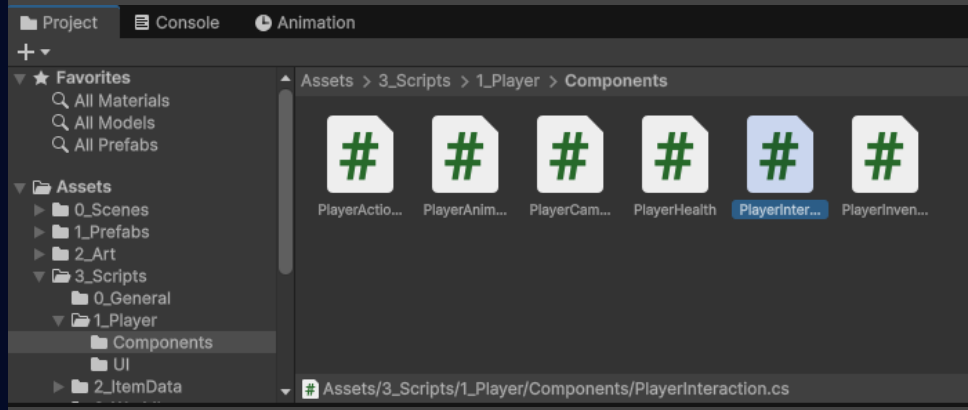
```
IEnumerator MethodName()
{
                    Yield return … (to wait)
}
StartCoroutine(MethodName());
StopCoroutine(MethodName());
```

Chapter III - Coroutines

Time to PRACTICE !

Open

PlayerInteraction.cs



Chapter III - Raycast

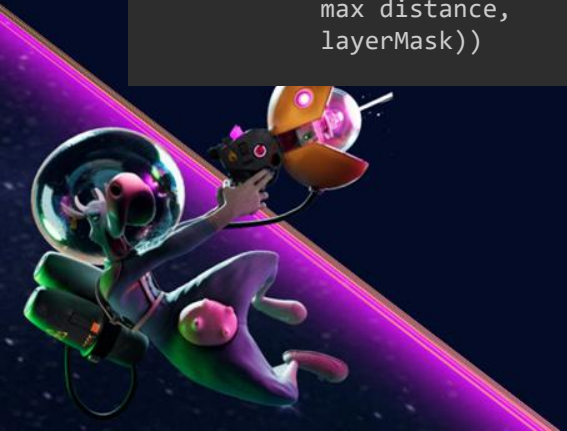
Implement it !

Use:

```
RaycastHit hit;
```

and

```
if (Physics.Raycast(
            from,
            direction,
            out hit,
            max distance,
            layerMask))
```



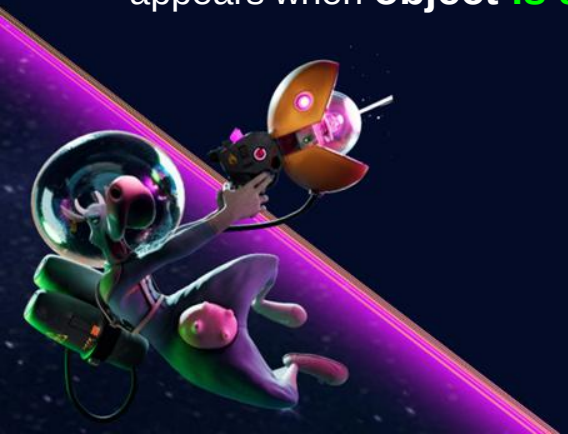
Chapter III - Gizmos

Here are some useful tips to use in unity:

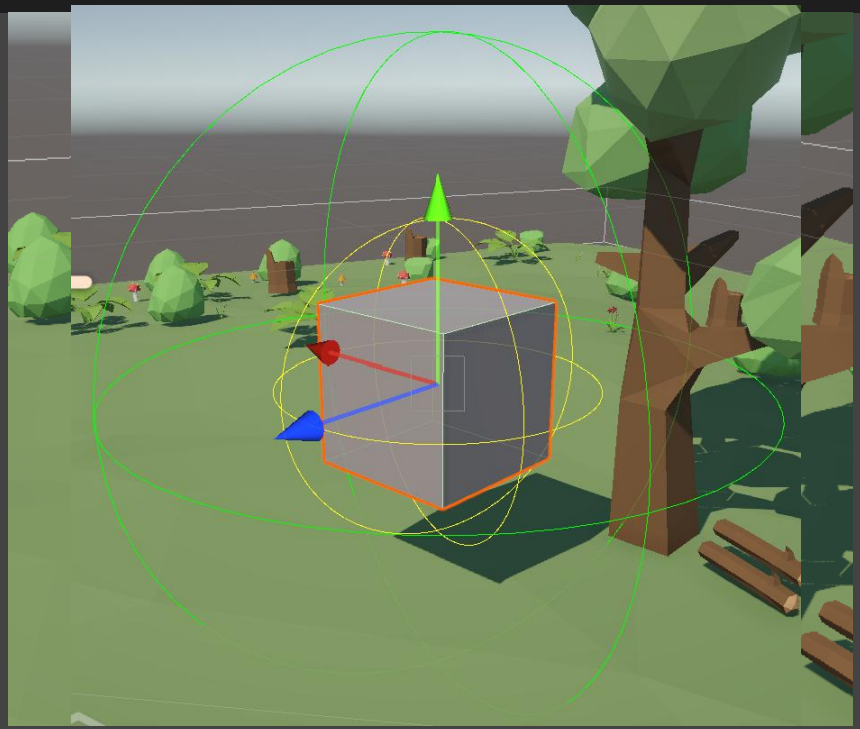
Gizmos with:

- OnDrawGizmos()
  and
- OnDrawGizmosSelected()

As the name says the Selected version only appears when **object is selected**.



```
0 references
void OnDrawGizmosSelected()
{
    // Display the explosion radius when selected
    Gizmos.color = Color.green;
    Gizmos.DrawWireSphere(transform.position, radius: 2.0f);
}
```



Chapter III - Gizmos

Here are some useful tips to use in unity:

Gizmos allow you to draw:
multiple shapes
and colours

See the documentation:
[Unity - Scripting API: Gizmos (unity3d.com)]

DrawCube	Draw a solid box at center with size.
DrawFrustum	Draw a camera frustum using the currently set Gizmos.matrix for its location and rotation.
DrawGUITexture	Draw a texture in the Scene.
DrawIcon	Draw an icon at a position in the Scene view.
DrawLine	Draws a line starting at from towards to.
DrawLineList	Draws multiple lines between pairs of points.
DrawLineStrip	Draws a line between each point in the supplied span.
DrawMesh	Draws a mesh.
DrawRay	Draws a ray starting at from to from + direction.
DrawSphere	Draws a solid sphere with center and radius.
DrawWireCube	Draw a wireframe box with center and size.
DrawWireMesh	Draws a wireframe mesh.
DrawWireSphere	Draws a wireframe sphere with center and radius.

color	Sets the Color of the gizmos that are drawn next.
exposure	Set a texture that contains the exposure correction for LightProbe gizmos. The value is sampled from the red channel in the middle of the texture.
matrix	Sets the Matrix4x4 that the Unity Editor uses to draw Gizmos.
probeSize	Set a scale for Light Probe gizmos. This scale will be used to render the spherical harmonic preview spheres.

SOLID

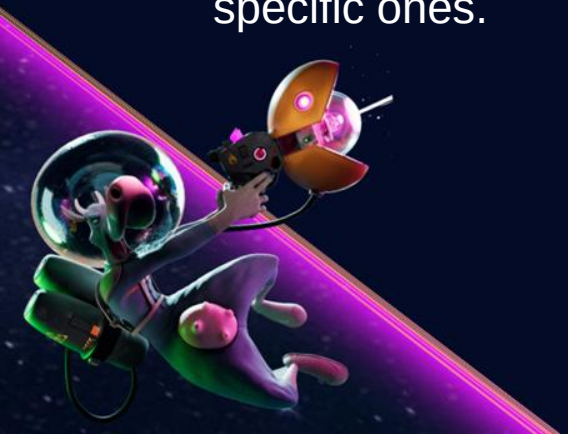
Coding best practices

I = Interface segregation Principle

“No client should be forced to depend on methods it does not use."

In other words:

Split large interfaces in smaller, specific ones.



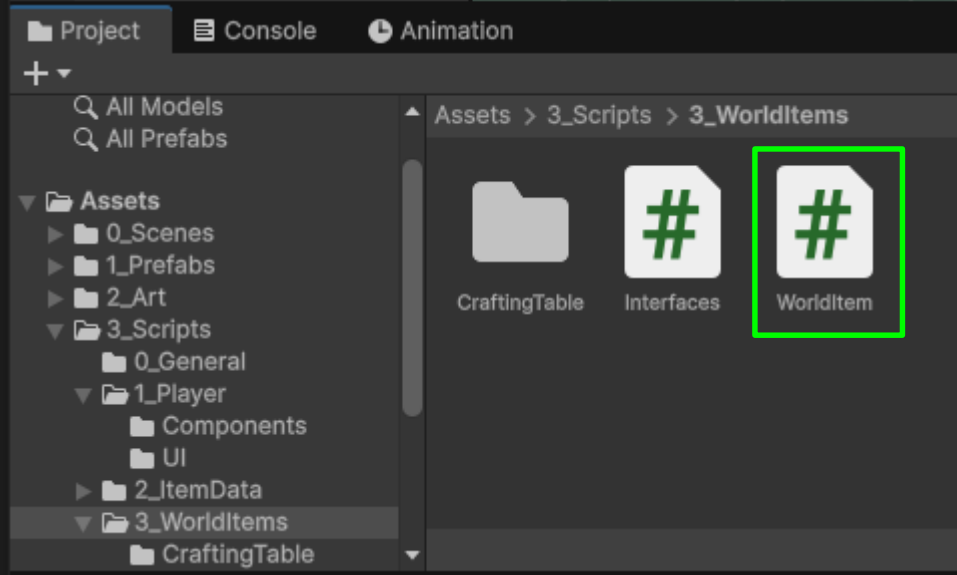
```
1 reference
public interface IExplodable
{
    // Triggers an explosion (e.g. particles or other GameObject effects)
    2 references
    void Explode();
}

/// <summary>
/// Defines a contract for triggering effects, such as particle systems or sound effects, at a specific
/// </summary>
2 references
public interface IEffectTrigger
{
    2 references
    void TriggerEffect(Vector3 position);
}

2 references
public interface IDamageable
{
    2 references
    void TakeDamage(float amount);
                                    need to explode, this method must be implemented.
}

/// <summary>
/// Alternative type of target that can explode and instantiate an effect on death. Here we
/// inherit from the base Target and add the IExplodable interface
/// </summary>
Unity Script (1 asset reference) | 0 references
public class ExplodableTarget : Target, IExplodable
{
    [Tooltip("Effect to instantiate on explosion")]
    [SerializeField] GameObject m_ExplosionPrefab;

    3 references
    protected override void Die()
    {
        base.Die();
        Explode();
    }

    2 references
    public void Explode()
    {
        if (m_ExplosionPrefab)
        {
            GameObject instance = Instantiate(m_ExplosionPrefab, transform.position, quaternion.identity);
        }

        // Add custom explosion logic here
    }
}
```

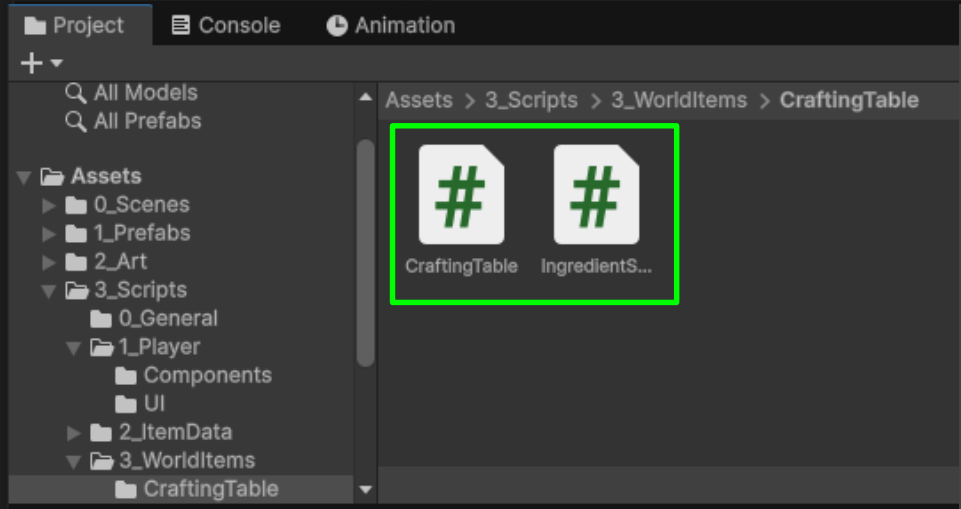
Chapter III - Interfaces

Open:

Assets/3_Scripts/3_WorldItems/World Item

Assets/3_Scripts/3_WorldItems/
CraftingTable & IngredientStation





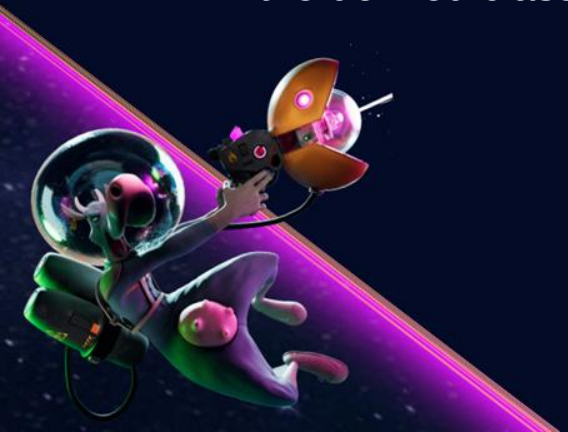
SOLID

Coding best practices

O = Open-Closed Principle

“Software entities should be open for extension but closed for modification."

In other words:
You should create an Interface for general purpose and implement logic in the derived classes.



```
0 references
public class UnrefactoredAreaCalculator
{
    // Non-SOLID implementation: not using Open-Closed principle. Though
    // this approach works well with a small number of effects, it does
    // not scale and becomes unwieldy as the project grows.

    0 references
    public float GetRectangleArea(Rectangle rectangle)
    {
        return rectangle.Width * rectangle.Height;
    }

    0 references
    public float GetCircleArea(Circle circle)
    {
        return circle.Radius * circle.Radius * Mathf.PI;
    }

    // Adds additional methods with additional shapes
    // e.g GetPentagonArea, GetHexagonArea, etc.
}

1 reference
public class Rectangle
{
    public float Height;
    public float Width;

}

1 reference
public class Circle
{
    public float Radius;
}
```

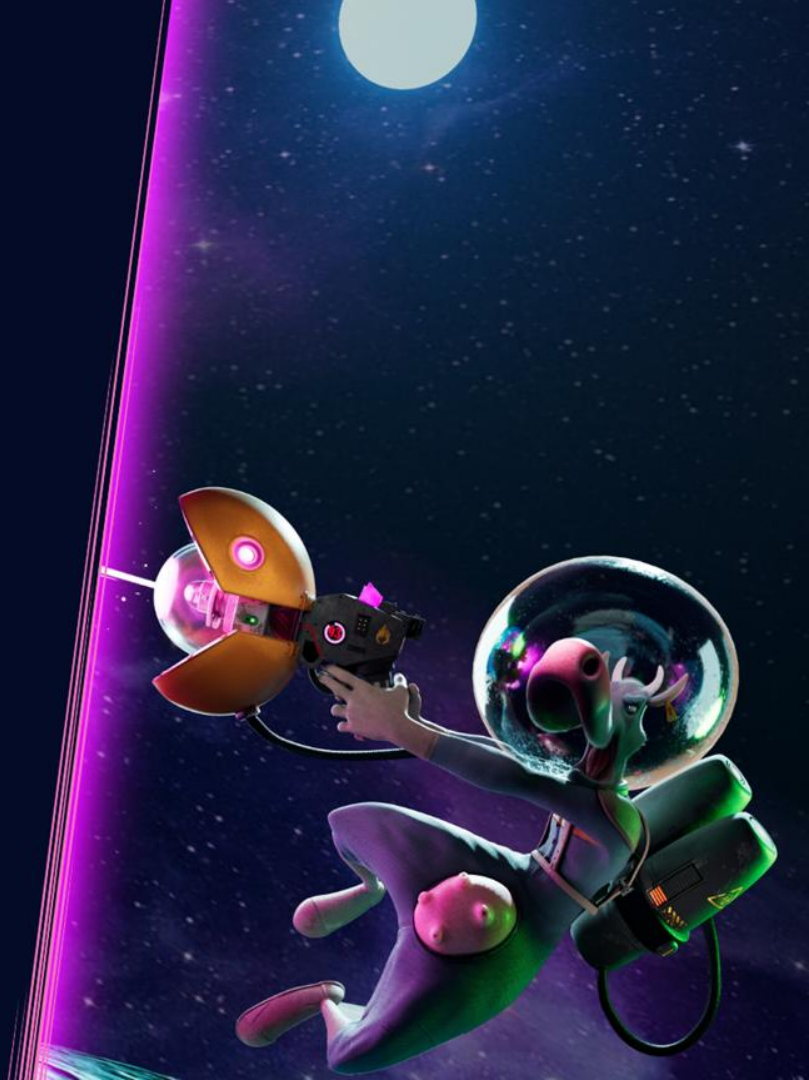
```
/// <summary>
/// Each AreaOfEffect subclass implements its own definition of CalculateArea
/// </summary>
/// <returns></returns>
6 references
public abstract float CalculateArea();
```

```
public class CircleEffect : AreaOfEffect
{
    [Header("Shape")]
    [Tooltip("The radius of the circle")]
    [SerializeField] float m_Radius;

    2 references
    public float Radius { get => m_Radius; set => m_Radius = value; }

    3 references
    public override float CalculateArea()
    {
        return Radius * Radius * Mathf.PI;
    }
}
```

```
public class TriangularEffect : AreaOfEffect
{
    [Header("Shape")]
    [Tooltip("The side length of the triangle")]
    [SerializeField] private float m_SideLength;

    3 references
    public override float CalculateArea()
    {
        return (Mathf.Sqrt(3) / 4) * m_SideLength * m_SideLength;
    }
}
```

Chapter IV

Saving
Singleton & Observer Pattern



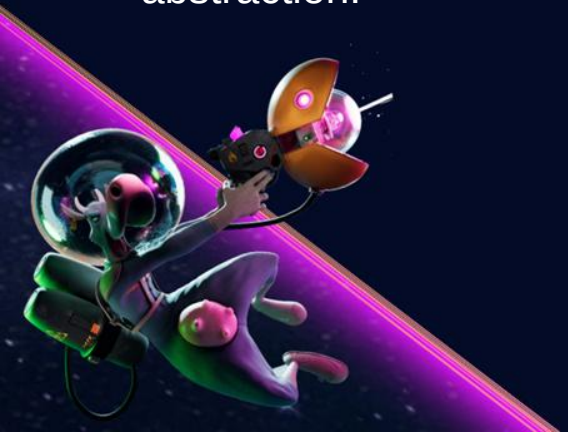
SOLID

Coding best practices

D = Dependency inversion Principle

“Depend upon abstractions, not concretes.”

In other words:
High level modules don’t depend on low level modules. Both depend on abstraction.

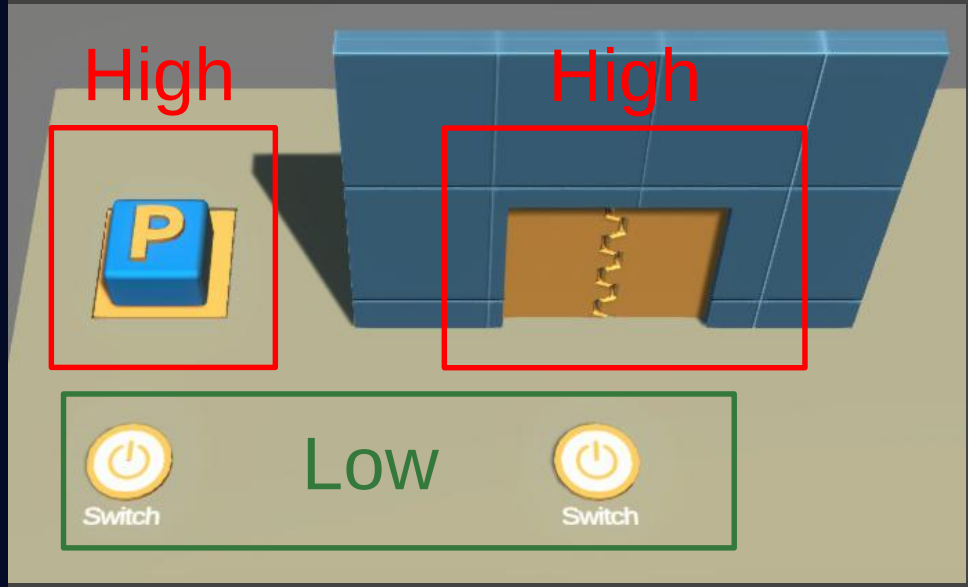


```
public class Switch : MonoBehaviour
{
    // Unity's serialization system does not directly support interfaces. Work around this limitation
    // by using a serialized reference to a MonoBehaviour that implements ISwitchable.

    [SerializeField] private MonoBehaviour m_ClientBehaviour;
    4 references
    private ISwitchable m_Client => m_ClientBehaviour as ISwitchable;


    // Toggles the active state of the associated ISwitchable client.
    0 references
    public void Toggle()
    {
        if (m_Client == null)
            return;

        if (m_Client.IsActive)
        {
            m_Client.Deactivate();
        }
        else
        {
            m_Client.Activate();
        }
    }
}

/// <summary>
/// The Trap class represents a physics-based trapdoor which implements ISwitchable.
/// </summary>
Unity Script (1 asset reference) | 0 references
public class Trap : MonoBehaviour, ISwitchable
{
    // Rigidbody component for physics interactions.
    private Rigidbody m_Rigidbody;

    // Original position of the trap, used for resetting its position.
    private Vector3 m_OriginalPosition;

    // Original rotation of the trap, used for resetting its rotation.
    private Quaternion m_OriginalRotation;

    // ISwitchable active state
    private bool m_IsActive;
    2 references
    public bool IsActive => m_IsActive;
```

SOLID

Coding best practices

Example : Trap,Door and Switch



SOLID

Coding best practices

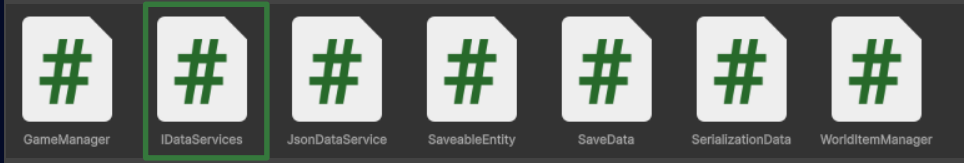
In our game we have:

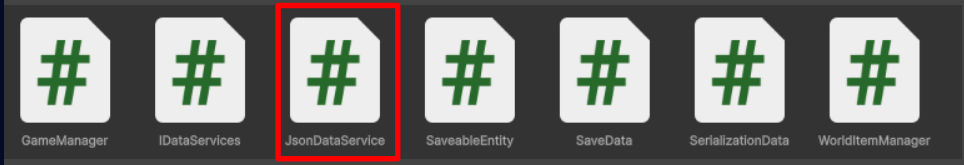
Assets/3_Scripts/4_Saving/IDataService

And

Assets/3_Scripts/4_Saving/JsonDataService

Low





High

Chapter IV – Observer Pattern

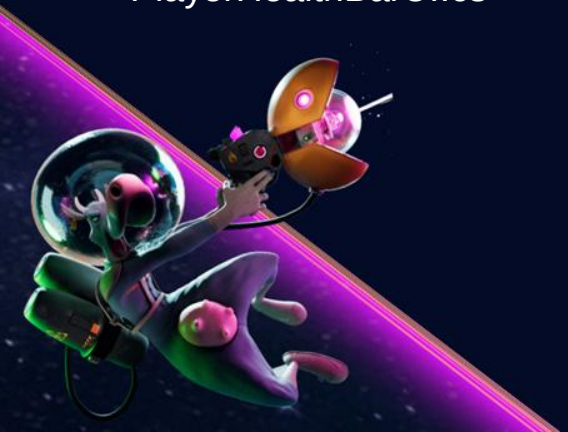
Time to PRACTICE !

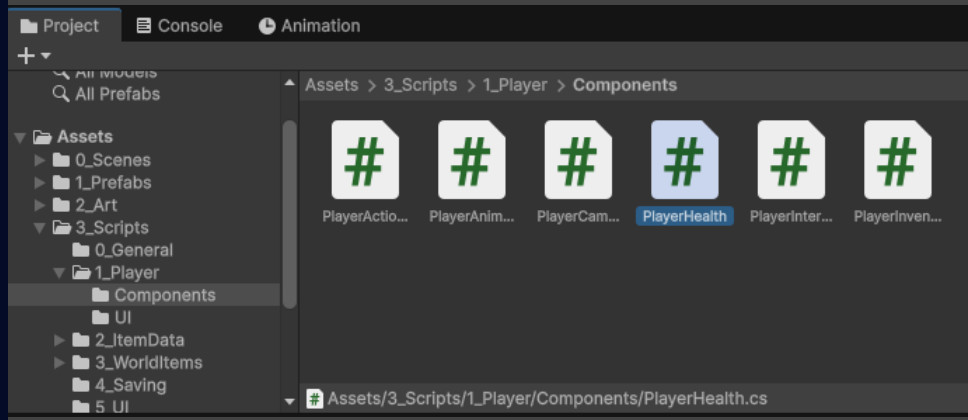
Open

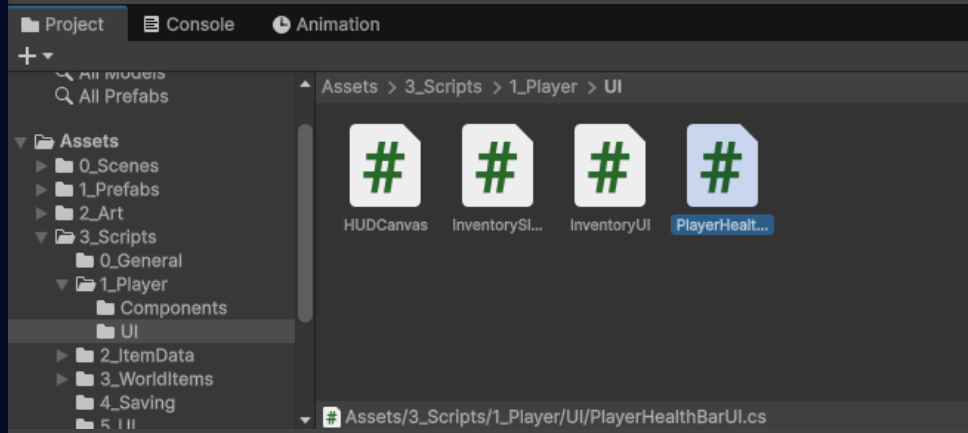
PlayerHealth.cs

and

PlayerHealthBarUI.cs







Chapter IV - Singleton

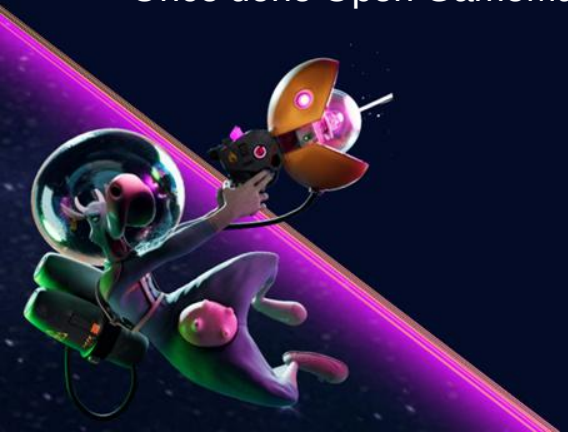
Open

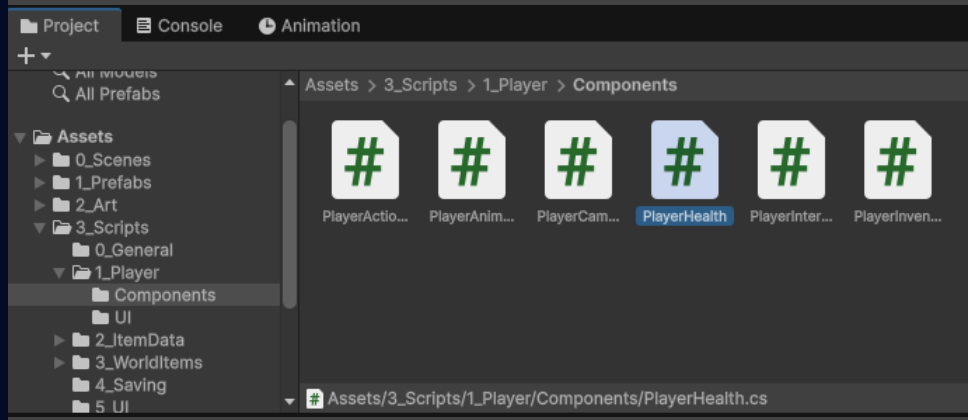
WorldItemManager

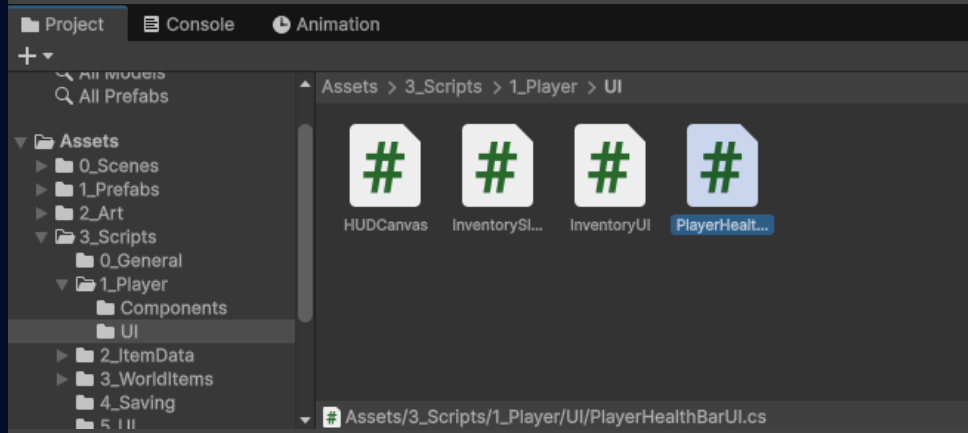
and

WorldItem

Once done Open GameManager

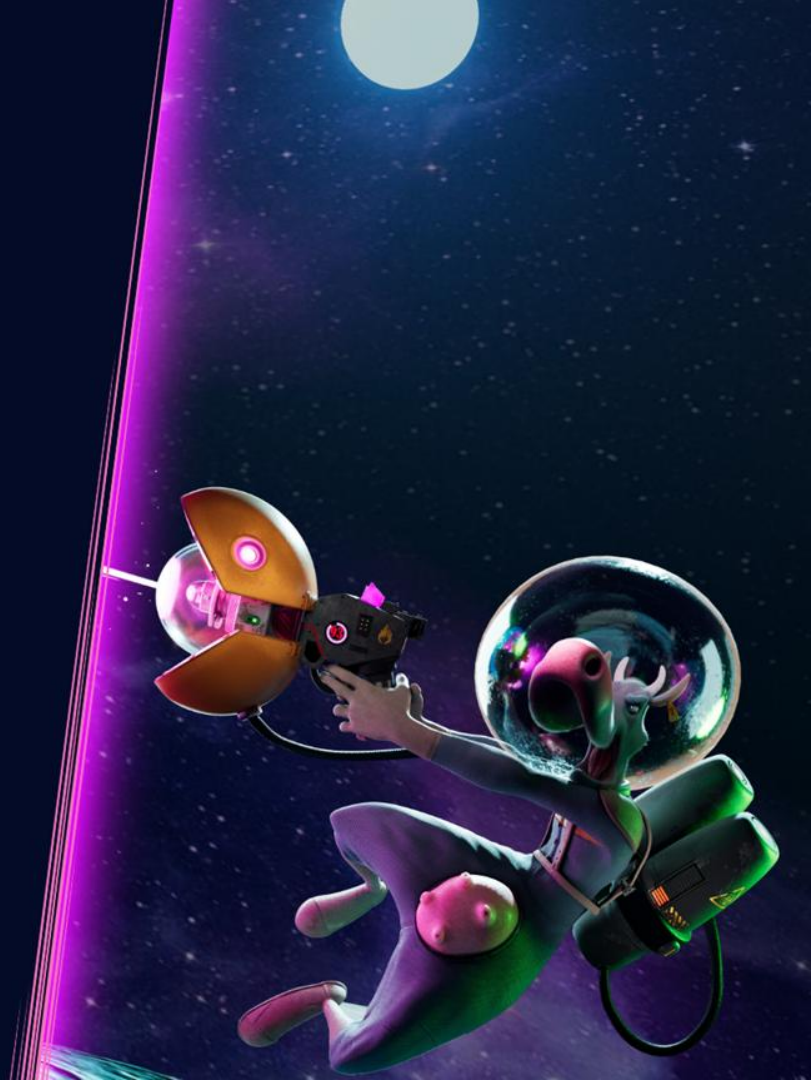






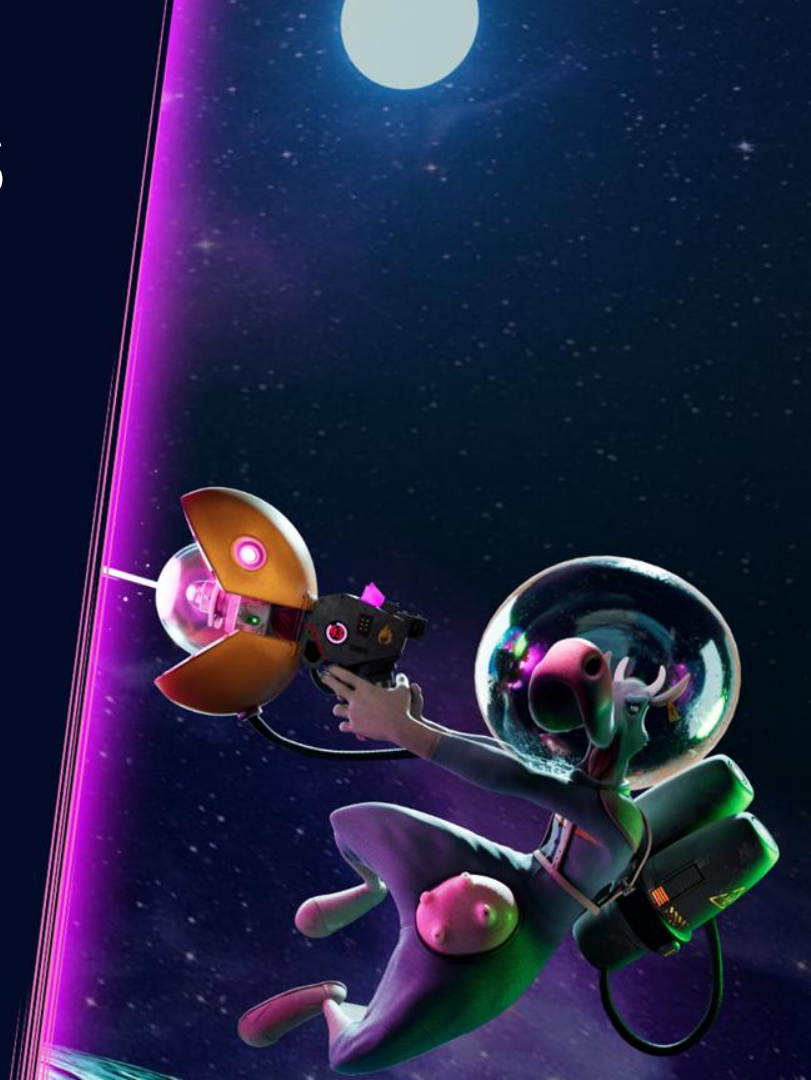
In conclusion we have looked at:

- Attributes
- Enums
- Heritage
- Scriptable Objects
- Delegates & Actions
- Coroutines
- Gizmos
- Raycasts
- Interfaces
- Saving (with JSON)
- Singleton pattern
- Observer pattern



Some references

Brackeys - YouTube
Learn Unity with simple projects
Mix and Jam - YouTube
Unity inspirational programming
Game Maker's Toolkit – YouTube
Diverse content Unity and others
#addon2023 - YouTube
Game conferences (diverse)
GDC - YouTube
Gaming industry conferences
Handmade Hero
Creating a game from nothing using C++ (600+ videos)
CppCon - YouTube
C++ talks
Améliorez votre code avec les design patterns et SOLID E-book | Unity
Design patterns & SOLID principles



Thank you for listening and

GOOD LUCK

:)