«Talento Tech»

Desarrollo de Videojuegos

Unity 3D

Clase 05

Buenos Aires Ciudad

Clase N° 5 | Delegados y Eventos I

Temario:

- Introducción a Delegates: qué son y cómo funcionan.
- Uso básico de Delegates & Events en C#.
- Aplicaciones iniciales: llamados a funciones específicas desde eventos.

Objetivos de la clase

A esta altura del curso ya tenés personajes que se mueven, mundos que reaccionan y animaciones que dan vida a la escena. Pero aún falta algo esencial: **comunicación interna entre los sistemas del juego**.

Por ejemplo, ¿cómo sabés que el jugador recogió un objeto? ¿Cómo se enteran otros scripts, como el UI o el GameManager, que ocurrió un evento importante?

Para resolver esto, hoy vas a aprender a usar dos herramientas fundamentales de C#:

- **Delegados (Delegates)**: permiten invocar funciones de forma flexible y dinámica.

- **Eventos (Events)**: permiten que distintas partes del juego escuchen y respondan a lo que ocurre sin estar acopladas directamente.

Estos conceptos no solo te permitirán optimizar tu código, sino también **crear experiencias reactivas**, donde cada acción del jugador puede generar múltiples respuestas en diferentes sistemas.

¿Qué son los Delegates y los Events?

Son herramientas que usaremos para comunicar de manera más eficiente nuestro juego. En vez de estar chequeando constantemente si algo sucedió o realizar una cadena de acontecimientos uno detrás de otro, podremos crear un Evento que avisará a todos los Scripts necesarios si esto llega a suceder.

Delegates:

Un **delegate** es un tipo de dato que puede almacenar referencias a métodos que cumplen con una firma específica. Pensá en ellos como contenedores o "puentes" para métodos: puedes usar un delegate para invocar un método sin saber exactamente cuál es en tiempo de compilación. Esto los hace muy útiles para ejecutar métodos de manera flexible y dinámica.

- **Firma específica**: Los métodos que un delegate puede almacenar deben coincidir en el tipo de retorno y los parámetros.
- **Seguridad**: A diferencia de punteros a funciones en otros lenguajes como C/C++, los delegates son seguros porque el compilador verifica las firmas de los métodos asociados.
- **Usos comunes en juegos**:
    - Ejecutar diferentes acciones en respuesta a un evento del juego.
    - Configurar lógica personalizada para IA o enemigos.

Ejemplo práctico: un delegate permite ejecutar diferentes funciones a través de una misma llamada. En este caso, usamos un delegate llamado `OnAction` que guarda una función (`Jump`) y luego la ejecuta.

Explicación paso a paso:

Un delegate que ejecuta diferentes comportamientos dependiendo del contexto:

```
public class DelegateExample : MonoBehaviour{
private delegate void ActionDelegate();
private ActionDelegate OnAction;
private void Start(){
    OnAction = Jump; // Asigna el método Jump al delegate
    OnAction(); // Invoca Jump
}
private void Jump(){
    Debug.Log("The player jumps!");
}
}
```

1. Declaración del delegate

```
public delegate void ActionDelegate();
```

- **delegate**: Esta palabra clave define un tipo que puede almacenar referencias a métodos.
- **ActionDelegate**: Es el nombre del tipo de delegate.
- **void y ()**: El delegate está diseñado para almacenar métodos que no tienen parámetros ni devuelven un valor.

En otras palabras:
ActionDelegate es como un "contenedor" que podrá guardar referencias a métodos con la firma void NombreMetodo().

2. Declaración de una variable de tipo delegate

```
public ActionDelegate OnAction;
```

- Aquí se declara una variable pública llamada OnAction que es del tipo ActionDelegate.
- Esta variable puede almacenar métodos cuya firma coincida con la definida por el delegate.

Por ahora, OnAction no tiene ningún método asignado.

3. Método Start

```
private void Start() {
    OnAction = Jump; // Asigna el método Jump al delegate
    OnAction(); // Invoca Jump
}
```

3.1. Asignación del método al delegate

```
OnAction = Jump; // Asigna el método Jump al delegate
```

- **Jump**: Es un método que coincide con la firma de ActionDelegate (no tiene parámetros y devuelve void).
- Esta línea asigna el método Jump a la variable OnAction.
  Ahora, OnAction es una referencia al método Jump.

3.2. Invocación del delegate

```
OnAction(); // Invoca Jump
```

- Aquí se invoca el delegate como si fuera un método.
- Esto llama al método que fue asignado a OnAction, que en este caso es Jump.

4. Método Jump

```
private void Jump(){
    Debug.Log("The player jumps!");
}
```

Este método imprime un mensaje en la consola que dice: "The player jumps!".
Dado que fue asignado al delegate OnAction, se ejecutará cada vez que el delegate sea invocado.

Funcionamiento completo del script:

1. **Inicio del script (Start)**:
    - Se asigna el método Jump al delegate OnAction.
2. **Invocación del delegate**:
    - OnAction() se ejecuta, lo que llama al método Jump.
3. **Resultado**:
    - Aparece en la consola el mensaje: "The player jumps!".

Ventajas

- Podés cambiar el método que se ejecuta en tiempo de ejecución.
- Podés agregar múltiples métodos al mismo delegate:
  `OnAction += Run;`
  `OnAction += Crouch;`
- Fomenta el desacoplamiento entre sistemas: un script no necesita conocer directamente a otros para que reaccionen.

¿Por qué es útil este enfoque?

Este ejemplo puede parecer simple, pero la verdadera utilidad de los delegates radica en su flexibilidad:

1. **Asignar diferentes comportamientos dinámicamente**: Puedes cambiar el método asociado al delegate en tiempo de ejecución.

```
OnAction = Run; // Cambia el comportamiento a otro método llamado Run
```

- **Llamar a múltiples métodos**: Los delegates pueden almacenar referencias a varios métodos al mismo tiempo si usas `+=` para agregar métodos:

```
OnAction += Jump;
OnAction += Run;
OnAction(); // Ejecuta Jump y luego Run
```

- **Desacoplamiento del código**: Permite que diferentes partes del sistema interactúen sin depender directamente unas de otras.

Events:

Un **event** es una extensión de los delegates que se utiliza para comunicar cambios o sucesos. Funcionan como un sistema de notificación: una clase puede "emitir" eventos y otras clases pueden "escuchar" y reaccionar a ellos.

- **Abstracción**: Los eventos se basan en delegates, pero con mayor control sobre quién puede invocarlos.
- **Desacoplamiento**: Los eventos permiten que las clases interactúen sin conocerse directamente. Esto mejora la modularidad y el mantenimiento del código.
- **Restricción**: Solo la clase que declara el evento puede invocarlo. Otros objetos solo pueden suscribirse o desuscribirse.

¿Qué significa esto en la práctica?

Imaginá que el jugador cae por un precipicio:

- Se reproduce una animación de caída.
- Se resta una vida.
- Se actualiza la interfaz (HUD).
- Se reinicia el nivel.

Todo eso puede dispararse desde un solo evento, y cada sistema escucha y actúa según su rol.

Ejemplo simple:
Un evento que notifica cuando un jugador recoge un ítem:

```
public class Player : MonoBehaviour{
    public delegate void ItemCollectedDelegate(string itemName);
    public event ItemCollectedDelegate OnItemCollected;
    public void CollectItem(string itemName){
        Debug.Log($"Collected: {itemName}");
        OnItemCollected?.Invoke(itemName); // Notifica a los suscriptores
    }
}
```

Explicación paso a paso

1. Declaración del delegate:

```
public delegate void ItemCollectedDelegate(string itemName);
```

- **delegate**: Define un nuevo tipo que puede referenciar métodos con una firma específica.
- **ItemCollectedDelegate**: Es el nombre del tipo del delegate.
- **Firma del método**:
    - void: No devuelve ningún valor.
    - (string itemName): Requiere un argumento de tipo string llamado itemName.

💡 **Resumen:** ItemCollectedDelegate puede almacenar métodos que acepten un parámetro de tipo string y no devuelvan nada.

2. Declaración del evento

```
public event ItemCollectedDelegate OnItemCollected;
```

- **event**: Define un evento basado en el delegate ItemCollectedDelegate.
- **OnItemCollected**: Es el nombre del evento.
- **Propósito**:
    - Este evento permitirá a otros scripts suscribirse para ser notificados cuando ocurra una acción (en este caso, cuando el jugador recoja un ítem).

  Nota importante: Un evento es más seguro que usar directamente un delegate, ya que **solo la clase que declara el evento puede invocarlo** (en este caso, solo la clase Player puede ejecutar OnItemCollected). Sin embargo, otros scripts pueden suscribirse y reaccionar cuando el evento se dispare.

3. Método CollectItem.

```
public void CollectItem(string itemName)  {
    Debug.Log($"Collected: {itemName}");
    OnItemCollected?.Invoke(itemName); // Notifica a los suscriptores  }
```

- **OnItemCollected**: Es el evento que hemos definido anteriormente.
- **?.**: Comprueba si el evento tiene suscriptores antes de invocarlo (esto evita errores si no hay nadie suscrito).
- Si nadie está suscrito, no hace nada.
- Si hay métodos suscritos, se ejecutan con el argumento proporcionado.
- **Invoke(itemName)**: Llama a todos los métodos suscritos al evento, pasando el parámetro itemName como argumento.

4. Funcionamiento del código

1. **El jugador recoge un ítem**:
    - Llamar al método CollectItem imprime el nombre del ítem en la consola y dispara el evento OnItemCollected.
2. **El evento notifica a los suscriptores**:
    - Todos los métodos suscritos al evento OnItemCollected se ejecutan con el nombre del ítem (itemName) como parámetro.

Ejemplo extendido: ¿Cómo usar este evento desde otro script?

Desde otro script (por ejemplo, `GameManager`), podemos **suscribirnos** al evento del jugador para reaccionar cuando se recolecte un ítem:

```
private void Start(){
    Player player = FindObjectOfType<Player>();
    player.OnItemCollected += HandleItemCollected; // Suscribirse al evento
}
private void HandleItemCollected(string itemName){
    Debug.Log($"GameManager: Player collected {itemName}");
}
```

- `+= HandleItemCollected` agrega nuestra función como suscriptora al evento.
- Cuando el jugador recoja un ítem, esta función se ejecutará automáticamente, mostrando el nombre del ítem en consola.

Esto hará que cuando el evento “OnItemCollected” sea invocado, este llame todas las funciones inscriptas en él, como en este caso sería “HandleItemCollected”.

Para terminar de entender sus diferencias:

Característica	Delegates	Events
Invocación	Cualquiera puede invocar un delegate si lo tiene como referencia.	Solo la clase que declara el evento puede invocarlo.
Seguridad	Menos restrictivo; otros objetos pueden modificar la referencia.	Más seguro; restringe la invocación solo al propietario.
Uso principal	Ejecutar métodos dinámicamente.	Notificar cambios de estado o sucesos.
Modularidad	Útil, pero puede acoplar clases si no se controla.	Promueve el desacoplamiento entre clases.

Eventos con múltiples receptores

En este ejemplo vamos a implementar una situación clásica de *PickUp*, es decir, **un ítem que el jugador puede recolectar**. Pero a diferencia de versiones anteriores, ahora vamos a incorporar el uso de **eventos** para que, al recoger el objeto, **ocurran múltiples acciones al mismo tiempo**.

El objetivo es que, al tocar el objeto:

- El jugador diga una frase (diálogo).
- Se incremente el puntaje en el `GameManager`.
- El objeto se destruya.
  Todo esto, sin que los scripts estén directamente conectados entre sí. Es decir, cada uno **responde al evento**, pero **no se conocen entre sí**.

Estructura del sistema

Vamos a trabajar con tres scripts:

1. **Clase5PickEvent (el ítem):**
   Es el emisor del evento. Detecta cuándo es recogido y transmite esa información.
2. **Player:**
   Se suscribe al evento para mostrar un diálogo cuando recoge un objeto.
3. **GameManager:**
   También se suscribe al evento para actualizar una variable de puntaje.

1) Script del ítem – `Clase5PickEvent`

```
private string name1;
private int value = 5;
private delegate void Format1(string item, int value);
public static event Format1 myPickEvent;
void Start() {
    name1 = gameObject.name;
}
private void OnTriggerEnter(Collider other)  {
    if (other.CompareTag("Player")) {
          FuiAgarrado(name1);
    }
}
private void FuiAgarrado(string a){
  if (myPickEvent != null){
    myPickEvent.Invoke(name1, value);
    Debug.Log($"Soy {name1} y me agarraron");
    Destroy(gameObject, 0.1f);
   }}
```

¿Qué hace este script?

- Declara un `delegate` llamado `Format1` que acepta un `string` y un `int`.
- Declara un `event` llamado `myPickEvent` basado en ese delegate.
- En `Start()`, guarda el nombre del objeto.
- Cuando detecta una colisión con el `Player` (`OnTriggerEnter`), ejecuta el método `FuiAgarrado()`.
- Si hay suscriptores al evento, lo invoca enviando el nombre del ítem y su valor, y luego se destruye.

💡 **Importante:** el evento es `static` para que pueda ser accedido desde cualquier otro script, incluso si el objeto se destruye.

2) Script del jugador – Player

```
void Start() {
    Clase5PickEvent.myPickEvent += PickedUp;
}
private void PickedUp(string item, int value) {
    Debug.Log($"Agarre {item} chaval. Junte {value} de oro");
}
```

¿Qué hace este script?

- En el método `Start()`, el jugador **se suscribe** al evento `myPickEvent`.
- Cuando el evento se dispara, ejecuta el método `PickedUp`, que muestra en consola un mensaje simulando un diálogo.

Este script **no necesita saber nada del objeto recogido**. Simplemente espera que el evento ocurra y reacciona.

3) Script de GameManager

```
void Start() {
    Clase5PickEvent.myPickEvent += AddScore;
}
private void AddScore(string item, int value) {
    score += value;
    Debug.Log($"Mi puntaje ahora es de {score}");
}
```

¿Qué hace este script?

- También se **suscribe** al evento `myPickEvent` en `Start()`.
- Su método `AddScore` recibe los mismos parámetros (`item`, `value`) y los utiliza para **sumar puntos al sistema de juego**.

💡 Este script debe colocarse en un **EmptyObject** en la escena, y su función es empezar a centralizar lógicas como el puntaje, la gestión de misiones o las condiciones de victoria.

¿Qué logramos con esto?

Al usar eventos:

- **Un solo evento** (el del ítem) puede activar **múltiples respuestas**.
- Los scripts **no están acoplados entre sí**.
- Podemos escalar fácilmente: agregar una respuesta más (como actualizar una UI) **sin tocar el código del ítem**.

Materiales y recursos adicionales.

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

Otro día en TalentoLab:

El cliente está cada vez más emocionado con los avances de Nexus, pero han identificado un problema crítico: los eventos dentro del juego carecen de interconexión. Las acciones del jugador no desencadenan reacciones significativas en el mundo del juego. Por ejemplo, al recolectar monedas o alcanzar un objetivo, no hay un sistema que informe al resto del juego sobre lo que sucede.

Para resolver esto, el cliente quiere que TalentoLab implemente un sistema de **Events** y **Delegates**, una herramienta poderosa para conectar los elementos del juego de manera eficiente. Este sistema será clave para asegurar que las acciones del jugador tengan un impacto tangible y para garantizar que el juego se sienta cohesionado.

Ejercicios prácticos:

La solicitud del cliente:
El cliente necesita ver ejemplos claros de cómo los eventos del juego se comunican entre sí, cómo recolectar objetos, desbloquear zonas o responder a la derrota del jugador. ¡Es momento de convertir Nexus en un mundo lleno de reacciones dinámicas!

(*Tengan en cuenta que esta situación será resuelta entre esta clase y la siguiente*)

¡Roberta puede ver tu gran potencial! Así que te asignará la siguiente tarea para probar tus capacidades!

1) Sistema de "Game Over" con Eventos:
En Nexus, la muerte del personaje debe ser un momento dramático que refuerce la conexión emocional del jugador con el mundo del juego. TalentoLab quiere que al morir el personaje, se active un evento que:

1. Mostrar una **pantalla de "Game Over"** que detenga el flujo del juego.
    - *Tip:* Podés usar un `GameObject` tipo "pantalla" que esté desactivado y lo actives con `.SetActive(true)` al morir.
    - Alternativa: usar una escena de Game Over cargada con `SceneManager.LoadScene()`.

2. Reproducir **efectos visuales o de sonido** para dar dramatismo al momento.
    - *Tip:* Podés usar un `ParticleSystem`, animación o `AudioSource.Play()`.

Consigna técnica:

- Usá un **evento público** (`event`) que se dispare desde el script del `Player` al detectar la muerte.
- Hacelo **desencadenar las respuestas desde otros scripts**: uno para el UI (pantalla), otro para efectos.

2. Desbloqueo de zona secreta por recolección de monedas

En *Nexus*, TalentoLab quiere fomentar la exploración. El desafío es crear una **zona secreta** que solo se desbloquea si el jugador demuestra habilidad recolectando monedas.

Acciones requeridas:

1. **Teletransportar al jugador** a una zona secreta del mapa.
    - *Tip:* usá `player.transform.position = lugarSecreto.transform.position;`
2. (Opcional) Activar un efecto visual o sonoro al llegar.

Consigna técnica:

- Definí un **evento en el sistema de monedas** que se dispare al alcanzar el umbral deseado (ej: 10 monedas).
- Desde un script separado, suscribite a ese evento y realizá el cambio de posición.

Buenos Aires aprende

Agencia de Habilidades para el Futuro

Buenos Aires Ciudad