«Talento Tech»

Desarrollo de Videojuegos

Unity 2D

Clase 12

«Talento Tech»

Buenos Aires Ciudad

Clase N° 12 | POO

Temario:

- Clases
- Herencia

Poo.

En esta clase, empezaremos a ver la programación orientada a objetos con un poco más de detalle. Si bien ya venimos usando varias de sus características hoy profundizaremos en algunas de sus ventajas. Pero primero, recordemos ¿Qué es el **POO**?

La **Programación Orientada a Objetos** es un paradigma de programación basado en el concepto de "objetos", que son instancias de clases. En lugar de centrarse solo en funciones o procedimientos, como en la programación estructurada, la POO organiza el software en torno a entidades que combinan **datos** (atributos) y **comportamientos** (métodos o funciones).

Los pilares fundamentales de la POO:

1. **Clases**:
    - Una clase es una plantilla o un plano para crear objetos. Define los atributos (propiedades) y los métodos (funciones) que los objetos creados a partir de esa clase tendrán.
    - **Ejemplo**: Si tenés una clase Coche, podría tener atributos como color, marca y velocidad, y métodos como acelerar() o frenar().
2. **Objetos**:
    - Un objeto es una instancia concreta de una clase. Cada objeto puede tener valores específicos para sus atributos.
    - **Ejemplo**: A partir de la clase Coche, puedes crear el objeto miCoche con un color rojo y una velocidad 0, y el objeto tuCoche con un color azul y una velocidad 20.
3. **Encapsulamiento**:
    - El encapsulamiento es el concepto de ocultar los detalles internos de la implementación de un objeto y exponer solo lo necesario a través de métodos públicos. Esto ayuda a proteger los datos y asegura que no puedan ser modificados directamente desde fuera de la clase.

- **Ejemplo**: Los atributos de un objeto pueden ser privados, y se puede proporcionar un método público para acceder a ellos o modificarlos de manera controlada (getter y setter).

4. **Herencia**:
   - La herencia permite crear nuevas clases basadas en las clases existentes. Una clase hija puede heredar atributos y métodos de una clase padre, lo que facilita la reutilización del código.
   - **Ejemplo**: Si tenes una clase Animal, podrías crear una clase Perro que herede las características de Animal, pero también agregarle características adicionales, como ladrar().
5. **Polimorfismo**:
   - El polimorfismo permite que objetos de diferentes clases sean tratados como objetos de una clase común. A través de la herencia, puedes usar métodos de manera uniforme en diferentes tipos de objetos, aunque cada tipo de objeto pueda tener su propia implementación de estos métodos.
   - **Ejemplo**: Si tenes una clase Animal con un método hacerSonido(), tanto un Perro como un Gato pueden implementar este método de manera diferente (el perro puede ladrar y el gato maullar), pero ambos pueden ser llamados de la misma manera: animal.hacerSonido().

Varias de estas características como las Clases y los Objetos ya lo venimos trabajando, así que hoy entraremos en los conceptos de Herencia y Polimorfismo. Dejaremos el encapsulamiento para una de las clases siguientes.

Herencia.

Como se mencionó antes, la herencia es la capacidad de recibir características de otra clase. Llamaremos Clase “Padre”(Parent) a la plantilla base y Clase “Hijo”(Child), a la que heredara sus propiedades.
Veamos cómo se crean…

Creación.

Anteriormente, nosotros teníamos clases similares a esta:

```
public class Hero : MonoBehaviour
{
```

Nuestra clase “Hero” podría poseer todo lo necesario para nuestro personaje, como el ataque, curación, daño recibido, movimiento, animaciones y todas las variables necesarias. Y acá hay 2 cosas a notar.

1) **¿Por qué dice “MonoBehaviour” al lado?** Esto es porque nuestra clase, justamente HEREDA de MonoBehaviour. Una Class general de Unity que nos permite utilizar todas las funciones propias del Motor Gráfico.

2) **¿Qué pasaría si deseamos hacer más personajes y no solo aquellos jugables? Cada personaje no tendría sus animaciones, Movimiento, vida, poder, mana, etc. ¿Tendríamos que codear todo nuevamente para cada tipo de Player o NPC?**
   La respuesta es **NO**. Gracias a la Herencia y el Polimorfismo podremos ahorrar horas de código y hasta conseguir un mejor orden.

Para hacer que nuestra Class herede de otra, bastará con llamarla luego de los 2 puntos, al igual que figura más Arriba. Por ejemplo, suponiendo que haya creado una Class llamada “Character” para guardar todo el comportamiento principal de Cualquier personaje, esto se veria asi:

```
public class Hero : Character
{
```

Entonces quiere decir que mi Class “Hero” ya no hereda de “Monobehaviour”? NO, lo seguirá haciendo, pero de manera indirecta. Porque al crear la Class “Character”, está si heredara de Monobehaviour:

```
public class Character : MonoBehaviour
{
```

Sigamos con este ejemplo y empecemos a armar nuestra clase Character.

Pensando el “Parent Character”.

Si tuviéramos que hacer una Class general para que las demás hereden, ¿qué debería tener?

Para empezar, enumeramos algunos de los datos y comportamientos necesarios. Primero sus variables más básicas

```
    [SerializeField] private float power;
   [SerializeField] private float vida;
    [SerializeField] private float speed;
```

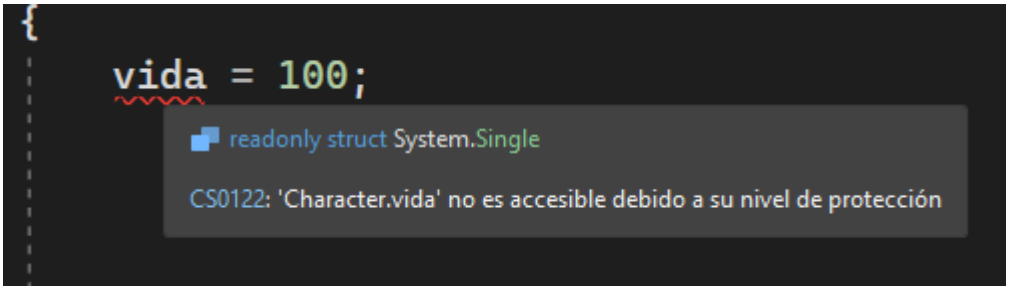
Bien, ahora crearemos una Class cualquiera que herede de esta, seguiremos con el ejemplo de Hero.

```
public class Hero : Character
{
```

Ya que Hero hereda de Character, llamaremos a la variable “vida” para asignarle un valor en el Start();

```
void Start()
    {
        vida = 100;
    }
```

Veremos rápidamente que nos saldrá un error



Si bien, reconoce la variable “vida” de character, me dice que no podemos acceder a ella. Esto es porque es **private**. ¿Pero entonces, deberíamos de hacerla **public**?

Si hacemos eso, TODAS las otras Class tendrán acceso a nuestra variable y eso, podría no ser muy bueno. Así que, para estos casos, tenemos un concepto nuevo. **El Protected.**

Protected.

Si en nuestra **Class Parent** llamada Character, le cambiamos el **nivel de protección** a nuestras variables, notaremos que el error desaparece.

```
    [SerializeField] protected float power;
   [SerializeField] protected float vida;
    [SerializeField] protected float speed;
```

Esto se debe a que Protected, es un nivel intermedio que permite SOLAMENTE manejar el dato o función a la **Class creadora** y sus **Hijos.**
Así podremos reasignar o utilizar cualquier elemento creado en “Character” desde nuestro “Hero”.

Funciones.

Crearemos una función en nuestra Parent Class. Algo sencillo como un “Heal”

```
    public void Heal(float a){
        vida += a;
        if (vida > maxHP){
            vida = maxHP;
        }
    }
```

Aparte de curarnos, le pondremos un **if** sencillo que tendrá la utilidad de **normalizar nuestra vida**, así evitamos que se acumule demasiado.

Como nuestra **Class Hero** hereda de “Character”, tendrá acceso a la misma función sin necesidad de volver a crearla en ella.
Para comprobarlo, será suficiente con hacer un Script sencillo con una variable de referencia a Hero e intentar llamar a la función para curar a mi personaje.

```
public class Potion : MonoBehaviour{
     Hero p;
    void Start(){
        if (p == null){
            p = GameObject.Find("Warrior").GetComponent<Hero>();
            p.Heal(10);
```

```
        }
    }
```

Creamos la variable y en el Start preguntamos, Si mi **variable p** es **nula**, es decir, no tiene valor, utilizamos el “**GameObject.Find()**”, para buscar al Objeto “Warrior” (**Mi personaje**) en la escena, obtener su componente “**Hero**” y terminar por usar la función “Heal” para curarlo 10 de vida.

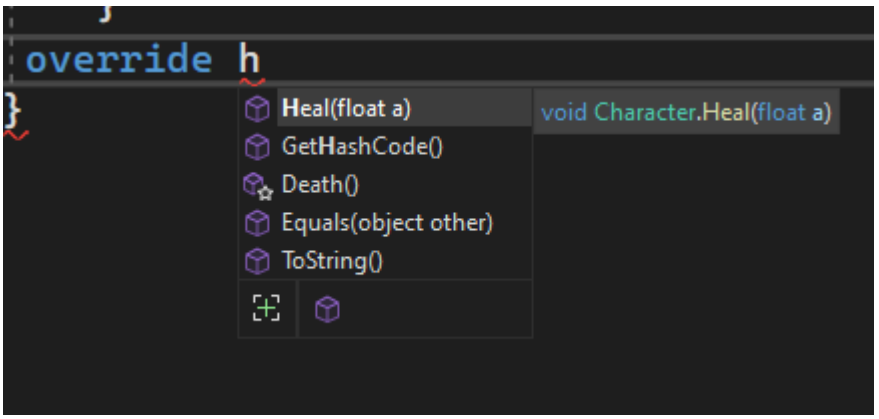
Override.

¿Ahora qué pasaría si la función Heal, no fuera suficiente. La clase anterior creamos una barra de vida y esta función no contempla el acto de cambiarle el valor a la HealthBar. ¿Deberíamos de crear otra función y también llamarla abajo del “p.Heal(10);”? No hace falta. Para estos casos existe el comando “**Override**” que nos permitirá “Sobreescribir” una función del Parent en el Child. Podremos sumarle más código o simplemente hacer algo completamente distinto. Es importante entender que esto **solo sucederá para el Child** y no afectará en nada a nuestro **Parent**.

Para empezar, debemos hacer que **nuestra función** tenga la capacidad de ser **sobreescrita**. Y para esto le daremos la propiedad de “**Virtual**”, quedando de la siguiente manera

```
    public virtual void Heal(float a){
        vida += a;
        if (vida > maxHP){
            vida = maxHP;
        }
    }
```

Ahora si, desde nuestro Child “Hero”, empezaremos tipeando “override” y nos dejará elegir la función que queramos. En este caso “Heal()”



Al AutoCompletar quedará de la siguiente manera:

```
    public override void Heal(float a)
{
     base.Heal(a);
}
```

El “`base.Heal(a);`” indica el uso de la función Heal() del Parent. “Base” es otra forma de referirse a la Parent Class. Acá nosotros podremos utilizar el código de esa función o descartarlos:

```
    public override void Heal(float a)
{
}
```

Todo depende de lo que queramos hacer.

En mi caso, lo dejaré puesto, porque quiero que mi función Heal(), me siga curando de esta manera. Pero le añadiré una línea de código que usamos la clase anterior para darle valor a nuestra HealthBar:

```
    public override void Heal(float a)
{
     base.Heal(a);
     healthBar.fillAmount = (vida / 100);
}
```

Si no recuerdan de dónde sale esta línea de código, pueden revisar la clase 11. O por razones de testeo, bastará con reemplazarlo con un “Debug.Log()”

Ya con esto colocado, nuestra función SobreEscrita, no solo nos curará, sino que representará este valor en nuestra Barra de Vida.

Ejercicios prácticos:

Crear mínimamente una Class parent que puedan Heredar a más de 1 Class. Por ejemplo:

1) La Class Character que será parent de Varios de nuestros personajes (NPCs y/o Players)

2) Una Class “Item” que tendrá las características básicas de cada ítem Agarrable/PickUp y será parent de distintos ítems que podamos crear.

Buenos Aires aprende

Agencia de Habilidades para el Futuro

Buenos Aires Ciudad