# Programación básica en Unity®: Introducción al scripting C#

por Héctor Costa Guzmán · hektorprofe.net

# Preparación

# Instalación de Visual Studio

- VS Community es un IDE (entorno de desarrollo integrado).
- Es el IDE recomendado para scripting en Unity.
- Se encuentra como módulo en Unity Hub.

# Configuración del proyecto

1. Crear un proyecto 3D llamado *Scripting* y una escena *Fundamentos*.
2. Comprobar que en el proyecto tenemos el paquete *Visual Studio Editor* en: ***Window > Package Manager > Unity Registry***.
3. Comprobar que el editor *Visual Studio* está configurado en: ***Edit > Preferences > External tools > External Script Editor***.
4. Para añadir un script a un GameObject podemos:
    - Crear un script en los recursos del proyecto y arrastrarlo al objeto.
    - Añadir un nuevo componente al objeto y escribir el nombre del script.
5. Añadir un script llamado *BloqueControl* al objeto *Bloque*.
6. Abrir el script haciendo doble clic desde el recurso o el componente.

# Fundamentos

# ¿Qué significa scripting?

- Crear scripts (*manuscripts*, series de instrucciones).
- No son programas, sino fragmentos de los programas:
  - Con la *programación* puedes crear tus propios programas.
  - Con los *scripts* puedes definir el comportamiento de los programas.
- Los scripts en Unity sirven para definir el comportamiento de los *GameObjects*, concretamente sus *componentes* y *propiedades*.
- Los scripts también son componentes y se programan en *C#* (*C Sharp*).

# Estructura de un script

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;                          Cabeceras

public class BloqueControl: MonoBehaviour
{
    void Start()
    {

    }

    void Update()
    {

    }                          Métodos
}                                           Clase
```

- **Cabeceras**: Sirven para importar definiciones del sistema y del motor.
- **Clase**: Define un nuevo tipo de dato con sus instrucciones, identificado con un nombre.
- **Métodos**: Son bloques de código con nombre. Éste puede ser dado por el programador o estar enlazado a un evento del ciclo de vida:
  - ***Start***: Evento llamado justo antes del primer fotograma del videojuego.
  - ***Update***: Evento llamado al actualizar cada fotograma del videojuego.

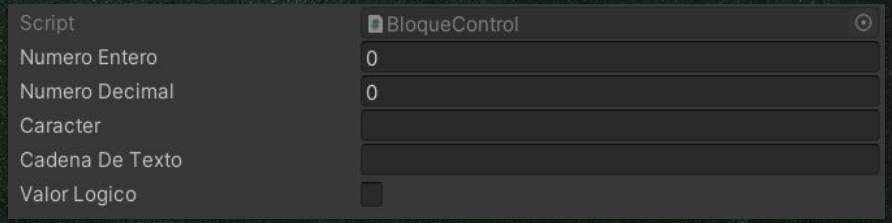
# Ciclo de vida de un script

- Los scripts permiten ejecutar código en diferentes momentos.
- En la documentación *encontramos un repaso* de esos momentos.
- El ciclo de vida empieza al reproducir la primera escena.
- Termina al parar la reproducción, cerrar el juego o destruir el GameObject.
- Eventos más comunes programados al sobreescribir los métodos de clase:
  - Inicialización: *Awake* y *Start*.
  - Actualización: *FixedUpdate*, *Update* y *LateUpdate*.
  - Desmantelamiento: *OnApplicationQuit* y *OnDestroy*.

# Variables y tipos de datos

- Las *variables* sirven para almacenar información (datos) en la memoria.
- En C# los *datos* se dividen en:
  - **Estructuras** *(tipo-valor)*:
    - Números *(int, float...)*
    - Caracteres *(char)*
    - Lógicos *(bool)*
    - Vectores, cuaterniones, etc.
  - **Clases** *(tipo-referencia)*:
    - Cadenas de texto *(string)*
    - Listas, diccionarios, scripts, etc.
- Se recomienda la notación *camelCase*.
- Su *encapsulación* es *privada* por defecto, pero si se cambia a *pública*, Unity tomará la variable como una ***propiedad***.

```
public class BloqueControl : MonoBehaviour
{
    // Definición de las variables de clase
    public int numeroEntero;
    public float numeroDecimal;
    public char caracter;
    public string cadenaDeTexto;
    public bool valorLogico;

    void Start() { }
}
```



# Ejemplos de valores literales

A través de código los valores de un los diferentes tipos se escriben con diferente sintaxis:

| | | | |
|---|---|---|---|
| Enteros | int | 10 -6 0 9999 -321 | * *Sin nada específico* |
| Dobles | double | 3.14 123.456 0 -10.5 | * *La coma es un punto* |
| Flotantes | float | 3.14f 123.456f 0f -10.5f | * *Con punto y F al final* |
| Caracteres | char | 'A' 'c' '5' 'p' | * *Entre comillas simples* |
| Lógicos | bool | true false | * *Son palabras reservadas* |
| Cadenas | string | "Hola mundo" "Soy un texto" | * *Entre comillas dobles* |
| Vectores 2D | Vector2 | (0, 0) (1, 0) (-0.5f, 1.5f) | * *2 números entre paréntesis* |
| Vectores 3D | Vector3 | (0, 0, 0) (1, 0, 0) (-0.5f, 1.5f, 2.5f) | * *3 números entre paréntesis* |
| Cuaterniones | Quaternion | (0, 0, 0, 0) (0.1f, 0.2f, 0.3f, 0.4f) | * *4 números entre paréntesis* |

# Manipulación de datos

- *Operador de asignación* para almacenar un valor en una variable: **=**
- *Operaciones aritméticas* entre dos variables o valores numéricos:
  Suma: **+** Resta: **-** Producto: ***** División: **/** Módulo: **%**
  Si la expresión tiene varios operadores la precedencia se calcula automáticamente.
- *Operaciones en asignación* sobre la propia variable numérica:
  Suma: **+=** Resta: **-=** Producto: ***=** División: **/=** Módulo: **%=**
- *Operadores de incremento y decremento* sobre la propia variable numérica:
  Incremento en 1: **++** Decremento en 1: **--**

```
public int numero1, numero2, resultado;

void Start()  // Se ejecuta una vez al principio
{
    resultado = numero1 + numero2;
}
```

```
public int numero1, numero2, resultado;

void Update()  // Se ejecuta en cada fotograma
{
    resultado = (numero1 + numero2) * 10 / 2;
}
```

# Propiedades del componente Transform

- *Posición*: Propiedad representada por una estructura *Vector3*.
- *Rotación*: Propiedad representada por una estructura *Quaternion*.
- *Escala*: Propiedad representada por una estructura *Vector3*.

```
void Start() {
    print(transform.position);
    print(transform.rotation);      // La rotación se gestiona en cuaterniones
    print(transform.eulerAngles)    // Podemos consultar su equivalencia en ángulos
    print(transform.localScale);    // Esta escala se manipula respecto al propio objeto
}
```

- Se trata de estructuras compuestas que no se pueden modificar directamente:

```
void Start() {
    transform.position.x = 10;    // Error
}
```

# Manipulación de un Vector3

- Es necesario crear el dato antes de asignarlo (o usar un valor predeterminado):

```
void Start()
{
    Vector3 nuevaPosicion;

    nuevaPosicion.x = 2;
    nuevaPosicion.y = 0.5f;
    nuevaPosicion.z = 0;

    transform.position = nuevaPosicion;
}
```

```
void Start()
{
    Vector3 nuevaPosicion;

    // Asignación abreviada mediante new
    nuevaPosicion = new Vector3(2, 0.5f, 0);

    transform.position = nuevaPosicion;
}
```

```
void Start()
{
   transform.position = new Vector3(2, 0.5f, 0);   // Valor en asignación sin necesidad de variable
}
```

# Valores predeterminados de Vector3

- Se utilizan como atajos:
  - *Vector3.zero* → (0, 0, 0)
  - *Vector3.one* → (1, 1, 1)
  - *Vector3.left* → (-1, 0, 0)
  - *Vector3.right* → (1, 0, 0)
  - *Vector3.up* → (0, 1, 0)
  - *Vector3.down* → (0, -1, 0)
  - *Vector3.forward* → (0, 0, 1)
  - *Vector3.back* → (0, 0, -1)
- Son útiles como valores escalables.

```
void Update()
{
    // 1 metro a la derecha por fotograma
    transform.position += Vector3.right;
}
```

```
void Update()
{
    // 0.01 metro a la derecha por fotograma
    transform.position += Vector3.right/100;
}
```

# Intervalo entre fotogramas

- ¿Cómo podemos manipular datos en función del tiempo y no de los fotogramas?
- La respuesta se encuentra en la variable *Time.deltaTime*.
- Unity almacena en esta variable el tiempo entre cada fotograma.
- Al multiplicarla por un valor conseguimos ese valor en función del tiempo:

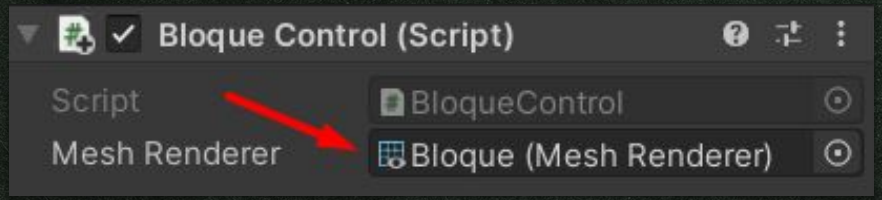
```
public float tiempoEntreFotogramas, fotogramasPorSegundo;

void Update()
{
    tiempoEntreFotogramas = Time.deltaTime;
    fotogramasPorSegundo = 1 / tiempoEntreFotogramas;
    transform.position += Vector3.right * Time.deltaTime;  // 1 metro a la derecha por fotograma
}
```

# Acceso a los componentes

- El componente *Transform* es el único que tiene un acceso directo, *transform*.
- Para referirnos a los demás componentes necesitamos referencias a ellos.
- Existen múltiples formas de recuperar la referencia a un componente.
- La más fácil consiste en crear una propiedad del tipo del componente:

```
public MeshRenderer meshRenderer;
```

- Y seleccionar el componente del *GameObject* que queremos referenciar:




```
void Start() {
    meshRenderer.material.color = Color.green;
}
```

# Extra: El método GetComponent

- Otra forma de recuperar un componente del *GameObject* dinámicamente mediante código es utilizando el método interno *GetComponent<Clase>()*. Generalmente se ejecuta en el *Start* como en el siguiente ejemplo:

```
public MeshRenderer meshRenderer;

void Start() {
    meshRenderer = GetComponent<meshRenderer>();
}
```

- Este método funcionará sólo cuando el *GameObject* actual tenga el componente especificado, en caso contrario ocurrirá un error de referencia nula.

# Control de flujo

# El tipo lógico

- Representa la veracidad o falsedad de una sentencia.
- Los valores verdadero y falso se escriben como *true* y *false*.
- En programación sirven para controlar el flujo del código.
- Se pueden negar mediante el operador de negación *!*.

```
void Start()
{
    print(true);
    print(false);

    print(!true);
    print(!false);
}
```

```
public bool logico, logicoNegado;

void Start() {
    logico = true;
}

void Update() {
    logicoNegado = !logico;
}
```

# Operaciones relacionales

- Sirven para comparar dos valores (números, cadenas, lógicos, vectores, etc).
- El resultado es siempre un valor lógico (*true* o *false*).

```
public int num1, num2;
public bool igual, distinto, mayor, menor, mayorIgual, menorIgual;

void Update()
{
    igual       =  num1 == num2;    // Igual que
    distinto    =  num1 != num2;    // Distinto de
    mayor       =  num1 >  num2;    // Mayor que
    mayorIgual  =  num1 >= num2;    // Mayor o igual que
    menor       =  num1 <  num2;    // Menor que
    menorIgual  =  num1 <= num2;    // Menor o igual que
}
```

# Condición if-else

- Permite ejecutar un bloque de código evaluando una expresión lógica.
- La expresión lógica se denomina condición y si es *true* se ejecutará el bloque.
- Se le puede enlazar un bloque *else* para ejecutar el código en el caso contrario.

```
public bool activado;
public MeshRenderer meshRenderer;

void Update()
{
   if (activado)
   {
      meshRenderer.material.color = Color.green;
   }
}
```

```
void Update()
{
   if (activado) {
      meshRenderer.material.color = Color.green;
   }

   else {
      meshRenderer.material.color = Color.red;
   }
}
```

# Variante if-else if-else

- Permite ejecutar múltiples comprobaciones anidadas.
- Cuando se cumple una de las condiciones, las que hay por debajo no se llegan a comprobar.
- *Else* se utiliza como condición por defecto en caso de que ninguna de las otras se cumpla.

```
public int valor;

void Update()
{
    if (valor > 5) {
        meshRenderer.material.color = Color.green;
    }
    else if (valor < -5) {
        meshRenderer.material.color = Color.red;
    }
    else {
        meshRenderer.material.color = Color.yellow;
    }
}
```

# Operaciones lógicas

- Surgen de la necesidad de conectar múltiples comparaciones.
- Las dos conectivas lógicas esenciales son la *conjunción* y la *disyunción*.
- En programación se indican con los operadores AND *&&* y OR *||* respectivamente.
- Su resultado es un valor lógico *true* o *false* dependiendo si se cumple la conectiva.
- Los resultados posibles son limitados y se resumen en dos tablas de la verdad.

```
public bool valorA, valorB, and, or;

void Update() {
    and = valorA && valorB;
    or = valorA || valorB;
}
```

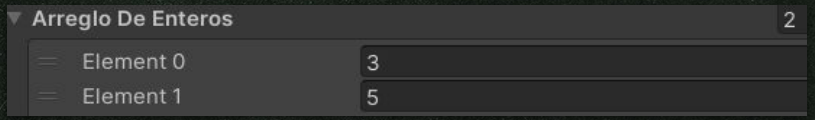
| A | B | A **AND** B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| A | B | A **OR** B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

# Arreglos de datos

- Son estructuras que permiten almacenar múltiples datos de un mismo tipo.
- El número de datos que pueden almacenar es fijo y lo establece el programador.

```
public int[] arregloDeEnteros;
```

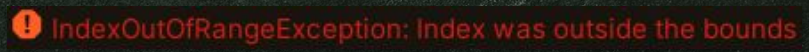



- Podemos manipular sus datos mediante un índice entre corchetes: *arreglo[indice]*
- El índice indica la posición empezando en **0** → *Primer valor*, **1** → Segundo valor...

```
arregloDeEnteros[1] = 99;
print(arregloDeEnteros[1]);
```


- Acceder a una posición fuera del tamaño del arreglo ocasionará una excepción:

```
print(arregloDeEnteros[999]);
```



# Bucle foreach (1)

- Los bucles sirven para ejecutar un mismo código múltiples veces.
- El bucle *foreach* se basa en ir recorriendo los valores de un arreglo.
- Se ejecutará tantas veces como valores haya en el arreglo.
- No permite modificar los valores del arreglo secuencialmente.

```
void Start()
{
    foreach (int numero in arregloDeEnteros)
    {
        print(numero * 10);
        numero *= 10;  // Error al intentar modificar un valor
    }
}
```

# Bucle foreach (2)

- Para modificar un valor es necesario acceder mediante el índice.
- Podemos definir una variable externa e incrementarla para replicar su posición.

```
int contador = 0;
foreach (int numero in arregloDeEnteros) {
    print(arregloDeEnteros[contador]);
    contador++;
}
```

- Usando el contador como índice podemos modificar los valores secuencialmente:

```
foreach (int numero in arregloDeEnteros) {
    arregloDeEnteros[contador] *= 10;
}
```

# Bucle for

- Este bucle no controla automáticamente el número de repeticiones.
- Se basa en definir un *contador*, una *condición* y un *paso iterativo*.
- El paso iterativo modifica el contador para eventualmente romper la condición.

```
public int repeticiones;

for (int contador=0; contador < repeticiones; contador++) {
    print("Número de repetición -> " + contador);
}
```

- Al definir el contador y el paso permite recorrer arreglos muy fácilmente.
- El número de repeticiones es la longitud del arreglo, *accesible vía arreglo.Length*.
- A menudo se denomina al contador simplemente *i debido al vector unitario*:

```
for (int i=0; i < arregloDeEnteros.Length; i++) { arregloDeEnteros[i] *= 10; }
```

# Práctica de scripting

# Movimiento de bloques en conjunto

```
public class MovedorControl : MonoBehaviour {
    public Transform[] bloques;
    public Vector3[] direcciones;

    void Start() {
        direcciones = new Vector3[] { Vector3.right, Vector3.left, Vector3.right, Vector3.left };
    }

    void Update() {
        for(int i=0;i<bloques.Length;i++) {
            bloques[i].position += (i + 1) * direcciones[i] * Time.deltaTime;
            if (bloques[i].position.x < -4) {
                direcciones[i] = Vector3.right;
            } else if (bloques[i].position.x > 4) {
                direcciones[i] = Vector3.left;
            }
        }
    }
}
```