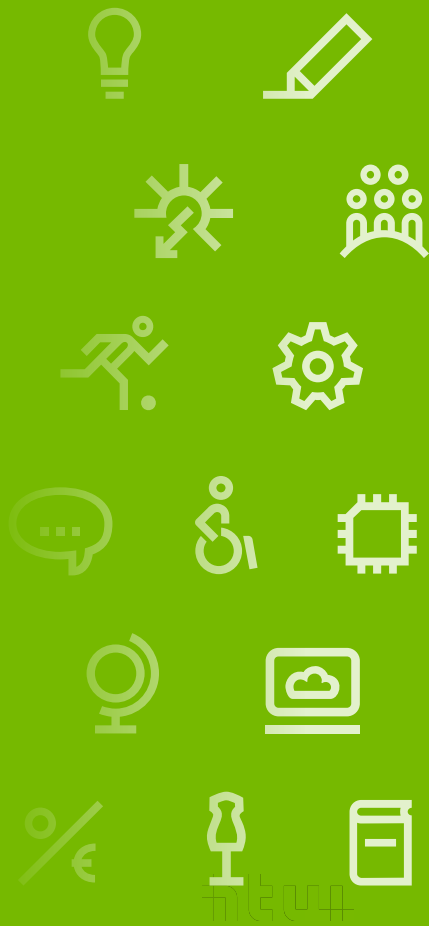
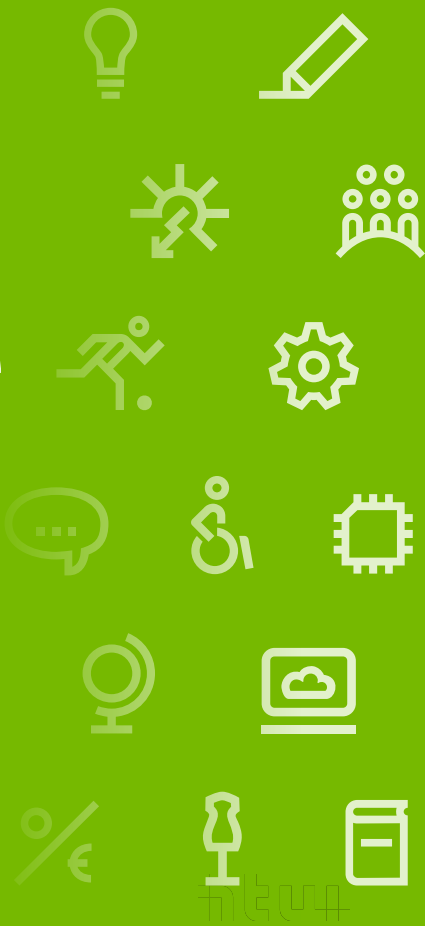
# 10. Game Engines: Advanced Game Programming II





Zaborniak, Erdem

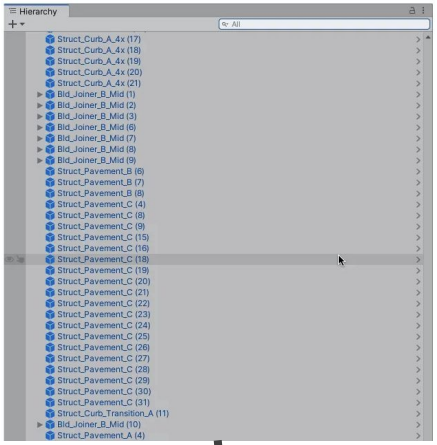
# This lecture: More Game Architecture



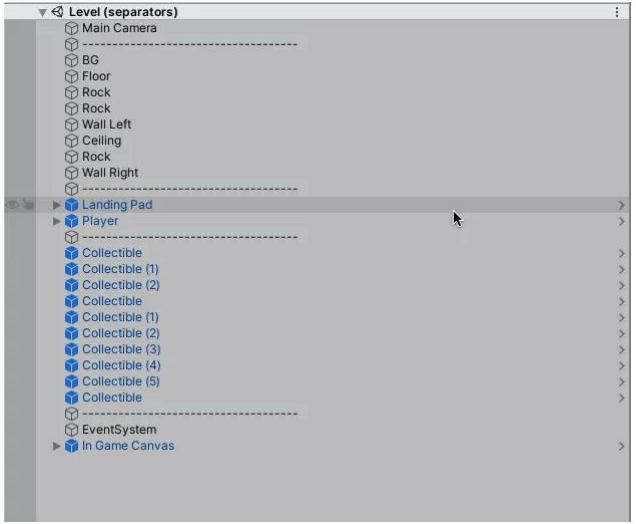
# 1. Organizing Project files

# Clean Hierarchy

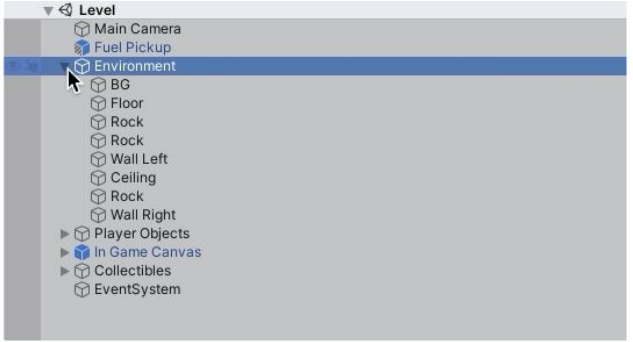
- Organise your hierarchy by grouping objects together by type or by location




#1 placing empty objects as spacers



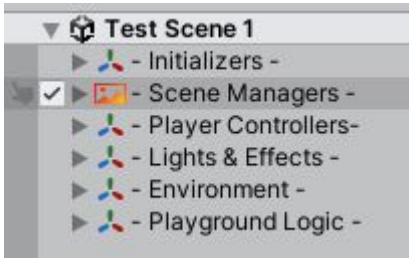

#2 grouping object (using empty objects as folders)





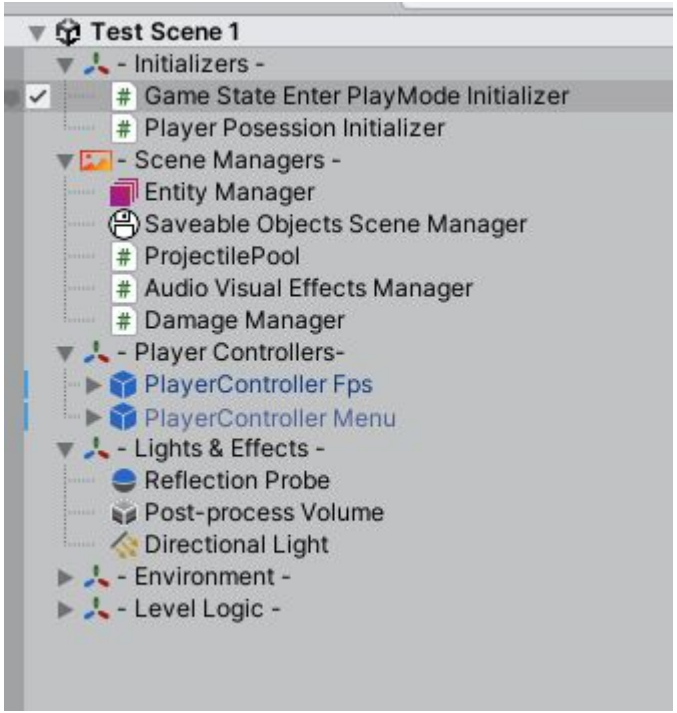

https://gamedevbeginner.com/how-to-structure-your-unity-project-best-practice-tips/

# Hierarchy Icons (Extra)





Tutorial



# Asset Structure

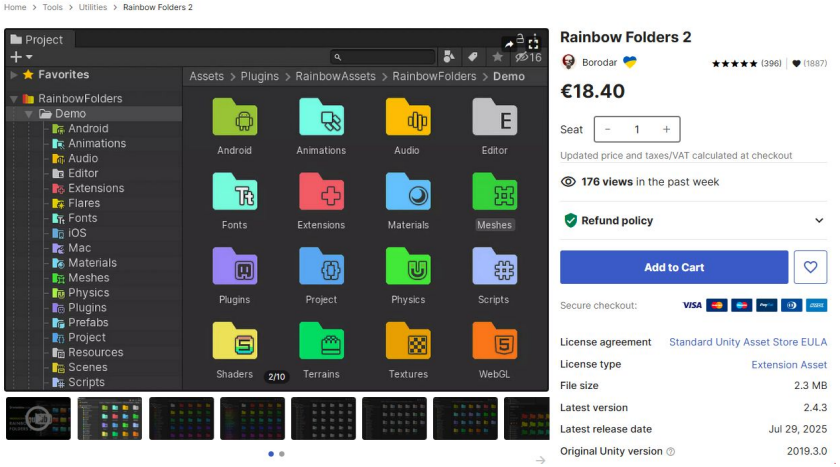
- Find a structure that suits your needs
- Mostly divided by type
- Might be a good idea to put all Visuals in a subfolder (as they have a lot of subtypes)

Example project structure

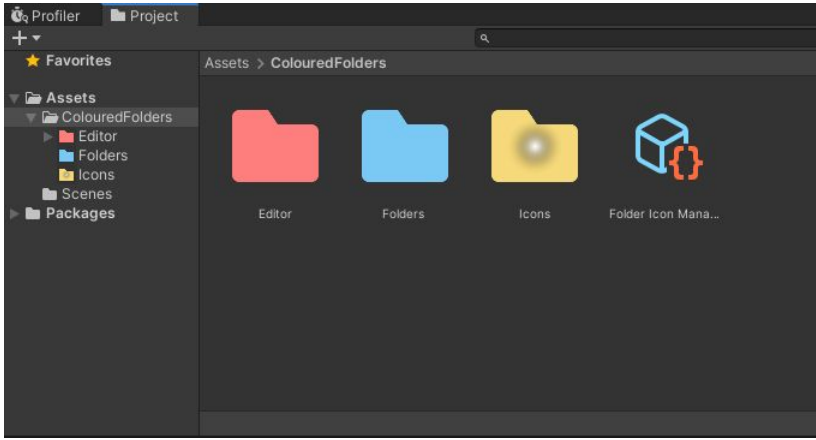
# Colored Folders & Icons

- Extra if it suits your taste

## Paid Option

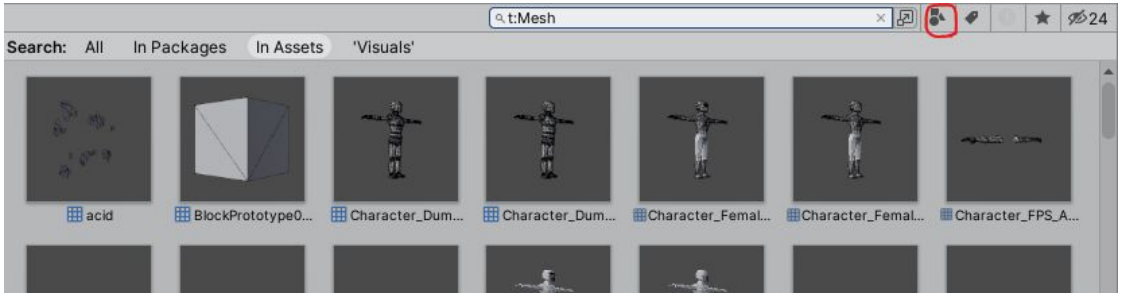


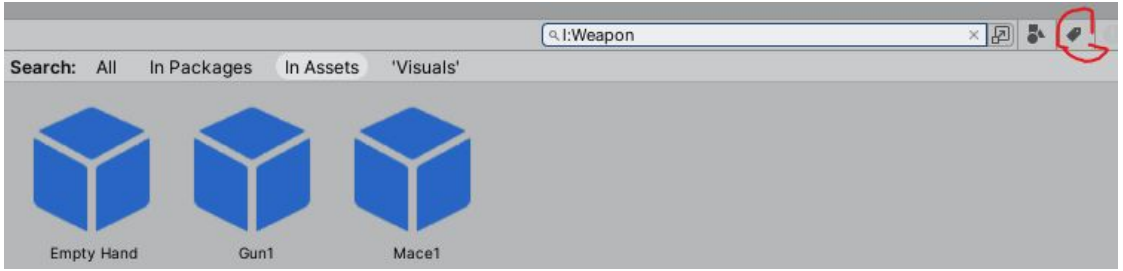

## Free alternative



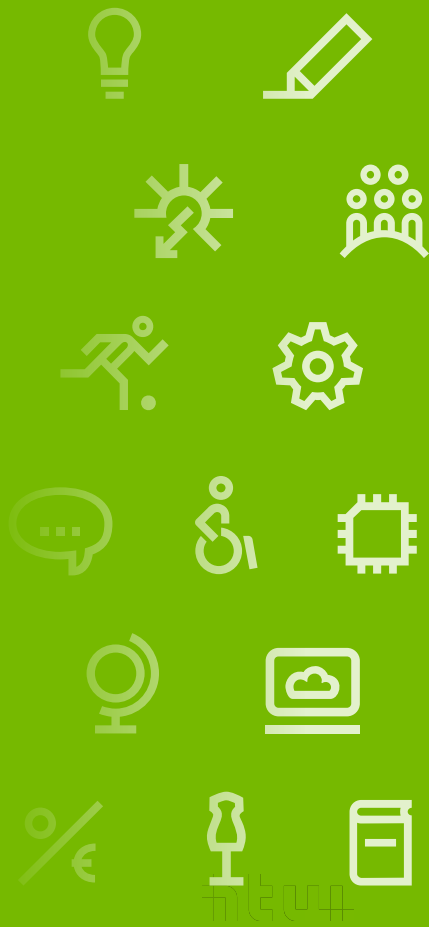
# Asset Labels

- Labels can be added to assets or they can be searched by name or by type



# 2. Organizing code

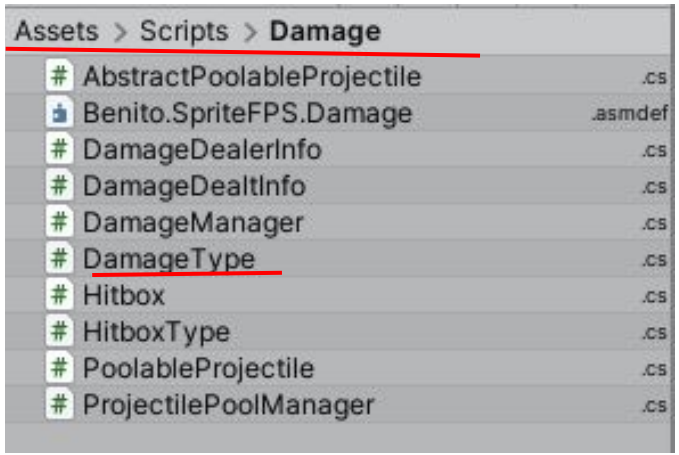


# Namespaces

- Organise namespaces to be similar to your code folder structure
- I use the format: “CompanyName.GameName.CodeGroup.CodeSubgroup…”
  - for example

```
namespace Benito.SpriteFPS.PlayerController.Fps.MovementStates
{
    public class Sprint : MovementStateMachineState
```

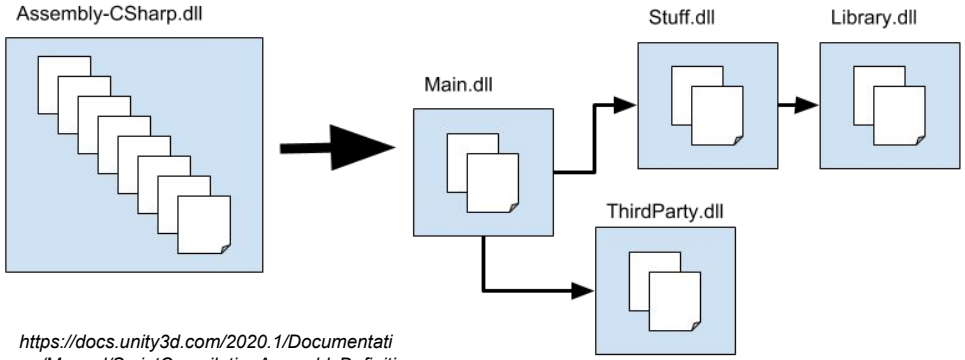
*Benito is the company name in this case*




```
namespace Benito.SpriteFPS.Damage
{
    public enum DamageType
    {
        Fire,
        Poison,

        Blunt,
        Pierce,
        Slash
    }
}
```

# Assembly Definition Files

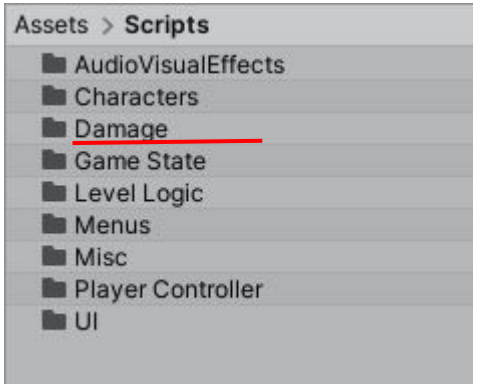
- Assembly Definition Files (.asmdef) are used to organize your scripts
- By default, Unity compiles all scripts in your project into a single assembly, but as your project grows, this can lead to
  - slow compilation times,
  - manageability issues
  - unnecessary dependencies between scripts.
- -> Better separate code into different assemblies

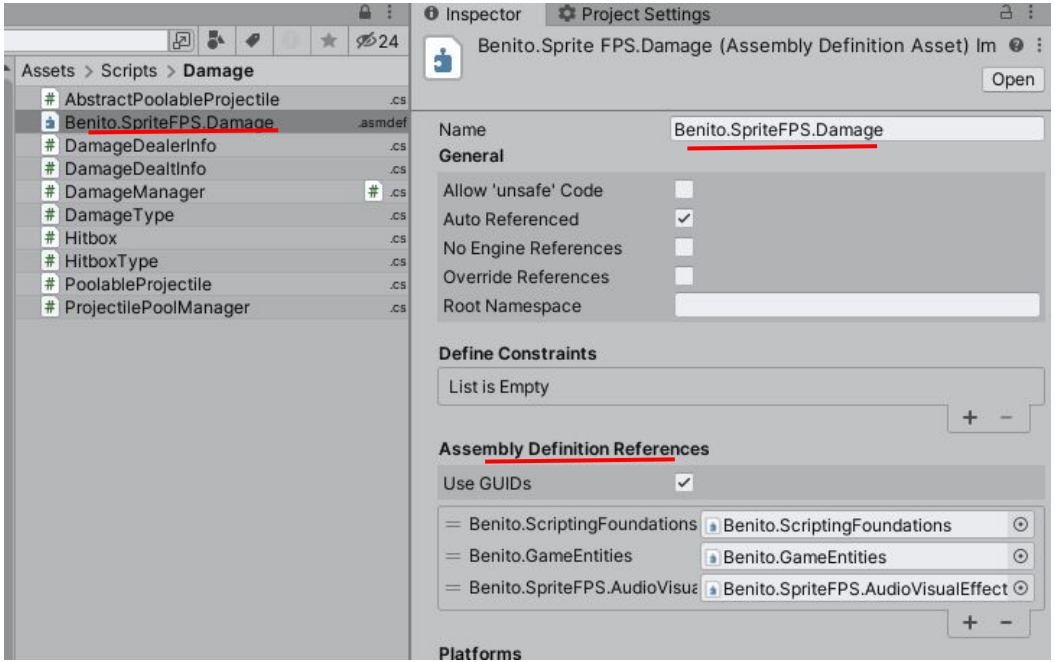



*https://docs.unity3d.com/2020.1/Documentation/Manual/ScriptCompilationAssemblyDefinitionFiles.html*

# Assembly Definition Files Examples

- Every of the folders shown below has an asmdef, a namespace and references to the other asmdefs



# Code folder structure

- There is no one-fits-all ruleset
- Depends on the game you are making
- Try to have around a maximum of 10 files per folder

*Rough structure idea*

**Player/:** Player controls

**Gameplay/**: Core mechanics like game logic, and interactions.

**UI/**: Scripts related to menus, HUDs, and other UI elements.

**Managers/**: Global systems like GameManager, AudioManager, or SaveSystem.

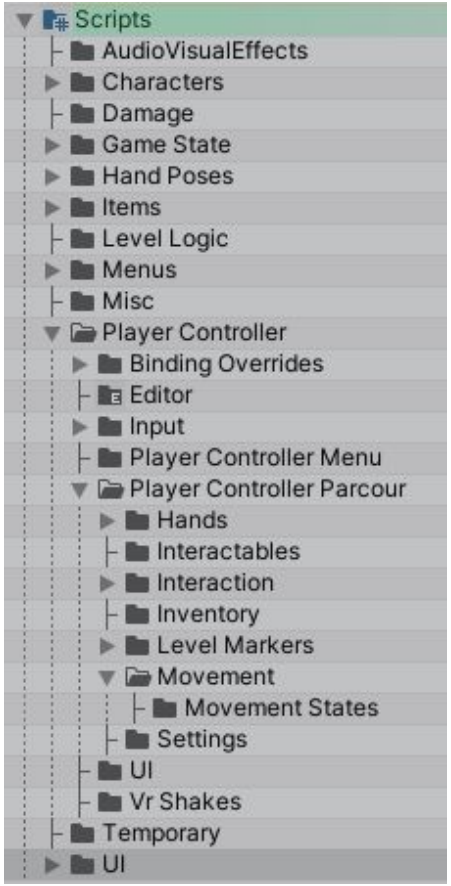
**Utilities/**: Helper scripts, extensions, and reusable utilities.

**AI/**: Scripts related to enemy behaviors or NPCs.

**Data/**: ScriptableObjects or data models (e.g., configuration settings).

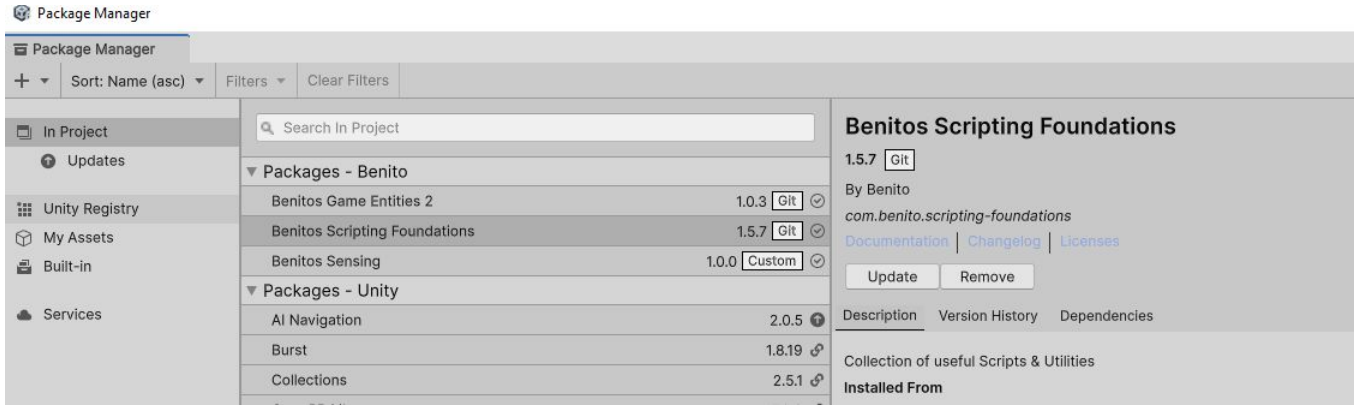
**Tests/**: Scripts for automated tests (if applicable).

*Example from a game*



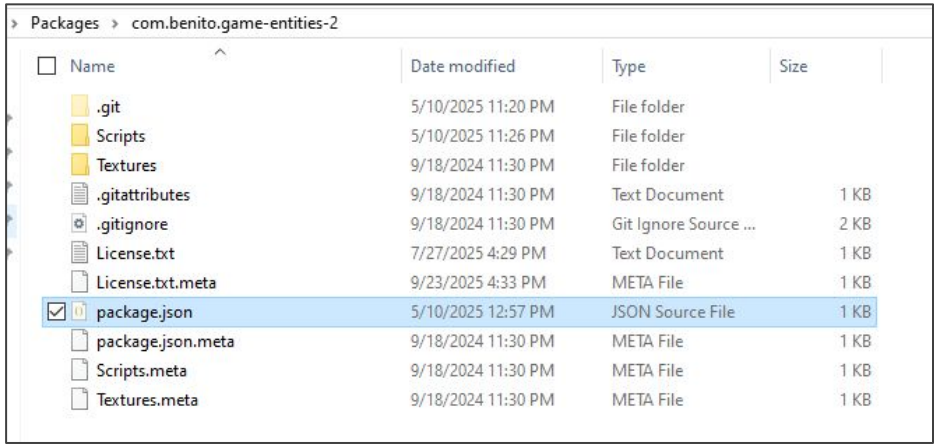
# Seperate Code into Packages

- You can share code across project by creating your own Unity packages
- Useful for sharing code for Utilities or other Systems that will be used in multiple games (For example: Navigation, Damage, AI System, …)
- Keeps projects code simpler and systems more modular
- Can be stored locally or in a git repository

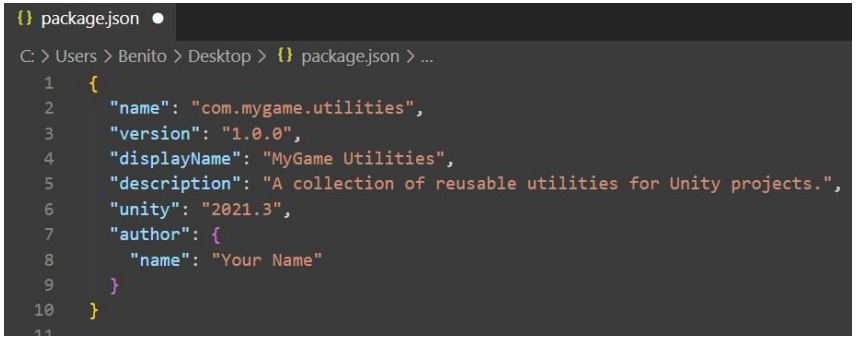


# Create Packages

- Create a folder with the appropriate package.json inside one of your projects
- Put package code and assets into this folder
- Upload this package folder to git

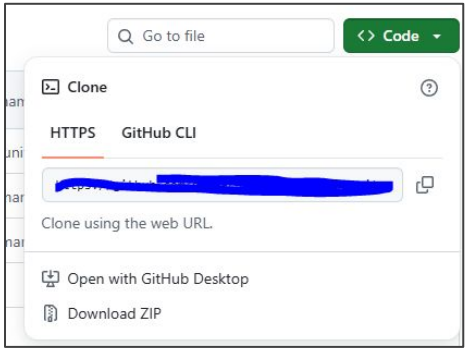


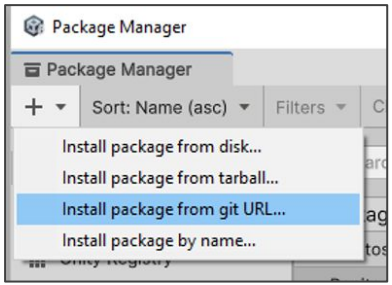

Example package.json

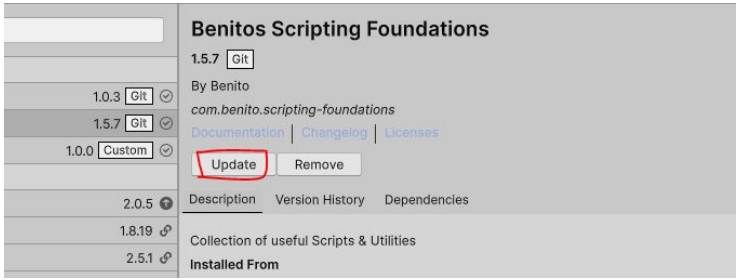


# Create Packages

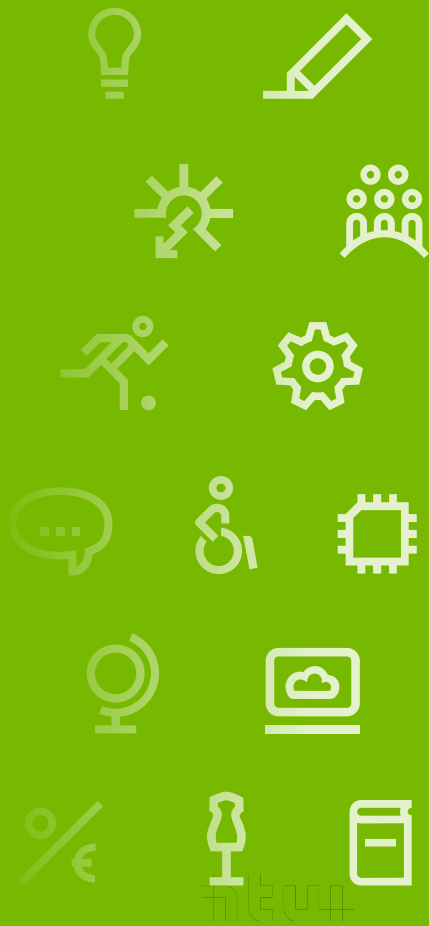
- Add this package to your other packages by using the Git HTTPS url for cloning
- Changes you made to your Github repository will be updated if you



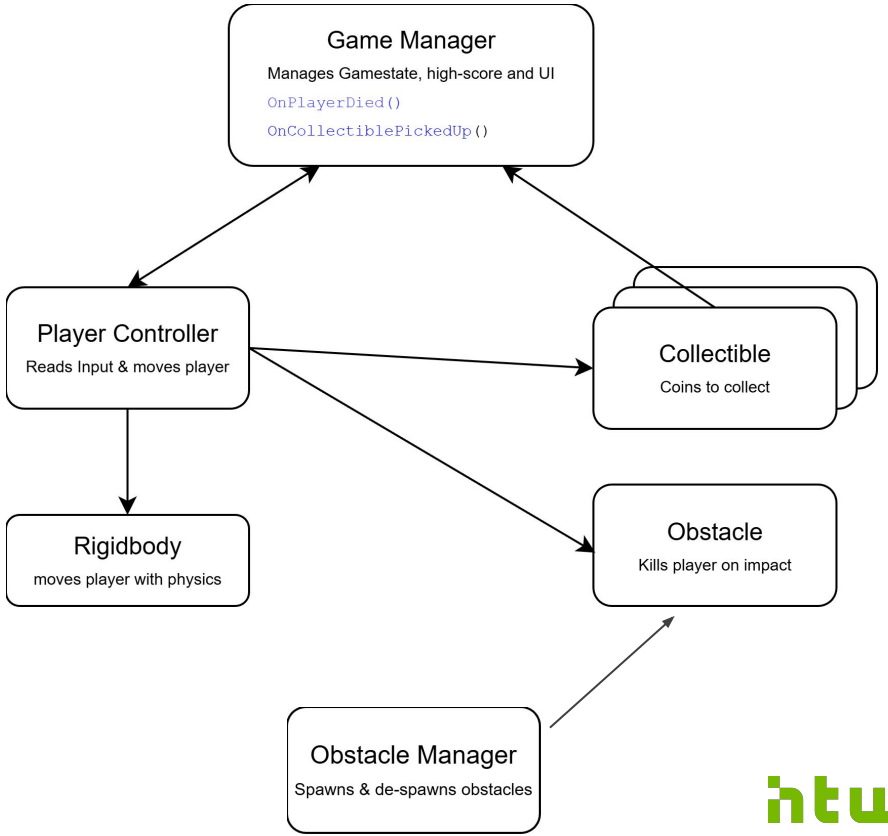



# 3. Creating Class References



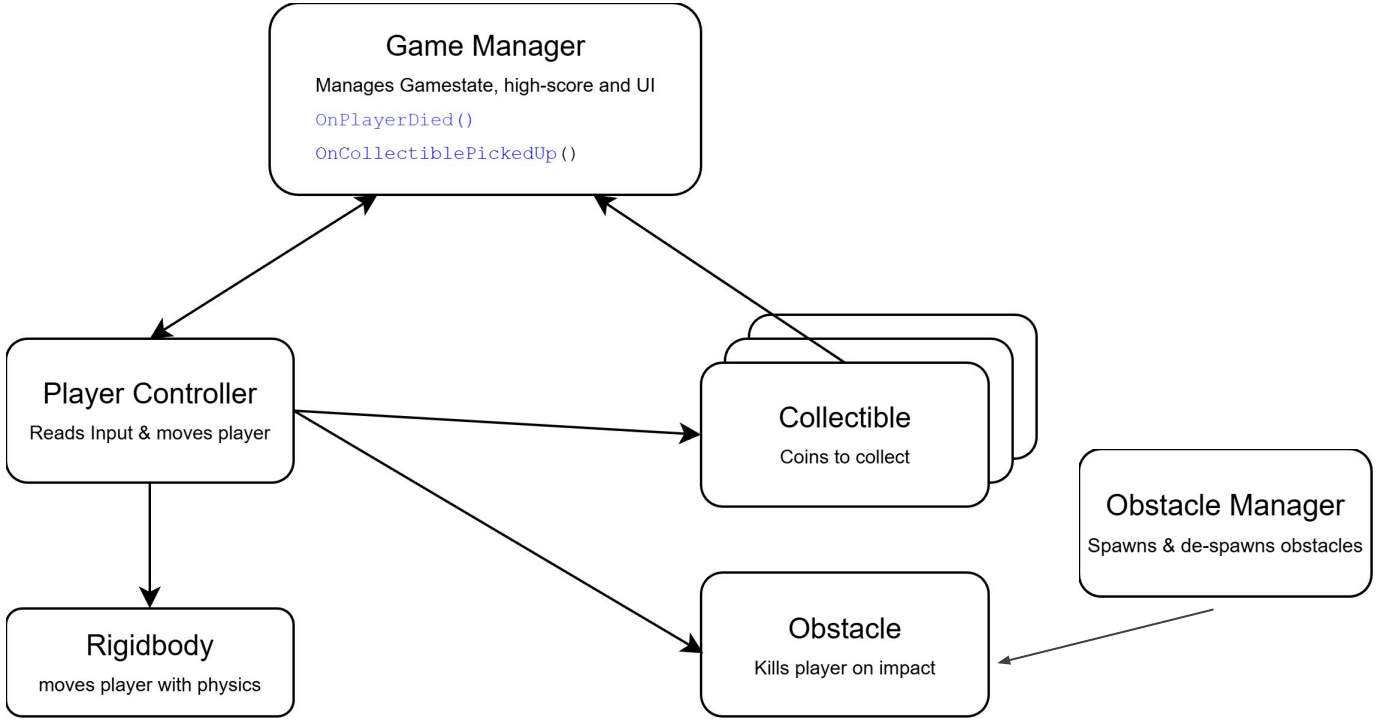
# Simple Casual Game Example

- Somehow all of those classes need to reference each other
- PlayerController useds Rigidbody to move
- PlayerController checks whether it's collided with an Obstacle or Collectible
- If PlayerController collided with an Obstacle, it calls the method OnPlayerDied() form the GameManager
- If a Collectible was picked up, it calls the CollectiblePickedUp() Method from the GameManager
- The Game Manager respawns the player upon restart



# Simple Casual Game Example

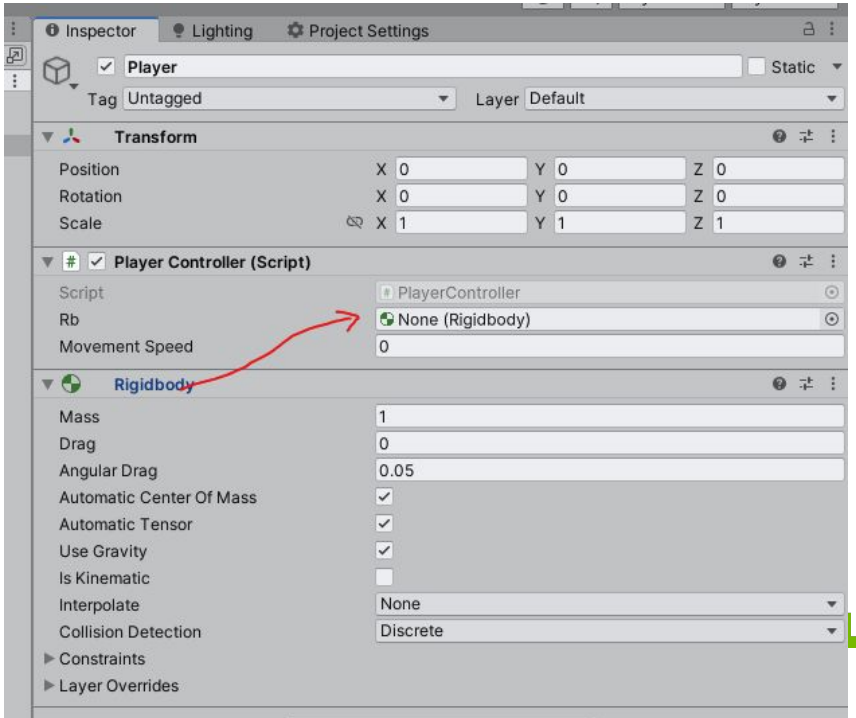
-> How can we create those references between objects?



# Assign via the inspector

- public or [SerializeField] fields are visible inside the Inspector and can be assigned via drag & drop

```
Unity Script | 0 references
public class PlayerController : MonoBehaviour
{
    [SerializeField] Rigidbody rb;

    [SerializeField] float movementSpeed;
    Vector3 desiredMove;


    Unity Message | 0 references
    void Update()
    {
        Vector2 moveInput = new Vector2(
            Input.GetAxis("Horizontal"),
            Input.GetAxis("Vertical")
            );
        desiredMove = new Vector3(moveInput.x, 0f, moveInput.y);
        desiredMove *= movementSpeed;
    }

    Unity Message | 0 references
    void FixedUpdate()
    {
        rb.velocity = desiredMove;
    }
}
```



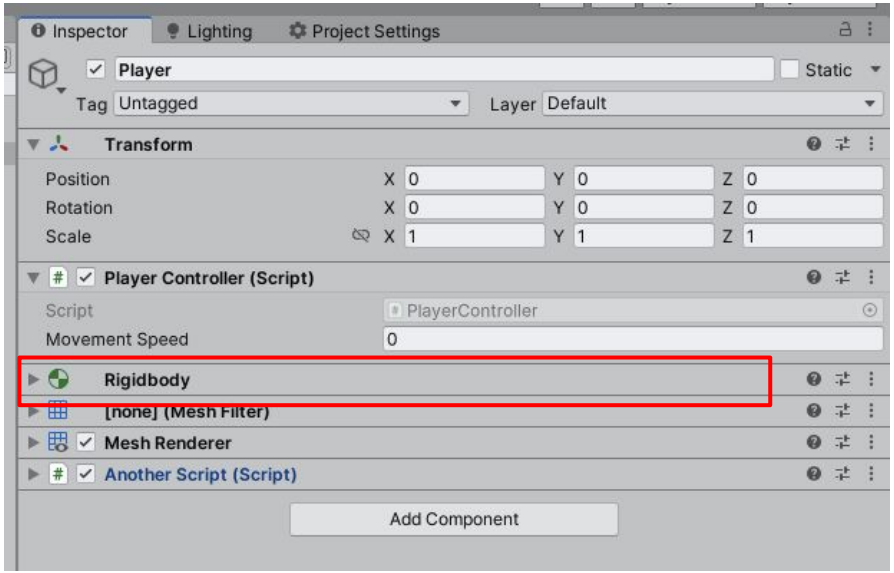
# Method 1: Assign via the inspector

- When not assigned, but used in code, they will give a NullPointerException
- Only works for objects that are already inside the Scene or a Prefab
- Does not work for objects created during the game
- Can be a lot of work to manually assign hundreds of references
- Useful to define references inside a Prefab or another hierarchy structure that won't change much during gameplay

# GameObject.GetComponent() on the same GameObject

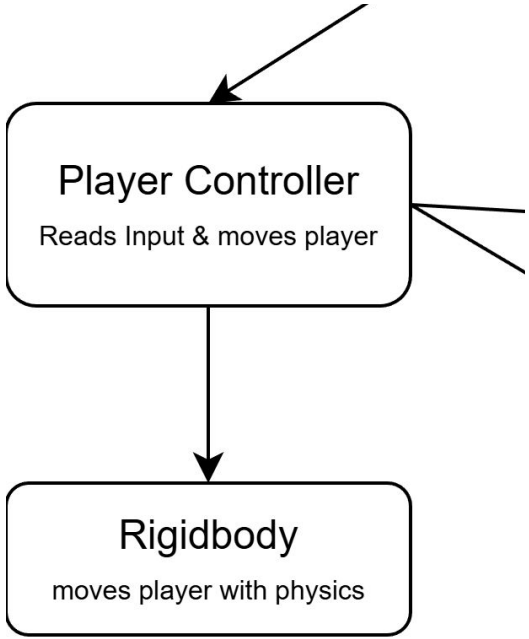
- GetComponent() returns a reference to the first Component of the specified type assigned to the GameObject
- The example below leads to the same result as assigning directly per drag and drop

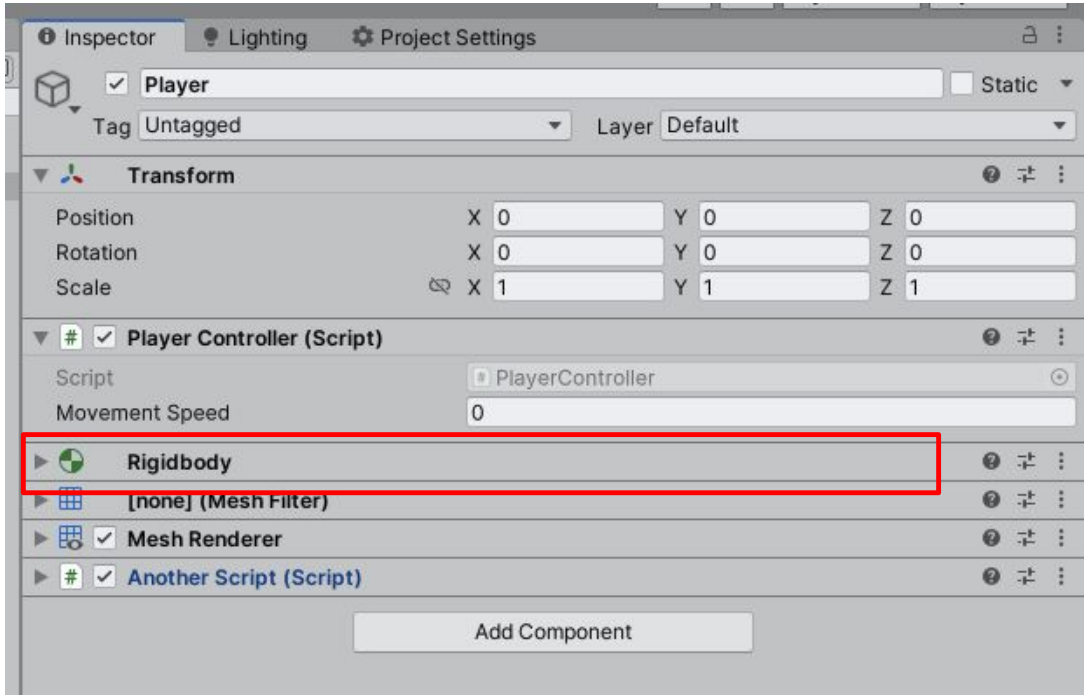
```
Unity Script (1 asset reference) | 0 references
public class PlayerController : MonoBehaviour
{
    Rigidbody rb;

    [SerializeField] float movementSpeed;
    Vector3 desiredMove;

    Unity Message | 0 references
    void Start()
    {
        // Get the Rigidbody component attached to this GameObject
        rb = GetComponent<Rigidbody>();
    }

    Unity Message | 0 references
    void Update()
    {
        Vector2 moveInput = new Vector2(
            Input.GetAxis("Horizontal"),
            Input.GetAxis("Vertical")
```



# GameObject.GetComponent() on the same GameObject

- For assigning references between components on the same GameObject



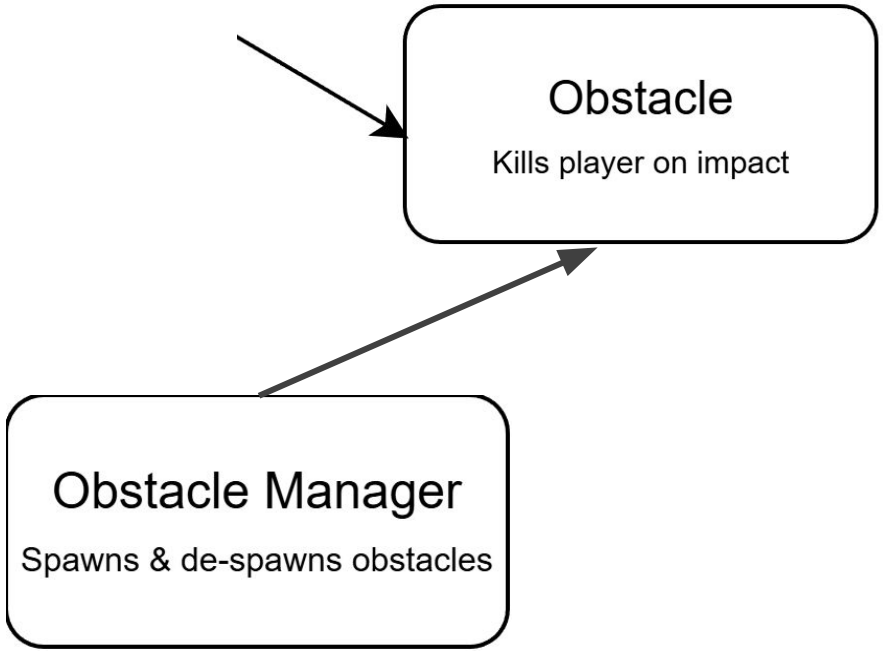
# GameObject.GetComponent() after Instantiate

- When **instantiating** Prefabs during runtime, GetComponent() can be used to establish a reference to a Component of the Prefab
- Instantiate() returns the reference to the GameObjects

```
Unity Script | 0 references
public class ObstacleSpawner : MonoBehaviour
{
    [SerializeField] float obstacleCount;
    [SerializeField] GameObject obstaclePrefab;

    Unity Message | 0 references
    void Start()
    {
        for (int i = 0; i < obstacleCount; i++)
        {
            GameObject obstacleGameObject = Instantiate(obstaclePrefab);
            Obstacle obstacle = obstacleGameObject.GetComponent<Obstacle>();
        }
    }
}
```

# GameObject.GetComponent() after Instantiate

- Useful for establishing reference between managers and managed objects



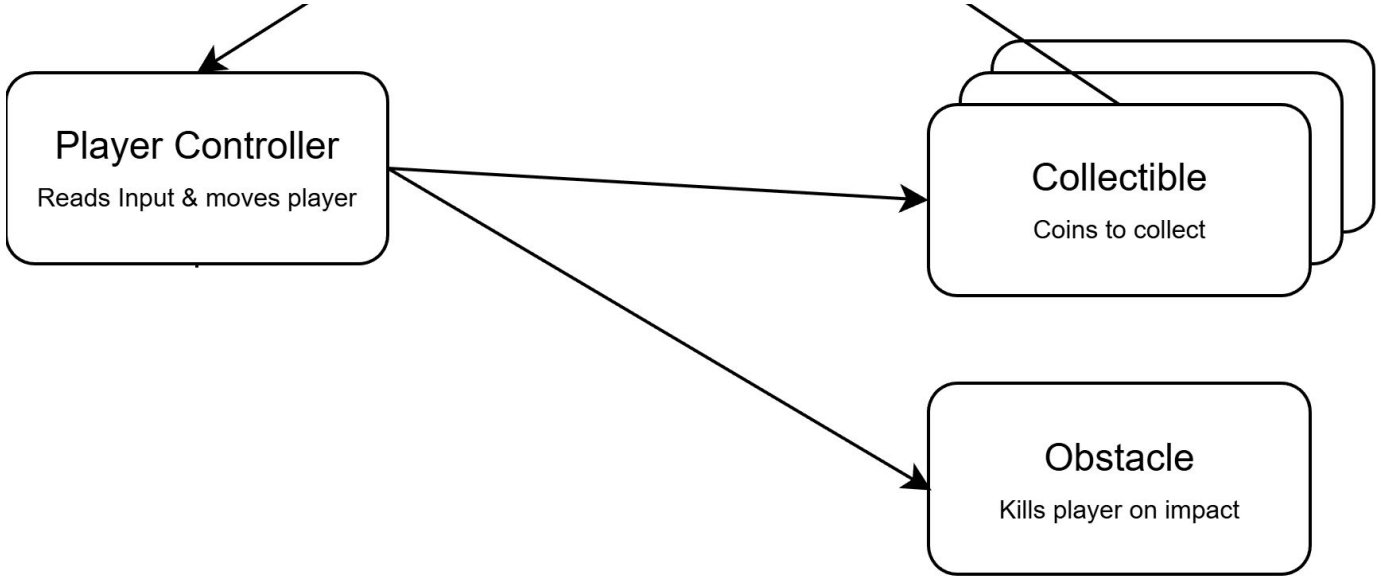
# GameObject.GetComponent() upon Collision

- Upon Collision or Trigger Enter/Exit/Stay
- Get Collision -> GameObject -> script you are searching for
- OnTrigger/OnCollision-Enter will only trigger, if at least one of the Colliders involved in the collision has a Rigidbody with kinematic set to false
- You can use collider.GetComponent() and collider.gameobject.GetComponent() interchangeably

```
Unity Message | 0 references
void OnCollisionEnter(Collision collision)
{
    //Check wether we collided with an obstacle
    Obstacle obstacle = collision.gameObject.GetComponent<Obstacle>();

    // If game object does not ocntain the obstacle component, null will be returned
    if (obstacle != null)
    {
        KillPlayerByObstacle();
    }
}
```

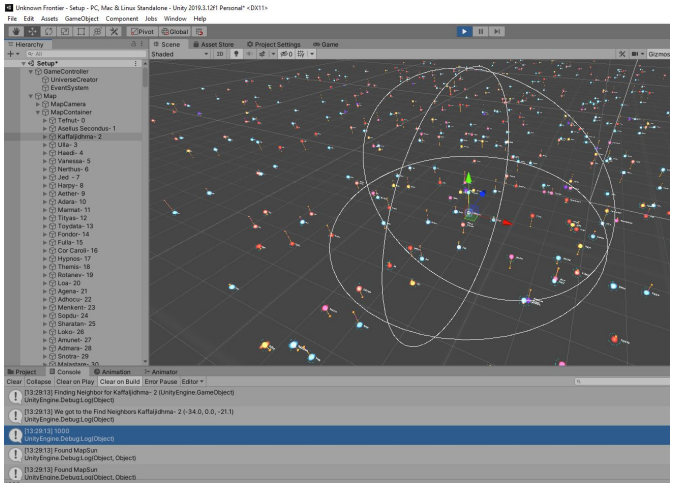
# GameObject.GetComponent() upon Collision

- Upon Collision or Trigger Enter/Exit/Stay
- Good for establishing references opon physical contact



# GameObject.GetComponent() upon Physics Cast & Overlap

- We can use the physics system to query for objects around us
- Physics.Raycast(), Physics.SphereCast(), etc… to check if a ray or sphere cast along a direction collides with objects
- Physics.OverlapSphere(), Physics.OverlapBox(), etc… to check for all objects found inside a primitive shape



```
0 references
void ScanForObstaclesAroundMe()
{
    float scanRadius = 5f;
    // Find all colliders in the scan radius
    Collider[] collidersAroundMe = Physics.OverlapSphere(transform.position, scanRadius);
    List<Obstacle> obstaclesFound = new List<Obstacle> ();

    // Iterate through each detected collider
    for (int i = 0; i < collidersAroundMe.Length; i++)
    {
        Obstacle obstacle = collidersAroundMe[i].GetComponent<Obstacle>();
        if(obstacle != null)
        {
            obstaclesFound.Add(obstacle);
        }
    }
}
```



*https://discussions.unity.com/t/physics-overlapsphere-issue/798515*

# Alternative way to get component

```
Unity Message | 0 references
private void OnTriggerEnter(Collider other)
{
    // Version 1
    Obstacle obstacle1 = other.GetComponent<Obstacle>();
    if (obstacle1 != null)
    {
        // player collided with obstacle1
    }

    // Version 2 - saves us one line :)
    if (other.TryGetComponent<Obstacle>(out Obstacle obstacle2))
    {
        // player collided with obstacle2
    }
}
```

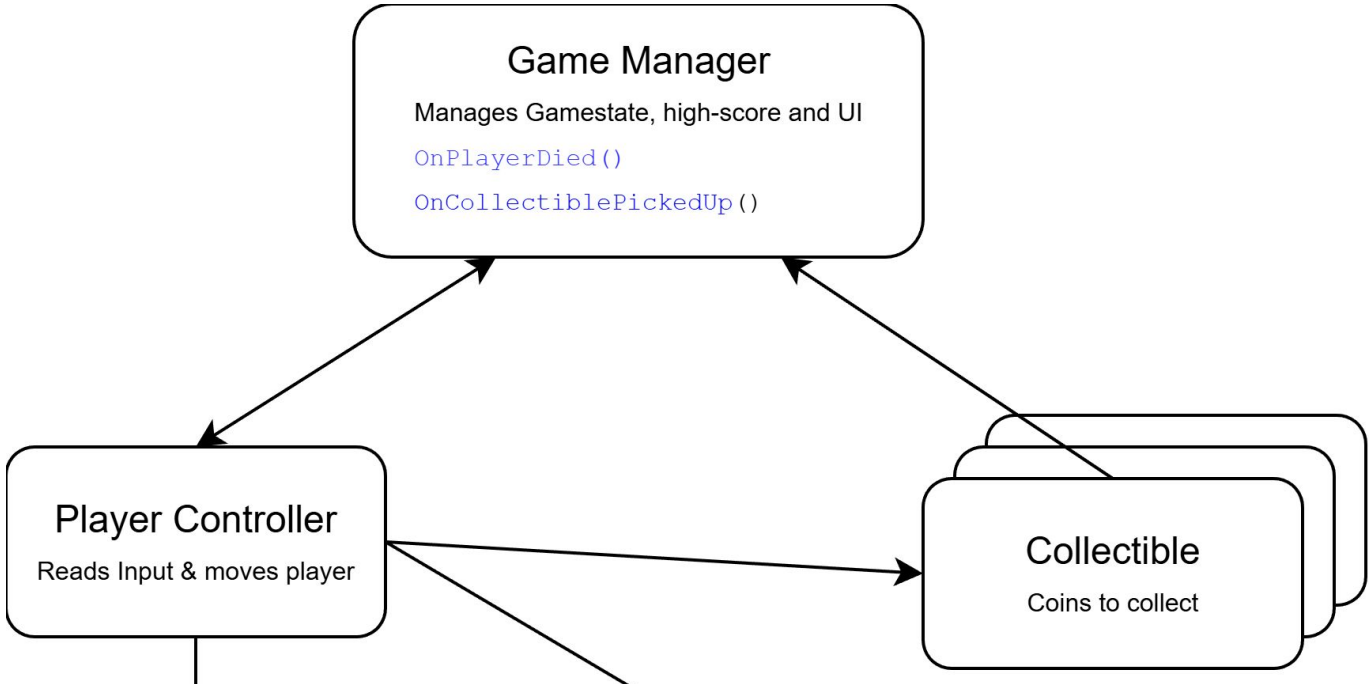
# Using Game Object.Find or FindObjectOfType

- To Find an object in the game world that is not being collided with
- Mostly used to find various managers
- Both methods can be avoided by using a Singleton for example
- It can cost too much performance, as Unity traverses the whole Scene hierarchy

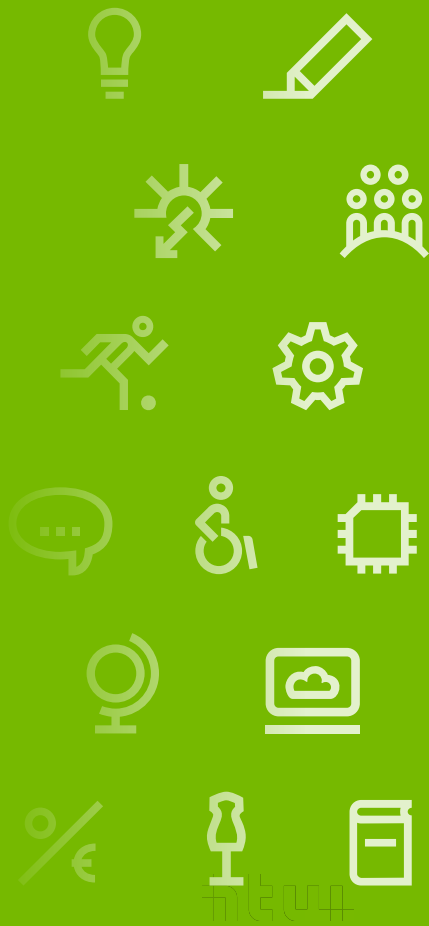
```
1 reference
void KillPlayerByObstacle()
{
    // Error prone and bad performance, renaming objects in the hierarchy might break your game
    GameManager gameManager1 = GameObject.Find("GameManager").GetComponent<GameManager>();

    // Bad performance, but can be used occasionally
    GameManager gameManager2 = GameObject.FindFirstObjectByType<GameManager>();
}
```

# Using Singleton Managers

- Good way for communication of managers with several objects in the game world

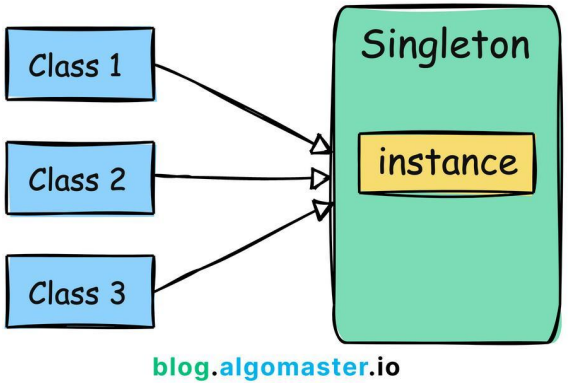


# 3.2. Singleton Managers

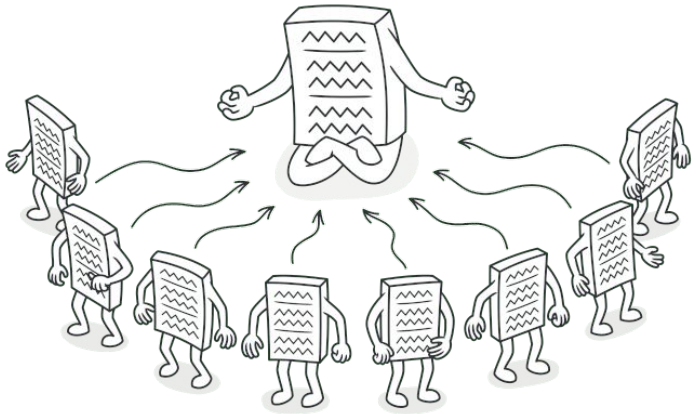


# Singletons

- Design pattern that ensures a class has only one instance and provides a global point of access to that instance
- Mostly used for managers that manage specific aspects of your game (GameManager, AudioManager, UIManager)
- Make sure to have only one Manager Instance




https://blog.algomaster.io/p/singleton-design-pattern



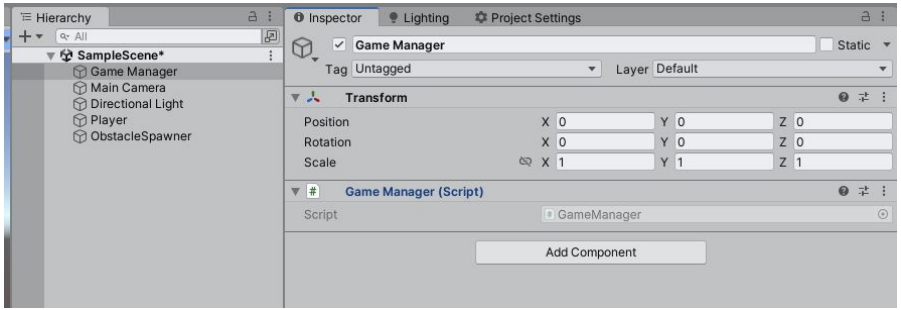
https://refactoring.guru/design-patterns/singleton

# Singletons - implementation

- The static property (Instance) allows global access without requiring explicit references.
- Manager can be accessed from anywhere by calling the static Manager.Instance field
- Awake() gets called before start

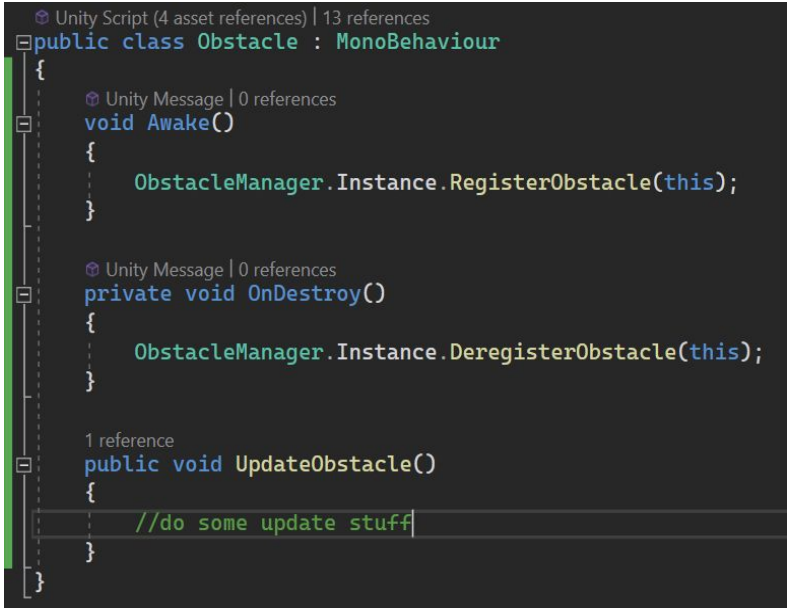
```
Unity Script | 5 references
public class GameManager : MonoBehaviour
{
    // The static instance that ensures there's only one GameManager
    3 references
    public static GameManager Instance { get; private set; }

    Unity Message | 0 references
    void Awake()
    {
        // Check if an instance already exists
        if (Instance != null && Instance != this)
        {
            Destroy(gameObject); // Destroy the duplicate
            return;
        }

        Instance = this; // Set the instance to this instance of GameManager
        //DontDestroyOnLoad(gameObject); // (optional) Persist across scenes
    }
}
```

```
2 references
void KillPlayerByObstacle()
{
    GameManager.Instance.OnPlayerDied();
}
```



# Singletons - objects self register example

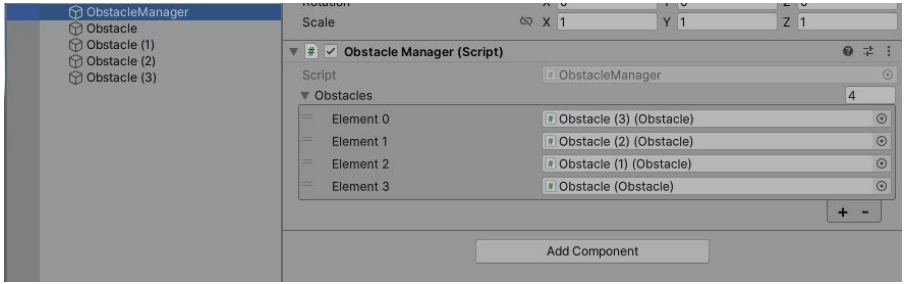
- Objects can register themselves to be managed by a Singleton manager
- Useful to create custom collections, for example:
  - for managing enemies/units
  - managing projectiles
  - spawn points manager
  - save/load system
  - optimization by updating scripts in a custom way



# Singletons - objects self register example

- Make sure that the manager's Awake() is called before other objects try to access it by modifying the script's [DefaultExecutionOrder] order

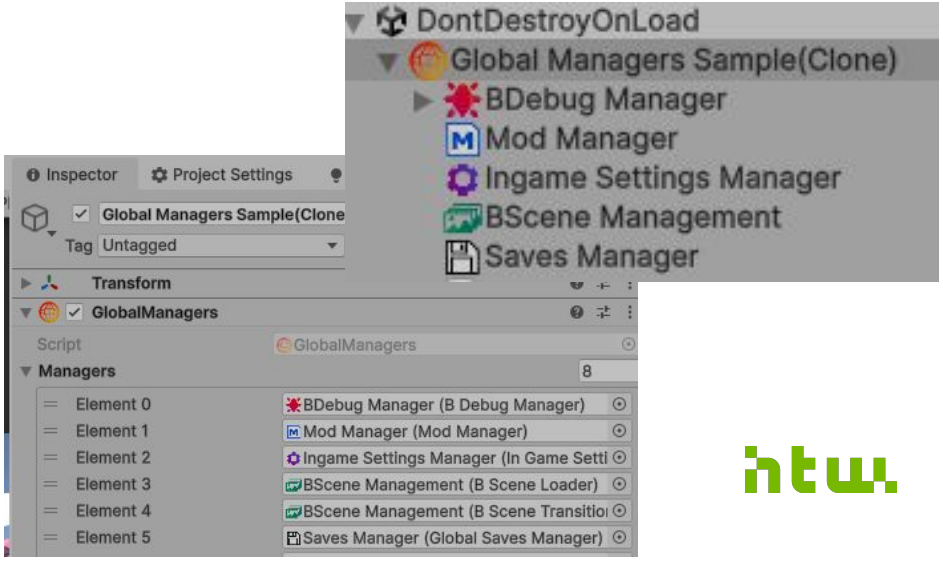
```
Unity Script (4 asset references) | 13 references
public class Obstacle : MonoBehaviour
{
    Unity Message | 0 references
    void Awake()
    {
        ObstacleManager.Instance.RegisterObstacle(this);
    }

    Unity Message | 0 references
    private void OnDestroy()
    {
        ObstacleManager.Instance.DeregisterObstacle(this);
    }

    1 reference
    public void UpdateObstacle()
    {
        //do some update stuff
    }
}
```




```
[DefaultExecutionOrder(-10)]
Unity Script (1 asset reference) | 3 references
public class ObstacleManager : MonoBehaviour
{
    [SerializeField] List<Obstacle> obstacles = new List<Obstacle>();
    5 references
    public static ObstacleManager Instance { get; private set; }

    Unity Message | 0 references
    void Awake()
    {
        // Check if an instance already exists
        if (Instance != null && Instance != this)
        {
            Destroy(gameObject); // Destroy the duplicate
            return;
        }

        Instance = this;
    }

    1 reference
    public void RegisterObstacle(Obstacle obstacle)
    {
        obstacles.Add(obstacle);
    }

    1 reference
    public void DeregisterObstacle(Obstacle obstacle)
    {
        obstacles.Remove(obstacle);
    }

    Unity Message | 0 references
    public void Update()
    {
        foreach (Obstacle obstacle in obstacles)
        {
            obstacle.UpdateObstacle();
        }
    }
}
```

# Manager Locator

- It may be good practice to only have one Singleton per scene and one Global Singleton inside the DontDestroyOnLoad scene
- This Singleton holds references to all managers and acts as an interface to access all managers -> Similar to a “ServiceLocator”
- -> Cleaner code, as we only need to implement the Singleton logic once

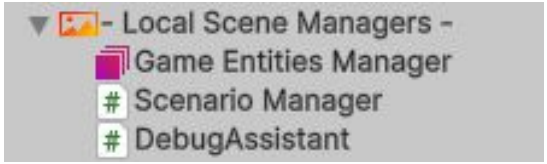



# Manager Locator - Example

- Reference all Managers via the local or global ManagerLocator

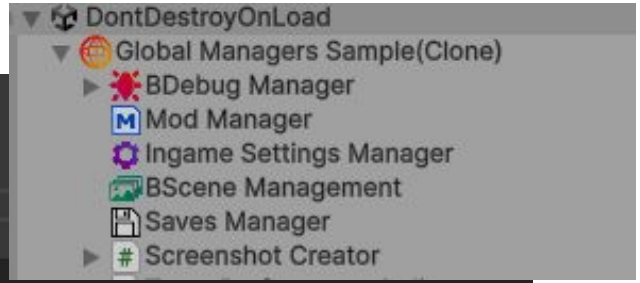
Access a local manager:

```
Unity Message | 0 references
protected virtual void Start()
{
    // Access from anywhere
    LocalSceneManagers.Get<GameEntityManager>().RegisterEntity(this);
}
```




Access a global manager:

```
public void CreateSaveFile()
{
    Texture2D saveScreenshot = GlobalManagers.Get<ScreenshotCreator>().
        CreateScreenshot(cameraToTakeScreenshotFrom, 1280,720);

    SceneSaveInfoCreateModel info = new SceneSaveInfoCreateModel...;

    GlobalManagers.Get<GlobalSavesManager>().CreateSceneSaveForCurrentScene(saveFolderPathInSavesFolder, info);
}
```



# Example Implementation

- Uses a dictionary internally to map the desired Manager Type to the correct manager instance
- Updates all registered Managers

```
#region Updating the Managers

Unity Message | 0 references
private void Start()
{
    for (int i = 0;i < Instance.managers.Count; i++)
    {
        Instance.managers[i].InitialiseManager();
    }
}

Unity Message | 0 references
private void Update()
{
    for (int i = 0; i < Instance.managers.Count; i++)
    {
        Instance.managers[i].UpdateManager();
    }
}

#endregion
```

```
[DefaultExecutionOrder(-5)]
Unity Script (1 asset reference) | 3 references
public class ManagersManager : MonoBehaviour
{
    [SerializeField] protected List<SingletonManager> managers = new List<SingletonManager>();

    public static ManagersManager Instance;

    Dictionary<Type, object> managerDictionary = new Dictionary<Type, object>();

    #region Singleton Code
    Unity Message | 0 references
    private void Awake()
    {
        if (Instance != null && Instance != this)
        {
            // Destroy if duplicate
            Destroy(gameObject);
            return;
        }

        Instance = this;

        // Set up Manager Dictionary
        for (int i = 0; i < Instance.managers.Count; i++)
        {
            Instance.managerDictionary.Add(managers[i].GetType(), managers[i]);
        }
    }
    #endregion

    2 references
    public static T Get<T>()
    {
        if (!Instance.managerDictionary.ContainsKey(typeof(T)))
        {
            Debug.LogError($"Type: {typeof(T)} could not be found inside the ManagerLocator");
            return default(T);
        }

        return (T)Instance.managerDictionary[typeof(T)];
    }
```

# Example Implementation

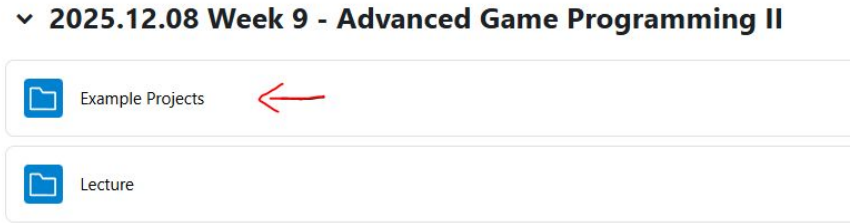
- Every Manager has to derive from Singleton Manager base-class

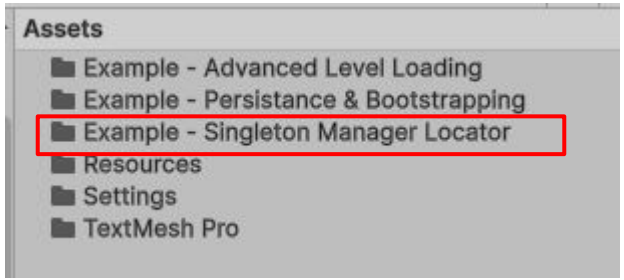
```
Unity Script | 4 references
public abstract class SingletonManager: MonoBehaviour
{
    3 references
    public abstract void InitialiseManager();

    3 references
    public abstract void UpdateManager();
}
```

```
Unity Script (1 asset reference) | 1 reference
public class ExampleManager1 : SingletonManager
{
    2 references
    public override void InitialiseManager()
    {
        Debug.Log("Initialize Example Manager 1");
    }

    2 references
    public override void UpdateManager()
    {
        //Debug.Log("Update Example Manager 1");
    }

    1 reference
    public void Manager1ExampleMethod()
    {
        Debug.Log("Manager 1 Example Method Called");
    }
}
```

```
Unity Script (1 asset reference) | 1 reference
public class ExampleManager2 : SingletonManager
{
    2 references
    public override void InitialiseManager()
    {
        Debug.Log("Initialize Example Manager 2");
    }

    2 references
    public override void UpdateManager()
    {
        //Debug.Log("Update Example Manager 2");
    }

    1 reference
    public void Manager2ExampleMethod()
    {
        Debug.Log("Manager 2 Example Method Called");
    }
}
```

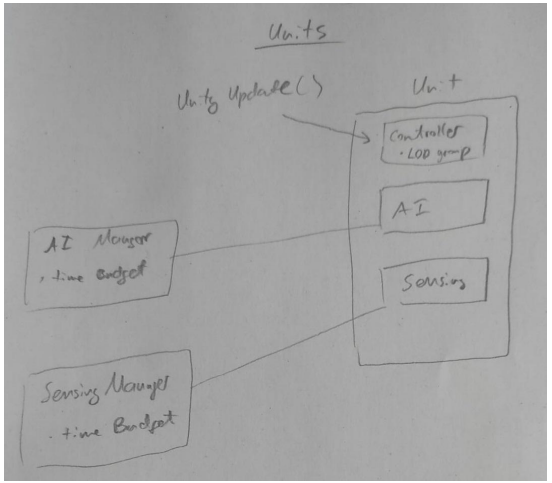
# Example Project



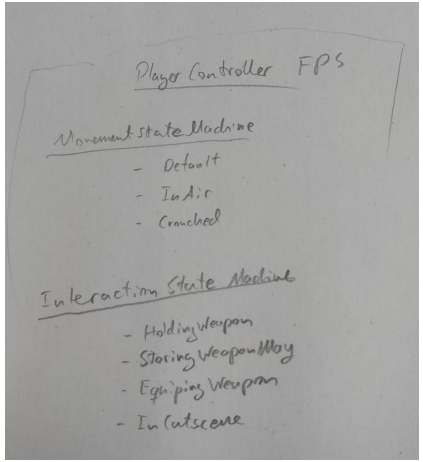



# Managing References Finishing Words -> Plan your code

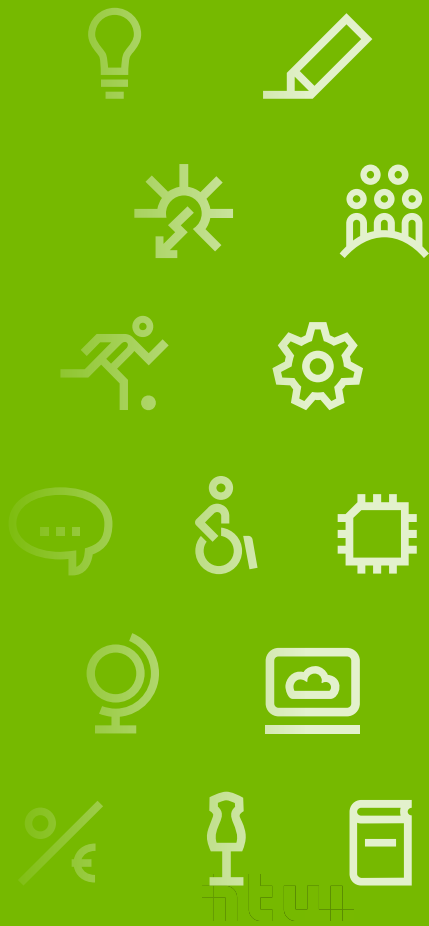
- Take a few minutes before starting to code
- **Sketch** out relationships of you classes
- Strive for **low coupling** and **encapsulation** -> less complexity, fewer bugs
- **Don’t over-engineer** - quick and dirty may be better than perfect but never finished




*Even quick and dirty sketches can be very helpful*



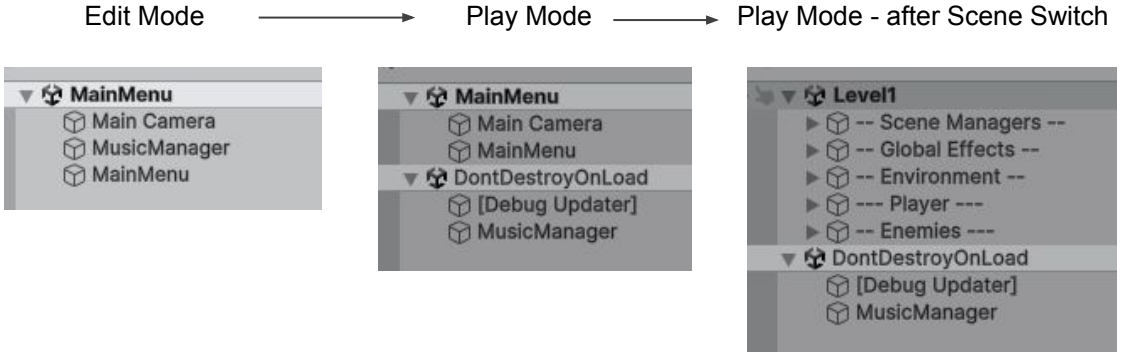
# 4. Bootstrapping & Persistence



# Bootstrapping & Persistence

- For a lot of games we will need certain Managers that will stay persistent across all scenes (For example Music, GameManager, SavesManager)
- One simple way to achieve this is to have those in the first scene of our game (for example - Main Menu) and let them register themselves into the DontDestroyOnLoad Scene
- However this system will break if we start our game through another scene (for example for testing purposes) -> the fixes for that are often tedious

```
Unity Script (1 asset reference) | 0 references
public class MusicManager : MonoBehaviour
{
    Unity Message | 0 references
    void Start()
    {
        DontDestroyOnLoad(gameObject);
    }

    0 references
    void PlayMusic(string trackName)
    {
        // play it
    }
}
```

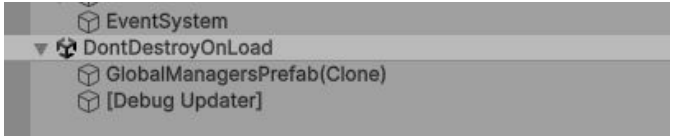


# Alternative Solution

The script does't need to be on a Gameobject, it should just be inside your codebase and will be run automatically.

- Alternative you can use the following script in combination with a Prefab inside the Resources folder
- Possible by using the [RuntimeInitializeOnLoadMethod] attribute

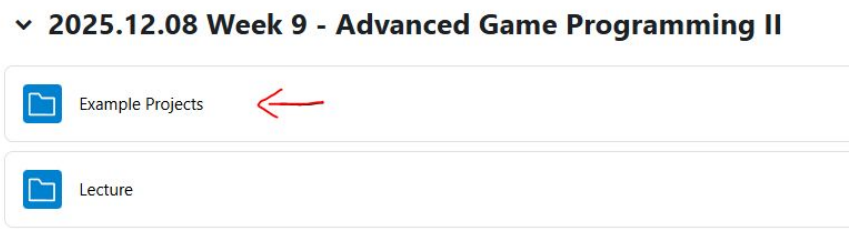

Make sure this prefab is present with all your global managers

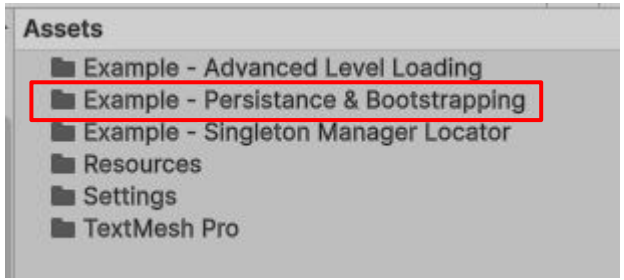



The prefab will be present in every scene, no matter in which scene you start

```
/// <summary>
/// Makes sure a DontDestroyOnLoad Collection with Global Managers is present in every Scene
/// </summary>
0 references
public class GlobalManagersInitializer
{
    static GameObject Instance = null;

    // Make sure there is a prefab inside Assets/Resources/Prefabs named GlobalManagersPrefab.
    // Update if necessary.
    private const string prefabPath = "Prefabs/GlobalManagersPrefab";


    [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]
    0 references
    static void InitializeGlobalManagers()
    {
        if (Instance == null)
        {
            GameObject globalManagersPrefab = Resources.Load<GameObject>(prefabPath);

            if (globalManagersPrefab != null)
            {
                // Instantiate the prefab
                GameObject globalManagers = GameObject.Instantiate(globalManagersPrefab);
                Instance = globalManagers;
                GameObject.DontDestroyOnLoad(globalManagers);
            }
            else
            {
                Debug.LogError("Global Managers Prefab not found in Resources folder.");
            }
        }
    }
}
```

# Example Project



# 5. Advanced Level Loading

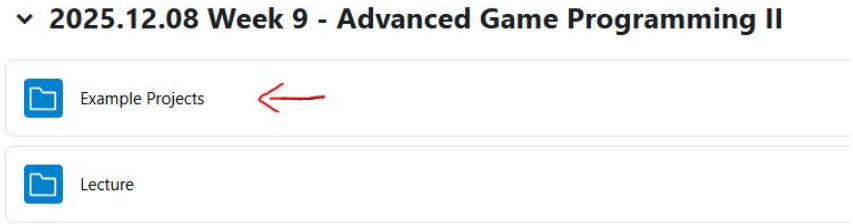
# Coroutine load level example

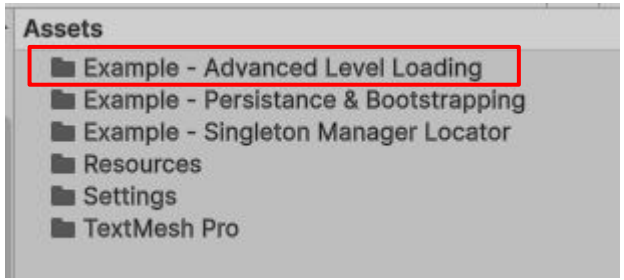
- When working with big scenes, your game can freeze while loading a scene on the main thread.
- Use SceneManager.LoadSceneAsync() to load a scene in a background and prevent your game from freezing

Example implementation with a progress bar

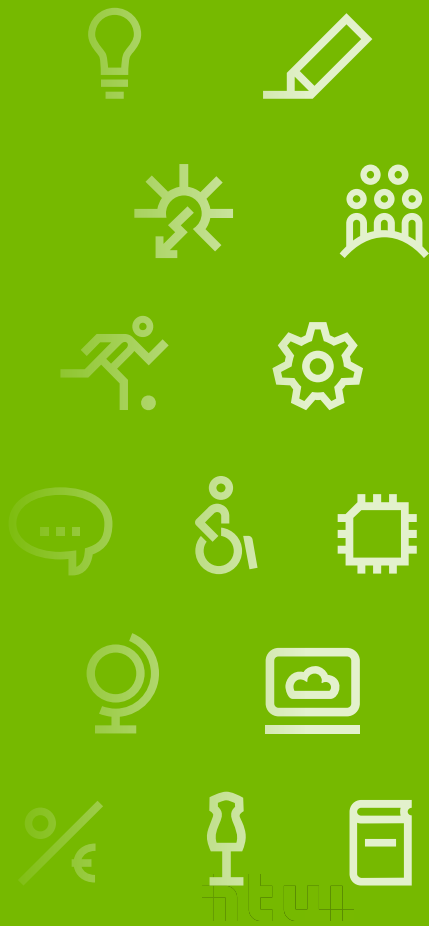
```
// Coroutine to load the scene asynchronously and update the progress bar
1 reference
private IEnumerator LoadSceneAsync()
{
    transitionScreen.SetActive(true);
    AsyncOperation asyncOperation = SceneManager.LoadSceneAsync(sceneToLoad);

    // Prevent scene from activating immediately, so we can track progress
    asyncOperation.allowSceneActivation = false;

    while (!asyncOperation.isDone)
    {
        progressBar.value = asyncOperation.progress;
        Debug.Log($"Loading next scene async progress: {asyncOperation.progress}");
        // For some reason in Unity the scene onlu loads till 0.9f / 90%
        if (asyncOperation.progress >= 0.9f)
        {
            progressBar.value = 1f;
            asyncOperation.allowSceneActivation = true;
            Debug.Log($"Loading next scene finished");
        }
        yield return null;
    }
}
```

# Example Project



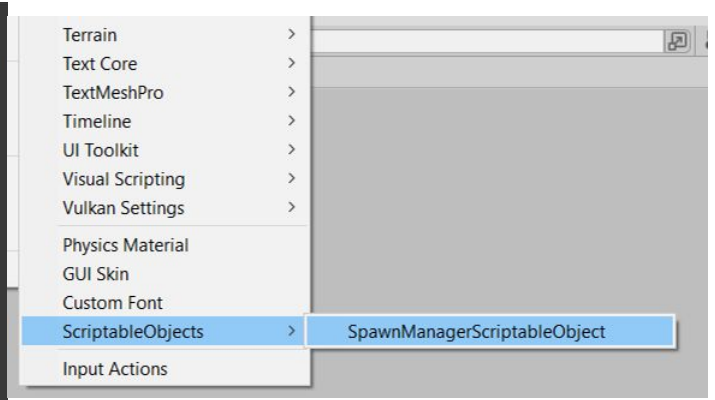
# 6. Scriptable Objects



# Scriptable Objects

- A class stored as .asset files in the project, editable via the Inspector
- Share data and logic across scenes without duplication
- Can hold variables and methods
- [CreateAssetMenu] Attribute describes how it can be created as an asset via RightClick

```
using UnityEngine;

[CreateAssetMenu(fileName = "Data", menuName = "ScriptableObjects/SpawnManagerScriptableObject", order = 1)]
0 references
public class SpawnManager : ScriptableObject
{
    public string prefabName;

    public int numberOfPrefabsToCreate;
    public Vector3[] spawnPoints;

    0 references
    public Vector3 GetRandomSpawnPoint()
    {
        return spawnPoints[Random.Range(0, spawnPoints.Length)];
    }
}
```





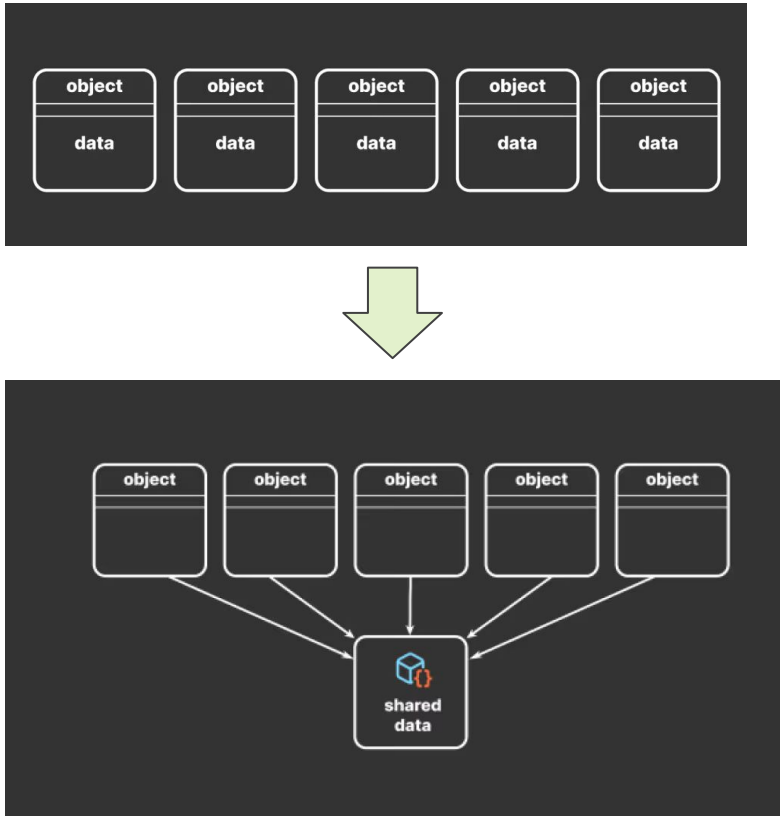

With the previous script in your Assets folder, you can create an instance of your ScriptableObject by navigating to Assets > RightMouseClick > Create > ScriptableObjects > SpawnManagerScriptableObject.

# Scriptable Objects - Uses

- Data Container
  - Store unchanging data of MonoBehaviour scripts inside Prefabs - “global variable holder” (e.g. settings and stats for Npc’s, items)
  - Game configuration  (e.g., difficulty settings, audio profiles)
- Used as extended enums
- As runtime sets
- Event broadcasting through custom event systems
- .. and many more
- Usage examples from Unity here

# Scriptable Objects - as Data Container

- Store stats or settings for a character or object in a game
- Every instance of this object in the game can reference the same settings file
- Improves performance and makes work with those settings easier for non-programmers

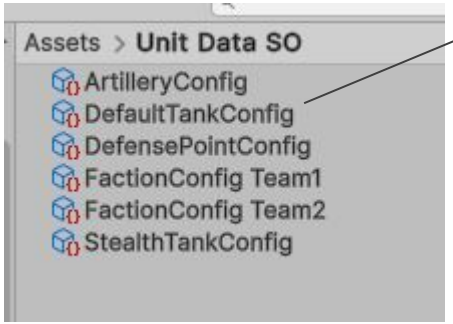


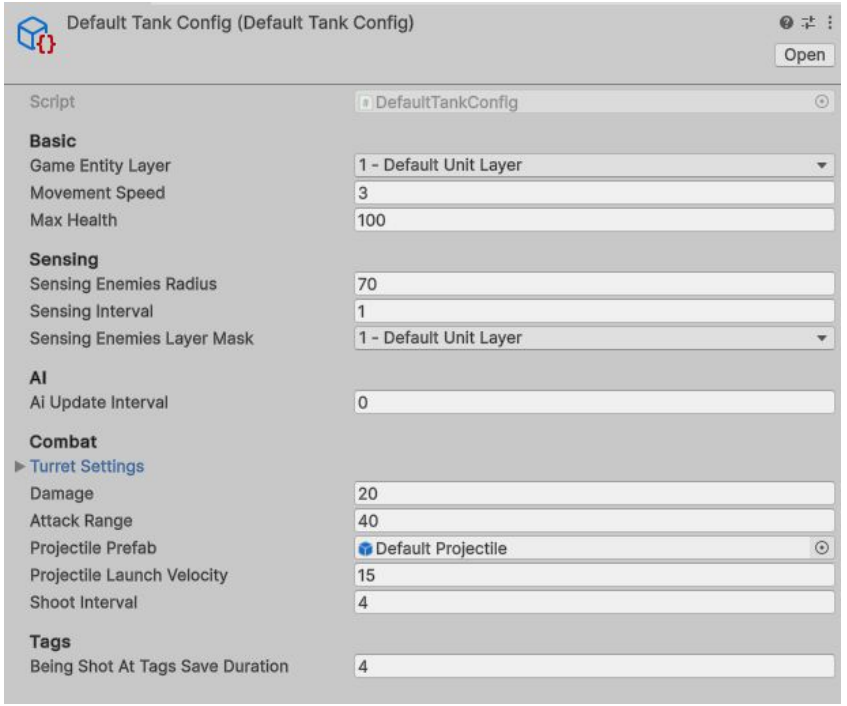



https://unity.com/blog/engine-platform/6-ways-scriptableobjects-can-benefit-your-team-and-your-code

# Scriptable Objects - as Data Container

- Example: Store stats for units of an RTS game like health, damage, etc…



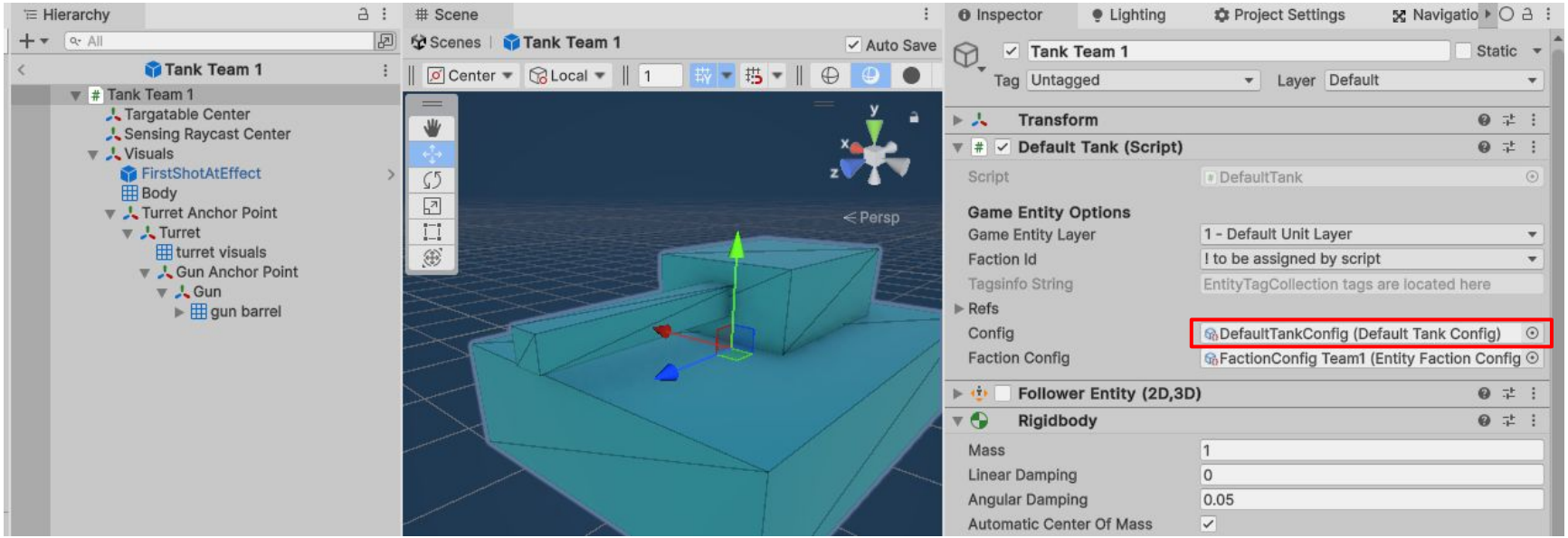
# SO as Data Container - Example Implementation

```
[CreateAssetMenu(fileName = "DefaultTankConfig", menuName = "Game Entities Sample/DefaultTankConfig", order = 1)]
Unity Script | 3 references
public class DefaultTankConfig : ScriptableObject
{
    [Header("Basic")]
    [EntityLayerDropdown]
    public int gameEntityLayer;

    public float movementSpeed;
    public float maxHealth;


    [Header("Sensing")]
    public float sensingEnemiesRadius;
    public float sensingInterval = 1;
    public EntityLayerMask sensingEnemiesLayerMask;

    [Header("AI")]
    public float aiUpdateInterval;

    [Header("Combat")]
    public TurretSettings turretSettings;

    public float damage;
    public float attackRange;
```

Referenced inside the Unit

```
Unity Script (1 asset reference) | 0 references
public class DefaultTank : GameEntity, IDamageable, IHasDebugInformation
{
    [SerializeField] DefaultTankConfig config;
```

# Scriptable Objects - as Data Container




Multi-edit with [Scriptable Object Table asset](#) (free)

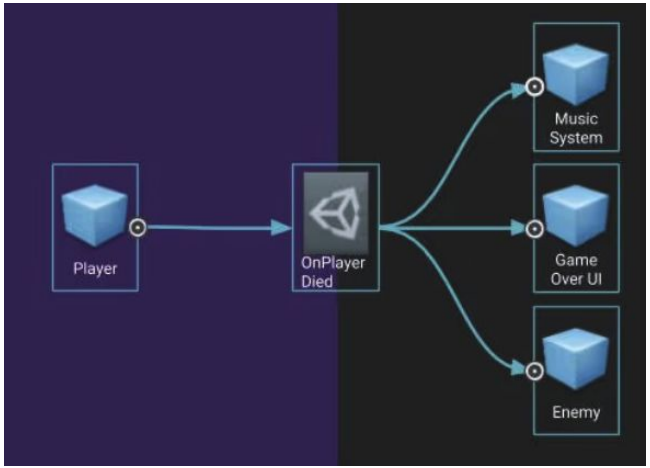
# Scriptable Objects - as Event Broadcaster

- Example: upon death, the player needs to notify multiple systems about the death
  - **UI System:** Change to game over UI
  - **Music System:** start playing defeat soundtrack
  - **NPC Manager:** enemies should stop chasing player


https://www.youtube.com/watch?v=W91QkppPpHI&ab_channel=RootGames

# Scriptable Objects - as Event Broadcaster

- Instead of referencing each other, all systems only reference the event ScriptableObject
- The player can invoke an event on the **OnPlayerDied** Scriptable Object to which other systems subscribe

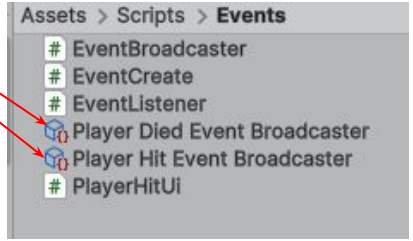



*https://unity.com/how-to/architect-game-code-scriptable-objects*

# Scriptable Objects - as Event Broadcaster

- Create an **EventBroadcaster** ScriptableObject, which can be referenced by listeners.
- The listeners register themselves to it
- Create **EventBroadcasters** for different Events like “**OnPlayerHit**” or “**OnPlayerDied**”

```
[CreateAssetMenu]
Unity Script | 1 reference
public class EventBroadcaster : ScriptableObject
{
    private List<EventListener> listeners = new List<EventListener>();

    1 reference
    public void RegisterListener(EventListener listener)
    {
        listeners.Add(listener);
    }

    1 reference
    public void DeregisterListener(EventListener listener)
    {
        listeners.Remove(listener);
    }

    0 references
    public void BroadcastEvent()
    {
        for (int i = 0; i < listeners.Count; i++)
        {
            listeners[i].RaiseEvent();
        }
    }
}
```

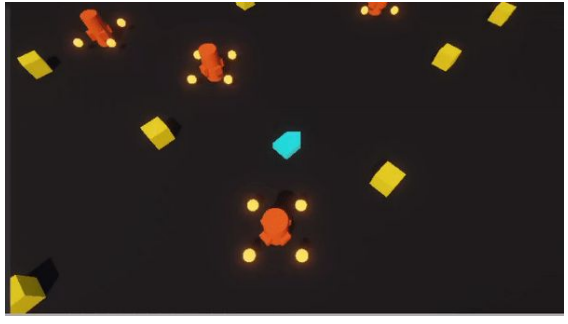



```
public class EventListener : MonoBehaviour
{
    [SerializeField] EventBroadcaster broadcaster;

    // Alternatively use UnityEvent class to assign
    // response methods inside the inspector
    public Action OnEventCalled;

    Unity Message | 0 references
    void OnEnable()
    {
        broadcaster.RegisterListener(this);
    }

    Unity Message | 0 references
    void OnDisable()
    {
        broadcaster.DeregisterListener(this);
    }

    1 reference
    public void RaiseEvent()
    {
        OnEventCalled.Invoke();
    }
}
```

# Player Hit Ui Effect Example

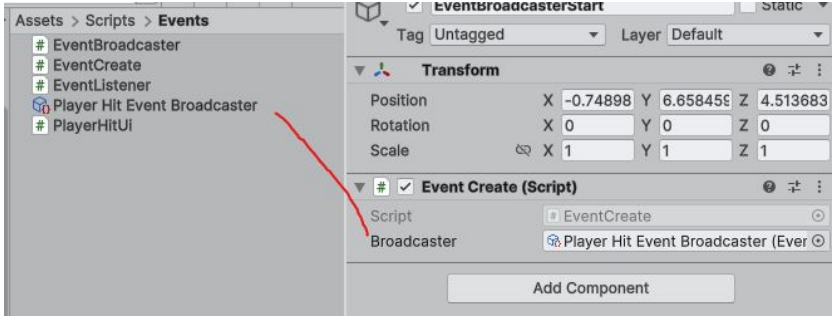
EventCreate calls the event

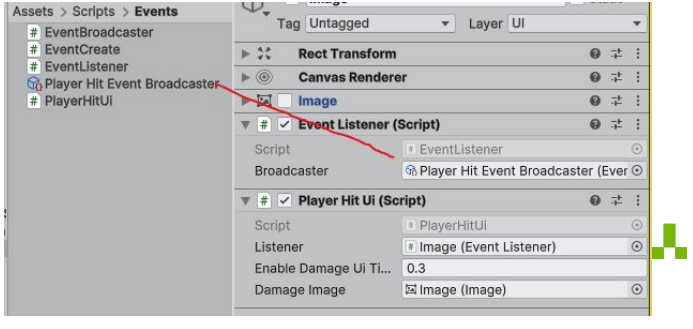
```
public class EventCreate : MonoBehaviour
{
    [SerializeField] EventBroadcaster broadcaster;

    Unity Message | 0 references
    void Update()
    {
        if(Input.GetKeyDown(KeyCode.Space))
        {
            broadcaster.BroadcastEvent();
        }
    }
}
```



```
public class PlayerHitUi : MonoBehaviour
{
    [SerializeField] EventListener listener;
    [SerializeField] float enableDamageUiTime;
    [SerializeField] Image damageImage;

    bool damageUiEnabled;
    float nextDisableUiTime;

    Unity Message | 0 references
    void Start()
    {
        listener.OnEventCalled += EnableDamageUi;
    }

    Unity Message | 0 references
    void Update()
    {
        if (damageUiEnabled)
        {
            if(Time.time > nextDisableUiTime)
            {
                DisableDamageUI();
            }
        }
    }

    1 reference
    void EnableDamageUi()
    {
        damageUiEnabled = true;
        nextDisableUiTime = Time.time + enableDamageUiTime;
        damageImage.enabled = true;
    }

    1 reference
    void DisableDamageUI()
    {
        damageUiEnabled = false;
        damageImage.enabled = false;
    }
}
```

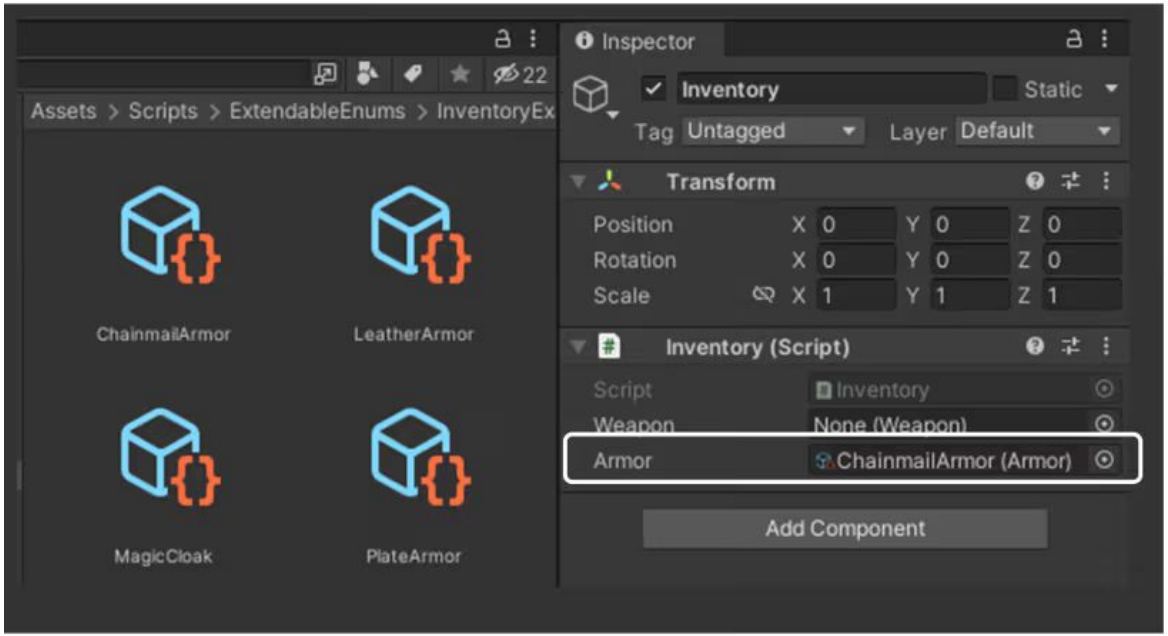
Player HitUi listens to the event



# Extended Enums

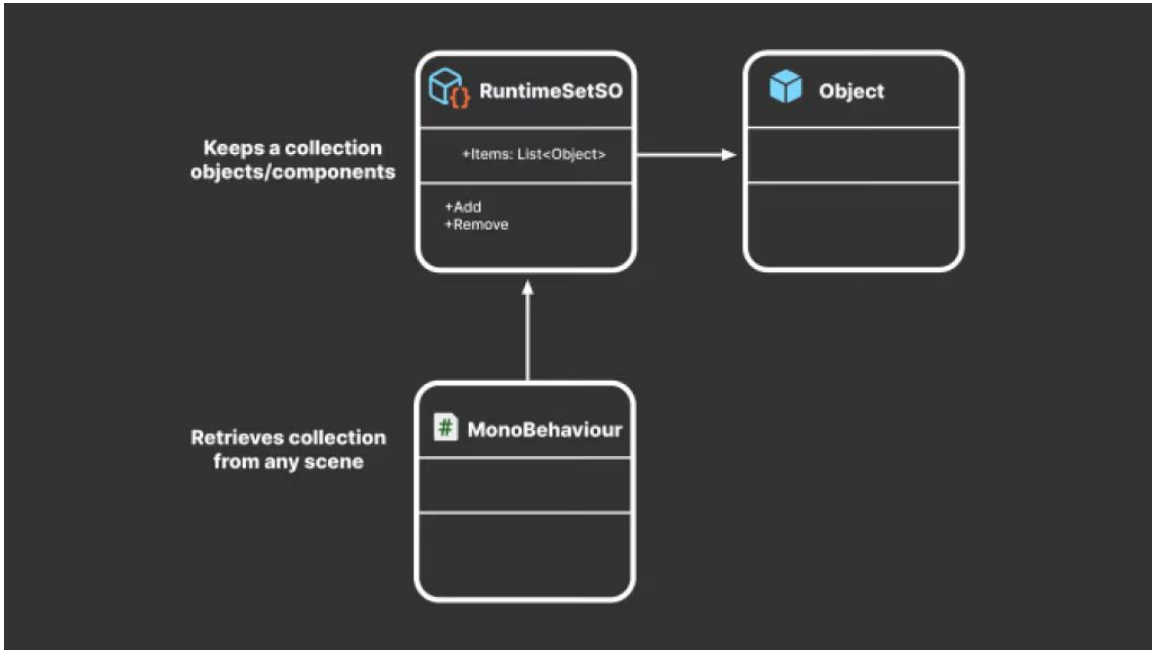
- Unlike normal enums, ScriptableObjects can have extra fields and methods
- While traditional enums have a fixed set of values, ScriptableObject enums can be created and modified at runtime




**Inventory slots categorized by ScriptableObject**

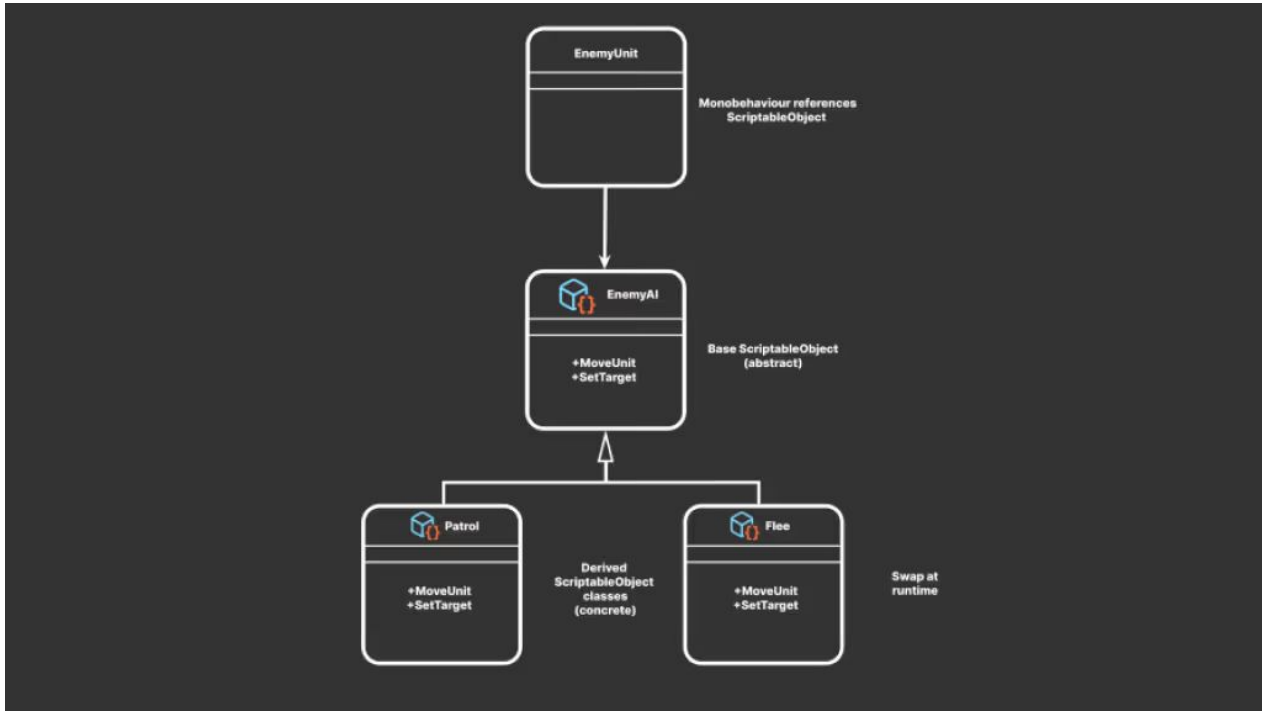
# Runtime Sets

- Dynamic lists that are used by multiple systems, can also be stored inside a Scriptable Object
- This Scriptable Object can then be referenced by objects

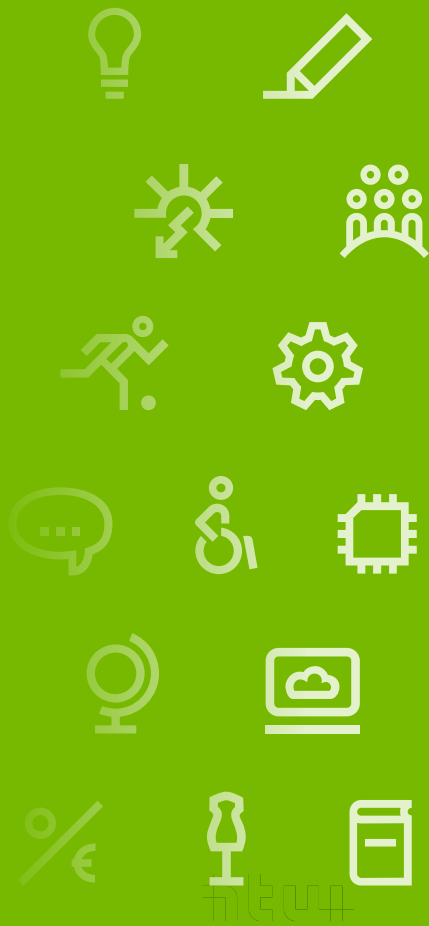


# Delegate Objects

- Moving logic from your MonoBehaviour into a Scriptable object can create more modular, pluggable Behaviour



# 7. JSON & Saving



# What is JSON?

- JavaScript Object Notation
- Lightweight format for storing and exchanging data.
- Human-readable and easy to parse
- Ideal for saving game states, player progress, settings, or data configurations.
- Platform-independent and works well with web APIs or external tools.


https://docs.kili-technology.com/docs/customizing-the-interface-through-json-settings

# Unity’s JsonUtility

- JsonUtility is built-in and highly optimized for Unity
- Converts between objects and JSON strings
- Limitations:
  - Doesn't support dictionaries
  - Only serializes public fields or [Serializable] classes
- Alternatively use something like Newtonsoft.Json or implement your own JSON serializer

| | |
|---|---|
| FromJson | Create an object from its JSON representation. |
| FromJsonOverwrite | Overwrite data in an object by reading from its JSON representation. |
| ToJson | Generate a JSON representation of the public fields of an object. |

https://docs.unity3d.com/ScriptReference/JsonUtility.html

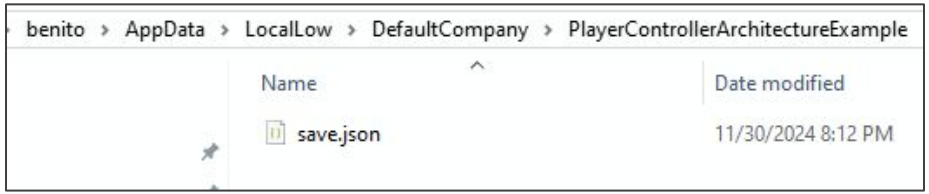
# Save & Load Example

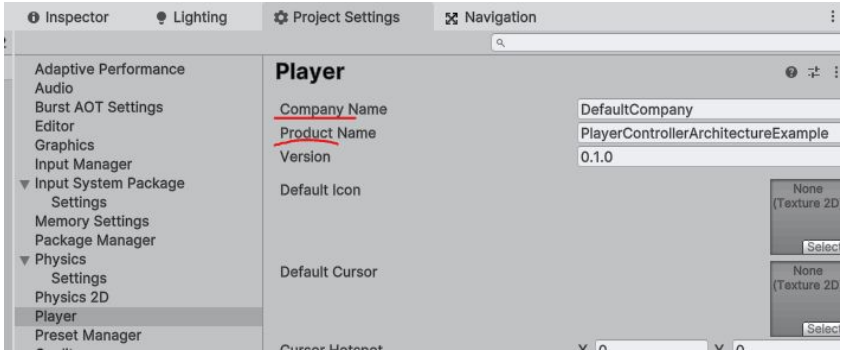
```
// Data container that will be saved,
// must be serialized
4 references
public class PlayerData
{
    public string playerName;
    public int level;
    public float health;
}
```

```
public class SaveLoadExample : MonoBehaviour
{
    private string saveFile;

    Unity Message | 0 references
    void Start()
    {
        saveFile = Application.persistentDataPath + "/save.json";
    }

    0 references
    public void SaveGame()
    {
        PlayerData data = new PlayerData
        {
            playerName = "Hero",
            level = 5,
            health = 75.5f
        };

        string json = JsonUtility.ToJson(data, true); // Pretty print
        System.IO.File.WriteAllText(saveFile, json);
        Debug.Log("Game Saved: " + json);
    }

    0 references
    public void LoadGame()
    {
        if (System.IO.File.Exists(saveFile))
        {
            string json = System.IO.File.ReadAllText(saveFile);
            PlayerData data = JsonUtility.FromJson<PlayerData>(json);
            Debug.Log("Game Loaded: " + data.playerName + ", Level: "
                + data.level + ", Health: " + data.health);
        }
        else
        {
            Debug.LogError("Save file not found!");
        }
    }
}
```

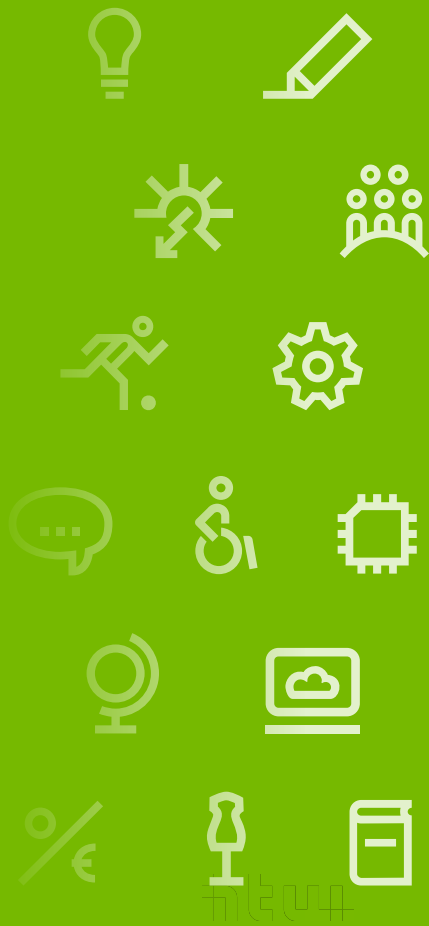
# Save Location - persistentDataPath

- Where is **Application.persistentDataPath** located?
- Depends on the platform
  - **Windows:** Appdata/LocalLow/CompanyName/GameName
  - Other platforms
- Company name and game name can be set inside **Project Settings**



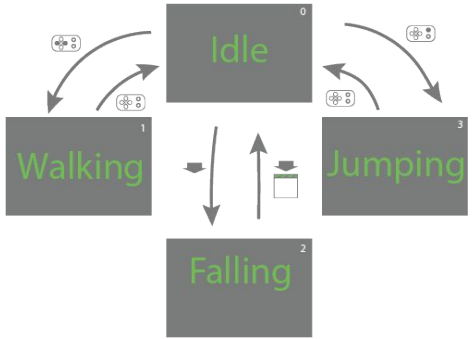



# 8. State Machines



# State Machine

- An architecture used to manage an entity's states and transitions.
- It defines how an object behaves depending on its current state.
- Can be used to manage game state, player controllers, enemy AI and more
- Improves maintainability and scalability by separating code into states




https://gamedev.stackexchange.com/questions/141737/finite-state-machine-for-player-states

# State Machine - Enum example

- Use an **Enum** in combination with a switch statement
- Useful for simple state machines like the **GameManager** for a casual game
- The simplest way to create a state machine
- Enums can be shown in the Unity Inspector for debug purposes


```
Unity Script | 0 references
public class GameManager : MonoBehaviour
{
    5 references
    public enum GameState
    {
        MainMenu,
        Playing,
        GameOver
    }

    public GameState currentState;

    Unity Message | 0 references
    private void Start()
    {
        currentState = GameState.MainMenu;
    }

    Unity Message | 0 references
    private void Update()
    {
        // Update the game based on the current state
        switch (currentState)
        {
            case GameState.MainMenu:
                //HandleMainMenu();
                break;
            case GameState.Playing:
                //HandlePlaying();
                break;
            case GameState.GameOver:
                //HandleGameOver();
                break;
        }
    }
}
```

# State Machine - classes

- Every state is a separate class.
- Every state defines what code is executed
  - OnStateEnter()
  - OnStateExit()
  - Every frame - UpdateState()
- A lot of work to set up, but offers great maintainability and scalability

```
1 reference
public interface IState
{
    1 reference
    public void OnStateEnter();

    1 reference
    public void OnStateExit();

    1 reference
    public void UpdateState();
}
```

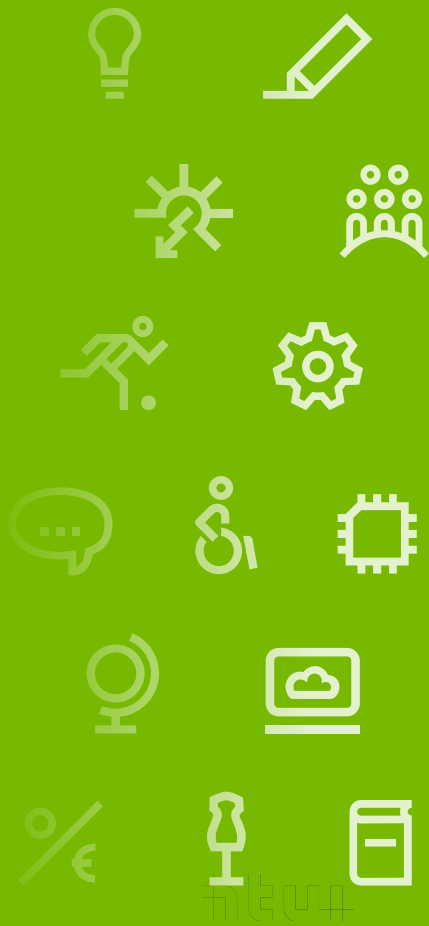
```
/// <summary>
/// The state machine only accepts T types that are
/// class objects and implement the IState interface
/// </summary>
0 references
public class StateMachine<T> where T : class, IState
{
    5 references
    public T CurrentState { get; protected set; }

    0 references
    public void SetState(T newState)
    {
        if (CurrentState == newState)
            return;

        CurrentState?.OnStateExit();
        CurrentState = newState;
        CurrentState?.OnStateEnter();
    }

    0 references
    public void Update()
    {
        CurrentState?.UpdateState();
    }
}
```

# State Machine - classes example use

- **State machine** and **states** need to be created inside script and set up
- **State machine** needs to be updated
- More in the **Player Controller Architecture Example** in the following slides

```
Unity Message | 0 references
void Start()
{
    // Set up movement statemachine
    state.movementStateMachine = new StateMachine<PlayerMovementState>();
    state.running = new MovementStates.Running();
    state.inAir = new MovementStates.InAir();

    state.running.SetUpState(refs,config,PlayerEventHooks,state);
    state.inAir.SetUpState(refs,config,PlayerEventHooks,state);

    // Running is the initial state
    state.movementStateMachine.SetState(state.running);
}

Unity Message | 0 references
void Update()
{
    UpdateInput();
    state.movementStateMachine.Update();
}

Unity Message | 0 references
private void FixedUpdate()
{
    state.movementStateMachine.FixedUpdate();
}
```

# 8.2 Example State Machine: Player Controller Architecture



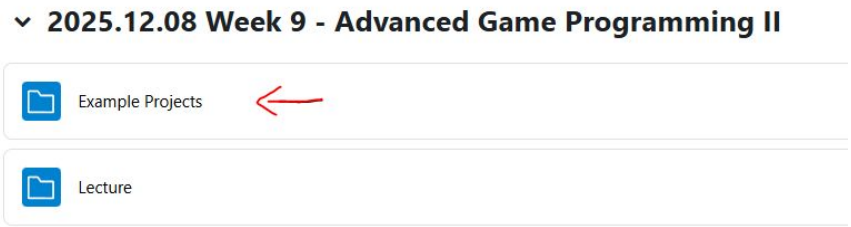
# Player Controller - Architecture

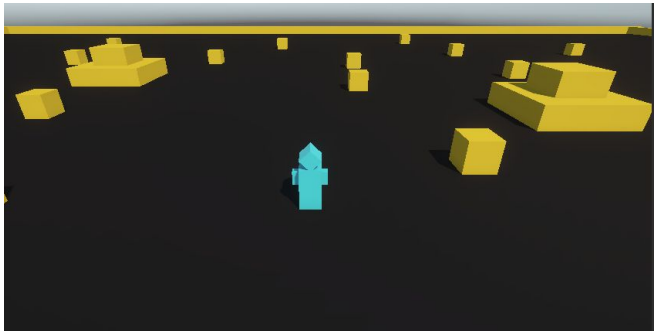
- Combines the following
  - Has **plain C# classes** as data containers that are shared among its systems
  - Uses **ScriptableObject** as a container for **configuration data**
  - Uses a **state machine** for **movement states**
  - Has **event callbacks**

- Initial set up is a lot of work, but it is worth it for more complicated player controllers in the long run
- Similar architecture can be used for NPC’s

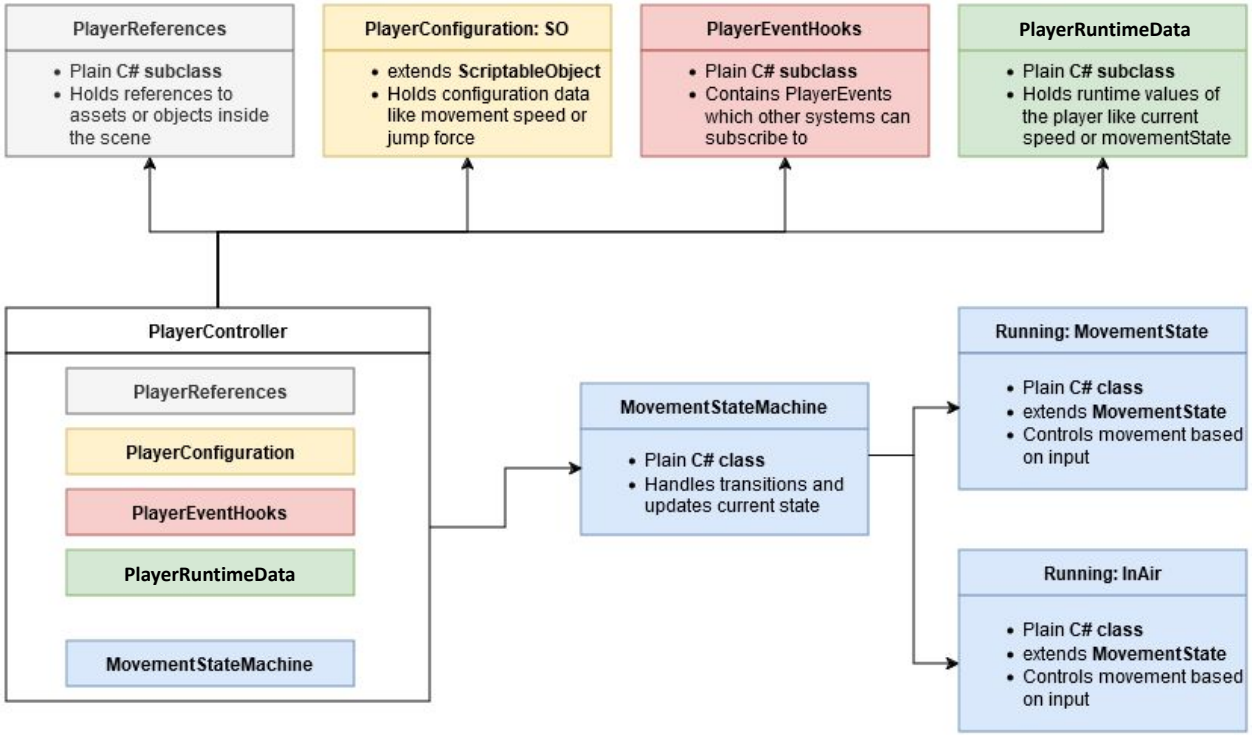
# Example project available in moodle

- Download the player controller example inside moodle



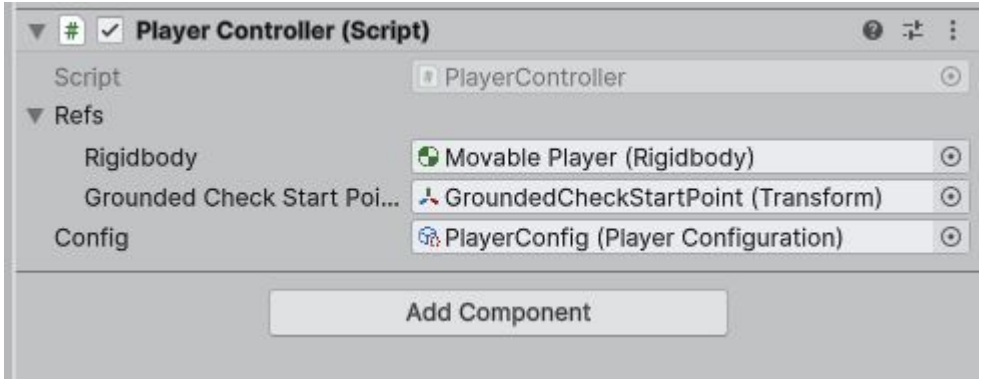
# Player Controller - Diagram



# Player References

**PlayerReferences**

- Plain C# **subclass**
- Holds references to assets or objects inside the scene




Reference inside PlayerController

```
Unity Script (1 asset reference) | 0 references
public class PlayerController : MonoBehaviour
{
    [SerializeField] PlayerReferences refs;
    [SerializeField] PlayerConfiguration config;
```

```
/// <summary>
/// Subclass for Player Controller
/// Holds References to other assets or objects in the
/// scene that the player needs
/// </summary>
[System.Serializable]
3 references
public class PlayerReferences
{
    public Rigidbody rigidbody;
    public Transform groundedCheckStartPoint;
}
```

# Player Config

**PlayerConfiguration: SO**

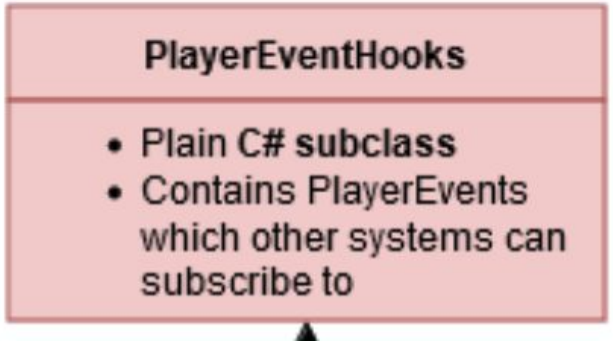
- extends **ScriptableObject**
- Holds configuration data like movement speed or jump force


Reference inside PlayerController

```
Unity Script (1 asset reference) | 0 references
public class PlayerController : MonoBehaviour
{
    [SerializeField] PlayerReferences refs;
    [SerializeField] PlayerConfiguration config;
```

```
/// <summary>
/// Data class, holds values for player controller like speed, various checks etc...
/// </summary>
[CreateAssetMenu(fileName = "PlayerConfig", menuName = "Player/PlayerConfig")]
Unity Script | 3 references
public class PlayerConfiguration : ScriptableObject
{
    [Header("Movement")]
    public float movementSpeed;
    public float inAirMovementSpeed;
    public float jumpForce;

    [Header("GroundedCheck")]
    public float groundedCheckOverlapSphereRadius;
    public LayerMask groundedCheckLayerMask;

}
```

# Player Events Hooks




```
/// <summary>
/// Allows other systems or player subs-systems
/// to hook themselves into player events.
/// Could also use UnityEvents or
/// ScriptableObject based events.
/// </summary>
4 references
public class PlayerEventHooks
{
    public Action OnPlayerGetDamage;
    public Action OnPlayerJump;
}
```

Inside Player Controller

```
3 references
public PlayerEventHooks PlayerEventHooks { get; private set; }

Unity Message | 0 references
void Start()
{
    PlayerEventHooks = new PlayerEventHooks();
```

# Player Runtime Data

**PlayerRuntimeData**

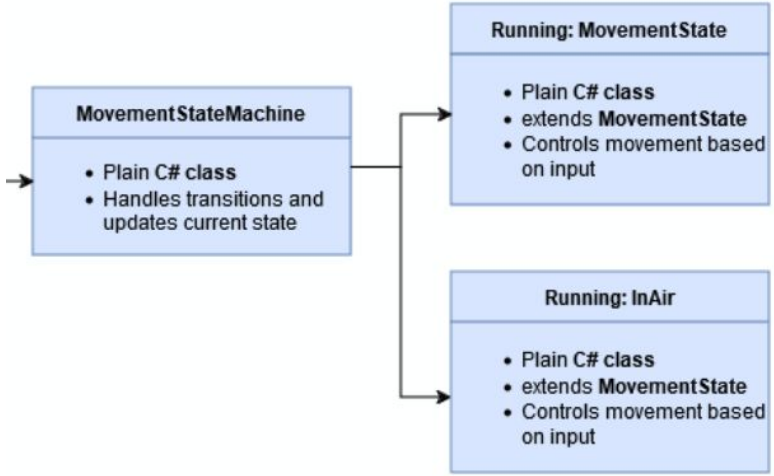
- Plain **C# subclass**
- Holds runtime values of the player like current speed or movementState

```
/// <summary>
/// Holds the global player state/ runtime data
/// Used and updated by all player subsystems like
/// movement statemachine, interaction statemachine etc
/// </summary>
4 references
public class PlayerRuntimeData
{
    // Player values
    public bool grounded;

    // Movement States
    public StateMachine<PlayerMovementState> movementStateMachine;

    public MovementStates.Running running;
    public MovementStates.InAir inAir;

    // Input
    public bool jumpInput;
    public Vector2 moveInput;
}
```

Initialize inside PlayerController

```
PlayerRuntimeData runtimeData;

Unity Message | 0 references
void Start()
{
    runtimeData = new PlayerRuntimeData();
```

# Player Controller - MovementStateMachine




Initialize inside PlayerController

```
Unity Message | 0 references
void Start()
{
    // Set up movement statemachine
    state.movementStateMachine = new StateMachine<PlayerMovementState>();
    state.running = new MovementStates.Running();
    state.inAir = new MovementStates.InAir();

    state.running.SetUpState(refs,config,PlayerEventHooks,state);
    state.inAir.SetUpState(refs,config,PlayerEventHooks,state);

    // Running is the initial state
    state.movementStateMachine.SetState(state.running);
}
```

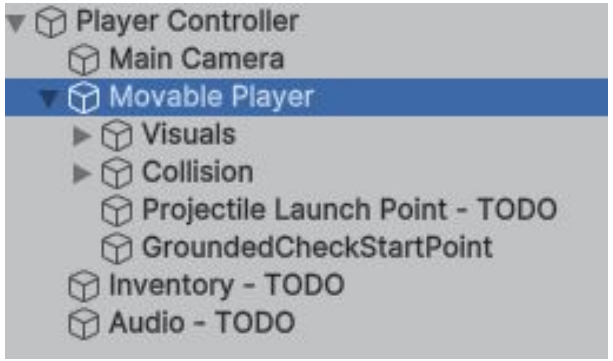
# Player Controller - States

```
public class InAir : PlayerMovementState
{
    bool performedDoubleJump;

    3 references
    public override void OnStateEnter()
    {
        performedDoubleJump = false;
        Debug.Log("Entering InAir State");
    }
    3 references
    public override void OnStateExit()
    {
        Debug.Log("Exiting InAir State");
    }

    3 references
    public override void UpdateState()
    {
        HandleDoubleJump();
    }

    3 references
    public override void FixedUpdateState()
    {
        CheckGrounded();
        if (state.grounded)
        {
            state.movementStateMachine.SetState(state.running);
            return;
        }

        DefaultRbMove(config.inAirMovementSpeed);

    }

    1 reference
    void HandleDoubleJump()
    {
        if (state.jumpInput && !performedDoubleJump)
        {
            // cut all vertical velocity before jump for better movement
            refs.rigidbody.linearVelocity = new Vector3(
                refs.rigidbody.linearVelocity.x, 0f, refs.rigidbody.linearVelocity.z);

            performedDoubleJump = true;
            refs.rigidbody.AddForce(Vector3.up * config.jumpForce, ForceMode.Impulse);
            state.jumpInput = false;
        }
    }
}
```

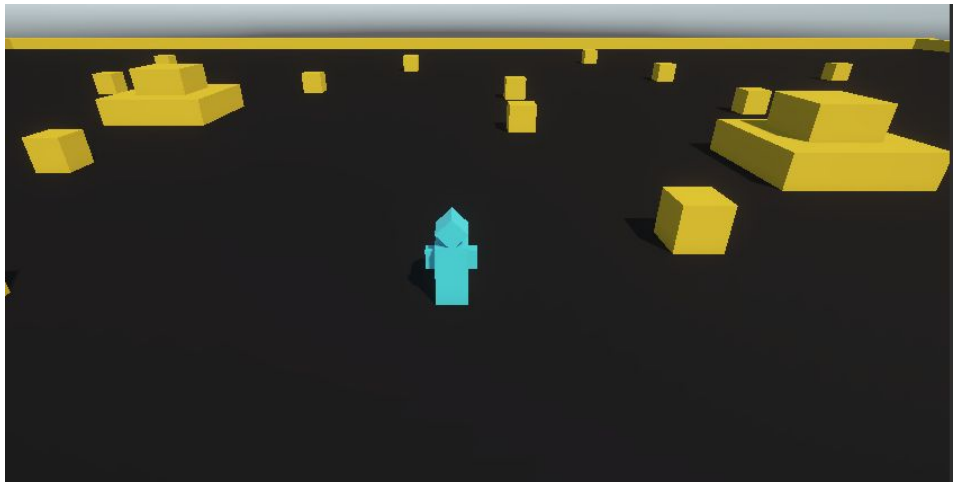
```
public class Running : PlayerMovementState
{
    3 references
    public override void OnStateEnter()
    {
        Debug.Log("Entering Running State");
    }
    3 references
    public override void OnStateExit()
    {
        Debug.Log("Exiting Running State");
    }

    3 references
    public override void UpdateState()
    {
        HandleJump();
    }

    3 references
    public override void FixedUpdateState()
    {
        CheckGrounded();
        if (!state.grounded)
        {
            state.movementStateMachine.SetState(state.inAir);
            return;
        }

        DefaultRbMove(config.movementSpeed);

    }

    1 reference
    void HandleJump()
    {
        if (state.jumpInput)
        {
            // cut all vertical velocity before jump for better movement
            refs.rigidbody.linearVelocity = new Vector3(
                refs.rigidbody.linearVelocity.x, 0f, refs.rigidbody.linearVelocity.z);

            refs.rigidbody.AddForce(Vector3.up * config.jumpForce, ForceMode.Impulse);
            state.jumpInput = false;
        }
    }
}
```

# Player Controller - Hierarchy

- Consider creating a lot of empty objects instead of putting all components on one game object
- Consider separating colliders and visuals
  - Not always feasible, but can make sense if the collider is much simpler
- Separate moving parts of your player from other systems like inventory etc… that don't need to follow the player's position
- Camera can be the child of player controller, but should not be the child of the player's moving part



# Player Controller - Live Presentation



# Thank you!