«Talento Tech»

Desarrollo de Videojuegos

Unity 2D

Clase 13

«Talento Tech»

Buenos Aires Ciudad

Clase N° 13 | Encapsulamiento

Temario:

- Get Set
- Encapsulamiento

Get Set.

El **get** y **set** se utilizan en las propiedades para controlar el acceso a los **campos** de una clase, permitiendo leer (obtener) o escribir (establecer) sus valores de forma controlada. **Las propiedades** con get y set son una forma segura y conveniente de exponer datos de una clase sin exponer directamente sus campos privados.

¿Que es un Campo?

Es una variable que pertenece a una clase o estructura y almacena datos relacionados con la instancia de esa clase (si es un campo de instancia) o con la clase en general (si es un campo estático).

Vamos a ampliar algunos conceptos vistos en clases anteriores…
Dentro de los Campos, hay 2 tipos:

- **Campos de instancia**: Son específicos de cada instancia de la clase. Cada objeto creado a partir de la clase tiene su propia copia de estos campos. Se declaran sin el modificador static.

```
public class Persona
{
    public string nombre;  // Campo de instancia
    public int edad;       // Campo de instancia
}
```

En este ejemplo, cada objeto **Persona** tendrá su propio **nombre y edad**.

- **Campos estáticos:** Son compartidos por todas las instancias de la clase y pertenecen a la clase en sí. Se declaran con el modificador static.

```
public class Persona
{
     public static int poblacion;   // Campo estático compartido por
todas las instancias
}
```

Acá, “población” es un campo compartido entre todas las instancias de Person.

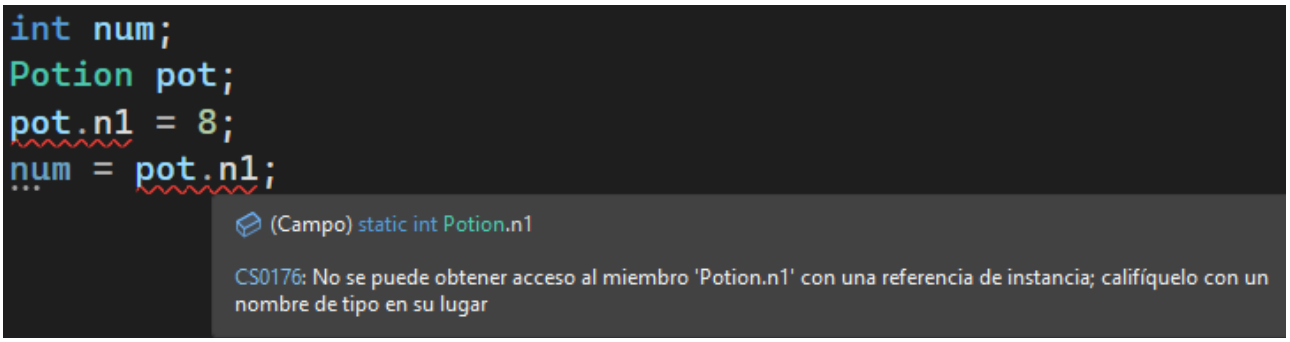
Pueden probar que, aunque coloquemos **[SerializeField]** o “**Public**”**,** esta no aparecerá en el inspector para ser modificada. En este caso “Public” SOLO nos permitirá acceder al campo desde otra Class únicamente si llamamos a la Class dueña del campo, en vez de su Instancia.

Ejemplos:

En una una Class genero un campo/variable **static:**

```
public class Potion : MonoBehaviour
{
    public static int n1 = 5;
```

Ahora intentaré llamarla desde otra Class, de la manera que veníamos trabajando:



Como podrán ver, no podemos acceder a la variable desde una instancia. Entonces lo haremos llamando desde la Class como tal:

```
int num;
Potion.n1 = 8;
num = Potion.n1;
```

```
int num;
Potion.n1 = 8;
num = Potion.n1;
```

Si la variable la hacemos **private** no podremos acceder a ella desde otra Class de ninguna forma.

Aunque los campos almacenan datos, es recomendable acceder a ellos mediante propiedades (usando get y set), en lugar de hacer los campos públicos. Las **propiedades**

permiten agregar validación o lógica de acceso, manteniendo el principio de **encapsulamiento** que terminaremos de definir más adelante.

¿Qué es un get y un set?

Get: Se usa para definir lo que sucede al leer la propiedad. Devuelve el valor del campo subyacente o calcula un valor a partir de otros datos en la clase.

Set: Se usa para definir lo que sucede al asignar un valor a la propiedad. Suele establecer el valor del campo subyacente y puede incluir lógica adicional para validaciones o transformaciones de datos.

Ejemplo Básico:

```
public class Player
{
    private int _health;

    public int Health
    {// al usar la variable, nos da el valor de _health(get)
        get { return _health; }
        set
        {// Si el valor asignado a Health es menor a 0, asigna 0 a
_health
            if (value < 0) // Validación en el set
                _health = 0;
            else// Si no es menor, le asigna el valor pasado.
                _health = value;
        }
    }
}
```

Obtener el valor (get): player.Health devolverá el valor actual de health.
Establecer el valor (set): player.Health = 10 asignará 10 a health, pero antes pasará por la validación en set, asegurándose de que no sea un valor negativo.

Primero, veremos que nuestra variable “Health” se abre utilizando las llaves “{}”. A esto se le llama **propiedades,** lo que nos permite ingresar circunstancias de control para los datos recibidos utilizando el **get;set**

Encapsulamiento.

El encapsulamiento en C# es un principio de la programación orientada a objetos (OOP) que consiste en ocultar los detalles internos de una clase (su implementación) y exponer sólo lo necesario mediante una interfaz pública controlada. Esto se logra a través del uso de modificadores de acceso como private, public, protected, entre otros, y mediante el uso de propiedades para controlar el acceso a los datos de una clase.

Ejemplo de uso.

Primero veamos un codigo simple:

```
        public class Player : MonoBehaviour
{
    public int health; // Campo público (no encapsulado)

    void Start()
    {
        health = 100; // Inicializamos la salud del jugador
    }

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            health -= 10; // Cualquier script puede modificar la salud
            Debug.Log("Salud actual: " + health);
        }
    }
}
```

Problemas de este enfoque:

- Cualquier otro script puede modificar health directamente, incluso con valores inválidos (por ejemplo, negativos o mayores al máximo permitido).

- No hay control sobre cómo se cambia el valor.

- Es fácil introducir errores o romper las reglas del juego.

Utilizando encapsulamiento seria algo asi:

```
public class Player : MonoBehaviour
{
  // Campo privado, solo accesible dentro de esta clase
  private int _health;
// Propiedad pública para acceder al campo privado de manera controlada
  public int Health {
   get { return _health; } // Permite leer el valor de health
 private set{// Restringe la modificación directa desde fuera de la
clase
  // Validación para asegurarnos de que health nunca sea menor a 0 o
mayor a 100
     _health = Mathf.Clamp(value, 0, 100);
        } }
    void Start() {
        Health = 100; // Inicializamos la salud del jugador
    }
    // Función pública para aplicar daño
   public void TakeDamage(int damage) {
     if (damage > 0){
       Health -= damage;
       Debug.Log("El jugador recibió daño. Salud actual: " + Health);
        }
    }
  public void Heal(int amount){// Función pública para curar al jugador
    if (amount > 0){
      Health += amount;
      Debug.Log("El jugador fue curado. Salud actual: " + Health);
        } } }
```

Verán entonces que casi todo lo que estuvimos viendo hasta ahora en las clases, nos servirá para entender y usar formas de Encapsulamiento. En este caso no solo aplicamos **propiedades**, sino también funciones que nos ayudaran con el manejo de los valores dependiendo de lo que queramos hacer (Sumar o Restar vida).

Formas de utilizar esto hay muchas. Podríamos haber llamado algún “Debug.Log()”, dentro del mismo set:

```
public int Health
        {
```

```
            get { return health; } // Permite leer el valor de health
                private set // Restringe la modificación directa desde
fuera de la clase
            {
                  // Validación para asegurarnos de que health nunca sea
menor a 0 o mayor a 100
                health = Mathf.Clamp(value, 0, 100);
                Debug.Log("Seteando health");
            }
        }
```

También, en este caso, si bien el **get** es **public**, queriendo decir que podemos ver el valor de la variable desde cualquier otra Class. Fijense que el **set**, es **private**. Dándonos SOLAMENTE la oportunidad de hacerlo, mediante las funciones "**Heal**" y "**TakeDamage**", a la vez, siendo limitados por el **Mathf.Clamp(value, 0, 100),** obligando a mi variable a permanecer dentro del rango numérico de 0-100, sin necesidad de colocar un **if** para preguntar su valor y volver a setearlo en 0 o 100.

Si quisiéramos asignar un valor a la variable _health desde otra Class, tendríamos entonces que llamar las funciones mediante una variable referencia. Algo similar a esto:

```
 public Player player; // Arrastra el objeto con el script Player en el
inspector

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            player.TakeDamage(10); // Reduce la salud del jugador en 10
        }

        if (Input.GetKeyDown(KeyCode.H))
        {
            player.Heal(20); // Cura al jugador en 20
        }
    }
```

Ejemplo de uso más complejo:

Haremos un código que nos permita ganar experiencia, nos haga subir de nivel y al hacerlo, reste la experiencia sobrante al siguiente nivel:

```
 // Campos privados
    private int experience; // Experiencia actual
    private int level;      // Nivel del jugador

    // Propiedades públicas para controlar el acceso
    public int Experience
    {
        get { return experience; } // Permite leer el valor de
experiencia
        private set // Restringimos que se pueda modificar directamente
desde fuera
        {
            experience = value;

            // Subir de nivel si la experiencia alcanza el umbral
            if (experience >= ExperienceToLevelUp())
            {
                experience -= ExperienceToLevelUp(); // Restamos el XP
necesario para subir de nivel
                LevelUp();
            }
        }
    }

    public int Level
    {
        get { return level; } // Permite leer el nivel
        private set { level = Mathf.Max(1, value); } // Asegura que el
nivel nunca sea menor a 1
    }

    // Función para inicializar valores
    private void Start()
    {
        Level = 1;
        Experience = 0;
    }
```

```
    // Función público para ganar experiencia
    public void GainExperience(int amount)
    {
        if (amount > 0)
        {
            Experience += amount;
            Debug.Log($"Ganaste {amount} de experiencia. Experiencia
actual: {Experience}, Nivel: {Level}");
        }
    }

    // Función privada para manejar la subida de nivel
    private void LevelUp()
    {
        Level++;
        Debug.Log($"¡Subiste al nivel {Level}!");
    }

    // Función privada para calcular la experiencia necesaria para
subir de nivel
    private int ExperienceToLevelUp()
    {
        return Level * 100; // Por ejemplo: XP necesaria = Nivel * 100
    }
```

Explicación del Ejemplo

1. **Campos privados (`experience` y `level`)**:
   - Estos datos están **ocultos** para otros scripts, lo que asegura que no puedan modificarse directamente.
2. **Propiedades públicas**:
   - `Experience`:
     - Se puede **leer** desde otros scripts (`get`).
     - Solo se puede **modificar desde dentro de la clase** (`private set`).
     - Contiene la lógica para verificar si el jugador ha acumulado suficiente experiencia para subir de nivel.
   - `Level`:
     - Se puede **leer** pero solo modificarse desde dentro de la clase.
     - Aseguramos que nunca sea menor a 1.
3. **Función pública `GainExperience()`**:
   - Permite que otros scripts otorguen experiencia al jugador.
   - Esto asegura que la experiencia solo se modifique bajo las reglas definidas.

4. **Subida de nivel**:
    - Cuando el jugador acumula suficiente experiencia, se llama a la función `LevelUp()`.
    - Restamos la experiencia necesaria para subir de nivel y subimos al jugador al siguiente nivel.
5. **Cálculo de experiencia necesaria**:
    - La experiencia requerida para subir de nivel aumenta con cada nivel, calculada con la función `ExperienceToLevelUp()`.

Explicación del recorrido del código:

Básicamente, nuestro código lo que hará es ir llamando a "**GainExperience()**",

```
    public void GainExperience(int amount)
    {
        if (amount > 0)
        {
            Experience += amount;
            Debug.Log($"Ganaste {amount} de experiencia. Experiencia
actual: {Experience}, Nivel: {Level}");
        }
```

yendo constantemente a la propiedad **set** de **Experiencie:**

```
 private set // Restringimos que se pueda modificar directamente desde
fuera
        {
            experience = value;

            // Subir de nivel si la experiencia alcanza el umbral
            if (experience >= ExperienceToLevelUp())
            {
                experience -= ExperienceToLevelUp(); // Restamos el XP
necesario para subir de nivel
                LevelUp();
            }
        }
    }
```

Esto hará que chequee si `experience >= ExperienceToLevelUp()`, función que devolverá “Level*100”. Es decir, si tenemos nivel1 será 100, si tenemos nivel2 será 200, etc.
Al acumular la experiencia necesaria, primero **restará lo consumido** para subir de nivel. Supongamos que **derrotó a un NPC** que me da **130xp,** siendo Nivel1 hará **130-100** dejando solo 30xp. Luego de esta cuentita, pasará a llamar a la función **LevelUp()**

```
    private void LevelUp()
    {
        Level++;
        Debug.Log($"¡Subiste al nivel {Level}!");
    }
```

Es muy importante el **timing**. Tengan en cuenta que debido al orden, todo esto sucederá mientras se cambia el valor de exp,es decir en esta línea:

```
     Experience += amount;
```

Por lo tanto, recién de hacer todo ese hermoso cambio, es que llamara al Debug.Log debajo de esta línea de código, dentro de la misma función:

```
          public void GainExperience(int amount)
    {
        if (amount > 0)
        {
            Experience += amount;
            Debug.Log($"Ganaste {amount} de experiencia.
Experiencia actual: {Experience}, Nivel: {Level}");
        }
```

El encapsulamiento es una metodología de trabajo **necesaria** para cuando se trabaja en grupo o tenemos proyectos largos que no recordamos al 100% su manejo. Su conocimiento es **solicitado** en cualquier puesto de trabajo de programador. Ahora parece mucho código para terminar haciendo lo mismo, pero con el tiempo verán sus grandes ventajas. La práctica hará la costumbre y de ella vendrá el progreso de sus habilidades. **¡Sigan Leveleando!**

Ejercicios prácticos:

> Selecciona un código ya hecho o crea uno nuevo para **encapsular** algunas variables con un **get set**. Elegí las que quieras, pero por ejemplo, podrías empezar con las variables más importantes como la vida, oro, mana, daño, etc.

Cuanto más practiques, más te podrás familiarizar con los conceptos y la práctica, lo que te permitirá incorporarlo con mayor facilidad en tus códigos.


Buenos Aires
aprende
Agencia de Habilidades para el Futuro
Buenos
Aires
Ciudad