
互動程式設計III
Interactive Programming Design Integration

Introduction to Message System

- In this lesson you'll learn:
  - A comprehensive overview of various inter-object message solutions in Unity C#.
  - How to choose and implement message channel between 2 Scripts.
- Learning Objectives
  - Gain a solid understanding of messaging systems and their components.
  - Learn how to implement delegates, events, and UnityEvent in real-world scenarios.
  - Master the concepts of event-driven programming in C# and Unity.
  - Understand the observer design pattern.
- Why This Chapter is Important
  - Decoupling Components:
    - Messaging systems help decouple components, promoting modularity and maintainability.
  - Enhancing Flexibility:
    - Delegates and events provide flexibility in how components interact, enabling dynamic behavior.
  - Real-Time Communication:
    - Essential for implementing real-time communication and interactions in software and games.
  - Unity Development:
    - UnityEvent is crucial for Unity game development, allowing for responsive and interactive gameplay.


互動程式設計III
Interactive Programming Design Integration
• Message System
• Method Calling
• Message
• Delegates
• Events
• UnityEvent

What is Direct Method Calling

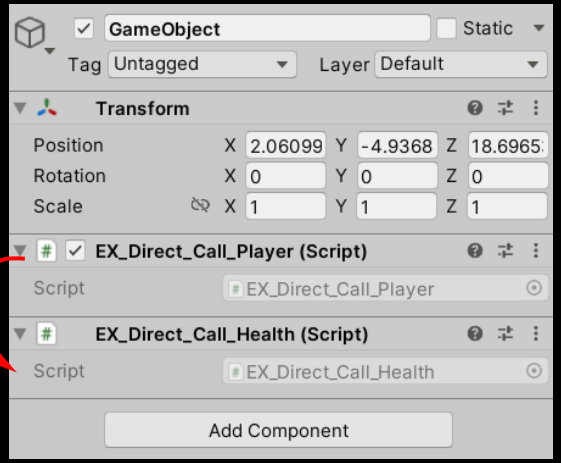
- Definition and Usage
  - Refers to obtaining a reference to an instance of a component or object and invoking its methods through reference directly.
  - Provides better performance and type safety compared to dynamic methods like SendMessage.
- Benefits and Considerations
  - Benefits:
    - Performance: Direct method calls are faster than SendMessage due to the lack of runtime lookups.
    - Type Safety: The compiler checks method calls, reducing runtime errors.
    - Clarity: Code is more readable and easier to understand.
  - Considerations:
    - Coupling: Direct references can increase coupling between components.
    - Initialization: Ensure the component reference is valid (not null) before calling methods.
    - Lifecycle: Be aware of the Unity lifecycle to avoid calling methods on destroyed or uninitialized objects (null reference).

EX: Direct Method Calling

- Steps to Call a Method Directly
  - Get a Reference to the Component:
    - Use methods like GetComponent<T>(), GetComponentsInChildren<T>(), FindObjectOfType<T>(), or FindObjectsOfType<T>() to obtain a reference.
  - Invoke the Method:
    - Call the method directly on the obtained reference.

```
using UnityEngine;

public class EX_Direct_Call_Player : MonoBehaviour
{
    void Start()
    {
        // Get reference to the Health component in Sibling layer
        EX_Direct_Call_Health health = GetComponent<EX_Direct_Call_Health>();

        // Find reference to the Health component in the whole scene.
        //EX_Direct_Call_Health health = FindObjectOfType<EX_Direct_Call_Health>();

        // Call the ApplyDamage method
        if (health != null)
        {
            health.ApplyDamage(10);
        }
    }
}
```

GetComponent<T>()




```
using UnityEngine;

public class EX_Direct_Call_Health : MonoBehaviour
{
    public void ApplyDamage(int damage)
    {
        // Apply damage to the health
        Debug.Log($"Damage applied: {damage}");
    }
}
```


互動程式設計III
Interactive Programming Design Integration
• Message System
• Method Calling
• Message
• Delegates
• Events
• UnityEvent

What is Message

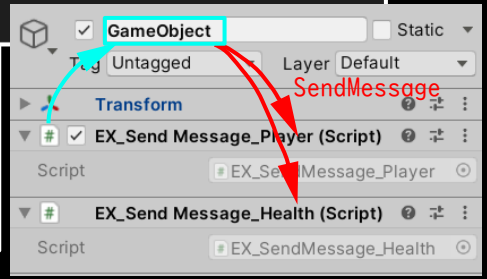
- Definition and Usage
  - **Sending a message to a GameObject** means calling a method by name on all components attached to the GameObject.
  - Allows for dynamic and decoupled communication between GameObjects and their components.
- Benefits and Considerations
  - Benefits:
    - Decoupling: Allows components to communicate without tight coupling. You just need to know the method name and get the reference to the GameObject.
    - Flexibility: Methods can be dynamically switched at runtime. You just need to change the target method name.
  - Considerations:
    - Performance: SendMessage() can be slower than direct method calls due to its dynamic nature.
    - Error Handling: Ensure proper handling with SendMessageOptions to avoid runtime errors.
    - Maintenance: Method names are passed as strings, which can lead to errors if names change or having typo.

EX: Send Message to a GameObject

- Syntax:
  - GameObject.SendMessage(string "methodName", object value, SendMessageOptions option);
    - Parameters:
      - string methodName: The name of the method to call.
      - object value (optional): An optional parameter to pass to the method.
      - SendMessageOptions option (optional): Specifies what to do if the method doesn't exist. Options include:
        - SendMessageOptions.RequireReceiver: Throws an error if no receiver is found.
        - SendMessageOptions.DontRequireReceiver: Does nothing if no receiver is found.
    - The value will be boxed to object type at sender's side. And being unboxed later at receiver's side.

```
1   using UnityEngine;
2
3   public class EX_SendMessage_Player : MonoBehaviour
4   {
5       private void Start()
6       {
7           // Sending a message to call a method named "ApplyDamage"
8           gameObject.SendMessage("ApplyDamage", (object) 10);
9
10          // Sending a message and ignoring if no method is found
11          //gameObject.SendMessage("ApplyDamage", (object) 10, SendMessageOptions.DontRequireReceiver);
12      }
13  }
```

```
1   using UnityEngine;
2
3   public class EX_SendMessage_Health : MonoBehaviour
4   {
5       public void ApplyDamage(int damage)
6       {
7           // Apply damage to the health
8           Debug.Log($"Damage applied: {damage}");
9       }
10  }
```




互動程式設計III
Interactive Programming Design Integration
• Message System
• Method Calling
• Message
• Delegates
• Events
• UnityEvent

What is Delegate

- Definition and Use Cases
  - Definition: A delegate is a type that represents references to methods with a specific **method signature** and return type (signature). Delegates are similar to function pointers in C++ but are type-safe and secure.
    - <From wikipedia>...method signatures in C# are composed of a method name and the argument number and argument types, where the last parameter may be an array of values.
    - The return type is not part of the signature
      - `int Add(int a, int b, params int[] values);  // return (a + b + values)`
- Use Cases:
  - Event Handling: Delegates are commonly used to define event handlers.
  - Callback Methods: Delegates allow methods to be passed as parameters, enabling callback mechanisms.
  - Multithreading: Delegates can be used to execute methods asynchronously.
  - Encapsulation of Method Calls: Delegates can encapsulate (or hide) a method call, making them useful in scenarios where the method to be invoked is not known until runtime.

EX: Use of Delegate

- Define a Signature for a Delegate
  - A delegate is a type that represents references to methods with a specific signature.
  - It can be declared using the delegate keyword followed by the method signature.
- Instantiate a New Delegate
  - Create an instance of the delegate and assign it to a method that matches the delegate's signature.
  - This binds the delegate to the method.
- Invoke the Delegate
  - Once the delegate is instantiated, it can be invoked like a method.
  - When invoked, it calls the method(s) it is bound to.

```
1  using UnityEngine;
2
3  public class EX_Delegate_01 : MonoBehaviour
4  {
5      // 1. Define a delegate
6      public delegate void MyDelegate(string message, int hour, int min, int sec);
7
8      void Start()
9      {
10         // 2. Instantiate the delegate
11         MyDelegate myDele = new MyDelegate(ShowTime);
12
13         // 3. Invoke the delegate
14         myDele("Hello, World!", 7, 5, 55);
15     }
16
17     // Method that matches the delegate signature
18     public void ShowTime(string message, int hour, int min, int sec)
19     {
20         Debug.Log(message);
21         Debug.Log($"The time now is: {hour}:{min}:{sec}");
22     }
23 }
```

EX: Re-assign of Delegate

-

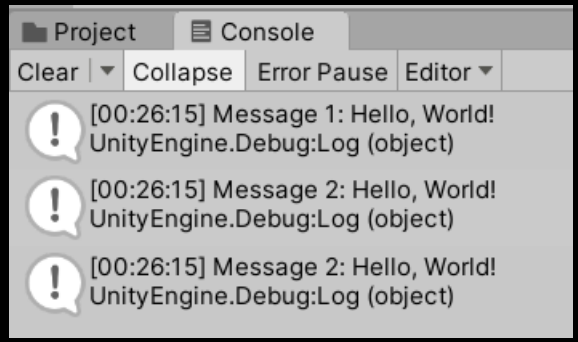
```
using UnityEngine;

public class EX_Delegate_02 : MonoBehaviour
{
    public delegate void MyDelegate(string message, int hour, int min, int sec);

    void Start()
    {
        // Assign myDele to ShowTime, and then invoke it
        MyDelegate myDele = new MyDelegate(ShowTime);
        myDele("Hello, World!", 7, 5, 55);

        // Re-assign myDele to ShowAlarmSettings, and then invoke it
        myDele = new MyDelegate(ShowAlarmSettings);
        myDele("Wake up in the morning", 7, 5, 55);
    }

    // 1st Method that matches the delegate signature
    public void ShowTime(string message, int hour, int min, int sec)
    {
        Debug.Log(message);
        Debug.Log($"The time now is: {hour}:{min}:{sec}");
    }

    // 2nd Method that matches the delegate signature
    public void ShowAlarmSettings(string lable, int hour, int min, int sec)
    {
        Debug.Log($"Alarm: <{lable}> will be ring on {hour:D2}:{min:D2}:{sec:D2}");
    }
}
```

EX: Multicast of Delegate

- A delegate can reference multiple methods, which are invoked in sequence.

```
1  using UnityEngine;
2
3  public class EX_Delegate_03 : MonoBehaviour
4  {
5      public delegate void MyDelegate(string message);
6
7      void Start()
8      {
9          // Create a delegate instance that references ShowMessage1
10         MyDelegate del = ShowMessage1;
11
12         // Add ShowMessage2 to the delegate invocation list
13         del += ShowMessage2;
14
15         // Invoke the delegate
16         del("Hello, World!");
17
18         // Remove ShowMessage1 from the delegate invocation list
19         del -= ShowMessage1;
20
21         // Invoke the delegate
22         del("Hello, World!");
23     }
24
```

```
24
25     public void ShowMessage1(string message)
26     {
27         Debug.Log("Message 1: " + message);
28     }
29
30     public void ShowMessage2(string message)
31     {
32         Debug.Log("Message 2: " + message);
33     }
34 }
35
```




互動程式設計III
Interactive Programming Design Integration
• Message System
• Method Calling
• Message
• Delegates
• Events
• UnityEvent

What is Event

- Definition:
  - An event in C# is a way for a class or object to provide **notifications** to other classes or objects when something of interest occurs. Events are based on the delegate model and provide a mechanism to broadcast a message to multiple subscribers.
  - Events enable a class or object to notify other classes or objects when something of interest happens, adhering to the publisher-subscriber pattern.
- Usage:
  - Event Handling: Events are commonly used to handle user actions such as button clicks, key presses, and other user interface interactions.
  - Decoupling: Events provide a way to decouple the sender of an event (the publisher) from the receivers (the subscribers), promoting a modular and maintainable design.
  - Asynchronous Programming: Events can be used to signal the completion of asynchronous operations.
  - Real-time Updates: Events are useful for implementing real-time updates and notifications within an application, such as updating the UI when data changes.
- Common Use Cases:
  - UI Events: Handling button clicks, form submissions, and other user interactions in GUI applications.
  - Custom Notifications: Creating custom events for various application-specific notifications, such as data updates, state changes, and error reporting.
  - Asynchronous Task Completion: Notifying when asynchronous tasks, such as file downloads or network requests, are completed.

EX: Defining an Event

- Create an event center and defining an event ❶:
  - Declare a delegate that specifies the signature for the event handler methods.
  - Declare an event based on the delegate.
- Implement the required event operation methods:
  - Subscribe/Unsubscribe: these are for the subscriber.
    - Please do remember to implement unsubscribe. Otherwise you'll have memory leakage when subscriber object is destroyed.
  - Invoke: this is for the publisher.
    - Use the ?. operator to ensure that there are subscribers before raising the event.

```
1   using UnityEngine;
2
3   public class EX_Event_Center : MonoBehaviour
4   {
5       // Delegate declaration
6       public delegate void MyEventHandler();
7
8       // Event declaration
9       private event MyEventHandler MyEvent;
10
11      // Invoke the Event
12      public void Invoke()
13      {
14          MyEvent?.Invoke();
15      }
16
17      // Hook the handler
18      public void Subscribe(MyEventHandler handler)
19      {
20          MyEvent += handler;
21      }
22
23      // Un-hook the handler
24      public void Unsubscribe(MyEventHandler handler)
25      {
26          MyEvent -= handler;
27      }
28  }
```

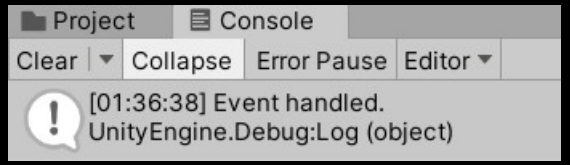
EX: Subscribing an Event

- Subscribing/Unsubscribing an Event(action binding):❶
  - Call subscribe when enabled(initialized).
  - Call unsubscribe when disabled.
    - Please do remember to call unsubscribe in OnDisable. Otherwise you'll have memory leakage when subscriber object is destroyed.
- Implement the handler body.❷

```
1  using UnityEngine;
2
3  public class EX_Event_Subscriber : MonoBehaviour
4  {
5      public EX_Event_Center eventCenter;
6
7      ❶ private void OnEnable()
8      {
9          eventCenter.Subscribe(HandleEvent);
10     }
11
12     private void OnDisable()
13     {
14         eventCenter.Unsubscribe(HandleEvent);
15     }
16
17     ❷ public void HandleEvent()
18     {
19         Debug.Log("Event handled.");
20     }
21 }
```

EX: Raising(Invoke) an Event

- Create the publisher class
- Raising an Event under proper condition:
  - Define the condition that raises(invoke) the event.




```
1  using UnityEngine;
2
3  public class EX_Event_Publisher : MonoBehaviour
4  {
5      public EX_Event_Center eventCenter;
6
7      private void Update()
8      {
9          if (Input.GetKeyDown("a"))
10         {
11             eventCenter.Invoke();
12         }
13     }
14 }
```

EX: Defining an Event(2)

- A Common and classic signature for an C# event:
  - sender(object): the reference to the sender object.
  - e (EventArgs): the event arguments to describe the event.
- The rest parts are the same.
  - Create an event center and defining an event:
    - Declare a delegate that specifies the signature for the event handler methods.
    - Declare an event based on the delegate.
  - Implement the required event operation methods:
    - Subscribe/Unsubscribe: these are for the subscriber.
      - Please do remember to implement unsubscribe. Otherwise you'll have memory leakage when subscriber object is destroyed.
    - Invoke: this is for the publisher.
      - Use the ?. operator to ensure that there are subscribers before raising the event.

```
1  using System;
2  using UnityEngine;
3
4  public class EX_Event_Center : MonoBehaviour
5  {
6      // Delegate declaration
7      public delegate void MyEventHandler(object sender, EventArgs e);
8
9      // Event declaration
10     private event MyEventHandler MyEvent;
11
12     // Invoke the Event
13     public void Invoke(object sender, EventArgs e)
14     {
15         MyEvent?.Invoke(sender, e);
16     }
17
18     // Hook the handler
19     public void Subscribe(MyEventHandler handler)
20     {
21         MyEvent += handler;
22     }
23
24     // Un-hook the handler
25     public void Unsubscribe(MyEventHandler handler)
26     {
27         MyEvent -= handler;
28     }
29 }
```

EX: Subscribing an Event(2)

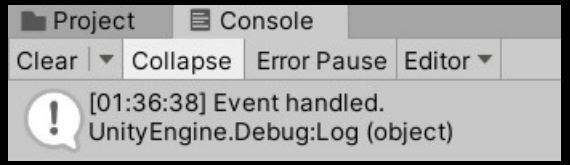
- The handler must have the same signature to the delegate. ❶
- The rest parts are the same.
  - Subscribing/Unsubscribing an Event(action binding):
    - Call subscribe when enabled(initialized).
    - Call unsubscribe when disabled.
      - Please do remember to call unsubscribe in OnDisable. Otherwise you'll have memory leakage when subscriber object is destroyed.
  - Implement the handler body.

```
1  using System;
2  using UnityEngine;
3
4  public class EX_Event_Subscriber : MonoBehaviour
5  {
6      public EX_Event_Center eventCenter;
7
8      private void OnEnable()
9      {
10         eventCenter.Subscribe(HandleEvent);
11     }
12
13     private void OnDisable()
14     {
15         eventCenter.Unsubscribe(HandleEvent);
16     }
17
18     public void HandleEvent(object sender, EventArgs e)
19     {
20         Debug.Log("Event handled.");
21     }
22 }
```

❶

EX: Raising(Invoke) an Event(2)

- When invoking, the arguments need to match the signature. ❶
- The rests parts are the same.
  - Create the publisher class
  - Raising an Event under proper condition:
    - Define the condition that raises(invoke) the event.




```
1  using System;
2  using UnityEngine;
3
4  public class EX_Event_Publisher : MonoBehaviour
5  {
6      public EX_Event_Center eventCenter;
7
8      private void Update()
9      {
10         if (Input.GetKeyDown("a"))
11         {
12             eventCenter.Invoke(this, new EventArgs());
13         }
14     }
15 }
```

❶

Differences Between C# Delegate and Event

- Key differences:
  - Access Control:
    - Delegate: Public, can be invoked by any class with access to the delegate instance.
    - Event: Restricted, can only be raised (invoked) within the declaring class.
  - Encapsulation:
    - Delegate: Directly accessible and invocable.
    - Event: Provides a layer of encapsulation, limiting invocation to the declaring class.
  - Usage Scenario:
    - Delegate: Used for callbacks and method references.
    - Event: Used for notifications and signaling state changes.
  - Multicast:
    - Both delegates and events support multicast (invoking multiple methods).
    - The return value of multicast delegate will be the return value of the last executed delegate. Since the order of execution is not guaranteed, it is suggested to use void as return type for multicast delegate/event.
- Conclusion
  - Delegates are more flexible and can be used in a wider range of scenarios where direct method references are needed.
  - Events provide a more controlled and encapsulated way to manage notifications and state changes, making them ideal for publisher-subscriber patterns.


互動程式設計III
Interactive Programming Design Integration
• Message System
• Method Calling
• Message
• Delegates
• Events
• UnityEvent

What is UnityEvent

- Definition:
  - UnityEvent is a specific class for event provided by the Unity Engine. It is designed to facilitate the creation and handling of events in Unity environment. It allows for methods to be called in response to certain triggers or actions within the Unity Editor.
- Usage:
  - Editor Integration: UnityEvents can be configured directly in the Unity Inspector, enabling designers and non-programmers to assign event handlers without writing code.
  - Component Communication: Used for communication between different components and game objects, making it easier to manage complex interactions in a modular way.
  - Callbacks: Commonly used for setting up callbacks for UI elements, game events, animations, and other interactive elements within a game.

EX: Defining an UnityEvent

- Create an event center and defining an event:
  - Declare a UnityEvent that doesn't has any argument ❶.
  - Later we are goint to Subscribe/Unsubscribe some UnityAction to that UnityEvent ❷.
  - Instead of +/-, use AddListener/RemoveListener to add and remove event listener methods❸.
- Implement the required event operation methods:
  - Subscribe/Unsubscribe: these are for the subscriber.
    - Please do remember to implement unsubscribe. Otherwise you'll have memory leakage when subscriber object is destroyed.
  - Invoke: this is for the publisher.
    - Use the ?. operator to ensure that there are subscribers before raising the event.

```
1  using UnityEngine;
2  using UnityEngine.Events;
3
4  public class EX_UnityEvent_Center : MonoBehaviour
5  {
6      public UnityEvent myEvent;
7
8      public void Invoke()
9      {
10         myEvent?.Invoke();
11     }
12
13     public void Subscribe(UnityAction action)
14     {
15         myEvent.AddListener(action);
16     }
17
18     public void Unsubscribe(UnityAction action)
19     {
20         myEvent.RemoveListener(action);
21     }
22 }
```

EX: Subscribing an UnityEvent

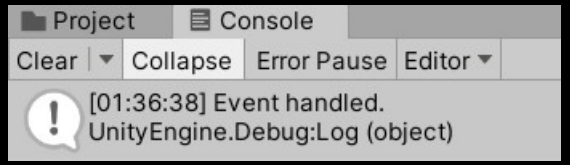
- Define the UnityAction with a Lambda expression (also known as anonymous function). ❶
  - Then bind this UnityAction to the event. ❷
- Subscribing/Unsubscribing an Event (action binding):
  - Call subscribe when enabled(initialized).
  - Call unsubscribe when disabled.
    - Please do remember to call unsubscribe in OnDisable. Otherwise you'll have memory leakage when subscriber object is destroyed.

```
1  using UnityEngine;
2  using UnityEngine.Events;
3
4  public class EX_UnityEvent_Subscriber : MonoBehaviour
5  {
6      public EX_UnityEvent_Center eventCenter;
7      public UnityAction action = new UnityAction(
8          // Lamda method: ()=>{}
9          () => { Debug.Log("Event handled."); }
10     );
11
12     private void OnEnable()
13     {
14         eventCenter.Subscribe(action);
15     }
16
17     private void OnDisable()
18     {
19         eventCenter.Unsubscribe(action);
20     }
21 }
```

❶ ❷

EX: Raising(Invoke) an UnityEvent

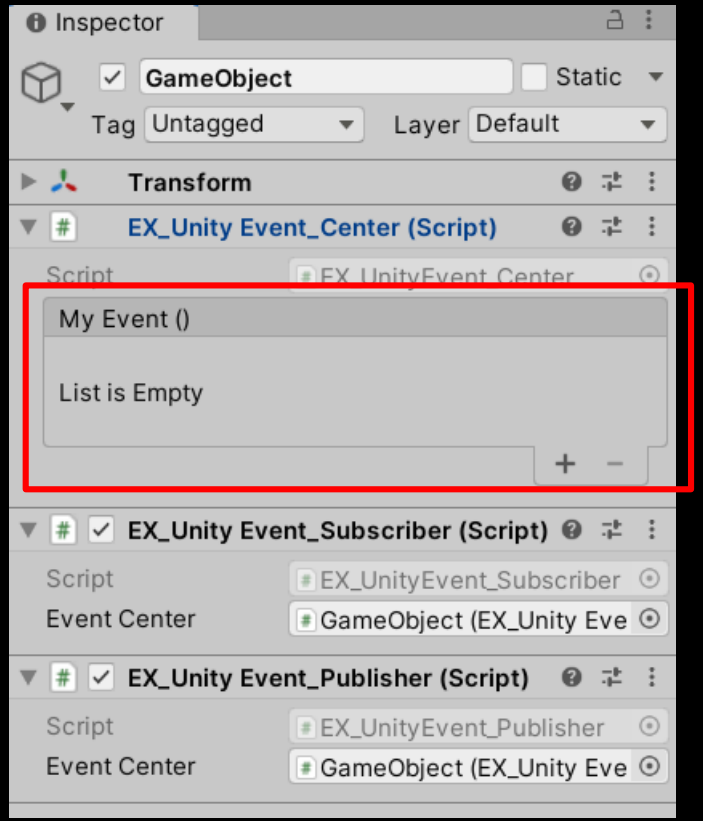
- Create the publisher class
- Raising an Event under proper condition:
  - Define the condition that raises(invoke) the event. ❶




```
1   using UnityEngine;
2
3   public class EX_UnityEvent_Publisher : MonoBehaviour
4   {
5       public EX_UnityEvent_Center eventCenter;
6
7       void Update()
8       {
9           if (Input.GetKeyDown("a"))
10          {
11              eventCenter.Invoke();  ❶
12          }
13      }
14  }
```

EX: Raising(Invoke) an UnityEvent

- UnityEvent can be maneuvered in UnityEditor
  - You can bind other actions to myEvent through UnityEditor.



Comparison of UnityEvent and C# Event

- C# Events:
  - Codes could be decoupled from UnityEngine.
    - Use standard event handling mechanism in C#. Events are members of a delegate type and are used for sending notifications.
    - Useful for high-performance, low-overhead event handling.
- UnityEvent:
  - Events could be maneuvered in UnityEditor.
    - A Unity-specific implementation of events, allowing for event handling via the Inspector.
    - Ideal for scenarios where you want to connect methods on different GameObject or components without direct drag and drop in UnityEditor.
- Key Differences:
  - Integration: UnityEvent integrates seamlessly with the Unity Editor, allowing event management directly in the Inspector, while C# events are managed purely through code.
  - Flexibility vs. Performance: UnityEvent offers more flexibility and ease of use in the Unity environment, but C# events are more performant due to lower overhead.
  - Use Cases: Use UnityEvent for quick prototyping, UI events, and designer-friendly workflows. Use C# events for performance-critical applications where the overhead of UnityEvent might be a concern.

EX: UnityEvent/UnityAction with Parameters

- If you need your UnityEvent/UnityAction to receive parameters, you have to declare the parameter as generic types to the UnityEvent/UnityAction.
  - In the example, the event will pass one string and one integer to the action.

```
UnityEvent<string, int>
```

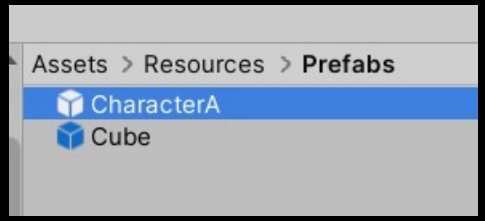
```
UnityAction<string, int>
```

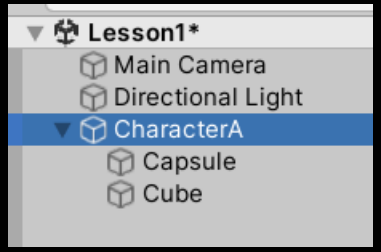
```
1  using UnityEngine;
2  using UnityEngine.Events;
3
4  public class EX_UnityEvent_Center_wParam : MonoBehaviour
5  {
6      public UnityEvent<string, int> myEvent;
7
8      public void Invoke(string str, int value)
9      {
10         myEvent?.Invoke(str, value);
11     }
12
13     public void Subscribe(UnityAction<string, int> action)
14     {
15         myEvent.AddListener(action);
16     }
17
18     public void Unsubscribe(UnityAction<string, int> action)
19     {
20         myEvent.RemoveListener(action);
21     }
22 }
```

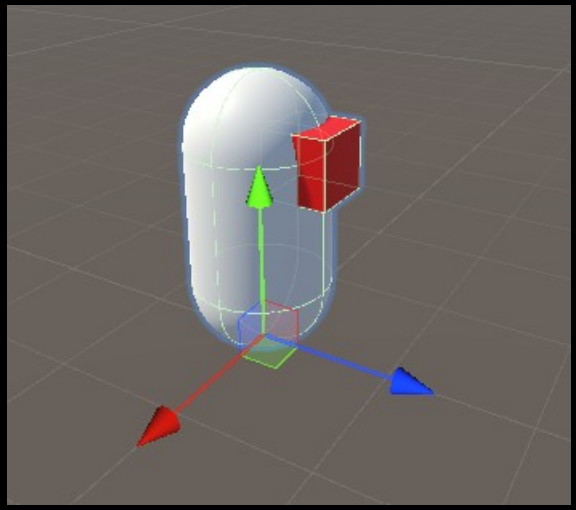

互動程式設計III
Interactive Programming Design Integration
• Message System
• Method Calling
• Message
• Delegates
• Events
• UnityEvent
• Examples

EX: Input Controller

- Use one Input Controller to control several Characters.
- Create a prefab <CharacterA> to represent for our character.
  - Capsule: the body of the character
  - Cube: the red eye of the character
  - Move the capsule 1m higher to match the position of capsule bottom and world ground.

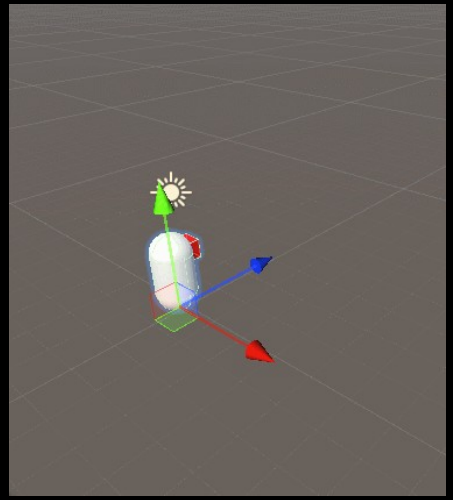






EX: Input Controller

- Create a class <EX_Character_wMove>
  - Create a Move function: move the character according to the received velocity vector.

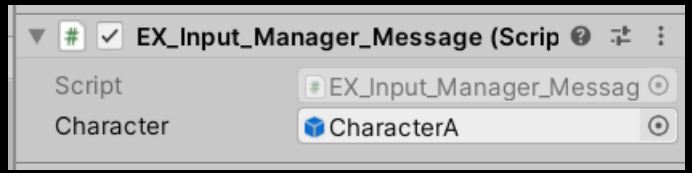


- Refactor <EX_Character_wMove> when test finished.
  - Remove all GetKey and delete Update.
  - Keep only the Move method.

```
1   using UnityEngine;
2
3   public class EX_Character_wMove : MonoBehaviour
4   {
5       private void Update()
6       {
7           if (Input.GetKey("w"))
8           {
9               Move(transform.forward * 2.0f);
10          }
11          if (Input.GetKey("s"))
12          {
13              Move(transform.forward * -2.0f);
14          }
15          if (Input.GetKey("d"))
16          {
17              Move(transform.right * 2.0f);
18          }
19          if (Input.GetKey("a"))
20          {
21              Move(transform.right * -2.0f);
22          }
23      }
24
25      public void Move(Vector3 velocity)
26      {
27          transform.position += velocity * Time.deltaTime;
28      }
29  }
```

EX: Input Controller - SendMessage

- Refactor <EX_Character_wMove>
  - Remove all GetKey



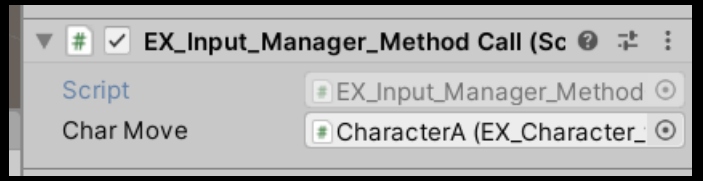

```
using UnityEngine;

public class EX_Input_Manager : MonoBehaviour
{
    public GameObject character;

    private void Update()
    {
        if (Input.GetKey("w"))
        {
            character.SendMessage("Move", (object)(transform.forward * 2.0f));
        }
        if (Input.GetKey("s"))
        {
            character.SendMessage("Move", (object)(transform.forward * -2.0f));
        }
        if (Input.GetKey("d"))
        {
            character.SendMessage("Move", (object)(transform.right * 2.0f));
        }
        if (Input.GetKey("a"))
        {
            character.SendMessage("Move", (object)(transform.right * -2.0f));
        }
    }
}
```

```
using UnityEngine;

public class EX_Character_wMove : MonoBehaviour
{
    public void Move(Vector3 velocity)
    {
        transform.position += velocity * Time.deltaTime;
    }
}
```

EX: Input Controller - Method Call

-

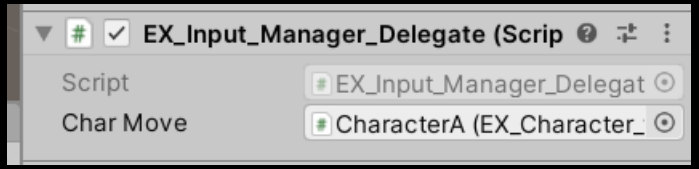
```
using UnityEngine;

public class EX_Input_Manager_MethodCall : MonoBehaviour
{
    public EX_Character_wMove charMove;

    private void Update()
    {
        if (Input.GetKey("w"))
        {
            charMove.Move(transform.forward * 2.0f);
        }
        if (Input.GetKey("s"))
        {
            charMove.Move(transform.forward * -2.0f);
        }
        if (Input.GetKey("d"))
        {
            charMove.Move(transform.right * 2.0f);
        }
        if (Input.GetKey("a"))
        {
            charMove.Move(transform.right * -2.0f);
        }
    }
}
```




```
using UnityEngine;

public class EX_Character_wMove : MonoBehaviour
{
    public void Move(Vector3 velocity)
    {
        transform.position += velocity * Time.deltaTime;
    }
}
```

EX: Input Controller - Delegate

```
using UnityEngine;

public class EX_Input_Manager_Delegate : MonoBehaviour
{
    public EX_Character_wMove charMove;
    public delegate void DelegateMove(Vector3 velocity);
    private DelegateMove delegateMove;

    private void OnEnable()
    {
        delegateMove += charMove.Move;
    }

    private void OnDisable()
    {
        delegateMove -= charMove.Move;
    }

    private void Update()
    {
        if (Input.GetKey("w"))
        {
            delegateMove(transform.forward * 2.0f);
        }
        if (Input.GetKey("s"))
        {
            delegateMove(transform.forward * -2.0f);
        }
        if (Input.GetKey("d"))
        {
            delegateMove(transform.right * 2.0f);
        }
        if (Input.GetKey("a"))
        {
            delegateMove(transform.right * -2.0f);
        }
    }
}
```




```
using UnityEngine;

public class EX_Character_wMove : MonoBehaviour
{
    public void Move(Vector3 velocity)
    {
        transform.position += velocity * Time.deltaTime;
    }
}
```

EX: Input Controller - C# Event

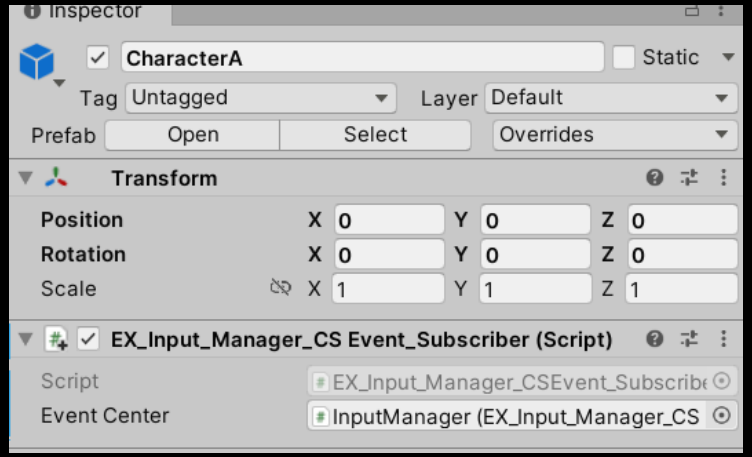
```cs
using UnityEngine;

public class EX_Input_Manager_CSEvent_EventCenter : MonoBehaviour
{
    public delegate void DelegateMove(Vector3 velocity);
    private event DelegateMove eventMove;

    public void Invoke(Vector3 velocity)
    {
        eventMove.Invoke(velocity);
    }

    public void Subscribe(DelegateMove action)
    {
        eventMove += action;
    }

    public void Unsubscribe(DelegateMove action)
    {
        eventMove -= action;
    }
}
```

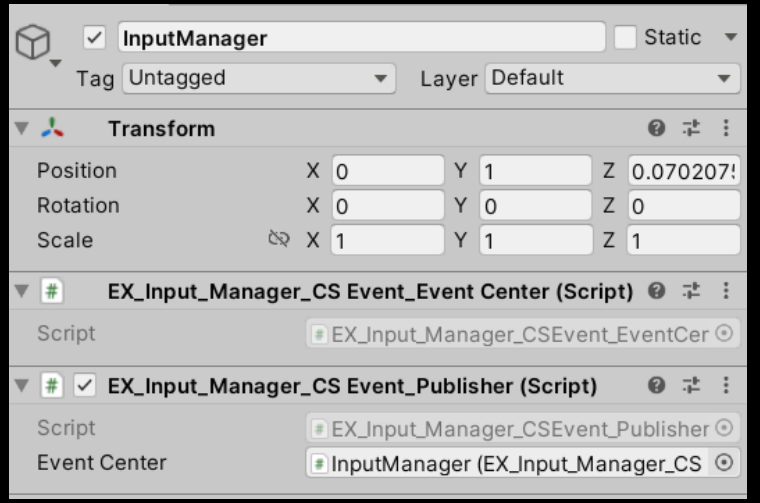
```cs
using UnityEngine;

public class EX_Input_Manager_CSEvent_Publisher : MonoBehaviour
{
    public EX_Input_Manager_CSEvent_EventCenter eventCenter;

    private void Update()
    {
        if (Input.GetKey("w"))
        {
            eventCenter.Invoke(transform.forward * 2.0f);
        }
        if (Input.GetKey("s"))
        {
            eventCenter.Invoke(transform.forward * -2.0f);
        }
        if (Input.GetKey("d"))
        {
            eventCenter.Invoke(transform.right * 2.0f);
        }
        if (Input.GetKey("a"))
        {
            eventCenter.Invoke(transform.right * -2.0f);
        }
    }
}
```

EX: Input Controller - C# Event

-

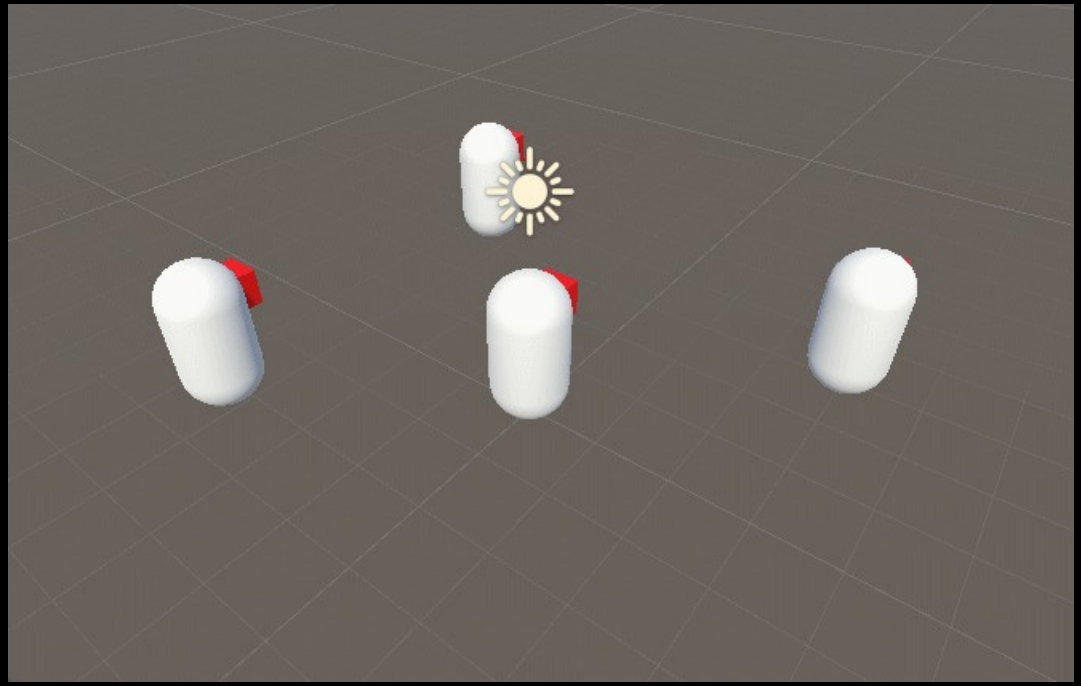
```
1   using UnityEngine;
2
3   public class EX_Input_Manager_CSEvent_Subscriber : MonoBehaviour
4   {
5       public EX_Input_Manager_CSEvent_EventCenter eventCenter;
6
7       private void OnEnable()
8       {
9           eventCenter.Subscribe(Move);
10      }
11
12      private void OnDisable()
13      {
14          eventCenter.Unsubscribe(Move);
15      }
16
17      public void Move(Vector3 velocity)
18      {
19          transform.position += velocity * Time.deltaTime;
20      }
21  }
```





EX: Input Controller - C# Event

- Create several Characters to see the effect.



EX: Input Controller - UnityEvent

-

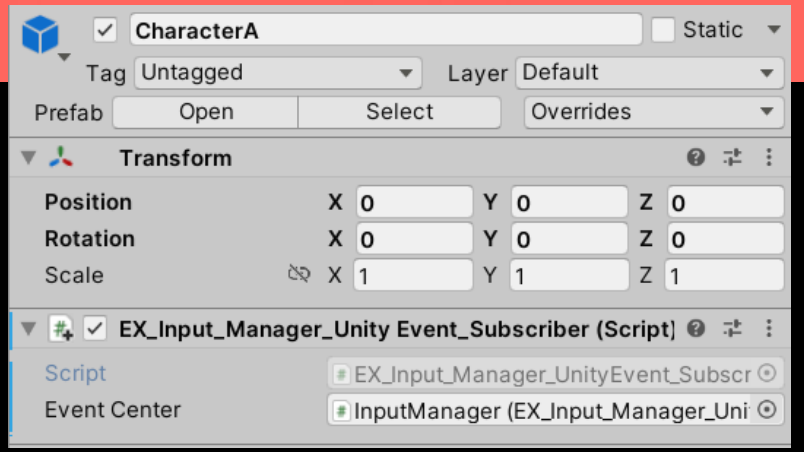
```cpp
using UnityEngine;
using UnityEngine.Events;

public class EX_Input_Manager_UnityEvent_EventCenter : MonoBehaviour
{
    public UnityEvent<Vector3> eventMove;

    public void Invoke(Vector3 velocity)
    {
        eventMove.Invoke(velocity);
    }

    public void Subscribe(UnityAction<Vector3> action)
    {
        eventMove.AddListener(action);
    }

    public void Unsubscribe(UnityAction<Vector3> action)
    {
        eventMove.RemoveListener(action);
    }
}
```

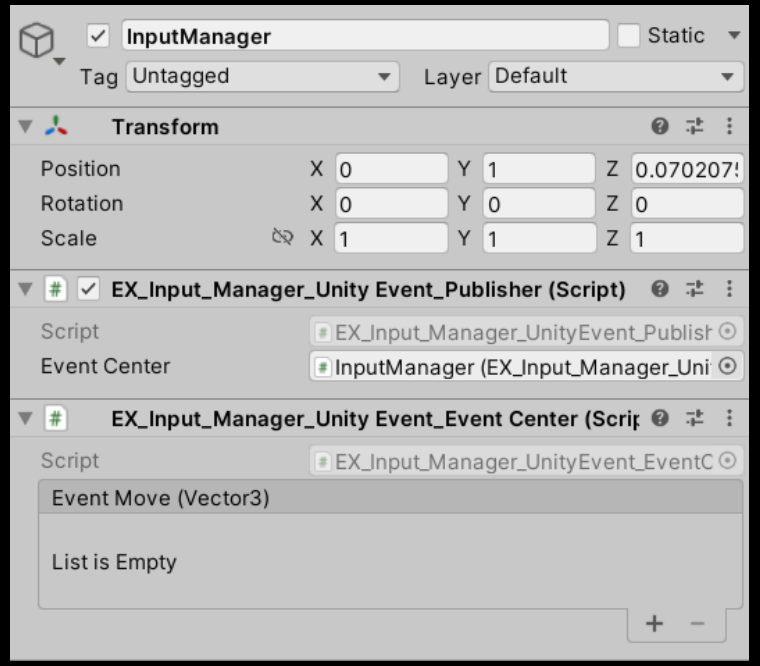
```cpp
using UnityEngine;

public class EX_Input_Manager_UnityEvent_Publisher : MonoBehaviour
{
    public EX_Input_Manager_UnityEvent_EventCenter eventCenter;

    private void Update()
    {
        if (Input.GetKey("w"))
        {
            eventCenter.Invoke(transform.forward * 2.0f);
        }
        if (Input.GetKey("s"))
        {
            eventCenter.Invoke(transform.forward * -2.0f);
        }
        if (Input.GetKey("d"))
        {
            eventCenter.Invoke(transform.right * 2.0f);
        }
        if (Input.GetKey("a"))
        {
            eventCenter.Invoke(transform.right * -2.0f);
        }
    }
}
```

EX: Input Controller - UnityEvent

-

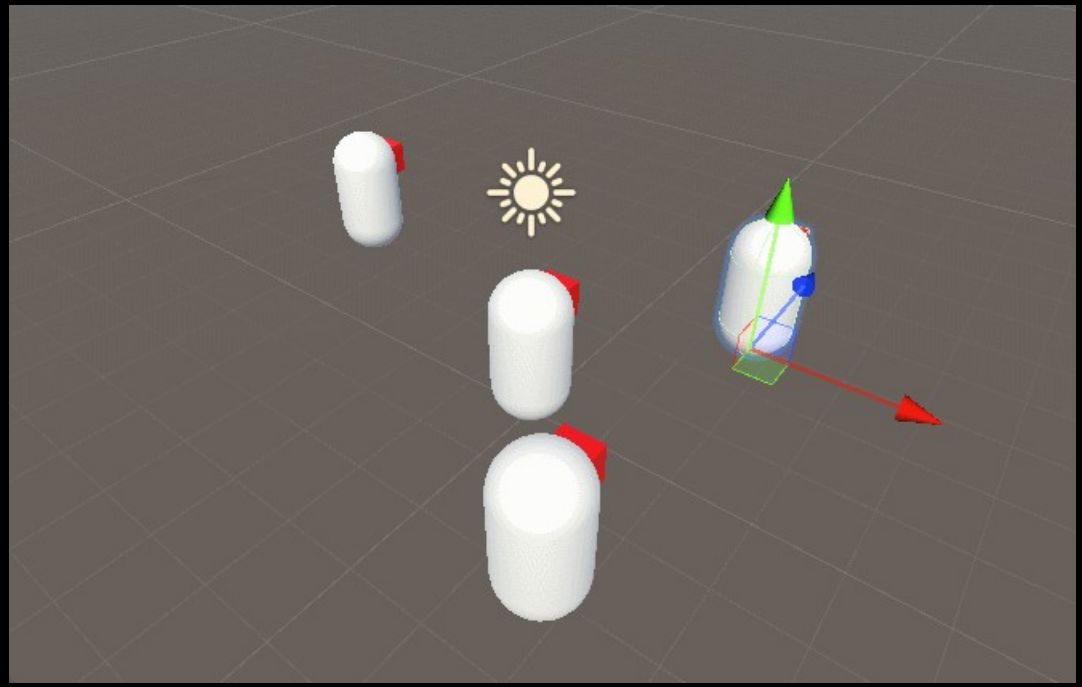
```
1   using UnityEngine;
2
3   public class EX_Input_Manager_UnityEvent_Subscriber : MonoBehaviour
4   {
5       public EX_Input_Manager_UnityEvent_EventCenter eventCenter;
6
7       private void OnEnable()
8       {
9           eventCenter.Subscribe(Move);
10      }
11
12      private void OnDisable()
13      {
14          eventCenter.Unsubscribe(Move);
15      }
16
17      public void Move(Vector3 velocity)
18      {
19          transform.position += velocity * Time.deltaTime;
20      }
21  }
```





EX: Input Controller - UnityEvent

- Create several Characters to see the effect.




互動程式設計III
Interactive Programming Design Integration
Q&A