

CE Desarrollo de videojuegos y realidad virtual

Programación y motores de videojuegos



**GENERALITAT
VALENCIANA**

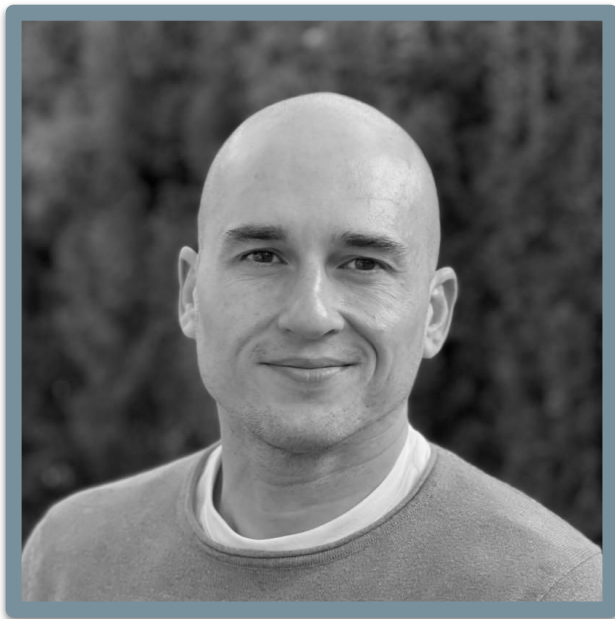
Conselleria d'Educació,
Investigació, Cultura i Esport



Unió Europea

Fons Social Europeu

L'FSE inverteix en el teu futur



Material elaborado por:

Edu Torregrosa Llácer

edutorregrosa.com

Esta obra está licenciada bajo la licencia **Creative Commons Atribución-NoComercial-Compartirigual 4.0 internacional**. Para ver una copia de esta licencia visita:
<https://creativecommons.org/licenses/by-nc-sa/4.0/>



**Attribution-NonCommercial-ShareAlike
4.0 International (CC BY-NC-SA 4.0)**

Máster Programación de videojuegos en Unity

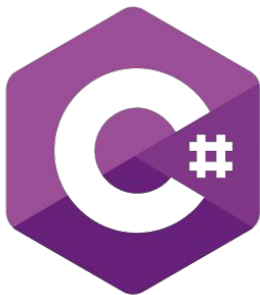


1. **Introducción a Unity y C#.**

2. Interfaz y espacio 3D en Unity.
3. GameObjects, componentes y prefabs.
4. API de Unity.
5. Inteligencia Artificial.
6. Multijugador.



Introducción a Unity y C#



1. Introducción a Unity y C#.
2. Espacio de trabajo en Unity.
3. Clase MonoBehaviour.
4. C# en Unity.
 - 4.1. Tipos de datos.
 - 4.2. Operadores y expresiones.
 - 4.3. Condicionales y bucles.
 - 4.4. Funciones, métodos y corrutinas.
 - 4.5. OO: Clases, herencia, interfaces, etc.
 - 4.6. Estructuras de datos. Arrays y colecciones.
 - 4.7. Scripts en el inspector de Unity.
 - 4.8. Eventos (teclado, ratón, componentes).
 - 4.9. Persistencia de datos.
 - 4.10. Optimización y buenas prácticas.

Introducción a Unity

Unity es uno de los motores de desarrollo de videojuegos más populares en la industria actual.

- **Interfaz Amigable:** Al abrir Unity, uno se encuentra con una interfaz modular que puede ser personalizada según las necesidades del proyecto. Está diseñada para ser intuitiva, permitiendo una fácil navegación entre escenas, la jerarquía de objetos, y los componentes de cada objeto. Los paneles pueden ser organizados de manera personal, lo que permite adaptar el espacio de trabajo a las preferencias individuales de cada desarrollador.
- **Desarrollo Centrado en Componentes:** Unity adopta un enfoque basado en componentes para el desarrollo. Cada objeto en una escena, ya sea un personaje, una luz o un simple cubo, es un "**GameObject**". Estos objetos se pueden personalizar y ampliar mediante la adición de componentes, como scripts, colisionadores o sistemas de partículas.
- **C# como Pilar Central:** Unity utiliza C# como lenguaje principal de programación. Este lenguaje, orientado a objetos. Es especialmente accesible para aquellos estudiantes que provienen de un trasfondo en JAVA.

Introducción a Unity

- **Física y Gráficos Avanzados:** Unity integra sistemas avanzados de física y gráficos. Esto permite crear juegos con físicas realistas y gráficos impresionantes sin necesidad de escribir sistemas complejos desde cero.
- **Colaboración y Comunidad:** La comunidad de Unity es vasta y activa. A través de foros, tutoriales y eventos, los desarrolladores pueden compartir conocimientos y solucionar problemas conjuntamente.
- **Realidad Virtual y Aumentada:** Unity es líder en el desarrollo de aplicaciones VR y AR, con soporte para la mayoría de las principales plataformas y dispositivos.
- **Asset Store:** La tienda de activos de Unity, conocida como Asset Store, ofrece miles de recursos preparados para ser utilizados en los proyectos, desde modelos 3D y texturas hasta scripts y sistemas completos.

[Instalar Unity](#)

Introducción a C#: HolaMundo

Introducción a C#

C# es un lenguaje de programación orientado a objetos desarrollado por Microsoft como parte de su plataforma .NET. Es un lenguaje moderno, fuertemente tipado y con una sintaxis similar a otros lenguajes como Java y C++. C# combina las capacidades de programación de alto nivel con la potencia de lenguajes de bajo nivel.

C# es el lenguaje utilizado para escribir scripts en Unity. Estos scripts controlan el comportamiento de los objetos en el juego, su lógica, y su interacción con otros objetos.

Unity ha hecho un gran trabajo en optimizar el rendimiento del código C# a través de herramientas que convierten el código C# a C++ y luego lo compila, logrando un rendimiento cercano al código nativo.

Ejemplo de HolaMundo en C#:

```
using System;
class HolaMundo
{
    static void Main(string[] args)
    {
        Console.WriteLine("¡Hola, Mundo!");
    }
}
```


Introducción a C#

HolaMundo en Unity con C#:

```
using UnityEngine;  
using UnityEngine.UI;
```

```
public class HolaMundoUI : MonoBehaviour  
{  
    public Text textoUI;  
  
    void Start()  
    {  
        textoUI.text = "Hola mundo desde Unity";  
    }  
}
```

```
using UnityEngine;  
public class HolaMundo : MonoBehaviour  
{  
    void Start()  
    {  
        Debug.Log("Hola mundo desde Unity");  
    }  
}
```

HolaMundo en Unity con C#
en la pantalla de juego:

Introducción a Unity: Espacio de trabajo

GameObject en Unity

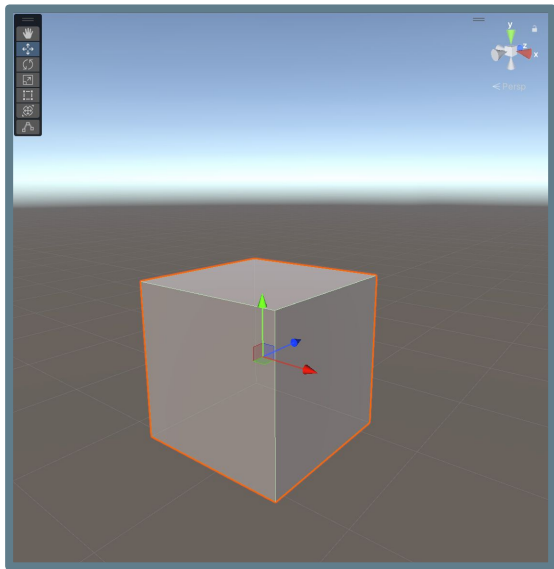
GameObject: Es el objeto base en Unity que representa cualquier entidad en una escena. Puede ser un personaje, un objeto, una luz, una cámara, etc. Los GameObjects pueden tener componentes asociados, como transformaciones, renderizadores y scripts, para definir su comportamiento y apariencia.

- **Jerarquía:** Los GameObjects pueden tener relaciones jerárquicas. Un GameObject puede ser hijo de otro, permitiendo estructuras anidadas.
- **Componentes:** La funcionalidad y el comportamiento de un GameObject se determinan por los componentes que se le adjunen.
- **Tags y Layers:** Los GameObjects pueden ser etiquetados con "Tags" para identificarlos y agruparlos fácilmente en el código. Además, pueden ser asignados a "Layers" para controlar aspectos como el renderizado y las colisiones.
- **Scripts Personalizados:** Mediante el uso de C# puedes escribir scripts personalizados y adjuntarlos como componentes a GameObjects. Estos scripts permiten definir comportamientos específicos y reacciones a eventos, dándote una gran flexibilidad para desarrollar la lógica de tu juego.

El espacio 3D en Unity

En Unity, el espacio 3D utiliza un sistema de coordenadas cartesianas con ejes X, Y y Z. Un punto en este espacio se representa mediante un vector tridimensional (**Vector3 en C#**), donde:

- **X**: Representa la coordenada horizontal (izquierda-derecha).
- **Y**: Representa la coordenada vertical (abajo-arriba).
- **Z**: Representa la coordenada de profundidad (delante-atrás).



Objetos 3D y Meshes: Los objetos 3D en Unity se representan mediante "meshes" o mallas. Una malla es una colección de vértices, aristas y caras que definen la forma de un objeto 3D. Estos objetos pueden ser importados de herramientas de modelado externas como Blender. Unity nos proporciona una serie de objetos 3D predefinidos, como cubos, cilindros, cápsulas, etc.

El espacio 3D en Unity

- **Componentes:** La funcionalidad y el comportamiento de un GameObject se determinan por los componentes que se le adjunten.
- **Componente Transform:** Cada objeto en la escena de Unity tiene un componente "Transform", que contiene información sobre su posición, rotación y escala. Mediante este componente, puedes mover, girar y escalar objetos en el espacio 3D.
 - **Position:** Define la ubicación del objeto en la escena en coordenadas X, Y y Z.
 - **Rotation:** Determina la orientación del objeto basada en los ángulos de rotación alrededor de los ejes X, Y y Z.
 - **Scale:** Define el tamaño del objeto a lo largo de los ejes X, Y y Z.

```
// mueve el objeto actual a la posición (3,2,1)
transform.position = new Vector3(3, 2, 1);
// mueve el objeto actual 1 unidad hacia la derecha
transform.position += Vector3.right; // Vector3.right → new Vector3(1,0,0)
// mueve el objeto actual 10 unidades hacia la derecha
transform.position += Vector3.right*10;
```

El espacio 3D en Unity

EJEMPLOS ROTACIÓN:

```
// rota el objeto 45 grados alrededor del eje Y:
transform.rotation = Quaternion.Euler(0, 45, 0);
// incrementa la rotación del objeto en 15 grados alrededor del eje Z
transform.rotation *= Quaternion.Euler(0, 0, 15);
// establece la rotación del objeto a su rotación original (sin rotación)
transform.rotation = Quaternion.identity;
```

EJEMPLOS ESCALADO:

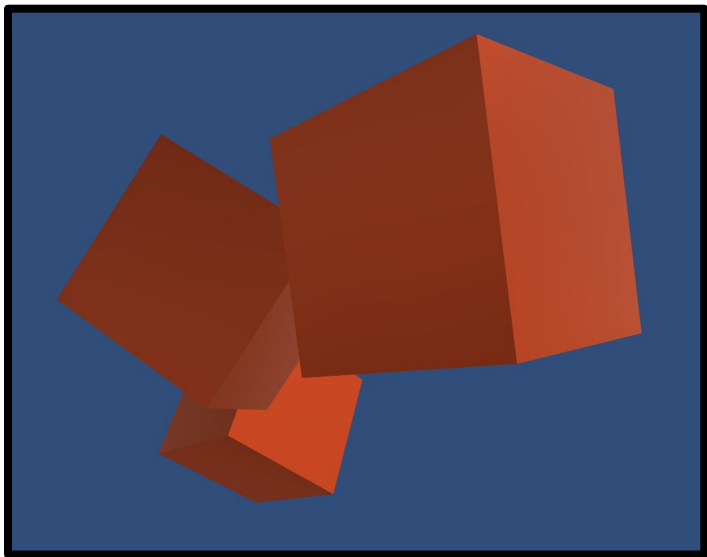
```
// escala a un tamaño específico, objeto a (2, 2, 2):
transform.localScale = new Vector3(2, 2, 2);
// aumenta el tamaño del objeto en una unidad en todos los ejes
transform.localScale += new Vector3(1, 1, 1);
// escala el objeto al doble de su tamaño actual
transform.localScale *= 2;
// incrementa el tamaño del objeto en el eje Y en 0.5 unidades
transform.localScale += new Vector3(0, 0.5f, 0);
// restablece el tamaño del objeto a su tamaño original
transform.localScale = Vector3.one;
```

El espacio 3D en Unity

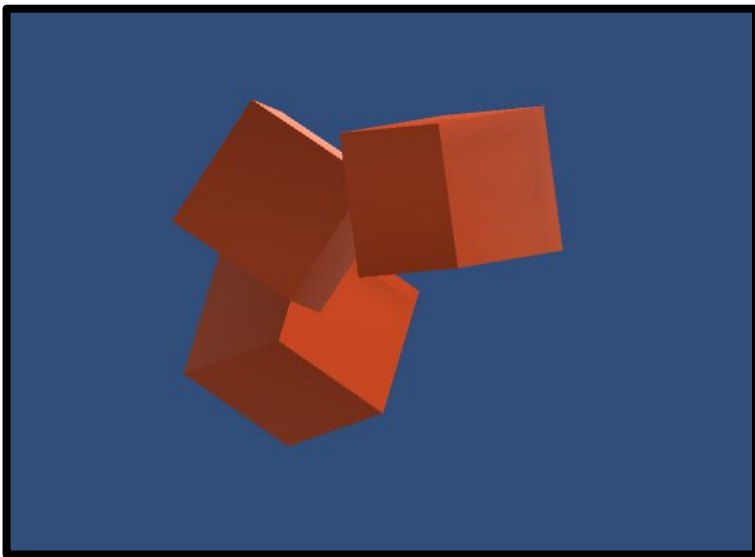
- **Iluminación:** La luz es esencial para determinar cómo se ven los objetos. Unity ofrece varios tipos de luces: direccional, puntual, spot y área. Las sombras, reflejos y otros efectos también están determinados por la configuración de las luces y las propiedades de los materiales de los objetos.
- **Materiales y Shaders:** Los materiales definen cómo se ve la superficie de un objeto 3D, incluyendo color, brillo, reflectividad, entre otros. Están asociados con "shaders", que son programas que determinan cómo se renderiza el material bajo distintas condiciones de luz.
- **Físicas:** Unity incorpora un motor de físicas 3D mediante el uso de componentes sobre los objetos, entre ellos podemos destacar el "**Rigidbody**" y el "**Collider**".
 - **Rigidbody:** añade físicas a un objeto en Unity. Permite que el objeto sea afectado por la gravedad, fuerzas y colisiones.
 - **Collider:** Es una envoltura invisible que define el área de un objeto donde puede ocurrir una colisión. Puede tener varias formas (esfera, caja, malla, etc.) y se usa en combinación con un Rigidbody para detectar y responder a colisiones en el juego.

El espacio 3D en Unity

- **Cámaras:** La cámara es esencial en un espacio 3D, ya que determina desde qué perspectiva ve el jugador el juego. Unity proporciona una cámara por defecto, pero puedes añadir más y configurar propiedades como el tipo de proyección (ortográfica o perspectiva), ángulo de visión, etc. (lo desarrollaremos más adelante)



PERSPECTIVA



ORTOGRÁFICA

Introducción en Unity: Clase MonoBehaviour

La clase MonoBehaviour: Métodos principales

MonoBehaviour es una clase base de la que heredan todos los scripts en Unity. A través de esta clase accedemos a la mayoría de las funciones y eventos que ofrece Unity.

- **Start():** Es un método que se llama automáticamente cuando se inicia un script, justo antes del primer frame de juego. Es ideal para inicializar variables, establecer referencias a otros componentes o hacer configuraciones iniciales.
- **Update():** **Update es llamado una vez por frame**, su frecuencia puede variar según el rendimiento de la máquina donde se ejecuta el juego. Es ideal para actualizar lógica de juego, movimientos, comprobar entradas (inputs) del usuario, etc.
- **FixedUpdate():** **Se llama en intervalos de tiempo fijos**, independientemente del frame rate. Es ideal para códigos relacionados con físicas, como aplicar fuerzas a un Rigidbody.

```
public class Ejemplo : MonoBehaviour {  
    void Start() { ... }  
    void Update() { ... }  
    void FixedUpdate() { ... }  
}
```

La clase MonoBehaviour: método FixedUpdate

Por defecto, FixedUpdate está configurado a 0.02, lo que significa que FixedUpdate se ejecuta 50 veces por segundo (cada 20 milisegundos). Si ajustas este valor a, por ejemplo, 0.01, entonces FixedUpdate se ejecutará 100 veces por segundo (cada 10 milisegundos).

- **Rendimiento:** Aunque puedes reducir el valor de "Fixed Timestep" para que FixedUpdate se ejecute con mayor frecuencia, esto puede aumentar la carga en la CPU, especialmente si tienes muchos objetos con físicas en la escena. Es importante hacer pruebas y monitorizar el rendimiento al ajustar este valor.
- **Precisión de la física:** Un "Fixed Timestep" más pequeño puede ofrecer una simulación física más precisa, especialmente en situaciones de rápido movimiento. Sin embargo, en muchos casos, el valor predeterminado es suficiente.

No debes poner lógica no relacionada con la física dentro de este método. Específicamente, la lógica de entrada del usuario y otras acciones no relacionadas con la física generalmente deben ir en Update. Para establecer la duración del FixedUpdate de forma manual:

Abrir Unity -> Edit -> Project Settings -> Time -> Fixed Timestep

La clase MonoBehaviour: deltaTime

El **deltaTime** es una herramienta esencial en Unity, y su comprensión es crucial para programar comportamientos que dependen del tiempo. En Unity, **Time.deltaTime** representa el tiempo que ha transcurrido entre el último y el actual fotograma (o frame). Cuando hablamos de videojuegos, estos funcionan con FPS, la pantalla se refresca muchas veces por segundo, y cada refresco muestra un "fotograma" del juego.

No todos los dispositivos ni todas las situaciones harán que tu juego se ejecute exactamente al mismo ritmo. Puede que un dispositivo lo ejecute a 60 fotogramas por segundo (fps) y otro a 30 fps. Si no haces uso de Time.deltaTime, las acciones en tu juego serían más rápidas en dispositivos con más fps y más lentas en dispositivos con menos fps.

Utilizar Time.deltaTime ayuda a normalizar esta diferencia. Al multiplicar una velocidad o movimiento por Time.deltaTime, estás esencialmente diciendo: "Quiero que esta acción ocurra a esta velocidad por segundo y no por fotograma". De este modo, independientemente de los fps, la acción tendrá lugar a un ritmo constante.

```
//mueve el objeto a 3 unidades por segundo hacia la derecha  
transform.position += Vector3.right * 3 * Time.deltaTime;
```

Ejemplos Clase MonoBehaviour

```
public float velocidad = 5f;
public GameObject cubo1;
public GameObject cubo2;
public GameObject cubo3;

void Update()
{
    // rotamos los cubos 1 y 2
    cubo1.transform.Rotate(0, velocidad*Time.deltaTime, 0);
    cubo2.transform.Rotate(0, velocidad*Time.deltaTime, 0);
}

void FixedUpdate()
{
    // rotamos cubo3
    cubo3.transform.Rotate(0, velocidad*Time.deltaTime, 0);
}
```

La clase MonoBehaviour: Métodos adicionales

- **Awake():** se invoca cuando se crea una instancia de script. Se ejecuta antes que `Start()`. Se suele utilizar para Inicializar variables o configuraciones que deben establecerse antes de cualquier otro método.
- **OnEnable() y OnDisable():** se llaman cuando un script es inicializado, justo después de `Awake()` y también cuando un objeto se activa. `OnDisable` se llama cuando el objeto se desactiva. Se utiliza para activar o desactivar funciones específicas, suscribirse o cancelar suscripciones a eventos.
- **LateUpdate():** Se llama después de todos los métodos `Update()` en un frame determinado. Útil para operaciones que deben ocurrir después de que se hayan procesado todos los movimientos o cálculos regulares, como ajustar la posición de una cámara siguiendo a un personaje.
- **OnDestroy():** Se invoca cuando un objeto `MonoBehaviour` va a ser destruido. Utilizado para limpiar referencias, cancelar suscripciones a eventos, liberar recursos, etc.

La clase MonoBehaviour: Métodos adicionales

- **OnCollisionEnter(), OnCollisionExit(), y OnCollisionStay():** se invocan cuando ocurre una colisión con un objeto que tiene un componente `Collider`. Se utiliza para detectar colisiones y llevar a cabo acciones específicas, como causar daño o activar eventos.
- **OnTriggerEnter(), OnTriggerExit(), y OnTriggerStay():** Similar a los eventos de colisión, pero para `Colliders` configurados como "triggers" (desencadenadores) en lugar de colisionadores sólidos. Un caso de uso puede ser, crear zonas donde ocurren eventos específicos sin una colisión física, como una zona de detección de enemigos.
- **OnMouseDown(), OnMouseUp(), OnMouseOver():** Métodos relacionados con eventos del ratón. Múltiples usos, detectar interacciones del usuario con objetos en la escena, seleccionar objetos, elegir opciones de la UI, pulsar botones, etc.
- **OnApplicationQuit():** Se invoca cuando la aplicación está a punto de cerrarse. Utilizado en muchas ocasiones para guardar datos o realizar acciones de limpieza final.

La clase MonoBehaviour: Métodos adicionales

- **Awake()**
- **OnEnable() y OnDisable()**
- **LateUpdate()**
- **OnDestroy()**
- **OnCollisionEnter(), OnCollisionExit(), y OnCollisionStay()**
- **OnTriggerEnter(), OnTriggerExit(), y OnTriggerStay()**
- **OnMouseDown(), OnMouseUp(), OnMouseOver()**
- **OnApplicationQuit()**

Cada uno de los métodos serán desarrollados más adelante. De momento con esto terminamos una pequeña introducción a Unity y ya seguiremos en el siguiente tema. A partir de ahora pasaremos a la parte de C# utilizando ejemplos dentro de Unity.

Introducción a C# en Unity:

Tipos de datos: enteros y reales

Tipos de Datos Primitivos en C# para Unity

Números enteros

- **byte**: Es un tipo de dato de 8 bits sin signo.
- **sbyte**: Es un tipo de dato de 8 bits con signo.
- **short**: Es un tipo de dato de 16 bits con signo.
- **ushort**: Es un tipo de dato de 16 bits sin signo.
- **int**: Es un tipo de dato de 32 bits con signo.
- **uint**: Es un tipo de dato de 32 bits sin signo.
- **long**: Es un tipo de dato de 64 bits con signo.
- **ulong**: Es un tipo de dato de 64 bits sin signo.

Números de coma flotante

- **float**: Es un tipo de dato de 32 bits de precisión simple. Es útil para almacenar valores decimales como posiciones o velocidades en Unity (entre 6 y 9 decimales de precisión)
- **double**: Es un tipo de dato de 64 bits de precisión doble. Es menos común en Unity, pero puede ser útil para cálculos que requieran una alta precisión (entre 14 y 17 decimales de precisión)

Tipos de Datos Primitivos en C# para Unity

Caracteres y cadenas

- **char**: Representa un carácter único y ocupa 16 bits. Se utiliza para representar caracteres Unicode. Su valor por defecto es el carácter nulo ('\0')
- **string**: Representa una secuencia de caracteres. Su valor por defecto es null.

Booleanos

- **bool**: Puede tener uno de dos valores, true o false. Es muy útil para controlar lógicas condicionales. Su valor por defecto es 'false'.

Otros

- **decimal**: Tipo de 128 bits con 28-29 dígitos significativos. Es menos común en Unity, pero puede ser útil para cálculos financieros o aquellos que requieran una alta precisión.

Ejemplos con tipos de datos en C#

```
int num1 = 1, num2 = 2;
float num3 = 0.5f, num4 = 3f;
Debug.Log(num1+num2); // 3
Debug.Log(num3+num4); // 3,5
```

```
int numeroMaximo = int.MaxValue; // 2147483647
int numeroMinimo = int.MinValue; //-2147483648
uint numeroMaximoU = uint.MaxValue; // 4294967295
uint numeroMinimoU = uint.MinValue; //0
```

```
int max = numeroMaximo + 1; // causará un overflow
int min = numeroMinimo - 1; // causará un underflow
uint maxU = numeroMaximoU + 1; // causará un overflow
uint minU = numeroMinimoU - 1; // causará un underflow
```

```
float floatNumero = 1.123456789f; // 6-9 dígitos de precisión aprox
double doubleNumero = 1.1234567890123456; // 14-17 dígitos de precisión aprox
Debug.Log("Número float: " + floatNumero); // 1.123457
Debug.Log("Número double: " + doubleNumero); // 1.12345678901235
```

Introducción a C# en Unity:

Tipos de datos: string

El tipo string

En C#, string no es un tipo primitivo en el sentido estricto de la palabra. Es similar a Java en este aspecto. En C#, string es una clase, lo que significa que es un tipo de referencia y no un tipo de valor como los tipos primitivos. Sin embargo, debido a su uso frecuente y la manera en que se maneja en el lenguaje, muchas veces se le trata o se le menciona junto a los tipos primitivos. Algunos puntos clave sobre string en C#:

- **Inmutabilidad:** Las cadenas en C# son inmutables, lo que significa que una vez que se crea una instancia de una cadena, no se puede modificar. Cualquier operación que parezca modificar una cadena en realidad está creando una nueva instancia.
- **Tipo de referencia:** Aunque string es un tipo de referencia, se comporta de manera diferente a otros tipos de referencia. Por ejemplo, el operador == compara el valor de las cadenas en lugar de sus referencias, el operador está sobrecargado.

```
string saludo1 = "Hola";  
string saludo2 = "Hola";  
if (saludo1 == saludo2) Debug.Log("Son iguales");  
saludo2 = saludo1;  
saludo1 = "Hola1"; //saludo1 → "Hola1" y saludo2 → "Hola"
```

Ejemplos para concatenar strings en C#

A continuación se muestran una serie de ejemplos de concatenación de strings. Podemos usar interpolación de strings(\$"...{variable}") para insertar variables directamente en la cadena.

```
int num1 = 1, num2 = 2;
float num3 = 0.5f, num4 = 2f, precio = 5.4567f;
string frase1 = "Hola";
string frase2 = "Mundo";
string resultado = num1+num2; //error
resultado = (num1+num2).ToString(); //3
resultado = num1+num2+frase1+frase2; //3HolaMundo
resultado = frase1+" "+frase2+num1+num2; //Hola Mundo12
resultado = frase1+" "+frase2+(num1+num2); //Hola Mundo3
resultado = $"{frase1} {frase2}{num1+num2}"; //Hola Mundo3
resultado = $"{frase1} {frase2}{num1}{num2}"; //Hola Mundo12
resultado = num3+num4; //error
resultado = $"{num3+num4}"; //2,5
resultado = frase1+frase2+"\n"+num3+num4; //HolaMundo
//0,52
resultado = $"El precio es {precio:F2}"; //El precio es 5,46
```

Métodos para Manipulación de Cadenas

```
string concatenada = string.Concat("Hola", " ", "Mundo"); //Hola Mundo
string enMinuscula = "aulaenlanube.com".ToLower(); //aulaenlanube.com
string enMayuscula = "aulaenlanube.com".ToUpper(); //AULAENLANUBE.COM
string sinEspacios = " aula en la nube ".Trim(); //aula en la nube
string subCadena = "aula en la nube".Substring(5, 5); //en la
string reemplazada = "aula en la nube".Replace("aula", "Unity"); //Unity en la nube
int longitud = "aulaenlanube.com".Length; //16
int indice = "aula en la nube".IndexOf("en"); //5
int indice2 = "aula en la nube".IndexOf("e"); //5
int indice3 = "aula en la nube".IndexOf("E"); //-1
bool comienzaCon = "aulaenlanube".StartsWith("aula"); //True
bool contiene = "aulaenlanube".Contains("en"); //True

string[] partes = "aula en la nube".Split(' '); // ["aula", "en", "la", "nube"]
char[] caracteres = "aula".ToCharArray(); // ['a', 'u', 'l', 'a']

string caracteresUnidos1 = new string(caracteres); //aula
string caracteresUnidos2 = string.Join(" ", caracteres); //a u l a
string partesUnidas = string.Join(" ", partes); //aula en la nube
int resultado = string.Compare(caracteresUnidos1 , caracteresUnidos2); //mayor que 0
```


Introducción a C# en Unity: Operadores

Operadores en C#

Podemos utilizar las distintas operaciones matemáticas para utilizarlas en expresiones e incluso combinarlas con variables.

Operador	Operación	Operador	Operación
+	Suma o signo	+=	Suma y asignación
-	Resta o signo	-=	Resta y asignación
*	Multiplicación	*=	Multiplicación y asignación
/	División	/=	División y asignación
%	Módulo	%=	Módulo y asignación
++	Incremento en 1	--	Decremento en 1

```
int precio1 = 5, precio2 = 10;
int coste = precio1*precio2*2;           //100
int media1 = precio1+precio2/2;          //10
int media2 = (precio1+precio2)/2;         //7
float media3 = (precio1+precio2)/2;       //7
float media4 = (precio1+precio2)/2f;      //7,5
```

Incrementos y decrementos en C#

Los operadores de incremento y de decremento se pueden escribir antes o después de la variable. Así, es lo mismo escribir estas dos líneas:

```
a++;  
++a;
```

Pero hay una diferencia si ese valor se asigna a otra variable "al mismo tiempo" que se incrementa/decrementa:

```
int c = 5;  
int b = c++;
```

Guarda como resultado c = 6 y b = 5, se asigna el valor a "b" **antes** de incrementar "c".

Sin embargo:

```
int c = 5;  
int b = ++c;
```

Guarda como resultado c = 6 y b = 6, se asigna el valor a "b" **después** de incrementar "c").

Operadores relacionales en C#

En C# también tenemos las expresiones lógicas. En este tipo de expresiones el resultado será de tipo booleano. Es decir, verdadero o falso. Dichas expresiones se pueden utilizar para evaluar condiciones. También se puede asignar su resultado a variables de tipo bool.

Operador	Operación
==	Igual
!=	Distinto
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que

```
int precio1 = 10, precio2 = 30;
precio1 += precio2;
precio2 -= precio1*-2;
bool barato = precio1 > precio2;           //False
bool iguales1 = 150 == (precio1 + precio2); //True
bool iguales2 = 150 == precio1 + precio2;  //True
bool distintos = precio1 != precio2;       //True
bool iguales2 = (40 == precio1) + precio2; //ERROR
```

Operadores lógicos en C#

Podemos combinar expresiones lógicas a través de operadores lógicos:

Operador	Operación	Significado
!	NOT	Negación lógica
&&	AND	Conjunción lógica (Y lógico)
	OR	Disyunción (O lógico)
^	XOR	Disyunción Exclusiva (XOR)

```
int a = 1, b = 2;
bool condicion1 = true, condicion2 = false, resultado;
resultado = condicion1 && condicion2;           //False
resultado = condicion1 || condicion2;           //True
resultado = condicion1 && a < b;                 //True
resultado = condicion2 && a < b;                 //False
resultado = condicion1 && !condicion2;           //True
resultado = !(condicion1 && !condicion2);        //False
resultado = !(condicion1 && condicion2);         //True
resultado = condicion1 ^ condicion2;            //True
resultado = condicion1 ^ a != b;                //False
resultado = !condicion1 ^ !(a != b);            //False
resultado = !(condicion1 ^ a != b);             //True
```

Precedencia de operadores en C#

La precedencia de operadores es un tema crucial en programación, ya que determina el orden en el que se evalúan las operaciones en una expresión. Entender la precedencia ayuda a evitar errores sutiles y a escribir código más eficiente y legible.

En la siguiente tabla se muestra la precedencia de los distintos operadores separada en grupos de más prioridad (grupo 1), a menos nivel de prioridad (grupo 13).

Grupo 1	()
Grupo 2	+ - ++ -- !
Grupo 3	* / %
Grupo 4	+ -
Grupo 5	<< >>
Grupo 6	> >= < <=
Grupo 7	== !=
Grupo 8	& ^
Grupo 9	&&
Grupo 10	^
Grupo 11	
Grupo 12	? :
Grupo 13	= += -= *= /= %= &= = ^= <<= >>= >>=

Ejercicio precedencia de operadores en C#

```
int a = 10, b = 20, c = 5, d = 2;  
bool condicion = true;  
int resultado = a + b * c / d - (condicion ? a : b) + (b > a ? c : d) + (a + b) /  
(c + 1) - (d > 1 ? (c * d + a) : (b / d + a));
```

Ejercicio precedencia de operadores en C#

```
int a = 10, b = 20, c = 5, d = 2;  
bool condicion = true;  
int resultado = a + b * c / d - (condicion ? a : b) + (b > a ? c : d) + (a + b) /  
(c + 1) - (d > 1 ? (c * d + a) : (b / d + a));
```

```
// 1. b * c = 20 * 5 → 100
```

```
// 2. 100 / d = 100 / 2 → 50
```

```
// 3. a + 50 = 10 + 50 → 60
```

```
// 4. (condicion ? a : b) = (true ? 10 : 20) → 10
```

```
// 5. 60 - 10 = 50
```

```
// 6. (b > a ? c : d) = (20 > 10 ? 5 : 2) → 5
```

```
// 7. 50 + 5 = 55
```

```
// 8. (a + b) / (c + 1) = (10 + 20) / (5 + 1) → 30 / 6 → 5
```

```
// 9. 55 + 5 = 60
```

```
// 10. (d > 1 ? (c * d + a) : (b / d + a)) = 2 > 1 → (5 * 2 + 10) → 20
```

```
// 11. 60 - 20 = 40
```

```
// el resultado final es 40
```


Introducción a C# en Unity:

Estructuras condicionales

Estructuras condicionales en C#

Al igual que otros lenguajes de programación, para evaluar condiciones, C# dispone de varios tipos de estructuras condicionales.

- **if:** es la estructura más básica. Evalúa si una condición es verdadera y, si lo es, ejecuta el bloque de código siguiente.
- **if-else:** permite especificar un bloque de código que se ejecutará si la condición es falsa.
- **if-else if-else:** permite verificar múltiples condiciones.
- **switch-case:** para situaciones donde hay muchas condiciones que pueden ser mutuamente excluyentes, el switch-case es más limpio y legible.
- **Operador ternario:** es una forma más corta de escribir una declaración if-else. Es especialmente útil para asignaciones condicionales.

```
if (vida < 0)
{
    Debug.Log("El personaje ha muerto");
}
```

IF

Estructuras condicionales en C#

IF - ELSE

```
int vida = 50;  
string estado;  
if (vida > 0)  
{  
    estado = "Vivo";  
}  
else  
{  
    estado = "Muerto";  
}
```

TERNARIA

```
int vida = 50;  
string estado = (vida > 0) ? "Vivo" : "Muerto";
```

IF - ELSE

```
int vida = 50;  
string estado;  
if (vida == 100)  
{  
    estado = "Max";  
}  
else if (vida > 0)  
{  
    estado = "Vivo";  
}  
else  
{  
    estado = "Muerto";  
}
```

TERNARIA

```
int vida = 50;  
string estado = (vida == 100) ? "Max" : (vida > 0) ? "Vivo" : "Muerto";
```

Estructuras condicionales en C#

```
string objeto = recogerObjeto();
```

IF - ELSE

```
if (objeto == "comida")
{
    Debug.Log("Has encontrado comida");
}
else if (objeto == "llave")
{
    Debug.Log("Has encontrado una llave");
}
else if (objeto == "gema")
{
    Debug.Log("Has encontrado una gema");
}
else if (objeto == "moneda")
{
    Debug.Log("Has encontrado una moneda");
}
else
{
    Debug.Log("Has encontrado un objeto extraño");
}
```

```
string objeto = recogerObjeto();
```

SWITCH

```
switch (objeto)
{
    case "comida":
        Debug.Log("Has encontrado comida");
        break;
    case "llave":
        Debug.Log("Has encontrado una llave");
        break;
    case "gema":
        Debug.Log("Has encontrado una gema");
        break;
    case "moneda":
        Debug.Log("Has encontrado una moneda");
        break;
    default:
        Debug.Log("Has encontrado un objeto extraño");
        break;
}
```

Estructuras condicionales en C#

SWITCH

```
string objeto = recogerObjeto();

switch (objeto)
{
    case "comida":
        Debug.Log("Has encontrado comida");
        break;
    case "llave":
        Debug.Log("Has encontrado una llave");
        break;
    case "gema":
        Debug.Log("Has encontrado una gema");
        break;
    case "moneda":
        Debug.Log("Has encontrado una moneda");
        break;
    default:
        Debug.Log("Has encontrado un objeto extraño");
        break;
}
```

SWITCH ABREVIADO (C# 8.0)

```
string objeto = recogerObjeto();

string mensaje = objeto switch
{
    "comida" => "Has encontrado comida",
    "llave" => "Has encontrado una llave",
    "gema" => "Has encontrado una gema",
    "moneda" => "Has encontrado una moneda",
    _ => "Has encontrado un objeto extraño"
};

Debug.Log(mensaje);
```

Estructuras condicionales en C#

```
string objeto = recogerObjeto();
```

SWITCH

```
switch (objeto)
{
    case "comida":
        comida++;
        agregarComida();
        break;
    case "llave":
        llave++;
        agregarLlave();
        break;
    case "moneda":
        moneda++;
        agregarMoneda();
        break;
    default:
        Debug.Log("Objeto extraño");
        break;
}
```

SWITCH ABREVIADO (C# 8.0)

```
string objeto = recogerObjeto();

Action accion = objeto switch
{
    "comida" => () => { comida++; agregarComida(); },
    "llave" => () => { llave++; agregarLlave(); },
    "moneda" => () => { moneda++; agregarMoneda(); },
    _ => () => Debug.Log("Objeto extraño")
};

accion();
```

Estructuras condicionales en C#

Operador condicional de acceso nulo [?.] Este operador permite acceder a miembros de un objeto (como propiedades o métodos) sólo si el objeto no es nulo; de lo contrario, devuelve null.

```
if (personaje != null)
{
    if(personaje.getTipo() == MAGO) return true;
}
```

```
if(personaje?.getTipo() == MAGO) return true;
```

Operador de coalescencia nula [??] Este operador devuelve el valor de su operando izquierdo si ese operando no es nulo; de lo contrario, devuelve el valor del operando derecho.

```
string nombre;
Persona p = new Persona();
if (p.Nombre != null)
{
    nombre = p.Nombre;
}
else
{
    nombre = "Pepe";
}
```

```
string nombre = p.Nombre ?? "Pepe"; //Pepe
```

```
string nombre = p.Nombre?.ToUpper() ?? "Pepe"; //Pepe
p.Nombre = "Sam";
nombre = p.Nombre ?? "Pepe"; //Sam
nombre = p.Nombre?.ToUpper() ?? "Pepe"; //SAM
```

Estructuras condicionales en C#

Operador de coalescencia nula con asignación [??=] Este operador asigna el valor de su operando derecho a su operando izquierdo solo si el operando izquierdo es nulo.

```
objeto ??= new Objeto();
```

```
public class Arma
{
    public string Nombre { get; set; }
    public int Poder { get; set; }
    // otros atributos y métodos
}

public class Personaje
{
    public Arma ArmaEquipada { get; set; }
    // otros atributos y métodos
}
```

```
public class ControladorJuego : MonoBehaviour
{
    public Personaje p;
    void Start()
    {
        // inicializar personaje, escena, etc.
        // aseguramos que el personaje tenga arma
        AsegurarArmaPersonaje();
    }
    void AsegurarArmaPersonaje()
    {
        p.ArmaEquipada ??= new Arma
        {
            Nombre = "Espada de Madera",
            Poder = 10
        };
        Debug.Log($"Arma:{p.ArmaEquipada.Nombre}");
    }
}
```


Ejemplos condicionales

```
public float velocidad = 5f;

void Update()
{
    float movimiento = velocidad * Time.deltaTime;
    // mueve el cubo de izquierda a derecha y viceversa al llegar a un límite
    transform.position += new Vector3(movimiento, 0, 0);
    if (transform.position.x > 5 || transform.position.x < -5)
    {
        velocidad = -velocidad; // invertimos la dirección
        //cambiamos el color del material
        GetComponent<Renderer>().material.color = Random.ColorHSV();
    }
}
```

Ejemplos condicionales

```
void Update()
{
    if (transform.position.y > 5)
    {
        GetComponent<Renderer>().material.color = Color.green;
    }
    else if (transform.position.y < -5)
    {
        GetComponent<Renderer>().material.color = Color.red;
    }
    else
    {
        GetComponent<Renderer>().material.color = Color.blue;
    }
}
```

```
void Update()
{
    GetComponent<Renderer>().material.color = transform.position.y switch
    {
        > 5 => Color.green,
        < -5 => Color.red,
        _ => Color.blue
    };
}
```

Introducción a C# en Unity: Ejercicios con condicionales

Ejercicios estructuras condicionales en C#

Ejercicio: Patrón de Movimiento Cuadrado en Unity

El objetivo de este ejercicio es programar un patrón de movimiento cuadrado para un GameObject en Unity. El GameObject deberá moverse en un cuadrado en el plano XZ, empezando desde un punto inicial y volviendo a él al completar el ciclo. Requisitos:

- Crea un nuevo proyecto de Unity y coloca un cubo en la escena.
- Adjunta un nuevo script C# al GameObject.
- En este script, implementa la lógica para mover el GameObject en un patrón de cuadrado.
- El movimiento deberá ser continuo y el GameObject deberá volver a su posición inicial después de completar un ciclo.
- Debes utilizar condicionales y la sentencia switch para controlar el cambio de dirección del GameObject.
- No se permite el uso de arrays ni ningún otro tipo de colecciones para almacenar los puntos del triángulo.
- Incluye variables para controlar la velocidad y la longitud del lado del cuadrado. Estas variables deben ser públicas para poder ajustarlas desde el editor de Unity.

Ejercicios estructuras condicionales en C#

```
public float velocidad = 5f, lado = 10f;
private const int ARRIBA = 0, ABAJO = 1, IZQUIERDA = 2, DERECHA = 3;
private Vector3 inicio, pos;
private int direccion = DERECHA;
void Start() { inicio = transform.position; }
void Update()
{
    Vector3 mov = direccion switch
    {
        DERECHA    => Vector3.right,    //new Vector3(1, 0, 0)
        ARRIBA     => Vector3.forward,  //new Vector3(0, 0, 1)
        IZQUIERDA  => Vector3.left,     //new Vector3(-1, 0, 0)
        ABAJO      => Vector3.back,     //new Vector3(0, 0, -1)
        -          => Vector3.zero     //new Vector3(0, 0, 0)
    };
    transform.position += mov * velocidad * Time.deltaTime;
    pos = transform.position;

    if (direccion == DERECHA    && pos.x >= inicio.x + lado)
    else if (direccion == ARRIBA && pos.z >= inicio.z + lado)
    else if (direccion == IZQUIERDA && pos.x <= inicio.x)
    else if (direccion == ABAJO  && pos.z <= inicio.z)
    }
}
```

Patrón de movimiento en forma de cuadrado sobre los ejes X-Z:

```
direccion = ARRIBA;
direccion = IZQUIERDA;
direccion = ABAJO;
direccion = DERECHA;
```

Ejercicios estructuras condicionales en C#

Ejercicio: Patrón de Movimiento Triangular en Unity

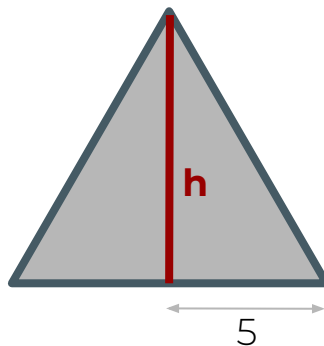
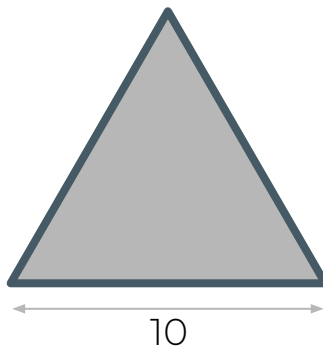
El objetivo de este ejercicio es programar un patrón de movimiento triangular para un GameObject en Unity. El GameObject deberá moverse en un triángulo equilátero en el plano XZ, empezando desde un punto inicial y volviendo a él al completar el ciclo. Requisitos:

- Crea un nuevo proyecto de Unity y coloca un cubo en la escena.
- Adjunta un nuevo script C# al GameObject.
- En este script, implementa la lógica para mover el GameObject en un patrón de triángulo equilátero.
- El movimiento deberá ser continuo y el GameObject deberá volver a su posición inicial después de completar un ciclo.
- Debes utilizar condicionales y la sentencia switch para controlar el cambio de dirección del GameObject.
- No se permite el uso de arrays ni ningún otro tipo de colecciones para almacenar los puntos del triángulo.
- Incluye variables para controlar la velocidad y la longitud del lado del triángulo. Estas variables deben ser públicas para poder ajustarlas desde el editor de Unity.

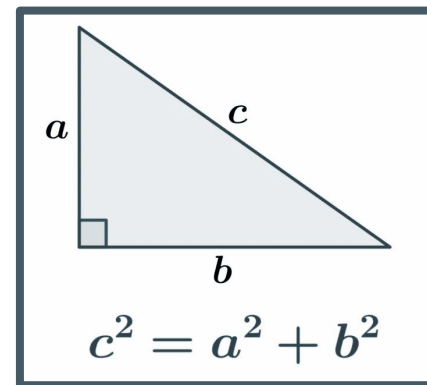
Ejercicios estructuras condicionales en C#

Ejercicio: Patrón de Movimiento Triangular en Unity

- Obtener los puntos de un triángulo equilátero de lado 10.



Teorema de pitágoras



$$h^2 = l^2 - \left(\frac{l}{2}\right)^2 = \frac{3l^2}{4}$$

$$h = \sqrt{\frac{3l^2}{4}} = \frac{\sqrt{3}}{2}l$$

```
float lado = 10f;  
float h = Mathf.Sqrt(3) * lado / 2;
```

Ejercicios estructuras condicionales en C#

Ejercicio: Patrón de Movimiento Triangular en Unity

- **Distancia** euclidiana entre 2 puntos 3D.

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

```
Vector3 p1 = new Vector3(0, 0, 0);  
Vector3 p2 = new Vector3(1, 1, 1);  
float distancia1 = Vector3.Distance(p1, p2); //1.732051  
float distancia2 = Vector3.Distance(p2, p1); //1.732051
```

- **Dirección de un vector:** se refiere a la orientación en el espacio en la que apunta el vector. Esto se hace restando el vector de la posición inicial del vector de la posición final. El resultado es un nuevo vector que apunta desde el punto inicial al punto final.

```
Vector3 direccion = punto1 - punto2;
```


Ejercicios estructuras condicionales en C#

Ejercicio: Patrón de Movimiento Triangular en Unity

- Tal y como se ha dicho, la dirección de un vector se refiere a la orientación en el espacio en la que apunta el vector. Múltiples vectores pueden tener la misma dirección pero diferentes magnitudes. La **magnitud** de un vector es el “tamaño” o “longitud” del vector. Por ejemplo, los vectores (1,0) y (2,0) apuntan en la misma dirección (a lo largo del eje x) pero tienen diferentes longitudes (magnitudes). Si normalizamos estos vectores, ambos se convierten en (1,0), manteniendo la dirección pero ajustando la **magnitud** a 1.

```
Vector3 miVector = new Vector3(3, 4, 0);  
float magnitud = miVector.magnitude; // la magnitud será 5
```

- Un vector **normalizado** es un vector que tiene la misma dirección que el vector original, pero cuya longitud o magnitud es 1. Es muy útil cuando se quiere aplicar una velocidad constante al objeto, independientemente de la distancia hasta el objetivo.

```
new Vector3(1,0,0).normalized; // (1,0,0)  
new Vector3(2,0,0).normalized; // (1,0,0)  
new Vector3(100,0,0).normalized; // (1,0,0)
```

$$\mathbf{N} = \frac{\mathbf{V}}{\|\mathbf{V}\|} \quad \mathbf{V} = (x, y, z)$$
$$\sqrt{x^2 + y^2 + z^2}$$

Ejercicios estructuras condicionales en C#

Patrón de movimiento en forma de triángulo equilátero en X-Z:

```
public float velocidad = 5f, lado = 10f;
private Vector3 inicio,puntoObjetivo;
private const int DERECHA = 0, ARRIBA_IZQUIERDA = 1, ABAJO_IZQUIERDA = 2; // constantes para dirección
private int direccion = DERECHA;
void Start()
{
    inicio = transform.position;
    puntoObjetivo = inicio + new Vector3(lado, 0, 0); // se inicia con movimiento hacia la derecha
}
void Update()
{
    Vector3 movimiento = (puntoObjetivo - transform.position).normalized * velocidad * Time.deltaTime;
    if (Vector3.Distance(transform.position, puntoObjetivo) > velocidad * Time.deltaTime)
        transform.Translate(movimiento);
    else
    {
        transform.position = puntoObjetivo;
        switch (direccion)
        {
            case DERECHA:
                puntoObjetivo = transform.position + new Vector3(-lado / 2, 0, Mathf.Sqrt(3) * lado / 2);
                direccion = ARRIBA_IZQUIERDA; break;
            case ARRIBA_IZQUIERDA:
                puntoObjetivo = transform.position + new Vector3(-lado / 2, 0, -Mathf.Sqrt(3) * lado / 2);
                direccion = ABAJO_IZQUIERDA; break;
            case ABAJO_IZQUIERDA:
                puntoObjetivo = transform.position + new Vector3(lado, 0, 0);
                direccion = DERECHA; break;
        }
    }
}
```

Ejercicios estructuras condicionales en C#

Ejercicio: Seguimiento y Evasión de un Objetivo en Unity

Desarrollar un script en C# para Unity que permita a un GameObject seguir o evitar a otro GameObject llamado "objetivo" dependiendo de la distancia entre ambos. Requisitos:

- Crea un nuevo proyecto de Unity y coloca dos cubos en la escena, un cubo lo moveremos de forma manual, y el otro deberá seguirlo.
- Adjunta un nuevo script C# al GameObject que debe seguir al cubo. En dicho script, implementa la lógica para seguir al cubo objetivo.
- Debes utilizar condicionales y la sentencia switch para controlar el seguimiento.
- Se debe permitir una distancia máxima para empezar a seguir al objetivo, si la distancia entre ambos cubos es superior, no deberá hacer ningún seguimiento.
- Se debe permitir una distancia mínima para empezar a evadir al objetivo, si la distancia entre ambos cubos es mínima, se deberá detener el seguimiento.
- No se permite el uso de arrays ni ningún otro tipo de colecciones.
- Incluye variables para controlar la distancia de seguimiento y la distancia de evasión. Estas variables deben ser públicas para poder ajustarlas desde el editor de Unity.

Ejercicios estructuras condicionales en C#

Seguimiento y Evasión: el GameObject asociado al Script seguirá al GameObject “objetivo” si está a una cierta distancia, y lo evitará si está demasiado cerca.

```
public GameObject objetivo;  
public float velocidad = 2f;  
public float distanciaSeguimiento = 5f;  
public float distanciaEvasion = 2f;
```

```
void Update()  
{  
    float distanciaObjetivo = Vector3.Distance(transform.position, objetivo.transform.position);  
  
    // condicional para seguir o evitar al objetivo  
    if (distanciaObjetivo < distanciaEvasion)  
    {  
        // evitar al objetivo  
        Vector3 direccion = (transform.position - objetivo.transform.position).normalized;  
        transform.position += direccion * velocidad * Time.deltaTime;  
        GetComponent<Renderer>().material.color = Color.blue;  
    }  
    else if (distanciaObjetivo < distanciaSeguimiento)  
    {  
        // seguir al objetivo  
        Vector3 direccion = (objetivo.transform.position - transform.position).normalized;  
        transform.position += direccion * velocidad * Time.deltaTime;  
        GetComponent<Renderer>().material.color = Color.red;  
    }  
}
```

Introducción a C# en Unity:

Estructuras repetitivas

Estructuras repetitivas en C#: bucles

En C# tenemos los siguientes tipos de estructuras repetitivas: **for**, **while**, **do-while** y **foreach**.

```
for (int contador = 0; contador < 10; contador++)  
{  
    Debug.Log($"contador vale: {contador}");  
}
```

FOR

```
int contador = 0;  
while (contador < 10)  
{  
    Debug.Log($"contador vale: {contador}");  
    contador++;  
}
```

WHILE

```
int contador = 0;  
do  
{  
    Debug.Log($"contador vale: {contador}");  
    contador++;  
} while (contador < 10);
```

DO-WHILE

```
string[] nombres = {"Pep", "Jon", "Tom"};  
foreach (string nombre in nombres)  
{  
    Debug.Log($"El nombre es: {nombre}");  
}
```

FOREACH

Ejemplos estructuras repetitivas en C#: bucles

Bucle FOR: script que utiliza un bucle for para crear una línea de 10 cubos de tamaño 1 en el eje X separados por 0.2 unidades.

```
for (int i = 0; i < 10; i++)  
{  
    Vector3 posicionNueva = new Vector3(i * 1.2f, 0, 0);  
    GameObject nuevoCubo = GameObject.CreatePrimitive(PrimitiveType.Cube);  
    nuevoCubo.transform.position = posicionNueva;  
}
```

FOR

Bucle WHILE: script que utiliza un bucle while para crear una línea de 10 cubos de tamaño 1 en el eje X separados por 0.2 unidades, alternando entre 2 colores para los cubos.

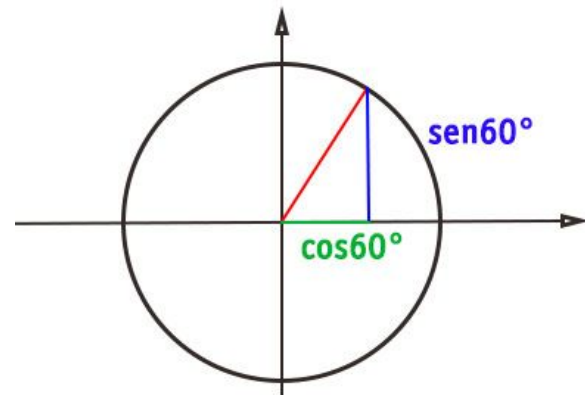
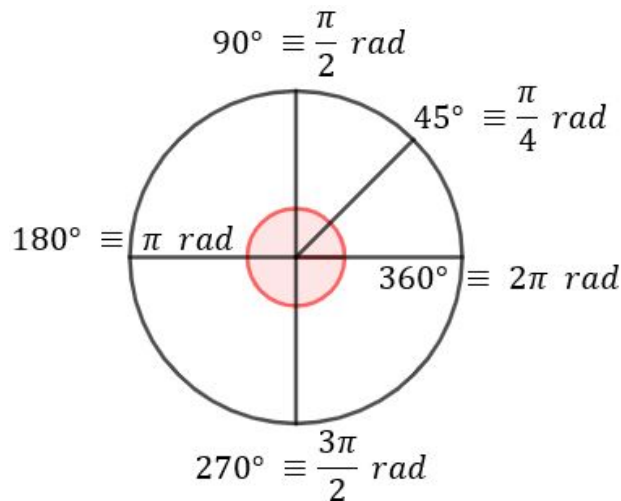
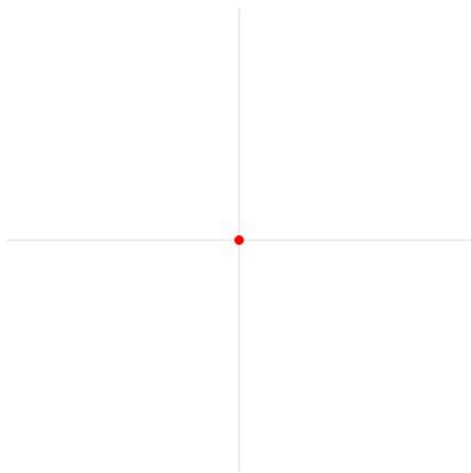
```
int x = 0;  
while (x < 10)  
{  
    GameObject cubo = GameObject.CreatePrimitive(PrimitiveType.Cube);  
    cubo.transform.position = new Vector3(x * 1.2f, 0, 0);  
    cubo.GetComponent<Renderer>().material.color = (x % 2 == 0) ? Color.red : Color.blue;  
    x++;  
}
```

WHILE

Ejemplos estructuras repetitivas en C#: bucles

Bucle DO-WHILE: script que utiliza un bucle do-while para crear una espiral de cubos. La espiral tendrá un radio, una cantidad de vueltas, y una cantidad de cubos por vuelta:

Ejemplos estructuras repetitivas en C#: bucles



$$\text{Grados} = \text{Radianes} \times \left(\frac{180}{\pi} \right)$$

$$\text{Radianes} = \text{Grados} \times \left(\frac{\pi}{180} \right)$$

```
float angulo1 = Mathf.PI * 2;    //360grados
float angulo2 = Mathf.PI;        //180grados
float angulo3 = Mathf.PI / 4;    //45grados
float angulo4 = 12;              //12*180/π=687.5grados
```

```
// para un círculo de radio 1
float angulo = Mathf.PI / 3;
float x = Mathf.Cos(angulo);    //0.5
float y = Mathf.Sin(angulo);    //0.866
```

Ejemplos estructuras repetitivas en C#: bucles

Bucle DO-WHILE: script que utiliza un bucle do-while para crear una espiral de cubos. La espiral tendrá un radio, una cantidad de vueltas, y una cantidad de cubos por vuelta:

```
public float radio = 2.0f;           // radio de la espiral
public int vueltas = 3;              // cantidad de vueltas
public int puntosPorVuelta = 100;    // cantidad de puntos por vuelta
int i = 0;
do
{
    float angulo = Mathf.PI * 2 * i / puntosPorVuelta;
    float x = radio * Mathf.Cos(angulo);
    float z = radio * Mathf.Sin(angulo);
    float y = i * 0.1f;

    GameObject nuevoCubo = GameObject.CreatePrimitive(PrimitiveType.Cube);
    nuevoCubo.transform.position = new Vector3(x, y, z);

    radio += 0.05f;
    i++;
} while (i < vueltas * puntosPorVuelta);
```

DO-WHILE

Ejemplos estructuras repetitivas en C#: bucles

Bucle FOREACH: script que utiliza un bucle foreach para recorrer los puntos de un Array.

```
public Vector3[] posicionesObjetivo = {
    new Vector3(0, 0, 0),
    new Vector3(1, 1, 0),
    new Vector3(2, 0, 2)
};
public float tiempoEntrePosiciones = 1f;
void Start()
{
    StartCoroutine(Mover());
}
IEnumerator Mover()
{
    while (true) // bucle infinito para reiniciar la secuencia
    {
        foreach (Vector3 posicionObjetivo in posicionesObjetivo)
        {
            transform.position = posicionObjetivo;
            yield return new WaitForSeconds(tiempoEntrePosiciones);
        }
    }
}
```

FOREACH

Introducción a C# en Unity:

Ejercicios bucles

Estructuras de control en C#: switch

Ejercicio 1: utilizar un bucle para crear una estructura en forma de escalera de cubos en la escena de Unity. La estructura de la escalera debe incluir 20 "peldaños". Amplía el script para que se pueda ajustar desde el inspector la anchura de la escalera(la anchura de los cubos). Realiza una segunda ampliación para poder ajustar la profundidad del escalón.

Ejercicio 2: utilizar un bucle para crear una estructura cuadrada de cubos sobre los ejes X y Z. La estructura debe ser un cuadrado de 10x10, y los cubos deben estar separados ligeramente en los ejes X y Z.

Ejercicio 3: utilizar un bucle para crear una estructura piramidal de cubos dentro de una escena en Unity. Se debe tener en cuenta que si el primer nivel tiene 10 cubos de lado, el nivel superior deberá tener 9, y así sucesivamente hasta el último nivel, que tendrá un único cubo. Cada uno de los niveles deberá estar centrado respecto al nivel inferior. Amplía el script para que pueda generar NxN pirámides formando un cuadrado

Restricciones para los 3 ejercicios:

- Cada cubo debe tener dimensiones de 1x1x1.
- Todo el código debe estar dentro del método Start.
- No se permite el uso de métodos adicionales, ni corrutinas.

Ejemplos estructuras repetitivas en C#: bucles

```
void Start()
{
    for (int i = 0; i < 20; i++)
    {
        GameObject cubo = GameObject.CreatePrimitive(PrimitiveType.Cube);
        cubo.transform.position = new Vector3(i, i, 0);
    }
}
```

Escalera V1

```
public float anchoEscalon = 5f;
void Start()
{
    for (int i = 0; i < 20; i++)
    {
        GameObject cubo = GameObject.CreatePrimitive(PrimitiveType.Cube);
        cubo.transform.localScale = new Vector3(1, 1, anchoEscalon);
        cubo.transform.position = new Vector3(i, i, 0);
    }
}
```

Escalera V2

```
public float anchoEscalon = 5f, profundidadEscalon = 2f;
void Start()
{
    for (int i = 0; i < escalones; i++)
    {
        GameObject cubo = GameObject.CreatePrimitive(PrimitiveType.Cube);
        cubo.transform.localScale = new Vector3(profundidadEscalon, 1, anchoEscalon);
        cubo.transform.position = new Vector3(i*profundidadEscalon, i, 0);
    }
}
```

Escalera V3

Ejemplos estructuras repetitivas en C#: bucles

Cuadrado de cubos

```
public int ancho = 10;           // cantidad de cubos de lado
public float separacion = 1.1f;  // distancia entre cubos

void Start()
{
    for (int z = 0; z < ancho; z++)
    {
        for (int x = 0; x < ancho; x++)
        {
            GameObject cubo = GameObject.CreatePrimitive(PrimitiveType.Cube);
            cubo.transform.position = new Vector3(x * separacion, 0, z * separacion);
        }
    }
}
```

Ejemplos estructuras repetitivas en C#: bucles

Pirámide de cubos

```
public int niveles = 10;           // niveles de la pirámide
public float separacion = 1.1f;    // distancia entre cubos

void Start()
{
    Vector3 inicio = Vector3.zero;
    for (int y = 0; y < niveles; y++)
    {
        for (int z = 0; z < niveles - y; z++)
        {
            for (int x = 0; x < niveles - y; x++)
            {
                GameObject cubo = GameObject.CreatePrimitive(PrimitiveType.Cube);
                cubo.transform.position = inicio + new Vector3(x*separacion, y, z*separacion);
            }
        }
        // calculamos la posición inicial para el nuevo nivel
        inicio.x += separacion / 2;
        inicio.z += separacion / 2;
    }
}
```


Ejemplos estructuras repetitivas en C#: bucles

Cuadrado con pirámides de cubos

```
public int niveles = 10;           // niveles de la pirámide
public float sep = 1.1f;          // distancia entre cubos
public int N = 5;                 // cantidad de pirámides

for (int fila = 0; fila < N; fila++)
{
    for (int columna = 0; columna < N; columna++)
    {
        Vector3 inicio = new Vector3(fila * niveles * sep, 0, columna * niveles * sep);

        for (int y = 0; y < niveles; y++)
        {
            for (int x = 0; x < niveles - y; x++)
            {
                for (int z = 0; z < niveles - y; z++)
                {
                    GameObject cubo = GameObject.CreatePrimitive(PrimitiveType.Cube);
                    cubo.transform.position = inicio + new Vector3(x * sep, y, z * sep);
                }
            }
            // calculamos la posición inicial para el nuevo nivel
            inicio.x += separacion / 2;
            inicio.z += separacion / 2;
        }
    }
}
```

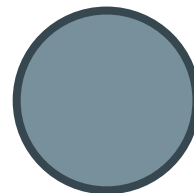
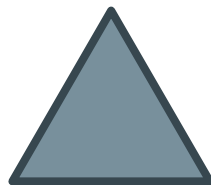
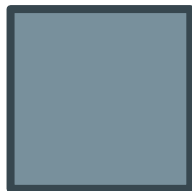
Estructuras de control en C#: switch

Ejercicio 4: con el objetivo de repasar conceptos vistos anteriormente. Se pide utilizar bucles para generar 3 formas básicas formadas por cubos, se deben colocar sobre los ejes XZ, es decir todos los cubos tendrán el mismo valor en el eje Y.

- Cuadrado de lado 'L', formado por 'N' cubos.
- Triángulo equilátero de lado 'L', formado por 'N' cubos.
- Círculo de radio 'R', formado por 'N' cubos.

Restricciones para los 3 ejercicios:

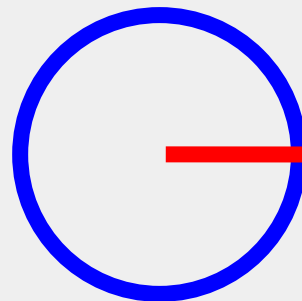
- Cada cubo debe tener dimensiones de 1x1x1.
- Todo el código debe estar dentro del método Start.
- No se permite el uso de métodos adicionales, ni corrutinas.
- Se deben definir como variables públicas el lado y el radio.
- Se debe definir como variable pública la cantidad de cubos utilizada para dibujar la forma.



Ejemplos estructuras repetitivas en C#: bucles

```
public int numCubos = 100;  
public float radio = 5f;  
  
void Start()  
{  
    for (int i = 0; i < numCubos; i++)  
    {  
        // posición angular en radianes  
        float angulo = i * 2 * Mathf.PI / numCubos;  
  
        // coordenadas x y z para cada cubo  
        float x = Mathf.Cos(angulo) * radio;  
        float z = Mathf.Sin(angulo) * radio;  
  
        // crear cubo y posicionarlo  
        GameObject cubo = GameObject.CreatePrimitive(PrimitiveType.Cube);  
        cubo.transform.position = new Vector3(x, 0, z);  
    }  
}
```

Círculo de cubos



Ejemplos estructuras repetitivas en C#: bucles

Cuadrado de cubos

```
public float lado = 10f;  
public int numCubos = 20;
```

```
void Start()  
{
```

```
    float perimetro = 4 * lado;           // perímetro del cuadrado  
    float espacio = perimetro / numCubos; // espacio entre cada cubo
```

```
    for (int i = 0; i < numCubos; i++)  
    {
```

```
        float posActual = i * espacio; // posición actual a lo largo del perímetro  
        Vector3 posCubo = Vector3.zero;
```

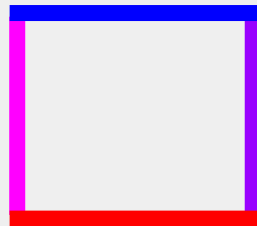
```
        // bordes del cuadrado: inferior, derecho, superior e izquierdo
```

```
            if (posActual < lado)      { posCubo.x = posActual; }  
        else if (posActual < 2 * lado) { posCubo.x = lado; posCubo.z = posActual - lado; }  
        else if (posActual < 3 * lado) { posCubo.x = 3 * lado - posActual; posCubo.z = lado; }  
        else                           { posCubo.z = 4 * lado - posActual; }
```

```
        GameObject cubo = GameObject.CreatePrimitive(PrimitiveType.Cube);  
        cubo.transform.position = posCubo;
```

```
    }
```

```
}
```

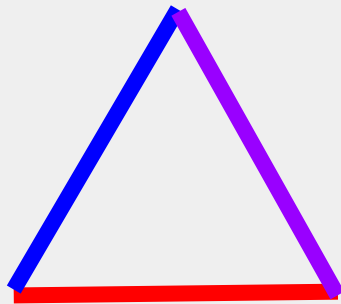


Ejemplos estructuras repetitivas en C#: bucles

Triángulo equilátero de cubos

```
public float lado = 4f;    // lado del triángulo
public int numCubos = 27;  // total de cubos
void Start()
{
    float espacio = 3 * lado / numCubos;    // espacio entre cada cubo
    float altura = Mathf.Sqrt(3) * lado / 2; // altura del triángulo
    for (int i = 0; i < numCubos; i++)
    {
        float posActual = i * espacio;    // posición actual a lo largo del perímetro
        Vector3 posCubo = Vector3.zero;

        if (posActual < lado)    posCubo.x = posActual;
        else if (posActual < 2 * lado)
        {
            posCubo.x = lado - (posActual - lado) * 0.5f;
            posCubo.z = (posActual - lado) * altura / lado;
        }
        else
        {
            posCubo.x = lado - (posActual - lado) * 0.5f;
            posCubo.z = altura - (posActual - 2 * lado) * altura / lado;
        }
        GameObject cubo = GameObject.CreatePrimitive(PrimitiveType.Cube);
        cubo.transform.position = posCubo;
    }
}
```



Introducción a C# en Unity: enum y struct

Estructuras en C#: enum

En Unity, enum (enumeraciones) y struct (estructuras) son dos tipos de datos que te permiten organizar y almacenar información de manera eficiente y legible.

- **enum:** permite definir un conjunto de constantes nombradas que hacen que tu código sea más legible y fácil de mantener. Son especialmente útiles para definir estados o configuraciones que sólo tienen un conjunto limitado de valores.
 - Mejora la legibilidad al dar nombres significativos a valores constantes.
 - Reduce la probabilidad de errores al restringir las opciones a un conjunto predefinido.

```
public enum EstadoJuego
{
    MenuPrincipal,
    Jugando,
    Pausa,
    GameOver
}
```

```
public enum NivelDificultad
{
    Facil,
    Medio,
    Dificil,
    Pesadilla
}
```

```
public enum TipoArma
{
    Espada,
    Arco,
    Lanza,
    Maza
}
```

Estructuras en C#: struct

En Unity, enum (enumeraciones) y struct (estructuras) son dos tipos de datos que te permiten organizar y almacenar información de manera eficiente y legible.

- **struct:** es un tipo de datos compuesto que puede contener múltiples variables bajo un solo nombre. En Unity, las estructuras son especialmente útiles para agrupar datos relacionados
 - Útil para agrupar datos pequeños y relacionados.
 - Menos costoso en términos de memoria que las clases debido a su almacenamiento en la pila.

```
public struct Punto3D
{
    public int x, y, z;
    public Punto3D(int x, int y, int z)
    {
        this.x = x;
        this.y = y;
        this.z = z;
    }
}
```


Estructuras en C#: enum

Uso de enum en Código: se suelen utilizar para controlar la lógica del juego de manera más clara y mantenible. Por ejemplo:

```
EstadoJuego estadoActual = EstadoJuego.Pausa;  
switch (estadoActual)  
{  
    case EstadoDelJuego.MenuPrincipal:  
        // código para el menú principal  
        break;  
    case EstadoDelJuego.Jugando:  
        // lógica del juego  
        break;  
    case EstadoDelJuego.Pausa:  
        // código para el menú de pausa  
        break;  
    case EstadoDelJuego.GameOver:  
        // código para el estado de Game Over  
        break;  
}
```

```
public enum EstadoJuego  
{  
    MenuPrincipal,  
    Jugando,  
    Pausa,  
    GameOver  
}
```

Estructuras en C#: enum y struct

Podemos combinar el uso de enum y struct para crear estructuras más complejas. Sin embargo, en ocasiones deberemos optar por crear clases.

```
public enum TipoArma
{
    Espada,
    Arco,
    Lanza
}
```

```
public struct DatosArma
{
    public string nombre;
    public int daño;
    public TipoArma tipo;
    public DatosArma(string nombre, int daño, TipoArma tipo)
    {
        this.nombre = nombre;
        this.daño = daño;
        this.tipo = tipo;
    }
}
```

```
DatosArma arma1 = new DatosArma("Cimitarra de la ira", 28, TipoArma.Espada);
DatosArma arma2 = new DatosArma("Arco de la guardia", 50, TipoArma.Arco);
DatosArma arma3 = new DatosArma("Tridente de escamas", 22, TipoArma.Lanza);
```

Estructuras en C#: enum y struct

Utilizar **enum** cuando:

- **Conjunto fijo de valores:** Si tienes un conjunto fijo de valores constantes que son mutuamente excluyentes, los enum son la mejor opción.
- **Legibilidad:** Los enum hacen que el código sea más legible, ya que puedes usar nombres descriptivos para los valores del conjunto, lo que facilita la comprensión del código.
- **Seguridad de tipo:** Utilizar enum evita errores al pasar un valor inválido a una función, ya que el compilador verificará que solo los valores dentro del enum se pueden utilizar.
- **Comparaciones rápidas:** Los enum son eficientes para las comparaciones ya que, internamente, son representados por tipos integrales. Son ideales para ser utilizados en switch y bucles, donde las comparaciones frecuentes son necesarias.

Utilizar **struct** cuando:

- **Agrupación de datos:** si necesitas agrupar múltiples campos relacionados pero no necesitas comportamientos complejos (métodos, herencia, etc.).
- **No se necesita herencia o polimorfismo:** cuando no necesitas herencia, ni polimorfismo para el tipo de datos que estás modelando.
- **Inmutabilidad y eficiencia:** si trabajas con datos que son inmutables y quieres evitar la sobrecarga del recolector de basura (que está asociada con las clases).

Introducción a C# en Unity: Funciones y métodos

Funciones y métodos en C#

Una función es un bloque de código diseñado para llevar a cabo una tarea específica. Un método es una función asociada con una clase o un objeto. Los términos a menudo se utilizan de forma intercambiable.

```
[modificador_de_acceso] tipo_retorno NombreMetodo(parametros)
{
    // cuerpo del método
}
```

```
public void MostrarMensaje()
{
    Debug.Log("aulaenlanube.com");
}
```

```
public
private
protected
internal
```

```
public void Sumar(int num1, int num2)
{
    int suma = num1 + num2;
    Debug.Log($"La suma es:{suma}");
}
```

```
public int Sumar(int num1, int num2)
{
    return num1 + num2;
}
```

Nomenclatura de funciones y métodos en C#

En C# y en el desarrollo con Unity, la convención de nomenclatura más comúnmente utilizada para los identificadores de métodos es estilo "**PascalCase**", donde la primera letra de cada palabra se escribe en mayúscula. Este estilo es coherente con las directrices de diseño de C# publicadas por Microsoft y es la norma en la comunidad de desarrolladores.

```
// correcto según la convención
public void MoverPersonaje()
{
    // ...
}

// incorrecto según la convención
public void moverPersonaje()
{
    // ...
}

// incorrecto según la convención
public void mover_personaje()
{
    // ...
}
```

Tipo valor y tipo referencia en C#

Al igual que en otros lenguajes, en C# según el tipo de dato, podemos tener tipos de valor y tipos de referencia, conocer la diferencia es esencial para saber cómo pasamos los parámetros a los métodos.

Tipo Valor: los tipos por valor son aquellos que almacenan una copia del valor en sí mismo. Los tipos de datos primitivos como int, float, double, char, bool y las estructuras (struct) son ejemplos de tipos por valor. Cuando pasas un tipo por valor como parámetro, se crea una copia del valor original, y las modificaciones en el método no afectan al valor original.

```
public void ModificarValor(int num)
{
    num += 10;
    Debug.Log($"Número en el método ModificarValor:{num}");
}
void Start()
{
    int miNumero = 5;
    ModificarValor(miNumero);
    Debug.Log($"Número en Start:{miNumero}"); // el valor de 'miNumero' no cambia
}
```

Tipo valor y tipo referencia en C#

Tipo Referencia: los tipos por referencia son aquellos que almacenan una referencia a la ubicación de memoria del valor, en lugar del valor en sí. Las clases (class) y los arrays son ejemplos de tipos por referencia. Cuando pasas un tipo por referencia como parámetro, cualquier modificación en el método afectará al objeto original.

```
public void ModificarObjeto(Vector3 pos)
{
    pos.x += 10;
    Debug.Log($"Posición en el método ModificarObjeto:{pos}");
}
public void ModificarObjeto(Posicion pos)
{
    pos.x += 10;
    Debug.Log($"Posición en el método ModificarObjeto:{pos}");
}
void Start()
{
    Vector3 miPosicion1 = Vector3.zero;
    Posicion miPosicion2 = new Posicion(0,0,0);
    ModificarObjeto(miPosicion1);
    ModificarObjeto(miPosicion2);
}
```


Tipo valor y tipo referencia en C#: enum

```
public enum TipoArma
{
    Espada,
    Arco,
    Lanza
}
```

```
public void CambiarTipoArma(TipoArma armaActual)
{
    armaActual = armaActual switch
    {
        TipoArma.Espada    => TipoArma.Arco,
        TipoArma.Arco      => TipoArma.Lanza,
        TipoArma.Lanza     => TipoArma.Espada
        -                   => TipoArma.Espada
    };
    Debug.Log("Arma actual: " + armaActual);
}
```

```
TipoArma miArma = TipoArma.Espada;
Debug.Log("Arma antes: " + miArma);           // Arma antes: Espada

CambiarTipoArma(miArma);                       // Arma actual: Arco
Debug.Log("Arma después: " + miArma);          // Arma después: Espada
```

Tipo valor y tipo referencia en C#: enum

```
public enum TipoArma
{
    Espada,
    Arco,
    Lanza
}
```

```
public TipoArma CambiarTipoArma(TipoArma armaActual)
{
    armaActual = armaActual switch
    {
        TipoArma.Espada    => TipoArma.Arco,
        TipoArma.Arco      => TipoArma.Lanza,
        TipoArma.Lanza     => TipoArma.Espada
        _                  => TipoArma.Espada
    };
    Debug.Log("Arma actual: " + armaActual);
    return armaActual;
}
```

```
TipoArma miArma = TipoArma.Espada;
Debug.Log("Arma antes: " + miArma);           // Arma antes: Espada

miArma = CambiarTipoArma(miArma);             // Arma actual: Arco
Debug.Log("Arma después: " + miArma);         // Arma después: Arco
```

Tipo valor y tipo referencia en C#: enum

```
public struct DatosArma
{
    public string nombre;
    public int daño;
    public TipoArma tipo;
    public DatosArma(string nombre, int daño, TipoArma tipo)
    {
        this.nombre = nombre;
        this.daño = daño;
        this.tipo = tipo;
    }
}
```

```
public enum TipoArma
{
    Espada,
    Arco,
    Lanza
}
```

```
public void CambiarDatosArma(DatosArma datos)
{
    datos.daño *= 2;
    datos.nombre += " X 2";
    Debug.Log($"{datos.nombre}:{datos.daño}");
}
```

```
DatosArma arma1 = new DatosArma("Cimitarra de la ira", 28, TipoArma.Espada);
CambiarDatosArma(arma1); //Cimitarra de la ira X 2:56
Debug.Log($"{arma1.nombre}:{arma1.daño}"); //Cimitarra de la ira:28
```

Introducción a C# en Unity: métodos con devolución múltiple

Devolución múltiple en métodos de C#

En C# y Unity, hay varias formas de devolver más de un valor desde un método. Vamos a explorar algunas de las más comunes:

- **Uso de out y ref:** la palabra reservada out permite a un método devolver más de un valor. La palabra clave ref tiene un propósito similar, pero requiere que la variable se inicialice antes de pasarse al método.

```
void Dims(out float ancho, out float alto)
{
    ancho = 10.5f;
    alto = 20.5f;
}
void Start()
{
    float miAncho;
    float miAlto;

    Dims(out miAncho, out miAlto);
    Debug.Log($"{miAncho}:{miAlto}");
}
```

```
void Mods(ref int a, ref int b)
{
    a += 10;
    b += 20;
}
void Start()
{
    int num1 = 5;
    int num2 = 10;

    Mods(ref num1, ref num2);
    Debug.Log($"{num1}:{num2}"); //15:30
}
```

Devolución múltiple en métodos de C#

```
public (int, string, Vector3) ObtenerDatosJugador()
{
    int puntuacion = 100;
    string nombre = "aula en la nube";
    Vector3 posicion = Vector3.zero;
    return (puntuacion, nombre, posicion);
}

public (int, string, Vector3) ObtenerDatosJugador(int p, string n, Vector3 v)
{
    p = 5;
    n = "aulaenlanube.com";
    v = Vector3.zero;
    return (p, n, v);
}

void Start()
{
    (int p, string n, Vector3 v) = ObtenerDatosJugador();
    Debug.Log($"{p}:{n}:{v}");           // 100:aula en la nube:(0.0, 0.0, 0.0)
    (p, n, v) = ObtenerDatosJugador(p, n, v);
    Debug.Log($"{p}:{n}:{v}");           // 5:aulaenlanube.com:(0.0, 0.0, 0.0)
}
```

TUPLAS: son una forma eficiente y sencilla de devolver múltiples valores.

Devolución múltiple en métodos de C#

```
public class DatosJugador
{
    public int puntuacion;
    public string nombre;
    public Vector3 posicion
}
```

```
public DatosJugador ObtenerDatos()
{
    DatosJugador datos = new DatosJugador();
    datos.puntuacion = 100;
    datos.nombre = "aula en la nube";
    datos.posicion = Vector3.zero;
    return datos;
}
```

```
void Start()
{
    DatosJugador datos = ObtenerDatos();
    Debug.Log($"{datos.nombre}:{datos.puntuacion}:{datos.posicion}");
}
```

Si los valores están relacionados entre sí, podría tener sentido encapsularlos en una clase o estructura.

Introducción a C# en Unity: Corrutinas

Corrutinas en C#

Una corrutina es una función especial que puede pausar su ejecución y retornar el control a Unity para que continúe con el próximo frame, y luego continuar desde donde lo dejó.

- **Sintaxis y Uso:** para definir una corrutina, usamos **IEnumerator** como tipo de retorno y empleamos la instrucción **yield** para pausar la ejecución.

```
IEnumerator MiCorrutina()  
{  
    Debug.Log("Visita aulaenlanube.com");  
    yield return new WaitForSeconds(2);  
    Debug.Log("2 segundos después...");  
}
```

- **Iniciar y Detener Corrutinas**

```
StartCoroutine("MiCorrutina");  
StopCoroutine("MiCorrutina");  
StartCoroutine(MiCorrutina());  
StopCoroutine(MiCorrutina());
```

Corrutinas en C#

Las corrutinas en Unity son extremadamente útiles para realizar tareas que toman tiempo en varios frames, permitiendo una forma más limpia y legible de escribir este tipo de lógicas.

- **Ejemplo:** supongamos que tienes un objeto que quieres que cambie de color en un ciclo de Rojo → Verde → Azul → Rojo y así sucesivamente, cada color durando un segundo.

```
IEnumerator CambiarColor()  
{  
    Renderer rend = GetComponent<Renderer>();  
    while (true) // bucle infinito  
    {  
        rend.material.color = Color.red;  
        yield return new WaitForSeconds(1);  
        rend.material.color = Color.green;  
        yield return new WaitForSeconds(1);  
        rend.material.color = Color.blue;  
        yield return new WaitForSeconds(1);  
    }  
}
```

```
StartCoroutine(CambiarColor());
```

```
StopCoroutine(CambiarColor());
```

Corrutinas en C#

Es posible invocar la misma corrutina múltiples veces y tener varias instancias de esa corrutina ejecutándose simultáneamente. Cada invocación de StartCoroutine genera una nueva instancia de la corrutina, y todas pueden ejecutarse en paralelo.

```
IEnumerator CambiarColor(int n)
{
    Renderer rend = GetComponent<Renderer>();
    while (true) // bucle infinito
    {
        rend.material.color = Color.red;
        yield return new WaitForSeconds(n);
        rend.material.color = Color.green;
        yield return new WaitForSeconds(n);
        rend.material.color = Color.blue;
        yield return new WaitForSeconds(n);
    }
}
```

```
StartCoroutine("CambiarColor", 1);
StartCoroutine("CambiarColor", 5);

// detiene todas las instancias
StopCoroutine("CambiarColor");
```

```
Coroutine c1 = StartCoroutine(CambiarColor(1));
Coroutine c2 = StartCoroutine(CambiarColor(5));
...
StopCoroutine(c2);
```

Corrutinas en C#

Las corrutinas en Unity son extremadamente útiles para realizar tareas que toman tiempo en varios frames, permitiendo una forma más limpia y legible de escribir este tipo de lógicas.

- **Ejemplo:** imagina que quieres hacer que un objeto 3D en tu escena parpadee, es decir, se haga visible e invisible repetidamente durante un tiempo determinado.

```
IEnumerator Parpadear()
{
    MeshRenderer rend = GetComponent<MeshRenderer>();
    float tiempoEspera = 0.5f; // duración del parpadeo en segundos
    int numeroParpadeos = 10;  // número de veces que el objeto parpadea

    for (int i = 0; i < numeroParpadeos; i++)
    {
        rend.enabled = !rend.enabled; // invertir visibilidad
        yield return new WaitForSeconds(tiempoEspera); // esperamos
    }
    rend.enabled = true; // aseguramos que sea visible al final
}
```

Ejemplo corrutinas en C#

Supongamos que queremos mover gradualmente un GameObject en Unity a una posición objetivo en una determinada cantidad de tiempo, podemos hacerlo con una corrutina.

```
IEnumerator MoverObjeto(Vector3 posObjetivo, float duracion)
{
    Vector3 posIni = transform.position;
    float tiempoTranscurrido = 0.0f;
    while (tiempoTranscurrido < duracion)
    {
        tiempoTranscurrido += Time.deltaTime;
        float t = tiempoTranscurrido / duracion;
        // calcular las nuevas coordenadas para cada eje
        float xNueva = posIni.x + (posObjetivo.x - posIni.x) * t;
        float yNueva = posIni.y + (posObjetivo.y - posIni.y) * t;
        float zNueva = posIni.z + (posObjetivo.z - posIni.z) * t;
        // actualizar la posición del objeto
        transform.position = new Vector3(xNueva, yNueva, zNueva);
        yield return null; // esperamos hasta el siguiente frame
    }
    transform.position = posObjetivo; // aseguramos que el objeto llegue a su destino
}
```

```
StartCoroutine(MoverObjeto(new Vector3(4, 8, 3), 2f));
```

Ejemplo corrutinas en C#

Supongamos que queremos mover gradualmente un GameObject en Unity a una posición objetivo en una determinada cantidad de tiempo, podemos hacerlo con una corrutina.

```
IEnumerator MoverObjeto(Vector3 objetivo, float duracion)
{
    Vector3 puntoInicial = transform.position;
    float tiempoPasado = 0;

    while (tiempoPasado < duracion)
    {
        tiempoPasado += Time.deltaTime;
        transform.position = Vector3.Lerp(puntoInicial, objetivo, tiempoPasado / duracion);
        yield return null;
    }
}
```

```
StartCoroutine(MoverObjeto(new Vector3(4, 8, 3), 2f));
```

Interpolación Lineal

$$\text{Lerp}(A, B, t) = A + (B - A) * t$$

La función Lerp toma tres parámetros: A, B, y t.

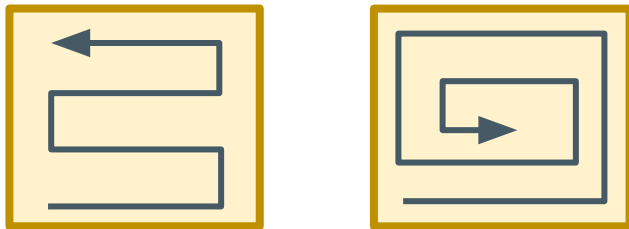
- A y B son los valores entre los que quieres interpolar.
- t es un valor entre 0 y 1 que representa la "distancia" a la que deseas llegar entre A y B.

Introducción a C# en Unity: Ejercicios Corrutinas

Ejercicios corrutinas en C#

Ejercicio 1: reutilizar el código que creamos para generar una estructura cuadrada de cubos sobre los ejes X y Z. El objetivo es generar un cubo cada 0,1s a través de un corrutina. Además, se deberán generar siguiendo un patrón en zig zag.

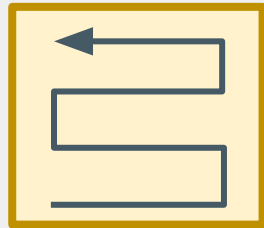
Ejercicio 2: reutilizar el código que creamos para generar una estructura cuadrada de cubos sobre los ejes X y Z. El objetivo es generar un cubo cada 0,1s a través de un corrutina. Además, se deberán generar siguiendo un en forma de espiral cuadrada.



Ejercicio 3: reutilizar el código que creamos para generar una estructura cuadrada de cubos sobre los ejes X y Z. El objetivo es que cada cubo cambie su escala en el eje Y de forma aleatoria. Además, el centro de cada cubo se debe posicionar de forma que la base de cada cubo se mantenga en $Y = 0$. Se deben crear variables públicas para indicar la escala máxima y mínima, y la duración que deberá utilizarse para realizar el escalado.

Ejercicios corrutinas en C#

```
IEnumerator CrearCuadradoZigZag()
{
    bool izquierdaADerecha = true;
    for (int z = 0; z < lado; z++)
    {
        if (izquierdaADerecha)
        {
            for (int x = 0; x < lado; x++)
            {
                CrearCubo(x, z);
                yield return new WaitForSeconds(intervalo);
            }
        }
        else
        {
            for (int x = lado - 1; x >= 0; x--)
            {
                CrearCubo(x, z);
                yield return new WaitForSeconds(intervalo);
            }
        }
        izquierdaADerecha = !izquierdaADerecha;
    }
}
```



CrearCubo

```
GameObject cubo = GameObject.CreatePrimitive(PrimitiveType.Cube);
cubo.transform.position = new Vector3(x, 0, z);
```

Ejercicios corrutinas en C#

```
private IEnumerator GenerarCuadradoEspiral()
{
    int x = 0, z = 0;
    int capas = (lado + 1) / 2;
    for (int i = 0; i < capas; i++)
    {
        for (int j = 0; j < lado - i * 2; j++) // derecha
        {
            CrearCubo(x, z); yield return new WaitForSeconds(intervalo);
            if (j < lado - i * 2 - 1) x++;
        }
        for (int j = 0; j < lado - i * 2 - 1; j++) // arriba
        {
            CrearCubo(x, ++z); yield return new WaitForSeconds(intervalo);
        }
        for (int j = 0; j < lado - i * 2 - 1; j++) // izquierda
        {
            CrearCubo(--x, z); yield return new WaitForSeconds(intervalo);
        }
        for (int j = 0; j < lado - i * 2 - 2; j++) // abajo
        {
            CrearCubo(x, --z); yield return new WaitForSeconds(intervalo);
        }
        x++;
    }
}
```



CrearCubo

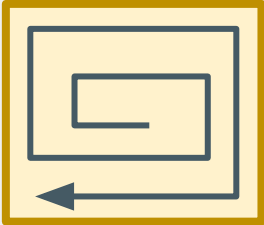
```
GameObject cubo = GameObject.CreatePrimitive(PrimitiveType.Cube);
cubo.transform.position = new Vector3(x, 0, z);
```

Ejercicios corrutinas en C#

```
private IEnumerator GenerarCuadradoEspiralInvertida()
{
    Vector2Int direccion = Vector2Int.right;
    Vector2Int posicion = new Vector2Int(0, 0);
    int pasos = 1;
    int totalCubos = lado * lado;

    for (int i = 0; i < totalCubos;)
    {
        for (int j = 0; j < 2; j++) // 2 direcciones por cada tamaño de pasos
        {
            for (int p = 0; p < pasos && i < totalCubos; p++)
            {
                CrearCubo(posicion.x, posicion.y);
                yield return new WaitForSeconds(intervalo);

                posicion += direccion; // avanzar en la dirección actual
                i++;
            }
            direccion = new Vector2Int(direccion.y, -direccion.x); //rotamos 90 grados
        }
        pasos++;
    }
}
```



CrearCubo

```
GameObject cubo = GameObject.CreatePrimitive(PrimitiveType.Cube);
cubo.transform.position = new Vector3(x, 0, z);
```

Ejercicios corrutinas en C#

```
public float duracion = 1f, escalaMinima = 1f, escalaMaxima = 3f;
private Vector3 escalaInicial, escalaObjetivo;
private IEnumerator AlternarEscala()
{
    escalaInicial = new Vector3(1, 1, 1);
    while (true)
    {
        float escalaYObjetivo = Random.Range(escalaMinima, escalaMaxima);
        escalaObjetivo = new Vector3(1, escalaYObjetivo, 1);
        yield return CambiarEscala(escalaInicial, escalaObjetivo); // escalar hacia arriba
        yield return CambiarEscala(escalaObjetivo, escalaInicial); // escalar hacia abajo
    }
}
private IEnumerator CambiarEscala(Vector3 desde, Vector3 hasta)
{
    float tiempoTranscurrido = 0f;
    while (tiempoTranscurrido < duracion)
    {
        tiempoTranscurrido += Time.deltaTime;
        float progreso = tiempoTranscurrido / duracion;

        float escalaY = Mathf.Lerp(desde.y, hasta.y, progreso); // interpolación lineal
        transform.localScale = new Vector3(1, escalaY, 1);

        transform.position = new Vector3(transform.position.x, escalaY/2, transform.position.z);
        yield return null;
    }
}
```

Introducción a C# en Unity:

Arrays

Arrays en C#

Al igual que en otros lenguajes de programación, los arrays en C#, son estructuras de datos que permiten almacenar múltiples valores del mismo tipo a través de un único identificador.

```
//declaración
tipo[] nombreArray;
//inicialización
nombreArray = new tipo[cantidadElementos];
```

Existen varias formas distintas de declarar e inicializar Arrays en C#

```
// declaración e inicialización juntas
int[] nums = new int[5];
string[] palabras = new string[3];
GameObject[] objetos = new GameObject[4];
//asignando valores en la declaración
int[] nums2 = { 1, 2, 3, 4, 5 };
string[] palabras2 = {"Hola", "mundo", "desde", "C#"};
```

Arrays en C#

Acceso a los Elementos de un Array: El acceso a un elemento específico del array se realiza mediante el índice del elemento. El índice comienza en 0 y termina en longitud-1.

```
int primerNumero = numeros[0];    // accede al primer elemento
numeros[3] = 10;                  // cambia el cuarto elemento a 10
```

Length: Retorna el número de elementos en el array.

```
int totalElementos = numeros.Length;
```

Arrays Multidimensionales: Los arrays pueden tener más de una dimensión.

```
int[,] matriz2D = new int[3, 4];    // Array 2D (3 filas, 4 columnas)
int[,,] matriz3D = new int[3, 5, 2]; // Array 3D (3filas, 5cols, 2prof)
matriz2D[2][1] = 5;
matriz3D[2][1][0] = 2;
```

Arrays en C#: recorrer Arrays

Existen distintas formas de recorrer Arrays, lo más típica es a través de un bucle for.

```
int[] nums1 = {1, 2, 3, 4, 5};  
int[,] nums2 = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
```

```
for(int i = 0; i < nums1.Length; i++)  
{  
    Debug.Log($"nums1[{i}]= {nums1[i]}");  
}
```

```
for(int i = 0; i < nums2.GetLength(0); i++)  
{  
    for(int j = 0; j < nums2.GetLength(1); j++)  
    {  
        Debug.Log($"nums2[{i}, {j}]= {nums2[i,j]}");  
    }  
}
```

Podemos recorrer Arrays, con foreach, aunque no podemos determinar la posición del elemento dentro del Array.

```
foreach(int num in nums2)  
{  
    Debug.Log($" {num}");  
}
```

En C#, una matriz bidimensional se considera un solo objeto, no un "array de arrays". Por lo tanto, no puedes usar un foreach para recorrer las filas y luego otro foreach interno para recorrer las columnas.

Arrays en C#: recorrer Arrays

En el caso de matrices escalonadas (también conocidas como "arrays de arrays" o "jagged arrays"), es posible utilizar bucles foreach anidados para recorrer los elementos.

En C#, una matriz escalonada se declara y se inicializa de la siguiente manera:

```
int[][] matrizEscalonada = {  
    new int[] {1, 2, 3},  
    new int[] {4, 5},  
    new int[] {6, 7, 8, 9}  
};
```

```
int[][] matrizEscalonada = new int[3][];  
matrizEscalonada[0] = new int[] {1, 2, 3};  
matrizEscalonada[1] = new int[] {4, 5};  
matrizEscalonada[2] = new int[] {6, 7, 8, 9};
```

```
int fila = 0;  
foreach (int[] filaActual in matrizEscalonada)  
{  
    string s = "";  
    foreach (int elemento in filaActual)  
    {  
        s += elemento + " ";  
    }  
    Debug.Log($"Fila{fila++}: {s}");  
}
```

SALIDA

```
Fila0: 1 2 3  
Fila1: 4 5  
Fila2: 6 7 8 9
```

Arrays en C#: Tipo referencia

Los arrays en C# son tipos de referencia, por lo tanto, si pasamos un array como parámetro a un método, le estamos pasando la referencia al espacio de memoria donde apunta el array.

```
void ModificarArray(int[] arr)
{
    for(int i = 0; i < arr.Length; i++) arr[i] += i;
}
```

Si pasamos a un método, un Array, las modificaciones dentro del método se reflejan en el Array.

```
int[] nums = new int[5];
nums[2] = 5;
Debug.Log(string.Join(", ", nums)); // 0, 0, 5, 0, 0
ModificarArray(nums);
Debug.Log(string.Join(", ", nums)); // 0, 1, 7, 3, 4
ModificarArray(nums);
Debug.Log(string.Join(", ", nums)); // 0, 2, 9, 6, 8
```

Arrays en C#: ejemplo de uso

Corrutina que permite mover un GameObject a través de un Array de Vector3. El movimiento se realiza a una velocidad constante y en bucle.

```
IEnumerator MoverEntrePuntos(Vector3[] puntos, float velocidad)
{
    if (puntos.Length < 2) yield break;
    int indiceSiguientePunto = 0;
    Vector3 inicio = transform.position;
    while (true)
    {
        Vector3 destino = puntos[indiceSiguientePunto];
        transform.position = Vector3.MoveTowards(inicio, destino, velocidad*Time.deltaTime);
        if (Vector3.Distance(transform.position, destino) < 0.1f)
        {
            inicio = destino;
            indiceSiguientePunto = ++indiceSiguientePunto % puntos.Length;
        }
        else inicio = transform.position;
        yield return null;
    }
}
```

A diferencia de **Vector3.Lerp**, **Vector3.MoveTowards** permite un movimiento a velocidad constante sin depender del tiempo transcurrido para la interpolación.

Introducción a C# en Unity:

Ejercicios Arrays

Ejemplos de Arrays en C# dentro de Unity

Ejercicio 1: Modificar el último ejemplo visto en la unidad para que el patrón de movimiento se haga en base a una duración. Es decir, deberá tardar lo mismo en moverse entre dos puntos del Arrays, independientemente de la distancia que haya entre dichos puntos. Además el movimiento entre dos puntos debe realizarse a una velocidad constante.

Ejercicio 2: Reutilizar el ejercicio de seguimiento y evasión para que cuando el seguidor haga contacto con el objetivo, haga un respawn en una posición aleatoria que deberá obtenerse de un Array de Vector3. Se considera contacto si la distancia entre ambos objetos es menor a 0.5.

Ejercicio 3: Reutilizar el ejercicio de seguimiento y evasión teniendo en cuenta que el cubo perseguidor se moverá en todo momento en un patrón en forma circular de radio R. En el momento que el GameObject objetivo esté más lejos que la distancia de seguimiento, el GameObject perseguidor deberá volver a la posición donde había dejado de seguirlo y seguir con su patrón de movimiento circular.

AMPLIACIÓN: Pensad cómo lo podemos plantear para que el patrón de movimiento forme un polígono regular de N lados. Para ello podemos hacer uso del teorema del coseno.

$$c^2 = a^2 + b^2 - 2ab \cos(\gamma)$$

Arrays en C#: ejercicios

Corrutina que permite mover un GameObject a través de un Array de Vector3. El movimiento se realiza partiendo de la duración entre dos puntos.

```
IEnumerator MoverEntrePuntos(Vector3[] puntos, float duracion)
{
    if (puntos.Length < 2) yield break;
    int indiceSiguientePunto = 0;
    Vector3 inicio = transform.position;
    float tiempoTranscurrido = 0f;
    while (true)
    {
        Vector3 destino = puntos[indiceSiguientePunto];
        tiempoTranscurrido += Time.deltaTime;
        float progreso = tiempoTranscurrido / duracion;
        transform.position = Vector3.Lerp(inicio, destino, progreso);
        if (progreso >= 1f) // si llegamos al punto destino
        {
            transform.position = destino;
            inicio = destino;
            tiempoTranscurrido = 0;
            indiceSiguientePunto = ++indiceSiguientePunto % puntos.Length;
        }
        yield return null;
    }
}
```

Arrays en C#: ejercicios

Reutilizar el ejercicio de seguimiento y evasión para que cuando el seguidor haga contacto con el objetivo, haga un respawn en una posición aleatoria que deberá obtenerse de un Array de Vector3. Se considera contacto si la distancia entre ambos objetos es menor a 0.5.

```
// variables públicas
```

```
public Transform objetivo;  
public float distanciaSeguimiento = 5.0f;  
public float distanciaRespawn = 0.5f;  
public Vector3[] posicionesRespawn;
```

```
void Update()
```

```
{
```

```
    float distanciaObjetivo = Vector3.Distance(transform.position, objetivo.transform.position);  
    Vector3 pos = transform.position;
```

```
    // respawn
```

```
    if (distanciaObjetivo < distanciaRespawn){  
        transform.position = posicionesRespawn[Random.Range(0,posicionesRespawn.Length)];  
    }
```

```
    // seguir
```

```
    else if (distanciaObjetivo < distanciaSeguimiento)  
    {  
        transform.position = Vector3.MoveTowards(pos, objetivo.position, velocidad * Time.deltaTime);  
    }
```

```
}
```

Seguimiento y Evasión con respawn random

Arrays en C#: ejercicios

Seguimiento y Evasión con patrón de movimiento circular en XZ

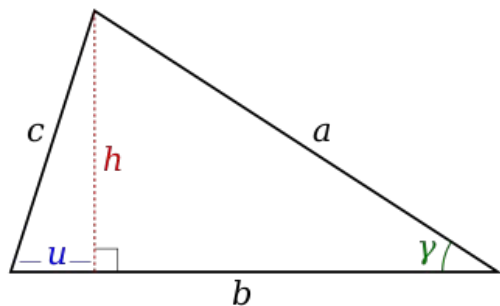
```
// variables públicas
```

```
void Update()
{
    float distanciaObjetivo = Vector3.Distance(transform.position, objetivo.transform.position);
    Vector3 pos = transform.position;
    if (distanciaObjetivo < distanciaEvasion) // evitar al objetivo
    {
        Vector3 direccion = (pos - objetivo.transform.position).normalized;
        transform.position += direccion * velocidad * Time.deltaTime;
    }
    else if (distanciaObjetivo < distanciaSeguimiento) // seguir al objetivo
    {
        transform.position = Vector3.MoveTowards(pos, objetivo.position, velocidad * Time.deltaTime);
    }
    else // volver al último punto de seguimiento y seguir el patrón circular
    {
        if (Vector3.Distance(pos, puntoCirculo) > 0.1f)
        {
            transform.position = Vector3.MoveTowards(pos, puntoCirculo, velocidad * Time.deltaTime);
        }
        else
        {
            indice = ++indice % cantidadPuntosCirculo;
            puntoCirculo = puntosCirculo[indice];
        }
    }
}
```


Arrays en C#: ejercicios

Podemos reutilizar el círculo para crear un polígono regular(todos los lados iguales) inscrito en un círculo de radio 1. Si tenemos un polígono regular de N lados inscrito en un círculo de radio 1, cada ángulo central asociado a un vértice del polígono será de 360 grados dividido por la cantidad de lados del polígono.

La longitud de cada lado del polígono se puede calcular mediante la siguiente fórmula utilizando la ley de los cosenos:



$$c^2 = a^2 + b^2 - 2ab \cos(\gamma)$$

$$L^2 = 1^2 + 1^2 - 2 \times 1 \times 1 \cos(360^\circ/n)$$

$$L^2 = 2 - 2 \cos(360^\circ/n)$$

$$L = \sqrt{2 - 2 \cos(360^\circ/n)}$$

$$L = \sqrt{2 - 2 \cos\left(\frac{360^\circ}{n}\right)}$$

Esta fórmula es válida para un círculo de radio 1. Si tienes un círculo de radio R, simplemente multiplicamos L por R.

Podemos reutilizar este ejemplo para generar los patrones de movimiento en forma de triángulo equilátero y cuadrado.

Arrays en C#: ejercicios

Podemos reutilizar el círculo para crear un polígono regular(todos los lados iguales) inscrito en un círculo de radio 1. Si tenemos un polígono regular de N lados inscrito en un círculo de radio 1, cada ángulo central asociado a un vértice del polígono será de 360 grados dividido por la cantidad de lados del polígono.

```
int numeroLados = 8;
float longitudLado = 5f;
float angulo = 360f / numeroLados; // grados para cada segmento de la figura
float radio = longitudLado / (2 * Mathf.Sin(Mathf.Deg2Rad * angulo / 2));
Vector3[] vertices = new Vector3[n];

for (int i = 0; i < numeroLados; i++)
{
    float ang = Mathf.Deg2Rad * i * angulo; // pasamos el ángulo a radianes
    vertices[i] = new Vector3(radio * Mathf.Cos(ang), 0, radio * Mathf.Sin(ang));
}
```

Introducción a C# en Unity: Delegados y eventos

Delegados en C#

Un delegado es un tipo que representa referencias a métodos con un tipo de firma y retorno particular. Básicamente, un delegado actúa como un envoltorio para un método y permite pasar métodos como parámetros. En el siguiente ejemplo, **MiDelegado1** es un delegado que puede hacer referencia a cualquier método que tome un entero como parámetro y no retorna nada (void). **MiDelegado2** es un delegado que recibe dos enteros y devuelve un entero.

```
public delegate void MiDelegado1(int numero);  
public delegate int MiDelegado2(int num1, int num2);
```

Los delegados son útiles en una variedad de situaciones en desarrollo de juegos con Unity, como:

- **Manejo de Eventos:** Los delegados pueden ser usados para implementar sistemas de eventos personalizados.
- **Callbacks:** Pueden ser usados para ejecutar una función cuando una tarea se ha completado.
- **Programación Modular:** Permiten escribir código más limpio y reutilizable al separar responsabilidades entre diferentes clases o componentes.

Eventos en C#

Los eventos en C# son una forma de permitir que un objeto notifique a otros objetos cuando algo ocurre. El objeto que lanza el evento (publicador) no necesita saber qué objetos están escuchando el evento (suscriptores).

- **Declaración de Eventos:** se usa la palabra clave **event** seguida por el tipo delegado que va a representar ese evento. Los delegados actúan como tipos para los métodos, permitiendo que los métodos se pasen como argumentos o se asignen a variables.

```
public delegate void MiDelegado(string mensaje);  
public event MiDelegado MiEvento;
```

- **Publicar Eventos:** para lanzar el evento, puedes usar el operador **?** para verificar si hay suscriptores. Si hay, se invoca el evento.
- **Suscribirse a Eventos:** se usa el operador **+=**, pasando el método que debe ser llamado cuando el evento se dispara.

```
if (MiEvento != null) MiEvento("Unity1");  
MiEvento?.Invoke("Unity2"); //simplificado
```

```
public void MetodoSuscriptor(string s)  
{  
    Debug.Log($"Evento recibido: {s}");  
}  
MiEvento += MetodoSuscriptor;
```

Eventos en C#

Evento de Puntuación

Supongamos que tenemos un juego en el que el jugador gana puntos por realizar ciertas acciones. Podemos usar un evento para notificar a diferentes partes del juego (como la interfaz de usuario) cuando la puntuación cambia.

```
public class Puntuacion : MonoBehaviour
{
    public delegate void EventoPuntuacion(int nuevaPuntuacion);
    public static event EventoPuntuacion AlCambiarPuntuacion;
    private int puntuacion;
    public void AumentarPuntuacion(int cantidad)
    {
        puntuacion += cantidad;
        AlCambiarPuntuacion?.Invoke(puntuacion);
    }
}
```

```
public class InterfazUsuario : MonoBehaviour
{
    [SerializeField] private Text textoPuntuacion;
    private void OnEnable() { Puntuacion.AlCambiarPuntuacion += ActualizarTextoPuntuacion; }
    private void OnDisable() { Puntuacion.AlCambiarPuntuacion -= ActualizarTextoPuntuacion; }
    private void ActualizarTextoPuntuacion(int nuevaPuntuacion)
    {
        textoPuntuacion.text = $"Puntuación: {nuevaPuntuacion}";
    }
}
```

Ejercicio eventos en C#

Ejercicio: Reutiliza el ejercicio del vídeo anterior de seguimiento y evasión con respawn para crear un evento que mantenga en todo momento la cantidad de respawns que se han producido. Crearemos un primer suscriptor que modificará un Gameobject con un Text. Además, deberemos crear otro suscriptor al evento desde otro script, el cual deberá mostrar un mensaje por consola en cada respawn, indicando en cada ocasión la posición(x, y, z) del respawn.

```
//eventos para actualizar la cantidad de respawns y la posición de respawn
public delegate void EventoCantidadRespawn(int n);
public static event EventoCantidadRespawn ActualizarCantidadRespawns;
public delegate void EventoPosicionRespawn(Vector3 v);
public static event EventoPosicionRespawn ActualizarPosicionRespawns;
private int cantidadRespawns = 0;
void Update()
{
    if (...) // respawn
    {
        transform.position = posicionesRespawn[Random.Range(0, posicionesRespawn.Length)];
        //lanzamos los eventos al actualizar la cantidad de respawns
        ActualizarCantidadRespawns?.Invoke(++cantidadRespawns);
        ActualizarPosicionRespawns?.Invoke(transform.position);
    }
    // ...
}
```

Ejercicio eventos en C#

```
public class InterfazUsuario : MonoBehaviour
{
    [SerializeField] private Text cantidadRespawns;
    private void OnEnable() { SeguidorRespawn.ActualizarCantidadRespawns += ActualizarRespawns; }
    private void OnDisable() { SeguidorRespawn.ActualizarCantidadRespawns -= ActualizarRespawns; }
    private void ActualizarRespawns(int cantidadRespawns)
    {
        this.cantidadRespawns.text = $"Respawns: {cantidadRespawns}";
    }
}
```

```
public class InterfazConsola : MonoBehaviour
{
    private void OnEnable() { SeguidorRespawn.ActualizarPosicionRespawns += MostrarRespawn; }
    private void OnDisable() { SeguidorRespawn.ActualizarPosicionRespawns -= MostrarRespawn; }
    private void MostrarRespawn(Vector3 posicionRespawn)
    {
        Debug.Log($"Respawn en : {posicionRespawn}");
    }
}
```


Introducción a C# en Unity: Eventos de teclado

Eventos de teclado

Los eventos de teclado son fundamentales en el desarrollo de videojuegos, especialmente en la interacción con el usuario. En Unity, el manejo de la entrada de teclado se realiza a través de la clase Input.

Los métodos más comunes para detectar eventos de teclado en Unity son `Input.GetKeyDown`, `Input.GetKey` e `Input.GetKeyUp`.

- **`Input.GetKeyDown(KeyCode tecla)`**: Se activa en el momento exacto en que se presiona una tecla.
- **`Input.GetKey(KeyCode tecla)`**: Se activa mientras se mantiene presionada una tecla.
- **`Input.GetKeyUp(KeyCode tecla)`**: Se activa en el momento exacto en que se libera una tecla.

```
void Update()
{
    if (Input.GetKeyDown(KeyCode.Space))    Debug.Log("Tecla Espacio presionada");
    if (Input.GetKey(KeyCode.A))            Debug.Log("Tecla A mantenida");
    if (Input.GetKeyUp(KeyCode.S))          Debug.Log("Tecla S liberada");
}
```

Eventos de teclado personalizados

```
public class ControlJugador : MonoBehaviour
{
    public delegate void EventoTeclado();           // delegado
    public static event EventoTeclado AlSaltar;     // evento usando delegado
    public static event EventoTeclado AlAgachar;    // evento usando delegado
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))        AlSaltar?.Invoke(); // detectar salto
        if (Input.GetKeyDown(KeyCode.LeftControl))  AlAgachar?.Invoke(); // detectar agacharse
    }
}
```

Puedes definir eventos que se disparen en función de ciertas combinaciones de teclas o estados del juego. Esto es especialmente útil cuando se requiere que múltiples sistemas respondan a una misma entrada.

```
public class EjemploSuscriptor : MonoBehaviour
{
    void OnEnable()           // suscribirse al evento
    {
        ControlJugador.AlSaltar += Saltar;
        ControlJugador.AlAgachar += Agachar;
    }
    void OnDisable()         // anular la suscripción al evento
    {
        ControlJugador.AlSaltar -= Saltar;
        ControlJugador.AlAgachar -= Agachar;
    }
    void Saltar()             { Debug.Log("Jugador ha saltado"); }
    void Agachar()            { Debug.Log("Jugador se ha agachado"); }
}
```

Eventos de teclado: combinaciones de teclas

Las combinaciones de teclas son esenciales en muchos tipos de juegos y aplicaciones para realizar acciones rápidamente. Unity facilita la detección de múltiples teclas presionadas al mismo tiempo a través del método `Input.GetKey()` junto con las declaraciones condicionales.

```
void Update()
{
    if (Input.GetKey(KeyCode.A) && Input.GetKey(KeyCode.D))
    {
        Debug.Log("Has pulsado A y D al mismo tiempo");
    }
    if (Input.GetKey(KeyCode.A) && Input.GetKey(KeyCode.S) && Input.GetKey(KeyCode.D))
    {
        Debug.Log("Has pulsado A, S y D al mismo tiempo");
    }
    if (Input.GetKey(KeyCode.LeftControl) && Input.GetKeyDown(KeyCode.S))
    {
        Debug.Log("Guardando el juego...");
    }
}
```

Ejemplo de movimiento de GameObject

Ejemplo de script que permite mover un GameObject en XZ con las flechas del teclado.

```
public class MoverConFlechas : MonoBehaviour
{
    public float velocidad = 5.0f;
    void Update()
    {
        float despHorizontal = 0;
        float despVertical = 0;
        if (Input.GetKey(KeyCode.UpArrow))        despVertical = 1;
        if (Input.GetKey(KeyCode.DownArrow))      despVertical = -1;
        if (Input.GetKey(KeyCode.LeftArrow))      despHorizontal = -1;
        if (Input.GetKey(KeyCode.RightArrow))     despHorizontal = 1;

        // calcular el vector de desplazamiento
        Vector3 desplazamiento = new Vector3(despHorizontal, 0, despVertical);

        // mover el GameObject
        transform.Translate(desplazamiento * velocidad * Time.deltaTime);
    }
}
```

Ejemplo de evento de seguimiento de la cámara

Ejemplo de script que permite a la cámara seguir a un GameObject en todo momento

```
public class SeguimientoCamara : MonoBehaviour
{
    public Transform objetivo;           // objeto que la cámara seguirá
    public float distancia = 10.0f;     // distancia entre la cámara y el objeto
    public Vector3 offset = new Vector3(0, 0, 0); // offset adicional

    void Update()
    {
        if (objetivo != null)
        {
            // calcular la nueva posición
            transform.position = objetivo.position - (objetivo.forward * distancia) + offset;
        }
    }
}
```

Introducción a C# en Unity:

Mini-juego cubos para 2 jugadores

Ejemplos de Eventos con el ratón

Crear un mini-juego para dos jugadores donde cada jugador deberá mover un cubo. Cada cubo deberá ser controlado por un jugador. Los requisitos son los siguientes:

Parte 1:

- Reutilizar el script de movimiento para que haga uso de 4 variables públicas que servirán para configurar las teclas que moverán al cubo. Reutilizar el script para los dos jugadores. La velocidad inicial será 10.
- Crear un script para que la cámara sea capaz de mantener visibles los cubos de cada uno de los jugadores en todo momento. Los cubos deberán estar encima de un plano.

Parte 2:

- Crear una moneda que debe aparecer en una posición aleatoria del plano cuando alguno de los cubos la toque. La moneda será un cilindro de deberá rotar constantemente. De forma aleatoria, al tocarla se podrá generar una moneda adicional, como máximo pueden haber 5 en pantalla.

Parte 3:

- Configurar un marcador para cada jugador. Dicho marcador mostrará la cantidad de puntos obtenidos por cada jugador.
- Crea distintos tipos de monedas: oro(5 puntos), plata(3 puntos), bronce(1 punto). Si un jugador recoge una moneda de oro, su velocidad se debe reducir durante 5 segundos. Se debe buscar un modo de que el juego termine, por tiempo, o por cantidad de puntos.

Ejemplos de Eventos al Hacer Click

Seguimiento de la cámara a ambos cubos:

```
public Transform cubo1;
public Transform cubo2;
public float margen = 5.0f;

void LateUpdate()
{
    Vector3 puntoMedio = (cubo1.position + cubo2.position) / 2;
    float distancia = (cubo1.position - cubo2.position).magnitude + margen;
    distancia = Mathf.Max(distancia, 10.0f); // distancia mínima 10 respecto al centro
    transform.position = puntoMedio - transform.forward * distancia;
}
```

Crear clon de la moneda

```
void CrearClonMoneda()
{
    Vector3 posicionAleatoria = new Vector3(Random.Range(1, 10), 0, Random.Range(1, 10));
    Quaternion rotacionInicial = Quaternion.Euler(90, 0, 0);
    Instantiate(prefabMoneda, posicionAleatoria, rotacionInicial);
}
```

Introducción a C# en Unity: Eventos de ratón

Eventos de ratón

Los eventos de ratón también son una parte crucial de la interacción en muchos videojuegos y aplicaciones interactivas. Al igual que con los eventos de teclado, la clase `Input` de Unity ofrece varios métodos para capturar estos eventos.

Los métodos más comunes para detectar eventos de ratón en Unity son `Input.GetMouseButtonDown`, `Input.GetMouseButton` e `Input.GetMouseButtonUp`.

- **`Input.GetMouseButtonDown(int boton)`**: detecta el momento en que se presiona un botón.
- **`Input.GetMouseButton(int boton)`**: detecta mientras se mantiene presionado un botón.
- **`Input.GetMouseButtonUp(int boton)`**: detecta el momento en que se suelta un botón.

Los botones del ratón se identifican con un número entero (0 para el botón izquierdo, 1 para el derecho y 2 para el botón del medio).

```
void Update()
{
    if (Input.GetMouseButtonDown(0)) Debug.Log("Botón izquierdo del ratón presionado");
    if (Input.GetMouseButton(1))      Debug.Log("Botón derecho del ratón mantenido");
    if (Input.GetMouseButtonUp(2))    Debug.Log("Botón central del ratón liberado");
}
```

Eventos de ratón personalizados

```
public class ControlRaton : MonoBehaviour
{
    public delegate void EventoRaton();           // delegado
    public static event EventoRaton AlHacerClic;  // evento usando delegado
    public static event EventoRaton AlLiberarClic; // evento usando delegado
    void Update()
    {
        if (Input.GetMouseButtonDown(0)) AlHacerClic?.Invoke(); // detectar pulsar
        if (Input.GetMouseButtonUp(0)) AlLiberarClic?.Invoke(); // detectar liberar
    }
}
```

Al igual que con los eventos de teclado, puedes definir eventos que se activan en función de ciertas acciones con el ratón. Esto permite que múltiples sistemas respondan a una misma entrada.

```
public class EjemploSuscriptorRaton : MonoBehaviour
{
    void OnEnable() // suscribirse al evento
    {
        ControlRaton.AlHacerClic += HacerClic;
        ControlRaton.AlLiberarClic += LiberarClic;
    }
    void OnDisable() // anular la suscripción al evento
    {
        ControlRaton.AlHacerClic -= HacerClic;
        ControlRaton.AlLiberarClic -= LiberarClic;
    }
    void HacerClic() { Debug.Log("Clic con el ratón"); }
    void LiberarClic() { Debug.Log("Liberado clic del ratón"); }
}
```

Posición del ratón

Además de los eventos de los botones del ratón, también podemos acceder a la posición del cursor en la pantalla a través de **Input.mousePosition**, que devuelve un vector Vector3 con las coordenadas x, y, z del cursor en coordenadas de pantalla.

```
void Update()
{
    if (Input.GetMouseButtonDown(0))
    {
        Ray rayo = Camera.main.ScreenPointToRay(Input.mousePosition);
        RaycastHit hitInfo;

        if (Physics.Raycast(rayo, out hitInfo))
        {
            GameObject obj = hitInfo.collider.gameObject;
            if (obj != null)
            {
                Debug.Log($"Has hecho click en: {obj.name}");
            }
        }
    }
}
```

```
void Update()
{
    if (Input.GetMouseButtonDown(0))
        Debug.Log("Posición del ratón: " + Input.mousePosition);
}
```

Para detectar un clic en un GameObject utilizamos **raycasting**. Debemos colocar el script en cualquier objeto de la escena (como la cámara principal). Al hacer clic en un GameObject que tenga un componente Collider o Collider2D se mostrará el mensaje por consola

Evento al Hacer Clic en un GameObject

Posición del ratón

En Unity, cuando realizas un raycast, este normalmente detecta el primer collider que encuentra en la dirección en la que se dispara el rayo. Si quieres que un raycast detecte todos los objetos que estén en la línea del rayo, puedes usar **Physics.RaycastAll**, éste devuelve un array de **RaycastHit** que contiene información sobre cada collider que el rayo ha tocado, ordenados por distancia desde el inicio del rayo.

```
void Update()
{
    if (Input.GetMouseButtonDown(0))
    {
        Ray rayo = Camera.main.ScreenPointToRay(Input.mousePosition);

        // lanzamos un raycast que detecte todos los colliders en la dirección del rayo
        RaycastHit[] impactos = Physics.RaycastAll(rayo, 100);

        foreach (RaycastHit impacto in impactos)
        {
            GameObject obj = impacto.collider.gameObject;
            Debug.Log($"El rayo impactó en: {obj.name}, distancia: {impacto.distance}");
        }
    }
}
```

Ejemplos de Eventos al Hacer Click

Evento al Hacer Clic en un Componente UI:

Supongamos que tienes un botón en tu escena y deseas que algo ocurra cuando el usuario haga clic en él. Unity facilita esto a través del sistema de eventos de su UI.

```
public class MoverJugadorUI: MonoBehaviour
{
    public Button botonArriba;
    public Button botonAbajo;
    public Button botonIzquierda;
    public Button botonDerecha;

    void Start()
    {
        botonArriba?.onClick.AddListener(Arriba);
        botonAbajo?.onClick.AddListener(Abajo);
        botonIzquierda?.onClick.AddListener(Izquierda);
        botonDerecha?.onClick.AddListener(Derecha);
    }
}
```

```
public class ClickBotonUI : MonoBehaviour
{
    public Button miBoton;

    void Start()
    {
        miBoton?.onClick.AddListener(TareaOnClick);
    }

    void TareaOnClick()
    {
        Debug.Log("Has hecho clic en el botón");
    }
}
```

Supongamos que tienes 4 botones en la UI y que cada uno sirve para mover el personaje en una de las 4 posibles direcciones.

Introducción a C# en Unity: ScreenToWorldPoint

ScreenToWorldPoint

ScreenToWorldPoint es una función muy útil en Unity que se utiliza para convertir una posición de la pantalla en una posición en el espacio del mundo 3D. Este método es esencial cuando se quiere interactuar con los objetos del espacio 3D. La pantalla en Unity se mide en píxeles y el origen (0,0) está en la esquina inferior izquierda. Por otro lado, el espacio del mundo se define en unidades de Unity, y el origen (0,0,0) está en el centro de la escena por defecto.

- El método **ScreenToWorldPoint** toma un argumento del tipo Vector3, donde x e y representan la posición en la pantalla, y z representa la distancia desde la cámara al plano en el que quieres proyectar el punto de la pantalla.
- **Transformación de perspectiva:** La cámara en Unity tiene una matriz de proyección que se utiliza para transformar los puntos del espacio del mundo en el espacio de la cámara, y viceversa. ScreenToWorldPoint utiliza la inversa de esta matriz para mapear las coordenadas de la pantalla a coordenadas del mundo.
 - Punto de entrada: Toma un Vector3, posición del ratón(x,y), profundidad que queremos calcular(z).
 - Proyección inversa: Internamente, ScreenToWorldPoint utiliza la matriz de proyección de la cámara para realizar una transformación inversa de las coordenadas de la pantalla a coordenadas del mundo.

ScreenToWorldPoint

Input.mousePosition en realidad devuelve un Vector3, pero es importante señalar que aunque este tipo de dato tiene componentes x, y y z, la posición del ratón en la pantalla solo utiliza los componentes x e y. La coordenada z de Input.mousePosition no representa una profundidad en el espacio 3D, sino la distancia desde la cámara al plano de la pantalla en unidades de Unity. Por defecto, este valor es 0.

Cuando se trabaja con Input.mousePosition y se quiere convertir esa posición a una posición en el espacio del juego, típicamente se usa la función **Camera.ScreenToWorldPoint()**. Esta función toma un Vector3, donde x e y son las coordenadas en pantalla y z es la distancia desde la cámara hasta el punto en el mundo 3D que quieres encontrar.

```
Vector3 posicionRatonEnPantalla = Input.mousePosition;
posicionRatonEnPantalla.z = 15f;
Vector3 posicionRatonEnMundo = Camera.main.ScreenToWorldPoint(posicionRatonEnPantalla);
Debug.Log($"El punto en el espacio del mundo es: {posicionRatonEnMundo }");
```

Si tienes un plano y quieres saber dónde en ese plano está apuntando el ratón, pondrías como z la distancia desde la cámara hasta ese plano.

Ejemplos de Eventos con el ratón

Mover y arrastrar un GameObject

Para mover un objeto en la escena de Unity al hacer clic y arrastrar, podemos hacerlo tanto la función `OnMouseDown` como la función `OnMouseDown`.

```
public class ArrastrarObjeto : MonoBehaviour
{
    private float z;
    void OnMouseDown()
    {
        z = Camera.main.WorldToScreenPoint(gameObject.transform.position).z;
    }
    private Vector3 ObtenerMouseAsWorldPoint()
    {
        Vector3 puntoRatonPantalla = new Vector3(Input.mousePosition.x, Input.mousePosition.y, z);
        return Camera.main.ScreenToWorldPoint(puntoRatonEnPantalla);
    }
    void OnMouseDown()
    {
        transform.position = ObtenerMouseAsWorldPoint();
    }
}
```

Introducción a C# en Unity: ScreenPointToRay

ScreenPointToRay

ScreenPointToRay crea un rayo desde la posición de la cámara pasando a través de un punto en la pantalla, en las coordenadas de píxeles proporcionadas. Este rayo puede utilizarse luego para realizar detecciones de colisión con objetos en el espacio del juego a través de raycasting. A diferencia de ScreenToWorldPoint, ScreenPointToRay no requiere que especifiques una profundidad porque el rayo tiene una dirección y se extiende infinitamente a través del espacio del juego.

```
Ray rayo = Camera.main.ScreenPointToRay(Input.mousePosition);
```

Debug.DrawRay es una función muy útil en Unity para visualizar rayos durante el desarrollo y depuración de juegos. Esta función dibuja una línea desde un punto de origen en una dirección específica y por una longitud determinada en el Editor de Unity. Es importante señalar que estos rayos solo se dibujan y se ven en el Editor y no en el juego compilado.

```
float longitudRayo = 10f;  
Debug.DrawRay(rayo.origin, rayo.direction * longitudRayo, Color.red);  
  
Vector3 origen = transform.position;  
Vector3 direccion = transform.forward; // dirección → hacia donde el objeto está mirando  
Debug.DrawRay(origen, direccion * longitudRayo, Color.green);
```

Ejemplos de Eventos con el ratón

Disparar un rayo - ScreenPointToRay

Un ejemplo típico donde se usa ScreenPointToRay es para disparar un rayo desde la posición de la cámara hacia donde el usuario hace clic en la pantalla. Este uso es común en juegos de disparos.

```
public class Disparar : MonoBehaviour
{
    public float alcanceDelRayo = 100.0f; // alcance máximo del rayo
    void Update()
    {
        Ray rayo = Camera.main.ScreenPointToRay(Input.mousePosition);
        RaycastHit hitInfo;
        if (Physics.Raycast(rayo, out hitInfo, alcanceDelRayo))
        {
            Debug.Log($"El rayo impactó en: {hitInfo.transform.name}");
            // ejemplo1: "destruir" el objeto golpeado
            Destroy(hitInfo.transform.gameObject);
            // ejemplo2: invocar una función en el objeto impactado
            hitInfo.transform.GetComponent<MiScript>().MiMetodo();
        }
        else Debug.Log("El rayo no impactó en nada dentro del alcance");
    }
}
```

Ejemplos de Eventos con el ratón

Seguimiento del Ratón

A menudo se quiere que un objeto siga la posición del cursor, por ejemplo una mirilla. Este es un ejemplo de script que muestra cómo lograrlo:

```
void Update()  
{  
    GetComponent<RectTransform>().position = Input.mousePosition;  
}
```

Para simular una mirilla en pantalla en Unity, podemos seguir los siguientes pasos.



1. Crear la textura de la mirilla.
2. Importar la textura a Unity.
3. Crear un UI Canvas
4. Añadir la imagen de la mirilla al Canvas
5. Ajustar la imagen de la mirilla.
6. Asegurar que la mirilla siga al ratón.
7. Crear funcionalidad de disparo.

Introducción a C# en Unity: Mini-juego de puntería

Mini-juego de puntería en Unity

Crear un mini-juego de puntería en el que debe aparecer una diana moviéndose. El jugador, a través de una mirilla deberá disparar lo más cerca posible del centro de la diana:

- La diana debe tener 5 áreas que determinarán la puntuación del disparo: blanco(1 punto), negro(2 puntos), azul (5 puntos), rojo (10 puntos), amarillo (25 puntos).
- La diana será móvil y se podrán mover en las 3 dimensiones. Para moverla se deberá reutilizar el ejercicio de patrón de movimiento que recibe un array de Vector3. Debemos poder configurar la velocidad de movimiento de la diana.
- En el momento que impactamos en la diana, se deberá cambiar el patrón de movimiento de la diana. Los distintos patrones de movimiento de deben almacenar en un array de arrays.
- Si conseguimos acertar en el centro(amarillo) la diana deberá realizar una animación de rotación.
- Configurar un límite de tiempo y límite de disparos en la partida.
- Incluir 3 marcadores: puntuación actual, disparos restantes y el tiempo restante. Realizar una primera versión donde los marcadores se deben actualizar sin hacer uso de eventos personalizados. Más tarde, modificar la versión, para que los marcadores se actualicen haciendo uso de eventos.

Introducción a C# en Unity:

Ejercicios repaso

Ejercicios en C#

Reutiliza los scripts utilizados para mover un GameObject a través de 4 KeyCode y para que la cámara siempre tenga visibles 2 GameObjects(cubos).

Requisitos Específicos:

- Debemos crear 2 cubos en la escena que podrán ser movidos a través del teclado.
- Cada cubo debe tener un texto flotante que muestre su posición actual.
- Utilizar un **LineRenderer** para dibujar una línea entre los dos cubos.
- Crear un texto adicional que debe mostrarse en el centro de la línea que une ambos cubos, indicando la distancia entre ellos.
- La línea debe actualizar su posición en tiempo real, siguiendo el movimiento de los cubos.
- Agrega propiedades públicas al script para personalizar el **LineRenderer**, incluyendo el ancho de la línea, color de inicio, color de fin y material.

Al final de este ejercicio, tendrás una escena interactiva donde los cubos pueden ser movidos, y la línea que los conecta, junto con los textos que muestran la distancia y las posiciones, se actualizan en tiempo real para reflejar los cambios en la escena.

Eventos en C#

- Para los ejercicios usar un plano con el logo de aulaenlanube.
- Plataformas para 2 jugadores, gana el primero en llegar. La cámara debe buscar la posición central entre los dos jugadores.
- Plataformas 3D llegar de A a B, primero sin saltos, pero con colliders.
- Luego con saltos, y con plataformas que se mueven, para 1 y 2 jugadores.
- Mover un cubo únicamente rotando de 90 en 90 grados y plantear un mini-juego con él.
- Medir distancia hasta llegar al objetivo.
- Crear mini-mapa del nivel.
- Plataformas 2D llegar de A a B, generar nivel de forma procedural en base a 3 dificultades.
- Circulo girando con el personaje en el centro, cronometrar el tiempo que tardamos en que nos toque alguno de los cubos que están girando. Crear distintos tipos de patrones.
- Patrón de giro del círculo, a distintas velocidades y frenando primero 1s, 2s, 5s y reiniciar.
- Espirales cuadradas formando laberinto.
- Laberintos.
- Espiral con la mínima cantidad de cubos, cambiando escala y posicionando de nuevo.

Introducción a C# en Unity: Recursividad

Recursividad en Unity

La recursividad es un concepto en la programación donde una función se llama a sí misma para resolver un problema más grande mediante la resolución de subproblemas más pequeños. En términos de sintaxis, una función recursiva no es diferente de cualquier otra función. La diferencia reside en cómo se llama a sí misma.

- **Caso Base:** El caso base es una condición dentro de la función recursiva que no requiere una llamada recursiva para ser resuelto. Sirve para detener la recursión, evitando así un bucle infinito.
- **Caso Recursivo:** El caso recursivo es donde la función se llama a sí misma, pero con un conjunto diferente de argumentos, acercándose cada vez más al caso base.

```
void Saludar(int n)
{
    if (n <= 0) // caso base
    {
        Debug.Log($"Bye");
    }
    else // caso recursivo
    {
        Debug.Log($"n:{n}");
        Saludar(--n);
    }
}
```

```
Saludar(4);
n:4
n:3
n:2
n:1
Bye
```

Ejemplos de recursividad en Unity

Consideraciones Importantes:

- **Eficiencia:** La recursividad puede ser menos eficiente que otros enfoques, especialmente en lenguajes donde la gestión de la pila es costosa. En Unity, la recursividad debe usarse con cuidado, ya que puede afectar el rendimiento si se manejan muchas llamadas recursivas por frame.
- **Profundidad de Recursión:** La profundidad de la recursión debe ser controlada para evitar el agotamiento de la pila, particularmente en juegos donde el rendimiento es crítico.

Caso de uso: método que permite desactivar todos los GameObjects hijo de un objeto, incluyendo los hijos de sus hijos, y así sucesivamente:

```
void DesactivarHijosRecursivamente(Transform padre)
{
    foreach (Transform hijo in padre)
    {
        hijo.gameObject.SetActive(false);
        DesactivarHijosRecursivamente(hijo);
    }
}
```

Ejemplos de recursividad en Unity

A continuación se muestran dos tipos de algoritmos recursivos muy utilizados:

- **Divide y vencerás:** Esta técnica divide el problema en subproblemas del mismo tipo y luego combina sus soluciones. Un ejemplo es el algoritmo de ordenamiento rápido (QuickSort).
- **Vuelta atrás(backtracking):** es una técnica de búsqueda utilizada para encontrar soluciones a problemas que pueden ser divididos en subproblemas más pequeños. Estos algoritmos funcionan revisando todas las posibles soluciones y haciendo «vuelta atrás» cuando encuentran una solución inválida, en lugar de continuar explorando esa rama.

[Más información](#)

Caso de uso: búsqueda de un GameObject por su nombre en una jerarquía de GameObjects.

```
GameObject BuscarObjetoRecursivamente(Transform padre, string nombreBuscado)
{
    if (padre.name == nombreBuscado) return padre.gameObject;
    foreach (Transform hijo in padre)
    {
        GameObject encontrado = BuscarObjetoRecursivamente(hijo, nombreBuscado);
        if (encontrado != null) return encontrado;
    }
    return null;
}
```


Ejercicio recursividad

Crea un método recursivo para generar una escalera formada por cubos, la escalera deberá generarse de modo que el primer escalón será de 1x1x1, el segundo de 1x2x1 y así sucesivamente, deberá avanzar sobre el eje X:

```
void GenerarEscalera(Vector3 posInicial, int alturaEscalon, int escalones)
{
    if (alturaEscalon > escalones) return;

    // instanciar un nuevo cubo
    GameObject cubo = GameObject.CreatePrimitive(PrimitiveType.Cube);
    cubo.transform.localScale = new Vector3(1, alturaEscalon, 1);
    cubo.transform.position = posInicial + new Vector3(0, alturaEscalon / 2f, 0);

    // avanzamos 1 unidad en el eje X
    posInicial += Vector3.right;

    // llamada recursiva para crear el siguiente escalón
    GenerarEscalera(posInicial, alturaEscalon + 1, escalones);
}
```

Introducción a C# en Unity:

Ejercicio repaso: recursividad, eventos y arrays

Ejercicio de repaso en Unity: recursividad, eventos y arrays

Crea un juego en Unity donde el jugador debe destruir grupos de cubos del mismo color. Cada vez que se hace clic en un cubo, todos los cubos adyacentes del mismo color deben ser destruidos de forma recursiva. El juego utilizará eventos para actualizar la puntuación, el tiempo transcurrido, los clics restantes y el combo actual(cantidad de cubos que se han destruido en el último clic). Requisitos:

- Crear un cuadrado formado por cubos. Los cubos deben tener colores aleatorios de una lista predefinida que crearemos desde el inspector. El lado del cuadrado debe ser impar(entre 7 y 21)
- Ajustar la cámara para que en todo momento los cubos sean visibles independientemente de la proporción de la pantalla, utilizar para ello en `Camera.main.aspect(ancho/alto)`.
- Al hacer clic en un cubo, se debe iniciar una búsqueda recursiva para destruir todos los cubos adyacentes del mismo color. Además, se deberá regenerar la cuadrícula de cubos después de que los cubos hayan sido destruidos, cambiando de forma aleatoria los colores de todos los cubos de la cuadrícula.
- Se deben utilizar eventos para actualizar los 4 textos de la interfaz de usuario. Puntuación, el tiempo transcurrido, la cantidad de clics y el combo actual.
- Por cada cubo destruido se conseguirá un punto.
- El juego termina cuando se alcanza el número máximo de clics permitidos, al terminar el juego podremos ver la puntuación y el tiempo transcurrido.

Introducción a C# en Unity:

P00: clases, herencia y polimorfismo

POO en C#

La programación orientada a objetos (POO) es un paradigma de programación que utiliza "objetos" para diseñar aplicaciones y programas de computadora. Los objetos son entidades que combinan estado (es decir, datos) y comportamiento (es decir, procedimientos o funciones). En el contexto de Unity y C#, este enfoque es fundamental para organizar y estructurar el código de manera eficiente y escalable.

Conceptos Clave de POO en C#:

- **Clase:** Una plantilla para crear objetos. Define las propiedades (atributos) y los métodos (acciones) que los objetos creados a partir de la clase pueden tener.
- **Objeto:** Una instancia de una clase.
- **Encapsulamiento:** La práctica de mantener los datos (variables) de un objeto ocultos y seguros de modificaciones externas, exponiendo solo métodos para interactuar con esos datos.
- **Herencia:** La capacidad de crear una nueva clase basada en una existente.
- **Polimorfismo:** La capacidad de tratar objetos de diferentes clases de manera similar, principalmente a través de la herencia.

P00 en C#

Imaginemos que estamos desarrollando un juego donde diferentes tipos de personajes tienen diferentes habilidades.

Paso 1: crear una clase base. Primero, definimos una clase base llamada Personaje. Esta clase tendrá propiedades comunes como vida y nombre, y un método para recibir daño.

```
public class Personaje
{
    public string nombre;    // no debe ser public
    public int vida;         // no debe ser public

    public Personaje(string nombre, int vida)
    {
        this.nombre = nombre;
        this.vida = vida;
    }
    public void RecibirGolpe(int n)
    {
        vida -= n;
        Debug.Log($"{nombre} ha recibido {n} puntos de daño");
    }
}
```

P00 en C#

```
public class Guerrero : Personaje
{
    public Guerrero(string nombre, int vida) : base(nombre, vida) { }
    public void AtaqueFuerte()
    {
        Debug.Log($"{nombre} realiza un ataque fuerte");
    }
}
```

```
public class Mago : Personaje
{
    public int puntosMana;
    public Mago(string nombre, int vida, int puntosMana) : base(nombre, vida)
    {
        this.puntosMana = puntosMana;
    }
    public void HechizoDeFuego()
    {
        if(puntosMana >= 10 )
        {
            Debug.Log($"{nombre} lanza un hechizo de fuego");
            puntosMana -= 10;
        }
        else Debug.Log($"{nombre} no tiene suficientes puntos de maná");
    }
}
```

Paso 2: Ahora, creamos dos clases derivadas: Guerrero y Mago. Estas clases heredarán de Personaje y tendrán habilidades específicas.

En C# **base** se utiliza para acceder a los miembros de la clase base

Getters y setters en C#

En el ejemplo anterior, las propiedades nombre y vida son públicas, lo cual no es una buena práctica de **encapsulamiento**. En general, las propiedades de un objeto deben ser privadas y accedidas a través de métodos públicos, para ello debemos utilizar getters y setters.

El encapsulamiento se refiere a la práctica de ocultar los detalles de la implementación de un objeto y exponer sólo lo que es seguro y necesario. En C# y, por extensión, en Unity, los getters y setters se utilizan para controlar el acceso a las propiedades de un objeto.

- **Getter:** Es un método que se utiliza para obtener el valor de una propiedad privada.
- **Setter:** Es un método que se utiliza para asignar un valor a una propiedad privada, permitiendo la validación o transformación del dato antes de asignarlo.

¿Por Qué Usarlos?

- **Seguridad:** Evita que el estado interno del objeto se modifique de manera imprevista o incorrecta.
- **Validación:** Permite verificar o modificar los datos antes de que sean asignados a la propiedad.
- **Control de Acceso:** Permite diferenciar entre la lectura y escritura de una propiedad.

Getters y setters en C#

```
public class Personaje
{
    private string nombre;
    private int vida;
    public Personaje(string nombre, int vida)
    {
        this.Nombre = nombre; // usa el setter
        this.Vida = vida;      // usa el setter
    }
    public string Nombre // getter y setter para nombre
    {
        get { return nombre; }
        set { nombre = value; } // 'value' en C# que representa el valor asignado
    }
    public int Vida //getter y setter para vida
    {
        get { return vida; }
        set { vida = value < 0 ? 0 : value; } // validación: la vida no puede ser negativa
    }
    public void RecibirGolpe(int n)
    {
        this.Vida -= n; // usa el setter con validación
    }
}
```

Vamos a tomar el ejemplo de la clase Personaje y agregar getters y setters para manejar las propiedades nombre y vida de manera más segura.

Getters y setters en Unity

Una práctica común es exponer propiedades en el Inspector de Unity para ser modificadas durante el diseño, pero controlar cómo se acceden y modifican durante la ejecución.

Por ejemplo, podrías querer que la vida de un personaje sea visible en el Inspector pero controlar cómo se modifica en tiempo de ejecución. Esto se puede lograr utilizando **[SerializeField]** junto con un campo privado y exponer un getter público.

```
public class Personaje : MonoBehaviour
{
    [SerializeField] private int vida;

    // ...
    public int Vida
    {
        get { return vida; }
        // el setter puede ser privado si solo quieres modificarlo internamente
        private set { vida = value < 0 ? 0 : value; }
    }
    // resto del código ...
}
```

Clases y métodos abstractos

Una clase abstracta es una clase que no se puede instanciar por sí misma y puede contener métodos abstractos. Un método abstracto es un método que se declara pero no se implementa en la clase abstracta, dejando la implementación a las clases derivadas.

Para añadir un método abstracto a la clase Personaje en nuestro ejemplo, primero necesitamos convertir Personaje en una clase abstracta.

```
public abstract class Personaje
{
    private string nombre;
    private int vida;

    // método abstracto
    public abstract void RealizarAccionEspecial();

    // resto del código ...
}
```

Ahora, cada vez que creamos una clase que herede de Personaje, estaremos obligados a definir cómo se comporta RealizarAccionEspecial() para esa clase específica. Esto asegura que cada subclase de Personaje tenga su propia implementación de la acción especial.

Polimorfismo

```
public class Mago : Personaje
{
    public Mago(string nombre, int vida) : base(nombre, vida) { }
    public override void RealizarAccionEspecial()
    {
        Debug.Log($"{Nombre} lanza un hechizo mágico");
    }
}
```

```
public class Guerrero : Personaje
{
    public Guerrero(string nombre, int vida) : base(nombre, vida) { }
    public override void RealizarAccionEspecial()
    {
        Debug.Log($"{Nombre} realiza un poderoso ataque");
    }
    public override void RecibirGolpe(int n) // sobrescritura de método
    {
        this.Vida -= Mathf.RoundToInt(n * 0.9f); // usa el setter con validación
        Debug.Log($"{nombre} ha recibido {n*0.9f} puntos de daño");
    }
}
```

Debe ser **virtual** en la clase base

Polimorfismo

El polimorfismo es un principio de la programación orientada a objetos que permite que objetos de diferentes clases sean tratados como objetos de una clase común. En el contexto de nuestro ejemplo con las clases Personaje, Guerrero y Mago, el polimorfismo se manifiesta en la capacidad de tratar tanto a los guerreros como a los magos como objetos de tipo Personaje, a pesar de sus diferencias en comportamiento y propiedades.

```
Personaje p1 = new Guerrero("Thorin", 100);
Personaje p2 = new Mago("Gandalf", 80);

p1.RealizarAccionEspecial();
p2.RealizarAccionEspecial();

p2.RecibirGolpe(50);
p1.RecibirGolpe(50);

p2.RecibirGolpe(50);
p1.RecibirGolpe(50);

Debug.Log($"{p1.Nombre}:{p1.Vida} - {p2.Nombre}:{p2.Vida}");
```

Polimorfismo

El polimorfismo es un principio de la programación orientada a objetos que permite que objetos de diferentes clases sean tratados como objetos de una clase común. En el contexto de nuestro ejemplo con las clases Personaje, Guerrero y Mago, el polimorfismo se manifiesta en la capacidad de tratar tanto a los guerreros como a los magos como objetos de tipo Personaje, a pesar de sus diferencias en comportamiento y propiedades.

```
Personaje p1 = new Guerrero("Thorin", 100);
Personaje p2 = new Mago("Gandalf", 80);

p1.RealizarAccionEspecial();      // ejecuta la versión de Guerrero
p2.RealizarAccionEspecial();      // ejecuta la versión de Mago

p2.RecibirGolpe(50);              // Gandalf ha recibido 50 puntos de daño
p1.RecibirGolpe(50);              // Thorin ha recibido 45 puntos de daño

p2.RecibirGolpe(50);              // Gandalf ha recibido 50 puntos de daño
p1.RecibirGolpe(50);              // Thorin ha recibido 45 puntos de daño

Debug.Log($"{p1.Nombre}:{p1.Vida} - {p2.Nombre}:{p2.Vida}"); // Thorin:10 - Gandalf:0
```

Introducción a C# en Unity: Colecciones

List<T>

Colecciones en C#: listas

Las colecciones son estructuras de datos que permiten almacenar y gestionar grupos de objetos. En Unity, las colecciones más comunes son los arrays, las listas y los diccionarios.

Las listas en C# son una de las estructuras de datos más versátiles y comúnmente usadas, especialmente en el desarrollo de juegos con Unity. Ofrecen flexibilidad para manejar colecciones de elementos, permitiendo agregar, eliminar y manipular elementos de manera eficiente. Son similares a los arrays pero son dinámicas, lo que significa que pueden cambiar de tamaño, muy útiles cuando no sabes cuántos elementos vas a necesitar. Ejemplo: si estás creando un juego donde los jugadores pueden recolectar objetos, una lista es ideal para almacenar estos objetos.

```
public class Recolector : MonoBehaviour
{
    public List<GameObject> objetosRecolectados = new List<GameObject>();

    private void RecolectarObjeto(GameObject objeto)
    {
        objetosRecolectados.Add(objeto);
    }
}
```


Colecciones en C#: declaración, inicialización y manipulación

Declarar e inicializar una lista implica hacer uso de la clase `List<T>`. Esta clase proporciona una manera de trabajar con colecciones de objetos. La `T` en `List<T>` representa un tipo de datos genérico, lo que significa que puedes crear listas de cualquier tipo de datos, por ejemplo: `int`, `string`, `GameObject`, `Personaje`, etc.

```
List<TipoDeDato> nombreDeLaLista;
```

```
List<int> listaEnteros;  
List<string> listaStrings;  
List<GameObject> listaGameObjects;  
List<Personaje> listaPersonajes;
```

```
List<int> listaEnteros = new List<int>();  
List<string> listaStrings = new List<string>();  
List<int> listaEnteros2 = new List<>(); // con inferencia de tipos(C# 7.1)  
List<string> listaStrings2 = new List<>(); // con inferencia de tipos(C# 7.1)  
var listaStrings3 = new List<string>();  
List<int> nums = new List<int> { 1, 3, 5, 7 };  
List<string> nombres = new List<string> { "Ana", "Pep", "Jon" };
```

List<T>: ejemplos métodos

Ejemplos de métodos útiles de la clase List<T>

```
var inventario = new List<string> {"Escudo", "Espada", "Arco", "Elixir", "Maza"};

inventario.Add("Escudo");           // Escudo-Espada-Arco-Elixir-Maza-Escudo
inventario.Remove("Escudo");       // Espada-Arco-Elixir-Maza-Escudo
if (inventario.Count > 1) inventario.RemoveAt(1); // Espada-Elixir-Maza-Escudo
bool tieneEspada = inventario.Contains("Espada"); // True
int cantidadItems = inventario.Count;           // 4
int indiceEspada = inventario.IndexOf("Espada"); // 0
int indiceEspada = inventario.IndexOf("espada"); // -1
inventario.Insert(0, "Poción"); // Poción-Espada-Elixir-Maza-Escudo
foreach (string item in inventario) Debug.Log(item); // muestra todos los elementos
inventario.Clear(); // elimina todos los elementos
```

Rendimiento: Aunque las listas proporcionan flexibilidad al permitir que se agreguen o eliminen elementos, es importante tener en cuenta el rendimiento. Por ejemplo, agregar elementos al final de la lista es rápido, pero insertar o eliminar elementos en medio de la lista puede ser más lento porque puede requerir desplazar otros elementos.

List<T>: ordenación

Para ordenar listas en C#, puedes utilizar el método Sort de la clase List<T>.

```
List<int> numeros = new List<int> { 3, 1, 4, 1, 5, 9, 2, 6, 5 };
numeros.Sort((a, b) => b.CompareTo(a)); // ordena de mayor a menor
numeros.ForEach(n => Debug.Log(n)); // 9, 6, 5, 5, 4, 2, 3, 1, 1
```

```
List<string> palabras = new List<string> { "aab", "aa", "abc", "b", "aaa", "a" };
palabras.Sort(); // ordena alfabéticamente
palabras.ForEach(s => Debug.Log(s)); // a aa aaa aab abc b
```

```
List<string> palabras = new List<string> { "aab", "aa", "abc", "b", "aaa", "a" };
palabras.Sort((a,b) => a.Length.CompareTo(b.Length)); //ordena por número de caracteres
palabras.ForEach(s => Debug.Log(s)); // b a aa aab abc aaa
```

```
palabras.OrderBy(s => s.Length).ToList().ForEach(s => Debug.Log(s)); // con System.Linq
```

```
personajes.Sort((personaje1, personaje2) => {
    int resultado = personaje2.Fuerza.CompareTo(personaje1.Fuerza);
    if (resultado == 0) resultado = personaje2.Vida.CompareTo(personaje1.Vida);
    return resultado;
}); // ordena por Fuerza descendente y luego por Vida descendente
```

Ejercicio List<T>

Crea un método que genere N cubos de un tamaño aleatorio entre escalaMin y escalaMax, que sean de colores aleatorios, y que los ordene por tamaño y los coloque uno detrás de otro en el eje z. El más pequeño irá el primero, y el más grande el último.

```
void GenerarCubos(int n, float escalaMin, float escalaMax)
{
    var cubos = new List<GameObject>();
    for (int i = 0; i < n; i++)
    {
        float escalaCubo = Random.Range(escalaMin, escalaMax);
        GameObject cubo = GameObject.CreatePrimitive(PrimitiveType.Cube);
        cubo.transform.localScale = new Vector3(escalaCubo, escalaCubo, escalaCubo);
        cubo.GetComponent<Renderer>().material.color = Random.ColorHSV();
        cubos.Add(cubo);
    }
    cubos.Sort((c1, c2) =>
c1.transform.localScale.magnitude.CompareTo(c2.transform.localScale.magnitude));
    float z = 0f;
    foreach (GameObject cubo in cubos)
    {
        cubo.transform.position = new Vector3(0, cubo.transform.localScale.y/2, z);
        z += cubo.transform.localScale.z;
    }
}
```

Introducción a C# en Unity: Colecciones

HashSet<T>

Colecciones en C#: HashSet<T>

HashSet<T> en C# es una colección que almacena elementos únicos, es decir, no permite duplicados, y se basa en una tabla hash para almacenar sus elementos.

- Si intentas añadir un elemento que ya existe en el conjunto, la operación no tendrá ningún efecto, y el método Add devolverá false.
- HashSet<T> es **altamente eficiente** para operaciones de búsqueda, inserción y eliminación, gracias a su uso interno de una tabla hash, tiempo promedio de $O(1)$.
- HashSet<T> utiliza los métodos **Equals** y **GetHashCode** para determinar si dos objetos son iguales. Por lo tanto, si estás usando tipos personalizados, debes implementarlos.
- Los elementos en un HashSet<T> **no tienen un orden específico**. El orden de los elementos puede variar cada vez que se modifican los contenidos del conjunto.
- A diferencia de las listas o arrays, los HashSet<T> **no permiten el acceso a sus elementos mediante un índice**.

```
HashSet<int> listaNumeros = new HashSet<int>();  
HashSet<string> listaJugadores = new HashSet<string>();  
HashSet<GameObject> listaObjetos = new HashSet<GameObject>();  
HashSet<Enemy> listaEnemigos = new HashSet<Enemy>();
```

Colecciones en C#: HashSet<T>

HashSet<T> es ideal para situaciones en las que la unicidad de los elementos es crucial y se requieren operaciones rápidas de inserción y búsqueda, lo cual es común en varios aspectos del desarrollo de videojuegos con Unity.

```
HashSet<int> listaNumeros = new HashSet<int> {1, 5, 8, 8, 1};  
listaNumeros.Add(4);  
listaNumeros.Add(5);  
listaNumeros.Add(6);  
listaNumeros.Remove(1);  
foreach (int num in listaNumeros) Debug.Log(num); // 5 8 4 6
```

```
HashSet<string> jugadores = new HashSet<string>();  
jugadores.Add("J1");  
jugadores.Add("J2");  
jugadores.Add("J2");  
jugadores.Add("J3");  
foreach (string jugador in jugadores) Debug.Log(jugador); // J1 J2 J3
```

Colecciones en C#: HashSet<T>

Ejemplo de HashSet que muestra, agrega y elimina enemigos.

```
private HashSet<GameObject> enemigos = new HashSet<GameObject>();
public void MostrarEnemigos()
{
    foreach (GameObject enemigo in enemigos) Debug.Log($"Enemigo:{enemigo.name}");
}
public void AgregarEnemigo(GameObject enemigo)
{
    if (enemigos.Add(enemigo)) Debug.Log($"Enemigo añadido: {enemigo.name}");
    else Debug.Log($"El enemigo ya estaba en el conjunto: {enemigo.name}");
}
public void EliminarEnemigo(GameObject enemigo)
{
    if (enemigos.Remove(enemigo)) Debug.Log($"Enemigo eliminado: {enemigo.name}")
    else Debug.Log($"El enemigo no se encontró en el conjunto: {enemigo.name}");
}
```


Colecciones en C#: Equals y GetHashCode

Por defecto la implementación de Equals y GetHashCode identifica instancias distintas

```
public class Enemigo
{
    private string nombre;
    private int vida;

    public Enemigo(string nombre, int vida)
    {
        this.nombre = nombre;
        this.vida = vida;
    }

    // getters, setters y otros métodos
}
```

```
Enemigo enemigo1 = new Enemigo("Orco", 100);
Enemigo enemigo2 = new Enemigo("Orco", 80);
Enemigo enemigo3 = new Enemigo("Orco", 100);
enemigo1.Equals(enemigo2); // false
enemigo1.Equals(enemigo3); // false
```

Colecciones en C#: Equals y GetHashCode

Implementación de Equals y GetHashCode que tiene en cuenta el nombre y vida

```
public class Enemigo
{
    private string nombre;
    private int vida;
```

// constructores, getters, setters y otros métodos

```
public override bool Equals(object obj)
{
    return obj is Enemigo enemigo
        && nombre == enemigo.nombre
        && vida == enemigo.vida
}
public override int GetHashCode()
{
    return HashCode.Combine(nombre, vida);
}
```

```
Enemigo enemigo1 = new Enemigo("Orco", 100);
Enemigo enemigo2 = new Enemigo("Orco", 80);
Enemigo enemigo3 = new Enemigo("Orco", 100);
enemigo1.Equals(enemigo2); // false
enemigo1.Equals(enemigo3); // true
```

Colecciones en C#: Equals y GetHashCode

Implementación de Equals y GetHashCode que tiene en cuenta el nombre:

```
public class Enemigo
{
    private string nombre;
    private int vida;
```

// constructores, getters, setters y otros métodos

```
public override bool Equals(object obj)
```

```
{
    return obj is Enemigo enemigo && nombre == enemigo.nombre;
}
```

```
public override int GetHashCode() { return nombre.GetHashCode(); }
```

```
public override string ToString()
```

```
{
    return $"Nombre: {nombre}, Puntos de Vida: {vida}";
}
}
```

```
Enemigo enemigo1 = new Enemigo("Orco", 100);
Enemigo enemigo2 = new Enemigo("Elfo", 100);
Enemigo enemigo3 = new Enemigo("Orco", 90);
enemigo1.Equals(enemigo2); // false
enemigo1.Equals(enemigo3); // true
```

Introducción a C# en Unity: Colecciones

Dictionary<K,V>

Diccionarios en C#: declaración e inicialización

Un diccionario almacena elementos en pares clave-valor. Las claves son únicas y se utilizan para buscar un elemento dentro de la colección.

Para inicializar un diccionario debemos utilizar dos genéricos, una para el tipo de la clave, y otro para el tipo de valor.

```
Dictionary<TipoClave, TipoValor> nombreDiccionario;
```

Por ejemplo, si quieres un diccionario para asociar una string (como el nombre de un enemigo) con un GameObject (el enemigo en sí en el juego):

```
Dictionary<string, GameObject> diccionarioEnemigos = new Dictionary<string, GameObject>();
```

Podemos inicializar el diccionario con un conjunto de valores iniciales:

```
Dictionary<string, GameObject> diccionarioEnemigos = new Dictionary<string, GameObject>
{
    { "Orco", orcoGameObject },
    { "Elfo", elfoGameObject }
};
```

Diccionarios en C#: agregar pares

Un diccionario almacena elementos en pares clave-valor. Las claves son únicas y se utilizan para buscar un elemento dentro de la colección. Podemos añadir pares a través del método `Add(clave, valor)`. Si intentas añadir un nuevo elemento a un diccionario utilizando una clave que ya existe en ese diccionario, se lanzará una excepción **ArgumentException**. Esto se debe a que en un diccionario, cada clave debe ser única.

Solución 1: comprobar si la clave ya existe utilizando el método `ContainsKey`.

```
if (miDiccionario.ContainsKey(laClave))
{
    miDiccionario[laClave] = elValor;
    Debug.Log("Se ha actualizado el valor de la clave");
}
else
{
    miDiccionario.Add(laClave, elValor);
    Debug.Log("Se ha agregado un nuevo par al diccionario");
}
```

Diccionarios en C#: agregar pares

Si intentas añadir un nuevo elemento a un diccionario utilizando una clave que ya existe en ese diccionario, se lanzará una excepción `ArgumentException`. Esto se debe a que en un diccionario, cada clave debe ser única.

Solución 2: capturar la excepción. Sin embargo, este enfoque generalmente no es el más recomendable, manejar excepciones suele ser más costoso en términos de rendimiento.

```
try
{
    miDiccionario.Add(laClave, elValor);
}
catch (ArgumentException)
{
    Debug.Log("No se ha podido añadir el par, clave duplicada");
}
```

```
miDiccionario[laClave] = elValor;
```

Diccionarios en C#: métodos para iteración

Los diccionarios en C# podemos recorrerlos fácilmente con un foreach.

```
Dictionary<string, int> enemigos = new Dictionary<string, int>
{ {"Orco", 100}, {"Elfo", 150}, {"Troll", 200} };

foreach (KeyValuePair<string, int> enemigo in enemigos)
{
    Debug.Log($"Enemigo: {enemigo.Key}, Puntuación: {enemigo.Value}");
}
```

```
foreach (string nombre in enemigos.Keys)
{
    Debug.Log($"Nombre: {nombre}");
}
```

```
foreach (int vida in enemigos.Values)
{
    Debug.Log($"Vida: {vida}");
}
```


Diccionarios en C#: borrar pares

Podemos borrar un elemento del diccionario a través de su clave

```
Dictionary<string, int> enemigos = new Dictionary<string, int>
{ {"Orco1", 100},
  {"Orco2", 150},
  {"Elfo", 150},
  {"Troll1", 100},
  {"Troll2", 200}
};
```

```
if(diccionario.Remove("Orco1"))
    Debug.Log("Se ha borrado Orco1 del diccionario");
else
    Debug.Log("Orco1 no existe en el diccionario");
```

```
int vidaMinima = 200; // borramos enemigos con menos de 200 de vida
List<string> clavesAEliminar = new List<string>();
foreach (var par in enemigos)
{
    if (par.Value < vidaMinima) clavesAEliminar.Add(par.Key);
}
foreach (var clave in clavesAEliminar)
{ enemigos.Remove(clave); }
```

Colecciones en C#: métodos en diccionarios

```
Dictionary<string, int> inventario = new Dictionary<string, int>();
inventario.Add("poción", 5);
inventario.Add("elixir", 3);

if (inventario.TryGetValue("poción", out int cantidadActual)) //devuelve booleano
    inventario["poción"] = cantidadActual + 2;

if (inventario.ContainsKey("poción"))
    Debug.Log($"El jugador tiene alguna poción en el inventario");

if (inventario.TryGetValue("poción", out int valor))
    Debug.Log($"El jugador tiene {valor} pociones en el inventario");

if (inventario.ContainsValue(3))
    Debug.Log($"El jugador tiene cantidad {valor} de algún elemento del inventario");

Debug.Log($"El jugador tiene {inventario.Count} objetos distintos en el inventario");

foreach (KeyValuePair<string, int> item in inventario)
    Debug.Log($"Item: {item.Key}, Cantidad: {item.Value}");
```

Colecciones en C#: Dictionarios vs HashSet

Un diccionario podríamos implementarlo de forma equivalente con un HashSet:

```
public class Enemigo
{
    private string nombre;
    private int vida;

    // constructores, getters, setters y otros métodos
    public override bool Equals(object obj)
    {
        return obj is Enemigo enemigo && nombre == enemigo.nombre;
    }
    public override int GetHashCode() { return nombre.GetHashCode(); }
}
```

```
void Start()
{
    Dictionary<string, int> enemigos2 = new Dictionary<string, int>();
    HashSet<Enemigo> enemigos2 = new HashSet<Enemigo>();
}
```

Colecciones en C#: ejercicio diccionario

Tenemos un diccionario que simula un inventario:

```
public Dictionary<string, int> items = new Dictionary<string, int>();
```

Debes crear los métodos agregar y eliminar item. El diccionario utilizará como clave el nombre del item, y como valor, la cantidad de ítems de ese tipo. Si un ítem ya existe y lo volvemos a añadir se deberá actualizar la cantidad de ese ítem. Al eliminar items sucederá lo mismo, se deberá decrementar la cantidad de ítems de ese tipo a avisar si no existen ítems de ese tipo.

```
public void AgregarItem(string nombre, int cantidad)
{
    if (items.ContainsKey(nombre)) items[nombre] += cantidad;
    else items[nombre] = cantidad;
}

public void EliminarItem(string nombre, int cantidad)
{
    if (items.ContainsKey(nombre))
        if (items[nombre] >= cantidad) items[nombre] -= cantidad;
        else Debug.Log($"No hay suficientes ítems de tipo {nombre} en el inventario");
    else Debug.Log($"No hay ítems de tipo {nombre} en el inventario");
}
```

Introducción a C# en Unity:

Ejercicio P00: gestión de inventario

Ejercicio: Gestión de inventario

Desarrollar un sistema de inventario que gestione distintos tipos de objetos, incluyendo armas, armaduras, consumibles, y artefactos. El sistema debe ser capaz de almacenar de cada objeto: la cantidad, un GameObject, una descripción y la rareza de cada objeto, así como su coste de venta en monedas (oro, plata, bronce). Requisitos técnicos:

- Cada objeto tiene un nombre único, dos objetos iguales tienen el mismo nombre.
- Dos objetos con un mismo nombre pueden tener valores distintos en sus atributos.
- Categorías de Objetos: Armas, Armaduras, Consumibles, y artefactos.
- Rarezas: Común, Raro, Épico, y Legendario.
- Las armas pueden mejorarse y tienen 4 niveles:
 - Inicial, avanzado, élite y leyenda.
 - Atributos: dps, velocidad de ataque, durabilidad, alcance
- Las armaduras pueden almacenar valores para los atributos, defensa, agilidad y durabilidad.
- La armadura puede pertenecer a distintas partes: centro, cabeza, brazos, manos, piernas, pies.
- Los artefactos tienen una propiedad especial.
- Los consumibles tienen un efecto, un porcentaje consumido y una duración del efecto.

Gestión de Inventario:

- MostrarInventario: muestra cada uno de los objetos del inventario con toda su información ordenados de mayor a menor valor. Crear una sobrecarga para mostrar los objetos de una rareza concreta.
- AgregarObjeto: añade objetos al inventario.
- EliminarObjeto: elimina un objeto del inventario.

Clase ObjetoInventario

```
public enum Rareza { Comun, Raro, Epico, Legendario }

public abstract class ObjetoInventario
{
    public string Nombre { get; private set; }
    public string Descripcion { get; set; }
    public Rareza Rareza { get; set; }
    public GameObject ObjetoVisual { get; set; }
    public int CosteOro { get; set; }
    public int CostePlata { get; set; }
    public int CosteBronce { get; set; }

    protected ObjetoInventario(string nombre, string descripcion, Rareza rareza, GameObject
objetoVisual, int costeOro, int costePlata, int costeBronce)
    {
        //...
    }
    public override bool Equals(object obj)
    {
        return obj is ObjetoInventario objInventario && Nombre == objInventario.Nombre;
    }
    public override int GetHashCode() { return Nombre.GetHashCode(); }
    public override string ToString() { //... }
}
```

Clase Arma

```
public enum NivelArma { Inicial, Avanzado, Elite, Leyenda }

public class Arma : ObjetoInventario
{
    public NivelArma Nivel { get; set; }
    public float DPS { get; set; }
    public float VelocidadAtaque { get; set; }
    public int Durabilidad { get; set; }
    public float Alcance { get; set; }

    public Arma(string nombre, string descripcion, Rareza rareza, GameObject objetoVisual, int
costeOro, int costePlata, int costeBronce, NivelArma nivel, float dps, float velocidadAtaque, int
durabilidad, float alcance)
        : base(nombre, descripcion, rareza, objetoVisual, costeOro, costePlata, costeBronce)
    {
        Nivel = nivel;
        DPS = dps;
        VelocidadAtaque = velocidadAtaque;
        Durabilidad = durabilidad;
        Alcance = alcance;
    }
    public override string ToString()    { //... }
}
```


Clase Armadura

```
public enum ParteArmadura { Cabeza, Central, Brazos, Manos, Piernas, Pies }

public class Armadura : ObjetoInventario
{
    public int Defensa { get; set; }
    public int Agilidad { get; set; }
    public int Durabilidad { get; set; }
    public ParteArmadura Parte { get; set; }

    public Armadura(string nombre, string descripcion, Rareza rareza, GameObject objetoVisual, int
costeOro, int costePlata, int costeBronce, int defensa, int agilidad, int durabilidad,
ParteArmadura parte)
        : base(nombre, descripcion, rareza, objetoVisual, costeOro, costePlata, costeBronce)
    {
        Defensa = defensa;
        Agilidad = agilidad;
        Durabilidad = durabilidad;
        Parte = parte;
    }

    public override string ToString()    { //...  }
}
```

Clase Consumible

```
public class Consumible : ObjetoInventario
{
    private string efecto;
    private float porcentajeRestante;
    private float duracionEfecto;

    public string Efecto
    {
        get => efecto;
        set => efecto = value;
    }
    public float DuracionEfecto
    {
        get => duracionEfecto;
        set => duracionEfecto = value;
    }

    public float PorcentajeRestante
    {
        get => porcentajeRestante;
        set => porcentajeRestante = Mathf.Clamp(value, 0, 100); // limitar valor entre 0 y 100
    }
    //constructor y toString
}
```

Clase Artefacto

```
public class Artefacto : ObjetoInventario
{
    public string PropiedadEspecial { get; set; }

    public Artefacto(string nombre, string descripcion, Rareza rareza, GameObject objetoVisual,
int costeOro, int costePlata, int costeBronce, string propiedadEspecial)
        : base(nombre, descripcion, rareza, objetoVisual, costeOro, costePlata, costeBronce)
    {
        PropiedadEspecial = propiedadEspecial;
    }

    public override string ToString()
    {
        return base.ToString() + $"Propiedad especial: {PropiedadEspecial}\n";
    }
}
```

Clase Inventario

```
public class Inventario
{
    private Dictionary<string, List<ObjetoInventario>> inventario;
    public Inventario() { inventario = new Dictionary<string, List<ObjetoInventario>>(); }
    public void MostrarInventario()
    {
        foreach (var listaActualObjetos in inventario.Values)
            foreach (var objetoActualLista in listaActualObjetos)
                Debug.Log(objetoActualLista.ToString());
    }
    public void MostrarInventarioPorValor() { //... }
    public void MostrarInventarioPorRareza(Rareza rareza) { //... }
    public void AgregarObjeto(ObjetoInventario objetoNuevo)
    {
        if (inventario.ContainsKey(objetoNuevo.Nombre)) inventario[objetoNuevo.Nombre].Add(objetoNuevo);
        else inventario.Add(objetoNuevo.Nombre, new List<ObjetoInventario>() { objetoNuevo });
    }
    public void EliminarObjeto(ObjetoInventario objetoBorrado)
    {
        if (inventario.ContainsKey(objetoBorrado.Nombre))
        {
            if (inventario[objetoBorrado.Nombre].Count > 1) inventario[objetoBorrado.Nombre].Remove(objetoBorrado);
            else if (inventario[objetoBorrado.Nombre].Count == 1) inventario.Remove(objetoBorrado.Nombre);
            return;
        }
        Debug.Log("El objeto con el Nombre proporcionado no existe en el inventario.");
    }
}
```

Introducción a C# en Unity: Interfaces

Interfaces en C#

Las interfaces en programación, incluyendo en el contexto del desarrollo de videojuegos con Unity, son una herramienta esencial para la creación de sistemas flexibles y mantenibles. En términos simples, una interfaz es una "plantilla de contrato" para las clases que la implementan. Define un conjunto de métodos y propiedades que las clases deben implementar, pero sin proporcionar una implementación concreta. Esto permite que diferentes clases puedan ser tratadas de manera uniforme bajo una misma interfaz.

Conceptos Clave de Interfaces:

- **Definición de Interfaz:** Una interfaz es una colección de firmas de métodos y/o propiedades. No contiene lógica de negocio ni implementación.
- **Implementación de la Interfaz:** Las clases que implementan una interfaz deben proporcionar una implementación concreta de cada método y propiedad definidos en la interfaz.
- **Polimorfismo:** Las interfaces son cruciales para el polimorfismo. Permiten que un objeto sea tratado como si fuera de diferentes tipos. Por ejemplo, diferentes objetos que implementan la misma interfaz pueden ser usados indistintamente.
- **Desacoplamiento:** Las interfaces ayudan a desacoplar el código, facilitando el mantenimiento y la escalabilidad.

Interfaces en C#

Supongamos que estamos desarrollando un juego y queremos tener diferentes tipos de enemigos con comportamientos distintos, pero todos deben tener una función de ataque. Podemos definir una interfaz IEnemigo que obligue a las clases que la implementen a tener un método Atacar.

```
public interface IEnemigo
{
    void Atacar();
}
```

Los métodos en una interfaz son implícitamente **public**

Luego, podríamos tener diferentes clases de enemigos:

```
public class Zombi : IEnemigo
{
    public void Atacar() { Debug.Log("El zombi ataca"); }
}

public class Robot : IEnemigo
{
    public void Atacar() { Debug.Log("El robot dispara"); }
}
```

Interfaces en C#

Los métodos de las interfaces permiten el polimorfismo, lo que significa que un objeto de una clase que implementa una interfaz puede ser referenciado o tratado como si fuera un objeto de tipo interfaz. En el siguiente ejemplo, nos permite interactuar con diferentes tipos de enemigos de manera uniforme.

```
public class EjemploInterfaces : MonoBehaviour
{
    private List<IEnemigo> enemigos;

    void Start()
    {
        enemigos = new List<IEnemigo> { new Zombi(), new Robot() };
        AtaqueMasivo();
    }

    public void AtaqueMasivo()
    {
        foreach (var enemigo in enemigos) enemigo.Atacar();
    }
}
```


Implementación múltiple de interfaces en C#

A diferencia de la herencia de clases, una clase en C# puede implementar múltiples interfaces. Esto permite una forma de herencia múltiple de tipos, proporcionando mucha flexibilidad en el diseño de clases.

```
public class Zombi : IEnemigo, IGolpeable
{
    public void Atacar()
    {
        Debug.Log("El zombi ataca");
    }
    public void RecibirGolpe(int fuerza)
    {
        Debug.Log($"El zombi ha sido golpeado con una fuerza de {fuerza}");
    }
}
public class Robot : IEnemigo, IGolpeable
{
    public void Atacar() { Debug.Log("El robot dispara"); }
    public void RecibirGolpe(int fuerza)
    {
        Debug.Log($"El robot ha sido golpeado con una fuerza de {fuerza}");
    }
}
```

```
public interface IGolpeable
{
    void RecibirGolpe(int fuerza);
}
```

Propiedades en interfaces en C#

Las interfaces pueden incluir definiciones de propiedades, pero no pueden contener campos (atributos) con estado. Una propiedad en una interfaz se define mediante un conjunto de métodos de acceso (get y/o set), sin implementación específica. Las clases que implementan la interfaz deben proporcionar implementaciones concretas para estos métodos de acceso.

```
public class Zombi : IEnemigo, IGolpeable
{
    private int resistencia;
    public int Resistencia
    {
        get { return resistencia; }
        set { resistencia = Math.Max(0, value); } // no permite valores negativos
    }
    public Zombi(int resistencia) { Resistencia = resistencia; }
    public void Atacar() { Debug.Log("El zombi ataca"); }
    public void RecibirGolpe(int fuerza)
    {
        int fuerzaReal = fuerza - resistencia;
        Debug.Log($"El zombi ha sido golpeado con una fuerza de {fuerzaReal}");
    }
}
```

```
public interface IGolpeable
{
    int Resistencia { get; set; }
    void RecibirGolpe(int fuerza);
}
```

Interfaces en C#

```
private List<IEnemigo> enemigos;  
private List<IGolpeable> enemigosGolpeables;  
  
void Start()  
{  
    enemigos = new List<IEnemigo> { new Zombi(10), new Zombi(20), new Robot() };  
    enemigosGolpeables = new List<IGolpeable> { new Zombi(10), new Zombi(20) };  
    AtaqueEnemigos();  
    GolpearEnemigos(30);  
}  
  
public void AtaqueEnemigos()  
{  
    foreach (var enemigo in enemigos) enemigo.Atacar();  
}  
  
public void RecibirGolpeMasivo(int fuerza)  
{  
    foreach (var enemigo in enemigosGolpeables) enemigo.RecibirGolpe(fuerza);  
}
```

Introducción a C# en Unity: Inventario avanzado

Ejercicio interfaces: inventario avanzado

Modificar el sistema de inventario desarrollado anteriormente para añadir funcionalidades adicionales. El nuevo inventario permitirá el comercio y la interacción con comerciantes en el juego.

- Crear interfaces para representar diferentes tipos de comportamientos y propiedades de los objetos en el inventario y las interacciones comerciales.
- Mejorar la forma de identificar cada objeto.
- Mejorar la forma de almacenar la cantidad de cada objeto.
- Implementar un sistema de inventario que pueda almacenar, agregar y remover objetos.
- Implementar un sistema de comercio que permita a los jugadores comprar y vender objetos. Los objetos del inventario pueden ser comerciables o no serlo.
- Implementar un sistema de combinaciones que permite combinar varios objetos en uno.
- Implementar un sistema de mejoras que permite mejorar los atributos de un objeto.

Algunas posibles interfaces pueden ser las siguientes:

- `IIventario`: métodos para agregar,remover y listar objetos del inventario.
- `IComerciable`: propiedad de precio, método para realizar transacciones (compra/venta).
- `IInteractuable`: método para interactuar con el objeto (usar, equipar, etc.).
- `IComerciante`: métodos para comprar y vender objetos. Inventario de objetos en venta.
- `ICombinable`: métodos para combinar objetos.
- `IMejorable`: métodos que permiten modificar los atributos de un objeto.
- `IConsumible`: métodos que permiten consumir objetos.

Ejercicio interfaces: inventario avanzado

La necesidad de un identificador único (id) para cada objeto en `ObjetoInventario` depende de cómo planeas usar estos objetos en el juego y cómo está diseñado el sistema de inventario.

Uso de la Propia Instancia como Identificador

- **Unicidad Garantizada:** Cada instancia de un objeto en C# ya tiene una identidad única en el sentido de que no hay dos instancias exactamente iguales en la memoria al mismo tiempo. Si utilizas la instancia en sí como identificador, estás aprovechando esta unicidad natural.
- **Simplicidad:** Evitar el uso de un ID adicional puede simplificar tu código, ya que reduces una capa de abstracción. No necesitas generar, asignar ni gestionar identificadores únicos.
- **Comparación de Objetos:** La comparación de objetos se basa en la instancia y no en un campo de identificación. Esto puede ser tanto una ventaja como una desventaja, dependiendo de la lógica de tu juego.

Uso de un Identificador Único (id)

- **Persistencia y Serialización:** Si necesitas guardar y cargar el estado del juego (por ejemplo, guardar el inventario en un archivo y luego cargarlo), tener un id único puede facilitar este proceso, especialmente si el estado de los objetos puede cambiar o si los objetos pueden ser intercambiables.
- **Identificación Independiente del Estado del Objeto:** Un id permite identificar un objeto independientemente de su estado actual. Por ejemplo, dos pociones de salud pueden ser "iguales" en tipo y función, pero una podría estar parcialmente usada. Un id ayuda a distinguir entre estas instancias.
- **Interacción con Sistemas Externos:** Si tu juego interactúa con sistemas externos (como bases de datos, servidores de juegos, etc.), un id puede ser crucial para rastrear objetos a través de estos sistemas.

Ejercicio interfaces: inventario avanzado

Crear un tipo enumerado (enum) con un valor para cada objeto posible en tu juego puede ser una buena idea en ciertos contextos, pero también tiene limitaciones y desventajas que debes considerar. Veamos los pros y los contras:

Ventajas de Usar un Enum para Cada Objeto

- Conjunto claro y definido de valores que hacen el código más legible y fácil de entender.
- Reduce la probabilidad de errores al escribir, ya que el compilador verifica si es un valor válido.
- Es más fácil y más limpio usar un enum en sentencias switch o comparaciones, ya que puedes referirte a los objetos por su nombre de enum en lugar de un número o cadena de texto.

Desventajas y Limitaciones

- A medida que tu juego crece y se agregan nuevos tipos de objetos, el enum puede volverse grande y difícil de manejar. Cada vez que añadas un nuevo tipo de objeto, tendrás que actualizar el enum y parte del código que depende de él.
- Los enum son estáticos y no pueden modificarse en tiempo de ejecución. Si tu juego requiere una gran variedad de objetos o objetos generados dinámicamente, un enum puede no ser la mejor solución.
- Si cada objeto tiene su propio conjunto de propiedades y comportamientos únicos, un enum no es adecuado para representarlos. En este caso, es mejor utilizar clases o estructuras.

Introducción a C# en Unity: Linq

Linq en C#

LINQ (Language Integrated Query) es una poderosa herramienta en C# que permite realizar consultas complejas y manipular colecciones de datos de una manera expresiva y declarativa.

- **Where:** Filtra una colección basada en una condición especificada.
 - Uso común: Encontrar elementos que cumplan con ciertas condiciones, como enemigos con salud baja o ítems de un tipo específico.
- **Select:** Transforma cada elemento de una colección.
 - Uso común: Cambiar la forma de los elementos, como obtener solo una propiedad específica de cada objeto.
- **OrderBy** y **OrderByDescending:** Ordena los elementos de una colección en orden ascendente (OrderBy) o descendente (OrderByDescending).
 - Uso común: Ordenar elementos, como enemigos por su distancia al jugador o ítems por su valor.
- **GroupBy:** Agrupa elementos que comparten una clave común.
 - Uso común: Agrupar elementos, como ítems por categoría o enemigos por tipo.
- **First** y **FirstOrDefault**, **Last** y **LastOrDefault:** Obtiene el primer o último elemento de una colección. FirstOrDefault y LastOrDefault devuelve un valor por defecto si la colección está vacía.
 - Uso común: Obtener un elemento específico, como el primer enemigo en una lista o el primer ítem de un tipo específico.

Linq en C#

- **Take:** selecciona un número específico de elementos desde el inicio de una colección.
 - Uso común: limitar el número de resultados obtenidos de una secuencia.
- **Any:** Verifica si alguna de las condiciones se cumple en los elementos de la colección.
 - Uso común: Comprobar si existen elementos que cumplen con una condición, como verificar si hay enemigos en un área.
- **All:** Verifica si todos los elementos cumplen una condición específica.
 - Uso común: Verificar si todos los elementos de una colección cumplen con una condición, como si todos los enemigos están inactivos.
- **Count:** Cuenta el número de elementos en una colección, opcionalmente que cumplen una condición específica.
 - Uso común: Contar elementos, como el número de enemigos o ítems disponibles.
- **Sum, Min, Max, Average:** Realizan cálculos agregados sobre una colección.
 - Uso común: Obtener sumas, mínimos, máximos, o promedios, como la salud total de todos los enemigos, el ítem con el valor más alto, etc.

Estos métodos pueden combinarse y encadenarse para crear consultas complejas y expresivas. El conocimiento de estos métodos y su aplicación práctica pueden mejorar significativamente la eficiencia y claridad del código en proyectos de Unity.

Linq en C#

Ejemplo 1: Filtrar enemigos por salud, supongamos que tienes una lista de enemigos y quieres obtener aquellos cuya salud está por debajo de un cierto umbral. Esto es muy común en juegos donde necesitas identificar enemigos débiles o casi derrotados.

```
public class ControladorEnemigos : MonoBehaviour
{
    List<Enemigo> enemigos;
    void Start()
    {
        enemigos = new List<Enemigo> {
            new Enemigo("Ogro", 100),
            new Enemigo("Esqueleto", 30),
            new Enemigo("Goblin", 50)};

        var enemigosDebiles = enemigos.Where(enemigo => enemigo.Salud < 60);

        var enemigosDebiles2 = enemigos
            .Where(enemigo => enemigo.Salud < 60);
            .OrderBy(enemigo => enemigo.Salud);

        foreach (var enemigo in enemigosDebiles2) Debug.Log(enemigo.Nombre);
    }
}
```

Linq en C#

Ejemplo 2: Ordenar objetos por distancia, supongamos que quieres ordenar una lista de objetos en el juego por su distancia a un punto específico, como podría ser la posición del jugador.

```
public Transform jugador;
List<ObjetoJuego> objetos;

void OrdenarObjetosPorDistancia()
{
    var objetosOrdenados = objetos
        .OrderBy(obj => Vector3.Distance(obj.transform.position, jugador.position));

    var gemasOrdenadas = objetos
        .Where(obj => obj.tipo == Tipo.Gema)
        .OrderBy(objeto => Vector3.Distance(objeto.transform.position, jugador.position));

    foreach (var objeto in objetosOrdenados)
    {
        Debug.Log($"Objeto ordenado por distancia: {objeto.name}");
    }
}
```

Linq en C#

Ejemplo 3: Agrupar elementos por tipos.

```
public enum TipoItem { Arma, Poción, Armadura }
public class Item
{
    public string nombre;
    public TipoItem tipo;
    public Item(string nombre, TipoItem tipo) { ... }
}
public class Inventario : MonoBehaviour
{
    List<Item> items;
    void AgruparItemsPorTipo()
    {
        var grupos = items.GroupBy(item => item.tipo);
        foreach (var grupo in grupos)
        {
            Debug.Log($"Items del tipo: {grupo.Key}");
            foreach (var item in grupo) Debug.Log(item.nombre);
        }
    }
}
```

Linq en C#

Ejemplo 4: Calcular la suma total de puntos de vida que le quedan a todos los enemigos.

```
public class Enemigo
{
    public string nombre;
    public int puntos;
    public Enemigo(string nombre, int puntos)    { ... }
}
public class ControladorEnemigos : MonoBehaviour
{
    List<Enemigo> enemigos;

    void Start()
    {
        enemigos = new List<Enemigo> {
            new Enemigo("Ogro", 100),
            new Enemigo("Esqueleto", 50),
            new Enemigo("Goblin", 75)};

        int totalPuntos = enemigos.Sum(enemigo => enemigo.puntos);
        Debug.Log($"Total de puntos de los enemigos: {totalPuntos}");
    }
}
```

Linq en C#

Ejemplo 5: Listar armas de tipo espada, de enemigos derrotados y ordenadas de mayor a menor Fuerza.

```
var armasPoderosas = enemigos
    .Where(enemigo => enemigo.derrotado)           // filtra enemigos derrotados
    .SelectMany(enemigo => enemigo.armas)          // selecciona todas sus armas
    .Where(arma => arma.tipo == "Espada")          // filtra por tipo de arma
    .OrderByDescending(arma => arma.Fuerza)        // ordena por Fuerza, mayor a menor
    .ToList();

foreach (var arma in armasPoderosas)
{
    Debug.Log($"Espada poderosa: {arma.Nombre}, Fuerza: {arma.Fuerza}");
}
```

```
var armasPoderosas = enemigos
    .Where(enemigo => enemigo.derrotado)           // filtra enemigos derrotados
    .SelectMany(enemigo => enemigo.armas)          // selecciona todas sus armas
    .Where(arma => arma.tipo == "Espada")          // filtra por tipo de arma
    .OrderByDescending(arma => arma.Fuerza)        // ordena por Fuerza, mayor a menor
    .ToList()
    .ForEach(arma => Debug.Log($"Espada poderosa: {arma.Nombre}"));
```

Linq en C#

Ejemplo 6: Listar los objetos del inventario por valor.

```
inventario.OrderByDescending(objeto => objeto.CosteOro)
    .ThenByDescending(objeto => objeto.CostePlata)
    .ThenByDescending(objeto => objeto.CosteBronce)
    .ToList()
    .ForEach(objeto => Debug.Log(objeto));
```

Ejemplo 7: Listar los objetos de una rareza concreta por valor.

```
inventario.Where(objeto => objeto.Rareza == rareza)
    .OrderByDescending(objeto => objeto.CosteOro)
    .ThenByDescending(objeto => objeto.CostePlata)
    .ThenByDescending(objeto => objeto.CosteBronce)
    .ToList()
    .ForEach(objeto => Debug.Log(objeto));
```


Linq en C#

Ejemplo 8: Listar las 5 armas del inventario de tipo espada con más durabilidad.

```
inventario.OfType<Arma>()  
    .Where(arma => arma.CategoriaArma == CategoriaArma.Espada)  
    .OrderByDescending(arma => arma.Durabilidad)  
    .Take(5)  
    .ToList()  
    .ForEach(objeto => Debug.Log(objeto));
```

Ejemplo 9: Listar los 3 materiales básicos más abundantes en el inventario.

```
inventario.OfType<MaterialBasico>()  
    .OrderByDescending(material => material.Cantidad)  
    .Take(3)  
    .ToList()  
    .ForEach(objeto => Debug.Log(objeto));
```

Introducción a C# en Unity:

Persistencia de datos

Persistencia de datos en C# con JSON

Para almacenar información de un juego en Unity en el dispositivo local utilizando archivos, hay varias opciones, pero una de las más comunes y eficientes es el uso de archivos JSON (JavaScript Object Notation). JSON es un formato ligero de intercambio de datos, fácil de leer y escribir. Primero, debes definir una clase que represente los datos que deseas guardar. Por ejemplo, si quieres almacenar el progreso del jugador, podrías tener una clase **ProgresoJugador**.

```
[Serializable]
public class ProgresoJugador
{
    public int nivel;
    public float tiempoJugado;
    public int puntos;
}
```



```
ProgresoJugador progreso = new ProgresoJugador();
progreso.nivel = 5;
progreso.tiempoJugado = 120.5f;
progreso.puntos = 1000;

string datosJson = JsonUtility.ToJson(progreso);
```

```
string ruta = Path.Combine(Application.persistentDataPath, "progresoJugador.json");
File.WriteAllText(ruta, datosJson);

if(File.Exists(ruta))
{
    string datosGuardados = File.ReadAllText(ruta);
    ProgresoJugador progresoCargado = JsonUtility.FromJson<ProgresoJugador>(datosGuardados);
}
```

Persistencia de datos en C#

Application.dataPath: propiedad que devuelve la ruta al directorio de "Assets" de tu proyecto. Es donde se almacenan tus assets, scripts, escenas, y otros archivos relacionados con el juego durante el desarrollo.

Uso Común: Se usa principalmente para acceder a archivos que son parte de tu proyecto de Unity durante el desarrollo, como leer archivos de configuración o datos que son parte integrante de tu juego. Es importante notar que en un juego compilado, esta ruta puede no comportarse como se espera, especialmente en ciertas plataformas. Ejemplo de Ruta en Windows: `C:\Users\Usuario\Documents\MyGame\Assets`.

En un juego compilado, la ruta será diferente y dependerá del sistema operativo y la plataforma.

Application.persistentDataPath: propiedad que proporciona una ruta de archivo donde puedes guardar y leer datos que se generan durante la ejecución de tu juego. Es una ubicación de almacenamiento de datos que está garantizada para ser accesible en todas las plataformas que Unity soporta.

Uso Común: Se utiliza para guardar datos que deben persistir entre sesiones de juego, como archivos de guardado, configuraciones del usuario, o datos de progreso. Es una ubicación segura y adecuada para escribir archivos durante la ejecución de un juego.

Ejemplo de Ruta en Windows: `C:\Users\Usuario\AppData\LocalLow\CompanyName\GameName`.

Las rutas específicas varían según el sistema operativo y la plataforma, pero siempre serán lugares donde es seguro escribir y leer durante la ejecución del juego.

Persistencia de datos en C# con serialización binaria

La serialización binaria es una forma eficaz de almacenar objetos de forma compacta y con un nivel más alto de seguridad en comparación con la serialización en texto plano como JSON.

```
public void GuardarProgreso(ProgresoJugador progreso)
{
    BinaryFormatter formatter = new BinaryFormatter();
    string ruta = Path.Combine(Application.persistentDataPath, "progresoJugador.dat");
    FileStream file = File.Create(ruta);
    formatter.Serialize(file, progreso);
    file.Close();
}
```

```
public ProgresoJugador CargarProgreso()
{
    string ruta = Path.Combine(Application.persistentDataPath, "progresoJugador.dat");
    if (File.Exists(ruta))
    {
        BinaryFormatter formatter = new BinaryFormatter();
        FileStream file = File.Open(ruta, FileMode.Open);
        ProgresoJugador progreso = (ProgresoJugador) formatter.Deserialize(file);
        file.Close();
        return progreso;
    } else return null;
}
```

Persistencia de datos en C#

Consideraciones:

- **Seguridad:** La serialización binaria es más segura contra la manipulación manual de archivos, pero aún puede ser vulnerable a determinados tipos de ataques. La encriptación de los datos antes de la serialización puede añadir una capa adicional de seguridad.
- **Compatibilidad:** la serialización binaria puede ser específica de la plataforma y la versión de .NET utilizada. Esto significa que los archivos serializados en una plataforma/version podrían no ser compatibles con otras.
- **Manejo de Versiones:** Si actualizas la clase ProgresoJugador, debes manejar la compatibilidad con versiones anteriores de tus archivos serializados.

Mejorar la seguridad de los datos:

- Una de las formas más efectivas de proteger los datos es encriptarlos antes de guardarlos en el archivo y luego desencriptarlos al cargarlos. Esto asegura que cualquier intento de manipulación directa en el archivo sea extremadamente difícil.
- Para datos críticos como puntuaciones altas o niveles desbloqueados, puedes usar PlayerPrefs, hablaremos de ello más adelante.
- Para la máxima seguridad, podemos almacenar datos críticos en un servidor. Esto impide que los usuarios accedan y modifiquen los datos localmente, sin embargo, se necesita conexión a Internet.
- Soluciones de terceros: Firebase, PlayFab, AWS, Microsoft Azure, Back4App, etc.

Ejercicios persistencia de datos en C#

Guardar las puntuaciones en el juego de puntería desarrollado anteriormente. Al terminar la partida se debe almacenar la puntuación final en un json y mostrar las distintas puntuaciones almacenadas de mayor a menor puntuación. Crear una solución adicional donde podamos incluir la cantidad de puntuaciones máximas a mostrar.

```
[Serializable] public class ListaPuntuacionesJuegoPunteria
{
    public List<int> listaPuntuaciones;
    public ListaPuntuacionesJuegoPunteria() { listaPuntuaciones = new List<int>(); }
}
```

```
public class PuntuacionesJuegoPunteria
{
    private static PuntuacionesJuegoPunteria instancia = null;
    private ListaPuntuacionesJuegoPunteria puntuaciones;
    private string rutaFicheroPuntuaciones;
    public List<int> ObtenerMejoresPuntuaciones() { }
    public List<int> ObtenerMejoresPuntuaciones(int cantidad) { }
    public void AgregarPuntuacion(int puntuacion) { }
    public void CargarPuntuaciones() { }
    private void GuardarPuntuaciones() { }
}
```

Introducción a C# en Unity: Audio

Audio en Unity

El sistema de audio en Unity es permite incorporar sonido de manera sencilla, para ello se utilizan los siguientes componentes clave:

- **Audio Clip:** Este es el recurso de sonido en sí, como una pista de música o un efecto de sonido. Puede ser importado en varios formatos como MP3, WAV, etc.
- **Audio Source:** Este componente se agrega a un GameObject y es usado para reproducir sonidos. Puedes asignar uno o varios Audio Clips.
- **Audio Listener:** Es el "oído" en la escena. Por lo general, se coloca en la cámara principal o en el personaje del jugador. Una escena debe tener un único Audio Listener para captar el sonido correctamente.

Los archivos de audio de alta calidad pueden ser grandes. Debemos priorizar formatos comprimidos pero de alta calidad sobre WAV para efectos de sonido cortos, esto nos ayuda a mantener el tamaño del juego optimizado.

Cada AudioSource puede reproducir un clip de audio a la vez. Sin embargo, también puedes usar un solo AudioSource y el método PlayOneShot para reproducir varios efectos de sonido superpuestos, lo que puede ser más eficiente y manejable en ciertas situaciones."

Audio en Unity

```
public class MusicaFondo : MonoBehaviour
```

```
{
```

```
    public AudioClip musicaDeFondo;
```

```
    private AudioSource fuenteAudio;
```

```
    void Start()
```

```
    {
```

```
        fuenteAudio = gameObject.AddComponent<AudioSource>();
```

```
        fuenteAudio.clip = musicaDeFondo;
```

```
        fuenteAudio.loop = true; // repetir continuamente
```

```
        fuenteAudio.Play();
```

```
    }
```

```
}
```

```
public class EfectoSonido : MonoBehaviour
```

```
{
```

```
    public AudioClip sonidoExplosion;
```

```
    private AudioSource fuenteAudio;
```

```
    void Start()
```

```
    {
```

```
        fuenteAudio = gameObject.AddComponent<AudioSource>();
```

```
    }
```

```
    void ReproducirSonidoExplosion() { fuenteAudio.PlayOneShot(sonidoExplosion); }
```

```
}
```

Introducción a C# en Unity:

Atributos en el inspector de Unity

Atributos en el inspector de Unity

En Unity, para mejorar la organización y legibilidad de los datos de un script en el Inspector, podemos utilizar varios atributos proporcionados por el sistema de serialización de Unity. Estos atributos te permiten dividir los datos en secciones, agregar títulos, subtítulos, tooltips y etiquetas.

[Header]: Permite agregar un título sobre un grupo de variables.

[Space]: Añade espacio vertical en el inspector para separar secciones.

[Tooltip]: Muestra un texto de ayuda cuando el cursor se sitúa sobre el elemento en el Inspector.

[SerializeField]: Fuerza a Unity a serializar un campo privado y mostrarlo en el Inspector.

[HideInInspector]: Oculta un campo público en el Inspector.

[Range]: Limita un valor numérico a un rango específico y proporciona un deslizador en el Inspector.

[TextArea]: permite convertir una variable de tipo string en un área de texto en el Inspector.

Atributos en el inspector de Unity

En Unity, para mejorar la organización y legibilidad de los datos de un script en el Inspector, podemos utilizar varios atributos proporcionados por el sistema de serialización de Unity. Estos atributos te permiten dividir los datos en secciones, agregar títulos, subtítulos, tooltips y etiquetas.

[RequireComponent]: se utiliza para indicar que un componente es necesario para el funcionamiento correcto de otro. Unity asegurará que el componente requerido esté presente, y si no, lo agrega automáticamente.

[Min]: permite especificar un valor mínimo para una propiedad pública.

[ContextMenu]: agrega una función del script al menú contextual del componente en el Inspector. Permite ejecutar métodos directamente desde el Inspector.

[ContextMenuItem]: similar a [ContextMenu], pero se aplica específicamente a una propiedad, permitiendo agregar opciones al menú contextual de dicha propiedad.

[SelectionBase]: se utiliza en un GameObject para asegurar que, cuando se seleccionan sus hijos en la escena, el objeto con este atributo será seleccionado en su lugar. Es útil para trabajar con objetos complejos compuestos por múltiples subobjetos.

Introducción a C# en Unity: Optimización y buenas prácticas

Optimización en C#: stack y heap

- En Unity y C#, la comprensión del uso del stack y el heap es crucial para la optimización y la escritura de código eficiente.
 - **Stack(Pila):** Almacena variables primitivas y referencias a objetos. Es rápido y maneja la memoria automáticamente, liberándola cuando la variable sale de alcance.
 - **Heap(Montón):** Almacena objetos y estructuras grandes. La asignación y desasignación de memoria es más lenta y está sujeta a recolección de basura.
- Reduce la creación de nuevos objetos especialmente en Update() o FixedUpdate(), esto permite evitar presión sobre la recolección de basura.
- Estructuras vs Clases: priorizar struct (almacenada en stack) sobre class (almacenada en heap) si el objeto es pequeño y de corta duración. Usa class para objetos más complejos o que requieran herencia y polimorfismo.
- Reutilización de Objetos: implementa patrones de diseño como Object Pooling para reutilizar objetos en lugar de crear y destruir constantemente.
- Pasar por Referencia vs Valor: al pasar grandes estructuras a funciones, considera pasarlas por referencia (usando ref o out) para evitar copias innecesarias.

Optimización en C#: stack y heap

En Unity y C#, la comprensión del uso del stack y el heap es crucial para la optimización y la escritura de código eficiente. Ejemplo práctico:

```
struct Punto2D
{
    public float x, y;
    public Punto2D(float x, float y)
    {
        this.x = x;
        this.y = y;
    }
}

void ActualizarPunto(ref Punto2D punto)
{
    punto.x += 1.0f;
    punto.y += 1.0f;
}

void UsoEficiente()
{
    Punto2D punto = new Punto2D(0, 0);
    ActualizarPunto(ref punto); // pasamos por referencia para evitar copia
}
```


Optimización en C#: strings

Las strings en C# son inmutables, lo que significa que cada vez que modificas una string, en realidad estás creando una nueva en la memoria. Esto puede llevar a una cantidad significativa de asignaciones de memoria en operaciones como concatenaciones repetidas, lo cual es especialmente problemático en un bucle o en métodos que se llaman frecuentemente como Update() en Unity.

```
string registro = "";
for (int i = 0; i < 100; i++)
    registro += $"Jugador {i} ha obtenido un nuevo récord\n";
//En este ejemplo, cada concatenación en el bucle crea una nueva string, lo que
resulta en 100 asignaciones de memoria distintas
```

```
StringBuilder registro = new StringBuilder();
for (int i = 0; i < 100; i++)
    registro.Append("Jugador ").Append(i).Append(" ha obtenido un nuevo récord\n");

string resultadoFinal = registro.ToString();
```

Optimización en C#: strings

Consideraciones Adicionales:

- Puedes inicializar un StringBuilder con una capacidad estimada si sabes aproximadamente cuán grande será la string final. Esto puede reducir aún más la necesidad de reasignaciones de memoria.

```
StringBuilder registro = new StringBuilder(1000);  
// inicializa con una capacidad de 1000 caracteres
```

- Si estás modificando una string constantemente (por ejemplo, actualizando un contador en la UI), StringBuilder es una opción mucho más eficiente que la concatenación de strings.
- Mientras que StringBuilder es eficiente para construir strings, no es la mejor opción para operaciones de lectura frecuente o para manipular pequeñas strings de manera ocasional.

Optimización en C#: excepciones

En el desarrollo de juegos con Unity, el uso excesivo de excepciones puede llevar a una disminución del rendimiento y a problemas de estabilidad. Limita su uso a situaciones donde realmente sean necesarias.

```
void Update()
{
    // evita tener un try-catch aquí si es posible
}
```

Captura las excepciones que puedes manejar de manera significativa. Capturar **Exception** de manera genérica es generalmente una mala práctica, ya que puede ocultar errores no anticipados y dificultar la depuración.

Si tu método no puede manejar la excepción de manera adecuada, es preferible dejar que la excepción se propague. Puedes capturarla en un nivel más alto donde su manejo sea más apropiado.

```
void MetodoDeBajoNivel()
{
    try
    {
        // código que puede fallar
    }
    catch (IOException ex)
    {
        throw; // propaga la excepción
    }
}
```

Optimización en C#: colecciones

Las colecciones en C# y su uso en Unity son fundamentales para la gestión de datos. Sin embargo, un manejo ineficiente puede llevar a problemas de rendimiento

- Listas vs Arrays: Debemos utilizar arrays cuando el tamaño es fijo y listas cuando necesites una colección dinámica.
- Dictionary y HashSet: Son muy eficientes para búsquedas.
- En lugar de crear y destruir objetos, considera usar un sistema de pool de objetos, especialmente para entidades que se crean y destruyen frecuentemente (como proyectiles, enemigos, efectos).
- Evitar el Boxing y Unboxing Innecesario: el boxing (convertir un tipo de valor a object) y unboxing (la operación inversa) son costosos. Evita colecciones no genéricas que requieren estas operaciones.
- Prevenir Reasignaciones Innecesarias: para List<T>, Dictionary<TKey, TValue>, y otras colecciones dinámicas, inicialízalas con una capacidad adecuada si conoces el volumen de datos aproximado para evitar reasignaciones frecuentes.

```
List<MiClase> miLista = new List<MiClase>(capacidadInicial);
```

Introducción a C# en Unity:

Proyecto 1 - Tetris 3D

Proyecto Tema 1 - Tetris 3D

Para asimilar los contenidos de la unidad se debe crear un proyecto final simulando el mítico juego del tetris pero en 3D, los requisitos del juego son los siguientes:

- Cada pieza debe estar formada por 4 cubos individuales de 1x1x1. Cada uno de los 4 cubos tendrá una profundidad distinta en z. Teniendo en cuenta que no se puede quedar suelto ningún cubo, es decir, cada cubo deberá tener contacto con al menos otro cubo de la pieza.
- Se deben crear un mínimo de 5 piezas diferentes.
- Programar el juego para jugar desde el teclado. Las piezas se podrán mover dentro del tablero hacia la izquierda, derecha y abajo. Si pulsamos espacio, la pieza deberá ir rotando. Si pulsamos arriba, la pieza bajará todo lo que pueda y se quedará fijada.
- El tablero se debe generar de forma dinámica al inicio. Ancho entre 4 y 20, alto entre 10 y 22.
- Las piezas deben bajar de forma automática, y deberán incrementar su velocidad a medida que completamos líneas.
- El juego deberá tener un marcador, se deberá configurar la puntuación de modo que se incremente la puntuación por cada línea completada. Además, la puntuación obtenida por cada línea dependerá de la velocidad, a más velocidad, más puntos por línea completada.
- Añadir una melodía y 3 sonidos distintos: al eliminar línea, rotar/mover y final de juego.
- Crea 3 ampliaciones únicas que hagan a tu Tetris 3D especial.
- **No se pueden utilizar ni Prefabs, ni componentes relacionados con físicas y colisiones.**

POSIBLE AMPLIACIÓN: crear una escena de inicio donde se debe incluir el título del juego, un botón para acceder a las mejores puntuaciones, y distintas opciones de configuración del juego.

Preguntas



Bibliografía:

<https://docs.unity3d.com/>

<https://learn.unity.com/>

Unity in Action" de Joe Hocking

"Learning C# by Developing Games with Unity" de Greg Lukosek



Ilustraciones:

<https://pixabay.com/>

<https://freepik.es/>

<https://lottiefiles.com/>