



Intermediate Scripting

Introduction to Game Development in
Unity
Spring 2026, 98-127, Lecture 8

Instructors: Jingxuan Chen, Dario Quintero, Shangyi Zhu, Jeffrey Wang



Table of Contents

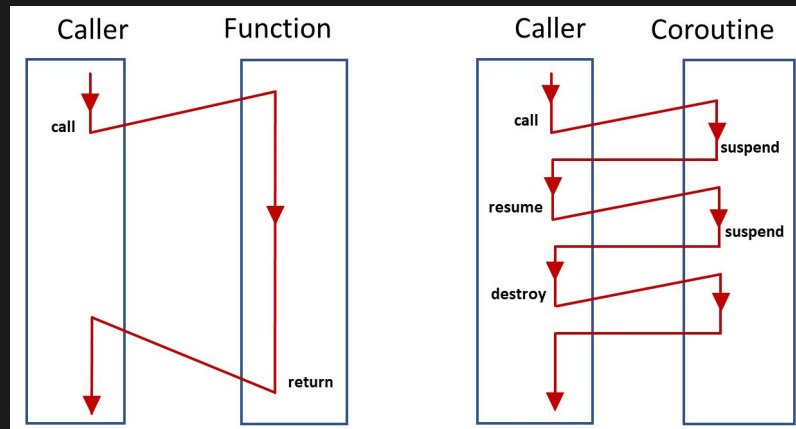
- Coroutines
 - Scriptable Objects
- Interfaces and Abstract Classes
 - Custom Editor Attributes



Coroutines

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

In Unity applications, this means coroutines can start running in one frame and then resume in another, allowing you to spread tasks across several frames.



Source: <https://docs.unity3d.com/6000.3/Documentation/Manual/Coroutines.html>



Consider This Function

```
void Fade()  
{  
    Color c = renderer.material.color;  
    for (float alpha = 1f; alpha >= 0; alpha -= 0.1f)  
    {  
        c.a = alpha;  
        renderer.material.color = c;  
    }  
}
```



Our Problematic Function

- In Unity functions always return in a single frame
- Therefore this object will instantly disappear rather than fade out over time
- How to translate this to a coroutine?

```
void Fade()
{
    Color c = renderer.material.color;
    for (float alpha = 1f; alpha >= 0; alpha -= 0.1f)
    {
        c.a = alpha;
        renderer.material.color = c;
    }
}
```



Translating our Function

```
void Fade()  
{  
    Color c = renderer.material.color;  
    for (float alpha = 1f; alpha >= 0; alpha -= 0.1f)  
    {  
        c.a = alpha;  
        renderer.material.color = c;  
    }  
}
```



1. Change return type to IEnumerator

```
IEnumerator Fade()  
{  
    Color c = renderer.material.color;  
    for (float alpha = 1f; alpha >= 0; alpha -= 0.1f)  
    {  
        c.a = alpha;  
        renderer.material.color = c;  
    }  
}
```



2. Insert yield return statement

```
IEnumerator Fade()  
{  
    Color c = renderer.material.color;  
    for (float alpha = 1f; alpha >= 0; alpha -= 0.1f)  
    {  
        c.a = alpha;  
        renderer.material.color = c;  
        yield return null;  
    }  
}
```

Brief Explanation of Changes

- IEnumerator: A C# interface that allows you to iterate through a collection one item at a time while keeping track of the current position.
- Unity uses the interface to keep track of execution of the coroutine

```
IEnumerator Fade()  
{  
    Color c = renderer.material.color;  
    for (float alpha = 1f; alpha >= 0; alpha -= 0.1f)  
    {  
        c.a = alpha;  
        renderer.material.color = c;  
        yield return null;  
    }  
}
```



Some Possible Wait Instructions

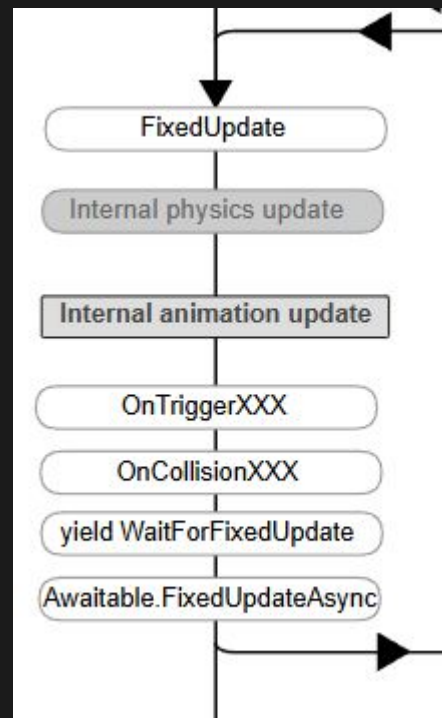
<code>yield return null;</code>	Resume next frame
<code>yield return new WaitForSeconds(t);</code>	Wait for scaled time (Changed with <code>Time.timeScale</code>)
<code>yield return new WaitForSecondsRealtime(t);</code>	Wait for real time (Use carefully)
<code>yield return new WaitForEndOfFrame();</code>	Resume after rendering
<code>yield return new WaitForFixedUpdate();</code>	Resume after physics step



PSA: Consult event execution order

This graph will show when yield instructions run relative to other events.

<https://docs.unity3d.com/6000.3/Documentation/Manual/execution-order.html>



Starting and Stopping Coroutines

Create a Coroutine reference.

Use StartCoroutine and StopCoroutine functions.

Note: When a coroutine is done running it will stop and be cleaned up automatically.

```
2 references
private Coroutine fadeCoroutine;
0 references
void BeginFade() {
    fadeCoroutine = StartCoroutine(Fade());
}

0 references
void StopFade()
{
    StopCoroutine(fadeCoroutine);
}
```



What happens if you call BeginFade twice?

```
2 references
private Coroutine fadeCoroutine;
0 references
void BeginFade() {
    fadeCoroutine = StartCoroutine(Fade());
}

0 references
void StopFade()
{
    StopCoroutine(fadeCoroutine);
}
```



An oversight with this code

StartCoroutine initializes a new coroutine each time it is called.

Therefore fadeCoroutine will be set to a new Coroutine, but the old one will still exist.

Two instances of the coroutine will be running at the same time! We will only have a reference to one!

```
2 references
private Coroutine fadeCoroutine;
0 references
void BeginFade() {
    fadeCoroutine = StartCoroutine(Fade());
}

0 references
void StopFade()
{
    StartCoroutine(fadeCoroutine);
}
```

The solution

We can ensure only a single instance of our coroutine is running with a null check like so:

```
0 references
void BeginFade() {
    if(fadeCoroutine == null)
    {
        fadeCoroutine = StartCoroutine(Fade());
    }
}
```

```
0 references
void StopFade()
{
    if(fadeCoroutine != null)
    {
        StopCoroutine(fadeCoroutine);
        fadeCoroutine = null;
    }
}
```



Final Coroutine Notes

Note you can just call StartCoroutine like so. This should be done when you have no need to stop a coroutine. This is often good enough 👍

```
0 references  
void BeginFade() {  
    StartCoroutine(Fade());  
}
```



Table of Contents

- Coroutines
- Scriptable Objects
- Interfaces and Abstract Classes
 - Custom Editor Attributes



Scriptable Objects

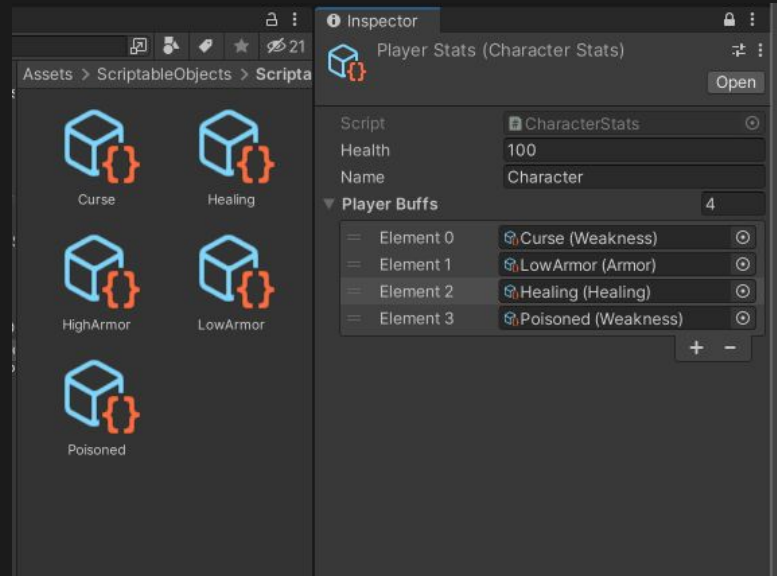
ScriptableObject is a Unity type for storing data separately from GameObjects.

Some considerations:

Unlike MonoBehaviours, ScriptableObjects are not attached to GameObjects as components.

Instead they **exist in the project as assets**, independent of GameObjects.

Consequently, you can drag or pick instances of them into fields in the Inspector.



Creating Scriptable Objects

```
using UnityEngine;
// Use the CreateAssetMenu attribute to allow creating instances of this ScriptableObject from the Unity Editor.
[CreateAssetMenu(fileName = "Data", menuName = "ScriptableObjects/SpawnManagerScriptableObject", order = 1)]
public class SpawnManagerScriptableObject : ScriptableObject
{
    public string prefabName;

    public int numberOfPrefabsToCreate;
    public Vector3[] spawnPoints;
}
```



■ Using SOs

```
public class ScriptableObjectManagedSpawner : MonoBehaviour
```

```
{
```

```
    // The GameObject to instantiate.
```

```
    public GameObject entityToSpawn;
```

```
    // An instance of the ScriptableObject defined above.
```

```
    public SpawnManagerScriptableObject spawnManagerValues;
```

```
    void SpawnEntities()
```

```
    {
```

```
        int currentSpawnPointIndex = 0;
```

```
        for (int i = 0; i < spawnManagerValues.numberOfPrefabsToCreate; i++)
```

```
        {
```

```
            // Creates an instance of the prefab at the current spawn point.
```

```
            GameObject currentEntity = Instantiate(entityToSpawn, spawnManagerValues.spawnPoints[currentSpawnPointIndex], Quaternion.identity);
```

```
            // Sets the name of the instantiated entity to be the string defined in the ScriptableObject and then appends it with a unique number.
```

```
            currentEntity.name = spawnManagerValues.prefabName + instanceNumber;
```

```
            // Moves to the next spawn point index. If it goes out of range, it wraps back to the start.
```

```
            currentSpawnPointIndex = (currentSpawnPointIndex + 1) % spawnManagerValues.spawnPoints.Length;
```

```
            instanceNumber++;
```

```
        }
```

```
    }
```

```
}
```

```
public class SpawnManagerScriptableObject : ScriptableObject
{
    public string prefabName;

    public int numberOfPrefabsToCreate;
    public Vector3[] spawnPoints;
}
```

Why Use Scriptable Objects?

- Enables you to separate data from logic
- Allows Sharing of data
 - Multiple GameObjects may have copies same data, with SOs this data can be shared
- Team Collaboration
 - Designer-friendly
 - Designers can modify SOs rather than Monobehaviors or prefabs

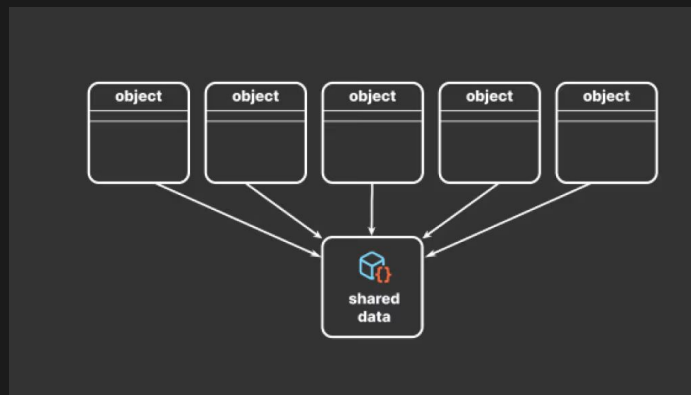
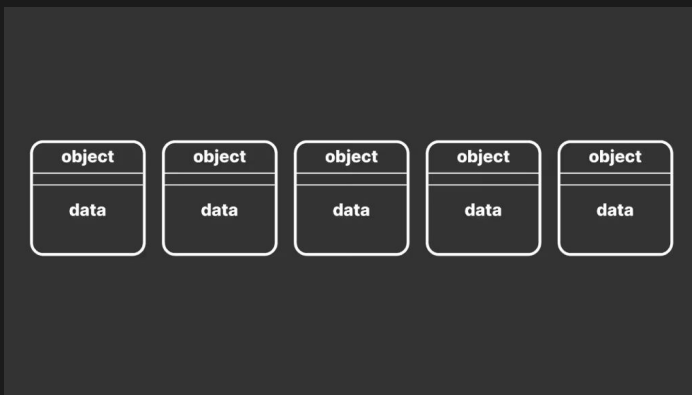


Table of Contents

- Coroutines
- Scriptable Objects
- Interfaces and Abstract Classes
 - Custom Editor Attributes



Interfaces

An interface is a collection of method signatures and properties.

You can think of interfaces like a contract that classes can implement.

You cannot have an instance of an interface the way you can with a class, you must have your object inherit from the interface.



A Case study

Interfaces are excellent for keeping code clean. Consider this interface:

```
public interface IDamageable
{
    void Damage();
}
```

We've written a contract that all Damagable objects will specify a Damage function.



Using Interfaces

We inherit from the interface to ascribe to this contract. We must then define `Damage()`.

```
public class Bat_Enemy : MonoBehaviour, IDamageable

private int _health = 10;

public void Damage()
{
    _health--;
    if (_health < 1)
    {
        Destroy(this.gameObject);
    }
}
```

Using Interfaces

Similarly, we could define a Crate class that is also Damagable like so.

Note the two could have completely different behavior when damaged, but both can take damage.

```
public class Crate : MonoBehaviour, IDamageable

public void Damage()
{
    Destroy(this.gameObject);
}
```

Tying Things Together

Here is some poor code that does not take advantage of the interface:

```
public class Player : MonoBehaviour, IDamageable
{
    private void OnTriggerEnter2D(Collision other)
    {
        if (other.GetComponent<Bat_Enemy>() != null)
        {
            other.GetComponent<Bat_Enemy>().Damage();
        }
        else if (other.GetComponent<Crate>() != null)
        {
            other.GetComponent<Crate>().Damage();
        }
        else if (other.GetComponent<Skeleton_Enemy>() != null)
        {
            other.GetComponent<Skeleton_Enemy>().Damage();
        }
        else if (other.GetComponent<Villager>() != null)
        {
            other.GetComponent<Villager>().Damage();
        }
        // and so on for everything the player can damage with this attack...
    }
}
```

Tying Things Together

And here is code doing that same thing in a much cleaner way:

```
public class Player : MonoBehaviour, IDamageable

private void OnTriggerEnter2D(Collision other)
{
    if (other.GetComponent<IDamageable>() ≠ null)
    {
        other.GetComponent<IDamageable>().Damage();
    }
}
```

Abstract classes

Abstract classes let us use inheritance and define contracts in a similar way, however with some additional options.



■
Consider this abstract class:

0 references

```
public abstract class PowerUp : MonoBehaviour
{
    1 reference
    protected abstract void ApplyEffect(GameObject player);
    1 reference
    protected virtual void PlayAnimation()
    {
        transform.Rotate(Vector3.up, 50f * Time.deltaTime);
    }

    0 references
    private void OnTriggerEnter(Collider foreignBody)
    {
        if (foreignBody.CompareTag("Player"))
        {
            ApplyEffect(foreignBody.gameObject);
            PlayAnimation();
            Destroy(gameObject, 0.5f);
        }
    }
}
```

Important keywords

protected

```
protected
```

Only the parent class and its subclasses can access this variable.

Will not show up in other scripts or components.

abstract

```
abstract void ApplyEffect(GameObject player);
```

Like declaring a method in an interface; You *must* also do so in the inheritor.

virtual

```
protected virtual void PlayAnimation()  
{  
    transform.Rotate(Vector3.up, 50f * Time.deltaTime);  
}
```

You must specify virtual methods in your abstract class, but you can optionally *override* them in you inheritor.

Here are two inheritors.

Note the following:

1. They both define ApplyEffect
2. MegaShield optionally overrides PlayAnimation

```
0 references
public class HealthPack : PowerUp
{
    2 references
    protected override void ApplyEffect(GameObject player)
    {
        Debug.Log("Healed the player!");
    }
}
```

```
0 references
public class MegaShield : PowerUp
{
    2 references
    protected override void ApplyEffect(GameObject player)
    {
        Debug.Log("Shield Activated!");
    }
    2 references
    protected override void PlayAnimation()
    {
        transform.localScale *= 1.1f;
    }
}
```

An Additional Note

We can also modify `Megashield` like so. This will run both the base class's `PlayAnimation` definition and the additional code we have written.

```
3 references  
protected override void PlayAnimation()  
{  
    base.PlayAnimation();  
    transform.localScale *= 1.1f;  
}  
}
```



Table of Contents

- Coroutines
- Scriptable Objects
- Interfaces and Abstract Classes
 - Custom Editor Attributes



Editor Attributes

Attributes in C# are metadata markers that can be placed above a class, property, or method declaration to indicate special behaviour.

```
[HideInInspector]  
public float strength;
```




```
[Header("Title")]
```

0 references |  Unity Serialized Field


```
public float float1;
```

```
[Space(10)]
```

0 references |  Unity Serialized Field


```
public float float2;
```

```
[HideInInspector]
```

0 references |  Unity Serialized Field


```
public float float3;
```

```
[Tooltip("This is the fourth float")]
```


0 references |  Unity Serialized Field

```
public float float4;
```

```
[Range(0f, 1f)]
```


0 references |  Unity Serialized Field

```
public float float5;
```

0 references |  Unity Serialized Field

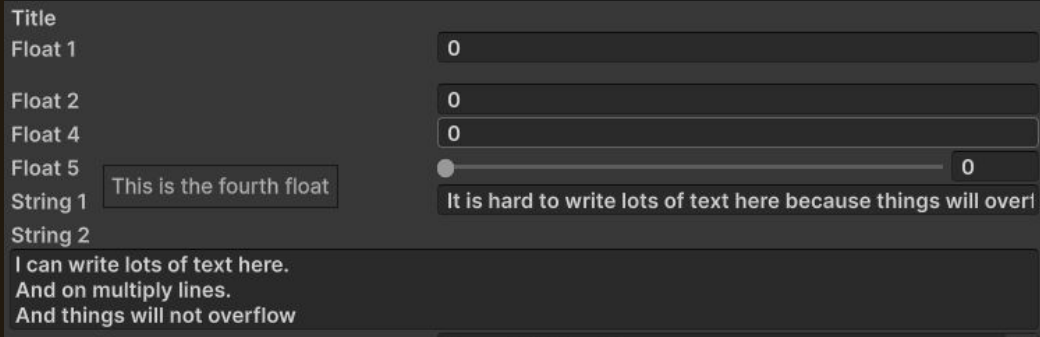
```
public string string1;
```

```
[TextArea(0, 60)]
```

0 references |  Unity Serialized Field

```
public string string2;
```

Here are some useful attributes for your inspector windows.



Other Attributes

<code>[SerializeField]</code>	Shows private variables in inspector
<code>[RequireComponent (typeof (T))]</code>	Requires <code>GameObject</code> to have component for script to be attached
<code>[DisallowMultipleComponent]</code>	Disallow multiple instances of script attached to same <code>GameObject</code>
<code>[ContextMenu ("Function Name")]</code>	Adds an option to the component's three-dot meatball menu. Clicking it runs function in Edit mode.
<code>[ExecuteAlways]</code>	Makes the script run its <code>Update</code> , <code>OnEnable</code> , and <code>OnDisable</code> functions even while the editor is not in Play mode.
<code>[SelectionBase]</code>	Clicking child in Scene View select parent instead



Custom Attributes

Unity also allows you to write your own custom attributes. To do this, you must define two things:

- A blank class inheriting from `PropertyAttribute`
- A class with the logic for your attribute inheriting from `PropertyDrawer`

The property drawer class must be in **Assets/Editor**. The attribute class must be *outside* of **Assets/Editor**.



An Example

This is a custom attribute called SceneName. It is used in front of a string to force the string to be the name of a Scene in our build profile.

SceneNameAttribute.cs

Assets > Attributes > SceneNameAttribute.cs > ...

```
1 using UnityEngine;
```

```
2
```

```
2 references
```

```
3 public class SceneNameAttribute : PropertyAttribute { }
```

```
4
```



And here is the
PropertyDrawer Class.

The Drawer both renders
and handles the logic for
the attribute.

```
SceneNameDrawer.cs
Assets > Editor > SceneNameDrawer.cs > ...
4
5 [CustomPropertyDrawer(typeof(SceneNameAttribute))]
  0 references
6 public class SceneNameDrawer : PropertyDrawer
7 {
  0 references
8   public override void OnGUI(Rect position, SerializedProperty property, GUIContent label)
9   {
10      if (property.propertyType != SerializedPropertyType.String)
11      {
12         EditorGUI.LabelField(position, label.text, "Use [SceneName] with strings only.");
13         return;
14      }
15
16      // Get all scenes enabled in Build Settings
17      var scenes = EditorBuildSettings.scenes
18         .Where(s => s.enabled)
19         .Select(s => System.IO.Path.GetFileNameWithoutExtension(s.path))
20         .ToArray();
21
22      if (scenes.Length == 0)
23      {
24         EditorGUI.LabelField(position, label.text, "No scenes in Build Settings.");
25         return;
26      }
27
28      // Find the index of the currently saved string
29      int currentIndex = Mathf.Max(0, System.Array.IndexOf(scenes, property.stringValue));
30
31      // Draw the popup/dropdown
32      currentIndex = EditorGUI.Popup(position, label.text, currentIndex, scenes);
33
34      // Update the actual string value based on selection
35      property.stringValue = scenes[currentIndex];
36   }
37 }
38
```

Demo Time

