KennethDevelops'

# EventManager

v2.0.0

# Table of Contents

# Introduction

EventManager is a tool designed to streamline and enhance communication between the elements of your game logic, making it simpler to notify key events to your GameObjects.

Unlike other tools, EventManager's unique features include:

- **Easy to use:** Just one line of code to subscribe, unsubscribe, and trigger events.
- **Plug & Play:** From purchase to implementation in your game in just 10 minutes.
- **Open Source:** The code is fully modifiable and extendable to suit your needs.

EventManager addresses one of the biggest challenges in game development, interconnecting different systems to work together as a unified whole, without creating dependencies. For instance, when an enemy dies, numerous systems should be aware of this event. This includes the audio system for sound effects, the enemy spawn system for enemy count, the reward system for loot drops, and so on. Traditional solutions create dependencies, meaning when an enemy dies, it'll call all these systems. This makes it so if one system changes, the developer will have to go through every script looking for these calls to modify them to accommodate the new requirements. With EventManager, such dependencies are a thing of the past. The enemy script will only "inform" everyone about the event "OnEnemyDied", and any script interested in that event will subscribe to it and react accordingly.

## Support

For any issues, queries, or feedback regarding the product, kindly reach out to us at kennethdevelops+support@gmail.com. Please provide a detailed explanation of the issue, including images, gifs, and/or video, if it can aid in understanding the problem better.

Remember, your feedback is invaluable in improving the product. So, don't hesitate to get in touch with us.

# Getting Started

EventManager is a robust tool designed to streamline and enhance communication within your game's logic. Its simplicity, ease of use, and 'plug-and-play' nature make it the perfect solution for managing complex events and interactions.

## Importing EventManager

After purchasing EventManager from the Asset Store, you'll be prompted to download and import the Asset. Upon completion of the import, the Unity Editor will load the asset package, and you'll be all set to start using EventManager!

## Creating Custom Events

Before diving into subscribing and triggering events, let's create a custom event. EventManager makes this process simple. To create a custom event, extend the base class `BaseEvent`:

```
public class OnMyEvent : BaseEvent<OnMyEvent> { }
```

For example, if you're working on a game where the player can collect items, you might create an OnItemCollected event:

```
public class OnItemCollected : BaseEvent<OnItemCollected>

{

    public Item collectedItem;

}
```

In this case, the OnItemCollected event contains information about the collected item. We will subscribe to this event in the next section.

You can learn more about creating your own Events here

## Basic Actions

EventManager operates primarily through three actions: Subscribe, Trigger, and Unsubscribe. Here's a brief guide on how to use them:

### Subscribe to Events

This action allows you to listen for specific events. For example, to subscribe to the OnItemCollected event, use the following code:

```
void Awake()

{

    EventManager.Subscribe<OnItemCollected>(OnItemCollectedHandler, this);

}
```

## Trigger Events

This action initiates the event, notifying all subscribed listeners. To trigger the OnItemCollected event, use the following code:

```
public void OnTriggerEnter(Collider collider)

{

    new OnItemCollected { collectedItem = item }.Trigger();

}
```

## Unsubscribe from Events

This action removes the listener from specific events. To unsubscribe from the OnItemCollected event, use the following code:

```
void OnDestroy()

{

    EventManager.Unsubscribe<OnItemCollected>(OnItemCollectedHandler);

}
```

Check out the other pages in this wiki for more examples and detailed instructions on how to use EventManager effectively.

# Creating your own Events

EventManager offers a powerful way to customize events, allowing you to communicate any type of information across different systems within your game logic with ease.

This customization is pivotal as it caters to various scenarios in your game and specific requirements which can vary based on the particular system listening to the event. For example, an `OnEnemyDied` event might need to convey the enemy's position, type, and who killed it. With EventManager, you can tailor your events to send the exact data needed for each scenario.

## Simple Steps

Creating your own events in EventManager is a straightforward process. You can create an event by extending the base class `BaseEvent`, as demonstrated below:

```
public class OnMyEvent : BaseEvent<OnMyEvent>
{

}
```

This is the basic structure of a custom Event in EventManager. However, the real power comes from being able to specify the data the event carries. You can add fields to your event to store this data like any regular class:

```
public class OnMyEvent : BaseEvent<OnMyEvent>
{
    public int intValue;
    public float floatValue;
    public string stringValue;
}
```

In this example, we've added fields for integer, float, and string data, but you have the freedom to use any data types you need. An event can include other classes, structs, pointers, and more; it is simply a regular C# class. For example, an OnItemCollected event might carry data about the item's type, its rarity, and the player's position when the item was collected:

```
public class OnItemCollected : BaseEvent<OnItemCollected> {
    public ItemType itemType;
```

```
    public ItemRarity itemRarity;
    public Vector3 playerPosition;
}
```

In case you prefer not to expose your fields publicly, you can use private fields and access them via properties/getters:

```
public class OnMyEvent : BaseEvent<OnMyEvent> {
    private int intValue;
    public int IntValue { get { return intValue; } set { intValue = value; } }

    private float floatValue;
    public float FloatValue { get { return floatValue; } set { floatValue = value; } }

    private string stringValue;
    public string StringValue { get { return stringValue; } set { stringValue = value; } }
}
```

## Event Naming Best Practices

While it's not mandatory, it's recommended to follow a consistent naming pattern for your events. Typically, you can use the "On[event verb past]" format, such as OnGameStarted, OnEnemyDied, or OnPlayerHealthUpdated. This naming convention not only promotes consistency but also provides clear understanding of when the event is expected to be triggered.

Remember, EventManager is designed to provide you with the flexibility to meet your game's unique requirements. As such, there are no restrictions on the types or number of fields you can add to your custom events.

# Subscribing to Events

EventManager offers flexible options for subscribing to events, tailored to fit your specific needs. This page walks you through these options and how to use them effectively.

## Basic Subscription

The simplest way to subscribe to an event looks like this:

```
EventManager.Subscribe<OnMyEvent>(MyEventHandler, this);
```

In this case, the `MyEventHandler` method gets called when the `OnMyEvent` event gets triggered. Here's what the `MyEventHandler` method might look like:

```
private void MyEventHandler(OnMyEvent e)
{
    // Perhaps the event carries data about an enemy that has been slain
    Debug.Log("Enemy " + e.enemyID + " was slain!");
}
```

Note that the `callback` (here `MyEventHandler`) has a void return type and takes `TEvent` as a parameter.

## Prioritized Subscription

If you need to ensure that your event handler gets executed before or after others, you can set a `priority`:

```
EventManager.Subscribe<OnMyEvent>(MyEventHandler, this, 0);
```

In this example, the `MyEventHandler` gets called before others with higher priority values. The default priority is `100`, and lower values get called sooner.

For instance, in a game where a player can collect items, you might have an inventory system that needs to process the item collection before an audio system plays a sound:

```
// In the inventory system script
EventManager.Subscribe<OnItemCollected>(HandleItemCollected, this, 0);

...

private void HandleItemCollected(OnItemCollected e)
```

```
{
    // Process item collection
}

// In the audio system script
EventManager.Subscribe<OnItemCollected>(PlayItemCollectedSound, this, 100);

...

private void PlayItemCollectedSound(OnItemCollected e)
{
    // Play sound
}
```

## Targeted Subscription

EventManager also allows you to subscribe to events from specific objects:

```
EventManager.Subscribe<OnMyEvent>(MyEventHandler, this, sender);
```

In this case, `MyEventHandler` only gets called when the object `sender` triggers the `OnMyEvent` event.

For example, if you have a game with multiple enemies and a specific action should only happen when a particular enemy dies, you could use targeted subscription:

```
// This will only react to 'OnEnemyDied' events triggered by 'specificEnemy'
EventManager.Subscribe<OnEnemyDied>(HandleSpecificEnemyDeath, this, specificEnemy);

...

private void HandleSpecificEnemyDeath(OnEnemyDied e)
{
    // Special action for this enemy's death
}
```

## Targeted and Prioritized Subscription

Finally, you can combine targeted and prioritized subscription:

```
EventManager.Subscribe<OnMyEvent>(MyEventHandler, this, sender, 150);
```

In this example, `MyEventHandler` only gets called when `sender` triggers the `OnMyEvent` event, and it gets called after all other handlers with priority below `150`.

```
// In a game with a scoring system and a UI that displays the score
// The scoring system updates the score and then the UI reads the new score and updates the display

// In the scoring system script
EventManager.Subscribe<OnPointsScored>(UpdateScore, this, player, 0);

...

private void UpdateScore(OnPointsScored e)
{
    // Update score
}

// In the UI script
EventManager.Subscribe<OnPointsScored>(UpdateScoreDisplay, this, player, 100);

...

private void UpdateScoreDisplay(OnPointsScored e)
{
    // Display new score
}
```

You can also do this by calling the Subscribe method from the object itself (using an extension on Plugins.KennethDevelops.Extensions):

```
sender.Subscribe<OnMyEvent>(MyEventHandler, this);
```

And you can optionally add a priority:

```
sender.Subscribe<OnMyEvent>(MyEventHandler, this, 0);
```

## Advanced Sender Filtering

Sometimes, you might want to filter event triggers based on more complex criteria than simply the sender. In these cases, you can access the sender (stored as a C# object) and cast it to the specific type you're interested in:

```
private void MyEventHandler(OnMyEvent e)
```

```
{
    if (e.sender is Enemy)
    {
        // Only handle event if it was triggered by an Enemy
        Enemy enemy = e.sender as Enemy;
        Debug.Log("Enemy " + enemy.name + " triggered an event!");
    }
}
```

In the example above, the event handler `MyEventHandler` will only process the event if it was triggered by an object of type `Enemy`. This allows for more sophisticated filtering based on the type of object triggering the event.

However, the sender filtering can be extended even further. You can filter by any other attribute of the sender, such as their game object, layer, or any other property that may be necessary. The only requirement is that you need to know the type of the sender to access its fields, methods, etc.

Here are a couple of additional examples:

```
private void MyEventHandler(OnMyEvent e)
{
    if (e.TryGetSenderGameObject(out GameObject senderGameObject))
    {
        // Only handle event if the sender is in Layer "Enemies"
        if (senderGameObject.layer == LayerMask.NameToLayer("Enemies"))
        {
            Debug.Log("GameObject in the Enemies layer triggered an event!");
        }
    }
}
```

```
private void MyEventHandler(OnMyEvent e)
{
    if (e.sender is Player senderPlayer)
    {
        // Only handle event if the sender has less than 50% health
        if (senderPlayer.Health < senderPlayer.MaxHealth / 2)
        {
            Debug.Log("Player with less than 50% health triggered an event!");
        }
    }
}
```

The first example uses the `TryGetSenderGameObject` method provided by the BaseEvent class. This method attempts to retrieve the GameObject of the sender, whether the sender is a GameObject itself, or a component derived from a GameObject (such as a MonoBehaviour). This makes it easy to filter events based on GameObject properties like its layer.

In the second example, the sender is a `Player` and the event is only handled if the player's health is below 50%.

## Unsubscribing

After subscribing to events, there may come a time when you no longer want to listen for those events. In that case, you would need to unsubscribe. See the Unsubscribing page for details on how to do this.

# Unsubscribing from Events

There are a few ways you can unsubscribe from an event with the `EventManager`:

## Unsubscribing a Specific Callback

If you want to stop a specific callback from being called when an event is triggered, you can unsubscribe that particular callback. For example, if you have a method called `HandleEnemyDefeated` that was subscribed to an `EnemyDefeatedHandler` event, you can unsubscribe it like so:

```
EventManager.Unsubscribe<OnEnemyDefeated>(EnemyDefeatedHandler);
```

## Unsubscribing All Callbacks from a Subscriber

If you want to stop all callbacks from a specific subscriber from being called when an event is triggered, you can unsubscribe the subscriber itself. This will remove all callbacks associated with that subscriber. Here's an example:

```
EventManager.Unsubscribe(this);
```

The `EventManager` will automatically unsubscribe any Unity object (MonoBehaviours, Components, etc.) that has been destroyed. This helps avoid undesired behaviours and errors. However, it's still good practice to manage your subscriptions and ensure you unsubscribe when appropriate.

## Best Practices for Subscription Management

When working with events, it's generally best to subscribe in the `Start` or `Awake` method and unsubscribe in the `OnDestroy` method. This ensures that your object only listens to events while it's active in the game world. Here's an example:

```
private void Start() {
    EventManager.Subscribe<OnPlayerHealthChanged>(HandlePlayerHealthChange);
    Debug.Log($"[EventManager] {this.GetType().Name} subscribed to
{nameof(OnPlayerHealthChanged)} event.");
}

private void OnDestroy() {
    EventManager.Unsubscribe<OnPlayerHealthChanged>(HandlePlayerHealthChange);
```

```
    Debug.Log($"[EventManager] {nameof(HandlePlayerHealthChange)} unsubscribed from
{nameof(OnPlayerHealthChanged)} event.");
}
```

However, there may be times when an object needs to stop listening to an event before it's destroyed, such as when a player character dies or an enemy is defeated. In these cases, it would be appropriate to unsubscribe in the method handling that logic.

# Triggering an Event

Triggering an event in EventManager is quite straightforward. You simply create a new instance of your event, fill in any necessary data, and call the `Trigger` method on it, passing the sender as a parameter. Here are a few examples:

## Simple Event Trigger

To trigger a simple event that does not require any additional data, you can do:

```
new OnGameStart().Trigger(this);
```

In this example, the `OnGameStart` event is triggered. This can be useful to signal the start of a game, which might cause subscribed listeners to reset their states, play an intro animation, etc.

## Event Trigger with Data

If your event class contains properties for carrying data, you can set these properties when creating the event, and then trigger it like this:

```
new OnEnemyDefeated{ EnemyType = "Goblin", LootDropped = "Gold" }.Trigger(this);
```

In this example, the `OnEnemyDefeated` event is triggered with two pieces of data: `EnemyType` and `LootDropped`. This can be useful in an RPG game, where defeating an enemy might result in a loot drop and various game systems might need to know the type of enemy that was defeated to determine the proper response.

## Declaring the Event Separately

Sometimes, it might be convenient to declare the event and fill in its data separately from triggering it. This can be done like so:

```
var levelUpEvent = new OnLevelUp();
levelUpEvent.Level = 5;
levelUpEvent.ExperienceNeeded = 1000;

levelUpEvent.Trigger(this);
```

In this example, we create a new `OnLevelUp` event, set its `Level` and `ExperienceNeeded` properties, and then trigger it. This might be used in an RPG game, where leveling up would change various game stats and might trigger a variety of game responses.

# Best Practices

1. Always include the sender when triggering an event. This allows subscribers to know the source of the event.

2. Use meaningful names and data properties for your events. This makes it easier for others reading your code to understand what the event represents and how it should be used.

3. Avoid triggering events in tight loops or performance-critical sections of your code. While the EventManager is optimized for performance, unnecessary event triggering can still lead to performance issues.

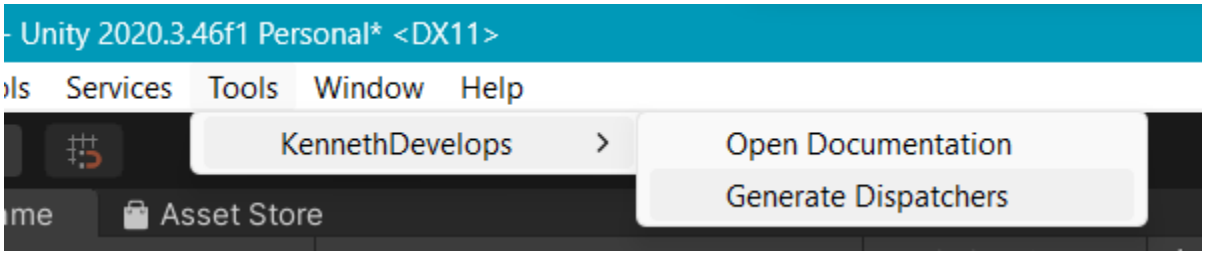
4. Consider the order of event triggering carefully. When multiple events are triggered in a sequence, the order in which they are triggered can have significant effects on the behavior of your game.

# Event Dispatcher

## EventDispatcher in Unity

### Overview

The **EventDispatcher** is not a component by itself but a series of components generated manually through `Tools/KennethDevelops/Generate Dispatchers Menu`. These components are automatically created for every class derived from `BaseEvent` that you have defined in your project.

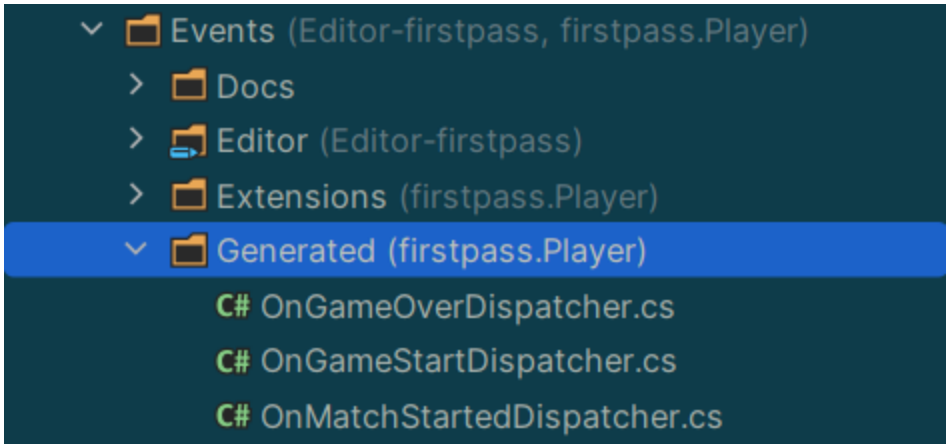



**Note**: Each class derived from `BaseEvent` needs to have the `[Serializable]` attribute, and all its public fields should be serializable for this to work, like this:

```
[Serializable]
public class OnMatchStarted : BaseEvent<OnMatchStarted> { }
```

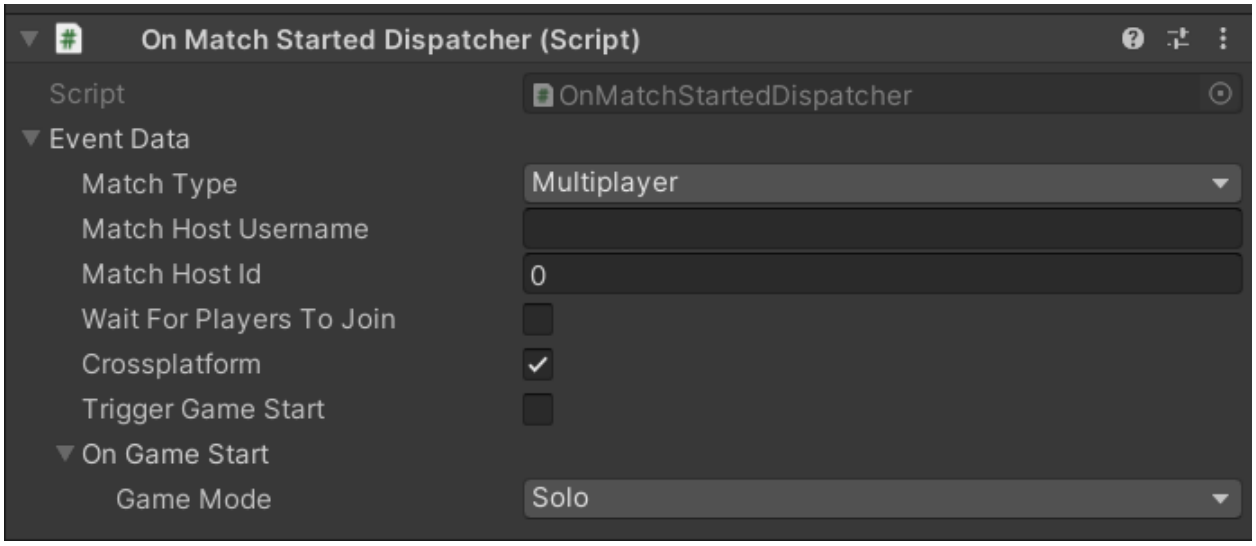
## How to Generate Event Dispatchers

1. Go to `Tools/KennethDevelops/Generate Dispatchers Menu` in the Unity editor.
2. Click on the `Generate Dispatchers` button.
3. Unity will create an `EventDispatcher` for each `BaseEvent` derived class in your project.



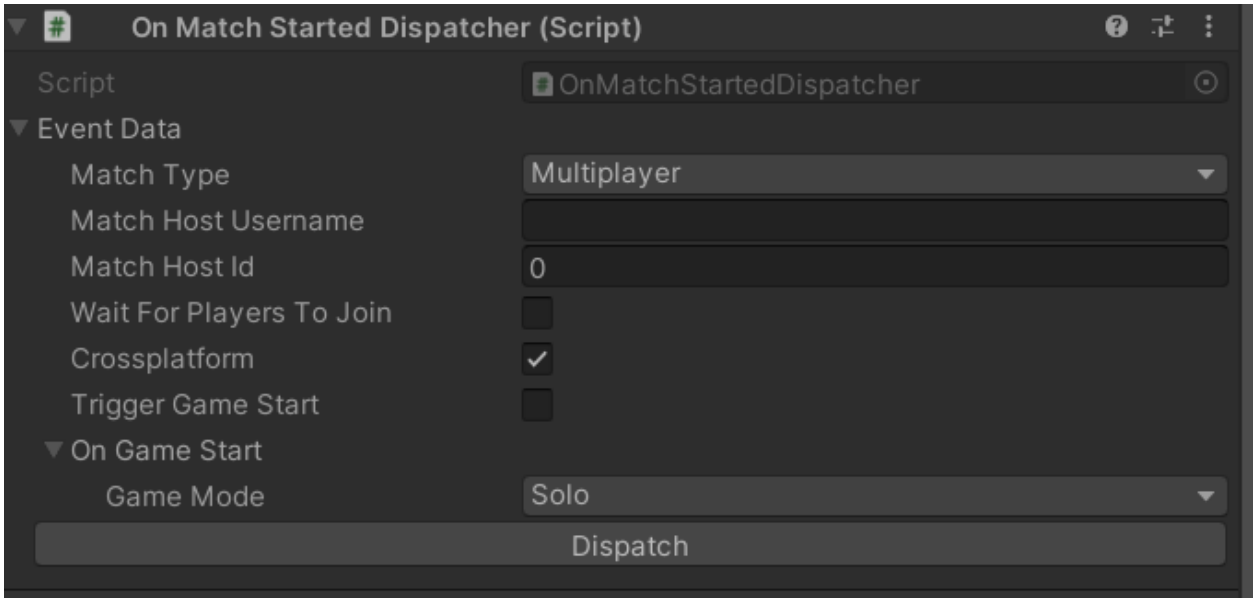

## eventData Field

Each generated `EventDispatcher` component will have a public field called `eventData`. This field contains all the data for the event and can be edited through the Unity Inspector. The data must be serializable by Unity, including but not limited to types like `string`, `int`, `float`, `bool`, `Vector3`, `Quaternion`, etc. Custom classes with the [Serializable] attribute can be used as well.




## Dispatch Method

Each `EventDispatcher` has a button called `Dispatch` which becomes enabled only in play mode. Clicking this button will manually trigger the event with the current `eventData`. This is particularly useful for debugging purposes.




To programmatically dispatch the event, you can call the `Dispatch()` method from another script.

## Further Reading

For more advanced scenarios where you might want to execute methods conditionally based on various triggers, consider using `DevUtils' ExecuteOn`.

## Reminder

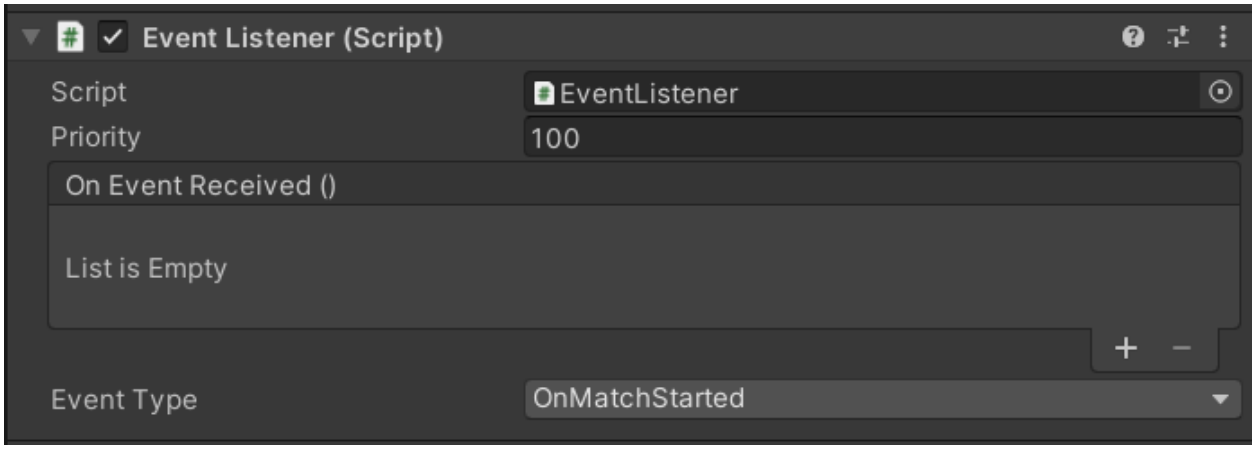
All classes in the Event Manager plugin are open-source and can be extended and edited by anyone.

# Event Listener

The `EventListener` is a powerful Unity component that allows you to subscribe to specific events in the Event Manager system directly from the Unity Inspector. This component can be attached to any GameObject and eliminates the need for writing extra scripts for simple actions triggered by events.

## Overview

The `EventListener` subscribes to a specific event derived from the `BaseEvent` abstract class during its `OnEnable()` method. When the event is triggered, a UnityEvent is called, allowing you to execute methods or change values directly from the Unity Inspector.



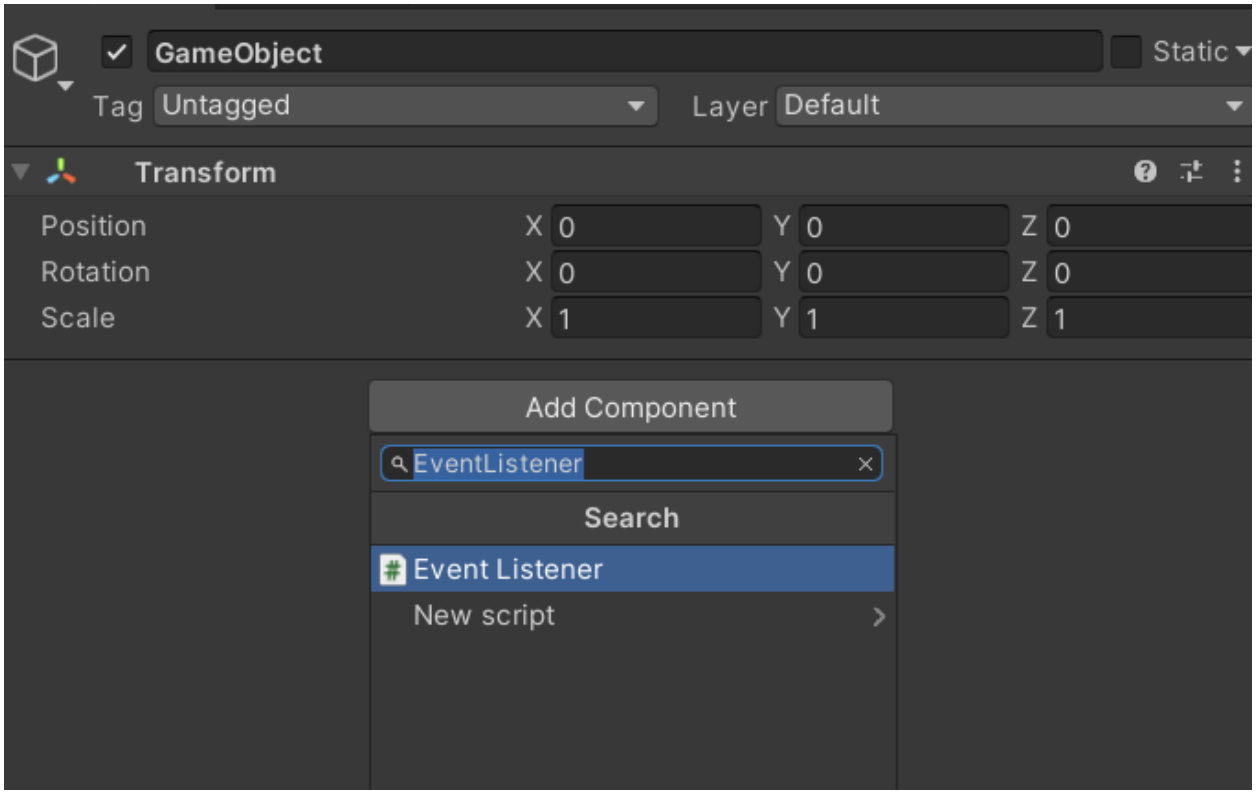

**Note**: Each `EventListener` can only subscribe to one event. If you wish to subscribe to multiple events, simply add multiple `EventListener` components to your GameObject.

**Note**: All classes in the Event Manager plugin are open source and can be modified or extended as needed.

## How to Use

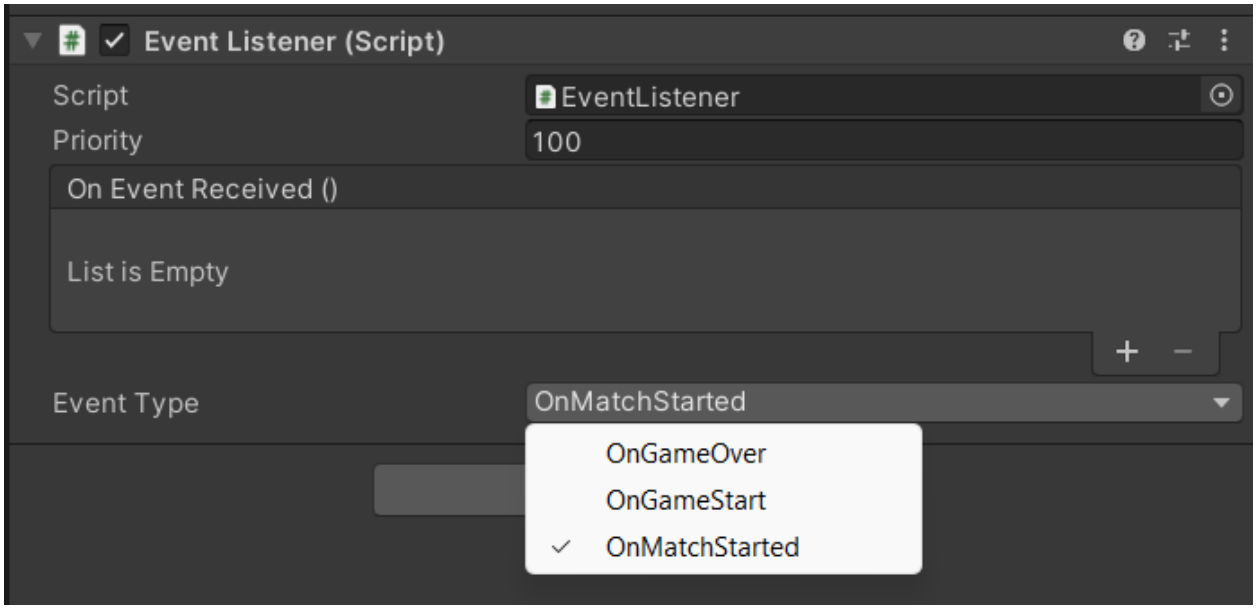
### Step 1: Add the EventListener Component

Attach the `EventListener` component to a GameObject from the Unity Inspector.



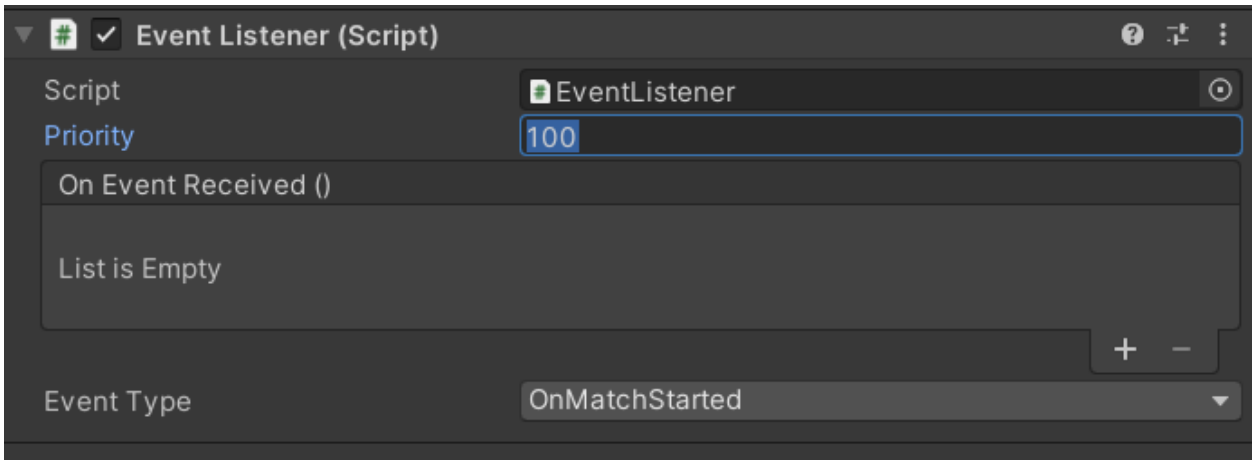

## Step 2: Choose an Event to Subscribe

Select an event derived from the `BaseEvent` abstract class in the Event Manager system.



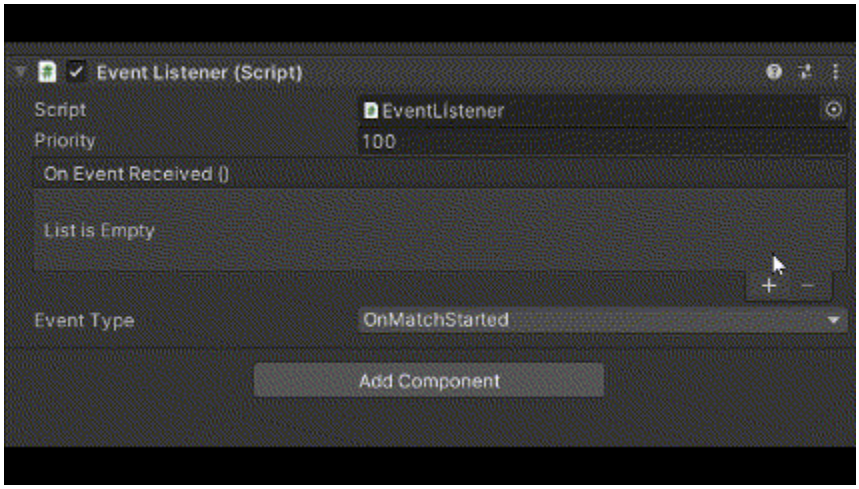
### Step 3: Set the Priority

Set the callback priority for this `EventListener`. Lower values get called before higher values. The default priority is 100.




### Step 4: Attach UnityEvent Actions

In the Unity Inspector, attach the methods or change the values you wish to be triggered when the event occurs.



# Use Cases

## Game Over Actions

Suppose you want to disable a group of enemies and display a "Game Over" UI panel when the player's health reaches zero. You could set up an `EventListener` to listen for an `OnGameOver` event, and then attach methods to disable the enemies (the parent game object perhaps) and display the GameOver UI panel.

## Timed Power-ups

Imagine your game has a power-up that lasts for a limited amount of time. You could use an `EventListener` to listen for an `OnPowerUpExpired` event and revert the player's stats when the event is triggered.

# Unsubscribing from Events

The `EventListener` automatically unsubscribes from its event during its `OnDisable()` method. You can disable the GameObject or the `EventListener` component itself to unsubscribe.