«Talento Tech»

Desarrollo de Videojuegos

Unity 3D

Clase 06

Buenos Aires Ciudad

Clase N° 6 | Delegados y Eventos II

Temario:

- Introducción al Observer Pattern
- Generics
- EventManager

Objetivos de la clase

Hasta ahora, aprendiste cómo usar Delegados y Eventos para comunicar acciones dentro de tu juego. Sin embargo, a medida que un proyecto crece en complejidad, la necesidad de un sistema de comunicación más ordenado y escalable se vuelve evidente.

En esta clase, vas a llevar tus habilidades al siguiente nivel usando tres herramientas clave:

- **Observer Pattern,** para modelar relaciones uno-a-muchos de forma limpia.
- **Generics,** para hacer tu código más reutilizable y seguro.
- **EventManager,** para centralizar y administrar eventos de forma eficiente.

Observer Pattern

El **Observer Pattern** (o patrón observador) es un patrón de diseño que establece una relación entre objetos para que cuando uno de ellos cambie su estado, los otros sean notificados automáticamente. Es muy utilizado en programación para manejar eventos o actualizaciones en tiempo real, y es la base conceptual detrás de los **eventos** en lenguajes como C#.

El patrón observador define una relación **uno a muchos** entre objetos, donde:

1. **Uno**: El objeto observado o sujeto (**Subject**).
    - Es quien tiene la información o el estado que otros quieren monitorear.
2. **Muchos**: Los objetos observadores (**Observers**).
    - Son quienes se registran (o suscriben) para recibir notificaciones sobre cambios en el sujeto.

Cuando el **sujeto** cambia su estado, automáticamente notifica a todos sus observadores.

Pensá en un canal de YouTube:

1. El canal es el **sujeto** (Subject).
2. Los usuarios suscritos son los **observadores** (Observers).
3. Cuando el canal sube un nuevo video, notifica automáticamente a todos los suscriptores.

Estructura básica:

1. **Sujeto**: La entidad que mantiene una lista de observadores y los notifica cuando ocurre un cambio.
2. **Observadores**: Los objetos que "se suscriben" al sujeto para recibir notificaciones.
3. **Desacoplamiento**: El sujeto no necesita conocer los detalles de los observadores, lo que facilita la escalabilidad y el mantenimiento.

Ejemplo Básico

Vamos a aplicar este patrón en un caso muy común:
El jugador gana puntos → el sistema de puntaje cambia → el HUD se actualiza automáticamente.

Clase Sujeto: ScoreManager

En nuestro ejemplo, el sujeto es el ScoreManager. Su función es gestionar el puntaje del juego y emitir un evento cuando ese puntaje cambia.

```
public class ScoreManager{
    // Definimos un delegado explícito
    public delegate void ScoreChangedDelegate(int newScore);
    public event ScoreChangedDelegate OnScoreChanged;
    private int score;

    public void AddScore(int points) {
        score += points;
        // Llamamos al evento cuando se actualiza la puntuación
        if (OnScoreChanged != null)
            OnScoreChanged.Invoke(score);
    }
}
```

¿Qué hace este código?

- ScoreChangedDelegate define la firma del método que será llamado por el evento. En este caso, cualquier método que reciba un int y no devuelva nada.
- OnScoreChanged es el evento que se dispara cada vez que el puntaje se actualiza.
- AddScore() suma puntos y luego invoca el evento con el nuevo valor.

💡 Así, cualquier objeto que esté “escuchando” el evento será notificado automáticamente cuando el puntaje cambie.

Clase Observador: ScoreUI

Ahora que tenemos un evento funcionando, necesitamos alguien que lo **escuche y reaccione**. En este caso, creamos un `ScoreUI` que se **suscribe al evento** del `ScoreManager` para **actualizar el texto del puntaje en pantalla**.

```
using TMPro;
public class ScoreUI : MonoBehaviour{
    [SerializeField] private TextMeshProUGUI scoreText;
 void Start(){
        OnEnable();
    }
  private void OnEnable(){
   //Nos suscribimos al evento para escuchar cambios en la puntuación
                    FindObjectOfType<ScoreManager>().OnScoreChanged   +=
UpdateScoreText;
    }
private void OnDisable(){
//Nos desuscribimos cuando el objeto se desactiva para evitar problemas
                    FindObjectOfType<ScoreManager>().OnScoreChanged   -=
UpdateScoreText;
    }
    private void UpdateScoreText(int newScore){
        scoreText.text = "Score: " + newScore;
    }
}
```

¿Qué hace este código?

- Al activarse (`OnEnable`), busca una instancia de `ScoreManager` en escena y **se suscribe al evento** `OnScoreChanged`.
- Cuando el evento se dispara, ejecuta `UpdateScoreText()`, que actualiza el texto con el nuevo puntaje.
- Al desactivarse (`OnDisable`), se **desuscribe** del evento para evitar errores si el objeto desaparece o se destruye.

> 💡 Este patrón permite tener **múltiples observadores** escuchando un mismo evento: podríamos tener un sonido, una animación o un logro activándose junto con la UI, todo desde el mismo `OnScoreChanged`.

Resultado

Con esta estructura:

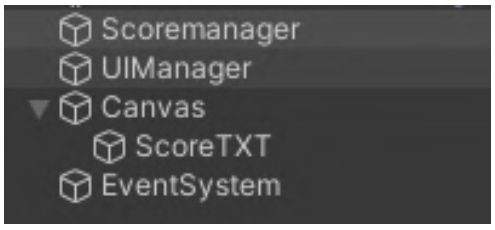
- El `ScoreManager` solo se encarga de sumar puntos.
- El `ScoreUI` se actualiza sin necesidad de que nadie lo llame directamente.
- Se puede agregar más observadores (logros, sonido, etc.) sin tocar el `ScoreManager`.

Configuración en Unity:

1. **Crear un objeto para el `ScoreManager`:**
    - Creá un **Empty Object** en la jerarquía y renombralo `ScoreManager`.
    - Asignale el script `ScoreManager`.

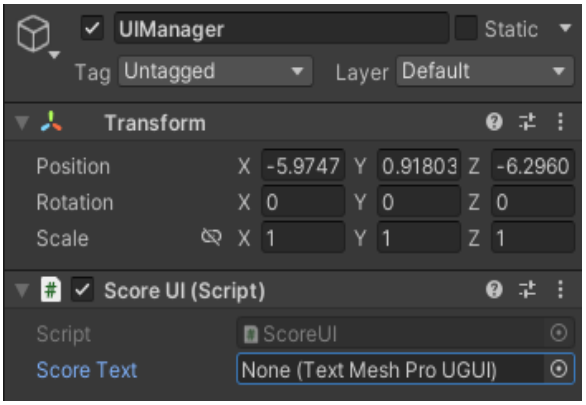
Este objeto puede permanecer en toda la escena o incluso usarse como parte de un sistema persistente.

2. **Crear la interfaz de usuario para mostrar la puntuación:**
    - Asegurate de tener un `Canvas` en la escena.
    - Dentro del Canvas, creá un objeto `TextMeshPro - Text` y nombralo `ScoreText`..



    - Este objeto mostrará el puntaje en pantalla.

3. **Añadir el script `ScoreUI`:**
    - Creá un nuevo GameObject vacío o dentro del Canvas, y agregale el script `ScoreUI`.
    - En el `Inspector`, arrastrá el `ScoreText` al campo `scoreText` del script `ScoreUI`.



Generics

Los **Generics** son una herramienta fundamental en C# que permite escribir código más **general, reutilizable y seguro**, sin perder el control sobre los tipos de datos.

En lugar de repetir una clase o método para cada tipo de dato, usamos Generics para decir: *"Esta lógica funciona con cualquier tipo... mientras cumpla ciertas condiciones."*

En C# y Unity, los **Generics Types** son una característica del lenguaje que permite definir clases, interfaces, métodos y estructuras que pueden funcionar con cualquier tipo de dato. La principal ventaja de los genéricos es que permiten la reutilización del código sin sacrificar la seguridad de tipos en tiempo de compilación. Esto ayuda a crear clases o métodos más flexibles y reutilizables, manteniendo la integridad de los tipos de datos.

¿Cómo funcionan los Generics en C#?

Un Generic se define mediante el uso de un tipo de marcador (comúnmente llamado parámetro de tipo) en lugar de un tipo específico. Este parámetro de tipo se reemplaza por un tipo concreto cuando se crea una instancia de la clase o se llama al método. Los parámetros de tipo se denotan con un tipo genérico en la definición.

¿Por qué usar Generics?

- Evita repetir código para distintos tipos de datos (por ejemplo, `int`, `string`, `GameObject`).
- Mejora la legibilidad y mantenimiento del proyecto.
- Garantiza que los tipos sean correctos en tiempo de compilación.
- Muy útil en **herramientas, sistemas de eventos o colecciones personalizadas**.

Ejemplo de un tipo genérico en C#:

```
public class Caja<T>{
    private T contenido;
    public void Guardar(T item) {
        contenido = item;
    }
    public T Obtener(){
        return contenido;
    }
}
```

¿Qué significa <T>?

- `<T>` es un **parámetro genérico**. Puede ser reemplazado por cualquier tipo de dato cuando se usa la clase.
- En este caso, `Caja<T>` funciona con `Caja<int>`, `Caja<string>`, `Caja<GameObject>`, etc.

En este ejemplo, T es un parámetro de tipo genérico. Puedes usar esta clase para almacenar cualquier tipo de objeto, por ejemplo:

```
    private void Start(){
        // Crear una caja de enteros y guardar un valor
        Caja<int> cajaDeEnteros = new Caja<int>();
        cajaDeEnteros.Guardar(10);
        int numero = cajaDeEnteros.Obtener();
        Debug.Log("Número guardado: " + numero);

        // Crear una caja de strings y guardar un valor
        Caja<string> cajaDeStrings = new Caja<string>();
        cajaDeStrings.Guardar("Hola Mundo");
        string mensaje = cajaDeStrings.Obtener();
        Debug.Log("Mensaje guardado: " + mensaje);
    }
```

📌 Mismo código, múltiples usos, sin duplicar clases ni métodos.

Aplicación directa en Unity: asegurarse de tener un componente:

Imagina que estás trabajando con varios componentes en un GameObject, como `Rigidbody`, `MeshRenderer`, etc., y necesitas un método genérico para acceder o agregar componentes según sea necesario. Aquí es donde los **Generics** resultan útiles.

Clase Genérica para Manejar Componentes

```
using UnityEngine;
public static class ComponenteHelper{
    // Método genérico para obtener o agregar un componente
    public static T ObtenerOAgregarComponente<T>(GameObject gameObject)
where T : Component{
        T componente = gameObject.GetComponent<T>();
        if (componente == null) {
            componente = gameObject.AddComponent<T>(); }
      return componente; }}
```

¿Qué hace este helper?

- Busca el componente del tipo T en el objeto.
- Si no lo encuentra, lo agrega.
  Siempre devuelve un componente del tipo solicitado.

¿Cómo usar esta clase en Unity?

Pueden usar este método genérico para asegurarte de que un componente específico esté presente en un GameObject, y si no está, se agrega automáticamente.

```
using UnityEngine;
public class EjemploUsoComponenteHelper : MonoBehaviour{
    void Start() {
        // Obtiene o agrega un Rigidbody al GameObject actual
        Rigidbody rb =
ComponenteHelper.ObtenerOAgregarComponente<Rigidbody>(gameObject);
        rb.mass = 5f; // Configura alguna propiedad del Rigidbody
        // Obtiene o agrega un MeshRenderer al GameObject actual
     MeshRenderer mr =
ComponenteHelper.ObtenerOAgregarComponente<MeshRenderer>(gameObject);
        mr.material.color = Color.red; // Cambia el color del material
    }}
```

Explicación

- El método genérico `ObtenerOAgregarComponente<T>` utiliza un parámetro de tipo genérico T, que debe ser un componente (`where T : Component`).
- Si el GameObject ya tiene un componente del tipo solicitado, simplemente lo devuelve.
- Si el componente no existe, se agrega automáticamente al GameObject usando `AddComponent<T>()`.

EventManager.

A medida que los proyectos crecen, usar eventos sueltos en distintas clases empieza a volverse difícil de mantener. Para resolver esto, vamos a construir un **EventManager genérico**: una clase que centraliza la administración de eventos usando **Generics y diccionarios**.

¿Para qué sirve?

- Agrupa todos los eventos del juego en una única estructura.
- Permite usar **eventos con diferentes tipos de parámetros** (como `int`, `string`, `bool`, etc.).
- Evita escribir múltiples `delegate` y `event` manualmente.
- Permite suscribirse a un evento solo con su nombre y tipo.

Como siempre, primero presentaremos el código y lo iremos desglosando parte por parte:

```
using System.Collections.Generic;
using UnityEngine;
using System;
public class EventManager3 : MonoBehaviour{
    // Define un delegate explícito sin parámetros
    public delegate void EventDelegate<T>(T param);
     private static Dictionary<string, Delegate> eventDictionary = new
Dictionary<string, Delegate>();
     public static void Subscribe<T>(string eventName, EventDelegate<T>
subscriber){
   if (eventDictionary.TryGetValue(eventName, out Delegate thisEvent)){
 eventDictionary[eventName] = (EventDelegate<T>)thisEvent + subscriber;
        } else{
            eventDictionary.Add(eventName, subscriber);
        }
    }
public static void Unsubscribe<T>(string eventName, EventDelegate<T>
subscriber){
   if (eventDictionary.TryGetValue(eventName, out Delegate thisEvent)){
            var currentDel = (EventDelegate<T>)thisEvent - subscriber;
            if (currentDel == null){
                eventDictionary.Remove(eventName);
            }
            else{
                eventDictionary[eventName] = currentDel;
            }
        }
```

```
    }
public static void Notify<T>(string thisEvent, T param){
          // Comprobamos si 'thisEvent' es de tipo 'EventDelegate<T>'
antes de intentar invocar el delegado
      EventDelegate<T> eventDelegate = thisEvent as EventDelegate<T>;
      if (eventDelegate != null){
      // Si la conversión es exitosa, invocamos el delegado pasando
'param' como argumento
         eventDelegate.Invoke(param);
      }
      else{
    // Si 'thisEvent' no puede ser convertido a 'EventDelegate<T>', no
hacemos nada o podemos manejarlo aquí.
         Console.WriteLine("El evento no es del tipo correcto.");
      }
     }
}
```

Desglosando el código:

Declaración del delegado

```
public delegate void EventDelegate<T>(T param);
```

Un **delegado** es un tipo que define la forma de un método. Aquí, `EventDelegate` es un delegado genérico que acepta un parámetro de tipo T y no devuelve nada (`void`).

Por ejemplo:

- Si `T` es un `string`, un método asociado al delegado tendría la forma `void MyMethod(string param)`.

Diccionario para manejar eventos

```
private static Dictionary<string, Delegate> eventDictionary = new
Dictionary<string, Delegate>();
```

El diccionario `eventDictionary` almacena una lista de eventos. Cada entrada tiene:

- **Clave (`string`)**: Nombre del evento.
- **Valor (`Delegate`)**: Delegado que representa una lista de suscriptores para ese evento.

Método Subscribe

```
public static void Subscribe<T>(string eventName, EventDelegate<T>
subscriber){
 if (eventDictionary.TryGetValue(eventName, out Delegate thisEvent)){
            eventDictionary[eventName] = (EventDelegate<T>)thisEvent +
subscriber;
    }
    else{
        eventDictionary.Add(eventName, subscriber);
    }
}
```

Este método permite **suscribir** un método a un evento:

- **Parámetros**:
    1. `eventName`: Nombre del evento.
    2. `subscriber`: Método que desea suscribirse.
- **Funcionamiento**:
    1. Usa `TryGetValue` para buscar el evento por su nombre:
        - Si existe, combina (+) el nuevo suscriptor con los ya existentes.
        - Si no existe, crea una nueva entrada en el diccionario.
- **Nota**: El operador + en delegados agrega un nuevo método a la lista de invocaciones del evento.

```
public static void Subscribe<T>(string eventName, EventDelegate<T>
subscriber)
```

- **`public static`**: Este método es accesible desde cualquier parte del código sin necesidad de crear una instancia del `EventManager`.
- **`<T>`**: Se utiliza un tipo genérico T, lo que significa que este método puede trabajar con eventos que acepten cualquier tipo de parámetro (como `int`, `float`, o incluso un objeto personalizado).
- **`eventName`**: Es una cadena que identifica el evento al que se desea suscribir.
- **`subscriber`**: Es el método o función (representado por un `delegate`) que se ejecutará cuando el evento ocurra.

```
if (eventDictionary.TryGetValue(eventName, out Delegate thisEvent))
```

- **`eventDictionary`**: Es el diccionario que almacena todos los eventos registrados. La clave es el nombre del evento (`string`), y el valor es un `Delegate`, que puede ser un delegado explícito o genérico.
- **`TryGetValue`**: Intenta buscar un evento con el nombre especificado (`eventName`) en el diccionario.

- Si el evento existe, lo asigna a la variable `thisEvent` y devuelve `true`.
- Si el evento no existe, `thisEvent` será `null` y la condición será `false`.

```
eventDictionary[eventName] = (EventDelegate<T>)thisEvent + subscriber;
```

Si el evento **ya existe** en el diccionario:

1. **`(EventDelegate<T>)thisEvent`**: Convierte(Castea) el `Delegate` existente (`thisEvent`) en un tipo más específico: `EventDelegate<T>`. Esto es posible porque sabemos que los eventos registrados deben coincidir con este tipo genérico.
2. **`thisEvent + subscriber`**: Agrega el nuevo método (`subscriber`) a la lista de métodos asociados a este evento. En C#, los `delegates` son combinables; puedes agregar funciones mediante el operador +.
3. **`eventDictionary[eventName]`**: Actualiza la entrada del diccionario para incluir al nuevo suscriptor.

```
else{
    eventDictionary.Add(eventName, subscriber);
}
```

Si el evento **no existe** en el diccionario:

1. Se crea una nueva entrada en el diccionario usando el nombre del evento (`eventName`) como clave.
2. Se asigna el nuevo suscriptor (`subscriber`) como el valor inicial del evento.

Método `Unsubscribe`

```
public static void Unsubscribe<T>(string eventName, EventDelegate<T>
subscriber){
if (eventDictionary.TryGetValue(eventName, out Delegate thisEvent)){
        var currentDel = (EventDelegate<T>)thisEvent - subscriber;
        if (currentDel == null){
            eventDictionary.Remove(eventName);
        }
        else{
            eventDictionary[eventName] = currentDel;
        }
    }
}
```

Este método permite **desuscribirse** de un evento:

- **Parámetros**:
    1. `eventName`: Nombre del evento.
    2. `subscriber`: Método que desea desuscribirse.
- **Funcionamiento**:
    1. Busca el evento en el diccionario:
        - Si existe, elimina (`-`) el suscriptor de la lista de delegados.
    2. Si no quedan más suscriptores (`currentDel == null`), elimina el evento del diccionario.
    3. De lo contrario, actualiza el diccionario con la nueva lista de delegados.

Método `Notify`

```
 public static void Notify<T>(string eventName, T param){
 if (eventDictionary.TryGetValue(eventName, out Delegate thisEvent)){
//Intentar convertir el delegado a EventDelegate<T> antes de invocarlo
   if (thisEvent is EventDelegate<T> eventDelegate){
        eventDelegate.Invoke(param);
    }
    else{
 Debug.LogWarning($"El evento '{eventName}' no es del tipo esperado.");
     }
 }
 else {
Debug.LogWarning($"No hay suscriptores para el evento '{eventName}'.");
  }
}
```

```
if (eventDictionary.TryGetValue(eventName, out Delegate thisEvent))
```

Busca si el evento identificado por el nombre `eventName` existe en el diccionario `eventDictionary`.

Resultado:

- Si existe, almacena el delegado asociado al evento en la variable `thisEvent`.
- Si no existe, pasa al bloque `else`.

```
if (thisEvent is EventDelegate<T> eventDelegate)
```

Verifica si `thisEvent` puede convertirse al tipo `EventDelegate<T>`, que es el tipo genérico esperado para este evento.

Resultado:

- Si la conversión es exitosa, guarda el delegado convertido en la variable `eventDelegate` y continúa.
- Si la conversión falla (por ejemplo, si se intenta notificar con un tipo de parámetro incorrecto), pasa al bloque `else`.

```
eventDelegate.Invoke(param);
```

Llama al delegado, que a su vez ejecuta todos los métodos suscritos al evento, pasando el parámetro `param` como argumento.

```
Debug.LogWarning($"El evento '{eventName}' no es del tipo esperado.");
```

Genera un mensaje en la consola de Unity advirtiendo que el evento no tiene el tipo esperado. Esto ayuda a identificar problemas de tipo durante el desarrollo.

```
Debug.LogWarning($"No hay suscriptores para el evento '{eventName}'.");
```

Si el evento no está registrado en el diccionario (es decir, no tiene suscriptores), genera un mensaje en la consola indicando que no hay nadie escuchando para ese evento.

Con este último paso, podremos crear un Event Manager que ayude a organizar todos nuestros eventos.

Materiales y recursos adicionales.

Generics:
https://learn.unity.com/tutorial/genericos#

Delegates:
https://learn.unity.com/tutorial/delegados#

Events:
https://learn.unity.com/tutorial/eventos-w#5e419557edbc2a0a62170fe6

Preguntas para reflexionar.

1. ¿Qué tan necesario puede ser un Manager?
2. ¿En que nos simplifica el uso de Events?
3. ¿Qué beneficios nos puede dar el uso de Generics?

Próximos pasos.

En la próxima clase empezaremos a trabajar el GameDesign. Estas clases fueron de mucho código y trabajos desafiantes, así que pasaremos a momentos de reflexión y creatividad sin perder el punto de vista técnico.

Más Eventos en TalentoLab:

Después de implementar los primeros eventos en Nexus, el cliente quedó satisfecho, pero pronto surgió un nuevo desafío. Con el creciente número de interacciones y eventos, el código del juego comenzó a volverse desorganizado y difícil de escalar. El cliente notó que mantener una comunicación eficiente entre los diferentes sistemas será crucial para el éxito del proyecto, especialmente si el juego continúa creciendo en complejidad.

Es aquí donde TalentoLab propone llevar la comunicación del juego al siguiente nivel con tres herramientas fundamentales: el Observer Pattern, el uso de Generics y la implementación de un EventManager. Estas herramientas permitirán organizar los eventos del juego de manera centralizada y flexible, asegurando que el código sea limpio, escalable y fácil de mantener.

Ejercicios prácticos:

Roberta, directora del proyecto Nexus en TalentoLab, quedó impresionada con tu implementación del sistema de eventos. Ahora te confía una nueva misión: **rediseñar toda la gestión de eventos del juego** para que sea más eficiente, ordenada y escalable.
Ya no se trata de resolver casos puntuales, sino de construir una **arquitectura sólida para todo el juego**.

Objetivo general

Reorganizar los eventos existentes en el proyecto (por ejemplo: `Game Over`, `Score`, `Zona secreta`) utilizando el nuevo **EventManager genérico** desarrollado en esta clase.

1. Implementar el `EventManager3`

- *Asegurate de tener el script del `EventManager` genérico en tu proyecto.*
- *Revisá que incluya los métodos: `Subscribe`, `Unsubscribe`, `Notify` y el diccionario central de eventos.*

2. Reemplazar eventos sueltos por el EventManager

*Elegí al menos **uno de los siguientes sistemas ya trabajados** y adaptalo para que use el EventManager genérico:*

- *El sistema de **puntaje (`ScoreManager`)***

- *La **pantalla de Game Over** al morir el jugador*
- *El **desbloqueo de zona secreta** al recolectar monedas*

> *💡 Tip: Podés usar directamente los scripts de la clase anterior (Clase 5) o del comienzo de esta clase.*

3. Crear nombres identificadores para los eventos

Usá `string` como clave para cada evento en el `EventManager`, por ejemplo:

- `"ScoreChanged"`
- `"PlayerDied"`
- `"SecretZoneUnlocked"`

4. Modificar los emisores

- *En el script correspondiente (por ejemplo, `ScoreManager`), **reemplazá la invocación de un evento directo** por una llamada a `EventManager3.Notify(...)`.*

5. Modificar los receptores

- *En los scripts que antes se suscribían a un evento directo, ahora usá `EventManager3.Subscribe(...)` en `OnEnable()` y `Unsubscribe(...)` en `OnDisable()`.*

Buenos Aires aprende

Agencia de Habilidades para el Futuro

Buenos Aires Ciudad