# Sprint 01

# Assignment 04 : Inventory Part II

# Part I

## Step 01: Setup

- As we have setup a Pickup System in the previous assignment, in this one we will create a common descriptive system for each Pickup Item along with Projectiles for each type of Ammo & Weapons.
- In total, we complete this assignment in 4 steps:
  - Common Descriptive System (IEquipable & IEquipableMain )
  - Weapon Scripts
  - Projectiles
  - Inventory Menu Screen
- So, start by creating two common descriptive script called "*IEquipable*" & "*IEquipableMain*".
- Then we attach these to the weapons(Gun & Mele) scripts by creating their instances in other scripts.
- Then for the Gun item we setup a Projectile System which detects the "Ammo Type" and fires the projectile.
- Finally, after all that, we create the Inventory Menu i.e., the Visual representation of the Menu, from where we can Equip or Drop an item.
- So first, we create the IEquipable & IEquipableMain scripts.

## Step 02: Script & Workflow

- The Scripts workflow for EquipmentController is like this:
  - LevelController > PlayerController > PlayerEquipmentController > IEquipable & IEquipableMain > All Pickups (Ammos, Supplies & Weapons).

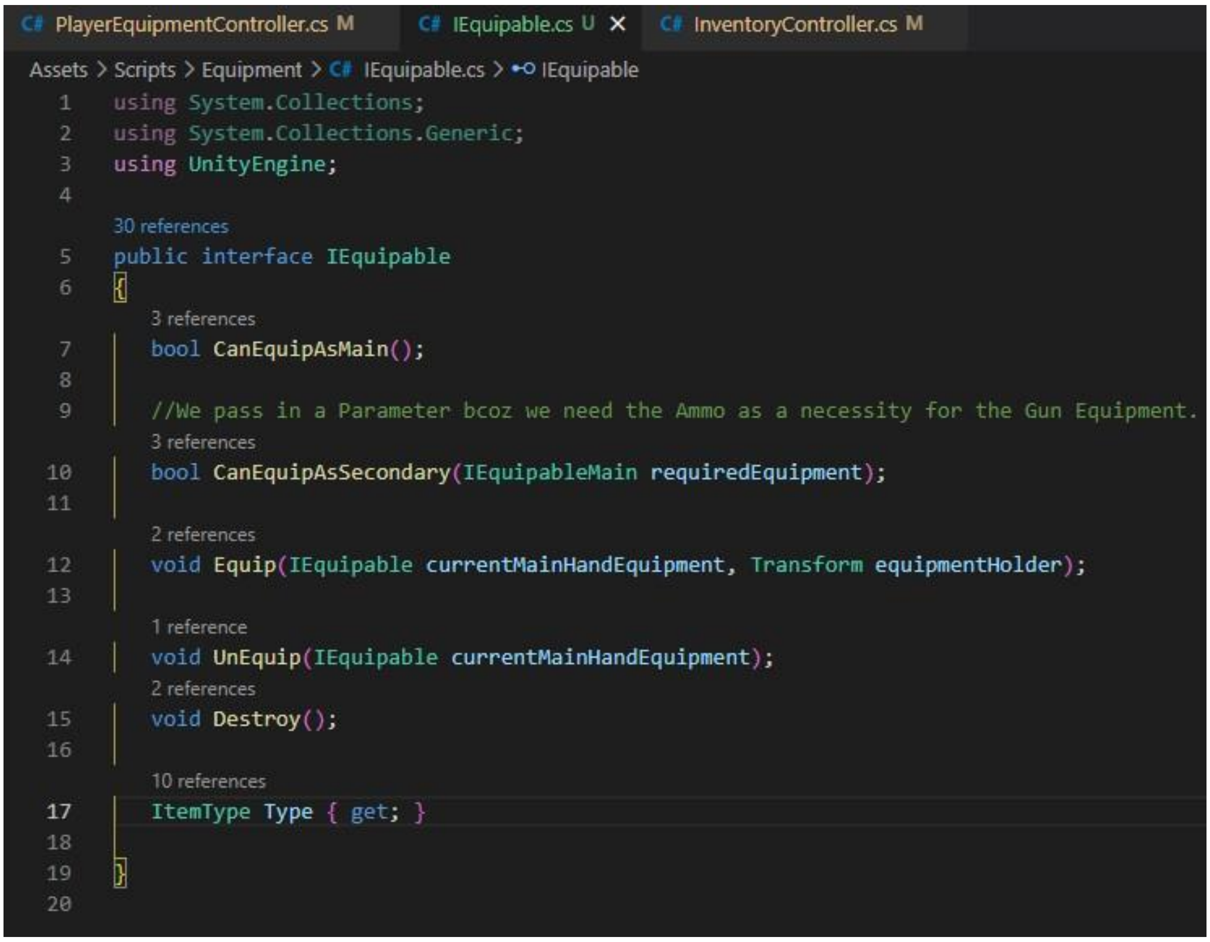
- The "*PlayerEquipmentController*" acts as the Main Centre Script to which all the other scripts are attached.
- IEquipable & IEquipableMain Interfaces are used in every Pickup items.
    - PlayerEquipmentController > IEquipable & IEquipableMain > Gun & Sword.
    - Gun > IGun > DamageAmmo, ExplosiveAmmo, GuardAmmo, Supplies, PlayerObjectData & ProjectilePool.
    - Sword > SwordAttackAnimation, PlayerObjectData & CollisionCallbacks.

- Next, the Script Workflow for Projectiles:
    - LevelController > LevelDependecies > ProjectileLibrary > ProjectileType > ProjectileManager > IProjectile > ProjectileFactory > ProjectilePool.

- Only ProjectileLibrary is a MonoBehavior script which is attached to the "*Dependencies*" GameObject in the scene.
- To this all Missiles or Ammos Prefabs are attached.
- All the ammos (DamageAmmo, ExplosiveAmmo, etc.) are generated in the equipment factory and fired using the other respective Projectile scripts.

❖ **IEquipable.**

- It is the first script which is also used by the IEquipableMain interface.
- It is also acts as a descriptive class for Secondary Equipable item.
- It consists of some descriptive variables that need to be set when accessing or implementing this script in other scripts compulsorily.

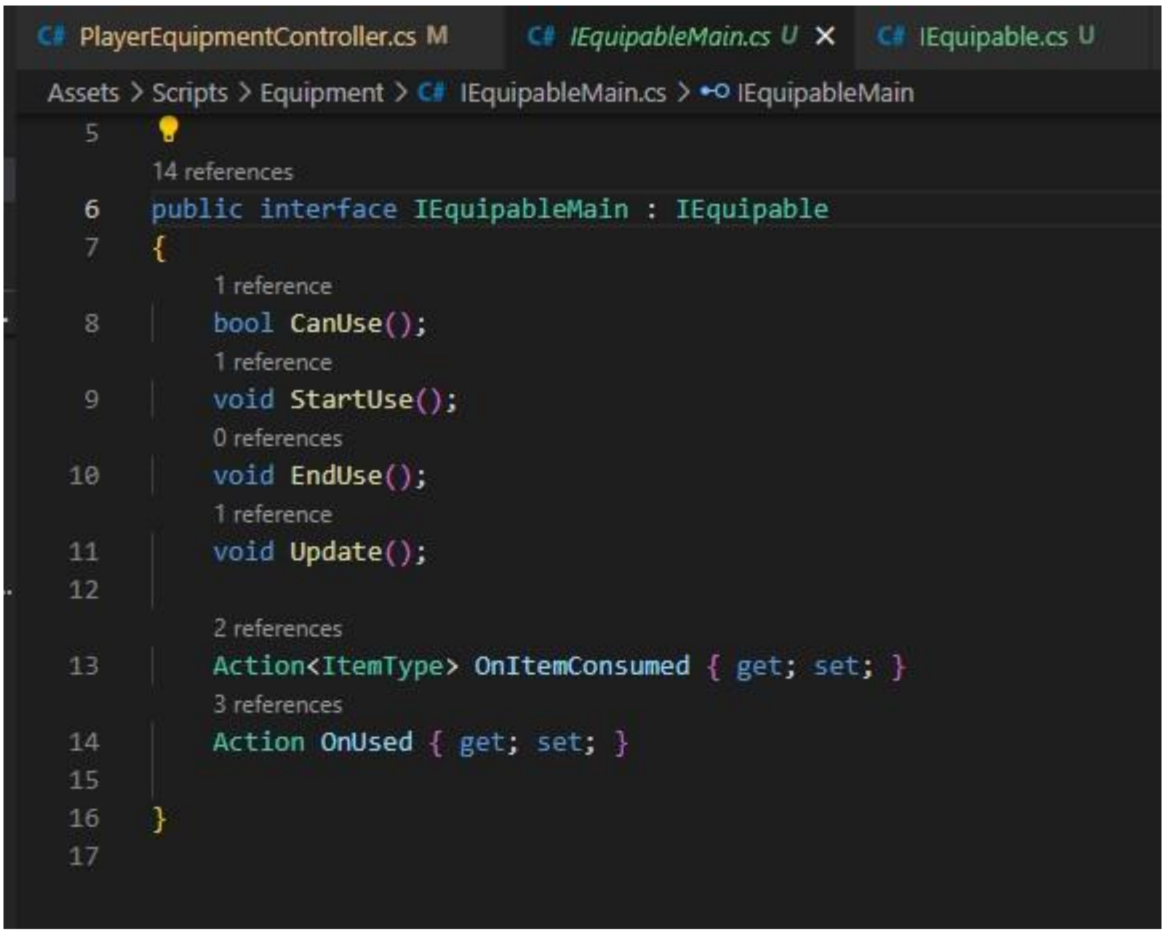


PlayerEquipmentController.cs M
IEquipable.cs U
InventoryController.cs M
Assets > Scripts > Equipment > IEquipable.cs > IEquipable

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

30 references
public interface IEquipable
{
    3 references
    bool CanEquipAsMain();

    //We pass in a Parameter bcoz we need the Ammo as a necessity for the Gun Equipment.
    3 references
    bool CanEquipAsSecondary(IEquipableMain requiredEquipment);

    2 references
    void Equip(IEquipable currentMainHandEquipment, Transform equipmentHolder);

    1 reference
    void UnEquip(IEquipable currentMainHandEquipment);
    2 references
    void Destroy();

    10 references
    ItemType Type { get; }

}
```

❖ **IEquipableMain.**

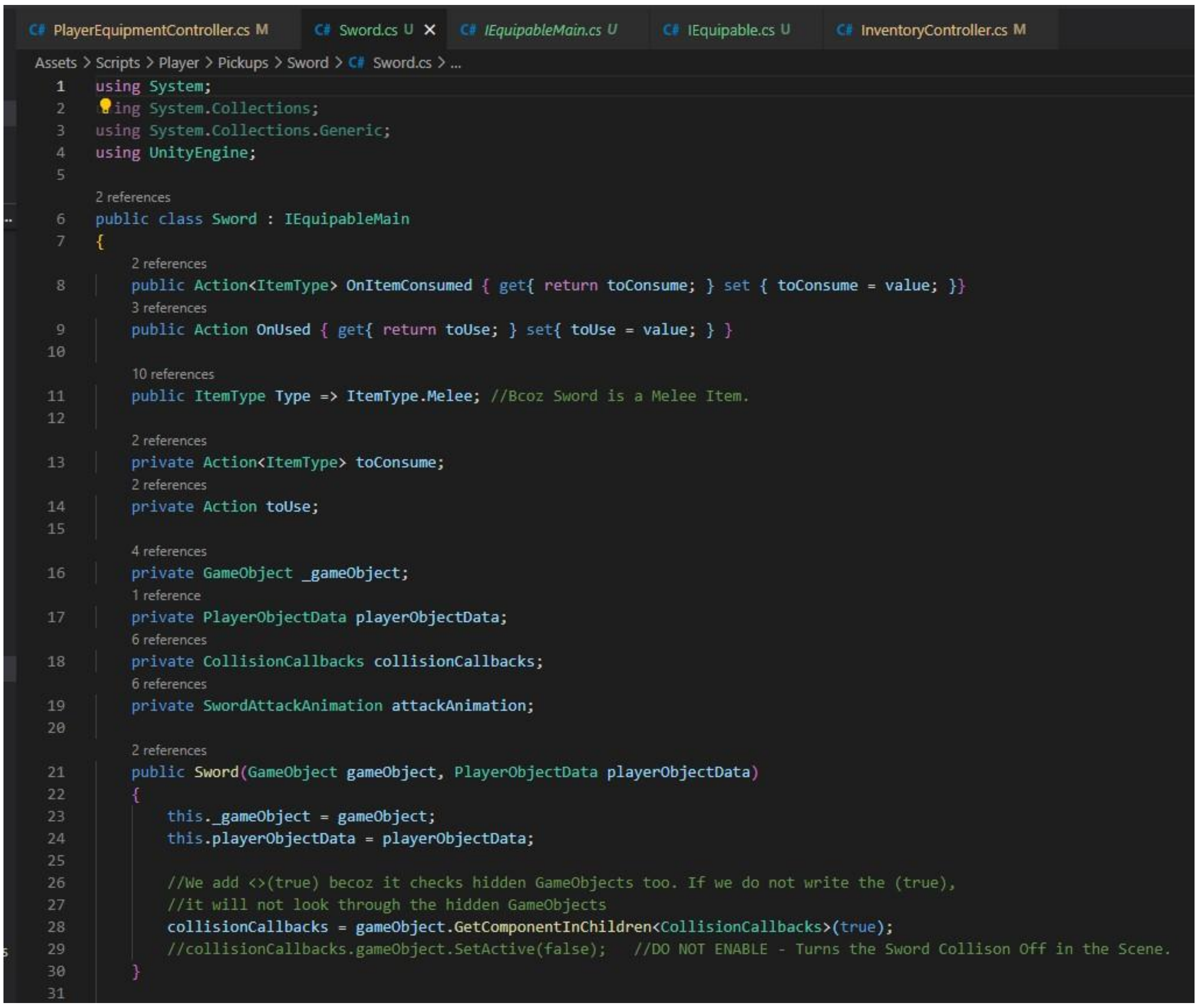
- It is the second script which uses the IEquipable interface.
- It also has its own descriptive variables.
- It is used for the Primary or Main Equipment.


PlayerEquipmentController.cs M
IEquipableMain.cs U
IEquipable.cs U
Assets > Scripts > Equipment > IEquipableMain.cs > IEquipableMain

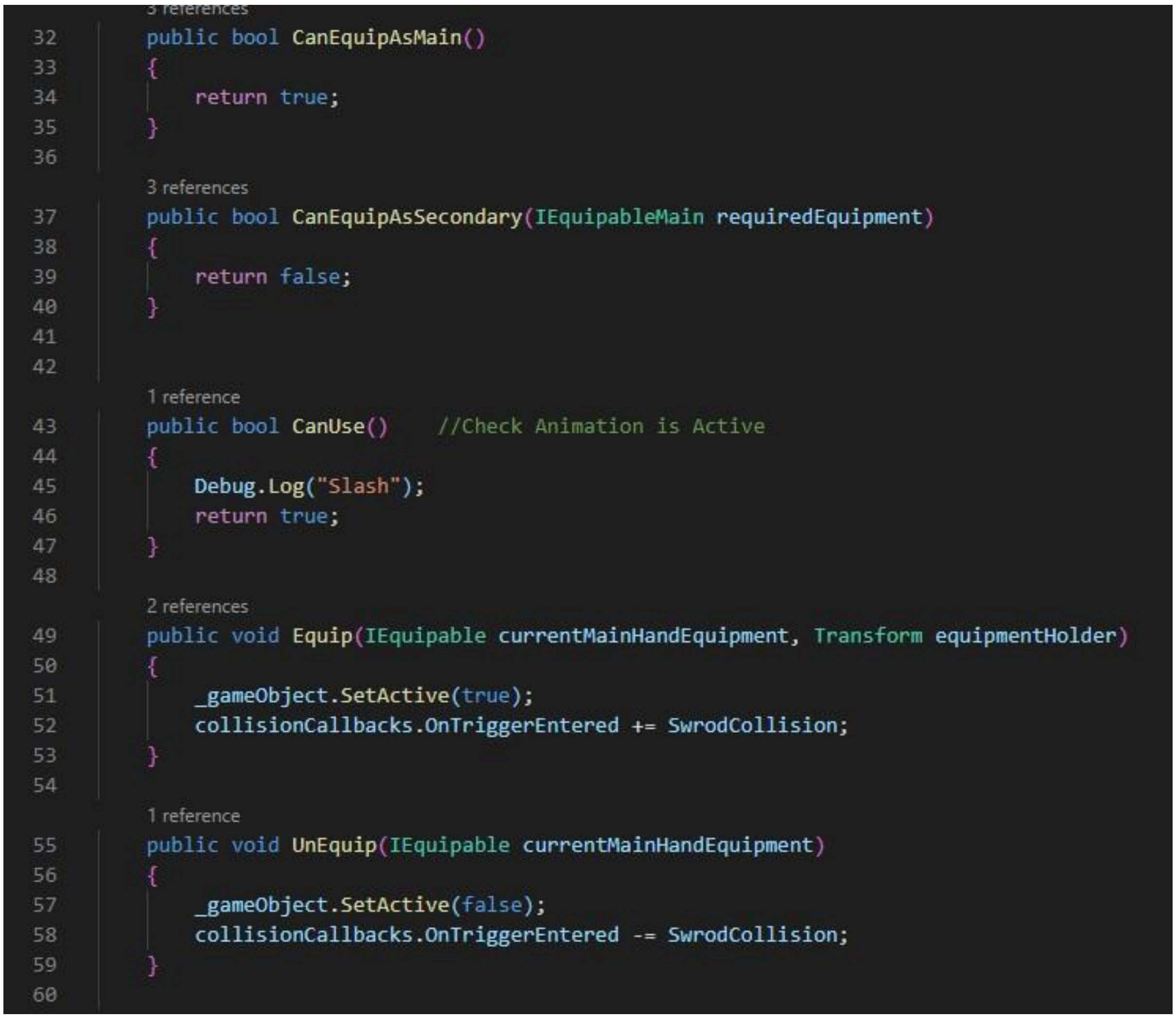
```
14 references
public interface IEquipableMain : IEquipable
{
    1 reference
    bool CanUse();
    1 reference
    void StartUse();
    0 references
    void EndUse();
    1 reference
    void Update();

    2 references
    Action<ItemType> OnItemConsumed { get; set; }
    3 references
    Action OnUsed { get; set; }

}
```

❖ **Sword.**

- This is a Main Weapon Equipment.
- This script simply is responsible for disabling the Sword GameObject when it is picked up, and enabling the actual Sword that is present on the Player GameObject.
- It also checks for Collisions and plays the Sword Animation.

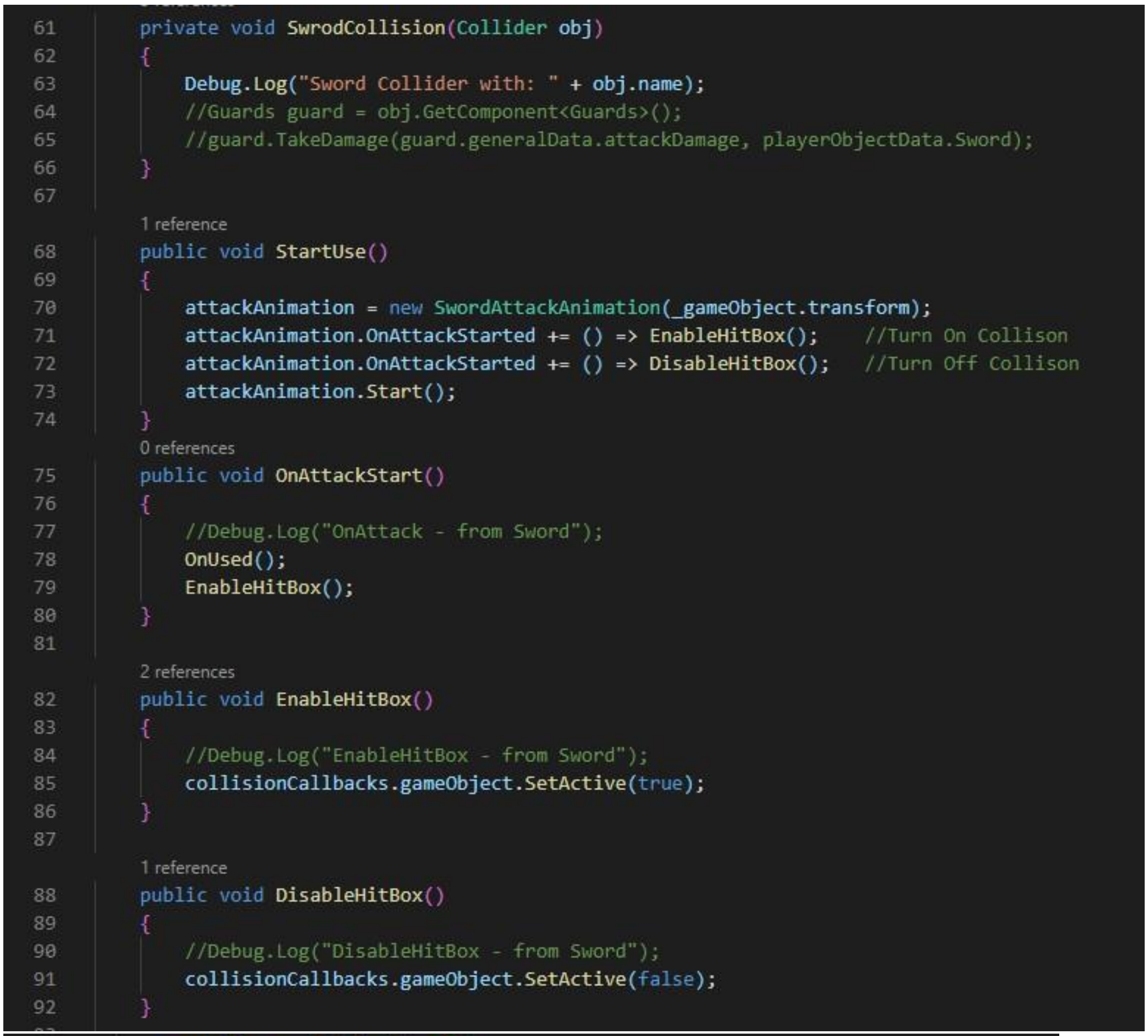

PlayerEquipmentController.cs M
Sword.cs U
IEquipableMain.cs U
IEquipable.cs U
InventoryController.cs M
Assets > Scripts > Player > Pickups > Sword > Sword.cs > ...

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

2 references
public class Sword : IEquipableMain
{
    2 references
    public Action<ItemType> OnItemConsumed { get{ return toConsume; } set { toConsume = value; }}
    3 references
    public Action OnUsed { get{ return toUse; } set{ toUse = value; } }

    10 references
    public ItemType Type => ItemType.Melee; //Bcoz Sword is a Melee Item.

    2 references
    private Action<ItemType> toConsume;
    2 references
    private Action toUse;

    4 references
    private GameObject _gameObject;
    1 reference
    private PlayerObjectData playerObjectData;
    6 references
    private CollisionCallbacks collisionCallbacks;
    6 references
    private SwordAttackAnimation attackAnimation;

    2 references
    public Sword(GameObject gameObject, PlayerObjectData playerObjectData)
    {
        this._gameObject = gameObject;
        this.playerObjectData = playerObjectData;

        //We add <>(true) becoz it checks hidden GameObjects too. If we do not write the (true),
        //it will not look through the hidden GameObjects
        collisionCallbacks = gameObject.GetComponentInChildren<CollisionCallbacks>(true);
        //collisionCallbacks.gameObject.SetActive(false);   //DO NOT ENABLE - Turns the Sword Collison Off in the Scene.
    }

```

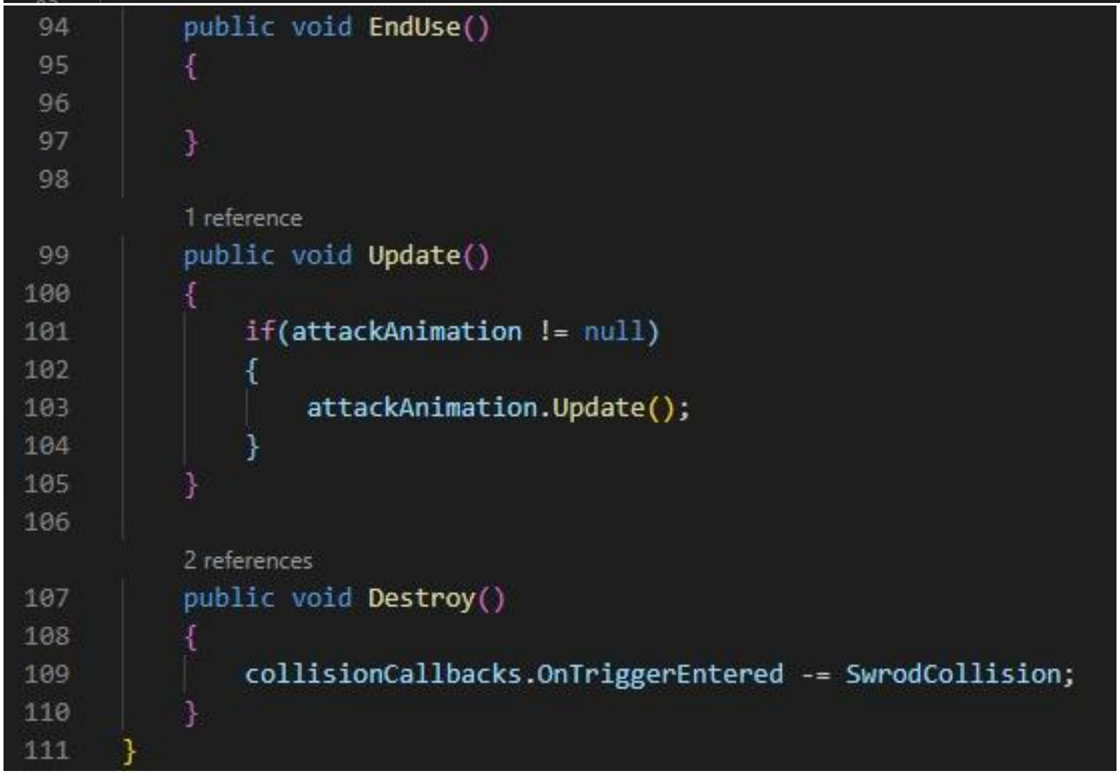


```
    3 references
    public bool CanEquipAsMain()
    {
        return true;
    }

    3 references
    public bool CanEquipAsSecondary(IEquipableMain requiredEquipment)
    {
        return false;
    }


    1 reference
    public bool CanUse()    //Check Animation is Active
    {
        Debug.Log("Slash");
        return true;
    }

    2 references
    public void Equip(IEquipable currentMainHandEquipment, Transform equipmentHolder)
    {
        _gameObject.SetActive(true);
        collisionCallbacks.OnTriggerEntered += SwrodCollision;
    }

    1 reference
    public void UnEquip(IEquipable currentMainHandEquipment)
    {
        _gameObject.SetActive(false);
        collisionCallbacks.OnTriggerEntered -= SwrodCollision;
    }

```

- We get the actual Sword GameObject form the PlayerObjectData, and enable it when we pick up the Sword that is present in the scene as a Pickable object.
- We then disable that picked up object and enable the Sword form PlayerObjectData.
- We also check for the its collisions with the environment, by accessing the respective Actions set in the “*CollisionCallBacks*” script.



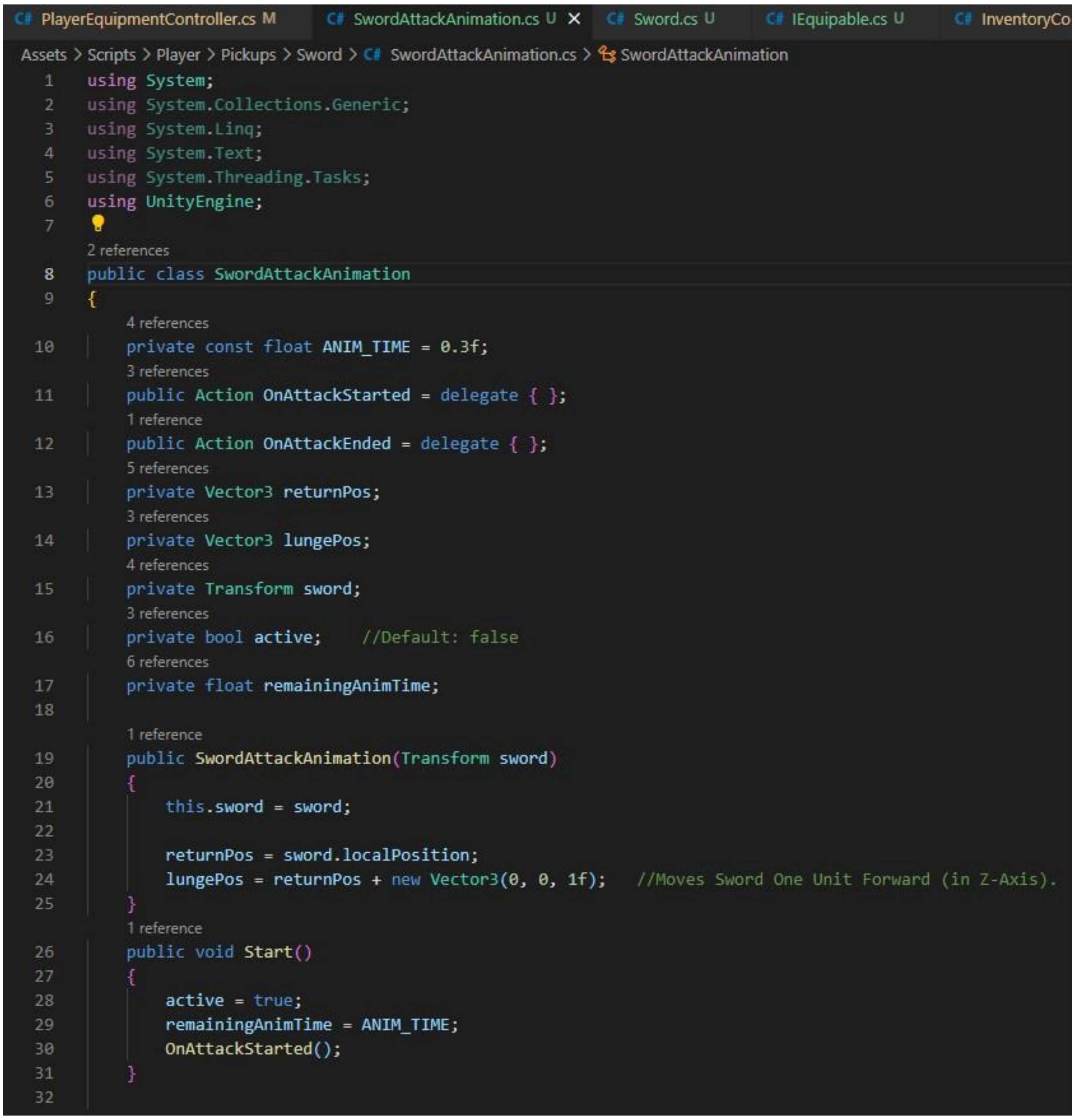
```
    private void SwrodCollision(Collider obj)
    {
        Debug.Log("Sword Collider with: " + obj.name);
        //Guards guard = obj.GetComponent<Guards>();
        //guard.TakeDamage(guard.generalData.attackDamage, playerObjectData.Sword);
    }

    1 reference
    public void StartUse()
    {
        attackAnimation = new SwordAttackAnimation(_gameObject.transform);
        attackAnimation.OnAttackStarted += () => EnableHitBox();    //Turn On Collison
        attackAnimation.OnAttackStarted += () => DisableHitBox();   //Turn Off Collison
        attackAnimation.Start();
    }
    0 references
    public void OnAttackStart()
    {
        //Debug.Log("OnAttack - from Sword");
        OnUsed();
        EnableHitBox();
    }

    2 references
    public void EnableHitBox()
    {
        //Debug.Log("EnableHitBox - from Sword");
        collisionCallbacks.gameObject.SetActive(true);
    }

    1 reference
    public void DisableHitBox()
    {
        //Debug.Log("DisableHitBox - from Sword");
        collisionCallbacks.gameObject.SetActive(false);
    }
```



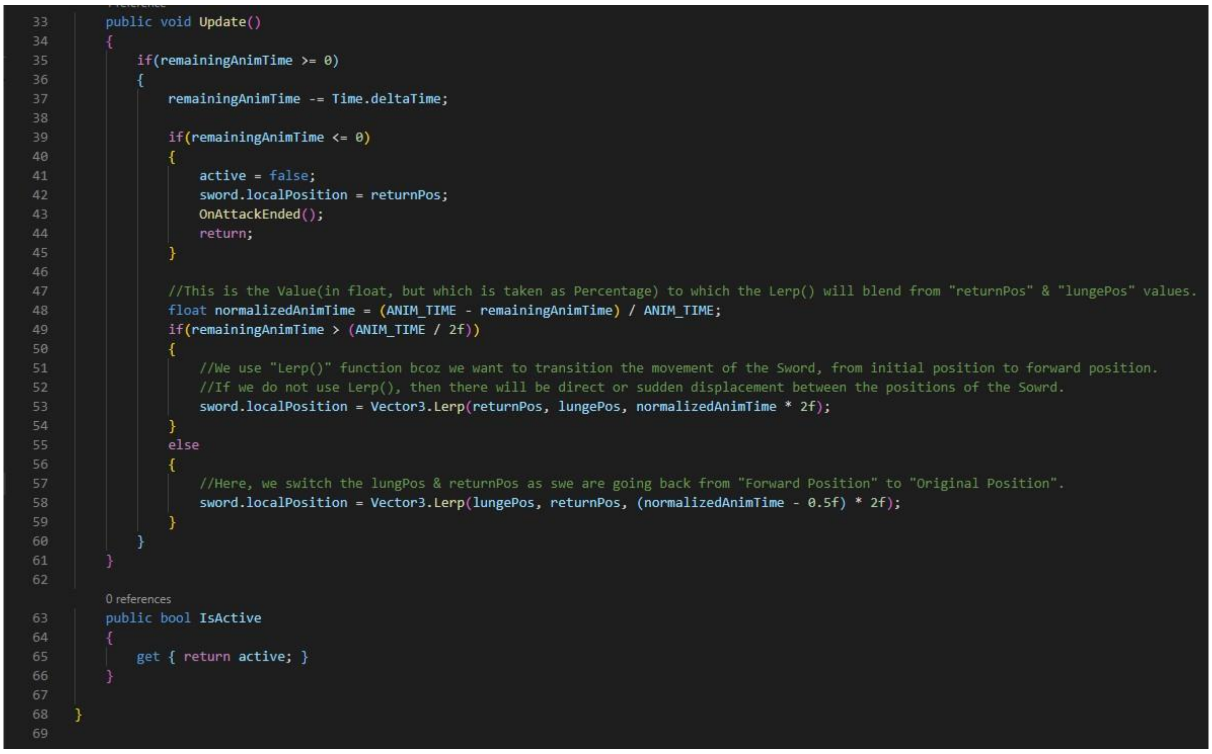
```
    public void EndUse()
    {

    }

    1 reference
    public void Update()
    {
        if(attackAnimation != null)
        {
            attackAnimation.Update();
        }
    }

    2 references
    public void Destroy()
    {
        collisionCallbacks.OnTriggerEntered -= SwrodCollision;
    }
}
```

❖ **SwordAttackAnimation.**

- This script is responsible for the Sword Animation.
- The Sword Attack Animation is done, by moving the Sword a step forward along its Z-Axis, so as to give the feel of attacking.
- We do not use any AnimationController or AnimationClip for the animation.


PlayerEquipmentController.cs M
SwordAttackAnimation.cs U
Sword.cs U
IEquipable.cs U
InventoryCo
Assets > Scripts > Player > Pickups > Sword > SwordAttackAnimation.cs > SwordAttackAnimation

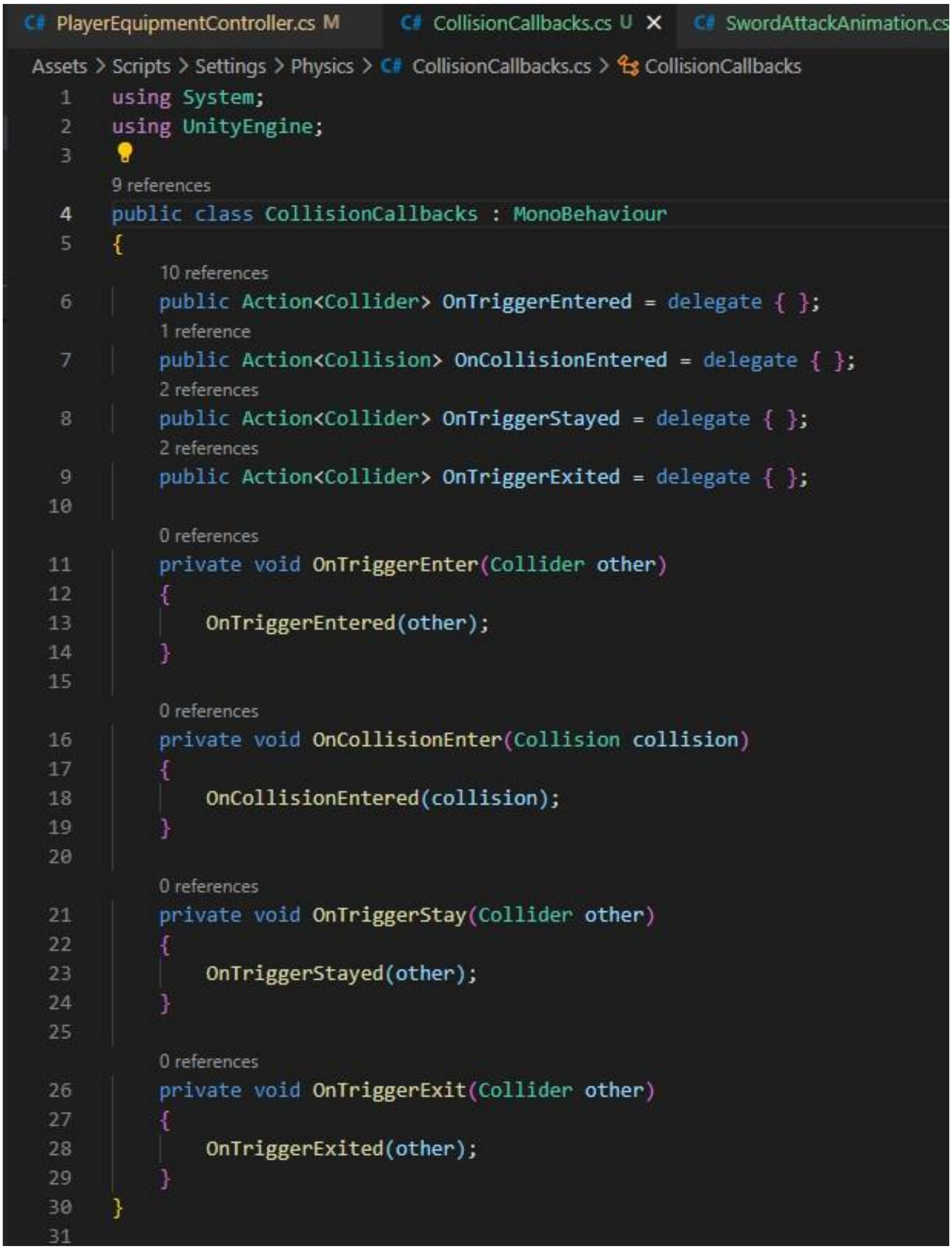
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using UnityEngine;

2 references
public class SwordAttackAnimation
{
    4 references
    private const float ANIM_TIME = 0.3f;
    3 references
    public Action OnAttackStarted = delegate { };
    1 reference
    public Action OnAttackEnded = delegate { };
    5 references
    private Vector3 returnPos;
    3 references
    private Vector3 lungePos;
    4 references
    private Transform sword;
    3 references
    private bool active;    //Default: false
    6 references
    private float remainingAnimTime;

    1 reference
    public SwordAttackAnimation(Transform sword)
    {
        this.sword = sword;

        returnPos = sword.localPosition;
        lungePos = returnPos + new Vector3(0, 0, 1f);   //Moves Sword One Unit Forward (in Z-Axis).
    }
    1 reference
    public void Start()
    {
        active = true;
        remainingAnimTime = ANIM_TIME;
        OnAttackStarted();
    }

```



```csharp
    public void Update()
    {
        if(remainingAnimTime >= 0)
        {
            remainingAnimTime -= Time.deltaTime;

            if(remainingAnimTime <= 0)
            {
                active = false;
                sword.localPosition = returnPos;
                OnAttackEnded();
                return;
            }

            //This is the Value(in float, but which is taken as Percentage) to which the Lerp() will blend from "returnPos" & "lungePos" values.
            float normalizedAnimTime = (ANIM_TIME - remainingAnimTime) / ANIM_TIME;
            if(remainingAnimTime > (ANIM_TIME / 2f))
            {
                //We use "Lerp()" function bcoz we want to transition the movement of the Sword, from initial position to forward position.
                //If we do not use Lerp(), then there will be direct or sudden displacement between the positions of the Sowrd.
                sword.localPosition = Vector3.Lerp(returnPos, lungePos, normalizedAnimTime * 2f);
            }
            else
            {
                //Here, we switch the lungPos & returnPos as swe are going back from "Forward Position" to "Original Position".
                sword.localPosition = Vector3.Lerp(lungePos, returnPos, (normalizedAnimTime - 0.5f) * 2f);
            }
        }
    }

    0 references
    public bool IsActive
    {
        get { return active; }
    }

}
```

❖ **CollisionCallBacks.**

- It basically consists of Action Events which are triggered when the respective GameObejct enters a collision event or collides in the scene.

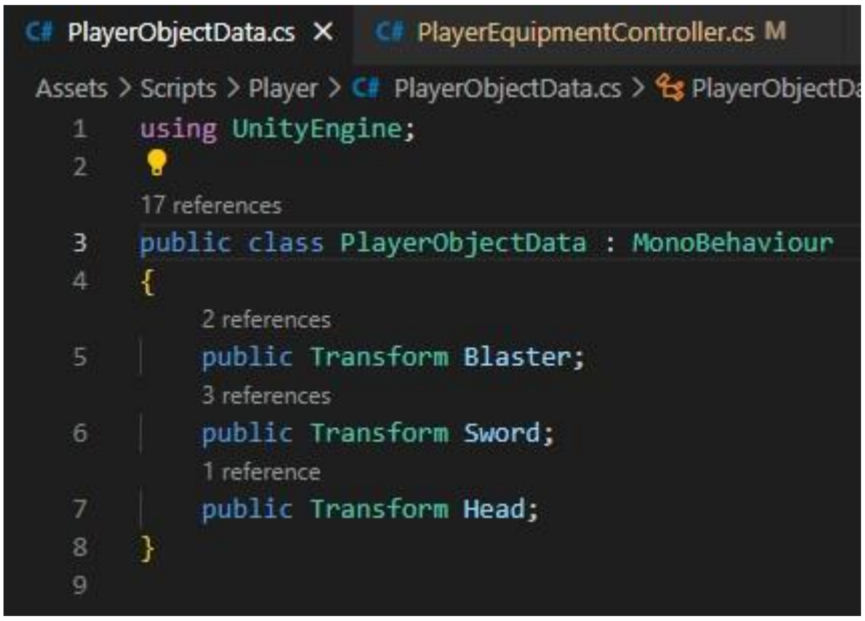


PlayerEquipmentController.cs M
CollisionCallbacks.cs U
SwordAttackAnimation.cs
Assets > Scripts > Settings > Physics > CollisionCallbacks.cs > CollisionCallbacks

```csharp
using System;
using UnityEngine;

9 references
public class CollisionCallbacks : MonoBehaviour
{
    10 references
    public Action<Collider> OnTriggerEntered = delegate { };
    1 reference
    public Action<Collision> OnCollisionEntered = delegate { };
    2 references
    public Action<Collider> OnTriggerStayed = delegate { };
    2 references
    public Action<Collider> OnTriggerExited = delegate { };

    0 references
    private void OnTriggerEnter(Collider other)
    {
        OnTriggerEntered(other);
    }

    0 references
    private void OnCollisionEnter(Collision collision)
    {
        OnCollisionEntered(collision);
    }

    0 references
    private void OnTriggerStay(Collider other)
    {
        OnTriggerStayed(other);
    }

    0 references
    private void OnTriggerExit(Collider other)
    {
        OnTriggerExited(other);
    }
}
```

❖ **PlayerObjectData.**

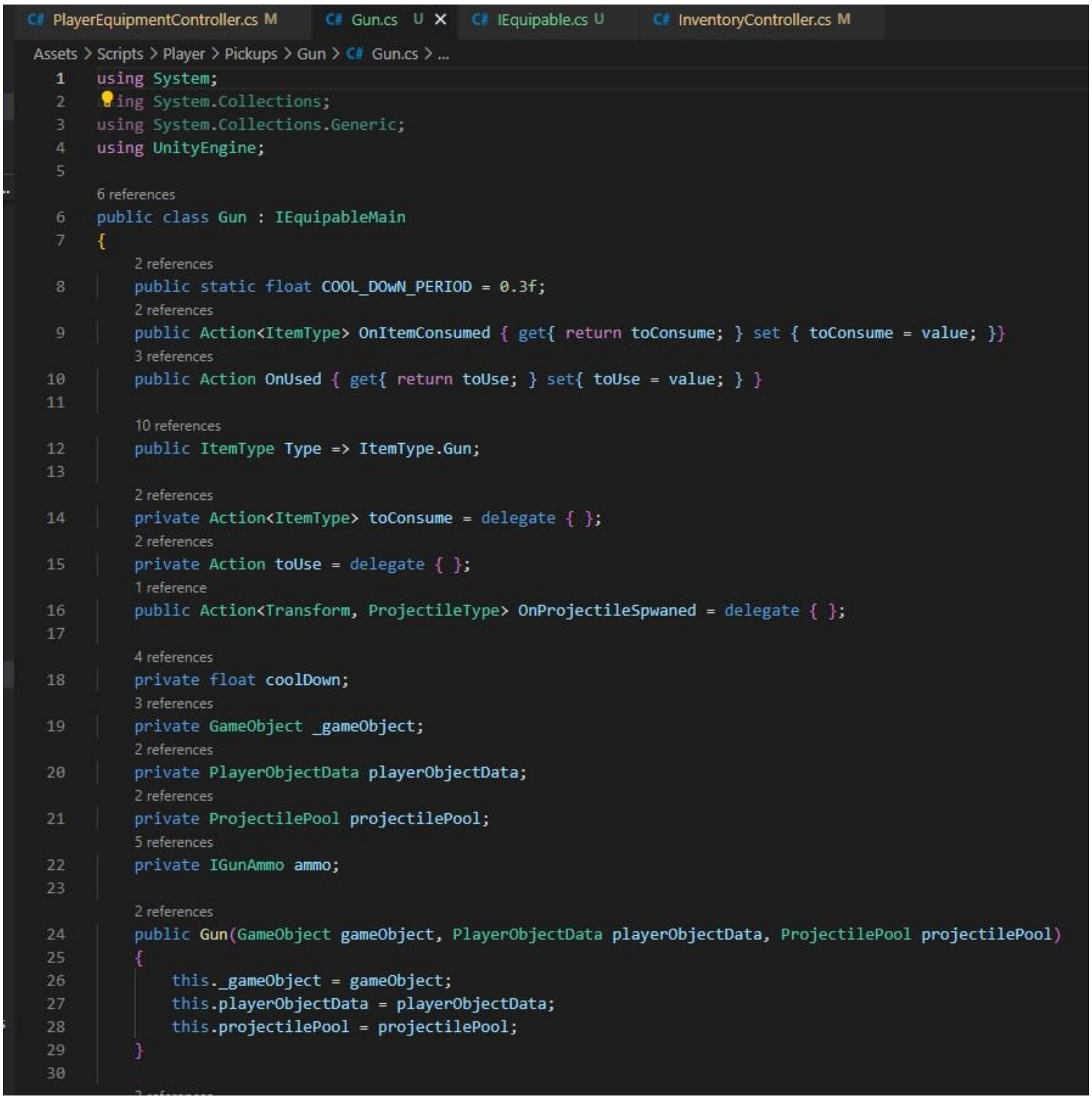
- It consists of all the Object that are attached to the Player.
- This script is attached to the Player GameObject in the scene.


PlayerObjectData.cs
PlayerEquipmentController.cs M
Assets > Scripts > Player > PlayerObjectData.cs > PlayerObjectD

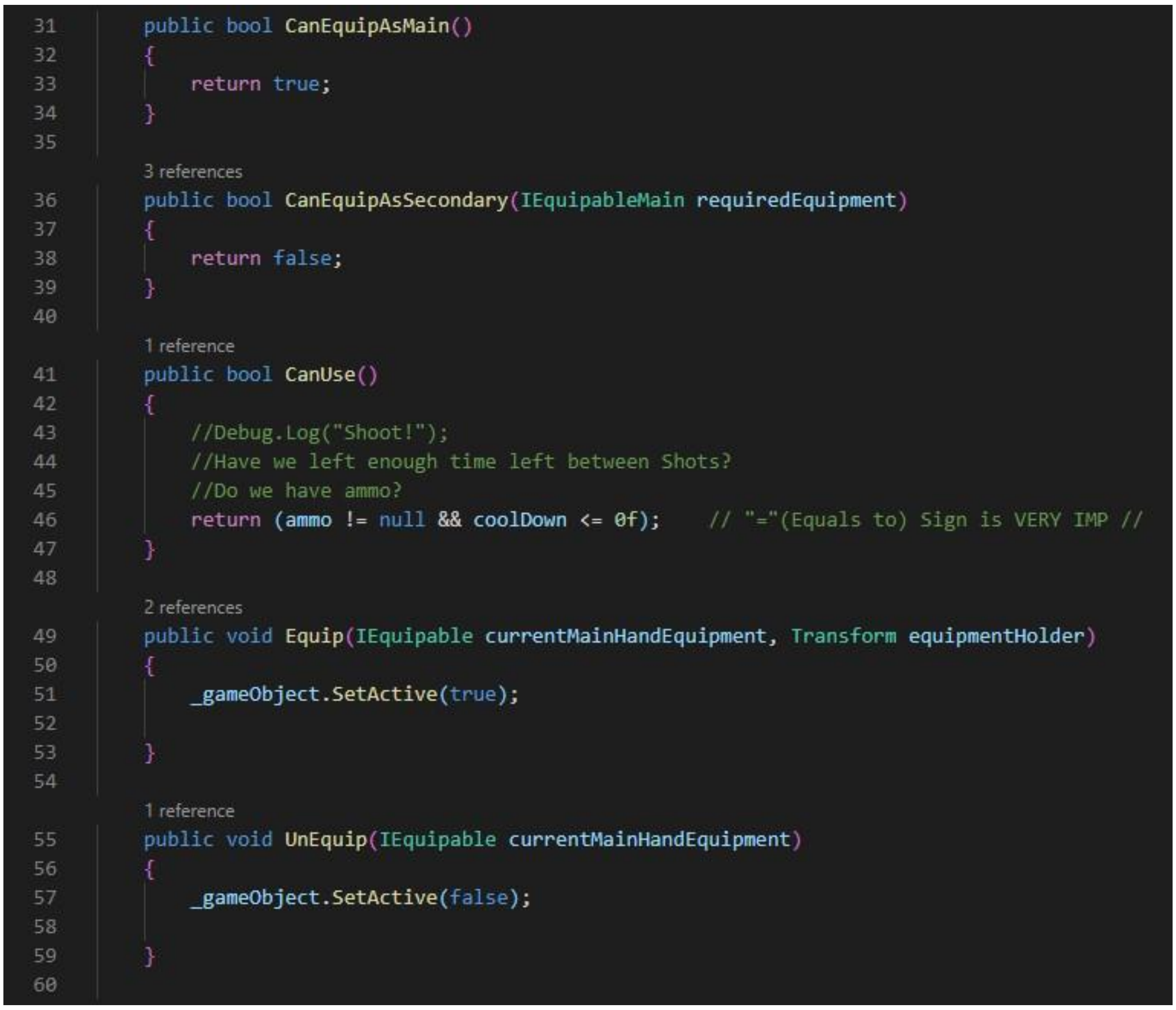
```csharp
using UnityEngine;

17 references
public class PlayerObjectData : MonoBehaviour
{
    2 references
    public Transform Blaster;
    3 references
    public Transform Sword;
    1 reference
    public Transform Head;
}
```

❖ **Gun.**

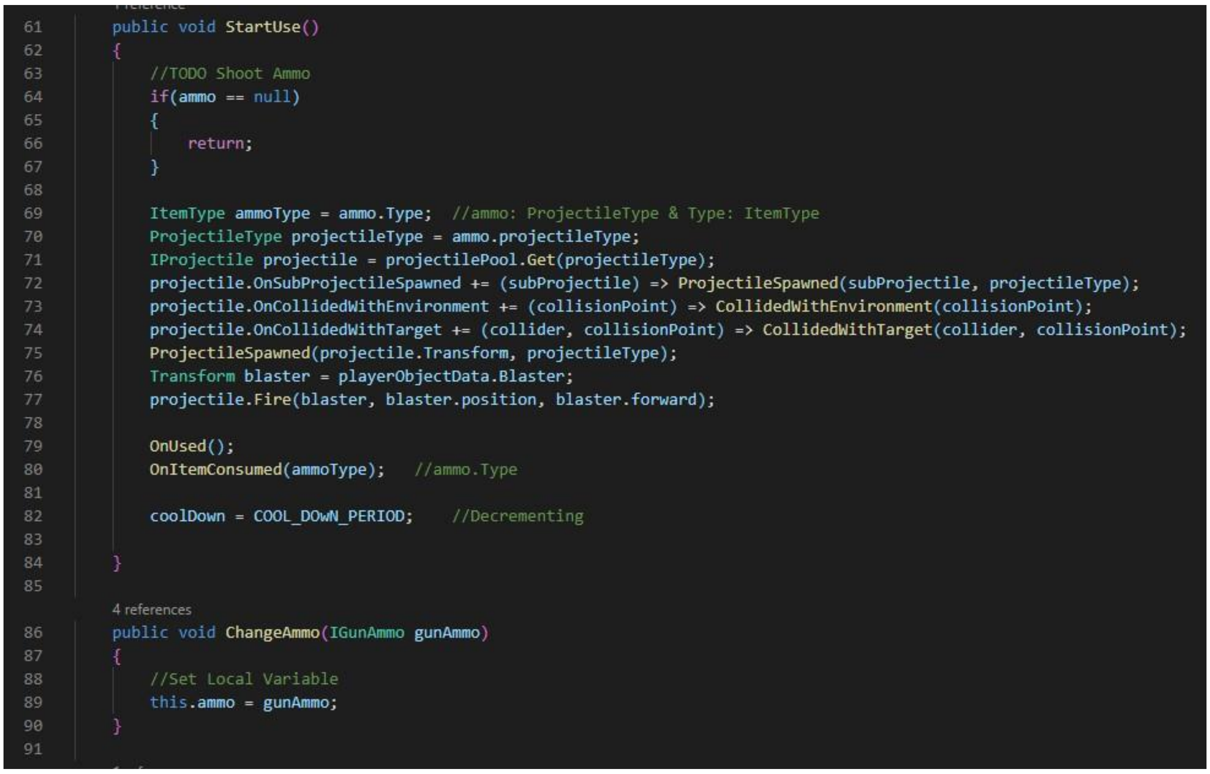
- It acts same as the Sword script, but the only difference in this is instead of playing some animation like Hitting, it actually fires bullets by accessing various scripts.
- It uses the IProjectile scripts to create projectiles and launch it.
- It first creates ProjectileType with the ItemType and then uses the other script to launch missiles and record its collisions with the Target and Environment.


C# PlayerEquipmentController.cs M
C# Gun.cs U
C# IEquipable.cs U
C# InventoryController.cs M
Assets > Scripts > Player > Pickups > Gun > C# Gun.cs > ...

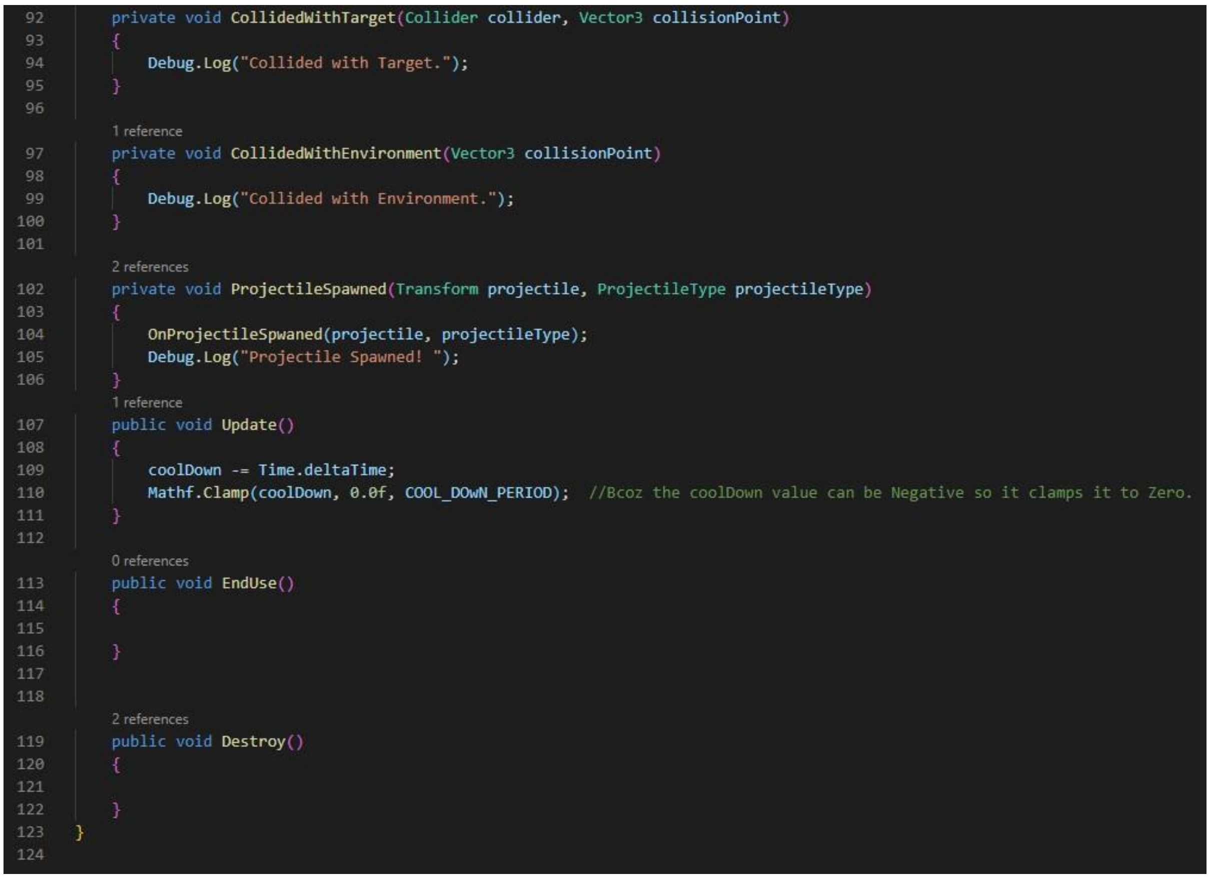
```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

6 references
public class Gun : IEquipableMain
{
    2 references
    public static float COOL_DOWN_PERIOD = 0.3f;
    2 references
    public Action<ItemType> OnItemConsumed { get{ return toConsume; } set { toConsume = value; }}
    3 references
    public Action OnUsed { get{ return toUse; } set{ toUse = value; } }

    10 references
    public ItemType Type => ItemType.Gun;

    2 references
    private Action<ItemType> toConsume = delegate { };
    2 references
    private Action toUse = delegate { };
    1 reference
    public Action<Transform, ProjectileType> OnProjectileSpwaned = delegate { };

    4 references
    private float coolDown;
    3 references
    private GameObject _gameObject;
    2 references
    private PlayerObjectData playerObjectData;
    2 references
    private ProjectilePool projectilePool;
    5 references
    private IGunAmmo ammo;

    2 references
    public Gun(GameObject gameObject, PlayerObjectData playerObjectData, ProjectilePool projectilePool)
    {
        this._gameObject = gameObject;
        this.playerObjectData = playerObjectData;
        this.projectilePool = projectilePool;
    }

```



```
    public bool CanEquipAsMain()
    {
        return true;
    }

    3 references
    public bool CanEquipAsSecondary(IEquipableMain requiredEquipment)
    {
        return false;
    }

    1 reference
    public bool CanUse()
    {
        //Debug.Log("Shoot!");
        //Have we left enough time left between Shots?
        //Do we have ammo?
        return (ammo != null && coolDown <= 0f);    // "="(Equals to) Sign is VERY IMP //
    }

    2 references
    public void Equip(IEquipable currentMainHandEquipment, Transform equipmentHolder)
    {
        _gameObject.SetActive(true);

    }

    1 reference
    public void UnEquip(IEquipable currentMainHandEquipment)
    {
        _gameObject.SetActive(false);

    }

```



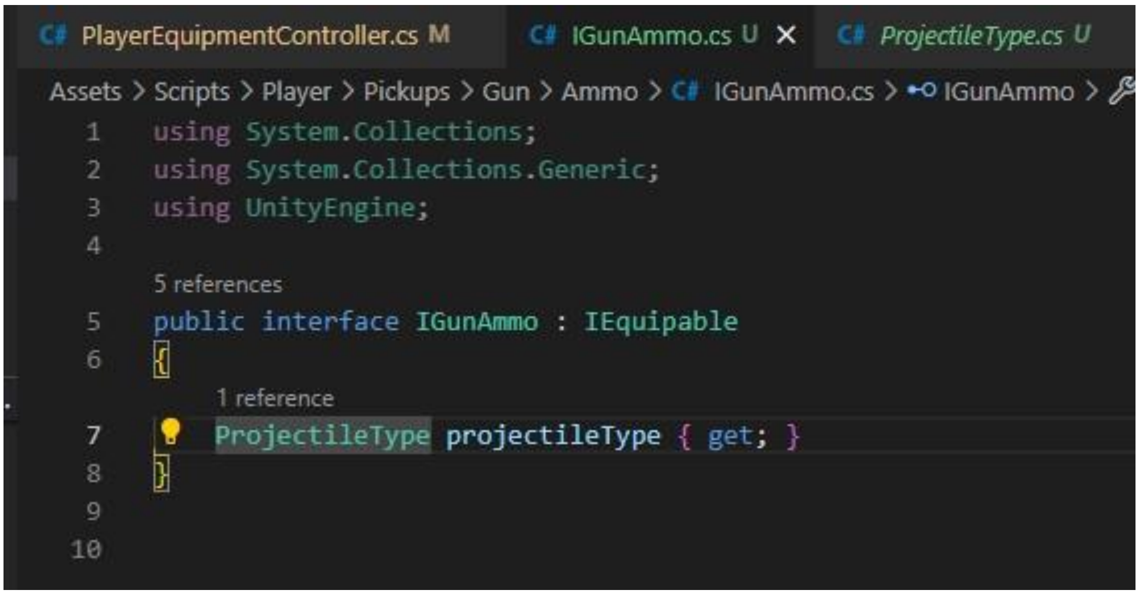
```
    public void StartUse()
    {
        //TODO Shoot Ammo
        if(ammo == null)
        {
            return;
        }

        ItemType ammoType = ammo.Type;  //ammo: ProjectileType & Type: ItemType
        ProjectileType projectileType = ammo.projectileType;
        IProjectile projectile = projectilePool.Get(projectileType);
        projectile.OnSubProjectileSpawned += (subProjectile) => ProjectileSpawned(subProjectile, projectileType);
        projectile.OnCollidedWithEnvironment += (collisionPoint) => CollidedWithEnvironment(collisionPoint);
        projectile.OnCollidedWithTarget += (collider, collisionPoint) => CollidedWithTarget(collider, collisionPoint);
        ProjectileSpawned(projectile.Transform, projectileType);
        Transform blaster = playerObjectData.Blaster;
        projectile.Fire(blaster, blaster.position, blaster.forward);

        OnUsed();
        OnItemConsumed(ammoType);   //ammo.Type

        coolDown = COOL_DOwN_PERIOD;    //Decrementing

    }

    4 references
    public void ChangeAmmo(IGunAmmo gunAmmo)
    {
        //Set Local Variable
        this.ammo = gunAmmo;
    }

```



```
    private void CollidedWithTarget(Collider collider, Vector3 collisionPoint)
    {
        Debug.Log("Collided with Target.");
    }

    1 reference
    private void CollidedWithEnvironment(Vector3 collisionPoint)
    {
        Debug.Log("Collided with Environment.");
    }

    2 references
    private void ProjectileSpawned(Transform projectile, ProjectileType projectileType)
    {
        OnProjectileSpwaned(projectile, projectileType);
        Debug.Log("Projectile Spawned! ");
    }
    1 reference
    public void Update()
    {
        coolDown -= Time.deltaTime;
        Mathf.Clamp(coolDown, 0.0f, COOL_DOWN_PERIOD);  //Bcoz the coolDown value can be Negative so it clamps it to Zero.
    }

    0 references
    public void EndUse()
    {

    }


    2 references
    public void Destroy()
    {

    }
}
```

❖ **IGunAmmo.**

- It simply consists of ProjectileType.
- It also implements the IEquipable interface.

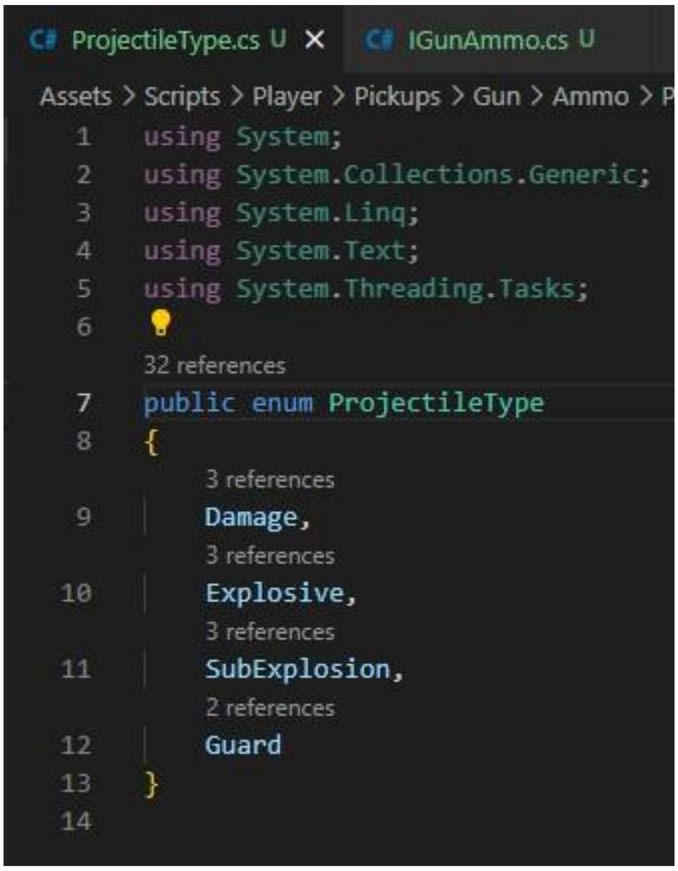


C# PlayerEquipmentController.cs M
C# IGunAmmo.cs U
C# ProjectileType.cs U
Assets > Scripts > Player > Pickups > Gun > Ammo > C# IGunAmmo.cs > IGunAmmo

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

5 references
public interface IGunAmmo : IEquipable
{
    1 reference
    ProjectileType projectileType { get; }
}
```

❖ **ProjectileType.**

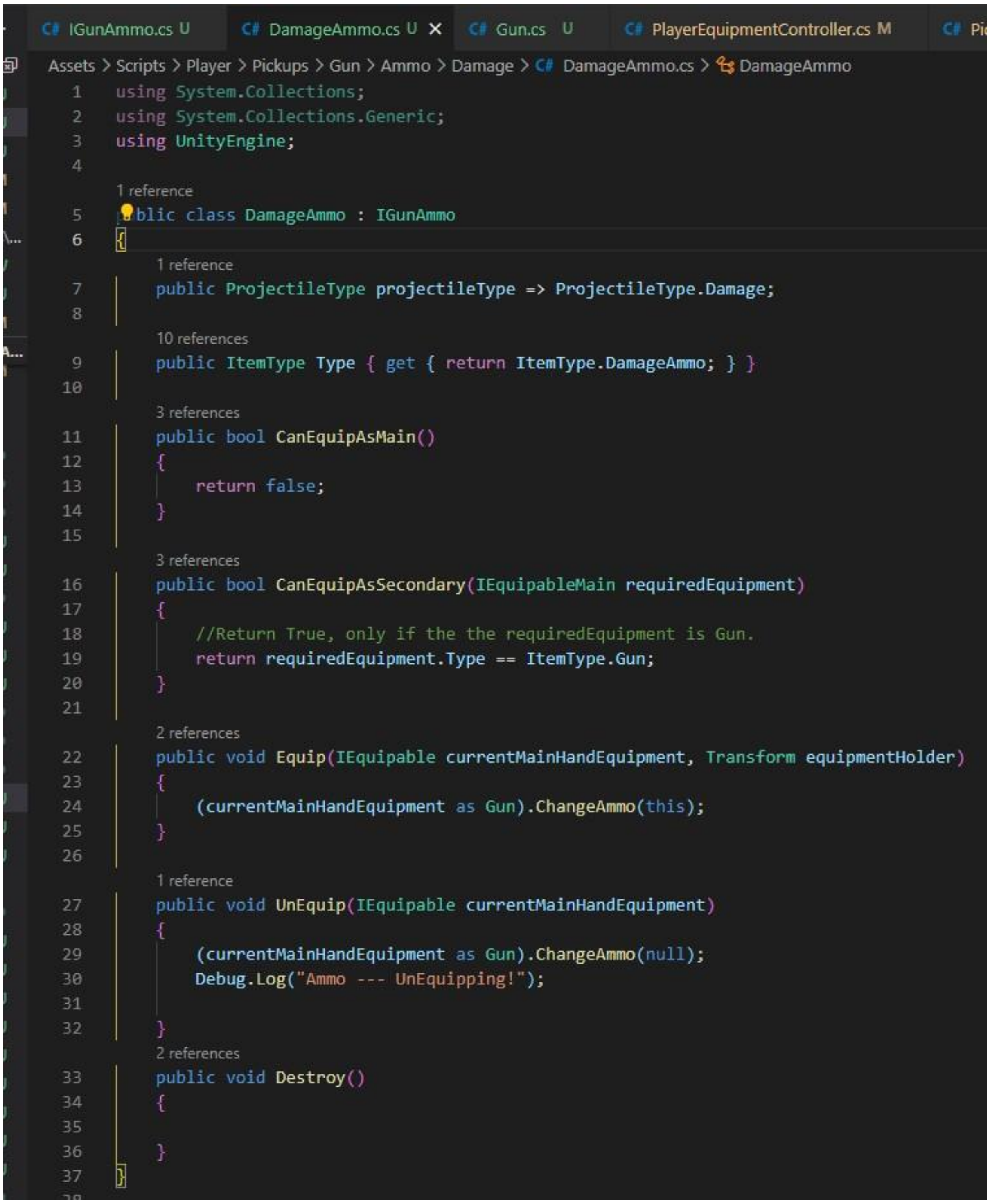
- It is an Enum which consists of the Types of Missiles or Ammos.
- It is similar to the ItemType, and is mainly used to check if the ItemType is equal to or matches with the ProjectileType.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

32 references
public enum ProjectileType
{
    3 references
    Damage,
    3 references
    Explosive,
    3 references
    SubExplosion,
    2 references
    Guard
}

```

- ❖ **DamageAmmo.**

- It implements the "*IGunAmmo*" interface.
- And because the IGunAmmo script uses the "*IEquipable*" interface, it also gets implemented in this script.
- This script basically creates ammo of the Type defined. (In this case it is DamageAmmo type)
- It first matches the ProjectileType with the ItemType, and then check whether it can be Equipped as Secondary, by passing a check statement that checks if the Main Required Equipment is equal to the Gun type.
- If it is, then it can be equipped, otherwise it cannot be equipped if it does not match the Gun type.
- That is, if we are holding a Sword or any other Weapon, other than Gun, we should be able to grab any type of Ammo as a Secondary Equipment.
- After checking the above statement, we pass this data into the Main or Primary Equipment class (Gun), under the function "*ChangeAmmo*" of the "*Gun*" script.
- This function basically sets current ammo that we are pass to the Gun script's ammo, which is "*IGunAmmo*" class.
- Then we use the Ammo.
- The amount of Ammo is set in the Inspector, with the "*Pickup*" script.
- And the ammo is decremented by decrementing its Count, from the "*InventoryController*" script.
- It is also displayed in the Inventory Menu UI.

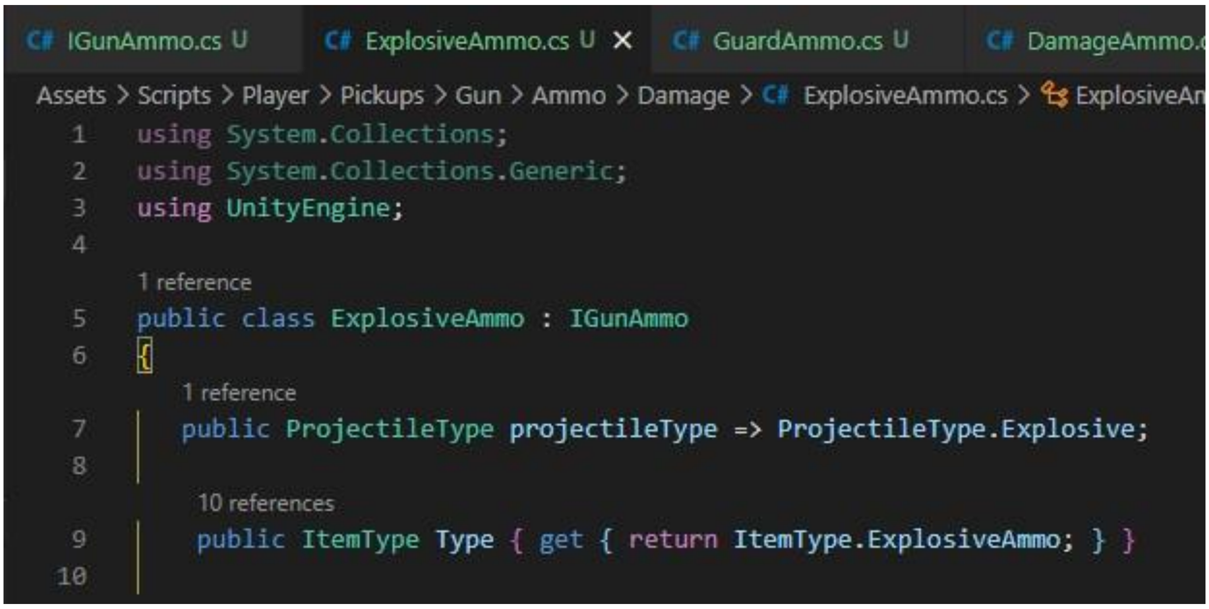


C# IGunAmmo.cs U
C# DamageAmmo.cs U ×
C# Gun.cs U
C# PlayerEquipmentController.cs M
C# Pi
Assets > Scripts > Player > Pickups > Gun > Ammo > Damage > C# DamageAmmo.cs > DamageAmmo

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

1 reference
public class DamageAmmo : IGunAmmo
{
    1 reference
    public ProjectileType projectileType => ProjectileType.Damage;

    10 references
    public ItemType Type { get { return ItemType.DamageAmmo; } }

    3 references
    public bool CanEquipAsMain()
    {
        return false;
    }

    3 references
    public bool CanEquipAsSecondary(IEquipableMain requiredEquipment)
    {
        //Return True, only if the the requiredEquipment is Gun.
        return requiredEquipment.Type == ItemType.Gun;
    }

    2 references
    public void Equip(IEquipable currentMainHandEquipment, Transform equipmentHolder)
    {
        (currentMainHandEquipment as Gun).ChangeAmmo(this);
    }

    1 reference
    public void UnEquip(IEquipable currentMainHandEquipment)
    {
        (currentMainHandEquipment as Gun).ChangeAmmo(null);
        Debug.Log("Ammo --- UnEquipping!");

    }
    2 references
    public void Destroy()
    {

    }
}
```

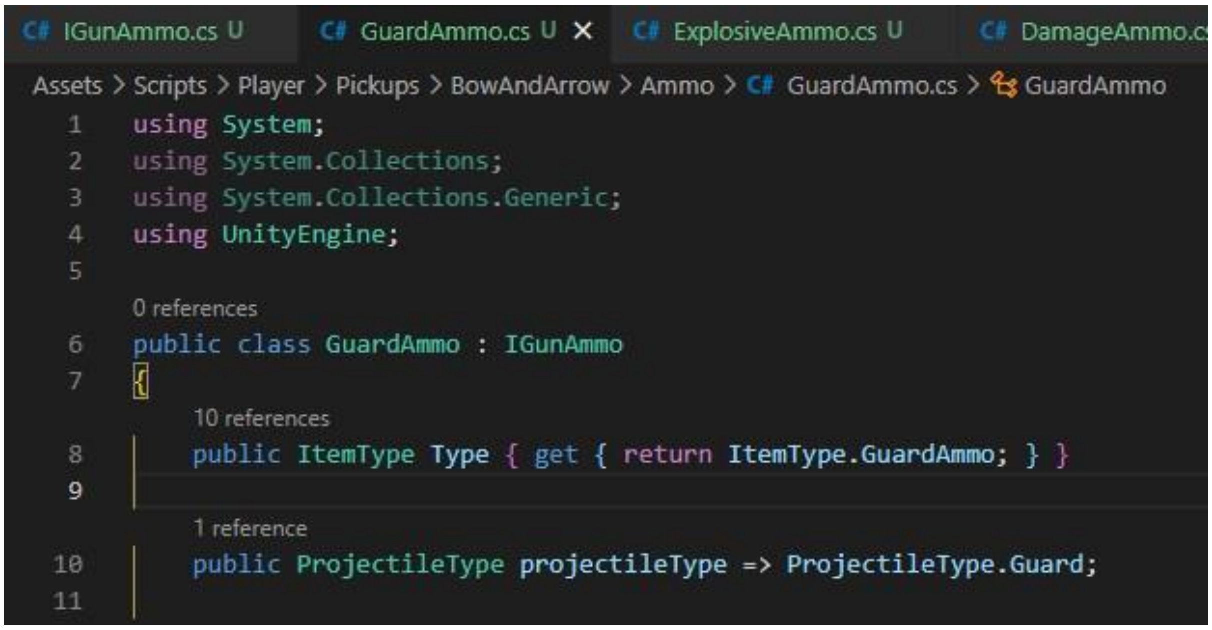
❖ **ExplosiveAmmo & GuardAmmo.**

- It is just same as DamageAmmo.
- Its functionality is same.
- Just the difference or change in these scripts is the "*ProjectileType*" and "*ItemType*" would be ExplosiveAmmo & GuardAmmo respectively.



C# IGunAmmo.cs U
C# ExplosiveAmmo.cs U ×
C# GuardAmmo.cs U
C# DamageAmmo.
Assets > Scripts > Player > Pickups > Gun > Ammo > Damage > C# ExplosiveAmmo.cs > ExplosiveAn

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

1 reference
public class ExplosiveAmmo : IGunAmmo
{
    1 reference
    public ProjectileType projectileType => ProjectileType.Explosive;

     10 references
     public ItemType Type { get { return ItemType.ExplosiveAmmo; } }
```

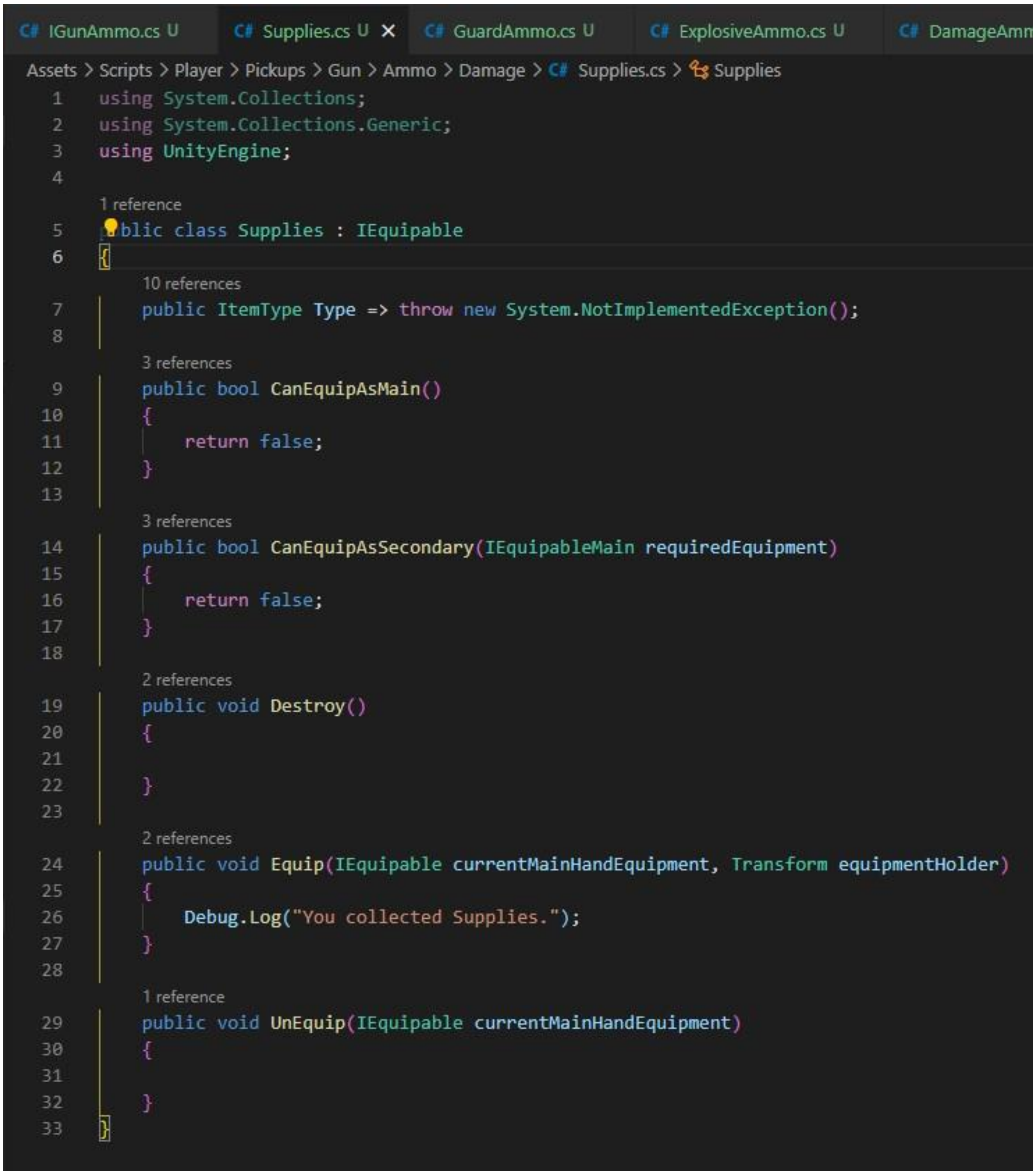


C# IGunAmmo.cs U
C# GuardAmmo.cs U ×
C# ExplosiveAmmo.cs U
C# DamageAmmo.c
Assets > Scripts > Player > Pickups > BowAndArrow > Ammo > C# GuardAmmo.cs > GuardAmmo

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

0 references
public class GuardAmmo : IGunAmmo
{
    10 references
    public ItemType Type { get { return ItemType.GuardAmmo; } }

    1 reference
    public ProjectileType projectileType => ProjectileType.Guard;
```

❖ **Supplies.**

- It is created to get rid of the "*DebugError*" that popes up when we picked up supplies.
- It simply prints a "*Debug.Log*" message when we pickup supplies.

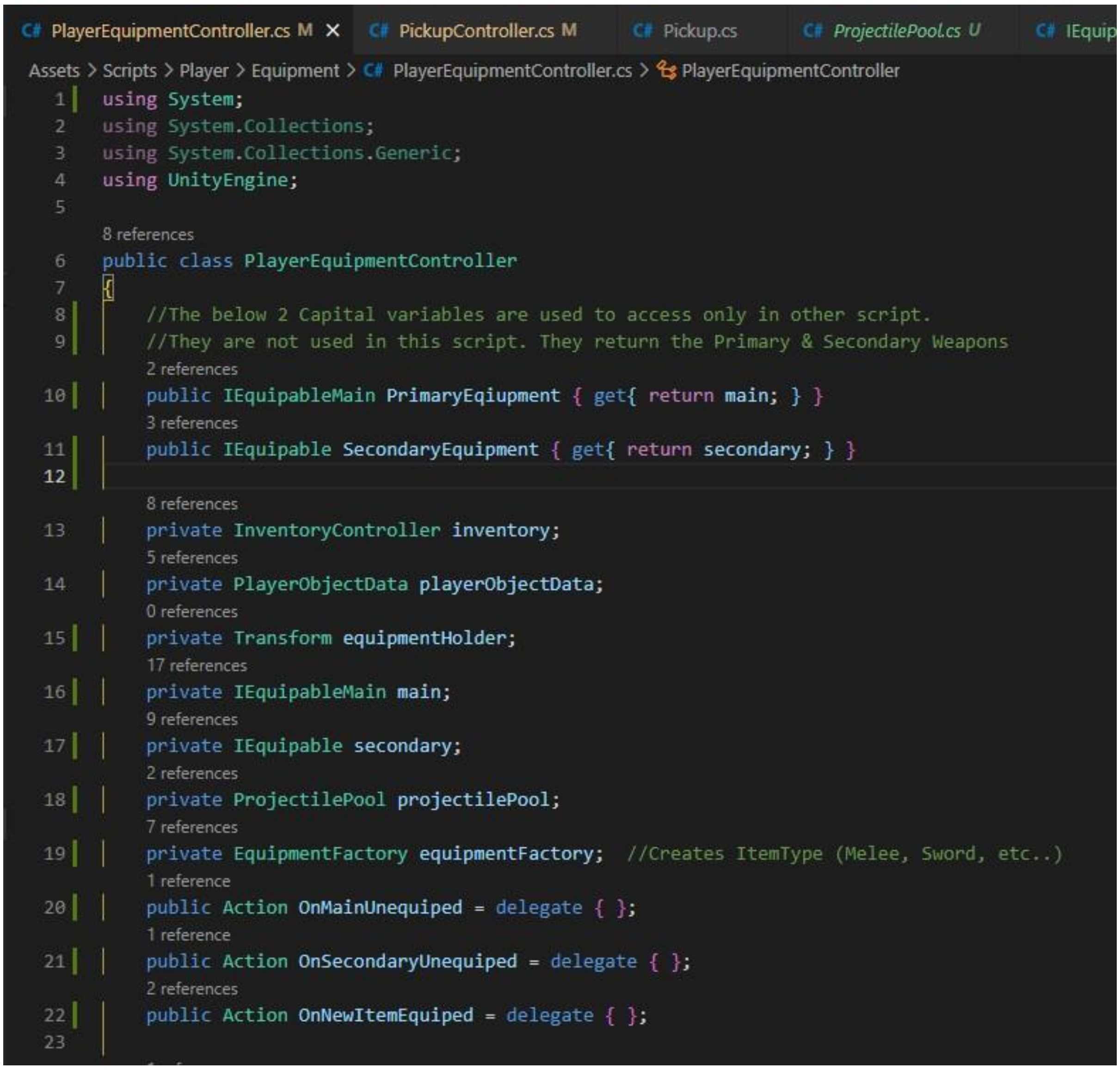


C# IGunAmmo.cs U
C# Supplies.cs U ×
C# GuardAmmo.cs U
C# ExplosiveAmmo.cs U
C# DamageAmm
Assets > Scripts > Player > Pickups > Gun > Ammo > Damage > C# Supplies.cs > Supplies

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

1 reference
public class Supplies : IEquipable
{
    10 references
    public ItemType Type => throw new System.NotImplementedException();

    3 references
    public bool CanEquipAsMain()
    {
        return false;
    }

    3 references
    public bool CanEquipAsSecondary(IEquipableMain requiredEquipment)
    {
        return false;
    }

    2 references
    public void Destroy()
    {

    }

    2 references
    public void Equip(IEquipable currentMainHandEquipment, Transform equipmentHolder)
    {
        Debug.Log("You collected Supplies.");
    }

    1 reference
    public void UnEquip(IEquipable currentMainHandEquipment)
    {

    }
}
```

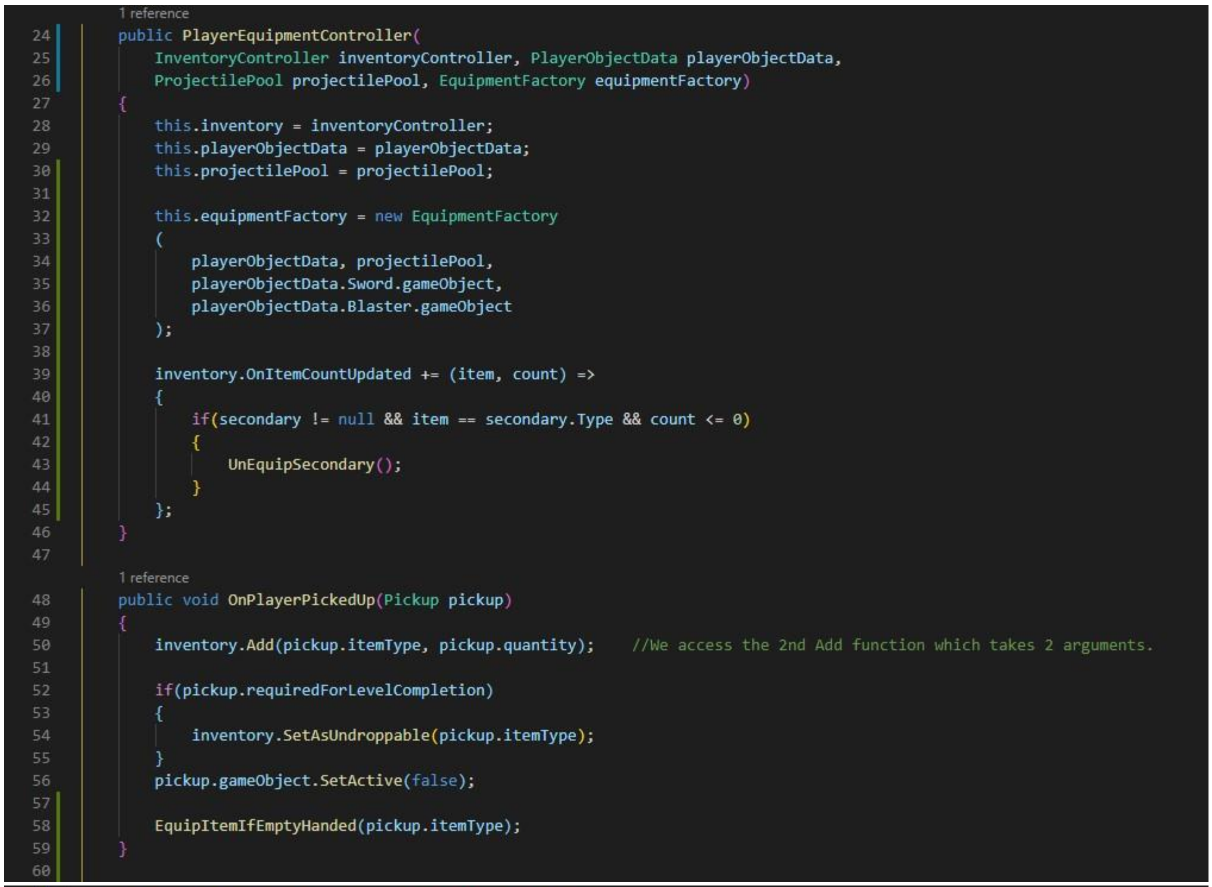
❖ **PlayerEquipmentController.**

- It is the Main Parent Script which controls all the other scripts.
- It consists of Functions , which examine the picked-up item, and then check whether it can be used as a Main Equipment or Secondary Equipment, and assign it, to the respective slots.
- It also has function that perform the Weapon Switch action as well as UnEquip item(s).

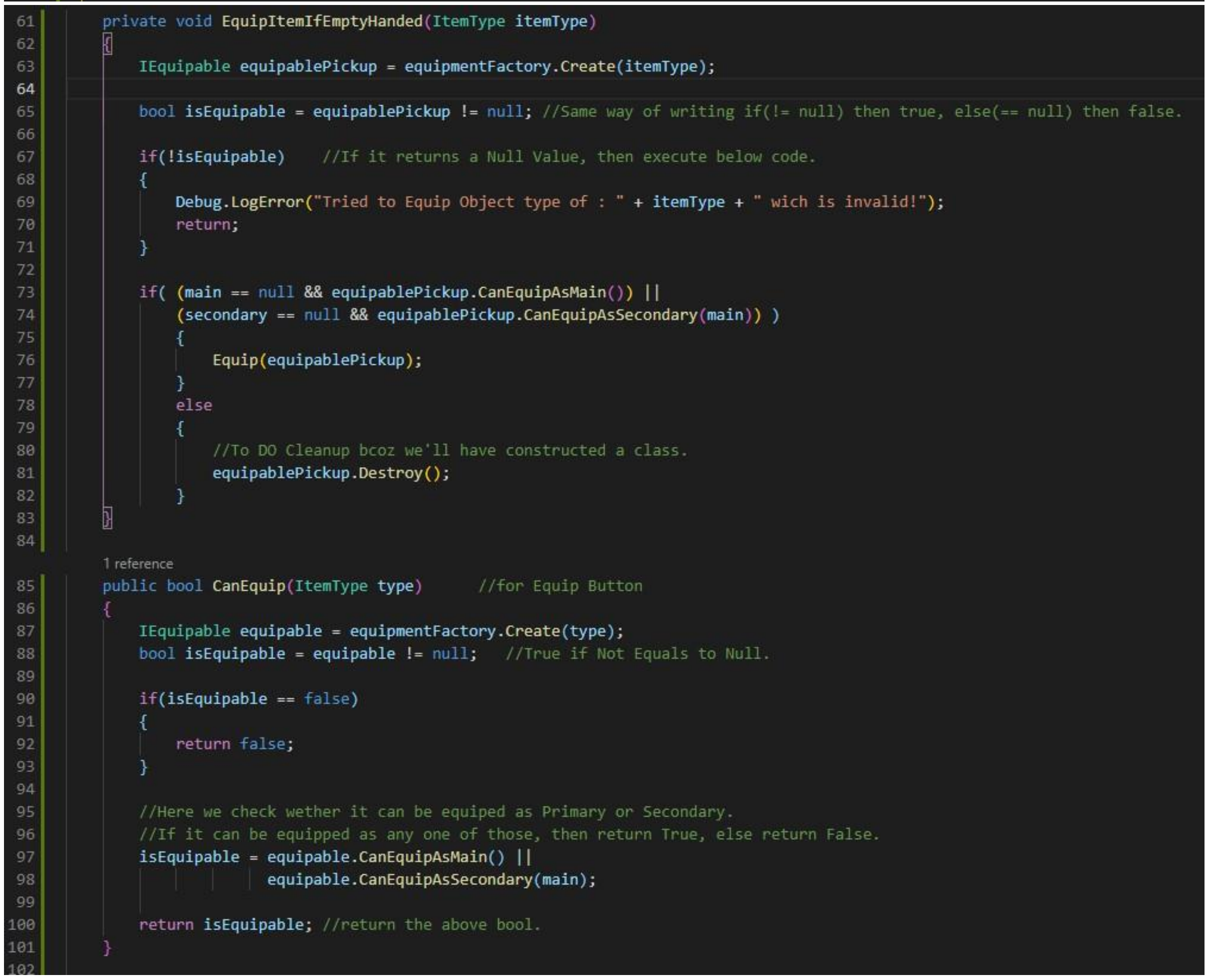


PlayerEquipmentController.cs M ×
PickupController.cs M
Pickup.cs
ProjectilePool.cs U
IEquip
Assets > Scripts > Player > Equipment > PlayerEquipmentController.cs > PlayerEquipmentController

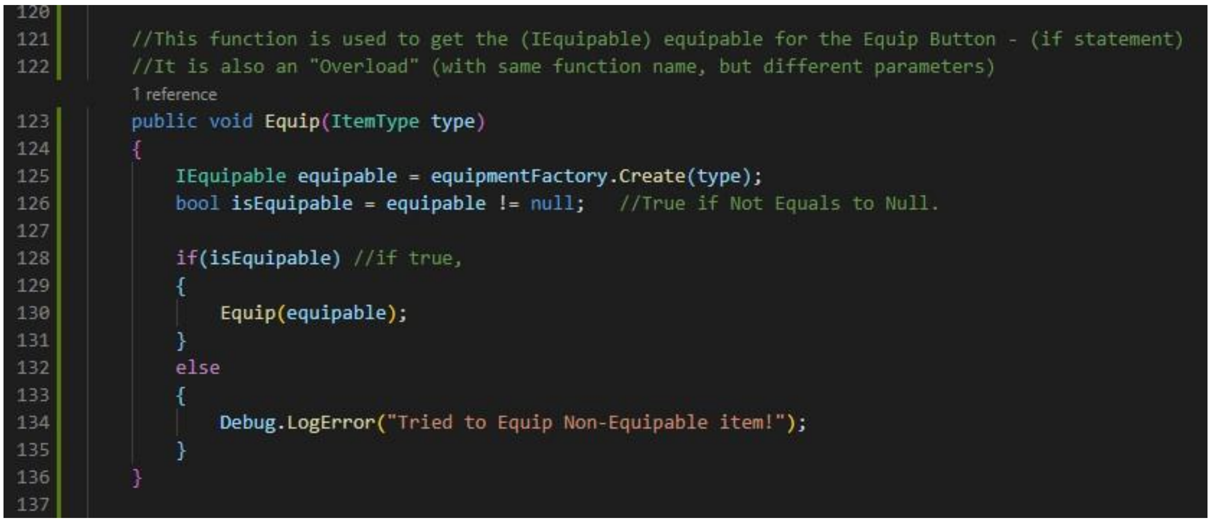
```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

8 references
public class PlayerEquipmentController
{
    //The below 2 Capital variables are used to access only in other script.
    //They are not used in this script. They return the Primary & Secondary Weapons
    2 references
    public IEquipableMain PrimaryEqiupment { get{ return main; } }
    3 references
    public IEquipable SecondaryEquipment { get{ return secondary; } }

    8 references
    private InventoryController inventory;
    5 references
    private PlayerObjectData playerObjectData;
    0 references
    private Transform equipmentHolder;
    17 references
    private IEquipableMain main;
    9 references
    private IEquipable secondary;
    2 references
    private ProjectilePool projectilePool;
    7 references
    private EquipmentFactory equipmentFactory;  //Creates ItemType (Melee, Sword, etc..)
    1 reference
    public Action OnMainUnequiped = delegate { };
    1 reference
    public Action OnSecondaryUnequiped = delegate { };
    2 references
    public Action OnNewItemEquiped = delegate { };

```



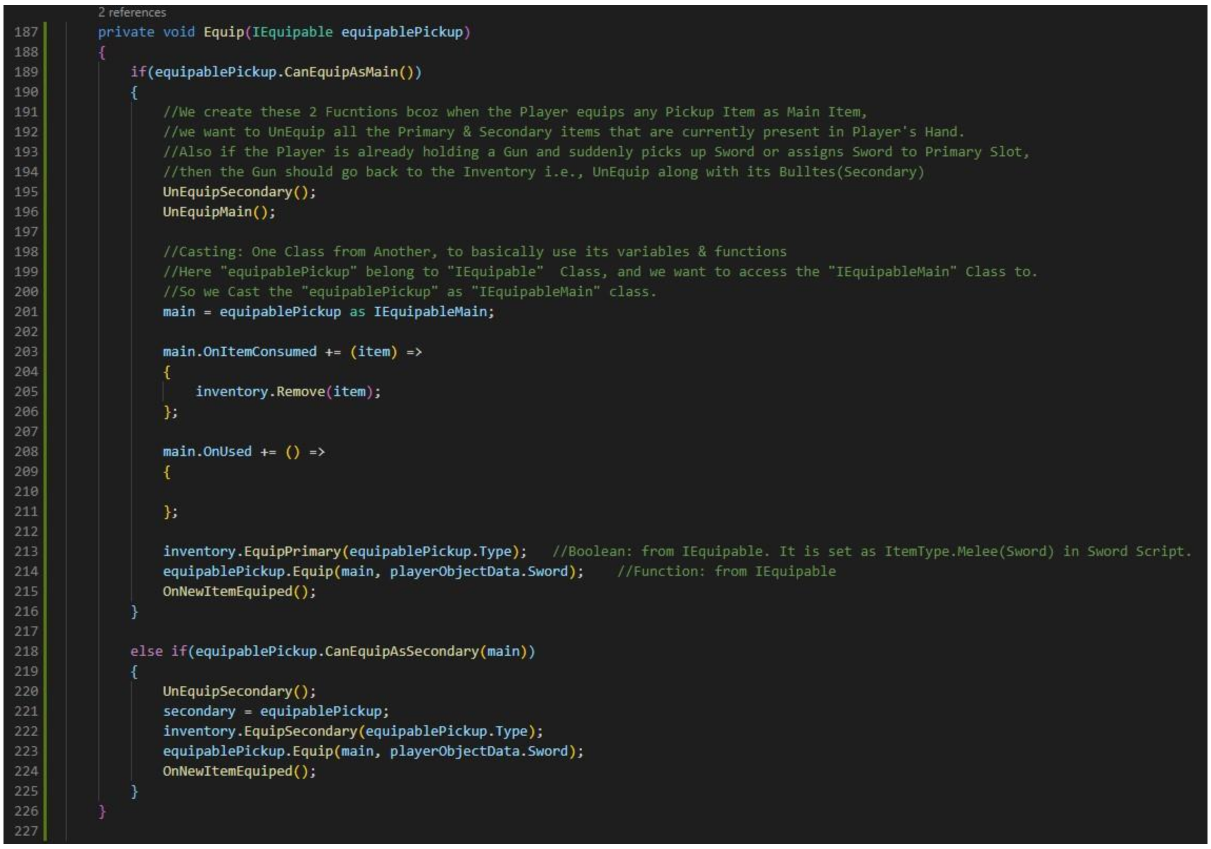
```csharp
1 reference
public PlayerEquipmentController(
    InventoryController inventoryController, PlayerObjectData playerObjectData,
    ProjectilePool projectilePool, EquipmentFactory equipmentFactory)
{
    this.inventory = inventoryController;
    this.playerObjectData = playerObjectData;
    this.projectilePool = projectilePool;

    this.equipmentFactory = new EquipmentFactory
    (
        playerObjectData, projectilePool,
        playerObjectData.Sword.gameObject,
        playerObjectData.Blaster.gameObject
    );

    inventory.OnItemCountUpdated += (item, count) =>
    {
        if(secondary != null && item == secondary.Type && count <= 0)
        {
            UnEquipSecondary();
        }
    };
}

1 reference
public void OnPlayerPickedUp(Pickup pickup)
{
    inventory.Add(pickup.itemType, pickup.quantity);    //We access the 2nd Add function which takes 2 arguments.

    if(pickup.requiredForLevelCompletion)
    {
        inventory.SetAsUndroppable(pickup.itemType);
    }
    pickup.gameObject.SetActive(false);

    EquipItemIfEmptyHanded(pickup.itemType);
}

```



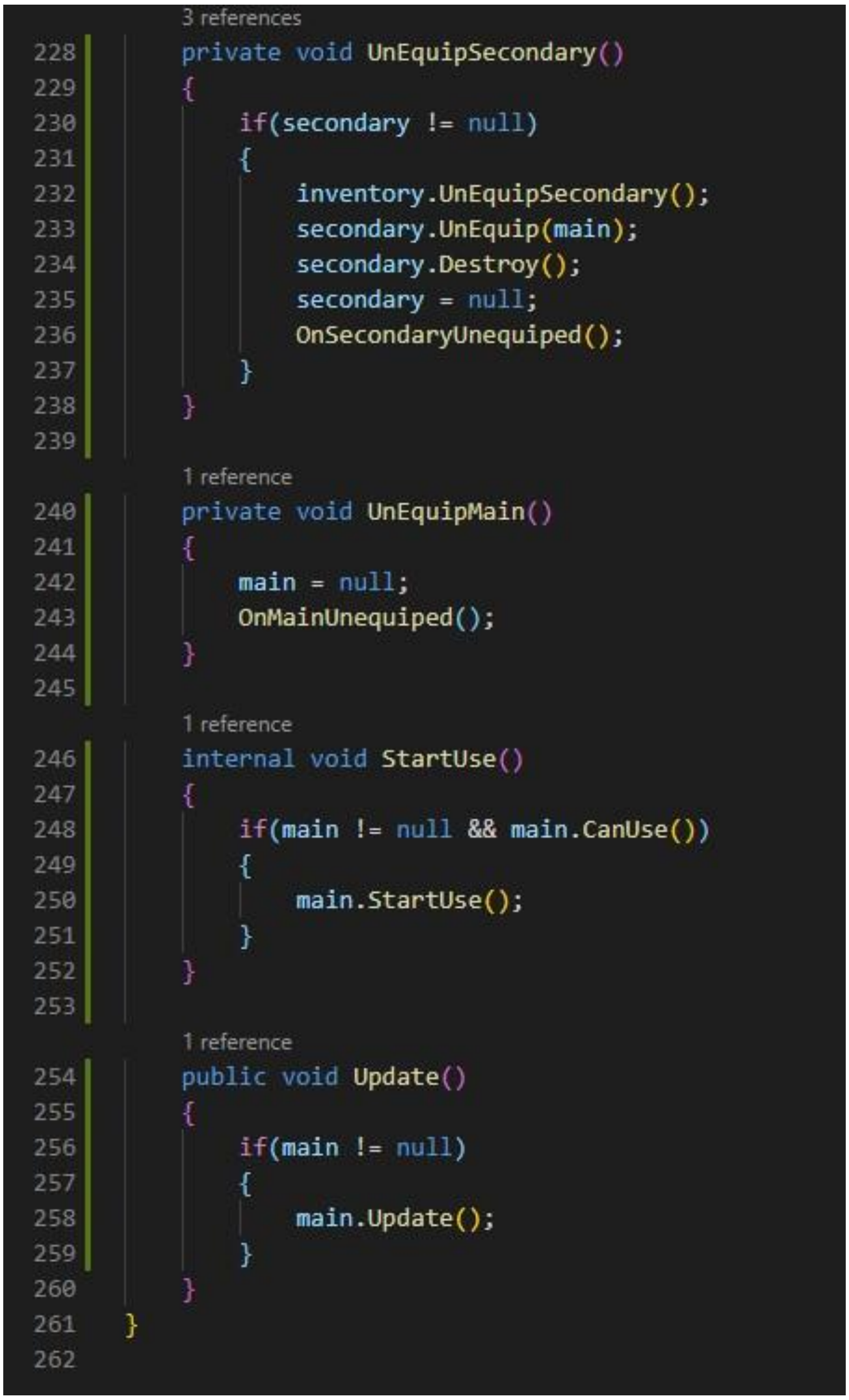
```csharp
private void EquipItemIfEmptyHanded(ItemType itemType)
{
    IEquipable equipablePickup = equipmentFactory.Create(itemType);

    bool isEquipable = equipablePickup != null; //Same way of writing if(!= null) then true, else(== null) then false.

    if(!isEquipable)    //If it returns a Null Value, then execute below code.
    {
        Debug.LogError("Tried to Equip Object type of : " + itemType + " wich is invalid!");
        return;
    }

    if( (main == null && equipablePickup.CanEquipAsMain()) ||
        (secondary == null && equipablePickup.CanEquipAsSecondary(main)) )
        {
            Equip(equipablePickup);
        }
        else
        {
            //To DO Cleanup bcoz we'll have constructed a class.
            equipablePickup.Destroy();
        }
}

1 reference
public bool CanEquip(ItemType type)      //for Equip Button
{
    IEquipable equipable = equipmentFactory.Create(type);
    bool isEquipable = equipable != null;   //True if Not Equals to Null.

    if(isEquipable == false)
    {
        return false;
    }

    //Here we check wether it can be equiped as Primary or Secondary.
    //If it can be equipped as any one of those, then return True, else return False.
    isEquipable = equipable.CanEquipAsMain() ||
                  equipable.CanEquipAsSecondary(main);

    return isEquipable; //return the above bool.
}

```



```
    //This function is used to get the (IEquipable) equipable for the Equip Button - (if statement)
    //It is also an "Overload" (with same function name, but different parameters)
    1 reference
    public void Equip(ItemType type)
    {
        IEquipable equipable = equipmentFactory.Create(type);
        bool isEquipable = equipable != null;   //True if Not Equals to Null.

        if(isEquipable) //if true,
        {
            Equip(equipable);
        }
        else
        {
            Debug.LogError("Tried to Equip Non-Equipable item!");
        }
    }

```



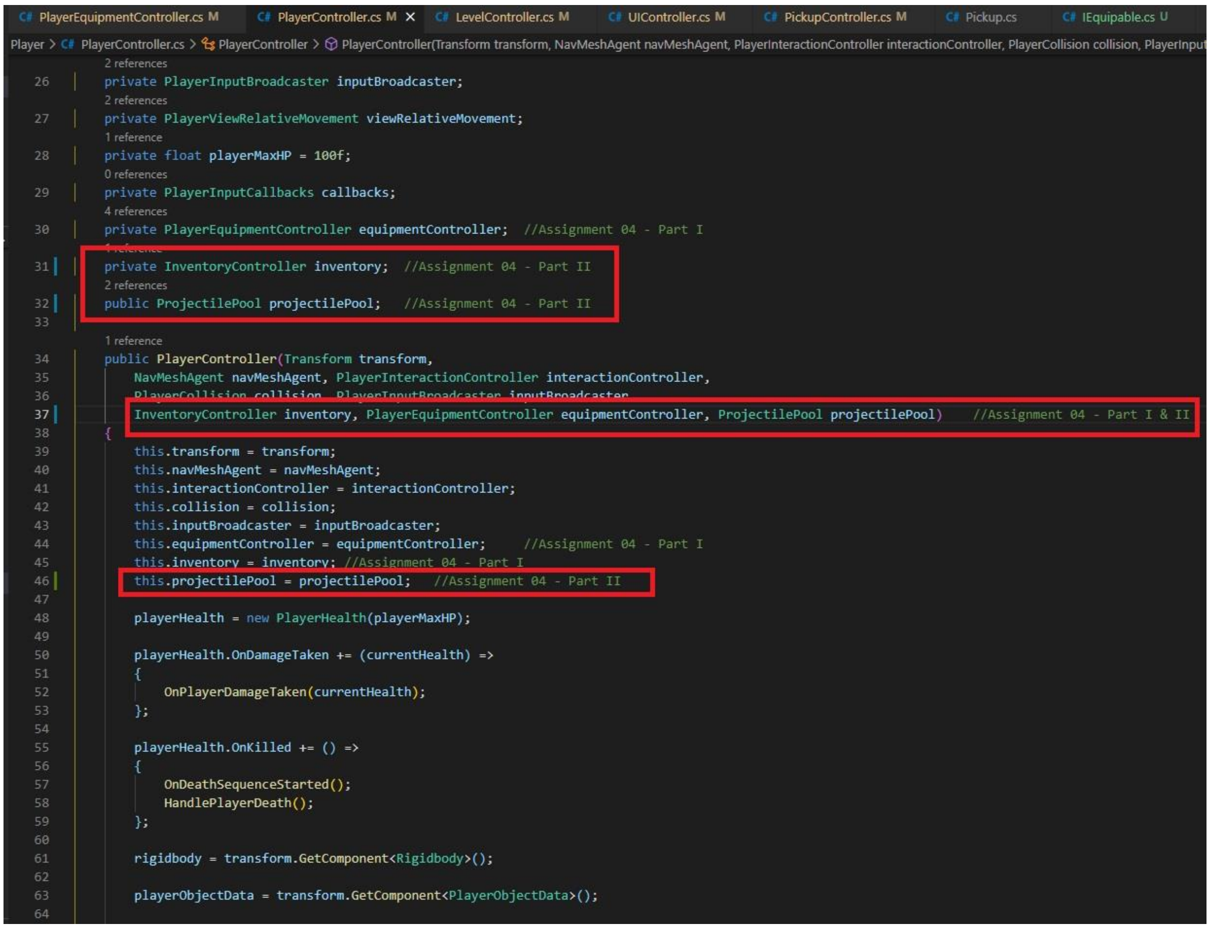
```
    2 references
    private void Equip(IEquipable equipablePickup)
    {
        if(equipablePickup.CanEquipAsMain())
        {
            //We create these 2 Fucntions bcoz when the Player equips any Pickup Item as Main Item,
            //we want to UnEquip all the Primary & Secondary items that are currently present in Player's Hand.
            //Also if the Player is already holding a Gun and suddenly picks up Sword or assigns Sword to Primary Slot,
            //then the Gun should go back to the Inventory i.e., UnEquip along with its Bulltes(Secondary)
            UnEquipSecondary();
            UnEquipMain();

            //Casting: One Class from Another, to basically use its variables & functions
            //Here "equipablePickup" belong to "IEquipable"  Class, and we want to access the "IEquipableMain" Class to.
            //So we Cast the "equipablePickup" as "IEquipableMain" class.
            main = equipablePickup as IEquipableMain;

            main.OnItemConsumed += (item) =>
            {
                inventory.Remove(item);
            };

            main.OnUsed += () =>
            {

            };

            inventory.EquipPrimary(equipablePickup.Type);   //Boolean: from IEquipable. It is set as ItemType.Melee(Sword) in Sword Script.
            equipablePickup.Equip(main, playerObjectData.Sword);    //Function: from IEquipable
            OnNewItemEquiped();
        }

        else if(equipablePickup.CanEquipAsSecondary(main))
        {
            UnEquipSecondary();
            secondary = equipablePickup;
            inventory.EquipSecondary(equipablePickup.Type);
            equipablePickup.Equip(main, playerObjectData.Sword);
            OnNewItemEquiped();
        }
    }

```



```
        3 references
        private void UnEquipSecondary()
        {
            if(secondary != null)
            {
                inventory.UnEquipSecondary();
                secondary.UnEquip(main);
                secondary.Destroy();
                secondary = null;
                OnSecondaryUnequiped();
            }
        }

        1 reference
        private void UnEquipMain()
        {
            main = null;
            OnMainUnequiped();
        }

        1 reference
        internal void StartUse()
        {
            if(main != null && main.CanUse())
            {
                main.StartUse();
            }
        }

        1 reference
        public void Update()
        {
            if(main != null)
            {
                main.Update();
            }
        }
    }

```

❖ **PlayerController.**

- We just pass a reference of the "ProjectilePool" script in here, so that it can be accessed in the LevelController.



PlayerEquipmentController.cs M | PlayerController.cs M | LevelController.cs M | UIController.cs M | PickupController.cs M | Pickup.cs | IEquipable.cs U

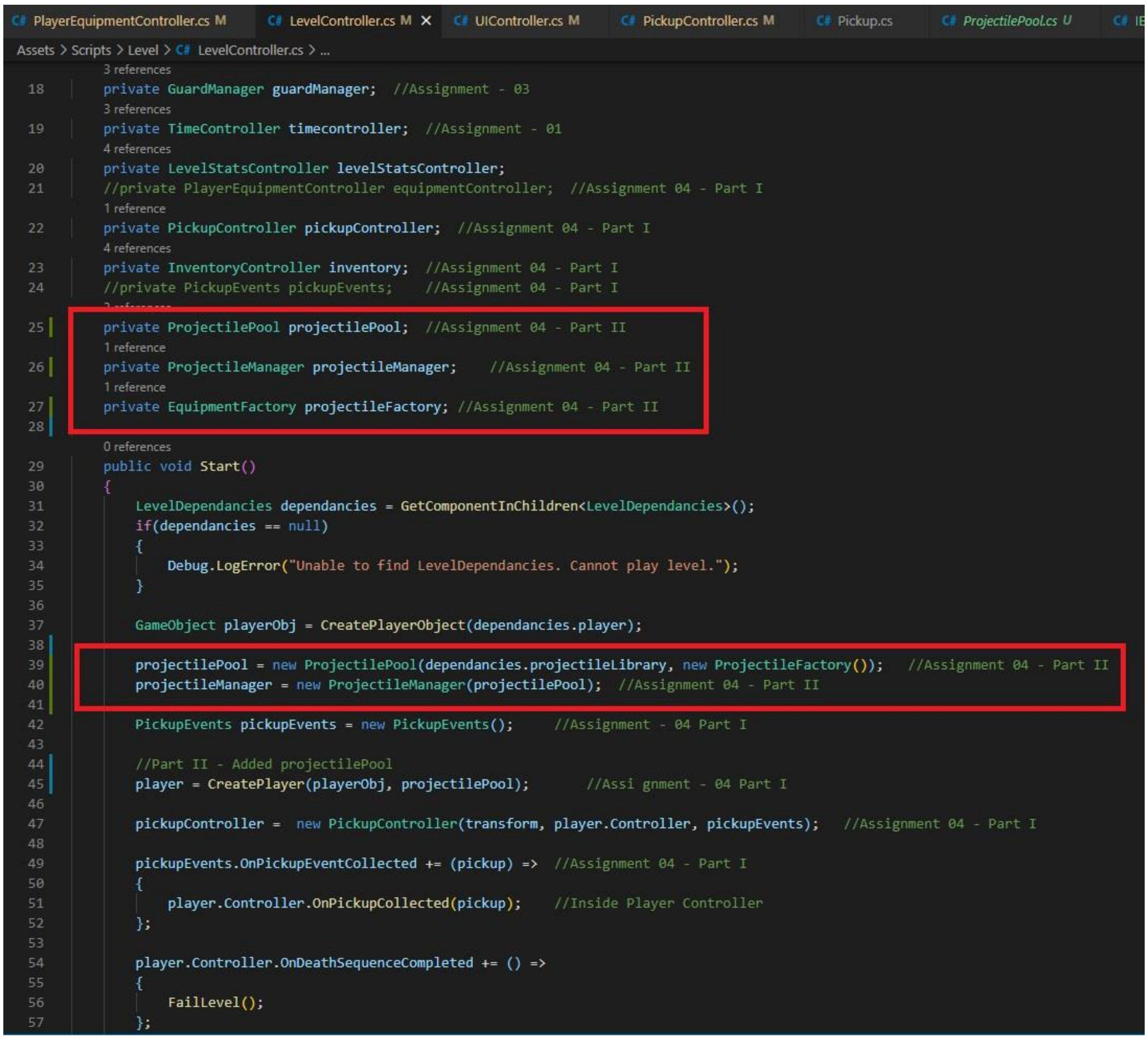
Player > PlayerController.cs > PlayerController > PlayerController(Transform transform, NavMeshAgent navMeshAgent, PlayerInteractionController interactionController, PlayerCollision collision, PlayerInput

```
    2 references
    private PlayerInputBroadcaster inputBroadcaster;
    2 references
    private PlayerViewRelativeMovement viewRelativeMovement;
    1 reference
    private float playerMaxHP = 100f;
    0 references
    private PlayerInputCallbacks callbacks;
    4 references
    private PlayerEquipmentController equipmentController;  //Assignment 04 - Part I
    private InventoryController inventory;  //Assignment 04 - Part II
    2 references
    public ProjectilePool projectilePool;   //Assignment 04 - Part II

    1 reference
    public PlayerController(Transform transform,
        NavMeshAgent navMeshAgent, PlayerInteractionController interactionController,
        PlayerCollision collision, PlayerInputBroadcaster inputBroadcaster,
        InventoryController inventory, PlayerEquipmentController equipmentController, ProjectilePool projectilePool)    //Assignment 04 - Part I & II
    {
        this.transform = transform;
        this.navMeshAgent = navMeshAgent;
        this.interactionController = interactionController;
        this.collision = collision;
        this.inputBroadcaster = inputBroadcaster;
        this.equipmentController = equipmentController;     //Assignment 04 - Part I
        this.inventory = inventory; //Assignment 04 - Part I
        this.projectilePool = projectilePool;   //Assignment 04 - Part II

        playerHealth = new PlayerHealth(playerMaxHP);

        playerHealth.OnDamageTaken += (currentHealth) =>
        {
            OnPlayerDamageTaken(currentHealth);
        };

        playerHealth.OnKilled += () =>
        {
            OnDeathSequenceStarted();
            HandlePlayerDeath();
        };

        rigidbody = transform.GetComponent<Rigidbody>();

        playerObjectData = transform.GetComponent<PlayerObjectData>();

```

❖ **LevelController.**

- We create some instances of the Projectile classes.
- We create an instance of "*ProjectilePool*" here, because we need to pass the data to the "*Dependencies*" components.
- We pass in the "ProjectileLibrary" and a new "ProjectileFactory" which gets created.
- Also, we pass in the "*ProjectileManager*" which controls the projectiles and pass it the "*projectilePool*" variable.

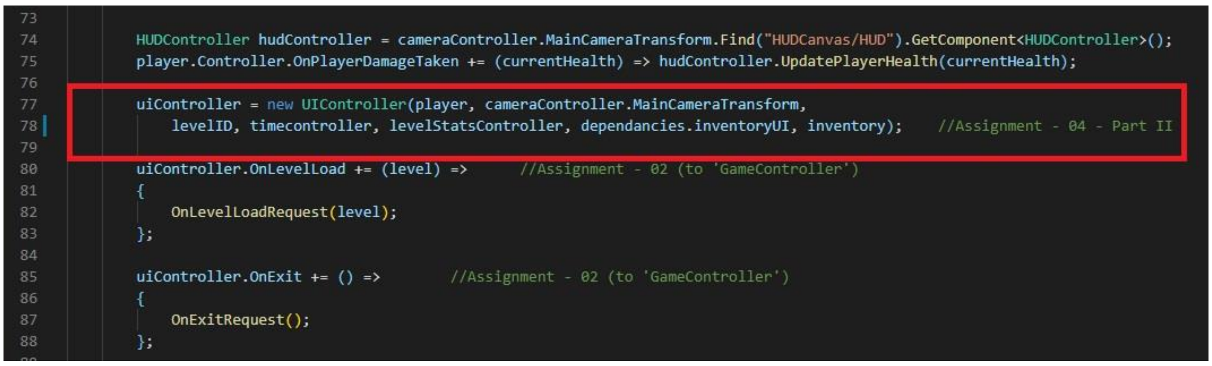
- Then we pass in the "*PlayerEquipmentController*" in the "*return Player()*" function and also in the "*Player*" Class.



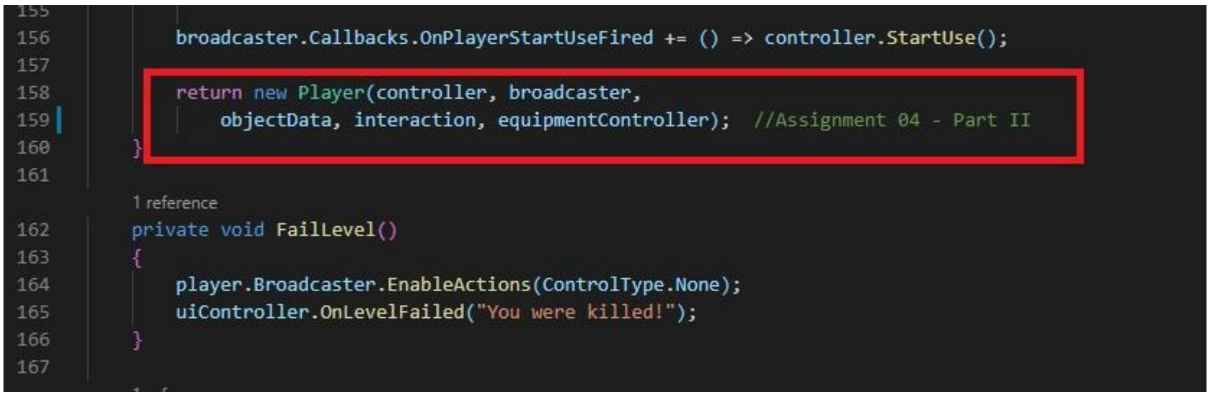
PlayerEquipmentController.cs M | LevelController.cs M × | UIController.cs M | PickupController.cs M | Pickup.cs | ProjectilePool.cs U

Assets > Scripts > Level > LevelController.cs > ...

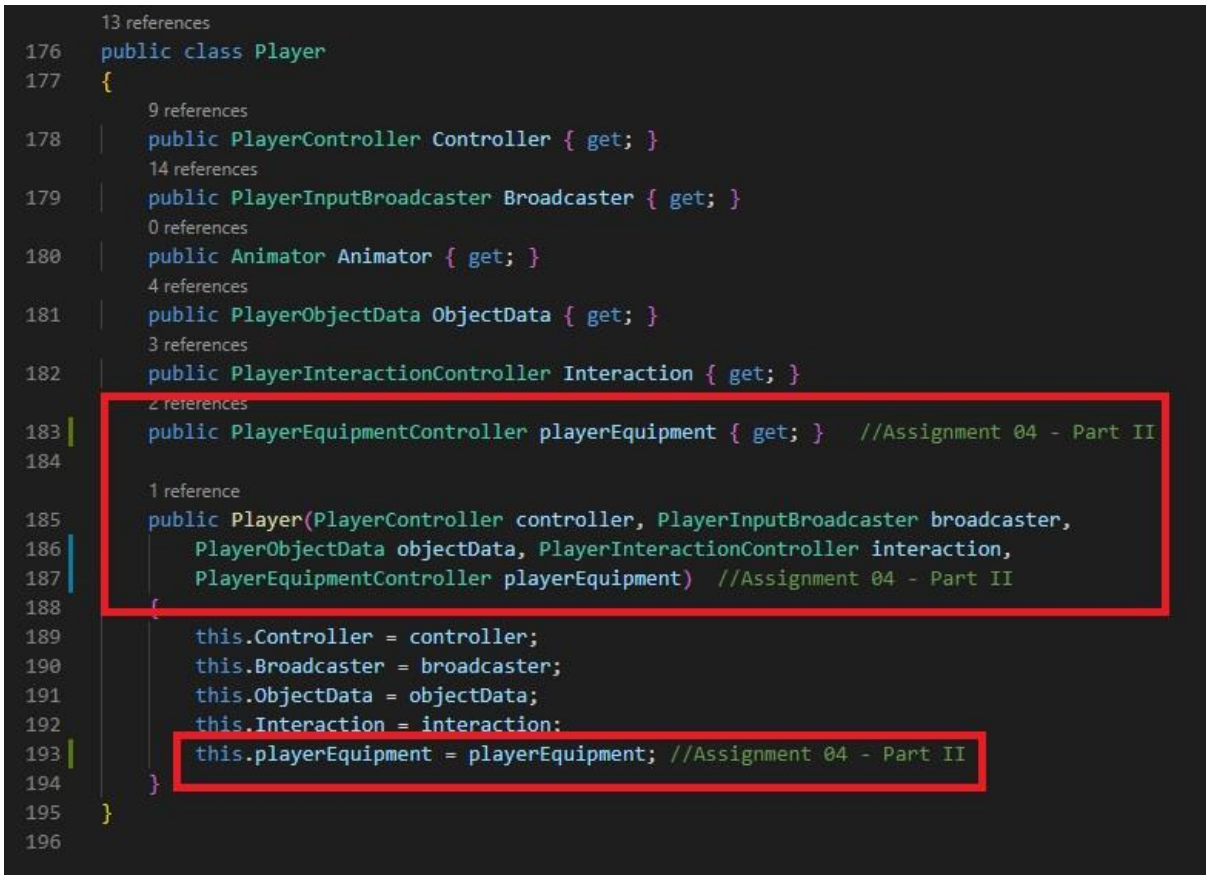
```
    3 references
    private GuardManager guardManager;  //Assignment - 03
    3 references
    private TimeController timecontroller;  //Assignment - 01
    4 references
    private LevelStatsController levelStatsController;
    //private PlayerEquipmentController equipmentController;  //Assignment 04 - Part I
    1 reference
    private PickupController pickupController;  //Assignment 04 - Part I
    4 references
    private InventoryController inventory;  //Assignment 04 - Part I
    //private PickupEvents pickupEvents;    //Assignment 04 - Part I

    private ProjectilePool projectilePool;  //Assignment 04 - Part II
    1 reference
    private ProjectileManager projectileManager;    //Assignment 04 - Part II
    1 reference
    private EquipmentFactory projectileFactory; //Assignment 04 - Part II

    0 references
    public void Start()
    {
        LevelDependancies dependancies = GetComponentInChildren<LevelDependancies>();
        if(dependancies == null)
        {
            Debug.LogError("Unable to find LevelDependancies. Cannot play level.");
        }

        GameObject playerObj = CreatePlayerObject(dependancies.player);

        projectilePool = new ProjectilePool(dependancies.projectileLibrary, new ProjectileFactory());   //Assignment 04 - Part II
        projectileManager = new ProjectileManager(projectilePool);  //Assignment 04 - Part II

        PickupEvents pickupEvents = new PickupEvents();     //Assignment - 04 Part I

        //Part II - Added projectilePool
        player = CreatePlayer(playerObj, projectilePool);       //Assi gnment - 04 Part I

        pickupController =  new PickupController(transform, player.Controller, pickupEvents);   //Assignment 04 - Part I

        pickupEvents.OnPickupEventCollected += (pickup) =>  //Assignment 04 - Part I
        {
            player.Controller.OnPickupCollected(pickup);    //Inside Player Controller
        };

        player.Controller.OnDeathSequenceCompleted += () =>
        {
            FailLevel();
        };
```



```
        HUDController hudController = cameraController.MainCameraTransform.Find("HUDCanvas/HUD").GetComponent<HUDController>();
        player.Controller.OnPlayerDamageTaken += (currentHealth) => hudController.UpdatePlayerHealth(currentHealth);

        uiController = new UIController(player, cameraController.MainCameraTransform,
            levelID, timecontroller, levelStatsController, dependancies.inventoryUI, inventory);    //Assignment - 04 - Part II

        uiController.OnLevelLoad += (level) =>      //Assignment - 02 (to 'GameController')
        {
            OnLevelLoadRequest(level);
        };

        uiController.OnExit += () =>        //Assignment - 02 (to 'GameController')
        {
            OnExitRequest();
        };
```



```
        broadcaster.Callbacks.OnPlayerStartUseFired += () => controller.StartUse();

        return new Player(controller, broadcaster,
            objectData, interaction, equipmentController);  //Assignment 04 - Part II
    }

    1 reference
    private void FailLevel()
    {
        player.Broadcaster.EnableActions(ControlType.None);
        uiController.OnLevelFailed("You were killed!");
    }

```

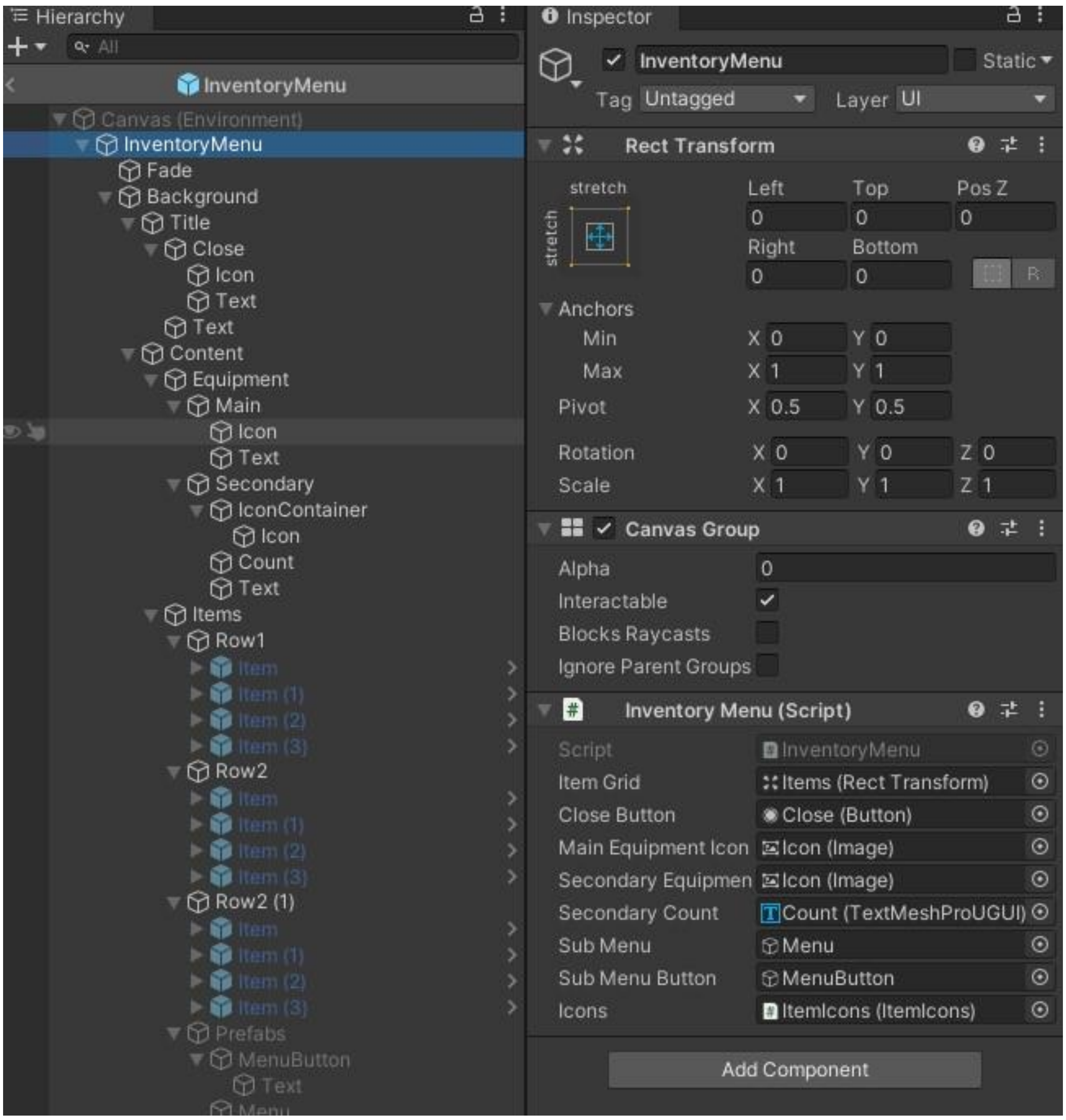


```
13 references
public class Player
{
    9 references
    public PlayerController Controller { get; }
    14 references
    public PlayerInputBroadcaster Broadcaster { get; }
    0 references
    public Animator Animator { get; }
    4 references
    public PlayerObjectData ObjectData { get; }
    3 references
    public PlayerInteractionController Interaction { get; }
    2 references
    public PlayerEquipmentController playerEquipment { get; }   //Assignment 04 - Part II

    1 reference
    public Player(PlayerController controller, PlayerInputBroadcaster broadcaster,
        PlayerObjectData objectData, PlayerInteractionController interaction,
        PlayerEquipmentController playerEquipment)  //Assignment 04 - Part II
    {
        this.Controller = controller;
        this.Broadcaster = broadcaster;
        this.ObjectData = objectData;
        this.Interaction = interaction;
        this.playerEquipment = playerEquipment; //Assignment 04 - Part II
    }
}

```

# Part II – Inventory Menu UI

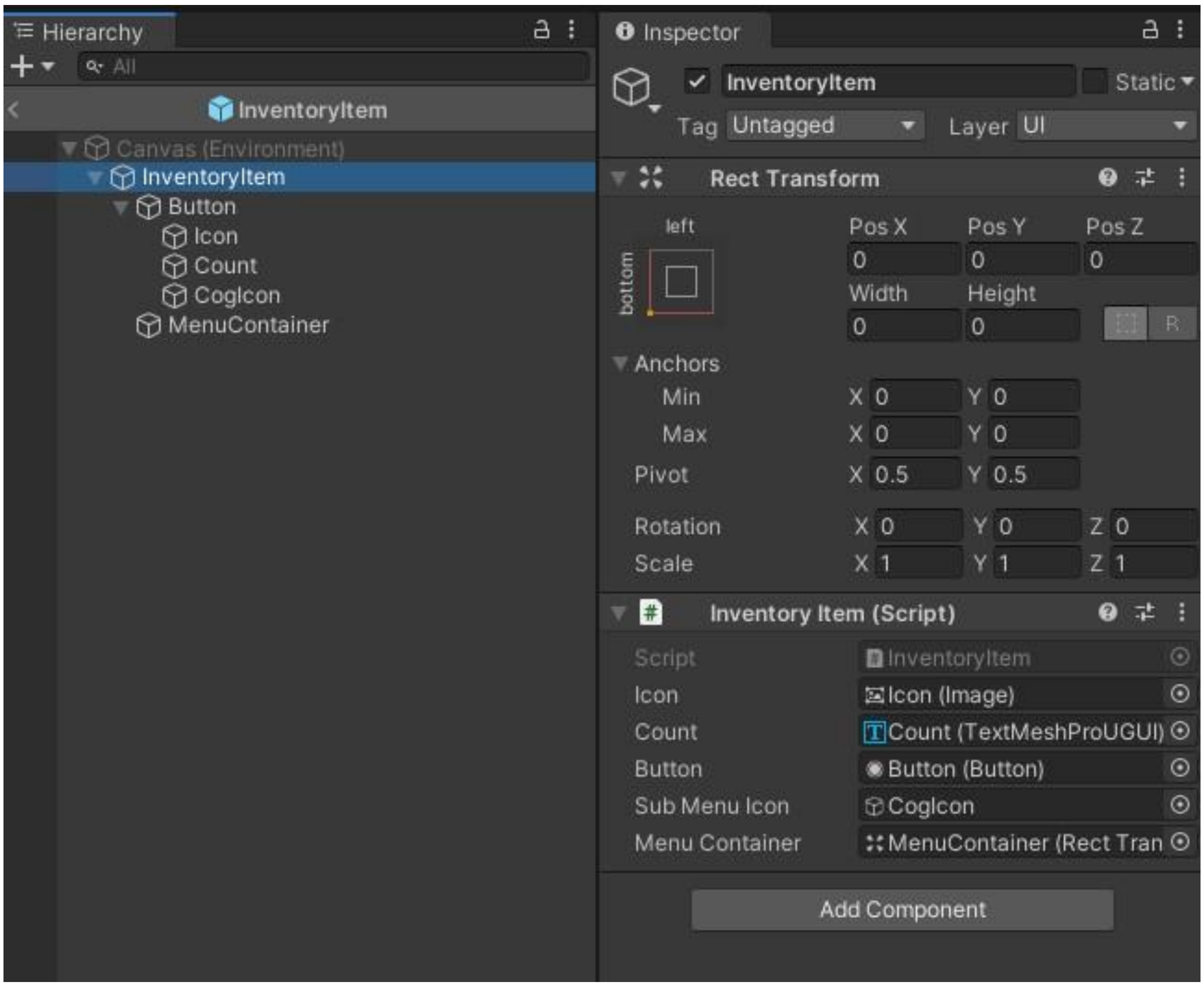
## Step 01: Setup

- There is already an "*InventoryUI*" created in the Prefabs > UI > InventoryMenu.
- Open the prefab, and set the "Alpha" value to 1, to make it visible.
- Then, we create scripts and attach them to the components.
- The Main Parent Script is the "*InventoryMenu*" script.
- After creating all the scripts, we assign them as follow:



```
Hierarchy
InventoryMenu
Canvas (Environment)
  InventoryMenu
    Fade
    Background
      Title
        Close
          Icon
          Text
        Text
      Content
        Equipment
          Main
            Icon
            Text
          Secondary
            IconContainer
              Icon
            Count
            Text
        Items
          Row1
            Item
            Item (1)
            Item (2)
            Item (3)
          Row2
            Item
            Item (1)
            Item (2)
            Item (3)
          Row2 (1)
            Item
            Item (1)
            Item (2)
            Item (3)
      Prefabs
        MenuButton
          Text
        Menu
```

```
Inspector
InventoryMenu    Static
Tag Untagged    Layer UI

Rect Transform
stretch
Left 0    Top 0    Pos Z 0
Right 0   Bottom 0
Anchors
  Min   X 0    Y 0
  Max   X 1    Y 1
Pivot   X 0.5  Y 0.5
Rotation X 0   Y 0   Z 0
Scale    X 1   Y 1   Z 1

Canvas Group
Alpha                 0
Interactable          ✓
Blocks Raycasts
Ignore Parent Groups

Inventory Menu (Script)
Script                  InventoryMenu
Item Grid               Items (Rect Transform)
Close Button            Close (Button)
Main Equipment Icon     Icon (Image)
Secondary Equipmen      Icon (Image)
Secondary Count         Count (TextMeshProUGUI)
Sub Menu                Menu
Sub Menu Button         MenuButton
Icons                   ItemIcons (ItemIcons)

Add Component
```



Hierarchy

All

InventoryItem

- Canvas (Environment)
  - InventoryItem
    - Button
      - Icon
      - Count
      - CogIcon
    - MenuContainer

Inspector

InventoryItem  Static

Tag Untagged  Layer UI

Rect Transform

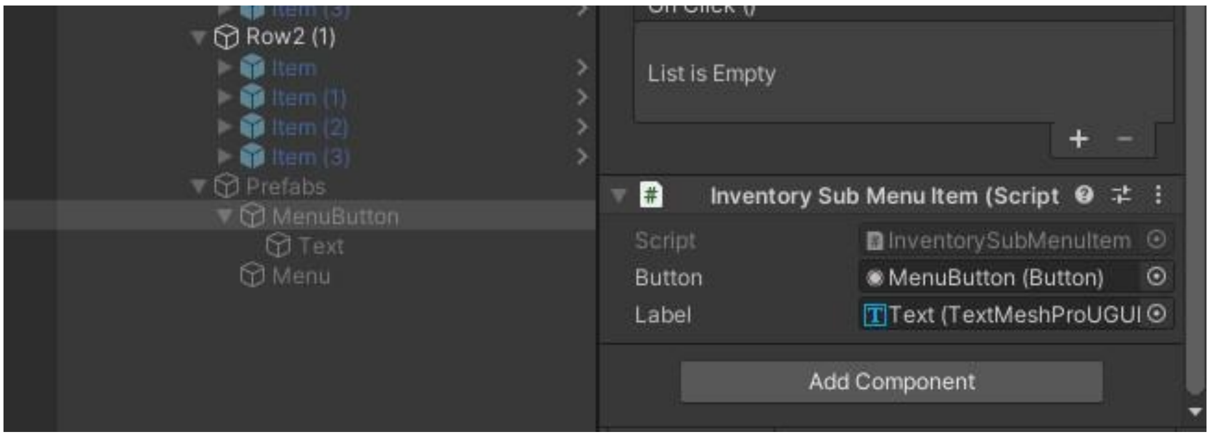
| | | | |
|---|---|---|---|
| | Pos X | Pos Y | Pos Z |
| | 0 | 0 | 0 |
| | Width | Height | |
| | 0 | 0 | |
| Anchors | | | |
| Min | X 0 | Y 0 | |
| Max | X 0 | Y 0 | |
| Pivot | X 0.5 | Y 0.5 | |
| Rotation | X 0 | Y 0 | Z 0 |
| Scale | X 1 | Y 1 | Z 1 |

Inventory Item (Script)

| | |
|---|---|
| Script | InventoryItem |
| Icon | Icon (Image) |
| Count | Count (TextMeshProUGUI) |
| Button | Button (Button) |
| Sub Menu Icon | CogIcon |
| Menu Container | MenuContainer (Rect Tran |

Add Component

(For each Item in Row1, Row2 and Row3)



- Row2 (1)
  - Item
  - Item (1)
  - Item (2)
  - Item (3)
- Prefabs
  - MenuButton
    - Text
  - Menu

List is Empty

Inventory Sub Menu Item (Script

| | |
|---|---|
| Script | InventorySubMenuItem |
| Button | MenuButton (Button) |
| Label | Text (TextMeshProUGUI |

Add Component

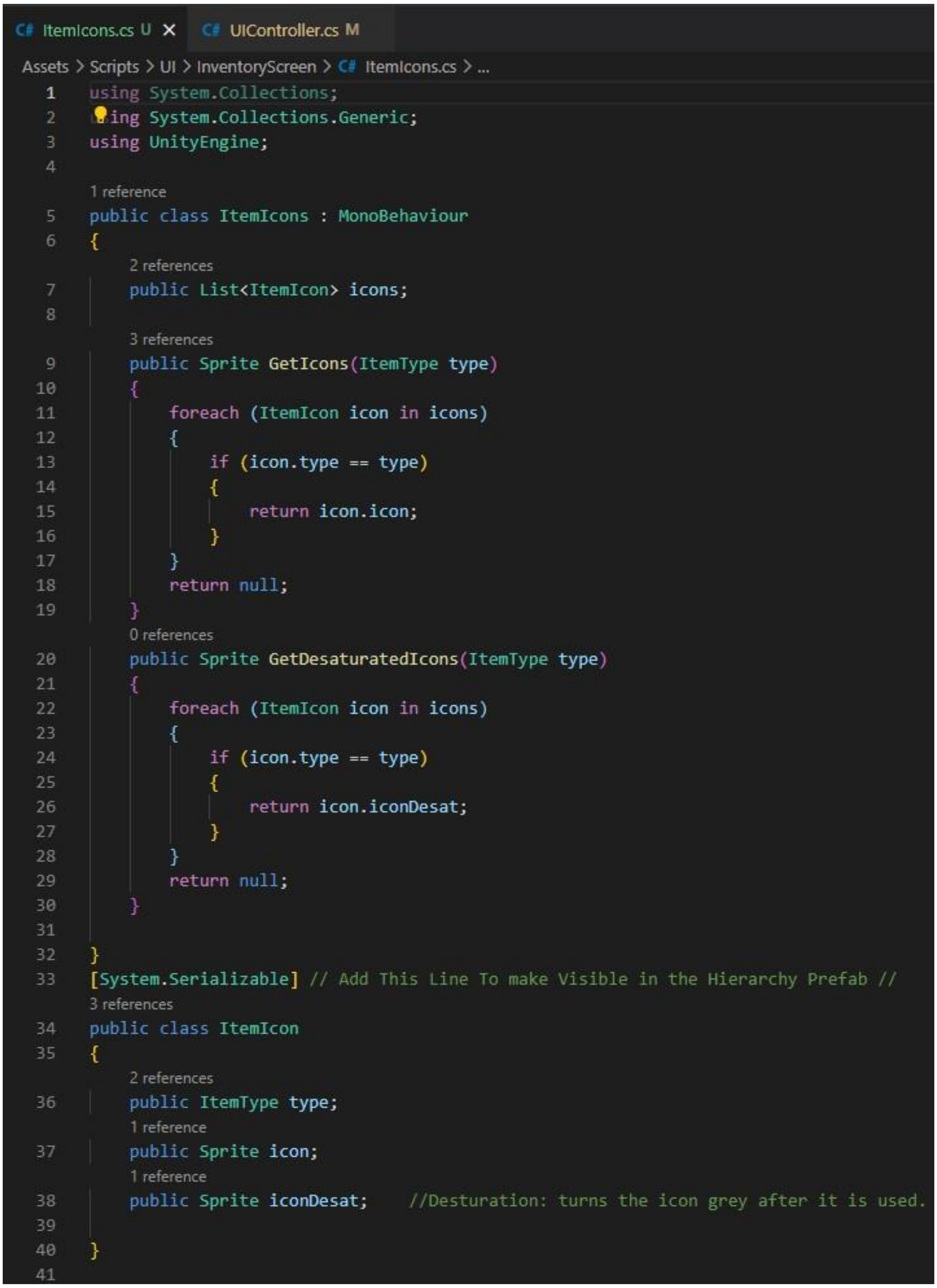
## Step 02: Script & Workflow

- The Script Workflow for Inventory Menu:
    - UIController > InventoryMenuController > InventorySubMenuController, InventoryMenu, InventoryItem, InventoryController, PlayerEquipmentController.
    - InventorySubMenuController > Entry (Sub Class within it) > InventorySubMenuItem.
    - InventoryMenu > ItemIcons (ItemIcon Sub Class).

- InventoryMenu is the Main Script attached to the Main Parent Inventory UI GameObject.
- InventoryItem, InventoryMenu, InventorySubMenuItem, ItemIcons are all MonoBehvior scripts and are attached to respective elements of the Main Parent Inventory UI GameObject.

❖ **ItemIcons.**

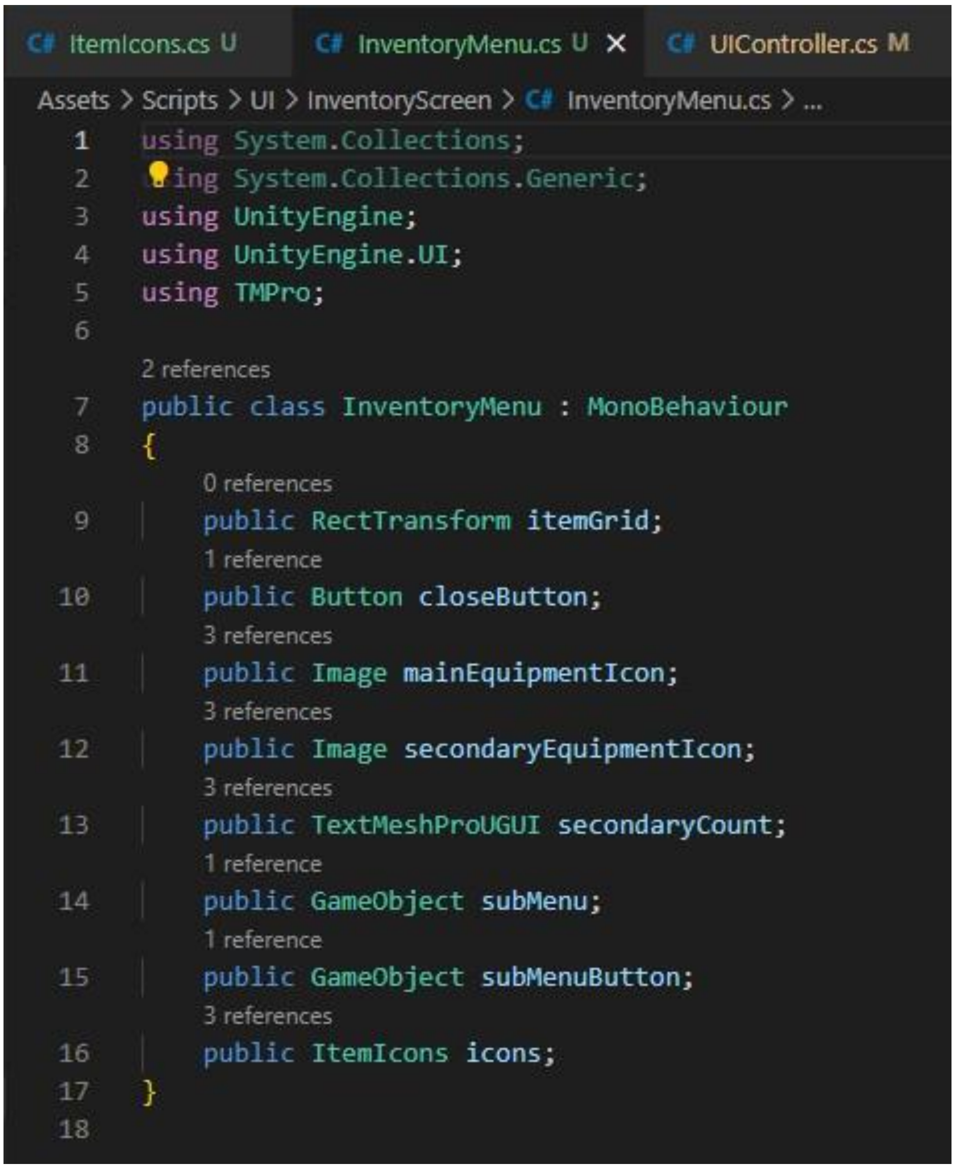
- This class simply consists of all the Pickup Item's Icons.
- It contains a List and also has functions: "*GetIcons*" which is used to get the Normal Icons, and "*GetDesaturatedIcons*" which is used to get the Desaturated form of the Icons.
- It also has a Sub Class which contains the basic elements, and these elements are later called in the above functions to set their images and make them access in other scripts respectively.


ItemIcons.cs U
UIController.cs M
Assets > Scripts > UI > InventoryScreen > ItemIcons.cs > ...

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

1 reference
public class ItemIcons : MonoBehaviour
{
    2 references
    public List<ItemIcon> icons;

    3 references
    public Sprite GetIcons(ItemType type)
    {
        foreach (ItemIcon icon in icons)
        {
            if (icon.type == type)
            {
                return icon.icon;
            }
        }
        return null;
    }
    0 references
    public Sprite GetDesaturatedIcons(ItemType type)
    {
        foreach (ItemIcon icon in icons)
        {
            if (icon.type == type)
            {
                return icon.iconDesat;
            }
        }
        return null;
    }

}
[System.Serializable] // Add This Line To make Visible in the Hierarchy Prefab //
3 references
public class ItemIcon
{
    2 references
    public ItemType type;
    1 reference
    public Sprite icon;
    1 reference
    public Sprite iconDesat;    //Desturation: turns the icon grey after it is used.

}
```

❖ **InventoryMenu.**

- It is the Main Script which is attached to the Main Parent Object of the "*Inventory Menu UI*"
- It consists of all the UI Sprites, which are then further accessed by other scripts.

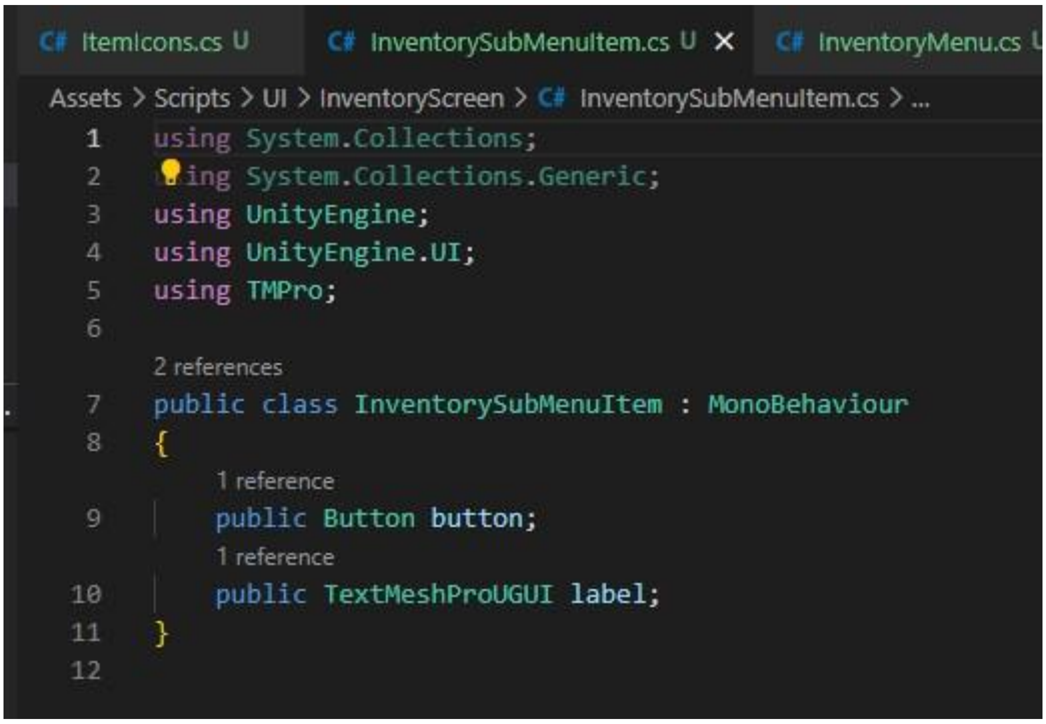


C# ItemIcons.cs U
C# InventoryMenu.cs U ×
C# UIController.cs M
Assets > Scripts > UI > InventoryScreen > C# InventoryMenu.cs > ...

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using TMPro;

2 references
public class InventoryMenu : MonoBehaviour
{
    0 references
    public RectTransform itemGrid;
    1 reference
    public Button closeButton;
    3 references
    public Image mainEquipmentIcon;
    3 references
    public Image secondaryEquipmentIcon;
    3 references
    public TextMeshProUGUI secondaryCount;
    1 reference
    public GameObject subMenu;
    1 reference
    public GameObject subMenuButton;
    3 references
    public ItemIcons icons;
}

```

❖ **InventorySubMenuItem.**

- This script consists of a String and Button which is used and accessed by other scripts.
- It is used to create a button with a label or name in the Inventory Menu UI.

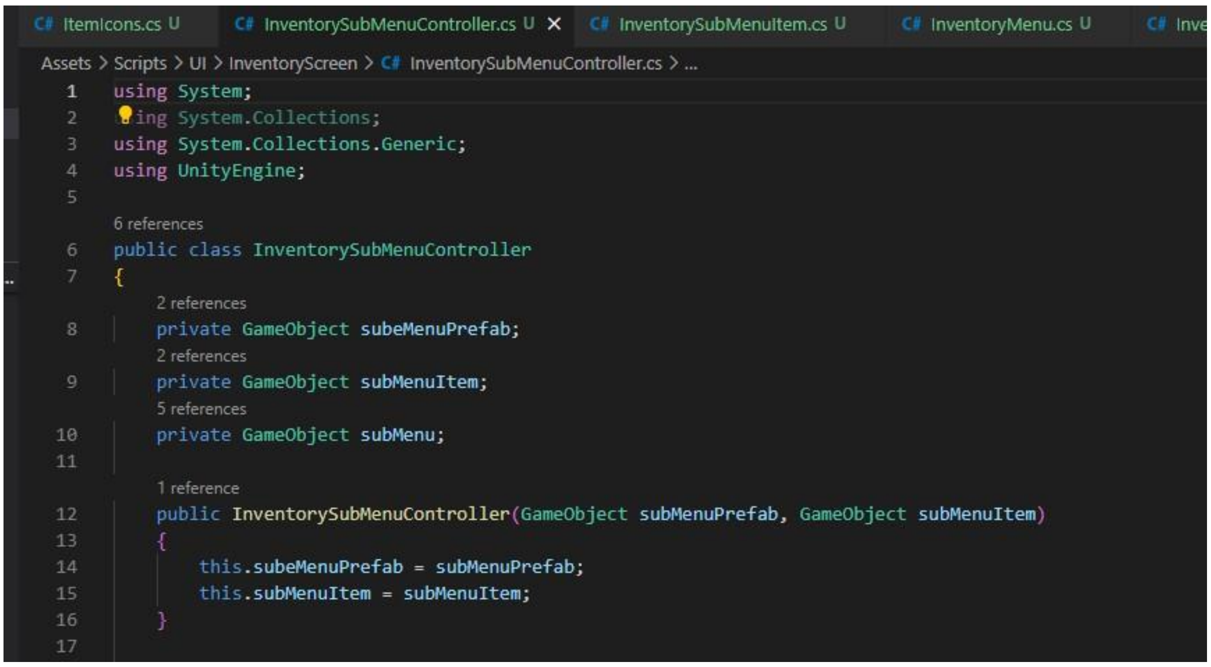

C# ItemIcons.cs U
C# InventorySubMenuItem.cs U
C# InventoryMenu.cs
Assets > Scripts > UI > InventoryScreen > C# InventorySubMenuItem.cs > ...

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using TMPro;

2 references
public class InventorySubMenuItem : MonoBehaviour
{
    1 reference
    public Button button;
    1 reference
    public TextMeshProUGUI label;
}
```

❖ **InventorySubMenuController.**

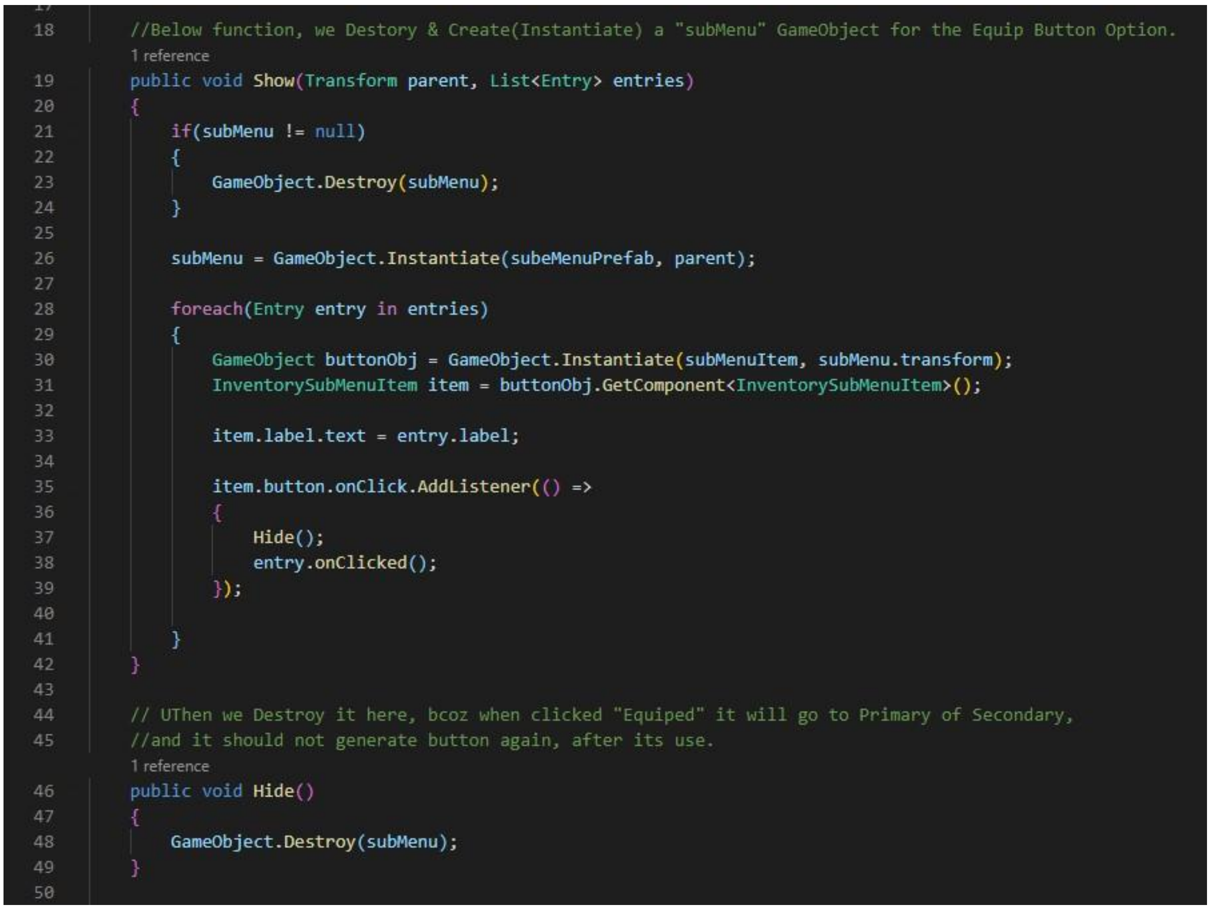
- This script takes the Button & String values of the "*InventorySubMenuItem*" Script and make the Instantiate in the Inventory Menu UI.
- Its "*Show()*" function is responsible for creating or generating the Button(s) and "*Hide()*" function which is used to hide the buttons once clicked.

- It also has a Sub Class called "*Entry*" which contains a String and an Action event called "*OnClicked*" which is then accessed in further scripts.

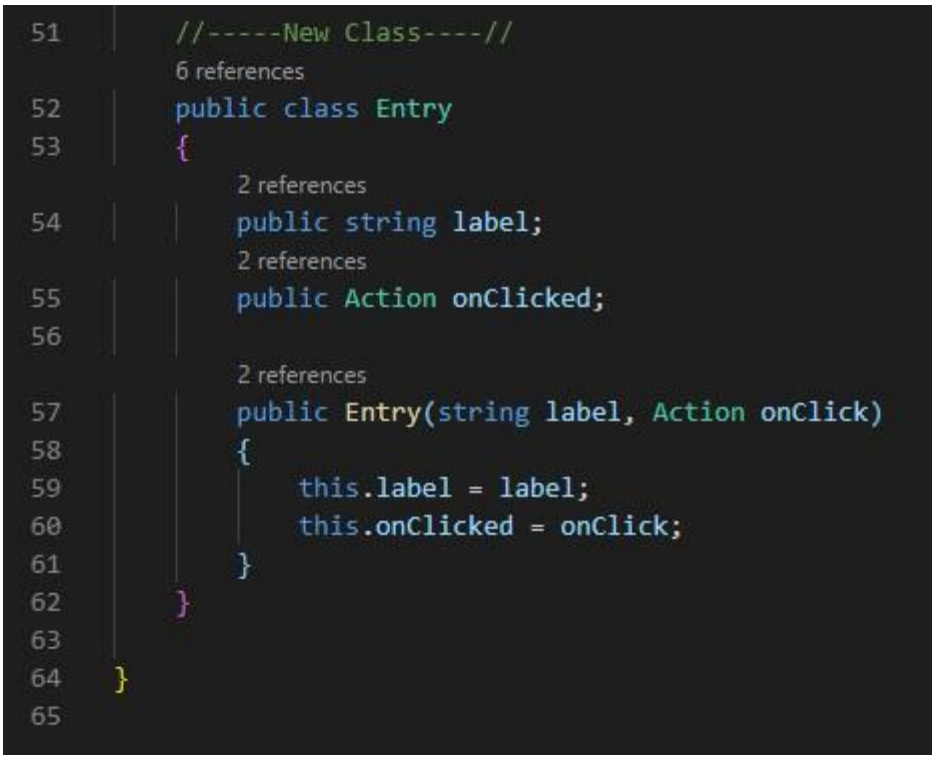


C# ItemIcons.cs U
C# InventorySubMenuController.cs U ×
C# InventorySubMenuItem.cs U
C# InventoryMenu.cs U
C# Inve
Assets > Scripts > UI > InventoryScreen > C# InventorySubMenuController.cs > ...

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

6 references
public class InventorySubMenuController
{
    2 references
    private GameObject subeMenuPrefab;
    2 references
    private GameObject subMenuItem;
    5 references
    private GameObject subMenu;

    1 reference
    public InventorySubMenuController(GameObject subMenuPrefab, GameObject subMenuItem)
    {
        this.subeMenuPrefab = subMenuPrefab;
        this.subMenuItem = subMenuItem;
    }

```



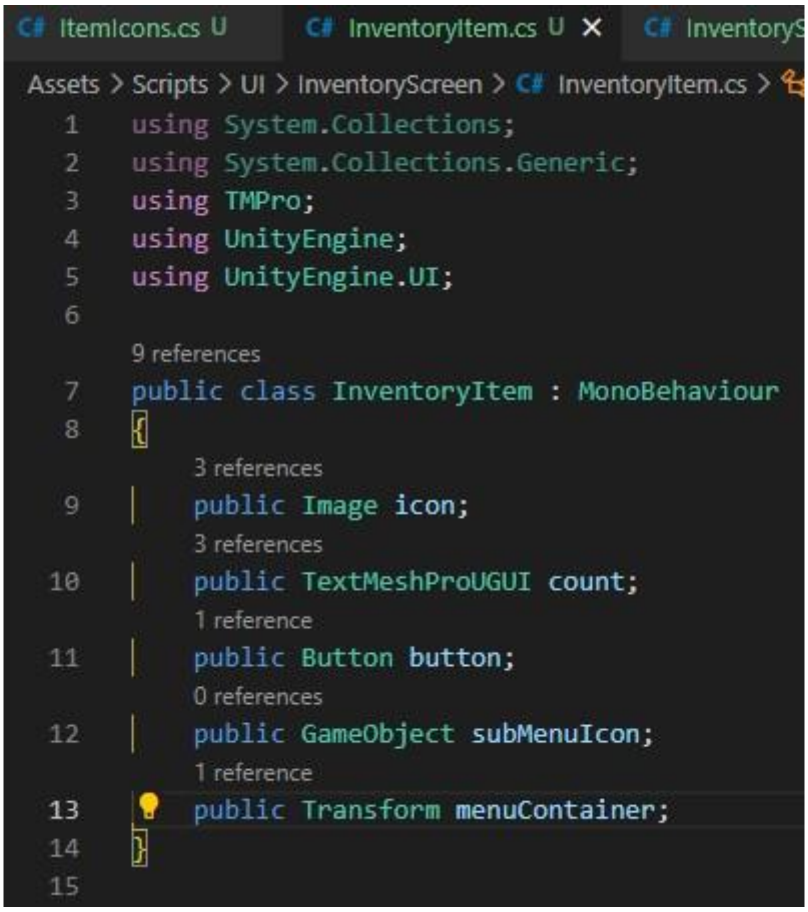
```csharp
    //Below function, we Destory & Create(Instantiate) a "subMenu" GameObject for the Equip Button Option.
    1 reference
    public void Show(Transform parent, List<Entry> entries)
    {
        if(subMenu != null)
        {
            GameObject.Destroy(subMenu);
        }

        subMenu = GameObject.Instantiate(subeMenuPrefab, parent);

        foreach(Entry entry in entries)
        {
            GameObject buttonObj = GameObject.Instantiate(subMenuItem, subMenu.transform);
            InventorySubMenuItem item = buttonObj.GetComponent<InventorySubMenuItem>();

            item.label.text = entry.label;

            item.button.onClick.AddListener(() =>
            {
                Hide();
                entry.onClicked();
            });

        }
    }

    // UThen we Destroy it here, bcoz when clicked "Equiped" it will go to Primary of Secondary,
    //and it should not generate button again, after its use.
    1 reference
    public void Hide()
    {
        GameObject.Destroy(subMenu);
    }

```



```
    //-----New Class----//
    6 references
    public class Entry
    {
        2 references
        public string label;
        2 references
        public Action onClicked;

        2 references
        public Entry(string label, Action onClick)
        {
            this.label = label;
            this.onClicked = onClick;
        }
    }

}

```

❖ **InventoryItem.**

- This script contains all the UI Sprites.
- It is attached to a Prefab called "*ItemIcons*" which is further attached to the main Inventory Menu UI parent object.
- It displays the Image of the Equipment and is attached to each Item in each and every row.



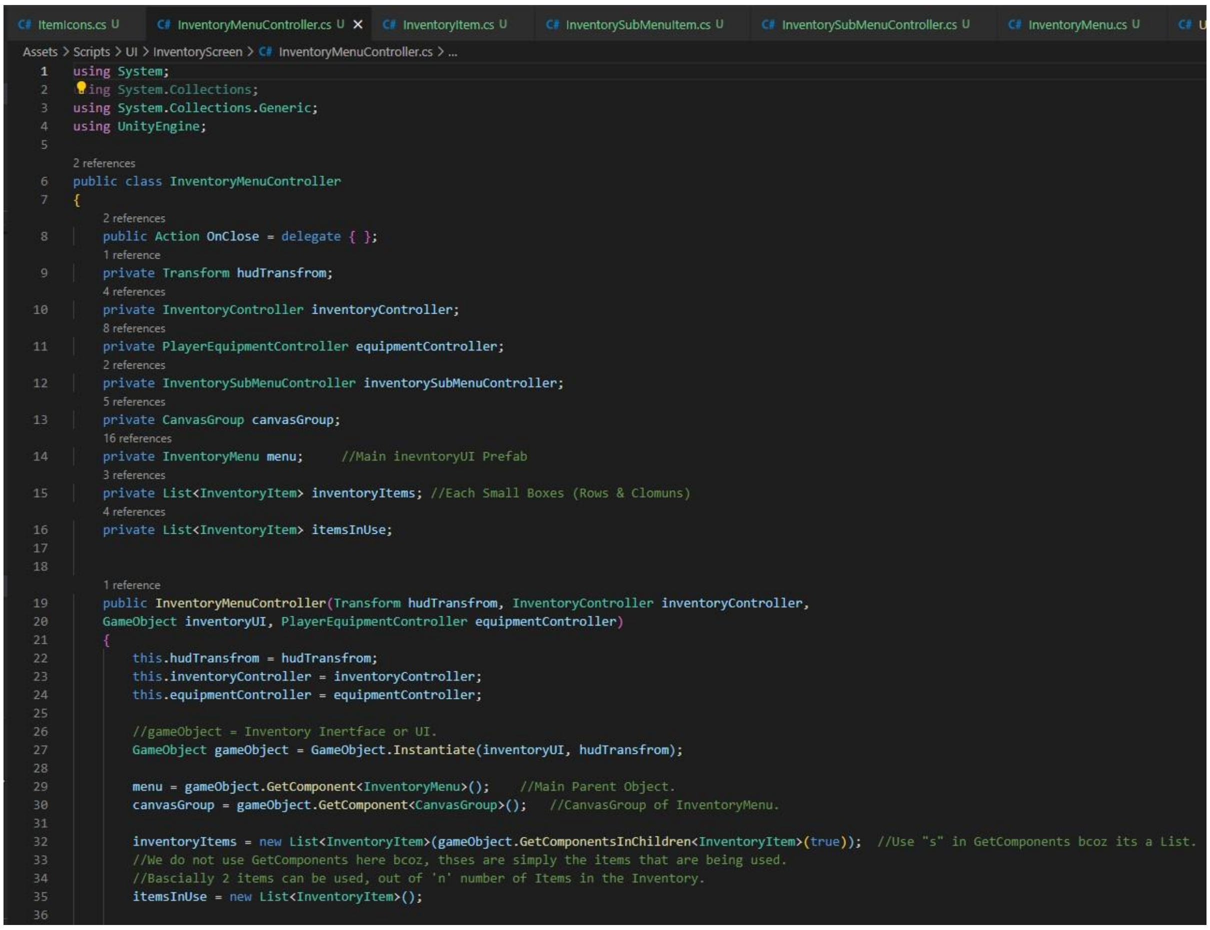
ItemIcons.cs U
InventoryItem.cs U
Inventory
Assets > Scripts > UI > InventoryScreen > InventoryItem.cs >

```
using System.Collections;
using System.Collections.Generic;
using TMPro;
using UnityEngine;
using UnityEngine.UI;

9 references
public class InventoryItem : MonoBehaviour
{
    3 references
    public Image icon;
    3 references
    public TextMeshProUGUI count;
    1 reference
    public Button button;
    0 references
    public GameObject subMenuIcon;
    1 reference
    public Transform menuContainer;
}
```

❖ **InventoryMenuController.**

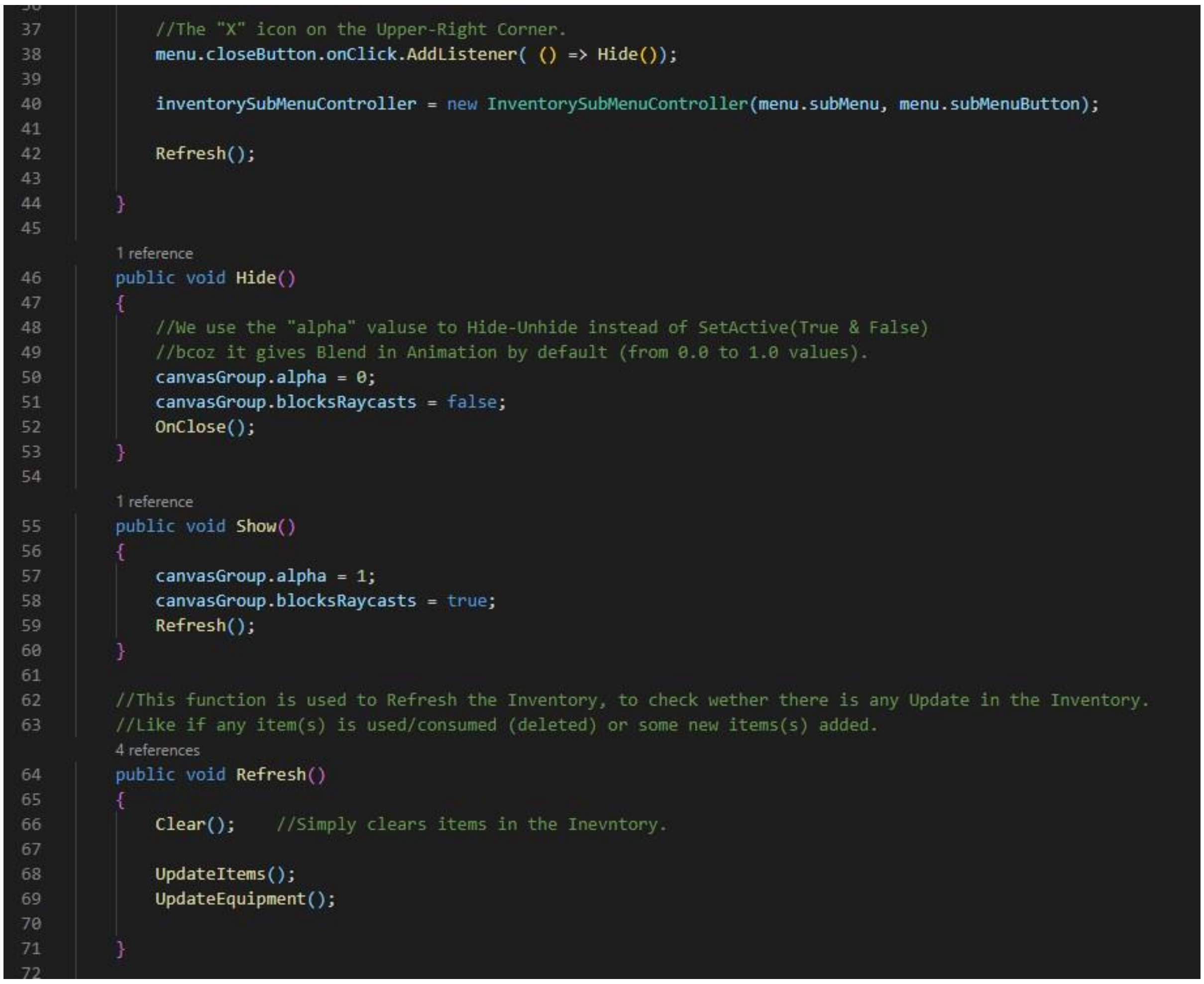
- It is the main parent script which gathers all the other scripts & components and make them work.
- It has a Lists of two ”*Inventorytem*”; “*inventoryItems*” – those items which are present in the Inventory, and “*itemsInUse*” – items that are currently in use.
- It consists of “*Show()*” and “*Hide()*” functions which simply make the InventoryUI Hide & UnHide.
- It has “*Refresh()”* function which refreshes the items present in the Inventory, each time a there is a change in the Inventory, i.e., when an item is Added, Dropped or Equipped as Primary or Secondary in the inventory.
- Also, it has “*Clears()*” function which disables the item when Dropped or been used in the Primary or Secondary slots.

- Then it has the “*UpdateItems()”* and “*UpdateEquipment()”* functions which simply update items & equipments in the inventory.
- The “*UpdateItems()”* function consists the “*Equip*” & “*Drop*” button code as it can be accessed when clicked on each item.

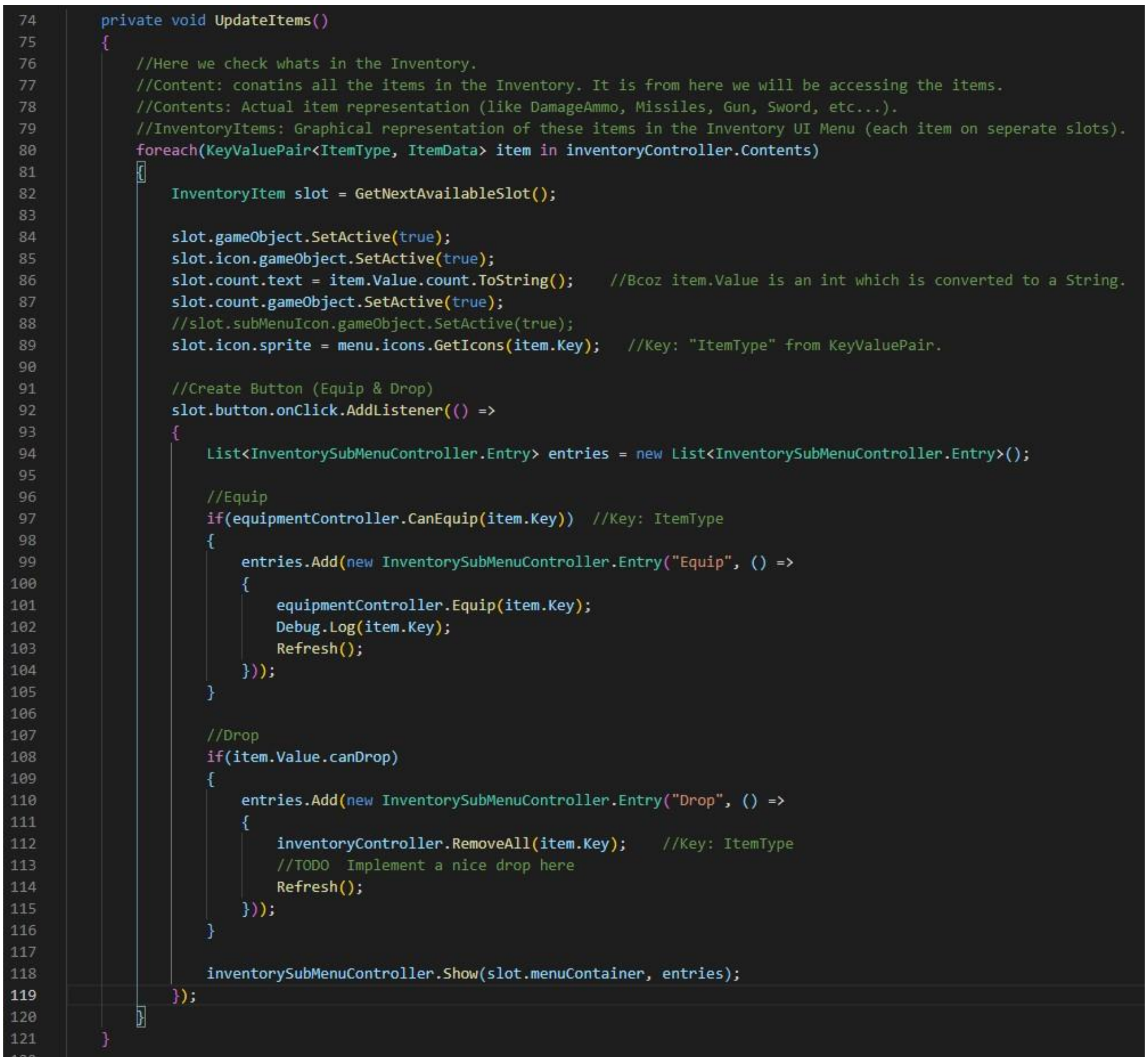

ItemIcons.cs U | InventoryMenuController.cs U | InventoryItem.cs U | InventorySubMenuItem.cs U | InventorySubMenuController.cs U | InventoryMenu.cs U

Assets > Scripts > UI > InventoryScreen > InventoryMenuController.cs > ...

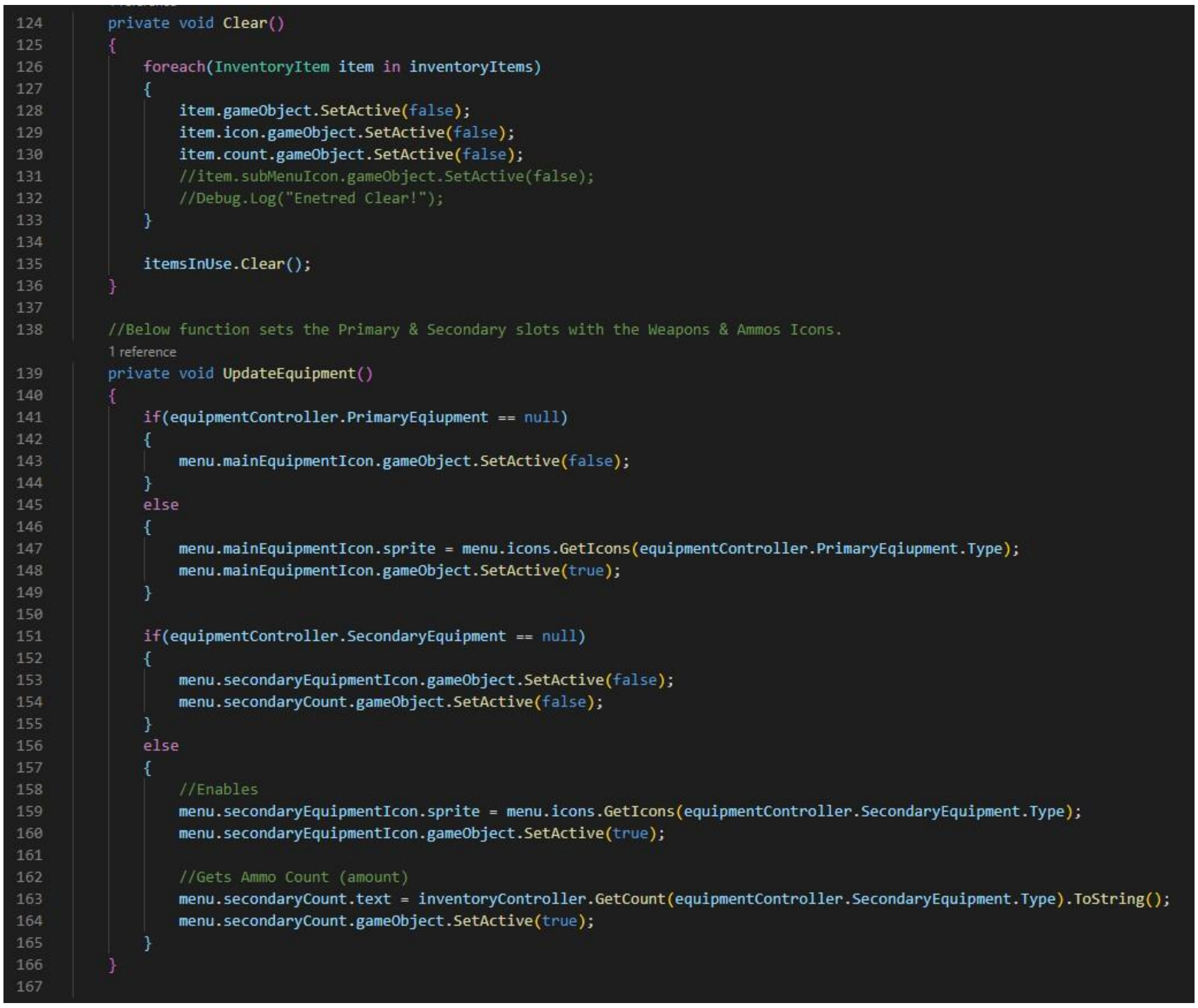
```csharp
using System;
ing System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class InventoryMenuController
{
    public Action OnClose = delegate { };
    private Transform hudTransfrom;
    private InventoryController inventoryController;
    private PlayerEquipmentController equipmentController;
    private InventorySubMenuController inventorySubMenuController;
    private CanvasGroup canvasGroup;
    private InventoryMenu menu;     //Main inevntoryUI Prefab
    private List<InventoryItem> inventoryItems; //Each Small Boxes (Rows & Clomuns)
    private List<InventoryItem> itemsInUse;


    public InventoryMenuController(Transform hudTransfrom, InventoryController inventoryController,
    GameObject inventoryUI, PlayerEquipmentController equipmentController)
    {
        this.hudTransfrom = hudTransfrom;
        this.inventoryController = inventoryController;
        this.equipmentController = equipmentController;

        //gameObject = Inventory Inertface or UI.
        GameObject gameObject = GameObject.Instantiate(inventoryUI, hudTransfrom);

        menu = gameObject.GetComponent<InventoryMenu>();    //Main Parent Object.
        canvasGroup = gameObject.GetComponent<CanvasGroup>();   //CanvasGroup of InventoryMenu.

        inventoryItems = new List<InventoryItem>(gameObject.GetComponentsInChildren<InventoryItem>(true));  //Use "s" in GetComponents bcoz its a List.
        //We do not use GetComponents here bcoz, thses are simply the items that are being used.
        //Bascially 2 items can be used, out of 'n' number of Items in the Inventory.
        itemsInUse = new List<InventoryItem>();

```



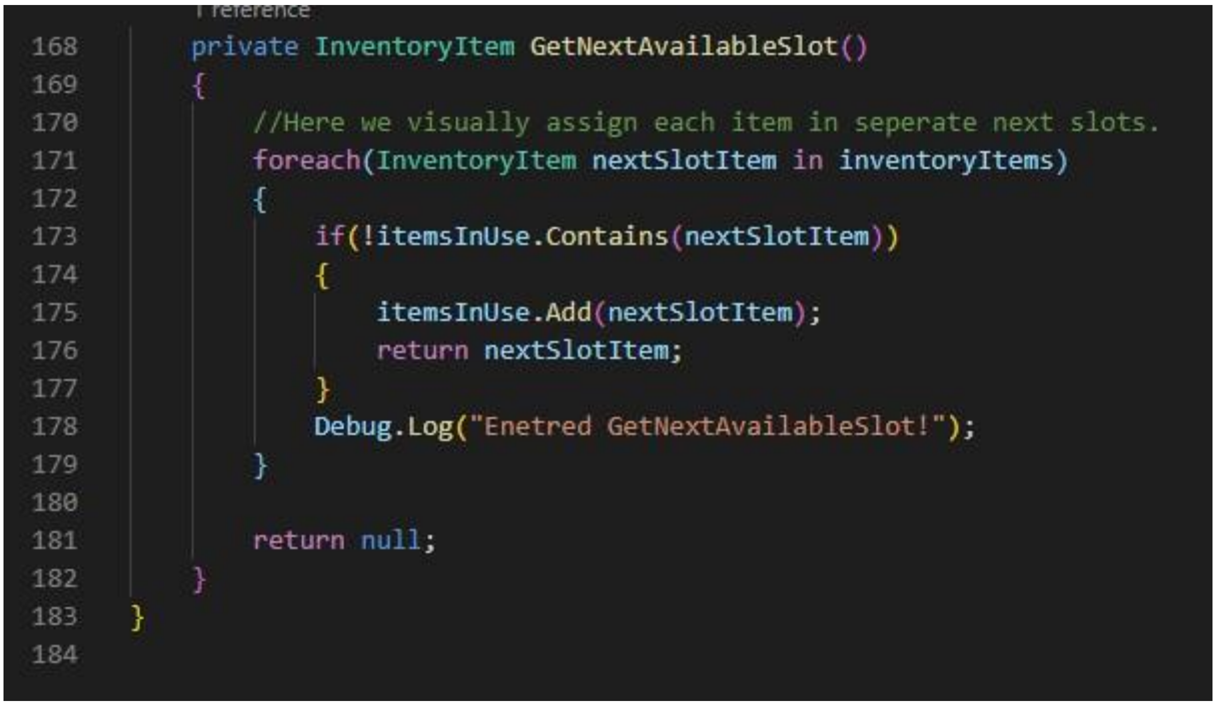
```
        //The "X" icon on the Upper-Right Corner.
        menu.closeButton.onClick.AddListener( () => Hide());

        inventorySubMenuController = new InventorySubMenuController(menu.subMenu, menu.subMenuButton);

        Refresh();

    }

    1 reference
    public void Hide()
    {
        //We use the "alpha" valuse to Hide-Unhide instead of SetActive(True & False)
        //bcoz it gives Blend in Animation by default (from 0.0 to 1.0 values).
        canvasGroup.alpha = 0;
        canvasGroup.blocksRaycasts = false;
        OnClose();
    }

    1 reference
    public void Show()
    {
        canvasGroup.alpha = 1;
        canvasGroup.blocksRaycasts = true;
        Refresh();
    }

    //This function is used to Refresh the Inventory, to check wether there is any Update in the Inventory.
    //Like if any item(s) is used/consumed (deleted) or some new items(s) added.
    4 references
    public void Refresh()
    {
        Clear();    //Simply clears items in the Inevntory.

        UpdateItems();
        UpdateEquipment();

    }
```



```csharp
    private void UpdateItems()
    {
        //Here we check whats in the Inventory.
        //Content: conatins all the items in the Inventory. It is from here we will be accessing the items.
        //Contents: Actual item representation (like DamageAmmo, Missiles, Gun, Sword, etc...).
        //InventoryItems: Graphical representation of these items in the Inventory UI Menu (each item on seperate slots).
        foreach(KeyValuePair<ItemType, ItemData> item in inventoryController.Contents)
        {
            InventoryItem slot = GetNextAvailableSlot();

            slot.gameObject.SetActive(true);
            slot.icon.gameObject.SetActive(true);
            slot.count.text = item.Value.count.ToString();    //Bcoz item.Value is an int which is converted to a String.
            slot.count.gameObject.SetActive(true);
            //slot.subMenuIcon.gameObject.SetActive(true);
            slot.icon.sprite = menu.icons.GetIcons(item.Key);   //Key: "ItemType" from KeyValuePair.

            //Create Button (Equip & Drop)
            slot.button.onClick.AddListener(() =>
            {
                List<InventorySubMenuController.Entry> entries = new List<InventorySubMenuController.Entry>();

                //Equip
                if(equipmentController.CanEquip(item.Key))  //Key: ItemType
                {
                    entries.Add(new InventorySubMenuController.Entry("Equip", () =>
                    {
                        equipmentController.Equip(item.Key);
                        Debug.Log(item.Key);
                        Refresh();
                    }));
                }

                //Drop
                if(item.Value.canDrop)
                {
                    entries.Add(new InventorySubMenuController.Entry("Drop", () =>
                    {
                        inventoryController.RemoveAll(item.Key);    //Key: ItemType
                        //TODO  Implement a nice drop here
                        Refresh();
                    }));
                }

                inventorySubMenuController.Show(slot.menuContainer, entries);
            });
        }
    }
```



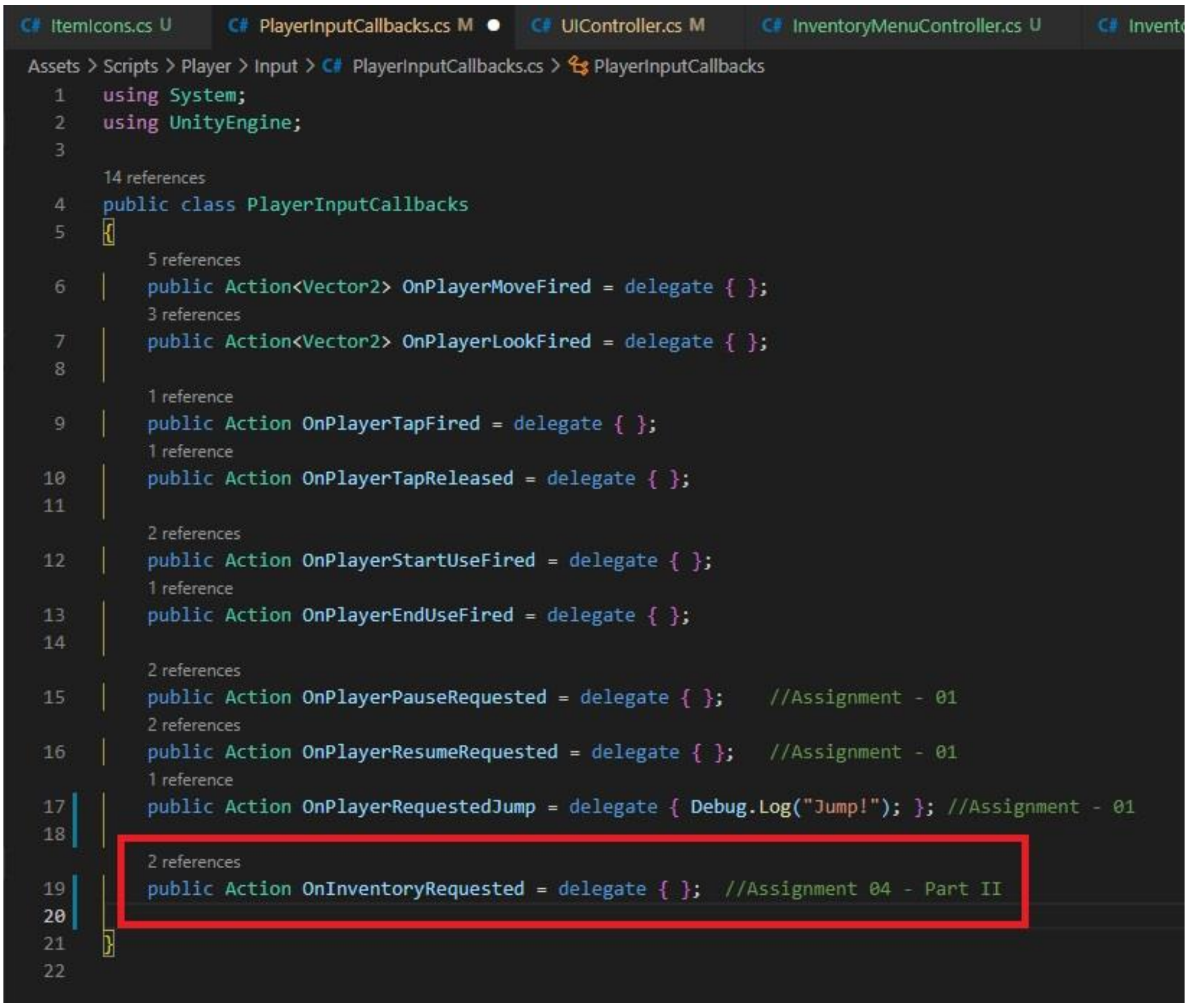
```
        private void Clear()
        {
            foreach(InventoryItem item in inventoryItems)
            {
                item.gameObject.SetActive(false);
                item.icon.gameObject.SetActive(false);
                item.count.gameObject.SetActive(false);
                //item.subMenuIcon.gameObject.SetActive(false);
                //Debug.Log("Enetred Clear!");
            }

            itemsInUse.Clear();
        }

        //Below function sets the Primary & Secondary slots with the Weapons & Ammos Icons.
        1 reference
        private void UpdateEquipment()
        {
            if(equipmentController.PrimaryEqiupment == null)
            {
                menu.mainEquipmentIcon.gameObject.SetActive(false);
            }
            else
            {
                menu.mainEquipmentIcon.sprite = menu.icons.GetIcons(equipmentController.PrimaryEqiupment.Type);
                menu.mainEquipmentIcon.gameObject.SetActive(true);
            }

            if(equipmentController.SecondaryEquipment == null)
            {
                menu.secondaryEquipmentIcon.gameObject.SetActive(false);
                menu.secondaryCount.gameObject.SetActive(false);
            }
            else
            {
                //Enables
                menu.secondaryEquipmentIcon.sprite = menu.icons.GetIcons(equipmentController.SecondaryEquipment.Type);
                menu.secondaryEquipmentIcon.gameObject.SetActive(true);

                //Gets Ammo Count (amount)
                menu.secondaryCount.text = inventoryController.GetCount(equipmentController.SecondaryEquipment.Type).ToString();
                menu.secondaryCount.gameObject.SetActive(true);
            }
        }

```



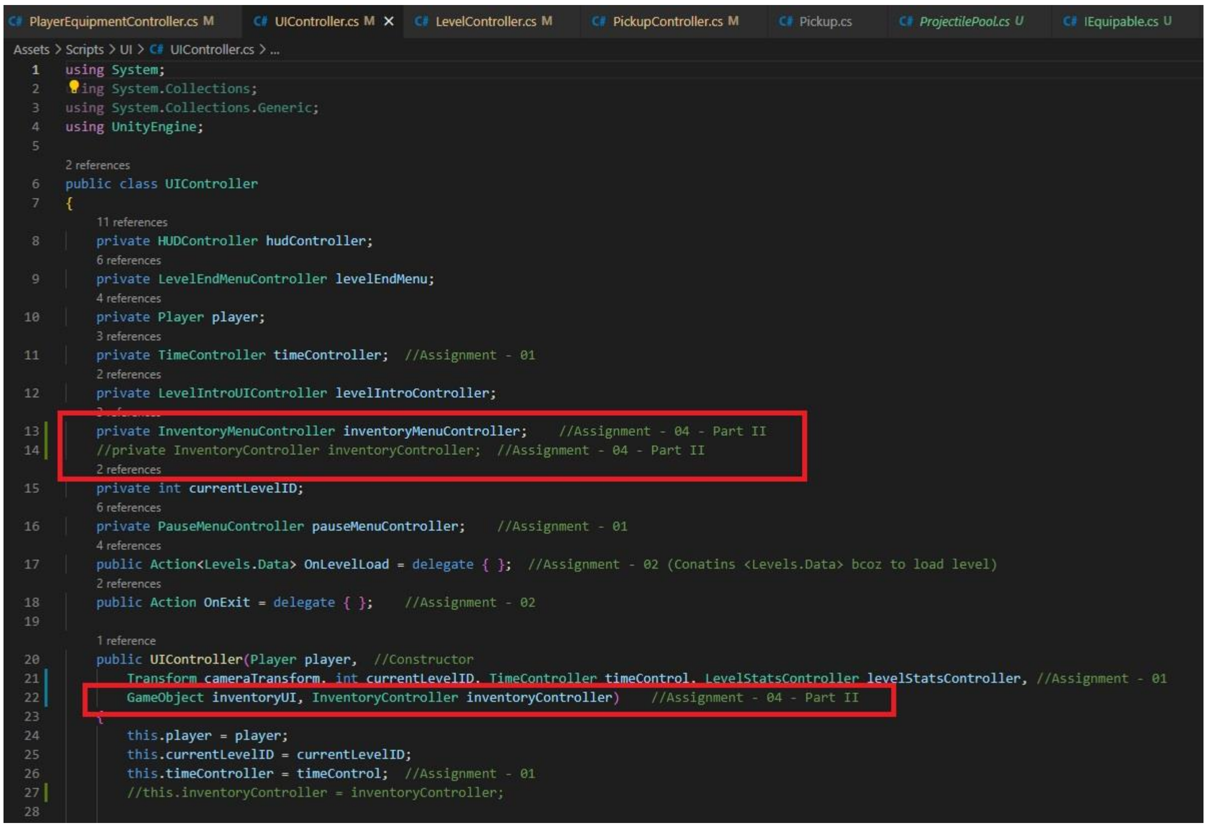
```
        private InventoryItem GetNextAvailableSlot()
        {
            //Here we visually assign each item in seperate next slots.
            foreach(InventoryItem nextSlotItem in inventoryItems)
            {
                if(!itemsInUse.Contains(nextSlotItem))
                {
                    itemsInUse.Add(nextSlotItem);
                    return nextSlotItem;
                }
                Debug.Log("Enetred GetNextAvailableSlot!");
            }

            return null;
        }
    }

```

❖ **UIController.**

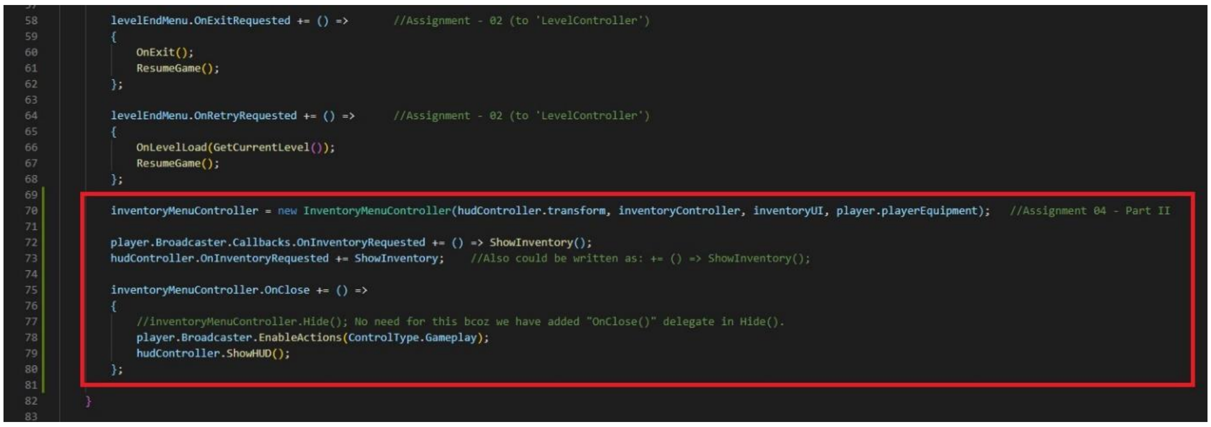
- We simply create an instance of the "*LevelMenuController*" here.
- Then we create function to Show the Inventory Menu UI, as well as to close it..
- We also create an Action delegate in the "*PlayerInputCallBacks*" script called "*OnInventoryRequested*", and pass it in the "*ShowInventory()*" function.


ItemIcons.cs U
PlayerInputCallbacks.cs M
UIController.cs M
InventoryMenuController.cs U
Invento
Assets > Scripts > Player > Input > PlayerInputCallbacks.cs > PlayerInputCallbacks

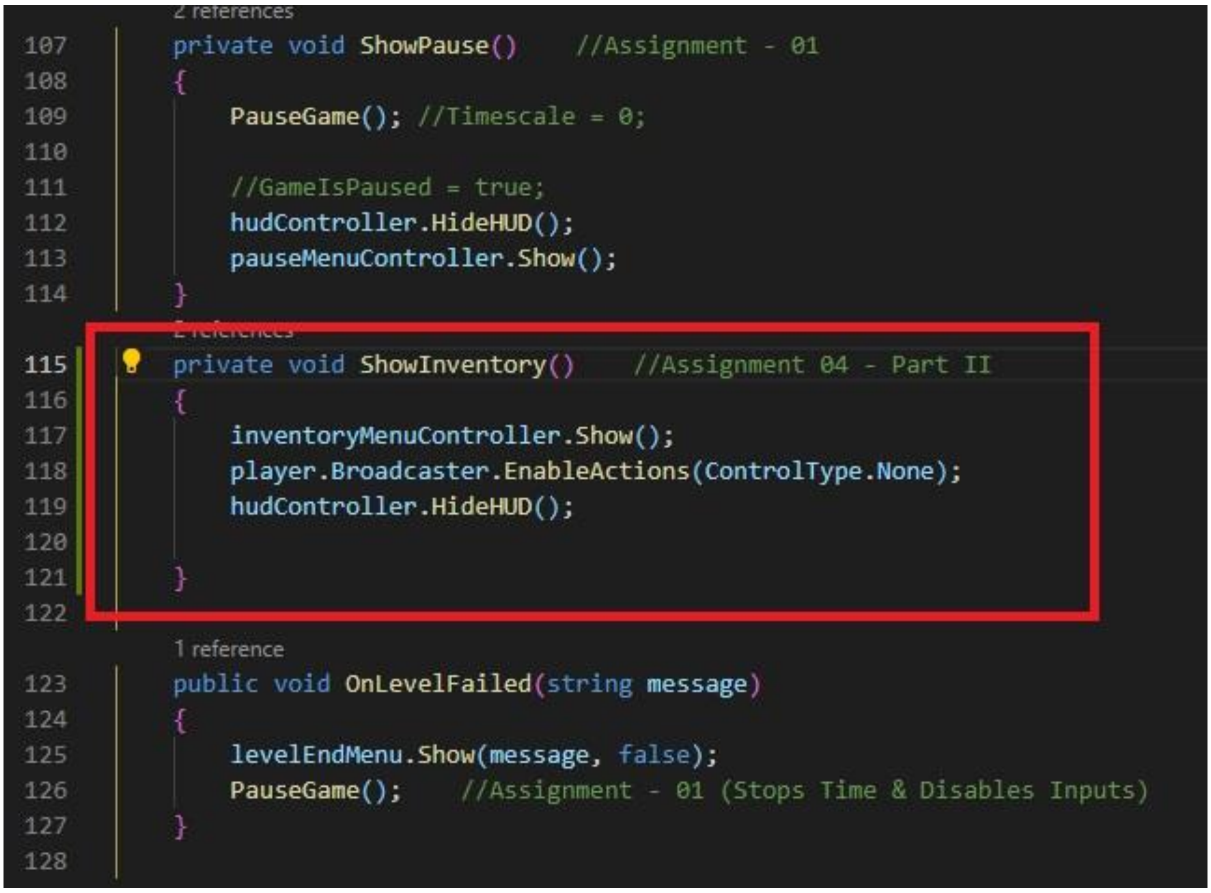
```
using System;
using UnityEngine;

14 references
public class PlayerInputCallbacks
{
    5 references
    public Action<Vector2> OnPlayerMoveFired = delegate { };
    3 references
    public Action<Vector2> OnPlayerLookFired = delegate { };

    1 reference
    public Action OnPlayerTapFired = delegate { };
    1 reference
    public Action OnPlayerTapReleased = delegate { };

    2 references
    public Action OnPlayerStartUseFired = delegate { };
    1 reference
    public Action OnPlayerEndUseFired = delegate { };

    2 references
    public Action OnPlayerPauseRequested = delegate { };    //Assignment - 01
    2 references
    public Action OnPlayerResumeRequested = delegate { };   //Assignment - 01
    1 reference
    public Action OnPlayerRequestedJump = delegate { Debug.Log("Jump!"); }; //Assignment - 01

    2 references
    public Action OnInventoryRequested = delegate { };  //Assignment 04 - Part II

}
```



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

2 references
public class UIController
{
    11 references
    private HUDController hudController;
    6 references
    private LevelEndMenuController levelEndMenu;
    4 references
    private Player player;
    3 references
    private TimeController timeController;  //Assignment - 01
    2 references
    private LevelIntroUIController levelIntroController;
    2 references
    private InventoryMenuController inventoryMenuController;    //Assignment - 04 - Part II
    //private InventoryController inventoryController;  //Assignment - 04 - Part II
    2 references
    private int currentLevelID;
    6 references
    private PauseMenuController pauseMenuController;    //Assignment - 01
    4 references
    public Action<Levels.Data> OnLevelLoad = delegate { };  //Assignment - 02 (Conatins <Levels.Data> bcoz to load level)
    2 references
    public Action OnExit = delegate { };    //Assignment - 02

    1 reference
    public UIController(Player player,  //Constructor
        Transform cameraTransform, int currentLevelID, TimeController timeControl, LevelStatsController levelStatsController, //Assignment - 01
        GameObject inventoryUI, InventoryController inventoryController)    //Assignment - 04 - Part II
    {
        this.player = player;
        this.currentLevelID = currentLevelID;
        this.timeController = timeControl;  //Assignment - 01
        //this.inventoryController = inventoryController;
```

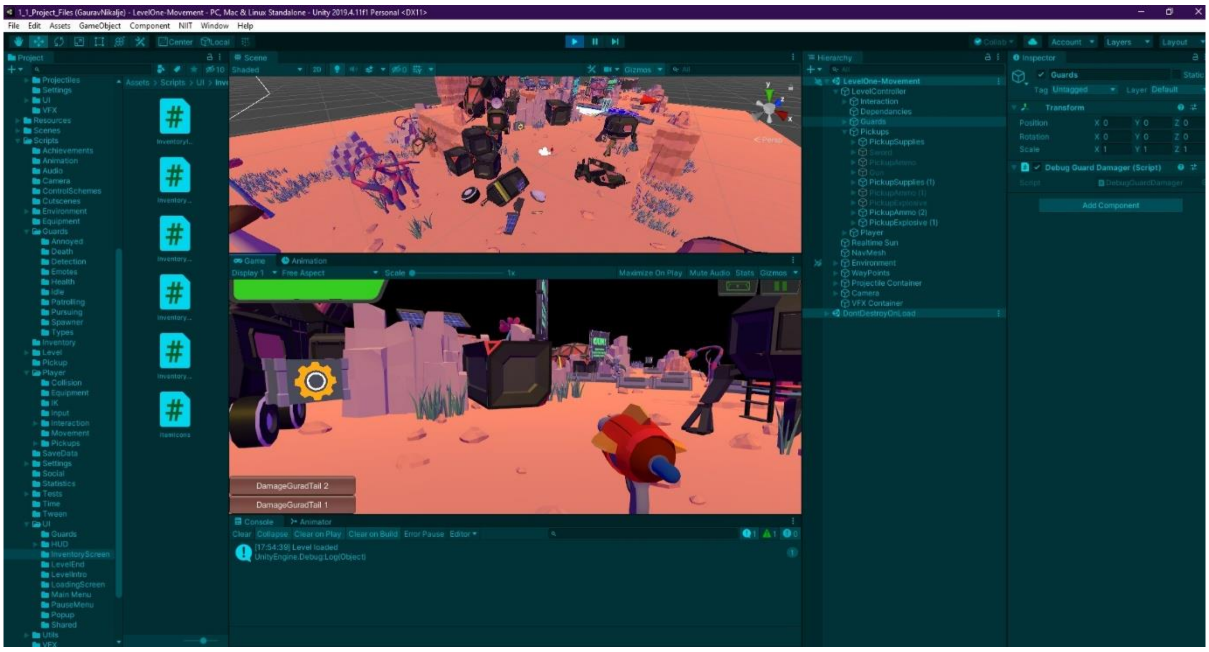


```csharp
        levelEndMenu.OnExitRequested += () =>       //Assignment - 02 (to 'LevelController')
        {
            OnExit();
            ResumeGame();
        };

        levelEndMenu.OnRetryRequested += () =>      //Assignment - 02 (to 'LevelController')
        {
            OnLevelLoad(GetCurrentLevel());
            ResumeGame();
        };

        inventoryMenuController = new InventoryMenuController(hudController.transform, inventoryController, inventoryUI, player.playerEquipment);   //Assignment 04 - Part II

        player.Broadcaster.Callbacks.OnInventoryRequested += () => ShowInventory();
        hudController.OnInventoryRequested += ShowInventory;    //Also could be written as: += () => ShowInventory();

        inventoryMenuController.OnClose += () =>
        {
            //inventoryMenuController.Hide(); No need for this bcoz we have added "OnClose()" delegate in Hide().
            player.Broadcaster.EnableActions(ControlType.Gameplay);
            hudController.ShowHUD();
        };

    }
```



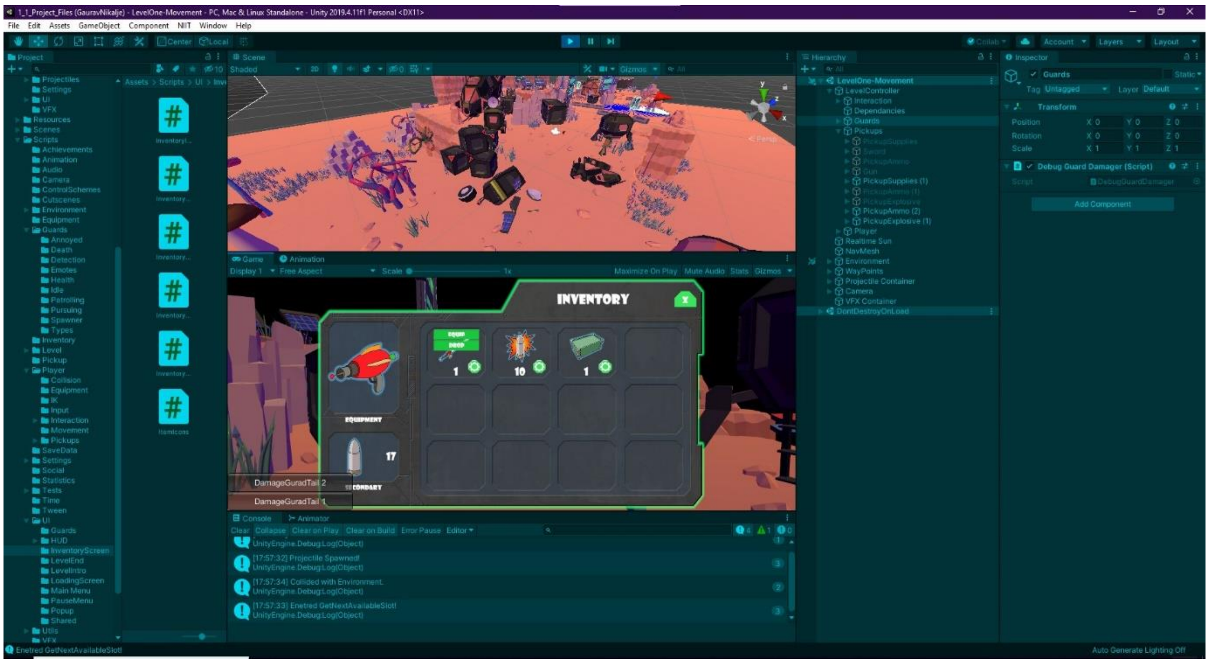
```
    2 references
    private void ShowPause()    //Assignment - 01
    {
        PauseGame(); //Timescale = 0;

        //GameIsPaused = true;
        hudController.HideHUD();
        pauseMenuController.Show();
    }
    2 references
    private void ShowInventory()    //Assignment 04 - Part II
    {
        inventoryMenuController.Show();
        player.Broadcaster.EnableActions(ControlType.None);
        hudController.HideHUD();

    }

    1 reference
    public void OnLevelFailed(string message)
    {
        levelEndMenu.Show(message, false);
        PauseGame();    //Assignment - 01 (Stops Time & Disables Inputs)
    }

```

❖ **Final Output.**



1_1_Project_Files (GauravNikalje) - LevelOne-Movement - PC, Mac & Linux Standalone - Unity 2019.4.11f1 Personal <DX11>

```
[17:54:39] Level loaded
UnityEngine.Debug:Log(Object)
```



```
UnityEngine.Debug:Log(Object)
[17:57:32] Projectile Spawned!
UnityEngine.Debug:Log(Object)
[17:57:34] Collided with Environment.
UnityEngine.Debug:Log(Object)
[17:57:33] Enetred GetNextAvailableSlot!
UnityEngine.Debug:Log(Object)
```

## <u>Step 03: What have I learnt</u>

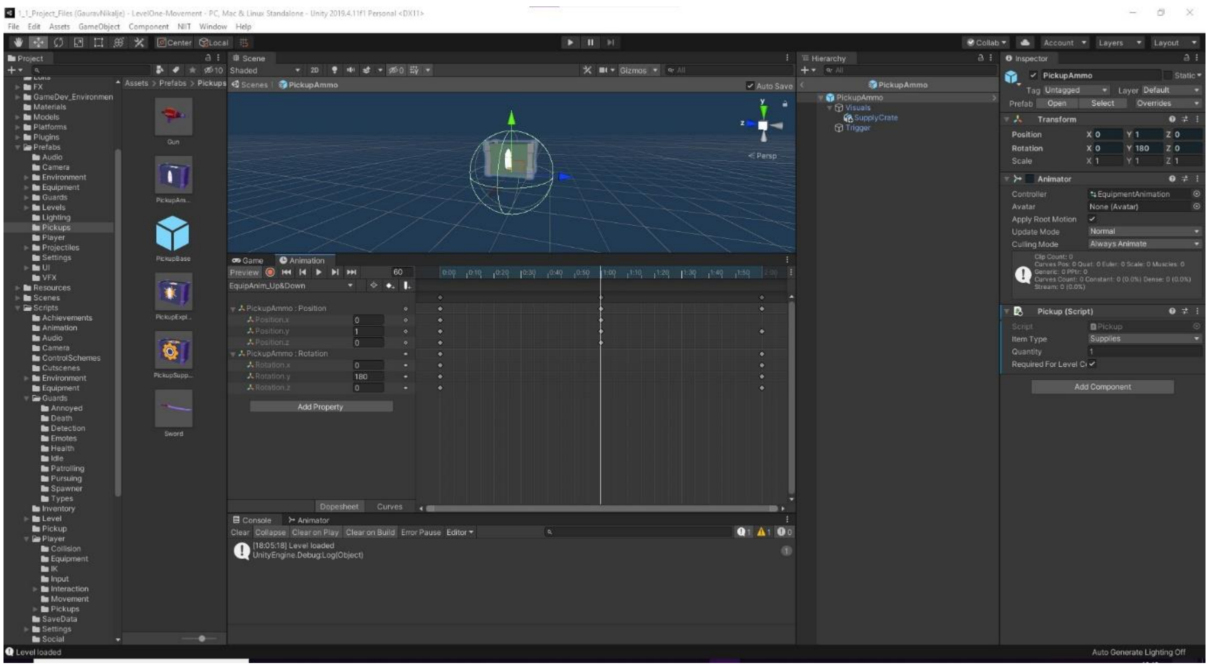
- **What is the factory pattern? Explain how it can be used in Unity. Give one non-inventory scenario where it could be useful.**

  - Factory patterns is a design pattern used for creating new instances or copies of the predetermined object.
  - It is a way to create object but the client or calling class will not know about how the objects were created.
  - In factory pattern subclass decides about instantiation of a class.

- We can Implement Factory Pattern to describe the basic functionality of guards and then apply these actions for various Guards.

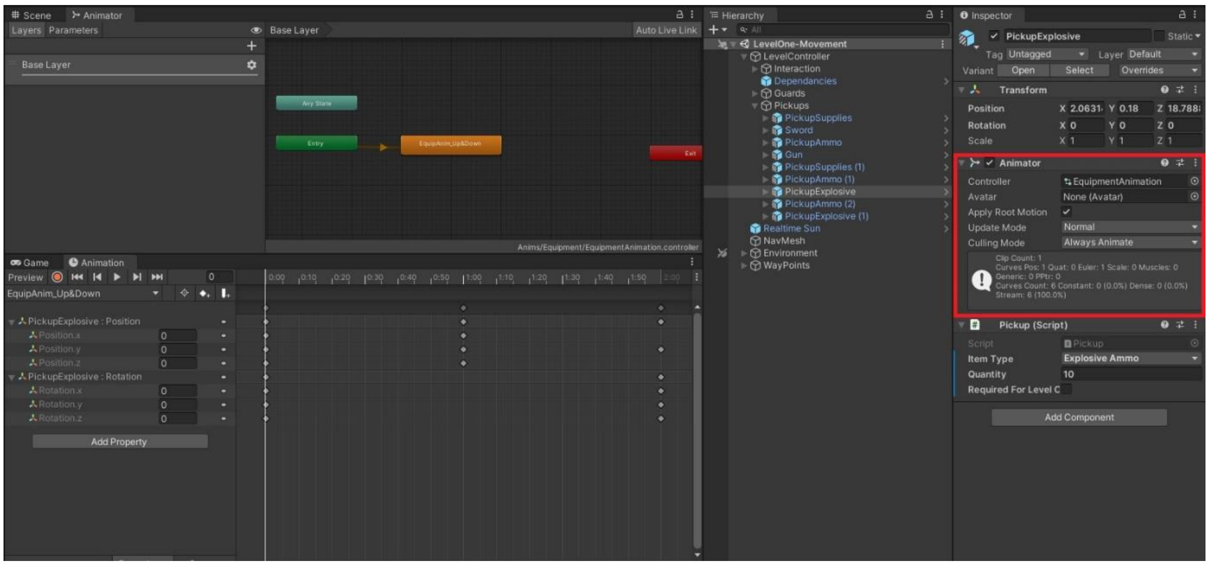
- **Are there any further improvements we could make to this inventory system? Choose 1 suggestion, explain what it is and give a rough run down of how you would implement it.**

  - We can create a rotation animation for the items.
  - We simply create an "AnimationClip" that moves the Pickup Item Up and Down.
  - For this select the main prefab "*PickupBase*" and in the "Animation" clip, add property of Transform type.
  - Then, on the First Frame set the object on position 0 and then on Second 1 in timeline, set its Y-Position to 1 and on second 2, again set it to 0.
  - Hence we have a Loop Animation.
  - And its "AnimationController" automatically gets created.
  - Now simply attach this AnimationClip to each and every Pickup Item Prefab-variant.



Position X 0 Y 1 Z 0
Rotation X 0 Y 180 Z 0
Scale X 1 Y 1 Z 1
Controller EquipmentAnimation
Avatar None (Avatar)
Apply Root Motion ✓
Update Mode Normal
Culling Mode Always Animate
Item Type Supplies
Quantity 1
Required For Level C ✓

```
[18:05:18] Level loaded
UnityEngine.Debug:Log(Object)
```



PickupExplosive

Tag Untagged  Layer Default

Transform

Position X 2.0631 Y 0.18 Z 18.788

Rotation X 0 Y 0 Z 0

Scale X 1 Y 1 Z 1

Animator

Controller EquipmentAnimation

Avatar None (Avatar)

Apply Root Motion ✓

Update Mode Normal

Culling Mode Always Animate

Clip Count: 1
Curves Pos: 1 Quat: 0 Euler: 1 Scale: 0 Muscles: 0
Generic: 0 PPtr: 0
Curves Count: 6 Constant: 0 (0.0%) Dense: 0 (0.0%)
Stream: 6 (100.0%)

Pickup (Script)

Script Pickup

Item Type Explosive Ammo

Quantity 10

Required For Level C

**----------------------THE END----------------------**