This package is created for raising and handling events easily without making things complicated.

Before Starting

Firstly, you need to add "**Event Manager**" prefab to your starting scene (the very first scene that loads at the beginning of your game).

You can use it in two ways:

1- Using components
2- Using by code

After that, you need to add your events into "**Events List**" asset which is located under "**Clean Events**" folder. After adding your events, by clicking "**Generate Enum**" it will make an `enum` file which contains all of your events. Because this system uses C# `enum` file, event names must follow C# naming convention, which means start a name with letters or underscore, then you can use letters, numbers and underscore, otherwise you will get an error and will not be able to generate enum file.

Using `enum`, has multiple advantages like:

- You don't lose track of your events in your code
- If you remove an event you will get compile-time error rather than runtime exception
- You can easily use `ListenTo` attributes and bind public methods to events cleanly
- Have faster runtime code, because enum comparison operation is much faster than string's

Note: Because Unity serializes enums as integers, changing order of events when using components might result in changing values on your event components, so if you removed an event go back and check it wherever you used it or as I highly recommend, do not remove events from events list, just append new ones. (If you use code, it doesn't matter)

Using components

Components are useful when you are designing UI. You can simply show/hide screens, menus and UI objects by listening to events. You can also raise events by using EventRaisers.
There are 2 types of components, listener and raisers.
Note: To see some demo, check Demo scene.

Event Raiser Components

By adding **EventRaiserSingle** to a gameObject and choosing an event, you can raise that event by calling Raise() from that component (i.e. from other components like UI Button or codes from you game).

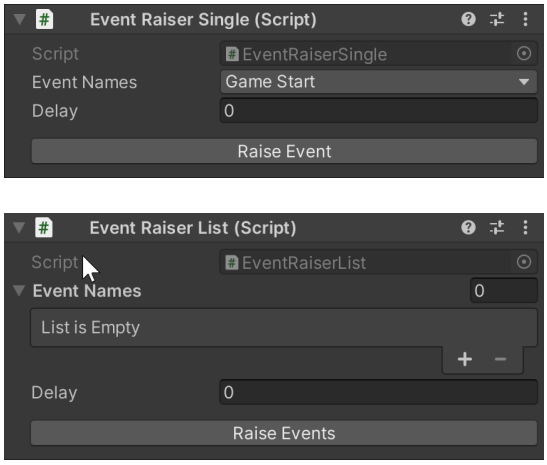
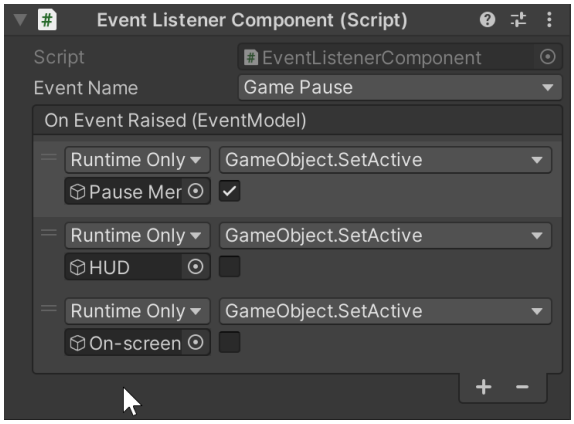
EventRaiseList works just like **EventRaiserSingle** but it can raise a list of events.

Note: There is a button on the component for you to test your event, although it works only when Unity is playing.

Event Listener

By adding **EventListener** to a gameObject, it will listen to that event and invoke "OnEventRaised" method list. It can be used in many ways, such as activate/deactivate a gameObject.

Following tables shows these components.

Event Components	
Event raisers	Event Listener
	

Using by code

Note: To see a sample code, check SampleCode.cs file.

Raising Events

To raise an event, simply call either of these methods:

```
EventsManager.RaiseEvent(EventName);

Or

EventsManager.RaiseEvent(EventModel); // you can even pass data here
```

Listen to events

To listen to events, all you need is to mark your `public` methods with `ListenTo` attribute. All event listener can be an action (a method without input parameter) or have an input parameter of `EventModel`. You can optionally, add a priority value to this attribute, so methods with **higher** priorities will be called **first** (by default, priority is 0). Here is an example:

```
[ListenTo(EventName.LevelStart, priority: 100)]
public void OnLevelStarted(EventModel eventInfo)
{
	// your codes here
}
```

By using `ListenTo` attribute, all marked methods on your `MonoBehaviour` objects will be found and added to list of listeners during `EventManager.Awake()`, but if you create objects during runtime or you want a `non-MonoBehaviour` object (i.e. POCO) to listen to a specific event you can add that object with code below:

```
EventsManager.Register(SampleMonoBehaviour );

Or
```

```
EventsManager.Register(new SomePocoClass());
```

So, all methods with `ListenTo` attribute in component object will be added to listeners list.

But if you want a single method to listen to an event, you can use code below. In this case, that method can be private.

```
EventsManager.AddListener(EventName, this, Listen);

public void Listen(EventModel model)
{
	// your codes here
}
```

Unsubscribe from events

Note (Important!): Don't forget to call following methods to unsubscribe from events when objects being destroyed or disposed, otherwise, you will get events even after objects destroyed! (Until Garbage Collector destroys them). You can find sample codes in SampleCode.cs and Event Listener components.

```
private void OnDestroy()
{
	EventsManager.RemoveListener(OBJECT)
	Or
	EventsManager.RemoveListener(EventName, OBJECT, ACTION);
}
```

`Note:` For the sake of simplicity, `EventBehaviour` class does above operations for you, you just need to drive your classes from `EventBehaviour` instead of `MonoBehaviour`.

Passing Data

To pass data, use **Payload** object in `EventModel`, here is descriptions:

```
public EventName EventName;
public object Sender;
public float Delay = 0;
public bool IsHandled = false;
public object Payload;
```

`EventName`: name of event which is defined in "Events List" asset and respective enum.
`Sender`: the object which raised the event.
`Delay`: holds the raising operation for defined value, in seconds.
`IsHandled`: If any of the listeners sets this to true, other objects won't get the event! So if you want use it, make sure your listener is added to list of listener before others.
`Payload`: You can use this object to send your data, however, you need to cast it to your desired type.