
KennethDevelops
EventManager
PRO

Thank you for purchasing EventManager!

EventManager is a tool that helps you ensure communication between the elements on your game logic. In other words, it makes it easier to **notify key events to your GameObjects**.

For example, **when the game is over**, you want to **notify every enemy to stop attacking** the player. With EventManager, you can simply **trigger the event** "OnGameOver" and **every element in the game that is subscribed** to this event (in this case, the enemies) **will be notified** and react accordingly.

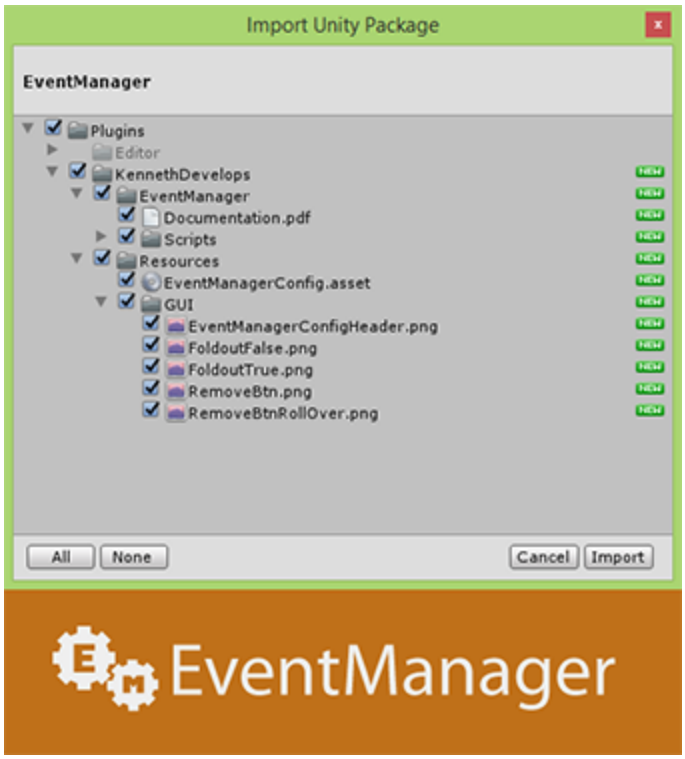
This document intends to serve as a guide to use and master this Asset Package. If possible, please use the online version of this document instead, it may often be updated.

Click here to go to the online version

Table of Contents

Getting Started

Just after purchasing EventManager in the Asset Store you should be prompted to download and import the Asset and then the following dialog will appear:



It is **important** that the location of the "KennethDevelops" folder remains inside the "Assets/Plugins" folder. Otherwise, the asset may not work properly.

After the import is done, you'll see the Unity Editor loading the asset package for a few seconds and then you'll be ready to begin using EventManager!

Check the other pages in this wiki to see examples about how to use the asset properly.

Support

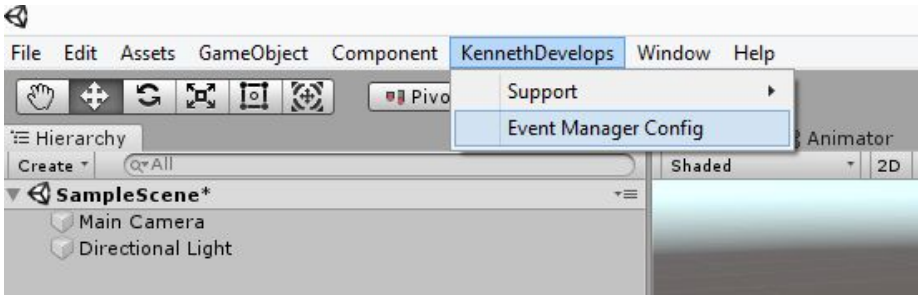
If you have encountered any issues with the product please fill this form:
https://goo.gl/forms/k2bxEeoUS5XkfZKo2

Please be kind enough to explain the issue as best as you can. Images, gifs and/or videos will also be appreciated if you think it could help understand the problem.

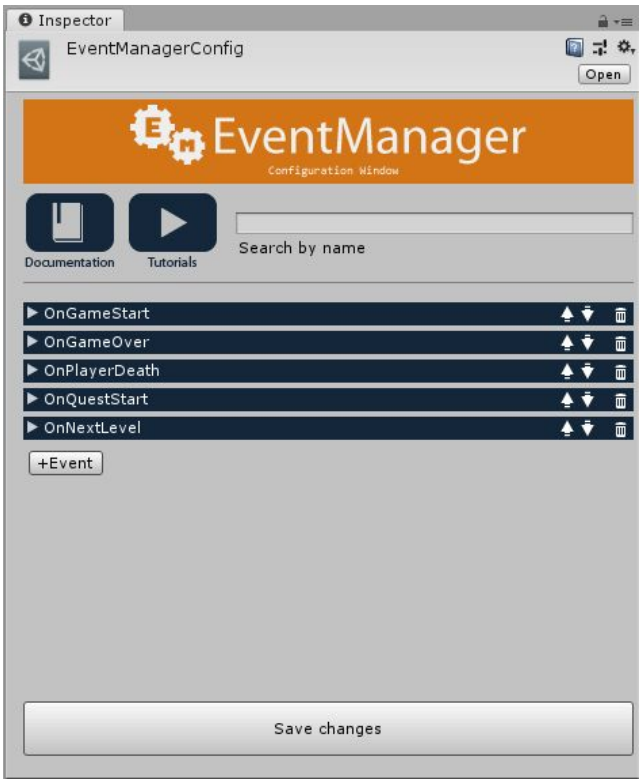
For any other comment on this wiki or the product itself, feedback or any other kind of suggestions you may have, you can fill this form:
https://goo.gl/forms/Bltw82mtyn6pDobf1

Configuration Window

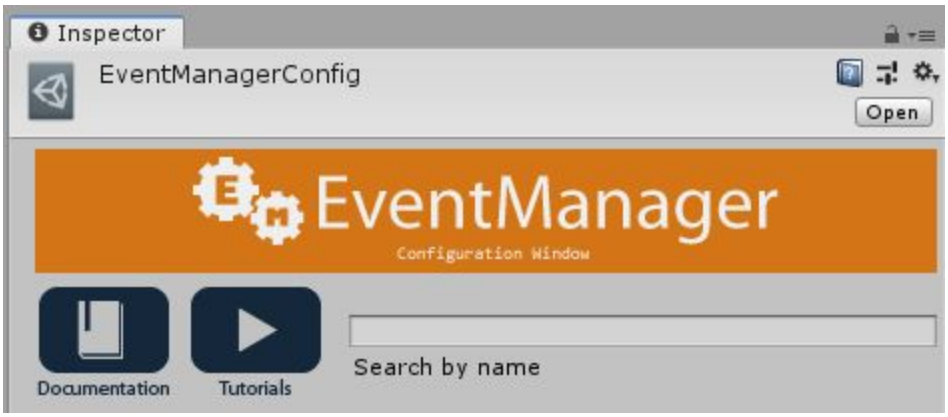
To start using the EventManager, first you have to configure your events. To do that, you can go to KennethDevelops->EventManager Config



And then the Inspector will show the configuration window. Here, you will immediately see all your configured events:

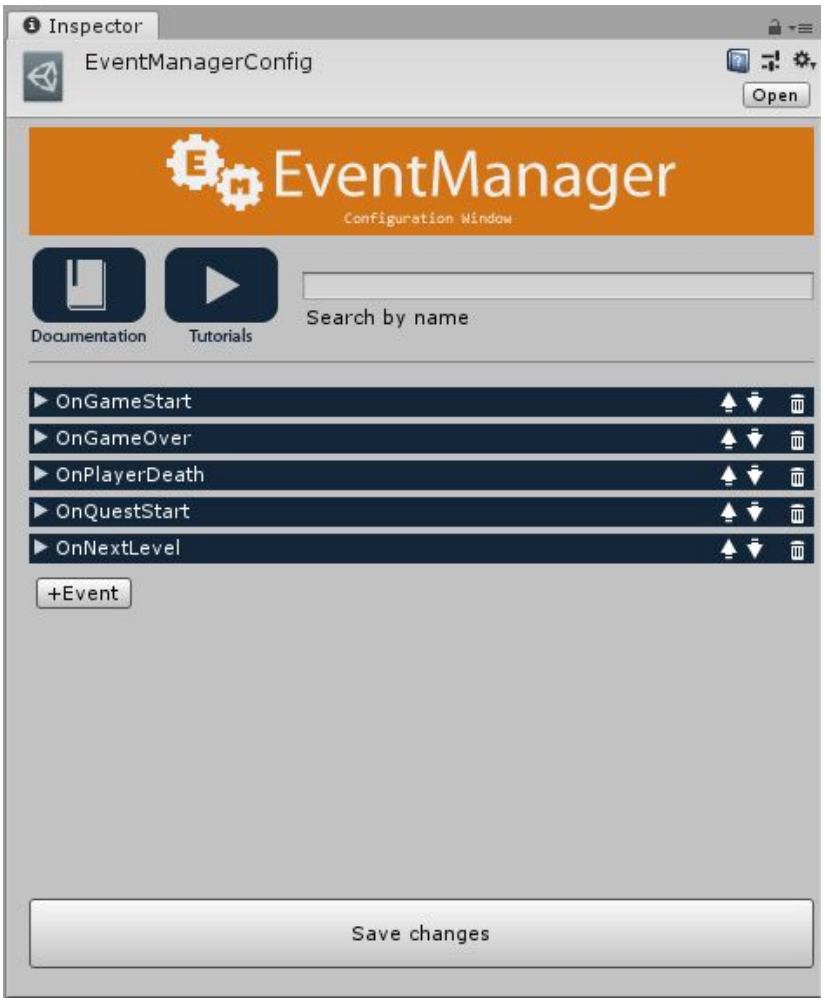


You can see that on the top you have a “Documentation” button, which will open your default Web Browser to show you the Online Documentation. And a “Tutorials” button that will open the Youtube Tutorials Playlist. And at the very right, you have a search bar that will search every Event that contains the string you type in that field.



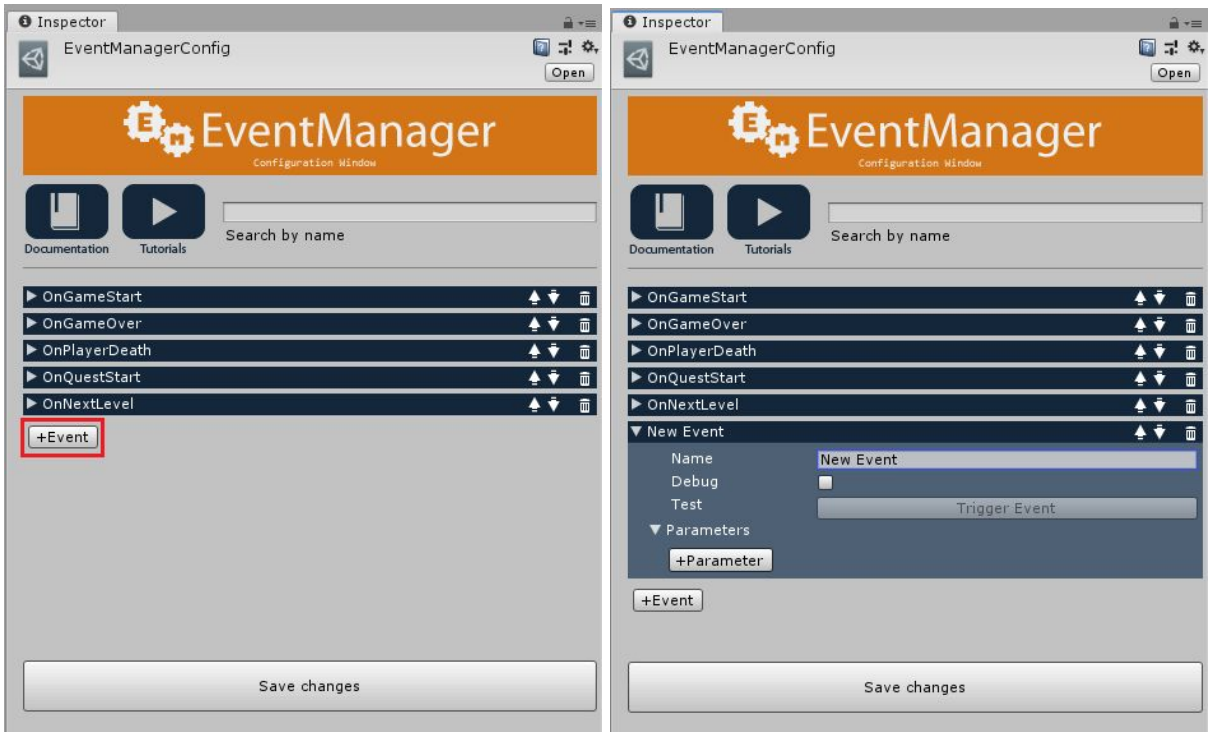
Events

An event is a message channel that you can subscribe to, to receive messages to be aware of an occurrence happening and receive vital information. You can trigger this events yourself and send information (parameters) of various types (bool, int, float, string, Vector2, Vector3, GameObject and Custom) for subscribers to use in the way they want.



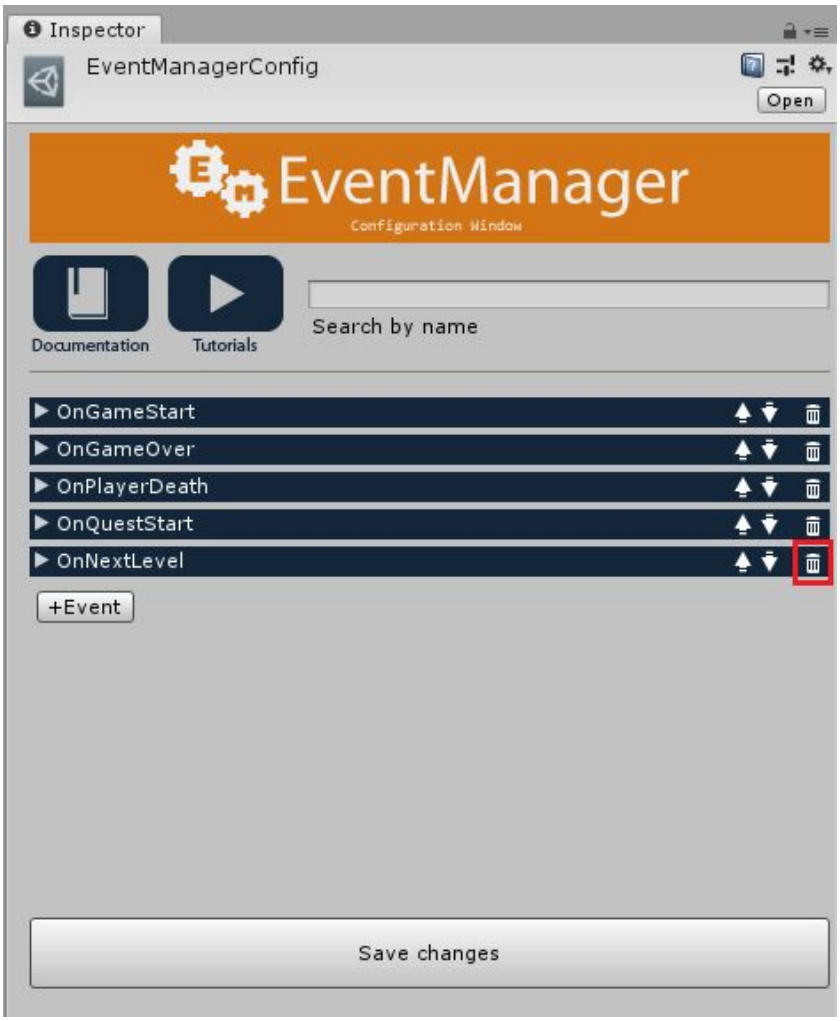
Adding Events

To add an event, simply open the configuration window and press the "+Event" button below the list of events.



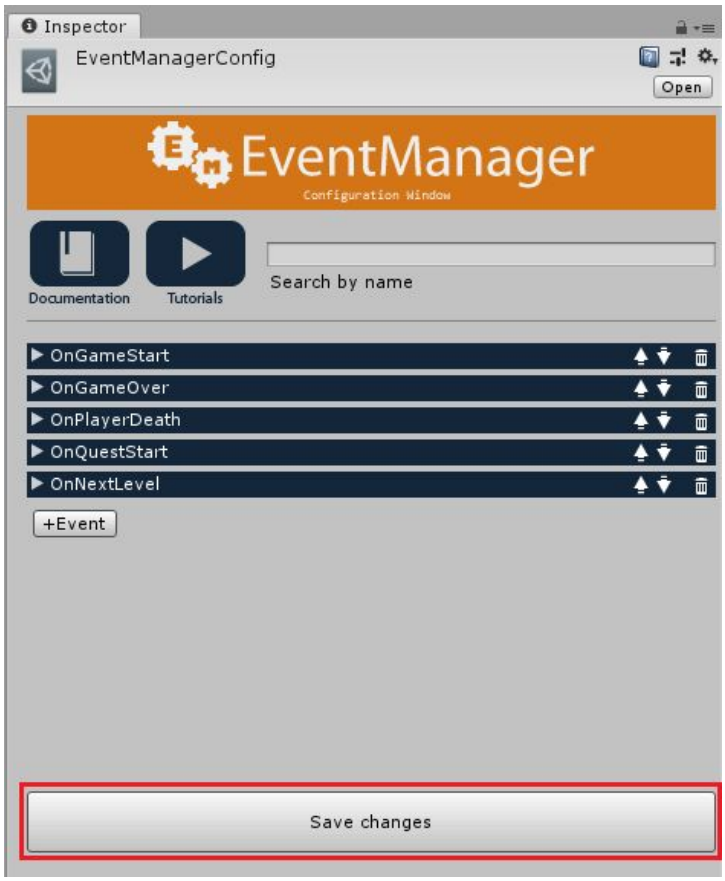
Removing Events

To remove an event, simply open the configuration window and press the trash icon button at the right side of each event name.



Saving Changes

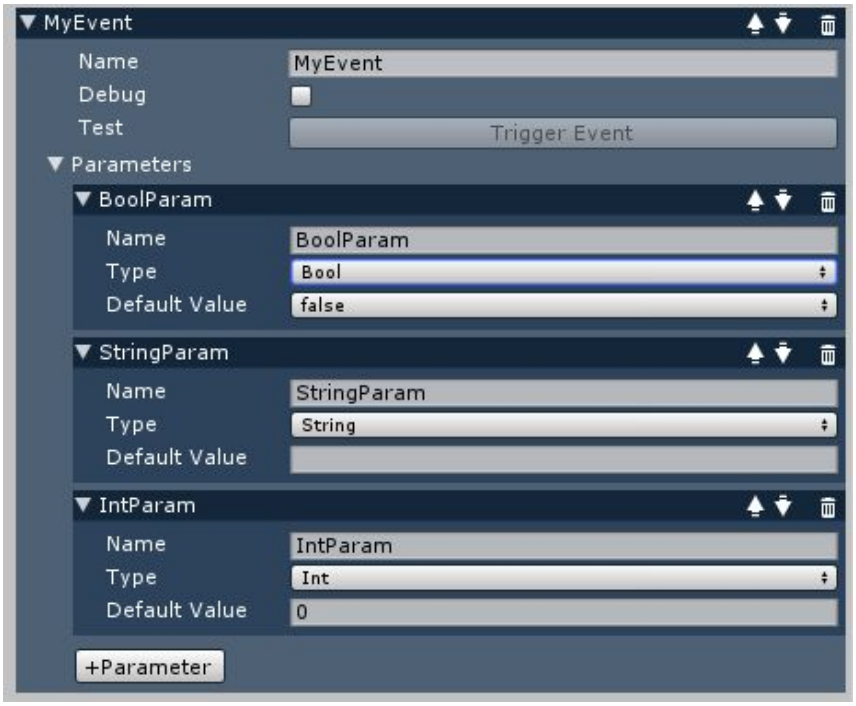
To save all changes, simply press the "Save changes" button at the bottom of the configuration window. Be sure to do this after every change. Some changes will seem to remain if you don't press this button, even during playmode, but after you restart Unity they will be gone unless you do.



Parameters

The event parameters are pieces of information that can be (optionally) sent when triggering an event.

Each parameter has a name to access it, a type to determine the value type that it carries and a default value that dictates the value that the Parameter will carry in case it is not given any value when the event is triggered.



Adding a Parameter is as simple as it is to add an event. Just unfold your event settings, and press the "+Parameter" button.

To remove a Parameter, you have to first unfold the Event, unfold the "Parameters" and press the “trash can” button at the right side of the Parameter's name you want to remove.

You can also reorder parameters with the arrow buttons.

Types

There are many types of parameters: *Boolean*, *Int*, *Float*, *String*, *Vector2*, *Vector3,* *GameObject* and *Custom*.

Game Object

For this type of parameter, the default value cannot be set. It will always be null as default.

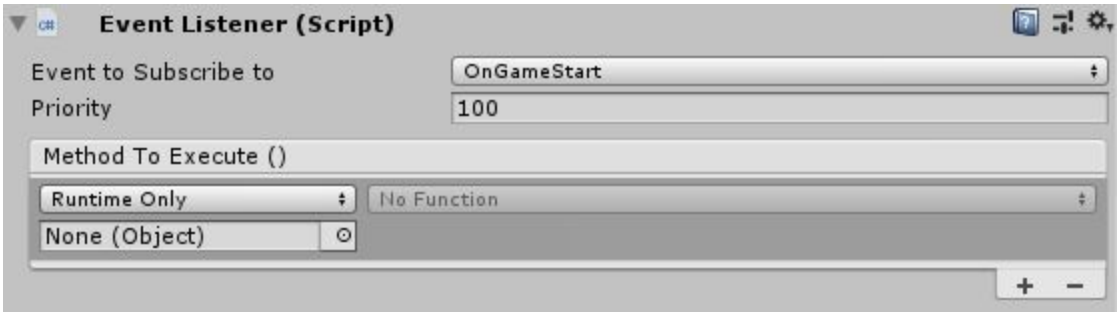
Custom

Custom parameters refer to every other possible type: Quaternions, Vector3, Vector2, or a class type you made on your own. For this to work, when you receive the parameter you have to cast it to whatever type it is. In the section "LINK" you will find examples on how to receive this parameter type.

For this type of parameter, the default value cannot be set. It will always be null as default.

Event Listener

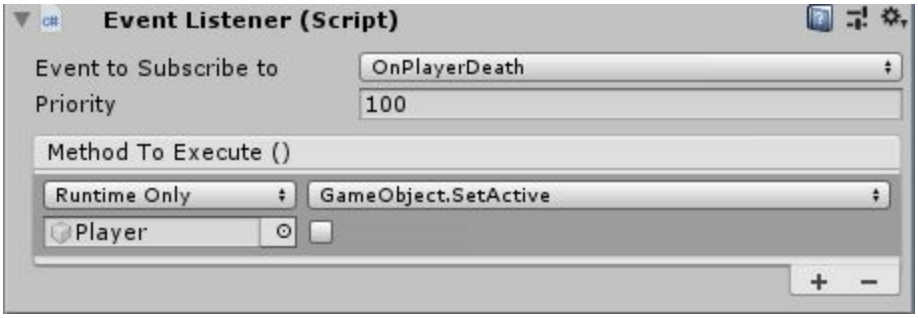
The Event Listener is a component that allows you to subscribe to an event and trigger one or more actions without the need to code.



To use it, we can simply select a GameObject, press the "Add component" button and add the "EventListener" component. Now that we are there, we can see that it asks us to configure the Event to which it will be subscribed to and the actions that it will be performing when the defined event gets triggered.

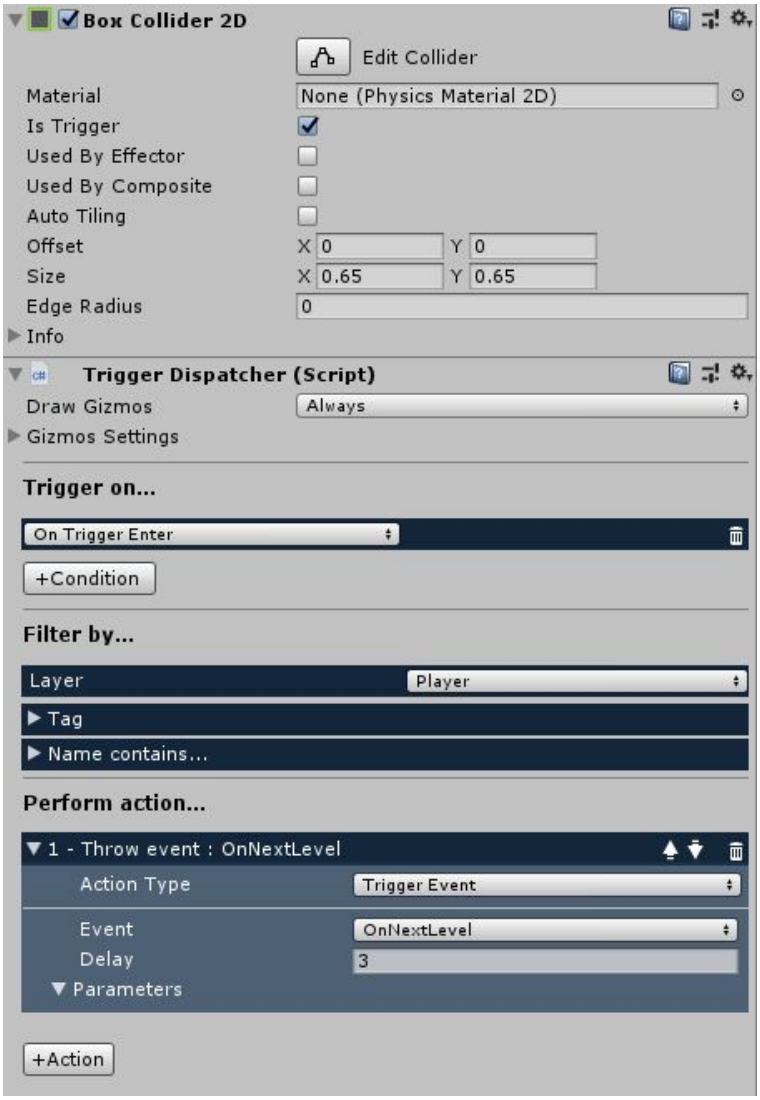
Also, you can set the **priority** of this listener, this way you can ensure that your configured actions will be executed before (or after) other methods subscribed to that same event. Lower priority values are executed first. Default value is always 100.

We can add an action and pass the object that has the method we want to execute (or the property we want to modify) and then we select said method or property.



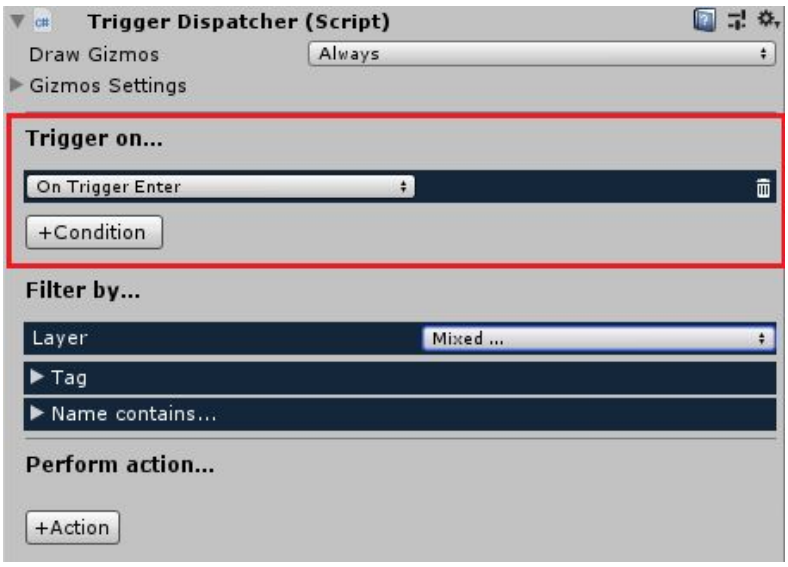
TriggerDispatcher

In a few words, TriggerDispatcher is a new feature that helps us easily perform certain actions when an object collides with it or enter its trigger area, without needing a single line of code.



Conditions

The conditions of the TriggerDispatcher are simply **when** the action will be trigger, will it be when it collided with an object? When an object entered its trigger area? When it exits it? Etc.



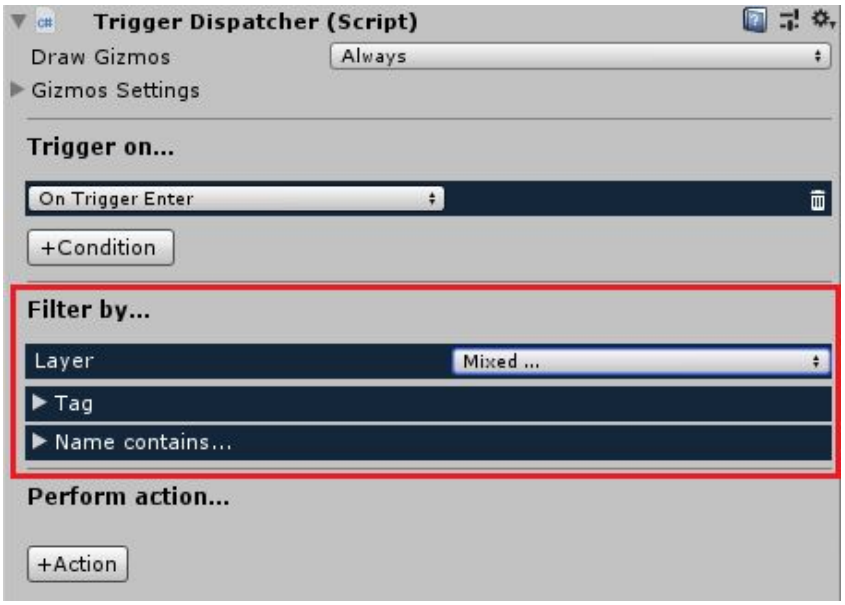
Available condition types are:

- OnTriggerEnter
- OnTriggerExit
- OnTriggerStay
- OnCollisionEnter
- OnCollisionExit
- OnCollisionStay

All of these conditions work for every type of Collider, 2D or 3D. As you can see, you can add as many conditions as you'd like with the "+Condition" button and also remove with the trash icon button at the far right.

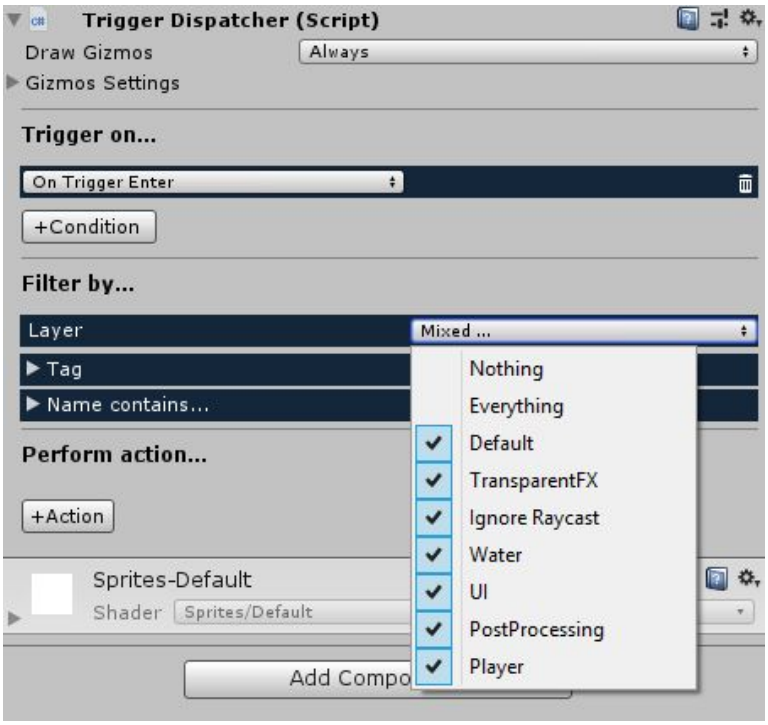
Filters

After the condition is triggered, TriggerDispatcher checks if the collider meets the requirements for the specified filters, either a layer, tag or name contains filter.



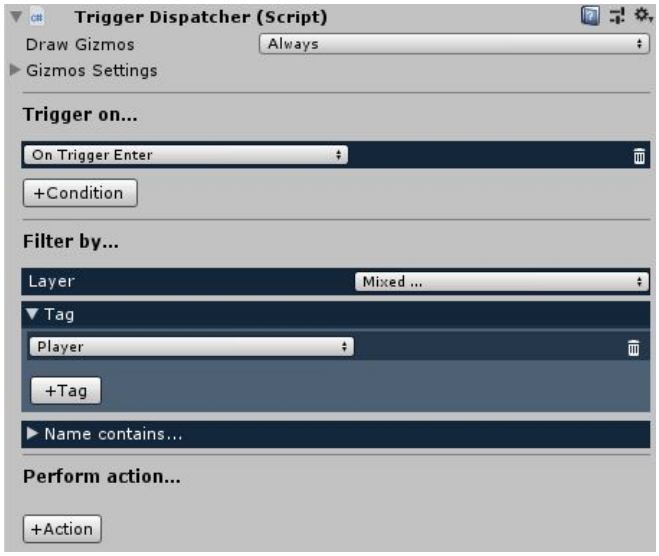
Layer Filter

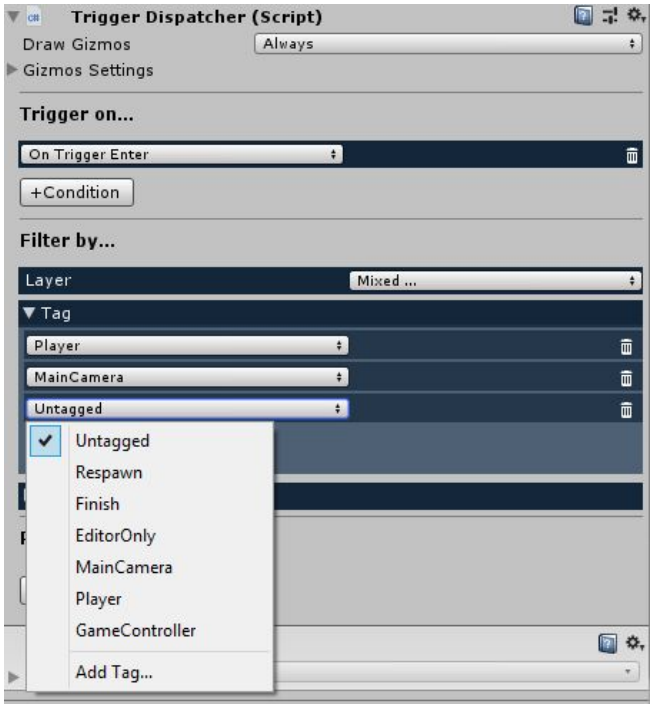
Provides a LayerMask to filter every collider that goes through the initially defined condition. You need to check every layer that you want to admit as valid. Default value is **Everything** which means that every layer is considered valid.



Tag Filter

You can filter colliders by tag too. The action will perform if the collider has any of the defined tags.You can add as many as you want by pressing the “+Tag” button and then removed then with the trash button on the far right.

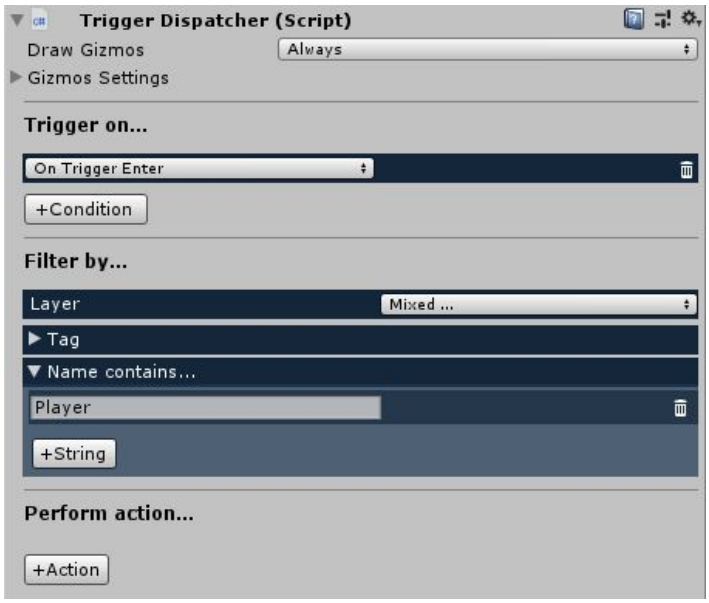




Name Filter

Finally, you can also filter colliders name. This filter will make the collision valid and perform the action if **any** of the defined strings are contained within the name of the collider's gameobject. This means that, if the collider's name is "Player Two" and your filter has the string "Player" defined, it will be valid because the name contains the defined string.

You can add as many as you want by pressing the "+String" button and then removed then with the trash button on the far right.



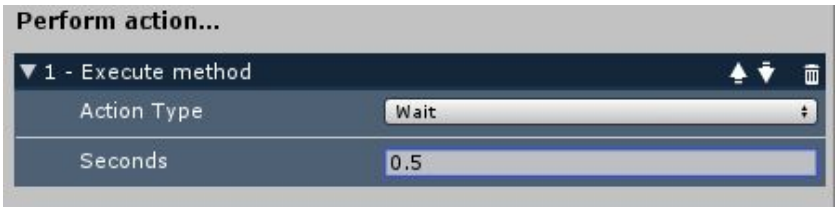
Actions

TriggerDispatcher will perform a series of actions defined by you when the condition is met and the filters are passed through. You can add an action with the “+Action” button and remove them with the trash button on the far right. You can also reorder them with the arrow buttons. Remember, execution order is important as it represents what actions will be performed first.

There are currently 3 types of actions: *Trigger Event*, *Execute Method* and *Wait*.

Wait

This is the simplest action of them all. This will get the TriggerDispatcher on hold for the specified time and continue executing the following actions after the wait.

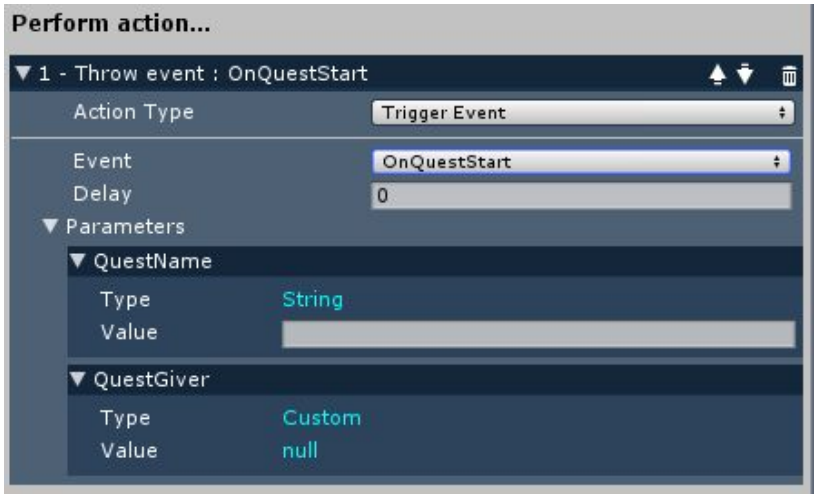


Trigger Event

As the name of the action says, this triggers an event (previously defined in Event Manager's Config Window).

You can then choose what Event to Trigger and the delay, meaning how much time (in seconds) to wait to trigger this event. Important Note: if an action is configured to trigger an event with delay, the following actions (if any) will still perform immediately instead of waiting for the event to trigger. If you want following actions to wait for the event to trigger, you should add a ***Wait*** action before and keep the delay value to 0.

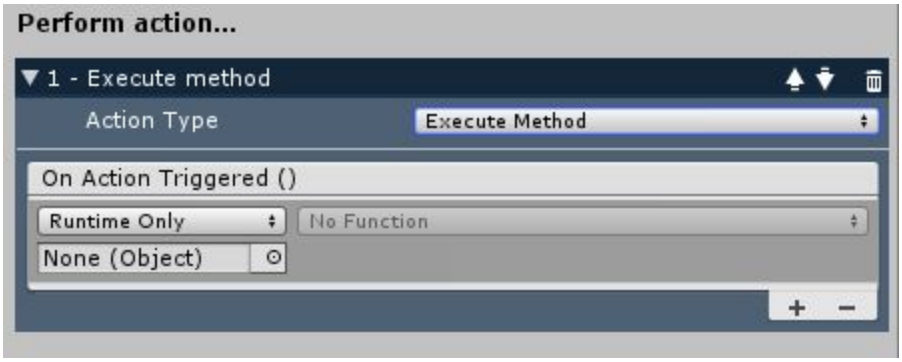
If the event you chose has parameters, you can configure what values you want to be sent:



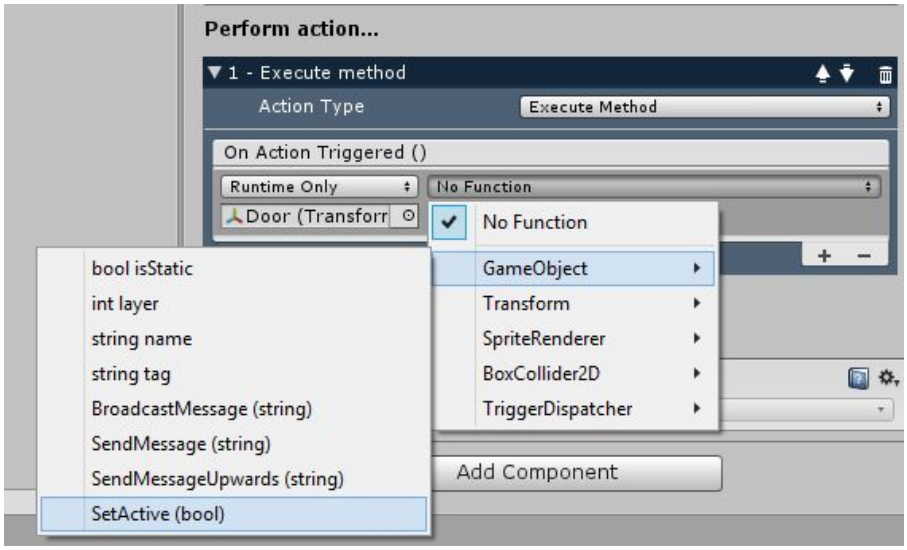
Please note that besides *Custom* parameters, every other parameter value can be set (bool, int, float, string, Vector2, Vector3 and GameObject)

Execute Method

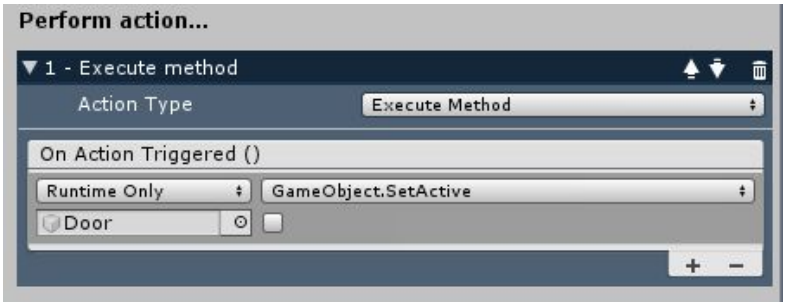
Similar to EventListener, this action will execute a series of custom methods defined by you.

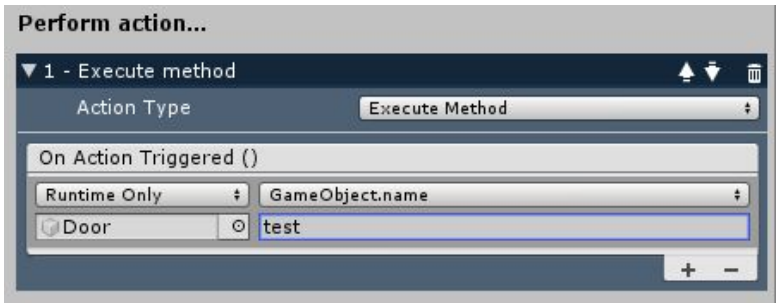


You can add an action and pass the object that has the method we want to execute (or the property we want to modify) and then we select said method or property.



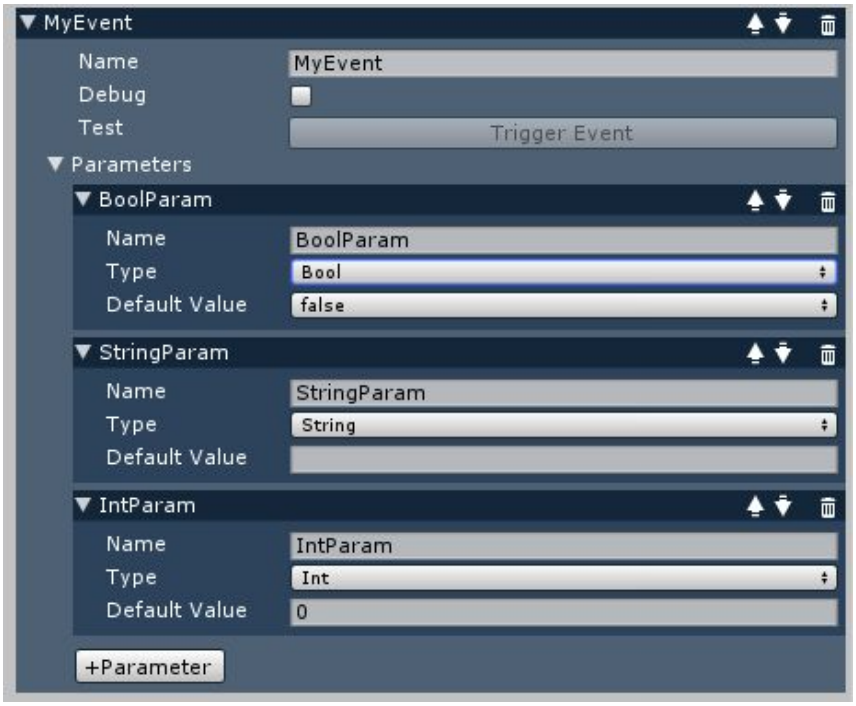
Finally, if the method requires a parameter to be sent as argument (or if you selected a property), you can set the value in the field below the method/property field.





Triggering a DefinedEvent

In this tutorial, we're going to trigger the event "MyEvent" defined here:



First, we need to import the namespace:

```
using KennethDevelops.Events;
```

And now, to trigger this event we simply type:

```
new DefinedEvent("MyEvent").Trigger();
```

As easy as that.

You can see that we're not sending any Parameters here, and that our event actually has 3. This means that every subscriber will receive all these Parameters with their **default value**.

To properly set the Parameters' values, we simply type:

```
new DefinedEvent("MyEvent").SetBoolParameter("BoolParam", true)
                           .SetIntParameter("IntParam", 15)
                           .SetStringParameter("StringParam", "test value")
                           .Trigger();
```

As we see above, we set the 3 Parameters of type Bool, Int and String, but we also have Float, GameObject and Custom types. To set those, we can use the following methods:

Float:

```
new DefinedEvent("MyEvent").SetFloatParameter("FloatParam", 4f)
                           .Trigger();
```

Vector2

```
new DefinedEvent("MyEvent").SetVector2Parameter("VectorParam", position2D)
                           .Trigger();
```

Vector3

```
new DefinedEvent("MyEvent").SetVector3Parameter("VectorParam", position)
                           .Trigger();
```

GameObject

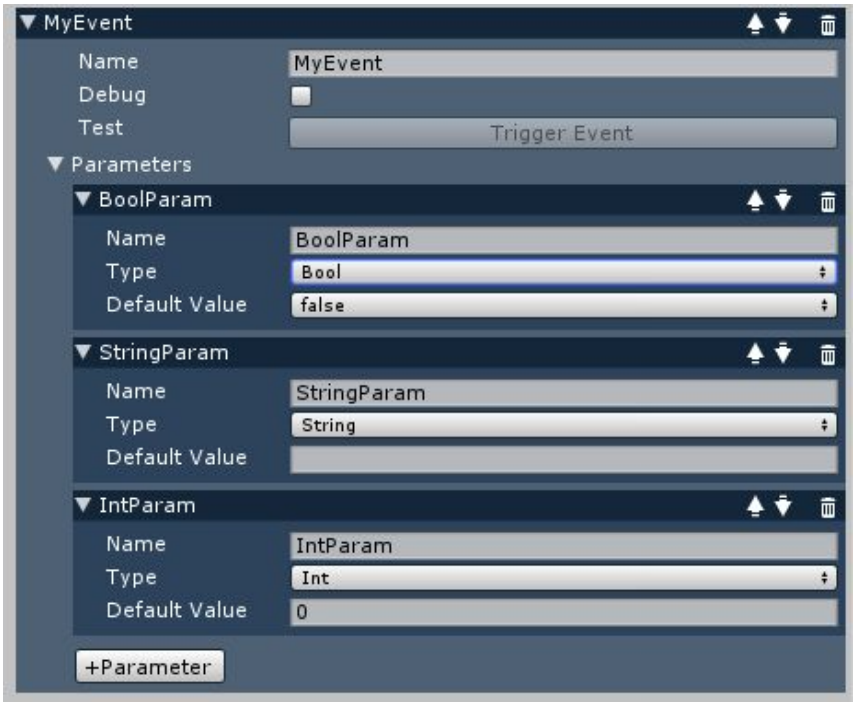
```
new DefinedEvent("MyEvent").SetGameObjectParameter("GameObjectParam", gameObject)
                           .Trigger();
```

Custom

```
new DefinedEvent("MyEvent").SetCustomParameter("CustomParam", new Quaternion())
                           .Trigger();
```

Subscribing to a DefinedEvent

In this tutorial, we're going to subscribe to the event "MyEvent" defined here:



First, we need to import the namespace:

```
using KennethDevelops.Events;
```

To subscribe to this event, we simply type:

```
void Start(){
    EventManager.SubscribeToEvent("MyEvent", OnMyEvent);
}

private void OnMyEvent(DefinedEvent definedEvent){
    //your code here
}
```

As we can see, we called the method SubscribeToEvent of the static class EventManager, passed the eventId and then the method that will be executed when our event is triggered.

Also, you can subscribe to an event with a **priority** value, this way you can ensure that your method will be executed before (or after) other methods subscribed to that same event. Lower priority values are executed first. Default value is always 100.

```
void Start(){
    EventManager.SubscribeToEvent("MyEvent", OnMyEvent, 100);
}
```

Parameters

To get the Parameters from the recently thrown event, you can type the following inside your OnMyEvent method, depending on the type:

Boolean

```
bool myParam = definedEvent.GetBoolParameter("myParameterName");
```

Int

```
int myParam = definedEvent.GetIntParameter("myParameterName");
```

Float

```
float myParam = definedEvent.GetFloatParameter("myParameterName");
```

String

```
string myParam = definedEvent.GetStringParameter("myParameterName");
```

Vector2

```
Vector2 myParam = definedEvent.GetVector2Parameter("myParameterName");
```

Vector3

```
Vector3 myParam = definedEvent.GetVector3Parameter("myParameterName");
```

GameObject

```
GameObject myParam = definedEvent.GetGameObjectParameter("myParameterName");
```

Custom

We have two ways of getting a Custom Parameter. The first, to get it as an *object*:

```
object myParam = definedEvent.GetCustomParameter("myParameterName");
```

Or to specify the type:

```
T myParam = definedEvent.GetCustomParameter<T>("myParameterName");
```

In the example above, T is the type of our Parameter. Let's see another example, this time with a *Quaternion* type:

```
Quaternion myParam = definedEvent.GetCustomParameter<Quaternion>("myParameterName");
```

Unsubscribe to a DefinedEvent

To unsubscribe to a DefinedEvent, you simply type:

```
EventManager.UnsubscribeToEvent("eventId", OnEvent);
```

The first string being the event's ID or event's name, and the second being the Callback (the method that you previously subscribed to this event).

For the example case we gave at the beginning of this page, it would be:

```
EventManager.UnsubscribeToEvent("MyEvent", OnMyEvent);
```