Messaging & Timers in Unity

SendMessage

Documentation link: SendMessage

Unity GameObjects can send each other messages directly using the **SendMessage** method.
SendMessage takes a name parameter for the method you wish to run on the GameObject.
SendMessage looks like this:

```
gameObject.SendMessage(“MyMethod” ,MyParameter);
```

- The first parameter is the method you wish to run
- The second parameter is an optional parameter for the method (you are only allowed one parameter!)

Example

Our Bullet hit a Zombie and we have an OnTriggerEnter2D to send it some damage

```
void OnTriggerEnter2D(Collider2D other)
{
    other.gameObject.SendMessage(“TakeDamage” 10);
}
```

We call the Zombie’s TakeDamage method and give it a value of 10

The Zombie must have a method called “TakeDamage” with an integer parameter on at least one of its scripts, else this will return an error!

UnityEvent

Documentation link: UnityEvent

Unity can send an event to multiple GameObjects at the same time using **UnityEvent**.
UnityEvent is run in code using its **Invoke** method
The Inspector allows you to setup each GameObject with their own methods to run (as long as the parameters match!)

NOTE: Before using UnityEvents, make sure you are using the **UnityEngine.Events** library:

```
using UnityEngine.Events;
```

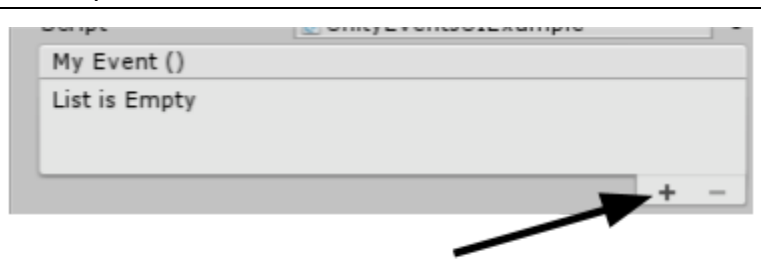
Declaring a **UnityEvent**

```
public UnityEvent MyEvent;
```

Using the **UnityEvent**

```
MyEvent.Invoke();
```

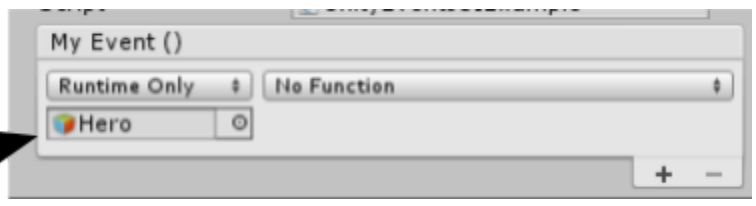
In the Inspector



Click here to add GameObjects to the event list

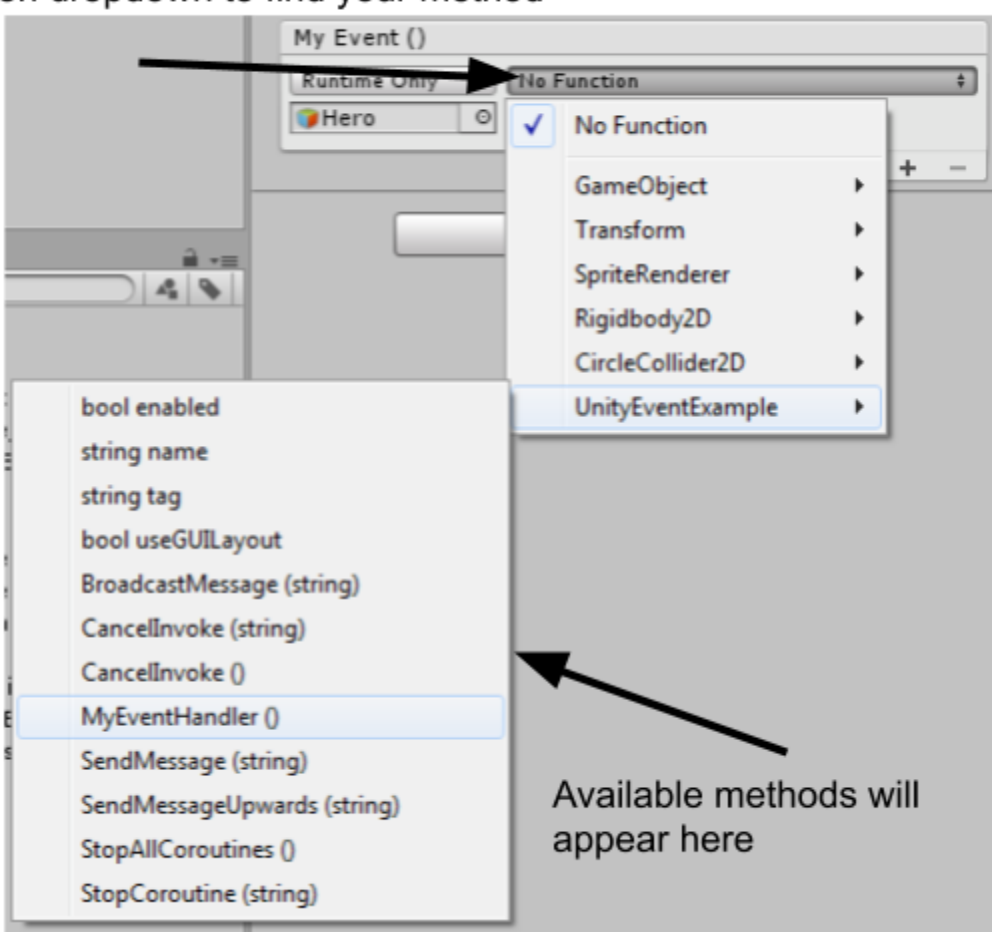
In the Inspector - setting up GameObjects

Drag the GameObject here

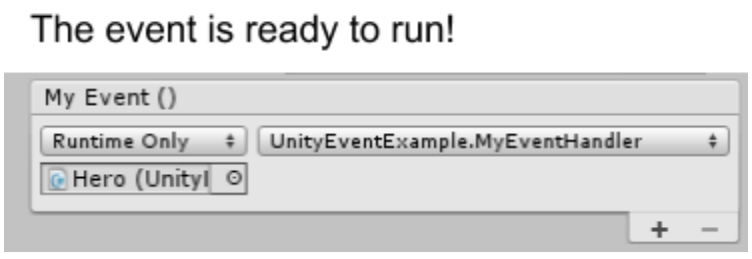


In the Inspector - adding a method

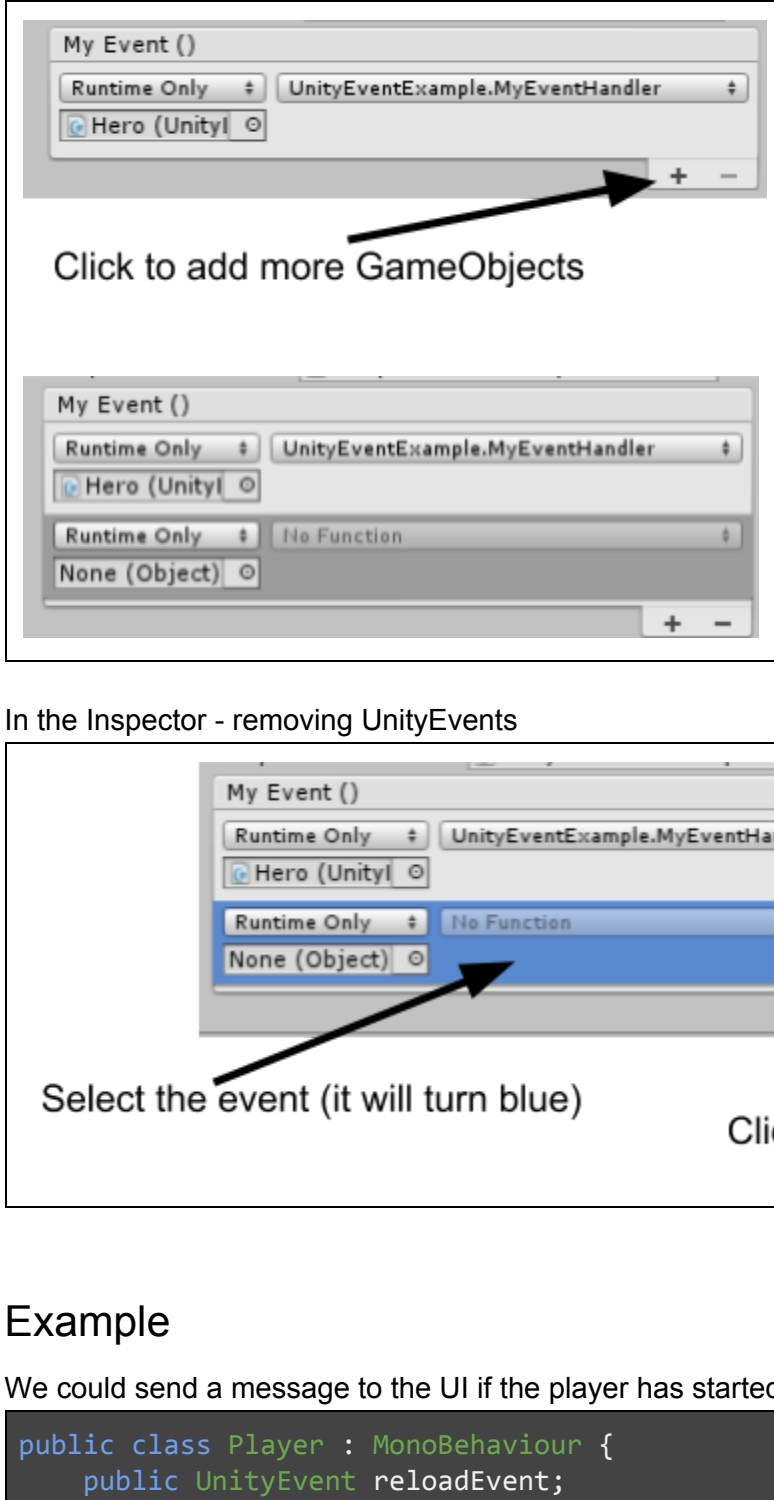
Use the function dropdown to find your method



In the Inspector - finished



In the Inspector - adding multiple UnityEvents



In the Inspector - removing UnityEvents

Example

We could send a message to the UI if the player has started reloading

```
public class Player : MonoBehaviour {
    public UnityEvent reloadEvent;

    void ReloadWeapon()
    {
        reloadEvent.Invoke();
    }
}
```

Custom UnityEvents

You can create custom **UnityEvents** with parameters (default is no parameters)
These are a custom class, often included at the top of the script using it
You can have up to 4 parameters on a custom **UnityEvent**

Documentation link: [Custom UnityEvent]

Here is a custom **UnityEvent** class with an integer as the custom parameter
The class definition is left empty

```
public class CustomUnityEvent:: UnityEvent<int>{}
```

Serialisation

If you wish to use the custom **UnityEvent** in the Inspector, place this line above the class declaration

```
[System.Serializable]
```

This will allow the custom class to be visible in the Inspector

Example using a Custom UnityEvent

Here is a Monobehaviour with a Custom UnityEvent class called DoDamageEvent

```
[System.Serializable]
public class DoDamageEvent: UnityEvent<int>{}

public class UnityEventExample : MonoBehaviour {
      public int startingHealth = 100;
      public DoDamageEvent doDamageEvent;

      void OnCollisionEnter2D(Collision2D other)
      {
            startingHealth -= 10;
            doDamageEvent.Invoke (startingHealth);
      }
}
```

Note we sure the OnCollisionEnter2D to actually invoke the custom UnityEvent

Delegates and events

Documentation link: Delegates
Documentation link: Events

Delegates

Delegates and events work with each other to create a “broadcast” system that other GameObjects can listen to.
An advantage of using this system is that the GameObjects using it don’t have to be created first (like if you spawn zombies or bullets)

Delegates can be used to point to a method, so you can rename a method:
While this may not seem useful, you can also add more methods to it, meaning one **delegate** can call many methods

Creating a delegate looks like this:

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

You need to create a variable of the delegate to use it

```
MyDelegate zombieDiedDelegate;
```

Here is a method we can to assign to our delegate

```
void Died() {

}
```

To assign the method, use the “+=” operator

```
zombieDiedDelegate += Died;
```

NOTE: we don’t use the brackets, parameters or return type of the method here, just the method name

We can call the delegate like this:

```
zombieDiedDelegate();
```

Which is the same as:

```
Died();
```

Not very useful right now, but there’s more!

We can add another method called ResetHealth() like this:

```
zombieDiedDelegate += ResetHealth;
```

Now, calling the delegate:

```
zombieDiedDelegate();
```

Is the same as this:

```
Died();
ResetHealth();
```

We can add as many methods as we like

Events

Events allow us to use delegates in other classes.
Combined with delegates, **events** allow us to make a one-way messaging system.
Also, because of the way delegates work, we don’t even need to know what other classes we are sending messages to!
Events also need to be static, as we use them between classes, not objects.

If we have a delegate like this:

```
public delegate void MyDelegate;
```

We create an event from that delegate like this:

```
public static event MyDelegate MyEvent;
```

NOTE: we use MyDelegate as the type of event

Broadcasting messages

Other classes can “subscribe” to our event.
When they do this, the event has a list of these subscribers it will send to when called.
If there are no classes subscribing to the event, it will be null.
If we use an event and it is null, it will cause an error!
So make sure to check the event is not null before using it

Use an event like this:

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

This is broadcast to event all subscribing classes

Subscribing to an event

A class that listens for an event needs to assign that event to a method
In Unity, we often assign these events in the OnEnable method
We also have to de-assign the event if the listening GameObject is destroyed

Assigning an event is like this: (inside an OnEnable method)

```
DelegateClass.MyEvent += CustomEvent;
```

Because the event is static we can call it from the DelegateClass directly

De-assigning an event is similar, but using the “-=” operator (should be inside an OnDisable method)

```
DelegateClass.MyEvent -= CustomEvent;
```

Example

We have a class called DelegateClass, with a delegate and event setup for broadcast like this:

```
public class DelegateClass: MonoBehaviour {
    public delegate void MyDelegate;
    public static event MyDelegate MyEvent;

    void Start() {
        if (MyEvent != null) {
            MyEvent();
        }
    }
}
```

We have a ReceiverClass that listens for the event like this:

```
public class ReceiverClass : MonoBehaviour {

    void OnEnable() {
        DelegateClass.MyEvent += CustomEvent;
    }

    void OnDisable() {
        DelegateClass.MyEvent -= CustomEvent;
    }

    void CustomEvent()
    {

    }
}
```

The CustomEvent method will run when it receives the MyEvent event from DelegateClass

Example in a game

The Zombie needs to tell the Spawner it has died.
The Spawner can then spawn another zombie

The Zombie class will broadcast when it dies
The Spawner needs a custom method to run when it receives the event

We use the OnDestroy method to send our event

```
public class Zombie: MonoBehaviour {
    public delegate void ZombieDied;
    public static event ZombieDied OnZombieDied;

    void OnDestroy() {
        if (OnZombieDied != null) {
            OnZombieDied();
        }
    }
}
```

Our Spawner will listen for the ZombieDied event on our Zombie class

```
public class ReceiverClass : MonoBehaviour {

    void OnEnable() {
        Zombie.ZombieDied += SpawnZombie;
    }

    void OnDisable() {
        Zombie.ZombieDied -= SpawnZombie;
    }

    void SpawnZombie()
    {

    }
}
```

Invoke

Documentation link: [Invoke]

Runs a method after a specified period of time

In code, it looks like this:

```
Invoke(“MyMethod” ,timeToWait);
```

The method you call cannot have any parameters or return type.

This is the type of method you can call from Invoke:

```
void MyMethod()
{

}
```

No parameters or return type

You can check if the Invoked method is currently running using:

```
IsInvoking(“MyMethod”);
```

This will return true if the Invoked method is currently running

Example

You can use this check in other parts of the code, like in the Update method:

```
void Update()
{
    if(IsInvoking(“ShootBullet”) == false)
    {
        Invoke(“ShootBullet”, 1);
    }
}
```

The code will check if ShootBullet is being Invoked, if not, Invoke it

You can stop the Invoke early like this:

```
CancelInvoke(“MyMethod”);
```

Coroutine

Documentation link: Coroutine

Coroutines run a sequence of timed code, like several Invokes in one method.

In code, it looks like this:

```
IEnumerator MyCoroutine() {

}
```

Note it requires a return type of IEnumerator, this is where the sequence of timed code comes in.
We can put a timer in the return type and carry on after that time has expired.

The return type code looks like this:

```
yield return new WaifForSeconds( timeToWait );
```

You can include as many of these as you want to create the sequence

Like this:

```
IEnumerator MyCoroutine() {
    // start sequence
    yield return new WaifForSeconds( timeToWait );

    // next part of sequence
    yield return new WaifForSeconds( timeToWait );

    // and next part
    yield return new WaifForSeconds( timeToWait );

    // and so on
    yield return new WaifForSeconds( timeToWait );
}
```

Example

Change colour on a Sprite from red to green to blue with a 1 second gap between each

```
IEnumerator ColourSequence() {
    GetComponent<SpriteRenderer> ().color = Color.red;
    yield return new WaifForSeconds(1);

    GetComponent<SpriteRenderer> ().color = Color.green;
    yield return new WaifForSeconds(1);

    GetComponent<SpriteRenderer> ().color = Color.blue;
    yield return new WaifForSeconds(1);
}
```