



UD2.2 Introducción a C#

2º CFGS
Desarrollo de Aplicaciones Multiplataforma
2024-25

1.- Scripting

Como toda aplicación informática, los juegos necesitan que se codifique su funcionalidad.

En Unity **la codificación de la funcionalidad** se realiza mediante los **scripts**.

Mediante los scripts se podrá:

- Responder a las entradas del jugador.
- Ejecución eventos en el momento adecuado.
- Crear efectos gráficos.
- Controlar el comportamiento físico de los GameObjects.
- Implementar un sistema de IA para los NPC del juego.
- ...

1.- Scripting

Actualmente el único lenguaje de programación recomendado para realizar proyectos con Unity es **C#**.

También hay soporte para el lenguaje **UnityScript** que se diseñó a partir de JavaScript específicamente para Unity, pero se desaconseja su uso. Es posible que algún proyecto actual aún tenga scripts con UnityScript.

Antiguamente también se podría utilizar el lenguaje **Boo** que tenía una sintaxis similar a Python. Hoy en día no tiene soporte.

1.- Scripting

C# fue creado en el año 2000 para ser compatible con la plataforma .NET de Microsoft.

Hereda lo mejor de C++, Java y Visual Basic.

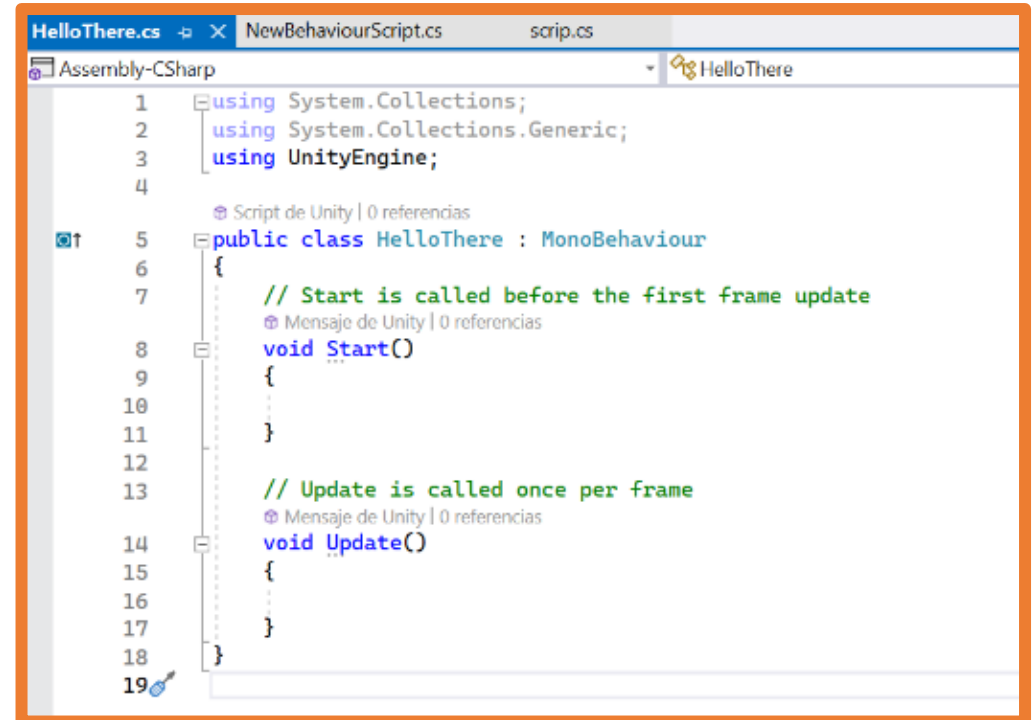
Es un lenguaje orientado a objetos nativo.

Gracias a la librería IL2CPP los scripts realizados en Unity se convierten a scripts C++ para que el compilador nativo finalice la compilación en la plataforma destino.

1.- Scripting

Un script para Unity tiene la siguiente estructura básica:

- **UnityEngine:** importa la clase MonoBehaviour necesaria para definir los scripts asociados a **GameObjects**.
- **System.Collections:** librería de .NET con listas, arrays, tablas hash...
- Clase definida que extiende a MonoBehaviour:
 - Método **Start**: se usa para inicializar el GameObject.
 - Método **Update**: se ejecuta una vez por frame, se le llama bucle del juego.



```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class HelloThere : MonoBehaviour
6 {
7     // Start is called before the first frame update
8     void Start()
9     {
10
11     }
12
13     // Update is called once per frame
14     void Update()
15     {
16
17     }
18 }
19
```

En C# todas las instrucciones deben acabar con ; igual que pasa en Java.

2.- Tipos de datos

Tipos de datos básicos de C# son:

int	Entero	4 bytes	$-2.147.483.648$ a $2.147.483.647$	0
float	Decimal	4 bytes	-1.5×10^{-45} a 3.4×10^{38}	0
string	Cadena de texto	Variable		Null
bool	Lógico	1 byte	True - False	False
uint	Entero positivo	4bytes	0 a 4.294.967.295	0
byte (sbyte)	Entero	1 byte	0 a 255 (-128 to 127)	0
double	decimal	8 bytes	-5.0×10^{-324} a 1.7×10^{308}	0

Tipos de datos compuestos:

- **Array**: conjunto de un tipo de dato.
- **List**: listas dinámicas (pueden modificar su tamaño).

2.- Tipos de datos

Tipos de datos específicos de C# para Unity son:

- **Vector3**: empaquetado de tres **floats** (x, y, z) que facilita el trabajo con puntos, vectores y direcciones dentro del espacio 3D.
- **GameObject**: referencia a un GameObject de la escena o a un **prefab** del proyecto.
- **Transform, Rigidbody** o cualquier otro componente: referencia a dicho componente dentro de un GameObject.
- **Texture, Material** o cualquier otro tipo de asset: referencia a un asset del proyecto

3.- Operadores

Los operadores son similares a los usados en Java:

Tipo	Operador	Operación
Asignación	=	Asigna un valor a una variable
Unarios	+ -	Permiten poner signo
Aritméticos	+ - * / %	Suma, resta, multiplicación, división, resto
	++variable	Pre-incremento en 1
	--variable	Pre-decremento en 1
	variable++	Post-incremento en 1
	variable--	Post-decremento en 1
Relacionales	< > <= >=	Menor que, mayor que, menor o igual, mayor o igual
	==	Comprueba si un dato igual a otro
	!=	Comprueba si dos datos son diferentes
Lógicos	&&	Comprueba si dos expresiones relacionales se cumplen
		Comprueba si una de dos expresiones relacionales se cumple
	!	Invierte el resultado de una expresión relacional
Concatenación	+	Une varias cadenas con cadenas y cadenas con variables

3.- Operadores

La clase **Mathf** permite cálculos con números tipo **float**:

```
Mathf.Abs(number)
Mathf.Round(number)
Mathf.Ceiling(number)
Mathf.Floor(number)
Mathf.Max(a, b, c, d)
Mathf.Min(a, b, c, d)
Mathf.Pow(number, exponent)
Mathf.Sqrt(number)
...
```

4.- Variables

Las variables en C# se definen igual que en java:

```
int age;
```

También se puede inicializar cuando se declara:

```
int age = 21;
```

Una vez declarada en cualquier momento se puede cambiar su valor:

```
age = 22;
```

4.- Variables

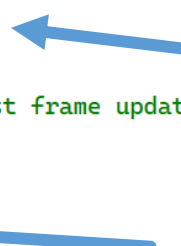
Si las variables se declaran para la clase como propiedades se pueden declarar su acceso como **public** o **private**.

Si la variable se declara dentro de un método no se debe declarar el tipo de acceso ya que solo será visible dentro de ese método.

```
public class HelloThere : MonoBehaviour
{
    private string username = "Rick";
    public int lives = 3;

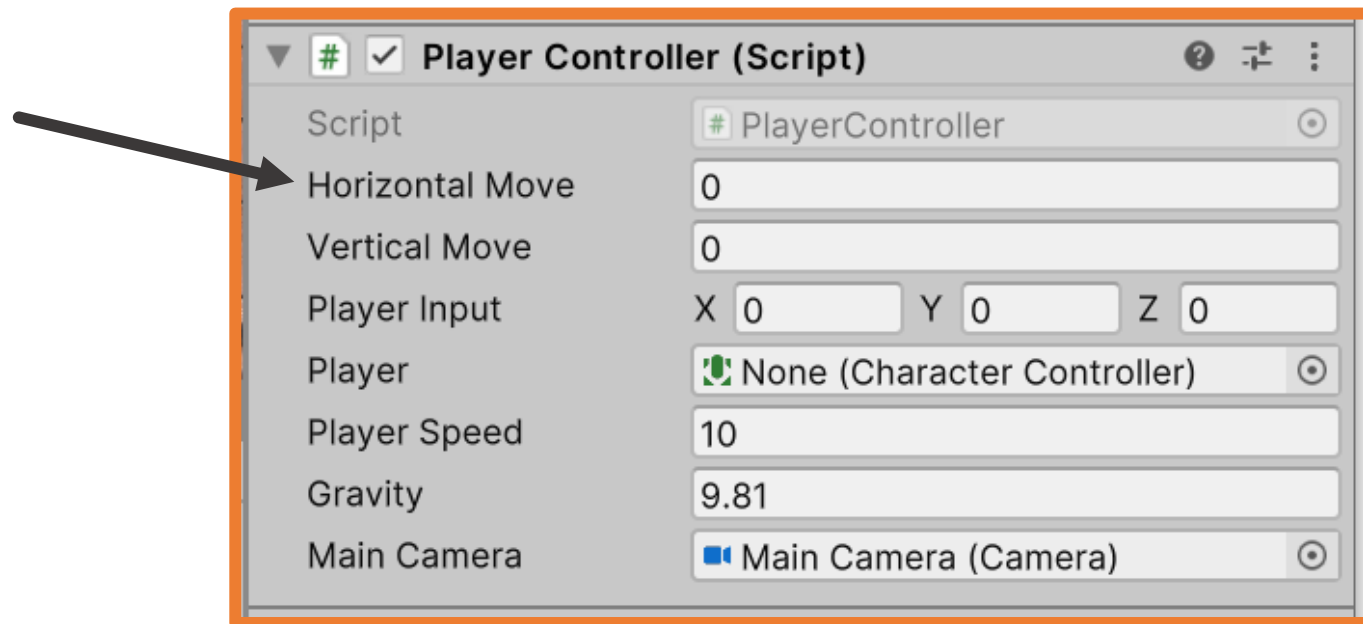
    // Start is called before the first frame update
    Mensaje de Unity | 0 referencias
    void Start()
    {
        int levelStars = 5;

        // Update is called once per frame
        Mensaje de Unity | 0 referencias
        void Update()
        {
            //Debug.Log("Update --> Hello there!");
        }
    }
}
```



4.- Variables

Las variables declaradas como propiedades con acceso public serán visibles desde el Inspector de Unity, permitiendo cambiar su valor allí.



4.- Variables

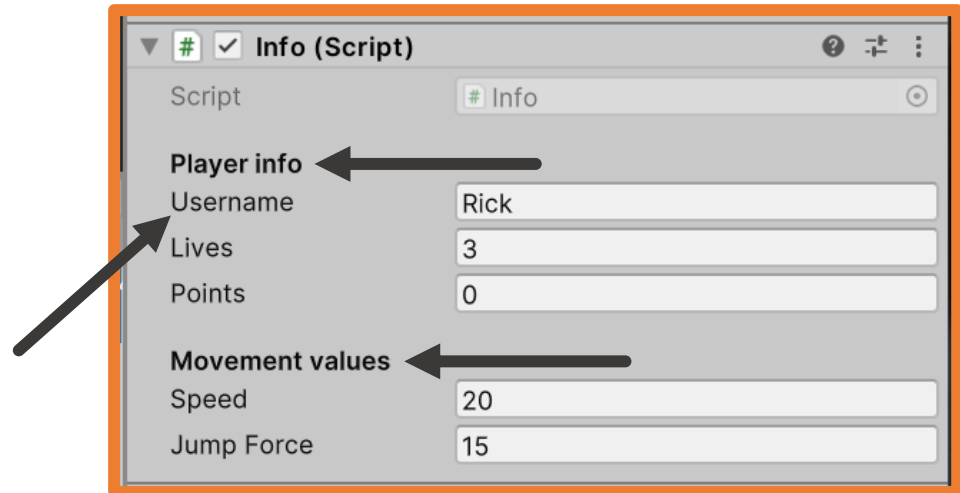
Se puede añadir cabeceras en la visualización en el inspector, e incluso se puede forzar la visualización de las variables privadas en el inspector.

```
public class Info : MonoBehaviour
{
    [Header("Player info")]
    [SerializeField] private string username = "Rick";
    public int lives = 3;
    public int points = 0;

    [Header("Movement values")]
    public float speed = 20;
    public float jumpForce = 15;

    // Start is called before the first frame update
    void Start()
    {
    }

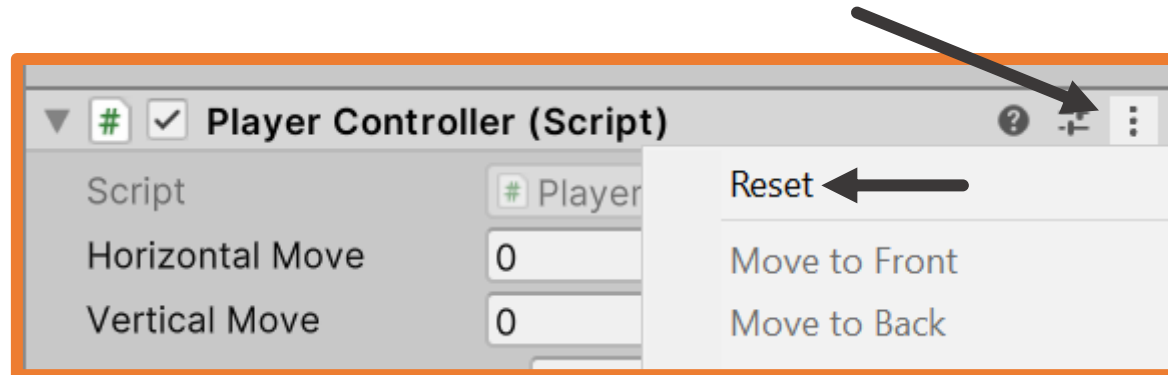
    // Update is called once per frame
    void Update()
    {
    }
}
```



4.- Variables

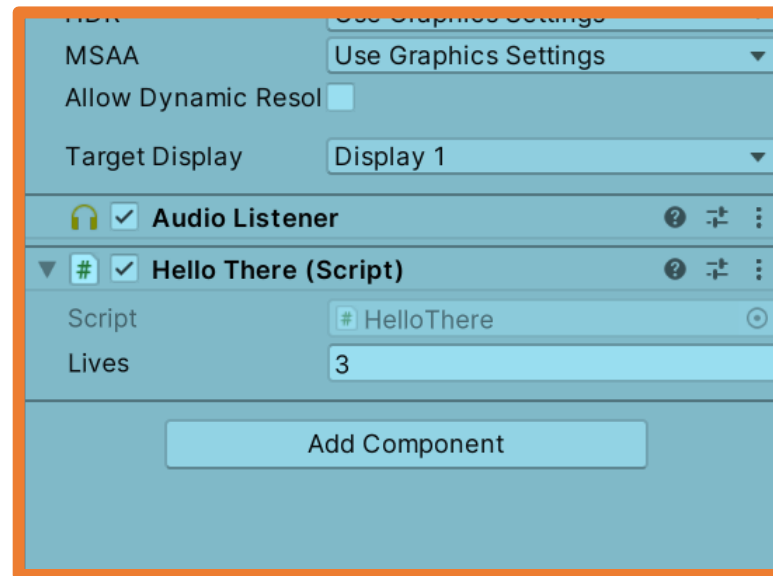
Si se modifica el valor desde el Inspector, este valor prevalece aunque se modifique en la declaración dentro del script.

Para que el Inspector vuelva a detectar el valor declarado en el script se debe resetear el componente.



4.- Variables

Hay que recordar que los cambios realizados durante la ejecución son cambios temporales y al parar la ejecución se volverá al estado previo a que se iniciara la ejecución.



4.- Variables

En C# se existen las **raw string** (cadena en la que se respetan todos los caracteres incluidos los saltos de línea) al envolver la cadena mediante tres comillas (C# v11).

Y también las **interpolated string** (cadenas que contienen variables).

Una **interpolated string** se declara poniendo el símbolo \$ delante de la string y poniendo las variables dentro de llaves {}.

```
string name = "Rick Sanchez";  
Debug.Log($"Bienvenido {name}");
```

Dentro de las llaves se pueden realizar operaciones si es necesario:

```
int number = Random.Range(0,11);  
Debug.Log($"El número {number} es {(number%2==0 ? "par" : "impar")}");
```

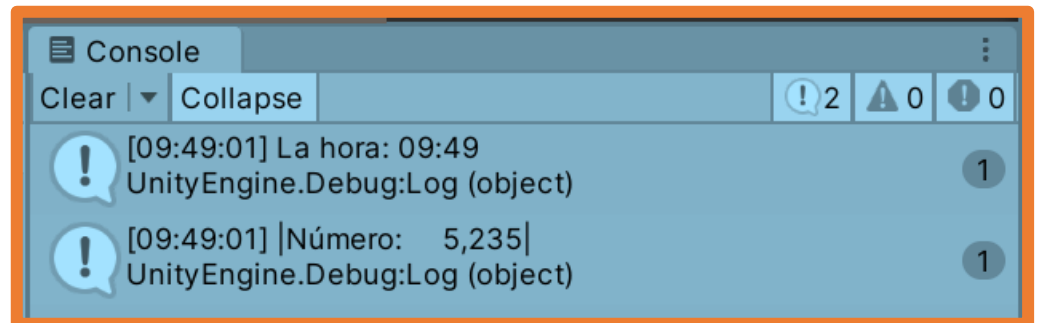

4.- Variables

El formato completo de una **interpolated string** es el siguiente:

```
{<interpolationExpression>[,<alignment>][:<formatString>]}
```

- **interpolationExpression**: la expresión a incluir. Puede ser una variable o una operación.
- **alignment**: parámetro opcional, indica la longitud que se ocupará la expresión a incluir y su alineación (positivo a la derecha y negativo a la izquierda).
- **formatString**: parámetro opcional, indica el formato que tendrá la expresión a incluir.

```
var time = DateTime.Now;  
Debug.Log($"La hora: {time:HH:mm}");  
  
float number = 5.2345113f;  
// Ocupa 9 espacios, alineado a la derecha y solo se muestran 3 decimales  
Debug.Log($"|Número: {number,9:F3}|");
```



5.- Estructuras de control

```
if (num >= 0) Debug.Log(num + " es positivo");
```

```
if (num >= 0) Debug.Log(num + " es positivo");  
else Debug.Log(num + " es negativo");
```


```
bool isPositive = (num >= 0) ? true : false;
```

```
for (int i = 1; i <= 20; i++)  
{  
    Debug.Log(i);  
}
```

```
int i = 1;  
do  
{  
    Debug.Log(i);  
    i++;  
} while (i <= 20);
```

```
int i = 1;  
while (i <= 20)  
{  
    Debug.Log(i);  
    i++;  
}
```

```
int vidas = 3;  
switch (vidas)  
{  
    case 1:  
        Debug.Log("Sólo te queda una vida");  
        break;  
    case 0:  
        Debug.Log("Te quedaste sin vidas");  
        break;  
    default:  
        Debug.Log("Esto no me cuadra");  
        break;
```



6.- Arrays

Declarar un **Array**

Se puede indicar su tamaño y posteriormente asignar valores:

```
int[] numbers = new int[10];  
string[] months = new string[12];
```

Se puede declarar asignando directamente los valores:

```
int[] numbers = { 0, 4, 7, 2, 5, 3 };  
string[] months = { "Enero", "Febrero", "Marzo", "Abril",  
                    "Mayo", "Junio", "Julio", "Agosto",  
                    "Septiembre", "Octubre", "Noviembre", "Diciembre"};
```

6.- Arrays

Recorrer un **Array**

Con un bucle **for**:

```
int[] numbers = { 0, 4, 7, 2, 5, 3 };  
for (int i = 0; i < numbers.Length; i++)  
{  
    Debug.Log("numbers[" + i + "] =" + numbers[i]);  
}
```

Con un bucle **foreach**:

```
int[] numbers = { 0, 4, 7, 2, 5, 3 };  
foreach (int number in numbers)  
{  
    Debug.Log(number);  
}
```

7.- Listas

Las **listas** permiten modificar su tamaño y añadir/eliminar elementos de la misma.

Se necesita importar la librería:

```
using System.Collections.Generic;
```

Declarar una lista vacía:

```
List<string> names = new List<string>();  
List<int> levelScores = new List<int>();
```

Declarar una lista asignando datos:

```
List<string> names = new List<string>(new string[] { "Rick", "Morty", "Summer" });
```

7.- Listas

Las **listas** permiten modificar su tamaño y añadir/eliminar elementos de la misma.

```
List<string> names = new List<string>(new string[] { "Rick", "Morty", "Summer" });
```

Modificar elementos se realiza igual que con los Array:

```
names[2] = "Beth";
```

Añadir elementos a una lista:

```
names.Add("Jerry");
```

```
names.Insert(1, "Jerry");
```

Eliminar elementos de una lista:

```
names.Remove("Summer");
```

```
names.RemoveAt(0);
```

Obtener la longitud de una lista:

```
int namesLength = names.Count;
```

7.- Listas

Las **listas** permiten modificar su tamaño y añadir/eliminar elementos de la misma.

```
List<string> names = new List<string>(new string[] { "Rick", "Morty", "Summer" });
```

Eliminar todos los elementos de una lista:

```
names.Clear();
```

Comprobar si un valor está en una lista:

```
if (names.Contains("Rick")) Debug.Log("Rick está en la lista.");
```

Ordenar una lista:

```
names.Sort();
```

Invertir el orden de una lista:

```
names.Reverse();
```

7.- Listas

Las **listas** permiten modificar su tamaño y añadir/eliminar elementos de la misma.

```
List<string> names = new List<string>(new string[] { "Rick", "Morty", "Summer" });
```

Recorrer una lista:

```
foreach (string name in names)
{
    Debug.Log(name);
}
```

Convertir una lista en un array:

```
string[] arrayNames = names.ToArray();
```

Convertir un array en una lista:

```
string[] arrayEnemies = { "Bowser", "Wario", "Waluigi", "Goomba" };
List<string> listEnemies = new List<string>(arrayEnemies);
```


8.- ArrayList

En un Array o en una Lista solo se pueden almacenar datos de un tipo.

En los **ArrayList** se permite almacenar datos de diferentes tipos además de no tener un tamaño predeterminado como las listas.

```
ArrayList userInfo = new ArrayList();
```

```
userInfo.Add(5);           // Vidas  
userInfo.Add(3541.5f);     // Dinero  
userInfo.Add("Rick Sanchez"); // Nombre  
userInfo.Add("TheRick");   // NickName
```

Se debe tener cuidado al cambiar los datos porque el compilador no realiza ninguna comprobación de tipo de dato:

```
userInfo[1] = "Hello there!";
```

9.- HashTable

Los **HashTable** también conocidos como diccionarios son un tipo de dato compuesto en el que cada elemento es un par clave-valor.

En los HashTable se accede a los datos a través de la clave.

```
Hashtable userInfo = new Hashtable();
```

```
userInfo.Add("vidas", 5);  
userInfo.Add("dinero", 3541.5f);  
userInfo.Add("name", "Rick Sanchez");  
userInfo.Add("username", "TheRick");
```

```
userInfo["username"] = "Rickbest";
```


Las claves siempre deben ser de tipo **string**.

El tamaño de una HashTable es dinámico como las listas.

9.- HashTable

Se puede comprobar si existe una clave con el método **Contains** y para usar el valor se debe realizar un casting:

```
if (userInfo.Contains("dinero"))  
{  
    Debug.Log("EL jugador tiene " + (float)userInfo["dinero"] + " monedas.");  
}
```



Recorrer un HashTable:

```
foreach (string key in userInfo.Keys)  
{  
    Debug.Log("La clave es: " + key);  
    Debug.Log("y su valor " + userInfo[key]);  
}
```

10.- Métodos

Como en cualquier **lenguaje orientado a objetos** dentro de las **clases** se pueden crear **métodos**.

En C# para crear un método se debe seguir la siguiente estructura:

- Tipo de acceso (si no se pone nada por defecto será private).
- Tipo de dato que devuelve (void si no devuelve nada).
- Identificador del método (primera letra en mayúscula y CamelCase).
- Paréntesis para indicar los parámetros si los hay.
- Llaves para indicar el bloque de código del método.

```
void Start()  
{  
    ...  
}
```

10.- Métodos

Declaración de un método:

```
void add(int number1, int number2)
{
    int result = number1 + number2;
    Debug.Log(result);
}
```

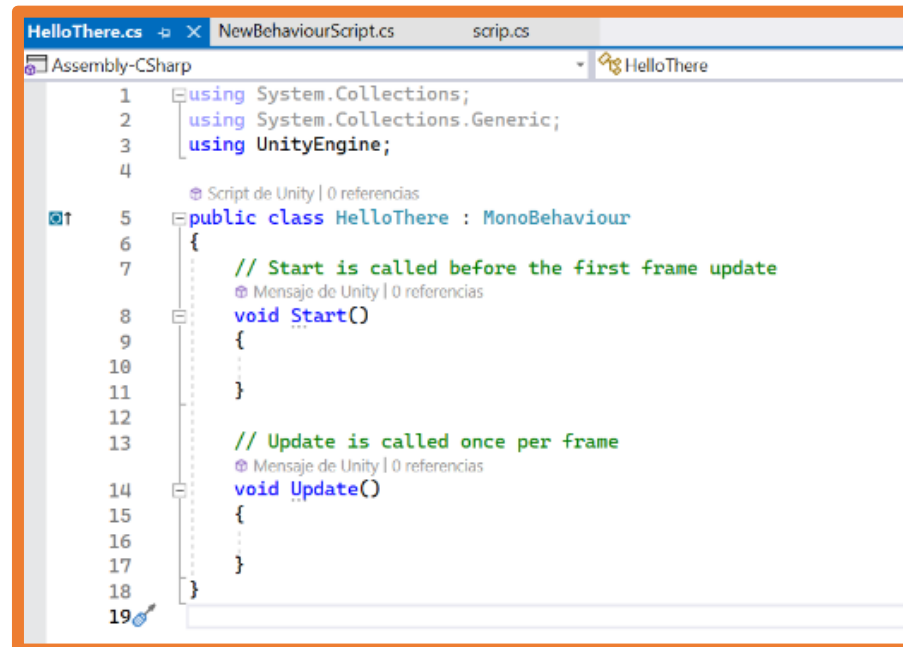
La llamada a un método se realiza igual que en cualquier lenguaje de programación orientado a objetos:

```
add(25267, 13937);
```

11.- Clase MonoBehaviour

Cuando se crea un script C# en Unity se puede observar que la clase que contiene el script hereda de la clase **MonoBehaviour**.

La clase **MonoBehaviour** es necesaria para los scripts que se asocian a los **GameObject** del proyecto.

A screenshot of the Unity C# script editor. The top bar shows three tabs: 'HelloThere.cs', 'NewBehaviourScript.cs', and 'scrip.cs'. The 'HelloThere.cs' tab is active. The script content is as follows:

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class HelloThere : MonoBehaviour
6 {
7     // Start is called before the first frame update
8     void Start()
9     {
10     }
11
12     // Update is called once per frame
13     void Update()
14     {
15     }
16 }
17
18
19
```

On the left side, there is a 'Hierarchy' pane showing 'Assembly-CSharp' and 'HelloThere' under it. The 'HelloThere' class is highlighted. The script content is displayed in a light blue font on a white background.

11.- Clase MonoBehaviour

Como **Unity funciona por eventos**, dentro de las clases que heredan de MonoBehaviour se pueden incluir una serie de métodos definidos por defecto que se ejecutarán cuando se produzcan los eventos asociados a dichos métodos.

Por defecto se añaden dos métodos que se ejecutan con determinados eventos:

- **Start:** se llama en el frame en el que el script se activa y justo antes de la primera llamada de cualquier método Update.
- **Update:** se llama en cada frame.

Hay muchos métodos más que se pueden consultar en la documentación:

<https://docs.unity3d.com/ScriptReference/MonoBehaviour.html>

13.- Bucle del juego

Se puede definir como **bucle del juego** las diferentes fases por las que pasa un script cuando se ejecuta:

- Inicialización
- Actualización de físicas
- Actualización de INPUT
- Lógica del juego
- Destrucción

13.- Bucle del juego

Cada una de estas fases tiene asociados un conjunto de métodos de la clase MonoBehaviour:

Fase	Métodos
Inicialización	Awake y Start
Actualización de físicas	FixedUpdate, OnTriggerEnter, OnTriggerStay, OnTriggerExit, OnCollisionEnter, OnCollision Stay...
Actualización de INPUT	OnMouseDown (Up, Drag, Enter, Exit, Over...), GetKey, GetKeyDown, GetKeyUp...
Lógica del juego	Update, LateUpdate.
Destrucción	OnApplicationQuit y OnDestroy

Al ejecutar el juego, Unity comienza a ejecutar el bucle del juego, de esta manera, en cada fase ejecutará los métodos correspondientes que se encuentren en los scripts añadidos a los GameObject de la escena.

13.- Bucle del juego

El método **LateUpdate** es interesante ya que se ejecuta siempre después de que acaben todos los métodos tipo Update.

De esta manera se pueden realizar operaciones asegurándose de que se hayan realizado previamente todos los movimientos y cálculos como por ejemplo, ajustar la cámara para que siga a un personaje.

13.- Bucle del juego

Ejemplo de un script implementando algunos de los métodos de la clase MonoBehaviour:

```
5 public class EventosEsfera : MonoBehaviour {
6
7     void Awake () {
8         Debug.Log ("Estoy en el Awake de la esfera");
9     }
10
11    void Start () {
12        Debug.Log ("Estoy en el Start de la esfera");
13    }
14
15    // Update is called once per frame
16    void Update () {
17        Debug.Log ("Estoy en el Update de la esfera");
18    }
19
20    void OnEnable () {
21        Debug.Log ("Se activa la esfera");
22    }
23
24    void OnDisable () {
25        Debug.Log ("Se desactiva la esfera");
26    }
27
28    |
29 }
```