# Unity Clases

Propiedades: Para acceder a un campo de una clase se pueden usar las propiedades. La utilidad de esto es que las variables publicas se pueden cambiar sin que el objeto tenga constancia, sin embargo con la propiedad es mejor:

public class Player{

private int experience;

public int Experience {

get{

return experience

}

set{

experience = value

}

}

}

Ahora se llama a Experience como si fuera la variable.

Int a = Player.Experience

Player.Experience = 1

Contructores: Siempre llevan el mismo nombre de la clase y se pueden declarar los que se quieran. Si se teckea ctro automáticamente te crea el constructor.

Ej: public Enemy(int vida, int nivel){

Life = vida;

Level = nivel;

}

Generics: Sirve para pasar un tipo genérico a una función o clase. Por ejemplo se le puede pasar o un RigidBody o un Collider, se usa el Generic para poder pasarle los dos.

Definicion: public T name<T>(T param){}

Además se le puede poner filtrar los valores que puede recoger añadiendo una constraint a la definición. Si se quiere añadir mas de una se separan por comas.

Definicion de referencia: public T name<T>(T param) where T : class{}

Definicion de valor: public T name<T>(T param) where T : struct{}

Definicion constructor sin parametros: public T name<T>(T param) where T : new(){}

Definicion de clase o derivada de clase: public T name<T>(T param) where T : nameClass{}

Como mencionamos, se puede usar también en clases.

Definicion: public class name<T>{}

Ejemplos:

public T Add<T>(T param1, T param2){

return param1+param2

}

string suma = add<string>(“Hola”, “Mundo”)

int suma = add<int>(1, 1)

public class Item<T>{

T item;

public void UpdateItem(T newItem){

item = newitem;

}

}

Item<int> myItem = new Item<int>();

myItem.UpdateItem<int<(5)

Overloading: Sirve para poder usar el mismo nombre en diferentes metodos en una clase. La única condición que debe de cumplir es que se deben de llamar igual y que los parámetros que se le pasen han de ser distintos.

Ej: public int Add(int para1, int param2){

return param1 + param2

}

Public string Add(string para1, string param2){

return param1 + param2

}

Herencia: Cuando una clase hereda de otra, obtiene todas las características de la clase que hereda. A esa clase se le llama clase padre y a la que hereda clase hija. Lo único que no se hereda son los constructores. Solo se puede heredar de una clase.

Declaracion: public class name: parentname{}

Ej: public class Bat : Enemy{}

Polimorfismo: Es una caracteristica de la herencia que permite a una clase tener mas de un tipo. Todas las clases hijos pueden ser clases padre. Cuando se necesita la clase padre, se puede usar la clase hija en su lugar. Ademas, un método en la clase padre, que se ha sobrescrito en la clase hijo, se pude usar desde la clase padre con el polimorfismo.

Ej: Enemy myEnemy = new Bat()

Member Hiding: En la herencia, los miembros de la clase padre estan automaticamente disponibles en la clase hijo. El member hiding es redeclarar los miembros de la clase padre en la clase hija. Se debe poner new delante del miembro. Es lo contrario al Overriding.

Ej: public class Enemy{

Int health = 100;

void SaySomething(string param1){Debug.Log(param1);}

}

public class Bat : Enemy{

new int Health = 500;

new void SaySomething(string param1){Debug.Log(“Hola, “ + param1};}

}

Overriding: Es cambiar un metedo de la clase padre en la clase hijo. Ademas se puede hacer que haga la funcionalidad del método padre mas lo añadido en el override. Con el polimorfismo, si se llama al método, se ejecutara el del hijo.

Definicion:

Clase Padre: public virtual void Speak(){}

Clase Hija: public override void Speak(){}

Usar metodo padre + funcionalidad:

Clase Hija: public override void Speak(){

Base.Speak();

}

Interfaces: Cualquier clase que implementa una interface, debe tener todos sus métodos y propiedades. No son clases y no pueden tener sus propias instancias. Se pueden implementar varias interfaces separadas por comas.

Declaracion:

public interface IKillable{

void Kill();

}

Ejemplo2:

public interface IDamageable<T>{

void Damage(T damageTaken);

}

public class Avatar : MonoBehavour, IKillable, IDamageable<float>{

public void Kill(){}

public void Damage(float damageTaken){}

}

Extension Methods: Sirve para crear metodos extra en una clase. Sobretodo son útiles cuando queremos añadir una funcionalidad a una clase pero no podemos editar esa clase. Siempre debe de ser estatica la función y la clase. El tipo de la función indica a que clase pertenecerá.

Ejemplo añadir funciona a la clase Transform:

public static class ExtensionMethods{

public static void ResetTransformation(this Transform trans){

trans.position = Vector3.zero;

trans.localRotation = Quaternion.identity;

}

}

Transform.ResetTransformation();

Namespaces: Son como contenedores de clases. Su objetivo es ayudar a organizar los scripts y prevenir conflictos entre scripts.

Declaracion:

namespace SampleNamespace{

class

}

Despues con using SampleNamespace se puede acceder a el desde otro script.

Delegates: Sirve para almacenar funciones y llamarlas desde la misma variable ambas.

delegate void myDelegate(int num)

myDelegate mydelegate;

void Start(){

mydelegate = Suma

mydelegate(5)

mydelegate = Multi

mydelegate(5)

}

void Suma(int num){

Debug.Log(“Suma = “ + (num+num);

}

void Multi(int num){

Debug.Log(“Multiplicacion = “ + (num\*num);

}

Sigleton: Nos permite llamar a métodos y acceder a variables de un componente desde cualquier script sin tener que definir una variable dentro de la clase de ese script. No se pueden tener dos objetos con el mismo script si tiene el sigleton.

Pasos:

Se define una variable estatica publica del tipo de la clase:

public static GameController instance;

En el Awake se rellena la variable con el objeto y se mira si tiene alguna referencia y se eliminan duplicidades:

void Awake(){

if (GameController.instance == null){

GameController.instance = this;

}else if(GameController.instance != this){

Destroy(gameObject);

}

}

Por último se crea el método OnDestroy para ponerse a null. Util cuando, por ejemplo, se cambia de escena.

void OnDestroy(){

if(GameController.instance == this){

GameController.instance = null;

}

}

## HASHCODE

Es un código único que devuelve cualquier objeto, variable etc. Es único, si una variable tiene un valor de 1, su HashCode siempre será el mismo.

Ejemplo:

int a = 1.5845

Debug.Log(a.GetHashCode());

Esto devolverá siempre 1070256357