Events & Delegates in Unity

In **Unity** by John French / December 15, 2021 / **54 Comments**

When learning how to build a game, one of the biggest challenges you're likely to face is how to connect all of the different moving parts together in a way that works.

For example, **making a character move**, **jump** or **adding up the score**, can be relatively easy to do on its own.

But, actually connecting all of the things that happen in your game, without making it confusing to work with, can be extremely challenging.

Especially for a beginner.

Luckily, there's a solution.

In Unity, it's possible to create modular connections between scripts and objects by using **events** and **delegates**, which allow you to trigger game logic as it happens, without relying on tight connections between scripts.

In this article, you'll learn the difference between events, delegates, actions and Unity Events, how they work and how you can use them to create event-based logic in your games.

Here's what you'll find on this page:

- **Scriptable Object Unity Events**

Let’s start with why you might want to use event-based logic in the first place.

Events & Delegates overview video

For a general overview of how to use events and delegates in Unity, try my **video**, or continue to the full article below.

Event-based logic in Unity

If you’re just getting started with Unity, so far, you might have placed a lot of your game’s logic in Update.

Update is called every frame so, for checks that are made constantly, such as handling input, placing your code in Update is appropriate and it works well.

However…

As you start to build your game, it can be all too easy to just keep using Update for other logic as well, making checks every frame to see if the script needs to do something or not.

Which can cause problems.

Having each script check on the state of other scripts requires you to keep tight connections between different objects that might not otherwise need to be there.

Which can make it extremely difficult to build your game.

Why? I’ll explain.

When a player dies in a game, for example, there may be many different things that need to happen as a result:

- The player character might dramatically collapse
- Player input is turned off
- The UI is updated

- Enemies no longer need to attack you
- And the scene needs to be restarted

To manage any of these events from a single script could quickly become impossible.

What's more, even if you could manage all of this from one script, without making a mess of your project, if anything changes later on, or if you want to add something new, you'll need to change the script again to be able to do it.

So what's the answer?

While there are many design approaches that you can use to organise your code when making games, one method is to use the Observer Pattern.

The Observer Pattern in Unity

The Observer Pattern is a software design pattern that allows you to create modular game logic that is executed when an event in the game is triggered.

It typically works by allowing observers, in this case, other scripts, to subscribe one or more of their own functions to a subject's event.

Then, when the event is triggered by the subject, the observers' functions are called in response.

Game Programming Patterns

Software design patterns are, put simply, architectural approaches to designing code in a way that avoids or solves certain problems.

There are many different patterns, however, the methods in this article loosely fit into the Observer Pattern, where a single event can be used to passively control many observing objects.

For a more accurate description of the Observer Pattern try **Game Programming Patterns** by Robert Nystrom, which is an incredibly useful and in-depth resource on game architecture.

For example, the player's health script could declare an On Player Death event, that's called when the player runs out of health.

Other systems in the game that need to respond to that event, such as other enemies, or elements of the UI, can subscribe to it and, when it's called, their subscribed functions will be called too.

Events allow scripts to respond to something when it happens, instead of constantly checking if it's happened.

The benefit of using a system like this is that the subject does not need to know about the observers and the observers don't need to know about each other.

The player's script only needs to manage the health of the player and call the death event when it runs out, while other scripts, such as the UI or enemies in the scene, only need to worry about what they will do when the event takes place.

This means that you can connect different pieces of game logic with the actual events of a game, but without needing to manage specific script to script connections.

Which could get messy…

And, if you ever want to add new functionality in response to an event, you won't need to go into the subject's script to do it.

You can simply add a new observer with a new response.

That's the theory anyway, but how does it actually work?

While there are many different ways to create an observer-style event system in Unity, a common method is to use **delegates**.

Delegates in Unity

Delegates are, essentially, function containers.

They allow you to store and call a function as if it were a variable.

To do that, however, you'll need to decide what type of function the delegate will be able to hold.

For example, you could define a type of delegate that has a void return type (meaning that it doesn't return anything) and that doesn't take any parameters.

When written as a normal function, it would look like this:

```
void MyFunction()
{
    // Normal function
}
```

To define it as a delegate instead, you'll need to add the delegate keyword and remove the function's body.

Like this:

```
delegate void MyDelegate();
```

This is a delegate signature, a reference for a type of delegate.

It defines what kind of function can be stored in delegate instances of this type but it isn't, itself, an instance.

Which means that, to actually use one, you'll need to declare an instance of that delegate type.

Like this:

```
delegate void MyDelegate();
MyDelegate attack;
```

The attack delegate can now hold any function that matches the "My Delegate" signature.

Meaning that you can assign any function to the attack delegate so long as it has a void return type and no parameters.

How to use delegates with parameters

When setting up a delegate, the return type and parameter signature define what type of function can be assigned to it.

Changing what a delegate can accept and what it will return, if anything, works in a similar way to setting up a regular method.

For example, instead of taking no parameters, you could instead define a delegate that takes a boolean value.

Like this:

```
public delegate void OnGameOver(bool canRetry);
```

The parameter's value is then passed in when the delegate gets called.

Any function that's assigned to the delegate must accept a matching set of parameters.

The benefit, however, is that the function will be able to use the data that's passed in when the delegate is triggered.

Assigning a function works in the same way as setting a variable's value, such as a float or a string.

All you need to do is pass in the name of the function that you want to set, without parentheses.

Like this:

```
delegate void MyDelegate();
MyDelegate attack;
void Start()
{
    attack = PrimaryAttack;
}
void PrimaryAttack()
{
    // Attack!
}
```

Now, when the attack delegate is triggered the Primary Attack function will be called.

For, example, when a key is pressed.

Like this:

```
void Update()
{
    if (Input.GetKeyDown(KeyCode.Space))
    {
        attack();
    }
}
```

It's important to only actually call a delegate if there's a function assigned to it, as trying to trigger an empty delegate will cause a null reference error.

For this reason, it's always a good idea to check if a delegate is empty before calling it.

Either with a null check, like this:

```
if (attack != null)
{
    attack();
}
```

Or by using a simplified shorthand check, which does the same thing,

Like this:

```
attack?.Invoke();
```

But how is all this useful?

One of the main benefits of using a delegate, is that you can change the function that is triggered when the delegate is called.

For example, you could change the attack function to, instead, trigger a secondary attack by using the number keys to switch attack types.

Like this:

```
public class DelegateExample : MonoBehaviour
{
    delegate void MyDelegate();
    MyDelegate attack;
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            if (attack != null)
            {
                attack();
            }
        }
        if (Input.GetKeyDown(KeyCode.Alpha1))
        {
            attack = PrimaryAttack;
        }
        if (Input.GetKeyDown(KeyCode.Alpha2))
        {
            attack = SecondaryAttack;
        }
    }
    void PrimaryAttack()
    {
        // Primary attack
    }
    void SecondaryAttack()
    {
        // Secondary attack
    }
}
```

This changes what happens when the player attacks, without changing the code that triggers the attack event. A single, unchanged trigger can be used to create different results.

A delegate can be useful for changing what a trigger does, without changing the trigger itself.

Which is easier to manage.

But, while this can be useful for managing functions in a single script, how can delegates be used to create a complete events system?

How can a delegate on one object, be used to trigger a response on another?

How to create an event system with delegates in Unity

In a single script, a delegate can be used to trigger different functions, depending on which method is assigned to it.

Which can be very useful.

However, delegates can also work as **multicast delegates** which means that, instead of triggering only one function, they can be used to trigger multiple functions all at once.

This works by adding the function to the delegate container, instead of replacing it, in the same way that you might add a number to a float variable to increase it.

Like this:

```
delegate void MyDelegate();
MyDelegate attack;
void Start()
{
    attack += PrimaryAttack;
    attack += SecondaryAttack;
}
```

In this example, both attack functions will now be called when the attack delegate is triggered.

This can be useful for calling multiple functions from a single event trigger, in the same way that being able to change what an action does, without changing the action itself, can be useful for making your code more manageable.

However, being able to call multiple functions from a delegate can also be useful for creating an events system between multiple scripts.

Here's how it works…

A player health script, for example, could be used to call a Game Over delegate once the player's health reaches zero.

```
public class PlayerHealth : MonoBehaviour
{
    float health=100;

    delegate void OnGameOver();
    OnGameOver onGameOver;
    public void TakeDamage(float damage)
    {
        health -= damage;
        if(health < 0)
        {
            onGameOver?.Invoke();
        }
    }
}
```

It might make sense to place the Game Over event in this script if, for example, the player losing all of their health is the only way that the game could be lost.

Then, when that happens, the Game Over delegate is triggered.

So, how can you connect other scripts to the delegate?

In order for other scripts to be able to easily access the delegate, it needs to be **public** and it will need to be **static**.

Like this:

```
public delegate void OnGameOver();
public static OnGameOver onGameOver;
```

Making the delegate public allows other scripts to access it, while making it static means that the reference is shared by all instances of the class.

This means that, if another script needs to subscribe one of its functions to the delegate, such as a function that restarts the game, all you would need is the name of the class that it is defined in to access it.

Like this:

```
public class GameController : MonoBehaviour
{
    void RestartGame()
    {
        // Restart the game!
    }
    private void OnEnable()
    {
        PlayerHealth.onGameOver += RestartGame;
    }
}
```

While there are many **methods for connecting scripts**, using a static reference like this makes it relatively straightforward to build a connection between an observing script and a delegate.

And, in this example, it allows the Game Controller script to subscribe its Restart Game function to the On Game Over event, without needing a reference to the player object first.

This happens when the controller object is enabled, in the **On Enable** function, which gets called automatically when an object is turned on, **happening after Awake, but before Start**.

However, because this does create a connection between the two scripts, it's *extremely* important to unsubscribe the function if it's no longer needed.

The simplest way to do this is to also subtract the function from the delegate if the controller object is ever disabled, in the **On Disable** event,

Like this:

```
public class GameController : MonoBehaviour
{
    void RestartGame()
    {
        // Restart the game!
    }
    private void OnEnable()
    {
        PlayerHealth.onGameOver += RestartGame;
    }
    private void OnDisable()
    {
        PlayerHealth.onGameOver -= RestartGame;
    }
}
```

This means that, if the controller object is ever turned off or destroyed, the delegate subscription is removed.

Which is important, as subscribing functions to events without cleaning them up afterwards can cause memory leaks.

However, the result is a modular events system that allows other scripts to respond to game events as they happen, without creating explicit connections between objects.

Which can make it much easier to manage and build your game.

There's just one problem.

Because any other script can access the delegate, it's technically possible for any script to call it or even clear the assigned functions from the delegate.

Like this:

```
// clears the delegate
PlayerHealth.onGameOver = null;
// calls the delegate
PlayerHealth.onGameOver();
```

Which is a problem.

Why?

Because one of the main benefits of using an observer-style system is to allow scripts to respond when something happens.

By design, it helps you to avoid tightly coupled connections between scripts that can make your game difficult to manage.

For example, it doesn't really make sense for a different script, other than the player health script to be able to decide if the player's health is below zero.

Meaning that a different script shouldn't be able to call the delegate that's associated with that event.

But, technically, because the delegate is public and static, any script could call it.

And while your scripts are unlikely to go rogue, calling any function they feel like, the ability to call or reset a delegate from anywhere makes it much easier to do it by accident.

So what can you do?

Luckily it's easy to prevent, by using the event keyword.

Events in Unity

Events in Unity are a special kind of multicast delegate and, generally speaking, they work in the same way as regular delegates.

However, while delegates can be called by other scripts, event delegates can only be triggered from within their own class.

This means that, when using events, other scripts can only manage their own relationship with the event. They can subscribe and unsubscribe their own functions but they cannot trigger the event, or alter the subscription of other functions from other classes.

This prevents a different script from doing something that, by design, you may not want it to do.

And, if it tries, you'll get an error.

Trying to trigger or clear an event from any other script will result in an error.

So, how can you use events in Unity?

To use events you'll need to define a delegate type and an instance, just like when setting up a normal delegate.

Then, to create an event instance, instead of a delegate, simply add the event keyword when declaring it.

Like this:

```
public delegate void OnGameOver();
public static event OnGameOver onGameOver;
```

Events vs delegates

Given that events have inherent security, since they can't be triggered by other scripts, you might be wondering why you'd use a delegate over an event at all.

While it's possible to use delegates in the same way as events, delegates are, generally speaking, simply meant to act as data containers for functions.

For example, it's possible to use a delegate as a parameter in a method, which would allow you to pass a function into a function, which could be useful for executing a different command after something has happened.

Whereas events, unsurprisingly, work well as part of an events system, where one object's actions trigger other scripts to respond.

So while events are, essentially just specialised delegates, they are more suited to this particular task of triggering an event that other, observing objects can then respond to but not interfere with.

Which can be useful when designing your scripts in a way that is easy to manage.

Using event delegates is an easy way to manage complex relationships between behaviours in your game.

However, while not difficult, it can sometimes be inconvenient to declare a new delegate type every time you want to use one.

Especially if all you want to do is create a basic event.

Luckily, **Actions** allow you to use a generic delegate type without needing to define it in your script first.

Actions in Unity

Actions are, essentially, ready-made delegates.

They can be used to easily create a delegate or delegate event with a void return type.

Which means that you can use an action in the same way as you would a delegate, the difference being that you don't need to declare it first.

So how do actions work?

Actions use the System namespace, so you'll need to add the **using System** directive to the top of your script.

Like this:

```
using System;
```

Then, instead of defining a delegate and creating an instance of it, you can, instead, create an action that will work in the same way.

Like this:

```
// this...
public static event Action OnGameOver;
// is basically the same as this...
public delegate void OnGameOver();
public static event OnGameOver onGameOver;
```

Actions have a void return type, meaning that they don't return a value, but they can be used with one, or more, parameters.

This works by adding the parameter type, or types if you want to use more than one, in angled brackets when declaring the action.

Like this:

```
public static event Action<string> OnGameOver;
public static event Action<float, bool> OnPlayerHurt;
```

Just like with regular delegates, any function that subscribes to the action needs to have the same method signature which, in the case of actions, is always a return type of void along with whatever parameters you choose to use.

Such as a string for example:

```
public static event Action<string> OnGameOver;
public void TakeDamage(float damage)
{
    health -= damage;
    if(health < 0)
    {
        OnGameOver?.Invoke("The game is over");
    }
}
```

System Action vs System Func

While system actions allow you to define a void delegate on the fly, **Func** works in a similar way, allowing you to create an encapsulated delegate. Just like actions, **Func** can take optional arguments but, unlike actions, it can also return a value.

Actions in Unity are a simple way to implement delegates in scripting without needing to explicitly define them.

Combined with the event keyword, they are ideal for creating a modular events-based system in your game.

However, while actions allow you to make your *scripts* more modular, you might find it more useful to work with events in the Inspector instead.

And for that, you'll need Unity Events.

Unity Events

Generally speaking, Unity Events work in a similar way to event delegates, in that they can be triggered from a script allowing other, observing, scripts and components to act in response.

The difference is, however, while event delegates are typically managed in scripting, Unity Events are serialisable, meaning that you can work with them in the Inspector.

Chances are, even if you're new to Unity, you may have already worked with Unity Events before.

UI controls, such as buttons, for example, carry an On Click Unity Event that gets called whenever the button is clicked.

If you're ever connected a script to a button, then you may have already used Unity Events.

Anything you connect to that event will be executed when the button is clicked, allowing you to connect behaviours visually in the inspector.

There are lots of reasons why this might be useful to you, such as allowing non-programmers to work with event-based behaviours, for example.

However, there's another reason, that can be helpful, even if you're comfortable with writing code.

Using Unity Events to manage the interactions between scripts can encourage you to limit the scope of each script to a singular purpose.

How? I'll explain…

It can be surprisingly easy to write a script that does too much.

Take the player health script that I've been using as an example in this article.

A simple player health script might manage a health value and it might expose a public damage function to other objects that could hurt the player.

This makes sense since, without health, the player cannot be hurt so, by adding this script to a player object, you're essentially giving them health and a means for their health to be removed.

Then, when the player is out of health, the player dies and the game is over.

At this point, it could be tempting to start to connect other, local systems to the player health behaviour in scripting, such as disabling movement, triggering sound effects and causing the player to fall down.

And why wouldn't you?

After all, these are local systems.

It's not like you're trying to trigger something to happen on an entirely different object, such as an enemy, or in the UI. Many of these systems will likely exist on the same game object or within a single collection of objects that make up the player.

However, you may find it more useful to limit the scope of the script to only managing the health of the player and calling the player death Unity Event if it drops below zero.

Then, if something needs to happen as a result, you can hook it up in the Inspector using the Unity Event.

Why do it this way?

Doing it like this makes the player health script modular and reusable, since it's now, essentially just a health script.

It's no longer specific to the player, meaning it could be used on an enemy, or even on a damageable object.

Which is useful, since the process of taking damage in the game may be the same for any object or character that can be hurt.

The only thing that actually needs to be different, is what happens when the object is damaged. For example, a player taking damage would require the UI to be updated while an enemy taking damage might earn you experience.

Breakable boxes on the other hand might produce loot but rarely scream in pain.

Although each outcome is different, the trigger is the same.

Same input, different results.

Which is why using Unity Events to manage the relationships of scripts, even when they're on the same object, can make a lot of sense.

So how do they work?

How to use Unity Events

To use a Unity Event, you'll need to add the **Unity Engine Events** namespace to your script.

Like this:

```
using UnityEngine.Events;
```

Then, simply declare a public Unity Event for the event that you want to trigger.

Like this:

```
using UnityEngine;
using UnityEngine.Events;
public class PlayerHealth : MonoBehaviour
{
    float health=100;
    public UnityEvent onPlayerDeath;
    public void TakeDamage(float damage)
    {
        health -= damage;
        if(health < 0)
        {
            onPlayerDeath.Invoke();
        }
    }
}
```

Notice that, when calling the event in this example, I haven't added a null check like when triggering a delegate.

This is because, while Unity Events act like delegates, they're technically a Unity class.

Which means that trying to call one when it's empty won't cause you an error.

Once the Unity Event is set up, you'll want to connect it to something.

Back in the Inspector, just like when working with UI elements, you'll see the Unity Event controls.

Except, instead of the familiar OnClick(), you'll see the Unity Event that you created:

Simply click the plus button to add a new function call to the list.

Then drag the script or component that you want to work with to the empty object field and select what you want to happen when the event is triggered.

Like this:

*In this example, I've selected an Input Controller script and selected **bool enabled**, leaving the checkbox unchecked (false), which will simply turn that script off when the event is triggered, disabling player controls.*

In this example, selecting the **enabled** function call on the Input Controller script, and leaving the bool checkbox unchecked (so, enabled = false) will turn off the input script when the **On Player Death** event gets called.

While simply turning off the script works fine, you may want to call a specific function from the script instead, as this would allow you to control exactly what happens when the input is disabled.

In which case, simply write a public function inside the script to handle disabling the input.

Like this:

```
public class InputController : MonoBehaviour
{
    private void Update()
    {
        // Input stuff
    }
    public void DisableInput()
    {
        Debug.Log("Input was disabled");
        enabled = false;
    }
}
```

Then, instead of turning the script off, select the **Disable Input** function in the Inspector dropdown.

Like this:

To call a function when a Unity event is triggered, make sure it's public and select it from the Unity Event dropdown.

How to use Unity Events with parameters

Normally, triggering a function that accepts an argument using a Unity Event allows you to pass a static parameter in.

This means that whatever you set the parameter to in the Inspector is what will be passed to the function.

Such as the enabled function call which allows you to turn something on or off by checking or unchecking the boolean parameter.

A lot of the time, this may be all you need to do, however, sometimes you may also want to pass data dynamically from the Unity Event itself to the function that's being called.

You may have seen this before when using Unity Events.

For example, instead of a fixed, static value, the Slider UI component is able to pass a **dynamic** value in its On Value Changed event, allowing a function to take the float value from the slider control when it is updated.

Which can be useful for creating **volume controls**, for example.

So how can you create a custom dynamic value Unity Event?

Instead of a zero argument event, like the Player Death example, how could you use a Unity Event to not only let other scripts know that the player was hurt but, also, how much damage was taken.

Here's how to do it…

First, you'll need to create a separate serializable class in your project for the custom Unity Event.

Like this:

```
using UnityEngine.Events;
using System;
[Serializable]
public class FloatEvent : UnityEvent <float> { }
```

In this case, I've created a class called **Float Event**, that inherits from the Unity Event class.

Except that this particular version of Unity Event takes a generic parameter which, in this case, I've set as a float, by adding "float" in angled brackets, after the class declaration, but before the class body, which is empty.

Next, in the subject script, where the Unity Event will be declared and triggered, instead of adding a Unity Event instance, add an instance of the class you just created.

In this case, a **Float Event**.

Like this:

```
using UnityEngine;
using UnityEngine.Events;
public class PlayerHealth : MonoBehaviour
{
    float health=100;
    public UnityEvent onPlayerDeath;
    public FloatEvent onPlayerHurt;
    public void TakeDamage(float damage)
    {
        health -= damage;
        onPlayerHurt.Invoke(damage);
        if(health < 0)
        {
            onPlayerDeath.Invoke();
        }
    }
}
```

Then, when the Float Event is called, you'll need to pass a float value into it, such as the amount of damage taken for example.

You'll then be able to use this data with any functions that accept a float argument.

Such as a function that updates the player's health bar, for example:

```
public class HealthBar : MonoBehaviour
{
    public void UpdateHealthBar(float value)
    {
        Debug.Log(value + " health was removed");
    }
}
```

To connect the event data with a function that can use it, select the dynamic option when choosing the response function.

Choose the dynamic float option to pass data to the function from the Unity Event. This will only appear when the function accepts a matching parameter signature. In this case, a single float.

Keep in mind, however, that the dynamic option will only appear if the function accepts a matching parameter signature which, in this case, is a single float value.

Choosing any of the static options will only allow you to pass in whatever data you enter in the Inspector, just like with regular zero-argument Unity Events.

When should you use a Unity Event?

Unity Events can be extremely useful for making logical connections between scripts in the Inspector.

They can help you to keep scripts modular and easy to manage, by encouraging you to connect other scripts and components' behaviours to the events that a script triggers, and not to the script directly.

This is especially true when working with local objects, such as the different parts that make up a character, as it can be all too easy to simply connect scripts together with public references, increasing the scope of what each script is responsible for, making them harder to manage.

However…

While Unity Events can be a great way to manage relationships between local scripts and components, you probably won't want to connect two remote objects in this way.

Hooking up scripts in the Inspector requires you to make a manual connection which may not work well for different objects in the scene, especially if they're created as the game runs.

Meaning that, when connecting events between unrelated objects, you may find it more useful to use event delegates instead.

But, event delegates only really work in scripting, which can be unhelpful if you're trying to keep your scripts modular by limiting what each script does.

So, how can you combine the convenience of Unity Events with the advantages of game-wide event delegates?

One option is to create Scriptable Unity Events.

Scriptable Object Unity Events

Scriptable object events work like Unity Events except that they allow you to work more easily between unrelated objects, like you would with event delegates or actions.

By using scriptable objects to create a common event variable, two unrelated game objects can react to the same event without needing to know about each other.

Which allows you to create Unity Event style functionality, but between any objects in the scene.

What are scriptable objects?

Scriptable objects are persistent data containers that are stored in your project like assets.

Put simply, when you create a scriptable object, you're really defining a template from which you can create unique copies, where each copy is an instance of the scriptable object type, with its own data and that can be saved as an asset.

While they have many uses, one advantage of scriptable objects is that they can be used to create global variables of a specific type, including global game events, as demonstrated by Ryan Hipple in his **Unite Austin presentation in 2017**.

Here's how it works.

You'll need two scripts, a **scriptable object Game Event**,

Which looks like this:

```
using System.Collections.Generic;
using UnityEngine;
[CreateAssetMenu(menuName ="Game Event")]
public class GameEvent : ScriptableObject
{
    private List<GameEventListener> listeners = new List<GameEventListener>();
    public void TriggerEvent()
    {
        for (int i = listeners.Count -1; i >= 0; i--)
        {
            listeners[i].OnEventTriggered();
        }
    }
    public void AddListener(GameEventListener listener)
    {
        listeners.Add(listener);
    }
    public void RemoveListener(GameEventListener listener)
    {
        listeners.Remove(listener);
    }
}
```

And a regular **Monobehaviour Game Event Listener**,

Which looks like this:

```
using UnityEngine;
using UnityEngine.Events;
public class GameEventListener : MonoBehaviour
{
    public GameEvent gameEvent;
    public UnityEvent onEventTriggered;
    void OnEnable()
    {
        gameEvent.AddListener(this);
    }
    void OnDisable()
    {
        gameEvent.RemoveListener(this);
    }
    public void OnEventTriggered()
    {
        onEventTriggered.Invoke();
    }
}
```

The **Game Event** is the subject. It keeps a list of observers that are interested in the event and will trigger all of them if its event is called.

The **Game Event Listener** is an observer. It's not a scriptable object, it's a regular MonoBehaviour, but does hold a reference to a global **Game Event**, which it uses to subscribe its own trigger function to that event, so that it can be called by the subject when the event takes place.

The idea is that the subject and the observer both share the same global Game Event variable, connecting the two scripts.

However, before that can happen, you'll need to create a Game Event asset, by right-clicking in the project window

Like this:

To make a new scriptable object event, right click in the project window.

You'll need to do this every time you want to create a different game event as this is the specific event asset that both the subject and observers will share.

To use the event, declare a Game Event variable in a subject script.

Then, when you need to call it, trigger the Game Event's **Trigger Event** function.

Like this:

```
public GameEvent gameEvent;
void Start
{
    // Triggers the event
    gameEvent.TriggerEvent();
}
```

On any observing objects, add a Game Event Listener script.

Then, to connect the observer to the subject, set the game event fields on both scripts to the same Game Event asset that you created in the project.

Like this:

The subject and observing scripts will need to share the same Game Event asset.

Finally, connect the Unity Event on the Game Event Listener to set up the response when the event is triggered.

So, why use this method?

While it can take more work to set up, this approach combines the convenience of using Unity Events to manage local behaviours with the scalability of global events, such as event delegates.

And, while it's not without its drawbacks (passing arguments, for example, requires you to create a new type of event and listener) it allows you to manage how scripts interact with each other using the Inspector.

Which can help you to limit each script to a single purpose, making your game easier to manage and easier to finish.

Now it's your turn

Now I want to hear from you.

How are you using events and delegates in your game?

Are you using Unity Events with the Inspector, or maybe you're using delegates in your scripts?

And what have you learned about using events and delegates in Unity that you know others will find useful?

Whatever it is, let me know by leaving a comment.

by John Leonard French

Game audio professional and a keen amateur developer.

YouTube **johnlfrench** **johnleonardfrench.com**

My favourite time-saving Unity assets

Rewired (*the* best input management system)

Rewired is an input management asset that extends Unity's default input system, the Input Manager, adding much needed improvements and support for modern devices. Put simply, it's much more advanced than the default Input Manager and more reliable than Unity's new Input System. **When I tested both systems**, I found Rewired to be surprisingly easy to use *and* fully featured, so I can understand why **everyone loves it**.

DOTween Pro (should be built into Unity)

An asset so useful, it should already be built into Unity. Except it's not. **DOTween Pro** is an animation and timing tool that allows you to animate anything in Unity. You can move, fade, scale, rotate without writing Coroutines or Lerp functions.

Easy Save (there's no reason not to use it)

Easy Save makes managing game saves and file serialization extremely easy in Unity. So much so that, for the time it would take to build a save system, vs the cost of buying Easy Save, I don't recommend making your own save system since Easy Save already exists.

Comments

DECEMBER 17, 2021 **REPLY**

Adam

This is quite a comprehensive summary of the observer pattern in Unity. For UI, this is something I wish I had known a few years ago instead of relying on the singleton pattern so much. Thank you for the write up.

In the article, you mentioned health and UI as use cases. Would you be able to write an article on other good use cases and some bad use cases for this pattern?

I personally find that I avoid events, especially Unity events, for important logic since it makes things very difficult to debug. Things that are supposed to occur suddenly don’t and it’s hard to track down what should’ve happened.

I ran into one example today of this using Animancer that took some time to debug. I had a Cinemachine State-Driven Camera set up which switched cameras based on whether the character was crouched or standing, but all of a sudden the camera was no longer changing when I crouched. It turns out that Animancer removed the Animator from my Player, which in turn removed the state machine that was changing the camera. This was super hard to find because it was all set up in the inspector and just silently disappeared.

DECEMBER 20, 2021 **REPLY**

John French

Author

Thanks for the tips. Regarding other use cases, I don’t want to be too prescriptive, but I think that this kind of approach works best when you want something to respond to something else passively. I think it can become tricky when you need to create game behaviour between objects that are co-dependent, like having one object interact directly with another in some way. Thanks for sharing your experience on this.

MAY 17, 2022 **REPLY**

Muhammad

Thanks Friend Finally I found What I was Looking For !

DECEMBER 20, 2021 **REPLY**

Mohamed

Thank you I learn a lot from this article

DECEMBER 21, 2021 **REPLY**

John French

Author

You’re welcome!

DECEMBER 31, 2021 REPLY

Ahmed

Thank you, I really needed this.

DECEMBER 31, 2021 REPLY

John French

Author

You're welcome!

FEBRUARY 23, 2022 REPLY

HapiFox

Thanks,great helps to me!

FEBRUARY 23, 2022 REPLY

John French

Author

You're welcome!

MARCH 13, 2022 REPLY

Karolis

I'm new to unity and the resources provided in this article are awesome! I went ahead and bought the Game Programming Patterns book.

My concern with using UnityEvents is that I feel like having things done through the inspector is just a way to forget or not know where things are or what's actually happening. Maybe it's my inexperience with Unity, but to me using inheritance of MonoBehaviours seems like a better approach.

So for example you can have the same script ObjectHealth which would manage health and give it to whatever needs their health managed. If you want to give it extra functionality for your player – create PlayerHealth. You just inherit ObjectHealth and override whatever methods were there (including Update and Start, etc…)

What do you think of my approach?

MARCH 13, 2022 REPLY

John French

Author

Thanks for your feedback, you're right. Ultimately, I feel like they're different methods of solving the same problem. I think that the Unity Event approach is useful for someone who likes to work in the Inspector, but that comes with drawbacks. It takes some setting up and the connections are harder to trace through code. The idea is to make things easier so, if it doesn't do that for you, absolutely do something else and your approach sounds like a good one too.

DECEMBER 28, 2022 REPLY

Ujjawal Saini

This was good . BUT can you please explain more about events in unity . How to register , unregister with examples

---

MARCH 13, 2022 REPLY

Dominique

Man, another quality article right here. I refer to your articles on a regular basis even though I've got almost two years of experience with Unity and C#.

There are still a lot of things I don't know and using events was one of them. Been working on a mobile game and have a lot of stuff depending on some events like when it is game-over or when a new character is purchased etc.

Thank you so much for these 🙂

MARCH 13, 2022 REPLY

John French

Author

You're welcome! So glad to hear they're helpful.

MARCH 17, 2022 REPLY

Johann Hiro

This article is live saving! Finally a tutorial of event in Unity that I could actually understand. These comparisons are really informative and helpful for understanding. I finally manage to fix some stupid object references in my project. Thank you so much for this!

MARCH 17, 2022 REPLY

John French

Author

You're welcome, I'm really glad to hear it was helpful!

APRIL 23, 2022 REPLY

Ali Ashrafi

A perfect article. Thank you so much mate!

APRIL 24, 2022 REPLY

John French

Author

You're welcome!

MAY 15, 2022 REPLY

Scully

Thanks for sharing your knowledge! I came in expecting to learn about delegates and ended up learning about so much more. This is very cool, I use Unity Events all the time in the inspector and never knew it was so easy to make my own. My project is going to look so much fancier now. This was super easy to understand by the way, great job. Really appreciate this!

MAY 16, 2022 REPLY

John French

Author

Thank you! I'm so glad to hear it helped.

MAY 21, 2022 REPLY

Daniel

Man, I missed reading an article as good as yours, congrats, it was amazing.

MAY 22, 2022 REPLY

John French

Author

Thanks so much!

MAY 28, 2022 REPLY

Ant

Are you a professional instructor!?? This is probably one of the BEST tutorials on any coding subject I've ever seen. You clearly delineate WHY you're supposed to do things in a logical step by step fashion, introducing new and progressively more complex concepts in an extremely helpful way. I wish everyone could teach like this. It takes both a real understanding of the material and the way people think to be able to teach like way. Your fan forever.

MAY 28, 2022 REPLY

John French

Author

You're too kind. Thank you so much for your generous feedback, I appreciate it. Great to hear you enjoyed the article.

MAY 31, 2022 REPLY

Foehammer

you are great bro , veryyyyyyyyyyyyyyyyy nice .

JUNE 4, 2022 REPLY

John French

Author

Thank you!

JUNE 9, 2022 REPLY

Jackie

Article quality: 10
Reading experience: 2.

Found the overuse of video ads/pop-ups/constantly changing ads every few paragraphs to be extremely distracting. I found it really difficult to finish the article.

JUNE 9, 2022 REPLY

John French

Author

Thanks for your feedback on this, the ads keep the site free for everyone, which is important to me, but so is the reading experience. I appreciate your feedback and I'll do what I can to improve this in the future.

JUNE 24, 2022 REPLY

JK_Bizcuits

Man, this article is great. I've been kind of circling this subject for over a week trying to sort out a dilemma in my project and this really helped me out. I kept watching so many videos that seemed to focus on just parts of this subject but never together as one grand picture with proper pros and cons. Really easy to understand even with my smooth brain and now I can look at all these other tutorials and understand them in the proper context. Thanks for writing this fam and keep it up. (^ ^) b

---

JUNE 26, 2022 REPLY

John French

Author

You're welcome! So glad to hear it helped you!

JULY 26, 2022 REPLY

Wangwang

Thank you so much. This helps me a lot.

JULY 26, 2022 REPLY

John French

Author

You're welcome!

AUGUST 15, 2022 REPLY

travhimself

This is by far the clearest and most concise summary on this topic that I've seen.

I'm an experienced web dev, but new to c# and Unity. I'm trying to make some good architecture decisions, and I've been looking for EXACTLY this sort of write-up to help me understand my options.

AUGUST 15, 2022

REPLY

John French

Author

Really happy to hear it was helpful, thanks so much!

OCTOBER 11, 2022

REPLY

Chris

You are the best

OCTOBER 12, 2022

REPLY

John French

Author

Thanks so much

OCTOBER 14, 2022

REPLY

Nl You

Thank you so much for sharing all this knowledge.
I'm a beginner. I think this is the best article I can find on the Internet. It really helps me to understand how events and delegates events work.

OCTOBER 14, 2022

REPLY

John French

Author

You're welcome, really happy to hear it's helped you.

OCTOBER 25, 2022

REPLY

Avi K

Hands down the best article on the Internet for understanding delegates and events in general in C#. You are precise and to the point, while being comprehensive enough to provide all aspects of the subject required to actually use it. Coming from other languages that are not well typed, delegates in C# and Unity development in particular were insanely foreign concepts to me. After reading about a dozen articles before this one and finally stumbling across yours, I now fully understand the benefits and uses of these concepts. Thank you again!

OCTOBER 25, 2022

REPLY

John French

Author

Thanks so much! I'm really happy to hear it helped.

JANUARY 27, 2023

REPLY

Klaus Jensen Jåtog

Great article with comprehensive explanations and easy to understand examples. Love it. Really appreciated.

JANUARY 30, 2023 REPLY

John French

Author

Thanks so much!

JANUARY 30, 2023 REPLY

The Wizard

Thank you!

JANUARY 30, 2023 REPLY

John French

Author

You're welcome!

FEBRUARY 9, 2023 REPLY

Thomas S

I normally don't comment on articles and such, but I wanted to say Thank you, I was struggling with this topic for a while and this helped me understand it better. Can't wait to try this out!

FEBRUARY 9, 2023 REPLY

John French

Author

You're welcome, happy to help!

FEBRUARY 16, 2023 REPLY

Bagus Harisa

I am amazed by the article's comprehensibility to be understood. Good job John. You help us all. After all these years, I finally understand the Unity's delegates and more.

FEBRUARY 16, 2023 REPLY

John French

Author

Thank you! I'm so happy it helped you!

MARCH 2, 2023 REPLY

Christian M

Great info. I stumbled upon this will looking trying to figure out best way to deal with dependencies in a project. My issue with the Scriptable Objects approach is that you're loosing the ability to follow the flow of the code. Considering in a full project you'll have more than a handfull of the same event type, you'll will most likley have an annoying time finding who is calling the specific event you're curious about. I don't think there's any out of the box tools to see what objects are referencing a specific SO instance either?

MARCH 2, 2023 REPLY

John French

Author

Thank you! You're right that's definitely a risk of using Scriptable Objects and one of the reasons that some people *don't* like using them.

MARCH 3, 2023 REPLY

Karly

In the Unity Events section, you mentioned they could be used for objects that share the same trigger but produce different results. How would that work? Unless I missed it, I do not think you explained that. Thank you ahead of time for answering my question.

MARCH 3, 2023 REPLY

John French

Author

Hi! So what I meant by that is that a single output from a script, such as a Unity Event, could then be connected manually, in the editor, to different other scripts and functions. Meaning that two instances of the same script could each do different things, even though their code is the same. In the same way that a Button uses Unity Events to trigger different things, but with your own scripts instead. Hope that answers your question, but let me know if not, either here or at **support@gamedevbeginner.com**.

MARCH 3, 2023 REPLY

Karly

Thank you for the quick response! I was mainly referring to your example with the player, an enemy, and an object sharing the health script. How would the damage be differentiated if they are all connected to the same event? How would one object take damage at one point but remain unaffected when another is damaged? Would a parameter need to be included that specifies who takes damage at a given time that every connected function checks before executing?

MARCH 3, 2023 REPLY

John French

Author

Ah I see what you mean. So in that example, the scripts that trigger the events are local instances. So a player event wouldn't affect the enemy's script at all. There's one for the player, one for the enemy etc. for as many instances of it as you want to use. This is different to Scriptable Object events which are typically global, but still use Unity Events as an endpoint, again because Unity Events are usually local and only really apply to the object they're on. Hope that helps.

Leave a Comment

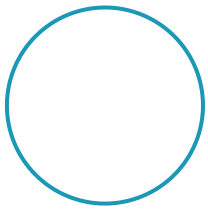
Your Comment *

Your Name *

Your Email *

Submit

WELCOME TO MY BLOG



I'm John, a professional game composer and audio designer. I'm also a keen amateur developer and love learning how to make games. **More about me**

LATEST POSTS

Addressable Assets in Unity

Async in Unity (better or worse than coroutines?)

State Machines in Unity (how and when to use them)

How to use Arrays in Unity

How to delay a function in Unity

THANKS FOR YOUR SUPPORT

Some of my posts include affiliate links, meaning I may earn a commission on purchases you make, at no cost to you, which supports my blog.

For more information view my **Affiliate Policy**.