

Agencia de
Aprendizaje
a lo largo
de la vida

Unity

Introducción al lenguaje C#

Scripts en Unity

Fuentes: <https://www.ackosmic.com> / <https://docs.microsoft.com>

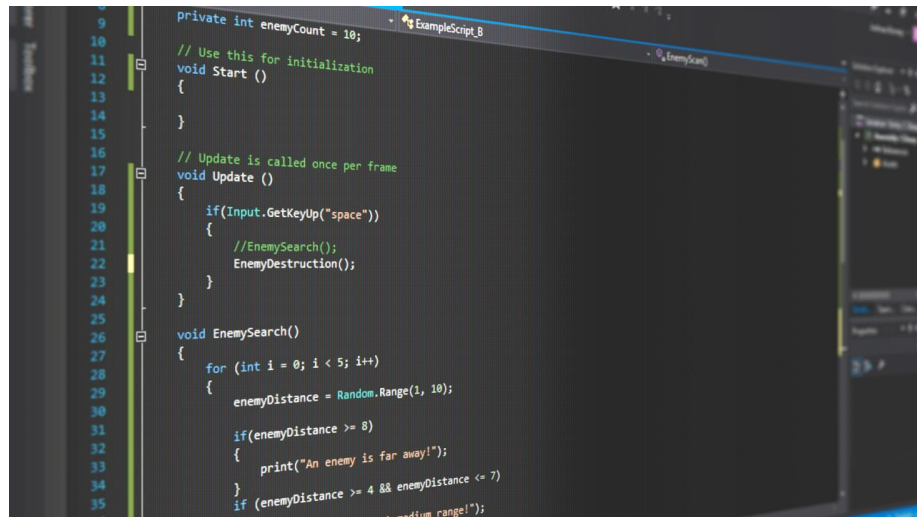
Conociendo nuestro Script

Vamos a conocer las partes que componen nuestro Script. Veremos lo que son:

- namespace
- clase
- objeto
- método

Esto con la finalidad de familiarizarnos con los términos y estructuras a implementar.

Antes de comenzar, debemos crear un **GameObject vacío** en el **panel de jerarquía (Hierarchy)**, al cual le agregaremos como componente los scripts que vayamos desarrollando para poder probar su funcionamiento.



Convención de nombres en los Scripts

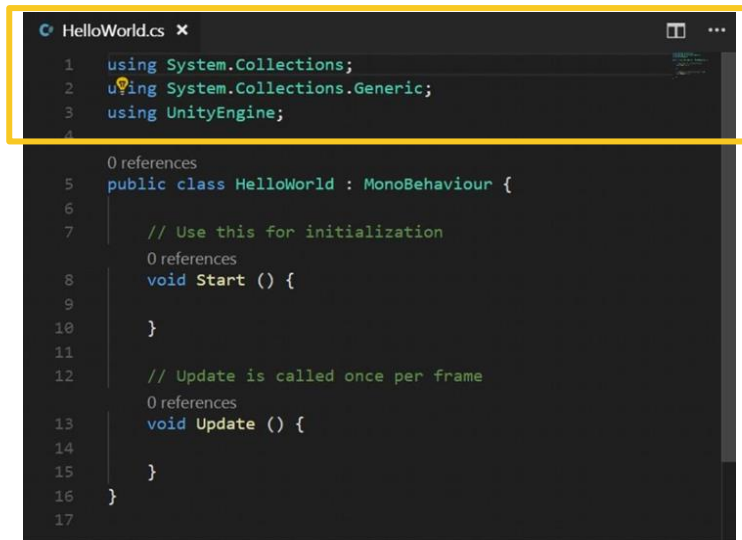
Hay algunas reglas básicas a seguir al nombrar un script:

- Solo usar caracteres alfanuméricos (nada de símbolos o espacios, porque pueden crear conflicto con otras instrucciones).
- No comenzar el nombre con caracteres numéricos (estos son válidos dentro del nombre, pero no como primer carácter).
- Usar mayúsculas y minúsculas (usar la notación tipo “Pascal Case”). Mayúsculas al inicio del nombre del archivo y al inicio de cada palabra (ejemplo: *HelloWorld*, *PrimerCodigo*, *PlayerController*).
- No usar como nombre, funciones específicas del sistema (de las librerías de Unity o de C#).

Debug.Log(); será el método que usaremos para mandar mensajes a la consola de Unity cuando se ejecute nuestro código; normalmente lo usamos cuando queremos saber si una sección específica de nuestro código se está ejecutando (por ejemplo, cuando queremos rastrear errores).

Namespace

Vamos a analizar un script recién creado dentro de Unity. Cuando lo abrimos por primera vez, en el editor de código podemos ver algo así:



```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 0 references
6 public class HelloWorld : MonoBehaviour {
7
8     // Use this for initialization
9     0 references
10    void Start () {
11
12    }
13
14    // Update is called once per frame
15    0 references
16    void Update () {
17
18    }
19 }
```

Esta es la estructura básica con la que un script creado en Unity se genera. Como podemos ver, **por defecto se añaden varias líneas de código.**

Las primeras tres líneas, representan algo conocido como **namespace**; para no complicarnos mucho, veamos a “**namespace**” como librerías de las cuáles podemos obtener recursos (otras secciones de código que están en otros archivos) que facilitan el desarrollo de nuestro código.

En nuestro caso, nosotros no estamos creando un **namespace**, sino que estamos utilizando algunos existentes; por ello aparece “using” antes de “System.Collections”, que significa que vamos a usar el “namespace” existente “System.Collections”.

Clases y Objetos

Hay varias formas de definir lo que es una **Clase** y un **Objeto**, pero en esencia podemos decir que **una clase es una especie de plantilla, esquema o molde, en el cual se indican una serie de atributos y métodos para ejecutar**. Y un **objeto sería el producto resultante de usar una clase** (la clase es la plantilla utilizada para crear objetos).

Cuando **construimos un objeto a partir de una clase, también se dice que estamos creando una instancia de la clase**.

Por ejemplo, una clase es como un molde para hacer figuras, y un objeto sería la figura resultante (las figuras pueden tener colores y texturas distintas, pero sus formas no cambiarían; porque

están definidas por el molde). En este caso, nosotros estamos haciendo figuras a partir de un molde o dicho de otra forma, estamos creando instancias del molde.

De esta manera, cuando nosotros **creamos un script en Unity, lo que estamos haciendo es crear una clase**. Y cuando asignamos ese script a uno o varios objetos de nuestro juego y ejecutamos la escena, se están creando una o varias Instancias de nuestra clase.

En el siguiente ejemplo, en la fila 5 podemos ver:

```
0 references
5 public class HelloWorld : MonoBehaviour {
6
```

Clases y Objetos

Aquí es donde se define el nombre de nuestra clase, este nombre **debe** ser igual al que pusimos cuando creamos el archivo en Unity (de lo contrario no se podrá ejecutar correctamente).



Nota: Si cambiamos el nombre de nuestro archivo, es necesario cambiarlo también dentro en esta sección..

```
0 references
5 public class HelloWorld : MonoBehaviour {
6
```

Con **public class** estamos diciendo que existirá una clase y que ésta puede ser utilizada por otras clases creadas dentro de nuestro proyecto en Unity, y su nombre será lo que pongamos al frente.

Después del nombre aparece: **MonoBehaviour**, esto significa que nuestra clase es parte de una clase base llamada así, y de la cuál se pueden obtener muchos beneficios, como acceso a funciones específicas para nuestros juegos y poder trabajar con la interfaz de edición de Unity.

Finalmente aparecen los “{”, y dentro todas las características y acciones de nuestra clase.

Métodos

Los métodos son un conjunto de instrucciones que se agrupan bajo un nombre. Los métodos pueden ser invocados en diferentes momentos del programa y así se evita tener que repetir las líneas de código que contiene varias veces, logrando una mejor modularización.

En el cuerpo de nuestra clase “HelloWorld” se crearon los métodos “Start” y “Update” (el primero en la fila 8 y el segundo en la fila 13).

```
7 // Use this for initialization
  0 references
8 void Start () {
9
10 }
11
12 // Update is called once per frame
  0 references
13 void Update () {
14
15 }
```

Ambos métodos pertenecen a la Clase Base “MonoBehaviour”, y sus funciones se pueden resumir como:

- **Start:** Las instrucciones que coloquemos dentro del cuerpo de este método (entre los símbolos de llave “{}”), serán ejecutadas “una sola vez” cuando la instancia de nuestra clase se active, al momento de ejecutar nuestro juego.
- **Update:** Las instrucciones que coloquemos dentro del cuerpo de este método, serán ejecutadas de manera continua e indefinida mientras la instancia de nuestra clase esté activa durante la ejecución de nuestro juego (este método se inicia justo después del método Start). Este método es llamado en cada frame.

Nota: “void” significa que el método solo se ejecuta y no devuelve ningún valor.

Modificadores de acceso

Estos modificadores nos sirven para definir la accesibilidad que las variables/constantes/métodos tendrán (en otras palabras, quienes pueden ver las variables que damos de alta, los métodos, etc.).

Veremos los modificadores de acceso **public** y **private**, existen otros pero los veremos más adelante.

```
public float posicionPersonajeEjeX;
```

Nota: Cuando el modificador de acceso **public** es asignado a una variable, y esta se encuentra dentro de una clase perteneciente a “MonoBehaviour”, el valor de dicha variable puede ser modificado desde la interfaz de edición de Unity (desde la ventana “Inspector”).

Modificador tipo public

Al asignarlo a un miembro de una clase (por ejemplo, a un método o a una variable), permite que se tenga acceso a este desde otro script (o desde otra clase) a dicho método/variable.

Esto solo se puede lograr si hay una instancia de la primera clase ejecutándose al momento de solicitar el acceso.

Ejemplo: En un script, crear una variable que contenga la posición sobre el eje “X” de nuestro personaje, y que dicha posición pueda ser leída por otros scripts (por ejemplo, el script de un personaje enemigo).

Modificadores de acceso

Modificador tipo **private**

Al asignarlo a un miembro de una clase (variable, constante o método), permite el acceso a este solo desde dentro de la misma clase.

Ejemplo: Crear una constante que contenga la fuerza de salto de nuestro personaje, dicha constante solo puede ser usada dentro de nuestra clase (no puede ser leída por otras).

```
private const int fuerzaSaltoPersonaje = 15;
```

Nota: Cuando creamos un miembro de nuestra clase y no especificamos un modificador de acceso, este miembro se comportará como si tuviera un modificador del tipo **private**.

Otro aspecto que tomar en cuenta cuando declaramos variables y constantes con modificador tipo **public** o **private** es, hacerlo dentro del cuerpo de nuestra clase, pero no dentro de un método (ya que las variables declaradas dentro de un método no pueden usarse en otras partes de la clase, solo dentro del mismo método).

Tendríamos el mismo comportamiento si creáramos la constante de la siguiente forma:

```
const int fuerzaSaltoPersonaje = 15;
```

Variables y Constantes

Fuentes: <https://www.ackosmic.com> / <https://docs.microsoft.com>

Variables y Constantes

Las **variables y las constantes se pueden definir como secciones de memoria en nuestra computadora reservadas para almacenar un valor.**

- Con las **variables**, el valor almacenado puede ser fijo o cambiante (variable).
- Con las **constantes**, el valor almacenado es fijo, y no cambiante (constante).

Al ser ambas secciones de memoria reservadas, quiere decir que podemos hacer uso de ellas durante la ejecución de nuestro código (mientras se ejecuta nuestra escena).

Nosotros usamos las variables y a las constantes para almacenar información que nos es de utilidad al momento de crear nuestros juegos (por ejemplo, la puntuación del jugador, si el juego ya terminó, el nombre de un personaje, etc.).

Variables y Constantes

Las variables y las constantes necesitan tener un nombre único para poder ser usadas (de lo contrario no sabríamos a cuál hacer referencia), y como lo vimos con los scripts, los nombres de las variables y de las constantes también tienen algunas reglas:

- Solo usar caracteres alfanuméricos (nada de símbolos o espacios, porque pueden crear conflicto con otras instrucciones).
- Usar minúsculas y mayúsculas. Minúsculas al inicio del nombre de la variable o constante y mayúsculas al inicio de la segunda palabra y posteriores (Por ejemplo: *puntuacionJugador* o *playerScore*, *juegoFinalizado* o *gameOver*, *nombreEnemigo* o *enemyName*).
- Tratar de siempre asignar nombres que describan correctamente el uso que le vamos a dar a la variable o constante (aunque puedan parecer extensos, siempre es mejor usar "*puntuacionJugador*" en lugar de solo "*puntuacion*").
- No usar como nombre, funciones específicas del sistema (de las librerías de Unity o de C#).

Variables y Constantes

Hay convenciones establecidas para nombrar las Variables y las constantes:

Pascal Case

Pascal Case es una convención de cómo nombrar diferentes elementos en programación.

Su característica principal es que no incluye espacios, sino que cada palabra queda identificada empezando con una mayúscula. Se utiliza principalmente para clases, propiedades y métodos.

Ej: un script “movimiento del jugador”, pasaría a ser *MovimientoJugador*.

Camel Case

Similar al Pascal Case. Sin embargo, en su uso la primera palabra no empieza con mayúscula. Se utiliza principalmente para variables.

Ej: Una variable “vida Jugador” sería *vidaJugador*, y una variable cuyo nombre es “posicion inicial del enemigo” sería *posicionInicialDelEnemigo*.

Tipos de Datos

Las Variables y las Constantes nos **sirven para almacenar valores**, veremos los tipos más comunes utilizados.

Tipo int

Este tipo de dato lo utilizamos para almacenar valores numéricos enteros de 32 bits (en un rango de -2,147,483,648 a 2,147,483,647).

Ejemplo: Si queremos una **variable** que almacene la cantidad de vidas de nuestro héroe, la declaramos así:

```
int vidasHeroe = 3;
```

Si queremos una **constante** que almacene el precio de una armadura, podemos declararla así:

```
const int precioArmadura = 100;
```

Nota: Como se puede ver en los ejemplos anteriores, para declarar una variable solo es necesario escribir el tipo de dato que va a contener y el nombre de la variable; pero en el caso de las constantes, hay que escribir “const”, antes del tipo de dato y el nombre de la constante.

Tipos de Datos

Tipo float

Este tipo de dato lo utilizamos para almacenar valores numéricos con punto decimal de 32 bits (en un rango del -3.402823×10^{38} al 3.402823×10^{38}).

Ejemplo: Si queremos una **variable** que almacene la velocidad de una nave espacial, podemos declararla así:

```
float velocidadNave = 17.74f;
```

Si queremos una constante que almacene el valor de la aceleración gravitacional de un planeta, podemos declararla así:

```
const float gravedadPlaneta = 4.978f;
```

Nota: Cuando declaramos variables o constantes tipo float; el valor a almacenar debe ser acompañado del sufijo "f" (para poder diferenciarlo de los otros tipos de datos que también usan valores reales).

Tipos de Datos

Tipo string

Este tipo de dato lo utilizamos para almacenar cadenas de caracteres, en otras palabras, **texto** (no hay una definición oficial sobre cuántos caracteres se puede almacenar; pero estamos seguros de que serán suficientes para nuestros propósitos, o hasta que la memoria del ordenador se desborde, lo que suceda primero).

Ejemplo: Si queremos una variable que almacene el nombre de nuestro personaje, la declaramos:

```
string nombrePersonaje = "Ronnie the Fox";
```

Si queremos una constante que almacene un mensaje de bienvenida:

```
const string mensajeBienvenida = "Hola";
```

Nota: Cuando declaramos variables o constantes tipo string; el texto a almacenar debe colocarse entre comillas dobles (" ").

Tipos de Datos

Tipo bool

Este tipo de dato lo utilizamos para almacenar los valores lógicos (booleanos) **true** (verdadero) o **false** (falso); **no soporta otro valor** y no es necesario escribirlos entre comillas dobles.

Ejemplo: Si queremos una variable que almacene el dato de si terminamos el juego:

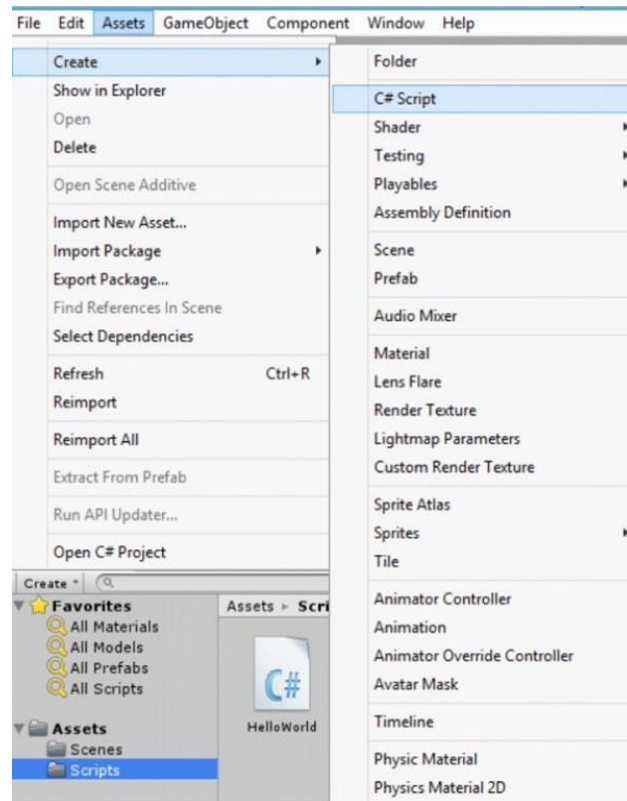
```
bool juegoTerminado = false;
```

Existen muchos otros, pero estos cuatro tipos de datos son los que usaremos con mayor frecuencia cuando comenzamos a desarrollar videojuegos.

Trabajando con Variables

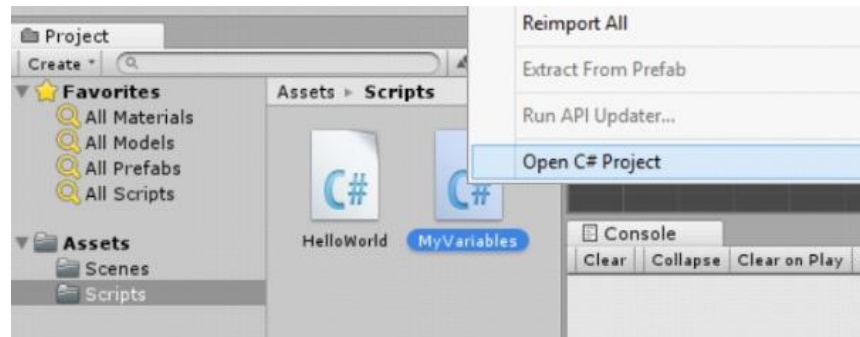
Ahora veremos **cómo implementar variables y constantes dentro de un script.**

Primero, dentro de Unity vamos a crear un nuevo script llamado “MyVariables” (en la ventana “Project”, hacemos clic derecho sobre nuestra carpeta “Scripts” y seleccionamos *Create* → *C# Script*). Otra manera de hacerlo es, seleccionar nuestra carpeta “Scripts” y en la barra de menú ir a *Assets* → *Create* → *C# Script*.



Trabajando con Variables

Una vez creado nuestro script “*MyVariables.cs*”, hacemos doble clic sobre éste para abrir el editor de código. Otra manera, es haciendo clic derecho sobre nuestro script y seleccionar “*Open C# Project*”.



Trabajando con Variables

Ya que tenemos nuestro Script abierto en el editor de código, **vamos a crear nuestra primera variable y nuestra primera constante** para después mostrarlas como mensajes en la consola de Unity.

Dentro de nuestra clase “MyVariables” pero antes de los métodos “Start” y “Update” vamos a declarar la siguiente variable y la siguiente constante:

```
//La cantidad de salud inicial de nuestro personaje  
int saludPersonaje = 100;  
//El deterioro de salud que recibe nuestro personaje  
const int deterioroSalud = 10;
```

Nota: El texto que se encuentra después de las dos diagonales (//) se llama **comentario** y nos sirve para dejar anotaciones dentro de nuestro código, anotaciones que ayuden como referencia para saber con un poco más de detalle lo que estamos haciendo. Los comentarios no son tomados en cuenta al momento de compilar y ejecutar nuestro código; lo que está después de las dos diagonales y sobre la misma fila es aquello a considerar como comentarios.

Trabajando con Variables

Ahora, dentro del método *Start()* vamos a escribir:

```
Debug.Log("Salud Inicial: " + saludPersonaje);  
Debug.Log("Daño que recibe: " + deterioroSalud);
```

El signo de “más” (+) entre el texto a desplegar y la variable o constante nos sirve para indicar que estamos “añadiendo” (concatenando) más información para mostrar en el mensaje.

Debug.Log nos sirve para desplegar mensajes en la consola de Unity, estos mensajes deben estar en formato de texto (entre comillas dobles); en el caso de nuestras variables y constantes, no las ponemos entre comillas dobles porque lo que queremos conocer es el valor que almacenan (de lo contrario en la consola de Unity veríamos solo el nombre de la variable o constante y no su valor).

Trabajando con Variables

Nuestro código debe verse así:

Guardamos nuestro código (Ctrl+S).

Ahora cambiamos a la interfaz de edición de Unity (recuerda esperar un poco mientras Unity toma los cambios hechos en el editor de código).

```
MyVariables.cs x
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  0 references
6  public class MyVariables : MonoBehaviour {
7
8  //La cantidad de salud inicial de nuestro personaje
9  1 reference
10 int saludPersonaje = 100;
11 //El deterioro de salud que recibe nuestro personaje
12 1 reference
13 const int deterioroSalud = 10;
14
15 // Use this for initialization
16 0 references
17 void Start () {
18     Debug.Log("Salud Inicial: " + saludPersonaje);
19     Debug.Log("Daño que recibe: " + deterioroSalud);
20 }
21
22 // Update is called once per frame
23 0 references
24 void Update () {
25
26 }
```

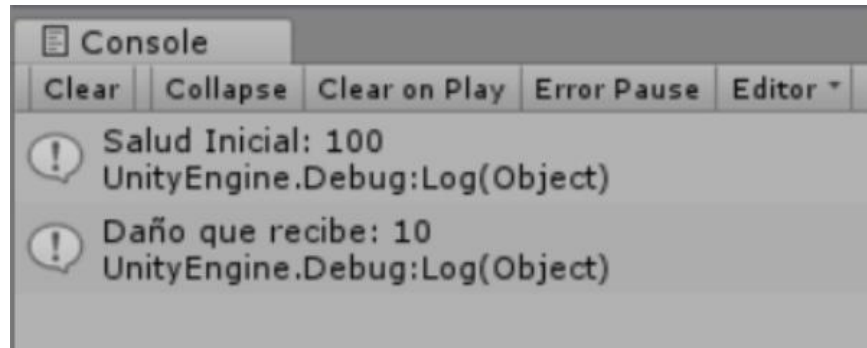
Trabajando con Variables

El siguiente paso es añadir a este mismo objeto del juego nuestro script “*MyVariables*” como su componente (haciendo clic en el botón “*Add Component*”, después en el menú que se despliega hacer clic en “*Scripts*” y por último seleccionar “*My Variables*”).

Ya que agregamos el componente “*My Variables*” (la instancia de nuestra clase “*MyVariables*”), ya podemos hacer clic en el botón “*Play*” de la Interfaz de Edición para ejecutar nuestra escena (mientras el botón “*Play*” este activo, su color cambiará de color “Negro” a color “Azul”, quedándose así mientras se ejecuta la escena).



En la consola de Unity veremos:



Volvemos a dar clic en el botón “*Play*” para desactivar la ejecución de la escena (el botón “*Play*” debe quedar en color negro nuevamente). Vamos de vuelta a nuestro editor de código para añadir más variables y constantes.

Trabajando con Variables

Después de la constante “*deterioroSalud*” escribimos:

```
//La velocidad con la que avanza nuestro personaje  
float velocidadPersonaje = 14.74f;  
/* El valor de la aceleración gravitacional en el planeta  
donde se encuentra nuestro personaje.  
Este valor se declara como una "Constante" */  
const float gravedadPlaneta = 7.44f;
```

Nota: Ahora podemos ver un ejemplo de un “Comentario” que abarca más de una fila; esto se logra escribiendo todo nuestros comentarios dentro de “ /* ” y “ */ ”.

Y dentro del método “Start” añadimos:

```
Debug.Log("Velocidad del Personaje: " + velocidadPersonaje + " km/h");  
Debug.Log("La Aceleración Gravitacional: " + gravedadPlaneta + " m/s^2");
```

Nota: Usamos el signo “+” cuando queremos añadir más información a desplegar en nuestro mensaje.

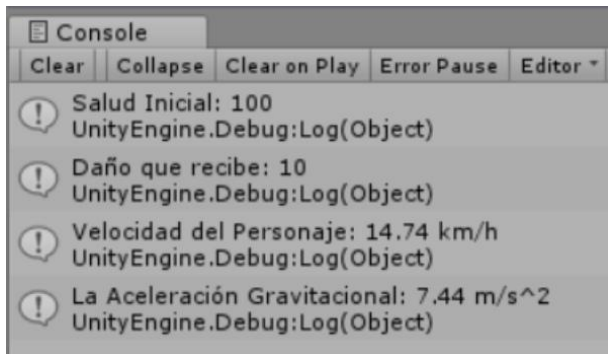
Trabajando con Variables

Nuestro Script queda así:

Guardamos nuestro código (Ctrl+S).

Cambiamos a la interfaz de edición de Unity y ejecutamos la escena.

En la Consola de Unity veremos:



```
MyVariables.cs x
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  0 references
6  public class MyVariables : MonoBehaviour {
7
8      //La cantidad de salud inicial de nuestro personaje
9      1 reference
10     int saludPersonaje = 100;
11     //El deterioro de salud que recibe nuestro personaje
12     1 reference
13     const int deterioroSalud = 10;
14     //La velocidad con la que avanza nuestro personaje
15     1 reference
16     float velocidadPersonaje = 14.74f;
17     /* El valor de la aceleración gravitacional en el planeta
18     donde se encuentra nuestro personaje.
19     Este valor se declara como una "Constante" */
20     1 reference
21     const float gravedadPlaneta = 7.44f;
22
23     // Use this for initialization
24     0 references
25     void Start () {
26         Debug.Log("Salud Inicial: " + saludPersonaje);
27         Debug.Log("Daño que recibe: " + deterioroSalud);
28         Debug.Log("Velocidad del Personaje: " + velocidadPersonaje + " km/h");
29         Debug.Log("La Aceleración Gravitacional: " + gravedadPlaneta + " m/s^2");
30     }
31 }
```

Trabajando con Variables

Volvemos a dar clic en el botón “Play” para desactivar la ejecución de la escena. Y regresamos nuevamente a nuestro editor de código para añadir más variables y constantes.

Después de la constante “*gravedadPlaneta*” escribimos:

```
//Mensaje de Bienvenida al jugador
string mensajeBienvenida = "Bienvenido a este Gran Juego. ";
//Mensaje para Iniciar la Partida
const string mensajeIniciarPartida = "Da Clic para Iniciar";
//Referencia para saber si el juego ya inició
bool partidaIniciada = true;
```

Y dentro del método Start() agregamos:

```
Debug.Log(mensajeBienvenida + mensajeIniciarPartida);
Debug.Log("Partida Iniciada: " + partidaIniciada);
```

Trabajando con Variables

Nuestro Script queda así:

```
MyVariables.cs X
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  0 references
6  public class MyVariables : MonoBehaviour {
7
8  //La cantidad de salud inicial de nuestro personaje
9  1 reference
10 int saludPersonaje = 100;
11 //El deterioro de salud que recibe nuestro personaje
12 1 reference
13 const int deterioroSalud = 10;
14 //La velocidad con la que avanza nuestro personaje
15 1 reference
16 float velocidadPersonaje = 14.74f;
17 /* El valor de la aceleración gravitacional en el planeta
18 donde se encuentra nuestro personaje.
19 Este valor se declara como una "Constante" */
20 1 reference
21 const float gravedadPlaneta = 7.44f;
22 //Mensaje de Bienvenida al jugador
23 1 reference
```

```
18 string mensajeBienvenida = "Bienvenido a este Gran Juego. ";
19 //Mensaje para Iniciar la Partida
20 1 reference
21 const string mensajeIniciarPartida = "Da Clic para Iniciar";
22 //Referencia para saber si el juego ya inició
23 1 reference
24 bool partidaIniciada = true;
25
26 // Use this for initialization
27 0 references
28 void Start () {
29     Debug.Log("Salud Inicial: " + saludPersonaje);
30     Debug.Log("Daño que recibe: " + deterioroSalud);
31     Debug.Log("Velocidad del Personaje: " + velocidadPersonaje);
32     Debug.Log("La Aceleración Gravitacional: " + gravedadPlaneta);
33     Debug.Log(mensajeBienvenida + mensajeIniciarPartida);
34     Debug.Log("Partida Iniciada: " + partidaIniciada);
35 }
36
37 // Update is called once per frame
38 0 references
39 void Update () {
40 }
41 }
```

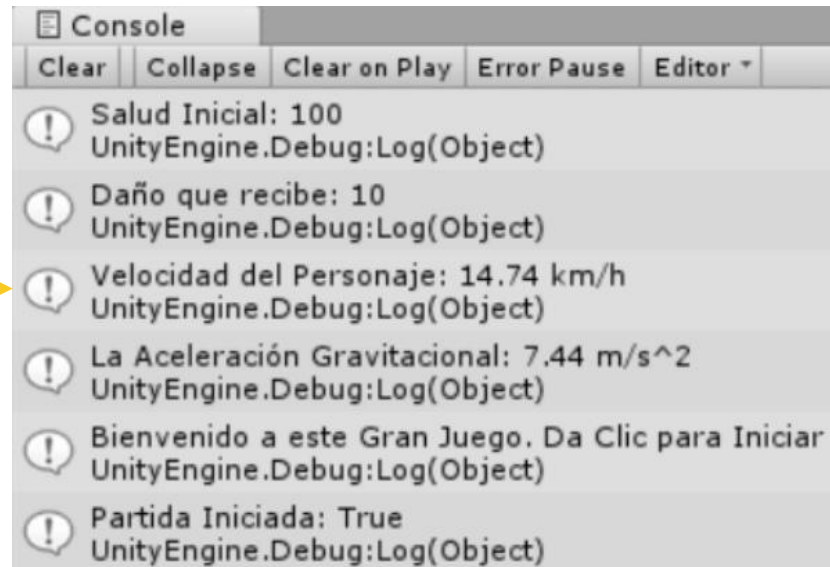
Trabajando con Variables

Cambiamos a la interfaz de edición de Unity (recuerda esperar un poco mientras Unity toma los cambios hechos en el editor de código) y ejecutamos la escena.

En la consola de Unity veremos:

Volvemos a dar clic en el botón “Play” para desactivar la ejecución de la escena.

Antes de finalizar, vamos a guardar nuestro trabajo (vamos a *File* → *Save Scenes* y después *File* → *Save Project*).



Tipos de Variables

Tipo C#	Rango de Valores	Tamaño Precisión	Valor por Defecto
sbyte	-128 a 127	Entero 8 bits con signo	0
byte	0 a 255	Entero 8 bits sin signo	0
short	-32.768 a 32.767	Entero 16 bits con signo	0
ushort	0 a 65.535	Entero 16 bits sin signo	0
int	-2.147.483.648 a 2.147.483.647	Entero 32 bits con signo	0
uint	0 a 4.294.967.295	Entero 32 bits sin signo	0

Tipo C#	Rango de Valores	Tamaño Precisión	Valor por Defecto
long	9.223.372.036.854.775.808 a 9.223.372.036.854.775.807	Entero 64 bits con signo	0
ulong	0 a 18.446.744.073.709.551.615	Entero 64 bits sin signo	0
float	$\pm 1,5 \times 10^{-45}$ a $\pm 3,4 \times 10^{38}$	7 dígitos	0.0f
double	$\pm 5,0 \times 10^{-324}$ a $\pm 1,7 \times 10^{308}$	15-16 dígitos	0.0d
decimal	$\pm 1,0 \times 10^{-28}$ a $\pm 7,9228 \times 10^{28}$	18-19 dígitos	0m
char	U+0000 a U+FFFF	Unicode 16 bits	x0000
bool	Booleano	true, false	false

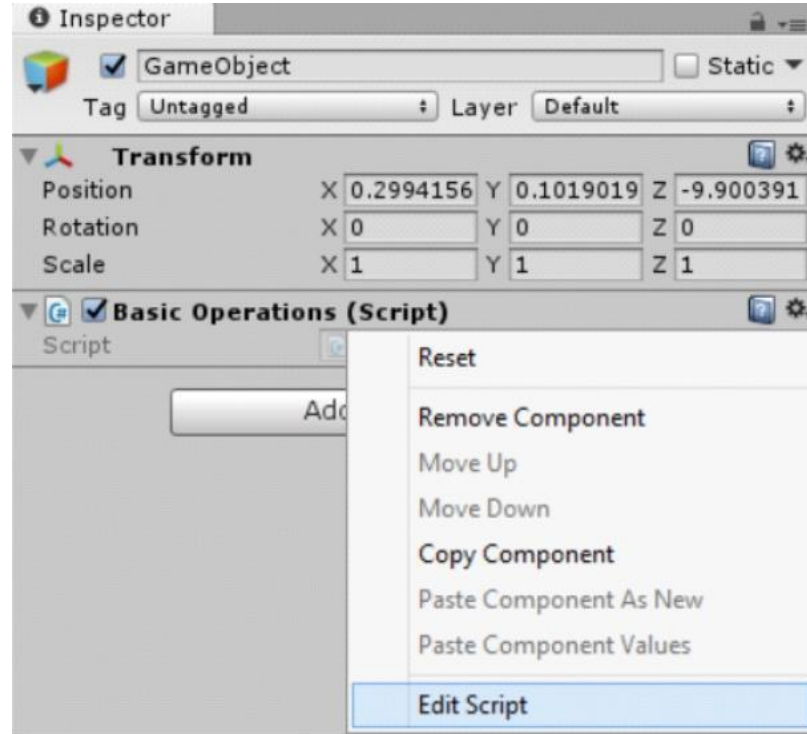
Operadores

Fuentes: <https://www.ackosmic.com> / <https://docs.microsoft.com>

Operadores

Vamos a crear un Script llamado “BasicOperations”, y lo vamos a añadir como un componente del GameObject, luego, vamos a quitar cualquier otro componente que sea instancia de algún script distinto.

Abrimos nuestro script en el editor de código; otra forma de abrir los archivos de código es haciéndolo desde la ventana inspector (cuando existe una instancia de nuestro script añadida como componente), solo hay que hacer clic sobre el pequeño engranaje que se encuentra en la esquina superior derecha del componente y elegir “Edit Script”.



Adiciones y Sustracciones

A continuación, comenzaremos a realizar operaciones matemáticas sencillas usando variables y constantes. El primer paso es tener nuestro script “*BasicOperations*” abierto en el editor de código.

Para realizar las operaciones adición y sustracción, además de las variables, usamos los operadores aritméticos (los símbolos que definen la acción a llevar a cabo):

- Para la adición: “ + ” (símbolo de “más” o de “positivo”)
- Para la sustracción: “ – ” (símbolo de “menos” o de “negativo”)

Nota:

Cuando creamos una **variable**, **podemos asignarle un valor o no**, si fueron creadas sin un valor en específico; al ejecutarse el código, estas variables iniciarán con el valor de cero, y este valor se mantendrá hasta que exista alguna línea de código que las modifique).

En el caso de las **constantes**, **siempre deben crearse con valor asignado**, de lo contrario se generará un error.

Operación de Adición

Vamos a crear las siguientes variables y constantes en la clase:

```
public int primerValor;  
public const int segundoValor = 10;  
int resultadoOperaciones;
```

En este caso, las variables *primerValor* y *resultadoOperaciones* fueron creadas sin un valor en específico, lo mismo que la constante “*segundoValor*”, lo que genera un error:

```
0 references  
7 public int primer  
0 references  
8 public const int segundoValor;  
A const field requires a value to be provided [Assembly-CSharp]  
int BasicOperations.segundoValor
```

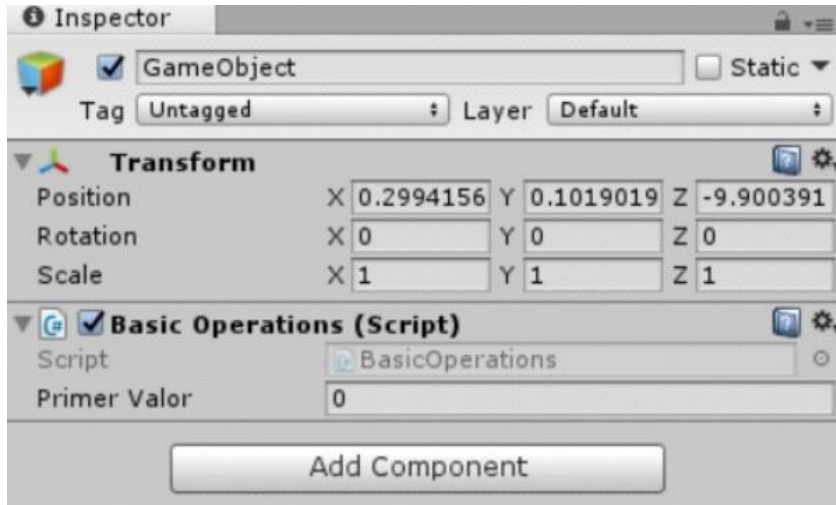
Ahora, vamos a escribir lo siguiente para que nuestro método *Start()* se vea así:

```
void Start(){  
    resultadoOperaciones = primerValor +  
    segundoValor;  
    Debug.Log("Resultado de la Adición = " +  
    resultadoOperaciones);  
}
```

Guardamos nuestro código y cambiamos a la interfaz de edición de Unity.

Operación de Adición

En Unity, si hacemos clic sobre GameObject (dentro de la ventana Hierarchy) y vemos sus componentes (dentro de la ventana Inspector), nos aparecerá lo siguiente:



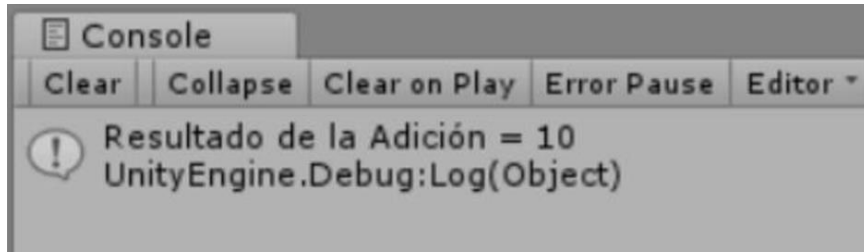
Podemos ver, que en el componente “*Basic Operations*” muestra un campo llamado “*Primer Valor*”. Este campo es la variable “*primerValor*” que creamos dentro de nuestro Script; y se hace visible dentro de la interfaz de edición de unity por tres razones (que siempre se deben cumplir juntas): es una variable, se creó con el modificador de acceso “*Public*” y la clase que contiene a esta variable pertenece a “*MonoBehaviour*”.

Nota: La constante “*segundoValor*” aunque también se dio de alta con el modificador de acceso *public*, no se muestra en la interfaz de edición de Unity por ser una constante (y no una variable).

Operación de Adición

El que podamos ver nuestra variable en la interfaz de edición de Unity es algo muy útil para nosotros; ya que esto significa que podemos modificar su valor desde este entorno sin tener que abrir el editor de código.

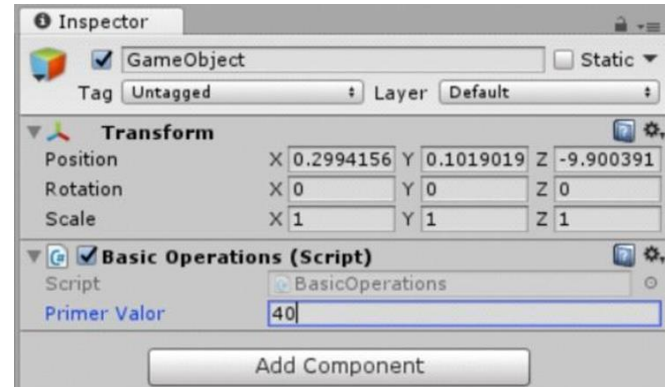
Antes de realizar algún cambio, vamos a ejecutar nuestra escena. La consola nos mostrará:



Se indica un resultado igual a “diez” (que es el valor de nuestra constante “segundoValor”) porque nuestra variable “primerValor” tiene un valor de “cero”.

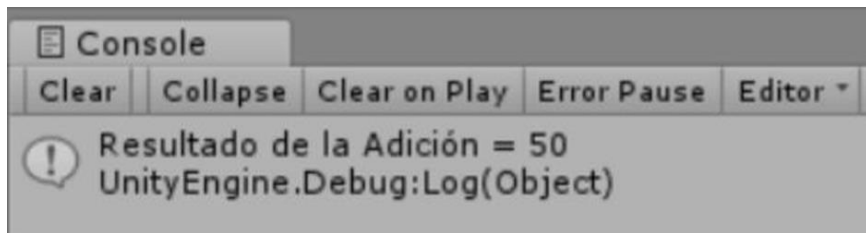
Detenemos nuestra escena.

Ahora sí, vamos a cambiar el valor para nuestra variable “primerValor” (de 0 a 40):



Operación de Adición

Al volver a ejecutar nuestra escena, vemos que la consola nos muestra:



La operación adición se llevó a cabo correctamente ($40 + 10 = 50$).

Nota: Los valores asignados a variables tipo *public* dentro de la interfaz de edición de Unity, sobrescribirán a aquellos que se encuentren dentro del editor de código (aunque la variable sea dada de alta con un valor en específico dentro del editor de código, si le cambiamos dicho valor dentro de la interfaz de edición de Unity, este nuevo valor será el utilizado al ejecutarse el código).

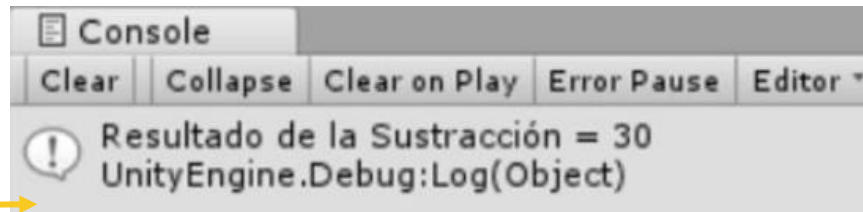
Operación de Sustracción

Después de detener nuestra escena, abrimos el editor de código y hacemos una modificación para que ahora se realice una sustracción (*primerValor - segundoValor*)

```
void Start()
{
    resultadoOperaciones = primerValor - segundoValor;
    Debug.Log("Resultado de la Sustracción = " +
resultadoOperaciones);
}
```

Guardamos nuestro código y cambiamos a la interfaz de edición de Unity.

Ya en Unity, ejecutamos nuevamente nuestra escena (sin realizar otros cambios). La consola de Unity nos mostrará:



Podemos ver que la sustracción se llevó a cabo correctamente ($40 - 10 = 30$).

Operaciones con Datos tipo String

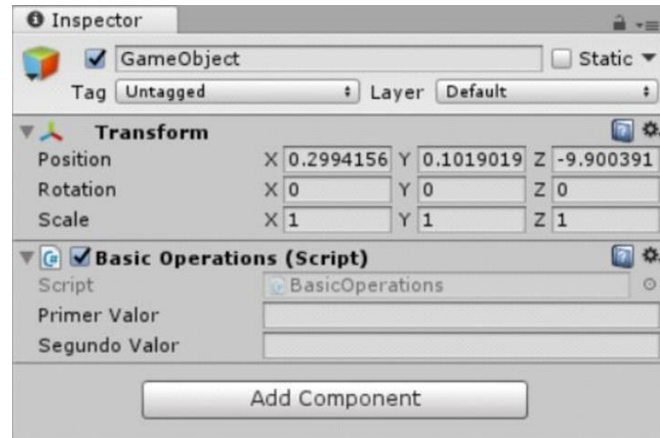
Ahora vamos a modificar un poco a las variables de nuestra clase, creándolas con las siguientes características:

```
public string primerValor;  
public string segundoValor;  
string resultadoOperaciones;
```

Y nuestro método "Start" debe quedar:

```
void Start()  
{  
    resultadoOperaciones = primerValor + segundoValor;  
    Debug.Log("Resultado de la Adición = " +  
resultadoOperaciones);  
}
```

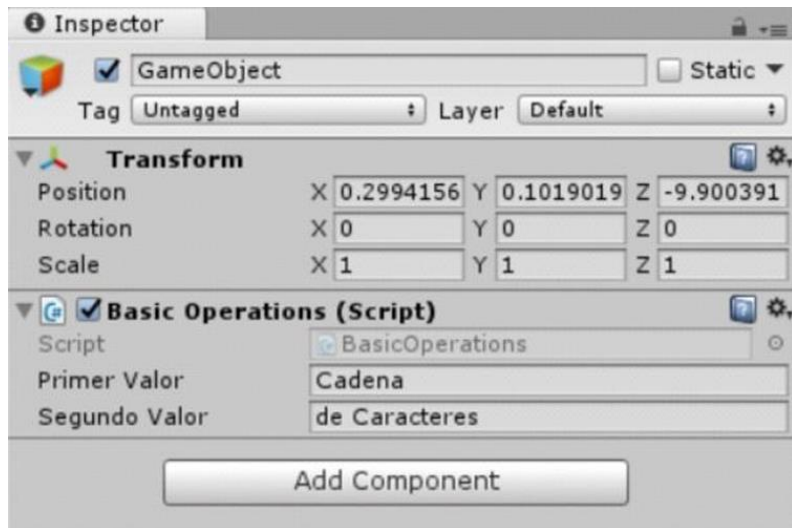
Guardamos nuestro código y vamos a Unity.
Ahora los componentes del GameObject se verán así:



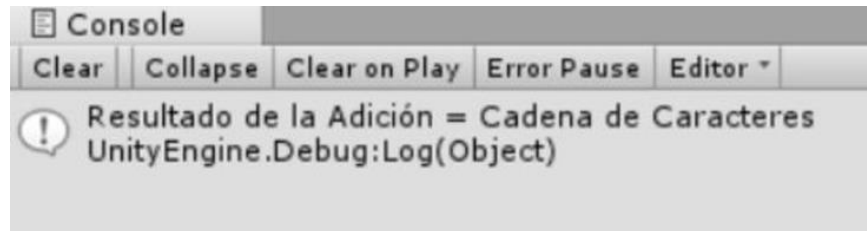
Operaciones con Datos tipo String

En el campo “*Primer Valor*” escribiremos “*Cadena*” (con un espacio al final).

En el campo “*Segundo Valor*” escribiremos “*de Caracteres*”.



Al ejecutar nuestra escena, la consola nos mostrará:



Podemos ver que la adición de variables tipo string se realizó correctamente. Esto provocará una **concatenación** (se “juntan” ambas strings dentro del mismo mensaje).

Nota: En el caso de la sustracción, no se puede realizar con variables tipo string (no como lo hicimos con la adición).

Multiplicaciones y Divisiones

Para realizar las operaciones multiplicación y división, además de las variables, usamos los operadores aritméticos (los símbolos que definen la acción a llevar a cabo):

- Para la Multiplicación: “*” (símbolo “asterisco”)
- Para la División: “/” (símbolo “diagonal” o “barra”)

Para ver cómo se realizan estas operaciones, vamos a modificar nuevamente las variables de nuestra clase:

```
public float primerValor;  
public int segundoValor;  
int resultadoOperaciones;
```

Multiplicación

Lo que queremos hacer es, multiplicar las Variables “*primerValor*” y “*segundoValor*”, pero como se puede ver, estas dos variables están definidas para trabajar con tipos de datos distintos (la primera es tipo “*float*” y la segunda tipo “*int*”). Además, el resultado de esta operación debe ser almacenado en una variable tipo “*int*” (“*resultadoOperaciones*”).

Multiplicación

Si escribimos la operación tal cual, tendríamos el siguiente mensaje de error:

```
0 references
5 public class BasicOperations : MonoBehaviour {
6
7     1 reference
    public float primerValor;
8     1 reference
    public int segundoValor;
9     2 references
    int resultadoOperaciones;
10
11     // Use this for initial
    0 references
    void Start () {
12         int BasicOperations.segundoValor
13         resultadoOperaciones = primerValor * segundoValor;
```

Cannot implicitly convert type 'float' to 'int'. An explicit conversion exists (are you missing a cast?) [Assembly-CSharp]

Este tipo de mensaje se muestra porque **no podemos usar una variable tipo “float” en una operación donde el resultado que se espera es tipo “int”**.

Multiplicación

Para solucionar estas situaciones, es necesario que todos los términos usados en nuestras operaciones sean del mismo tipo. Por ello, dentro de nuestro método “*Start*” escribiremos:

```
void Start()
{
    resultadoOperaciones = (int)primerValor * segundoValor;
    Debug.Log("Resultado de la Multiplicación = " + resultadoOperaciones);
}
```

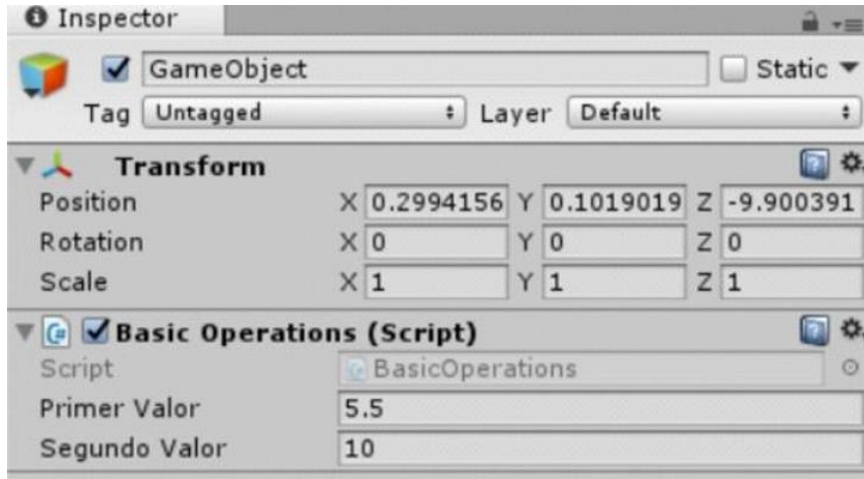
Al escribir “*(int)primerValor*” estamos indicando que, aunque esta variable sea tipo “*float*”, para esta operación se debe considerar como “*int*”, esta **operación se conoce como casting**.

El casting o simplemente **cast** nos permite hacer una conversión explícita de un tipo de dato a otro, a criterio del programador siempre y cuando estos tipos sean compatibles.

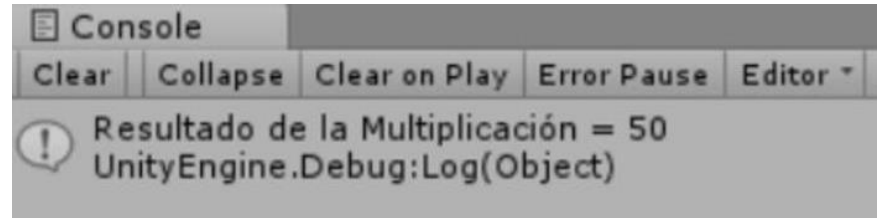
Multiplicación

Guardamos nuestro código, cambiamos a Unity, y en la ventana **Inspector** con el GameObject seleccionado escribimos:

- “5.5” para “*Primer Valor*”
- “10” para “*Segundo Valor*”



Ejecutamos la escena, y la Consola de Unity nos mostrará:

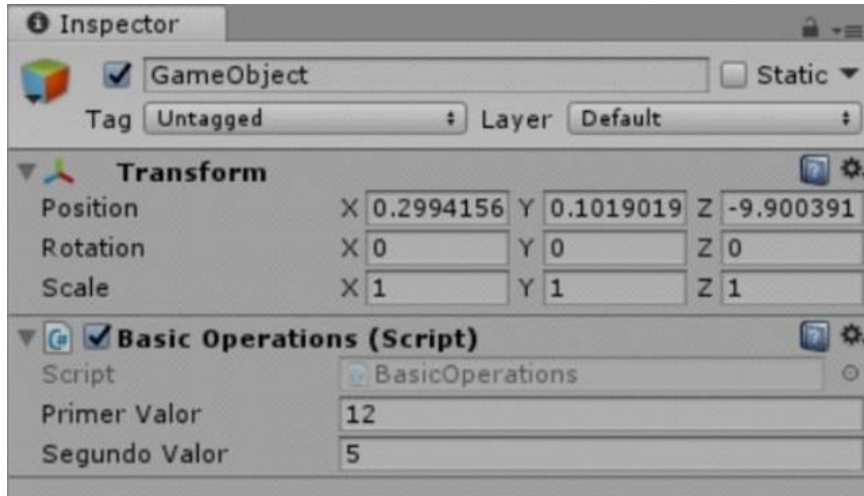


El resultado obtenido es “50”, pero “55” es el valor que esperábamos; esto sucede porque al usar “(int)primerValor”, no se está tomando en cuenta al componente “decimal” del valor de nuestra variable (al ser “5.5”, solo se toma en cuenta al valor entero “5” y se descarta el componente decimal “.5”); ya que es una operación de valores tipo “int”, esta se realiza como “5 * 10 = 50”.

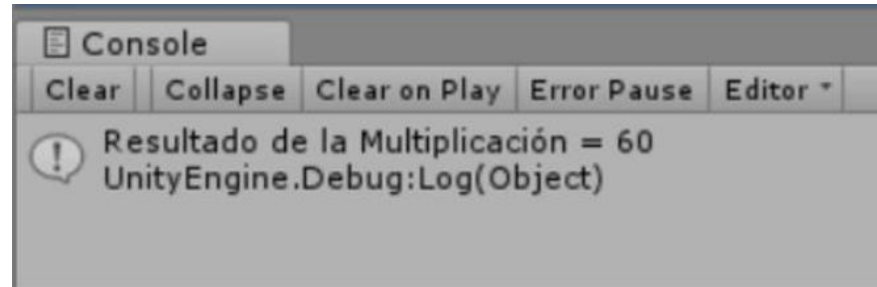
Multiplicación

Si cambiamos los valores de la siguiente manera:

- “12” para “*Primer Valor*”
- “5” para “*Segundo Valor*”



Y ejecutamos nuestra escena:



Podemos ver que la operación multiplicación se realizó correctamente ($12 * 5 = 60$)

Si modificamos las variables de nuestra clase:

```
public float primerValor;  
public int segundoValor;  
float resultadoOperaciones;
```

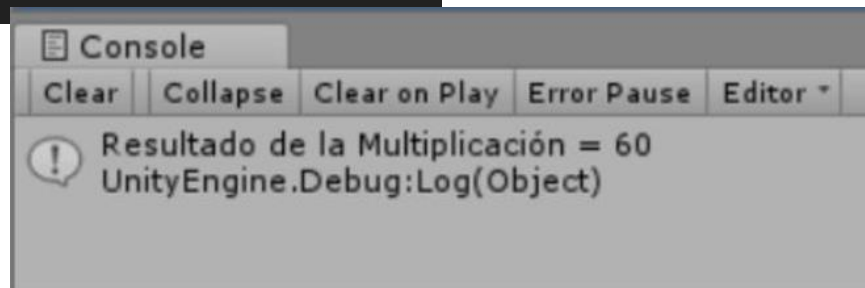
Multiplicación

Y también modificamos el contenido del método "Start":

```
void Start()  
{  
    resultadoOperaciones = primerValor * segundoValor;  
    Debug.Log("Resultado de la Multiplicación = " +  
resultadoOperaciones);  
}
```

Podemos ver que no se nos manda mensaje de error alguno (esto es porque, un valor tipo "int" si puede ser parte de una operación con resultado tipo "float").

Al guardar nuestro código y después ejecutar nuestra escena, veremos:



La Operación multiplicación se realizó correctamente ($12 * 5 = 60$)

División

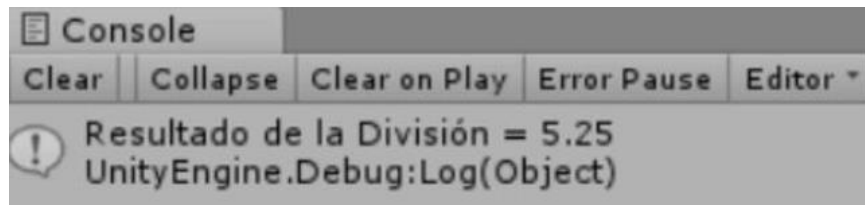
Ahora seguiremos con la Operación “División”, para esto, vamos a modificar las variables de nuestra clase:

```
public float primerValor;  
public int segundoValor;  
int resultadoOperaciones;
```

Y también modificamos el contenido del método “Start”:

```
void Start()  
{  
    resultadoOperaciones = 10.5f / 2;  
    Debug.Log("Resultado de la División = " + resultadoOperaciones);  
}
```

Al guardar nuestro código y después ejecutar nuestra escena, veremos:



La operación división se realizó correctamente ($10.5 / 2 = 5.25$).

División

En este caso no estamos involucrando a las variables *“primerValor”* y *“segundoValor”* para realizar la operación división, estamos ocupando solo valores numéricos; cabe destacar que, como el resultado de esta operación es almacenado en la variable *“resultadoOperación”* (tipo *“float”*), los valores numéricos con *“decimales”* deben incluir el sufijo *“f”* para ser tomados como tipo *“float”* (así *“10.5f”* es reconocido como *“float”*), de lo contrario se tomaría como otro tipo de dato que no es compatible con la variable *“resultadoOperación”* (con los valores numéricos *“enteros”* no hay problema alguno, estos siempre son reconocidos como *“int”* y no hay necesidad de modificación alguna).

División

Volvamos a modificar las variables de nuestra clase:

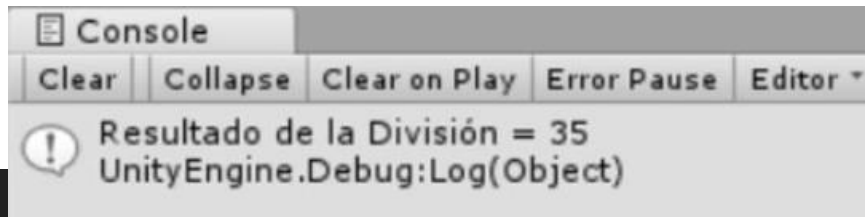
```
public int primerValor;  
public int segundoValor;  
int resultadoOperaciones;
```

Y también modificamos el contenido del método "Start":

```
void Start()  
{  
    resultadoOperaciones = primerValor / segundoValor;  
    Debug.Log("Resultado de la División = " + resultadoOperaciones);  
}
```

Guardamos nuestro código, cambiamos a Unity, y en la ventana **Inspector** el GameObject seleccionado escribimos:

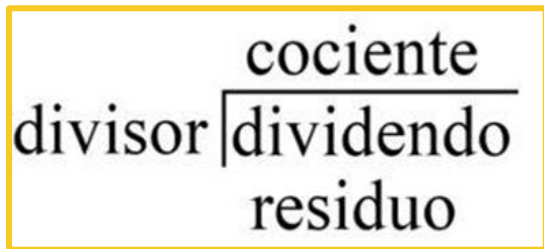
- "70" para "Primer Valor"
- "2" para "Segundo Valor"



La operación división se realizó correctamente ($70 / 2 = 35$).

Módulo

Por último, vamos a ver la operación **módulo**. Esta se define como la **obtención del resto o residuo de una división**.



A diagram illustrating the components of a division operation. It shows a horizontal line with 'cociente' (quotient) above it. Below the line, 'divisor' is on the left and 'dividendo' (dividend) is on the right. Below 'dividendo' is the word 'residuo' (remainder).

Su operador aritmético es “%” (“porcentaje”).

Su aplicación es muy amplia dentro de la programación. **El ejemplo más común de su uso es ayudar a reconocer si un número es par o impar.** Sabemos que un número se considera “par” si es divisible por 2 y no genera residuo (en otras palabras, una división con un residuo igual a cero).

Para llevar a cabo este ejemplo, crearemos las siguientes variables en nuestra clase:

```
public int primerValor;  
public const int segundoValor = 2;  
int resultadoOperaciones;
```

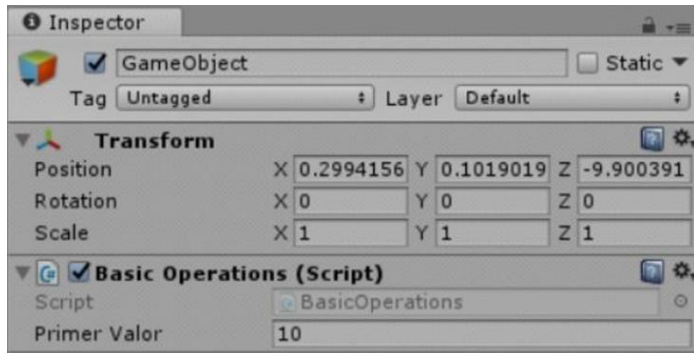
Módulo

Y modificamos el contenido del método “Start”:

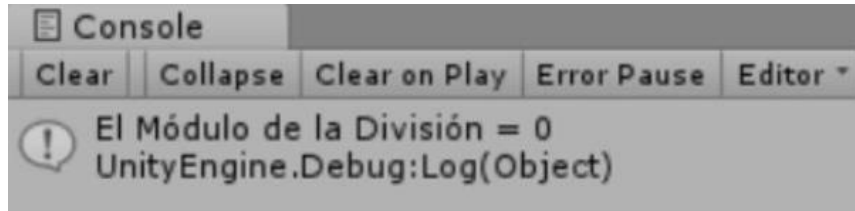
```
void Start()
{
    resultadoOperaciones = primerValor % segundoValor;
    Debug.Log("El Modulo de la División = " + resultadoOperaciones);
}
```

$$\begin{array}{r} 5 \\ 2 \overline{)10} \\ \underline{0} \end{array}$$

Ahora escribimos: “10” para “Primer Valor”



Y ejecutamos nuestra escena.



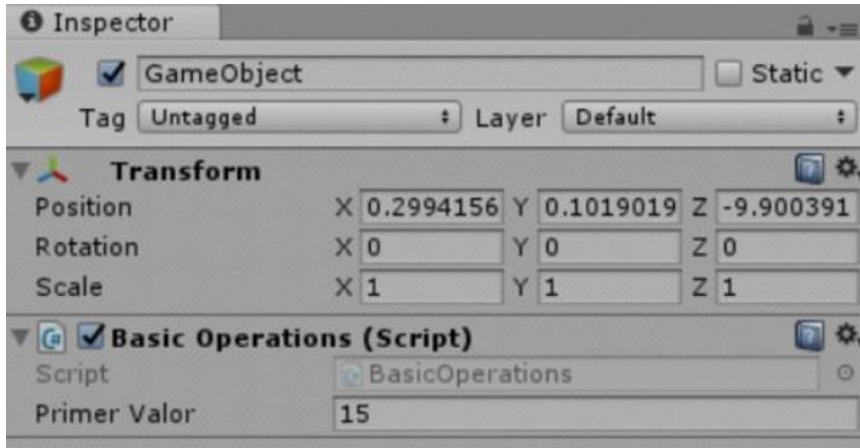
La operación módulo se llevó a cabo correctamente (10 / 2 = 5 y no hay residuo).

Podemos decir que “10” es un número “par”.

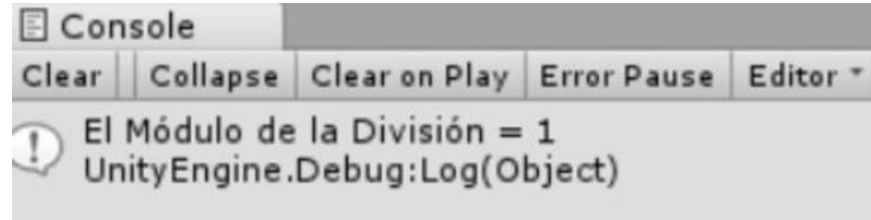
Módulo

Si intentamos con otro número:

- “15” para “Primer Valor”



Ejecutamos nuestra escena.



La operación módulo se llevó a cabo correctamente ($15 / 2 = 7$ y hay residuo = 1).

Podemos decir que **“15” es un número “impar”**.

$$\begin{array}{r} 7 \\ 2 \overline{) 15} \\ \underline{14} \\ 1 \end{array}$$

Tabla resumen

Operador	Operación	Descripción	Ejemplo
+	Adición	Suma dos valores	$x+y$
-	Sustracción	Resta dos valores	$x-y$
*	Multiplicación	Multiplica dos valores	$x*y$
/	División	Divide dos valores	x/y
%	Módulo	Se obtiene el resto de la división de dos valores	$x\%y$
++	Incremento	Incrementa una variable en 1	$x++$
--	Decremento	Decrementa una variable en 1	$x--$

Operadores de Asignación

Operador	Operación	Descripción	Ejemplo
=	Asignación	Asigna un valor a una variable	$x=y$
+=	Adición y Asignación	Suma el valor actual de la variable con otro valor y vuelve a asignar el valor a la variable.	$x+=3$ es equivalente a $x=x+3$
-=	Sustracción y Asignación	Resta el valor actual de la variable con otro valor y vuelve a asignar el valor a la variable.	$x-=3$ es equivalente a $x=x-3$
=	Multiplicación y Asignación	Multiplica el valor actual de la variable con otro valor y vuelve a asignar el valor a la variable.	$x=3$ es equivalente a $x=x*3$
/=	División y Asignación	Divide el valor actual de la variable con otro valor y vuelve a asignar el valor a la variable.	$x/=3$ es equivalente a $x=x/3$
%=	Módulo y Asignación	Divide el valor actual de la variable (quedándose con el resto) con otro valor y vuelve a asignar el valor a la variable.	$x\%=3$ es equivalente a $x=x\%3$

Conversiones de Tipos

Fuentes: <https://www.ackosmic.com> / <https://docs.microsoft.com>

Conversiones de Tipos

Dado que C# tiene tipos estáticos en tiempo de compilación, después de declarar una variable, no se puede volver a declarar ni se le puede asignar un valor de otro tipo a menos que ese tipo sea convertible de forma implícita al tipo de la variable. Por ejemplo, “string” no se puede convertir de forma implícita a “int”. Por tanto, después de declarar *i* como un valor “int”, no se le puede asignar la cadena "Hello", como se muestra en el código siguiente:

```
int i;  
// error CS0029: Cannot implicitly convert type 'string' to 'int'  
i = "Hola";
```

Pero es posible que en ocasiones sea necesario copiar un valor en una variable o parámetro de método de otro tipo. Por ejemplo, es posible que tenga una variable de entero que se necesita pasar a un método cuyo parámetro es de tipo “double”. O es posible que tenga que asignar una variable de clase a una variable de tipo de interfaz.

Conversiones de Tipos

Estos tipos de operaciones se denominan conversiones de tipos. En C#, se pueden realizar las siguientes conversiones de tipos:

- **Conversiones implícitas:** no se requiere ninguna sintaxis especial porque la conversión siempre es correcta y no se perderá ningún dato. Los ejemplos incluyen conversiones de tipos enteros más pequeños a más grandes, y conversiones de clases derivadas a clases base.
- **Conversiones explícitas:** las conversiones explícitas requieren una expresión *Cast*. La conversión es necesaria si es posible que se pierda información en la conversión, o cuando es posible que la conversión no sea correcta por otros motivos. Entre los ejemplos típicos están la conversión numérica a un tipo que tiene menos precisión o un intervalo más pequeño, y la conversión de una instancia de clase base a una clase derivada.

Conversiones Implícitas

Para los tipos numéricos integrados, se puede realizar una conversión implícita cuando el valor que se va a almacenar se puede encajar en la variable sin truncarse ni redondearse. Para los tipos enteros, esto significa que el intervalo del tipo de origen es un subconjunto apropiado del intervalo para el tipo de destino.

Por ejemplo, una variable de tipo “*long*” (entero de 64 bits) puede almacenar cualquier valor que un tipo “*int*” (entero de 32 bits) puede almacenar. En el ejemplo siguiente, el compilador convierte de forma implícita el valor de *num* en la parte derecha a un tipo *long* antes de asignarlo a “*bigNum*”.

```
// Conversión implícita, un long puede guardar cualquier valor que pueda guardar un int  
int num = 2147483647;  
long bigNum = num;
```

Conversiones Explícitas

Pero si no se puede realizar una conversión sin riesgo de perder información, el compilador requiere que se realice una conversión explícita, **casting**. El casting es una manera de informar explícitamente al compilador de que se pretende realizar la conversión y se es consciente de que se puede producir pérdida de datos o la conversión de tipos puede fallar en tiempo de ejecución.

Para realizar una conversión, hay que especificar el tipo al que se va a convertir entre paréntesis delante del valor o la variable que se va a convertir. El siguiente programa convierte un tipo “*double*” en un tipo “*int*”. El programa no se compilará si no se hace la conversión.

```
double x = 1234.7;  
int a;  
// Castear double a int  
a = (int)x;  
Debug.Log(a);
```

Acceso al componente Transform


Podemos acceder por script al componente **Transform** de nuestro GameObject.

Para esto se debe escribir el nombre del componente seguido por "." y el nombre del método o propiedad a acceder. Por ejemplo: *transform.position* o *transform.rotation*

Vector 3

En Unity nos encontraremos con tipos de datos que son exclusivos de Unity y los cuales nos permitirán editar algunos componentes de un GameObject como el Transform. Uno de los tipos de datos en cuestión es el Vector3 el cual permite guardar un vector de tres posiciones de la misma manera en que funciona la posición de cualquier GameObject en escena.





Agencia de
Aprendizaje
a lo largo
de la vida