2. PROGRAMACIÓN ORIENTADA A OBJETOS....2
Introducción UNITY....2
UNITY HUB....2
UNITY COLLAB....3
CONCEPTOS BÁSICOS POO....4
01. Nuestro primer script C#....4
1.1. Nuestra primera instrucción....6
02 – Ejecución secuencial....9
03. AÑADIENDO COMENTARIOS....18
Información de Ampliación: Flujo de trabajo de los Assets (Asset Workflow)....21
04 – CORRIGIENDO ERRORES....22
05 - Variables....30
06 - Definiendo arrays....37
07 - Operadores y expresiones. Parte 1....42
08 - Operadores y expresiones. Parte 2....42
09 - Operadores y expresiones. Parte 3....43
CONCEPTOS AVANZADOS POO....44
01 – Instrucción condicional: IF....45
02 - Instrucción de selección : SWITCH....45
03 – Estructura repetitiva: WHILE Y DO - WHILE....45
04 – Estructura repetitiva: FOR....46
05 - Estructura repetitiva FOREACH....46
06 – BREAK....46
07 – MÉTODOS....47
08 – MÉTODOS CON PARÁMETROS....48

2. PROGRAMACIÓN ORIENTADA A OBJETOS

Introducción UNITY

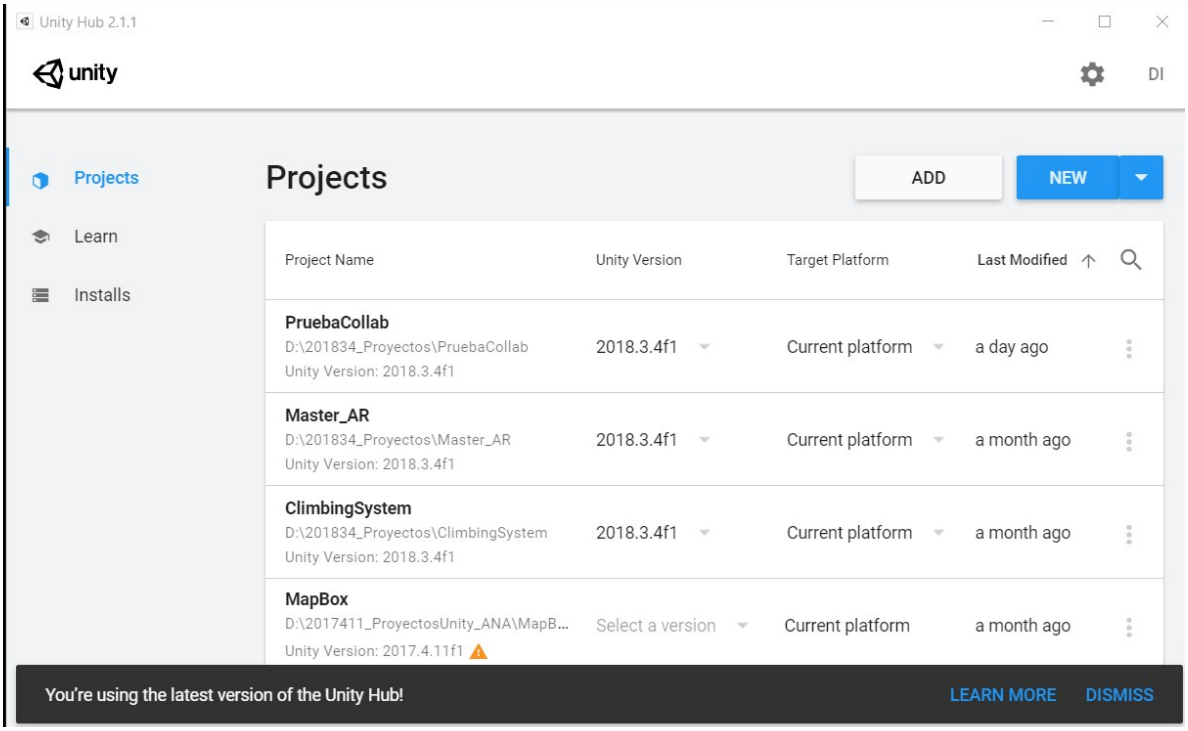
Unity es un motor de videojuego. Es compatible con Maya, Autodesk, Blender, Photoshop, Fireworks, Softimage y un sinfín más de herramientas. Además es multiplataforma, permitiendo el desarrollo en casi cualquier sistema: Xbox, PlayStation, Wii, iPad, OS X o Android. **Hace sencillo algo que hace apenas una década era un terrible dolor de cabeza**: cuando estás escribiendo código en la parte izquierda, en la derecha se reflejan los cambios en tiempo real. Cualquier bloguero o editor de WordPress sabrá ver el símil.

Permite crear *tags*(palabras clave) y atajos sobre los directorios de los archivos. Dispone de 2 tipos de licencia distintos: Professional y Personal.

Acceder a la página web de Unity y crear un usuario con la licencia Personal para acceder a los diferentes servicios que ofrece la plataforma.

UNITY HUB

Instalar Unitu Hub usando el ejecutable 'UnityHubSetup.exe' disponible en la carpeta de material o en la siguiente URL: descarga Unity Hub
Una vez instalado, es posible añadir diferentes versiones de la herramienta. De esta forma, se obtierne un entorno de trabajo de rápido acceso permitiendo gestionar los proyectos y las diferentes versiones de Unity en las que se crearon.



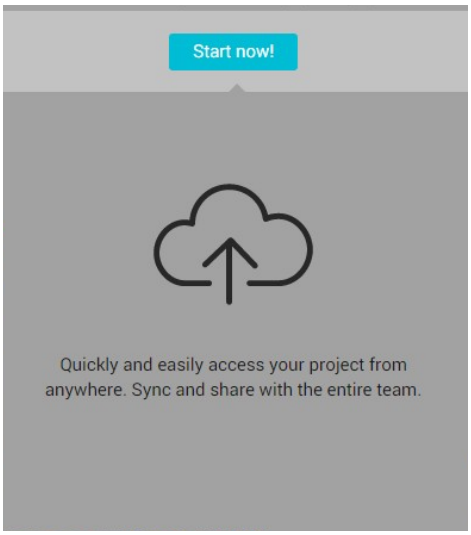
UNITY COLLAB

Acceder al servicio en linea que ofrece Unity para la colaboración en linea de los proyectos.

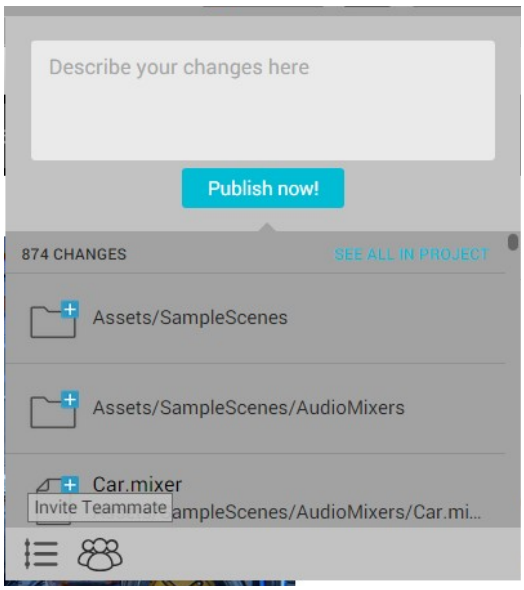
Mediante esta herramienta, con la licencia Personal, es posible compartir hasta 3 proyectos. Para el presente curso, es obligatorio dar de alta este servicio en la cuenta personal del alumno o alumna.

Documentación oficial Unity

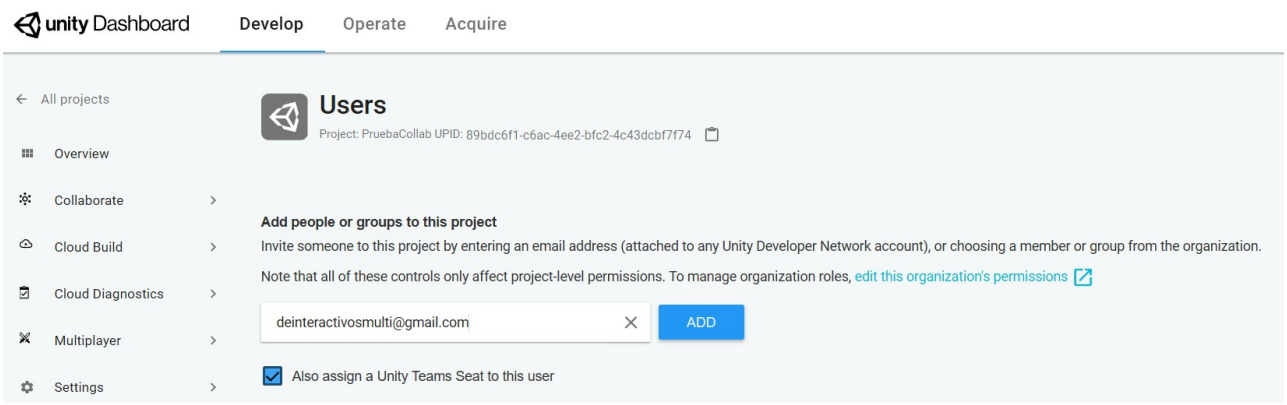
Al generar un proyecto, añadirlo a la nube y compartirlo con la siguiente dirección de correo electrónico: **deinteractivosmulti@gmail.com**



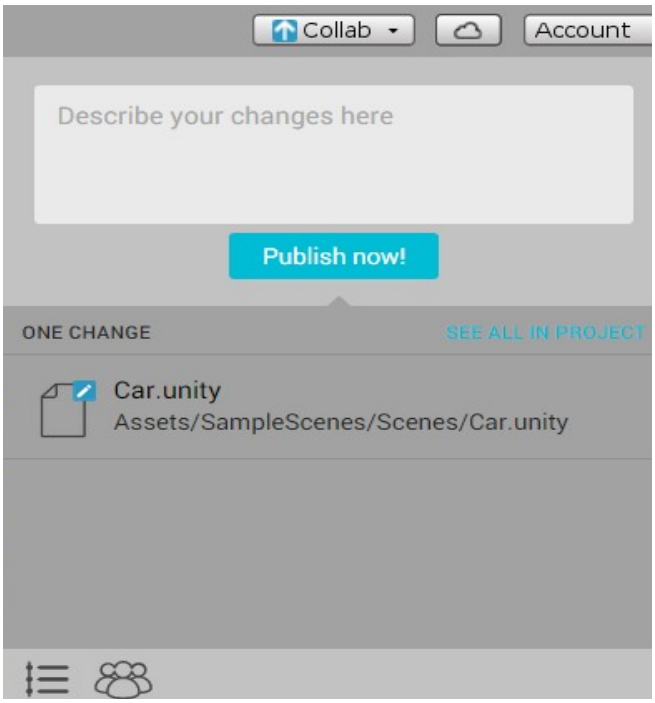
Una vez los archivos del proyecto están almacenados en la nube, compartir:



Accede al cuadro de mandos de nuestro usuario Unity : ‘unity Dashboard‘ donde es posible añadir la dirección electrónica indicada.



Una vez el proyecto está compartido, debe publicarse.



CONCEPTOS BÁSICOS POO

01. Nuestro primer script C#

Preparación del entorno de trabajo: dejar los paneles a nuestro gusto y ‘save layout’
Si en algún momento los paneles se descolocan, se selecciona la configuración guardada y se reestablecen los paneles

-Paneles

Project – Explorador de archivos que permite navegar por todos los archivos de nuestro proyectos

Crear una carpeta ‘Script’ donde ir guardando los scripts de este curso.
Botón derecho: ‘Crear Carpeta’

Creamos carpeta ‘Teoría’

Consola – Muestra los resultados de la ejecución del programa, errores…

Crear un script
Seleccionar la carpeta ‘Teoria’ : ‘Crear-C# Script’ --> T01

NOTA: El nombre de los scripts NO debe contener espacios, NI debe de empezar por un número. Se recomienda usar sólo letras y números. Si el nombre está compuesto de más de una palabra, ponerlas juntas, con cada primera letra de cada palabra en mayúscula. Ejemplo: EjemploDeScript.cs, SaltarAlto.cs, EjercicioNuevo.cs, EjecutarAccion12.cs, etc.

En proyectos reales, el nombre debe dar indicaciones de lo que va a hacer el script que vamos a programar.

Queremos que el personaje salte al pulsar una tecla, podemos llamarlo ‘ComportamientoSaltar’

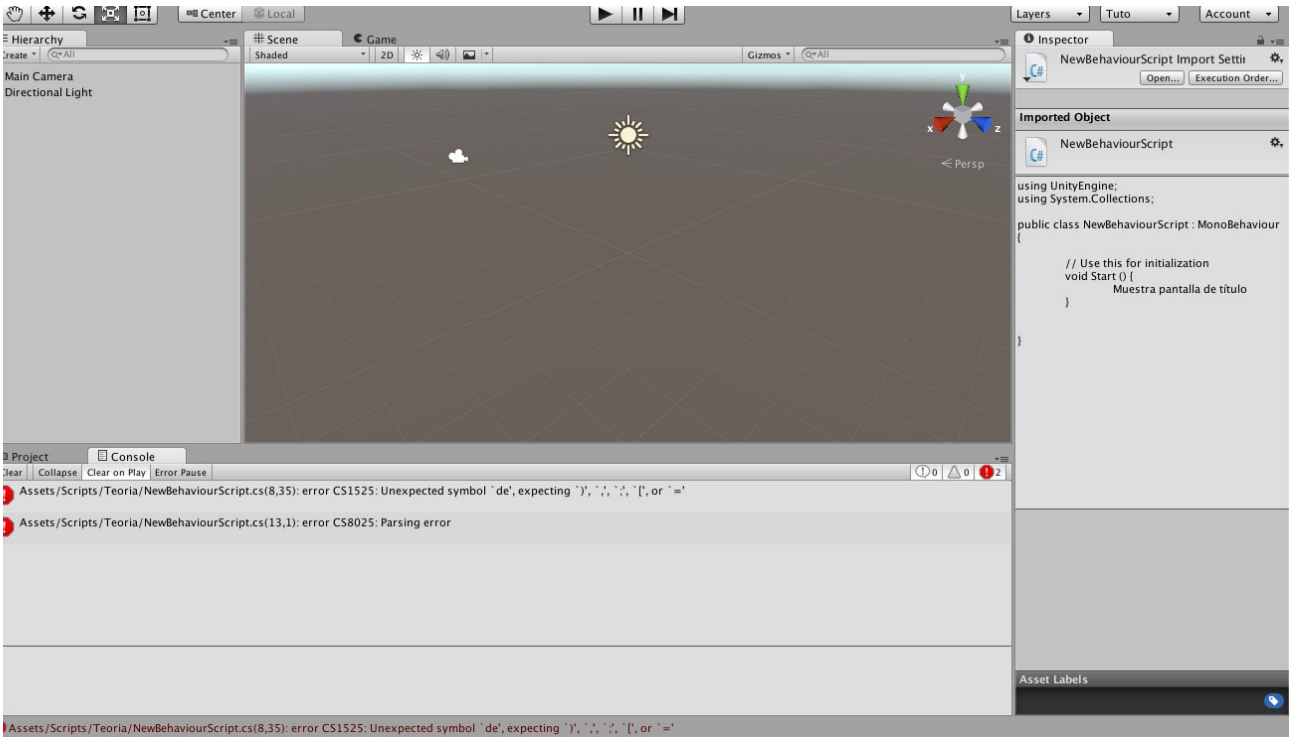
Doble click para abrirlo y ejecuta el editor de texto que viene con Unity por defecto:
Monodeveloper. En esta práctica se configura para usar Visual Studio 2019 Community.

Podemos observar que solo con crear un archivo ya se ha generado código
Como no lo vamos a necesitar, eliminamos el código desde la linea 11 a la 14
Entre la llave de la línea 7 y 9 vamos a escribir nuestro código, ignoramos el resto de código por ahora.

Queremos escribir un código que Unity lo ejecute y muestre la pantallas de título

```
1 using UnityEngine;
2 using System.Collections;
3
4 public class NewBehaviourScript : MonoBehaviour {
5
6     // Use this for initialization
7     void Start () {
8         Muestra pantalla de título
9     }
10
11
12 }
13
```

Guardamos el archivo y … error



Unity no entiende el lenguaje humano, entiende: JavaScript(parecido a), C# y Boo.
Para poder darle instrucciones a Unity debemos escribir scripts en uno de estos tres lenguajes

Eso vamos a aprender, a comunicarnos con Unity y poder darle instrucciones para indicarle que queremos que haga.

1.1. Nuestra primera instrucción

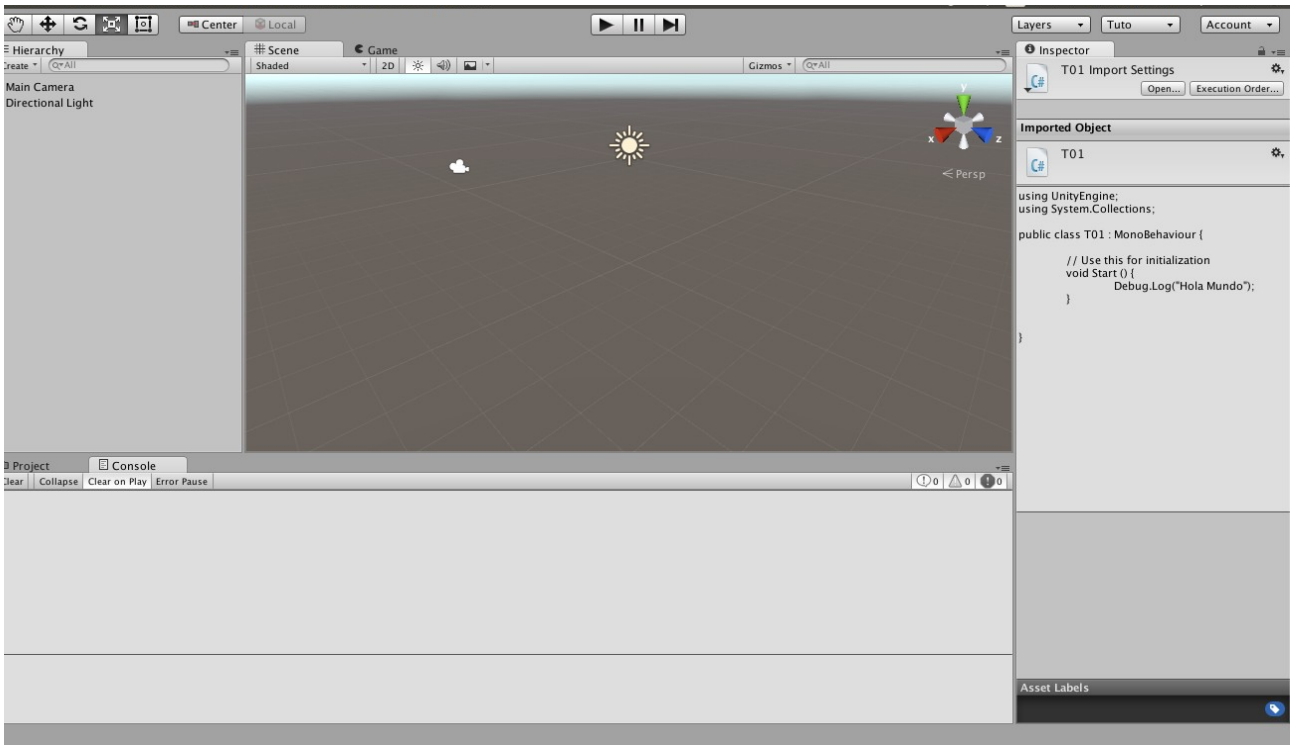
Una instrucción que vamos a usar bastante es:

Debug.Log(“Hola mundo”);

y guardar el archivo.

Debe escribirse tal y como está. Letras minusculas y mayusculas.
Parentesis que abre, debe cerrarse, igual con comillas
; indica el fin de la instrucción

Guardar y volver a Unity
Ha desaparecido del error



Si por ejemplo , cambiamos algo de la instrucción , Unity, devolverá error de nuevo.

Ya hemos dado nuestra primera instrucción y queremos que Unity la ejecute

Buscamos el botón de reproducir arriba de la pantalla y pulsamos.
Unity no muestra nada por consola.
Esta instrucción debería sacar Hola mundo por consola.
Cuando programamos scripts lo que estamos definiendo son comportamientos.

Si definimos un comportamiento debemos asignarselo a un objeto del proyecto, que se encuentre en la escena de Unity, si no se asigna a ningún objeto, no se va a ejecutar.

En Jerarquía, hay dos objetos: cámara y una foto guardad
Asignar un comportamiento a un objeto, diferentes formas:

- Arrastrando el script al objeto a la jerarquía
- Arrastrando el script al editor visual, no recomendable por poder aplicar un comportamiento a otro objeto no deseado
- Seleccionar un objeto en la jerarquía. El panel Inspector recibe información de ese objeto. Se arrastra el script hasta la parte inferior del panel Inspector
- Botón añadir componentes del Inspector y selecciona el script

Un mismo comportamiento se puede asignar a varios objetos.
Se puede eliminar comportamiento de un objeto con botón derecho o rueda de configuración al lado del script asignado al objeto

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

}
```

Ejercicio E01 C#

Igual a lo explicado para el script T01 pero programar que salga por consola vuestro nombre

Solución:
En la carpeta Ejercicios, generamos un script E01

```
using UnityEngine;
using System.Collections;

public class E01 : MonoBehaviour {

    // Use this for initialization
    void Start () {
     Debug.Log("Hola Unity, mi nombre es Ana");
    }

}
```

Se añade nuestro nuevo comportamiento a un objeto de nuestro proyecto Unity.
En este caso, vamos a añadir el nuevo comportamiento(script E01) a la camara.
Ejecutamos para ver que muestra el mensaje que se ha indicado en el script E01.

Para continuar con la práctica, vamos a limpiar los objetos de Unity de los comportamientos que le hemos añadido.

02 – Ejecución secuencial

Vamos a organizar instrucciones dentro del código y usar el depurador para ver como se ejecutan pasa a paso.

¿PARA QUÉ SIRVE DEPURAR EL CÓDIGO?

1. El uso más importante es que cuando programas algo, a veces el código no hace lo que el programador cree que debería ocurrir (por algún despiste del programador). Depurar paso a paso sirve para encontrar la línea exacta en el código donde la ejecución deja de comportarse como espera el programador, y cambiarlo para que haga lo que el programador espera. (Resumen: Depurar sirve para encontrar bugs)

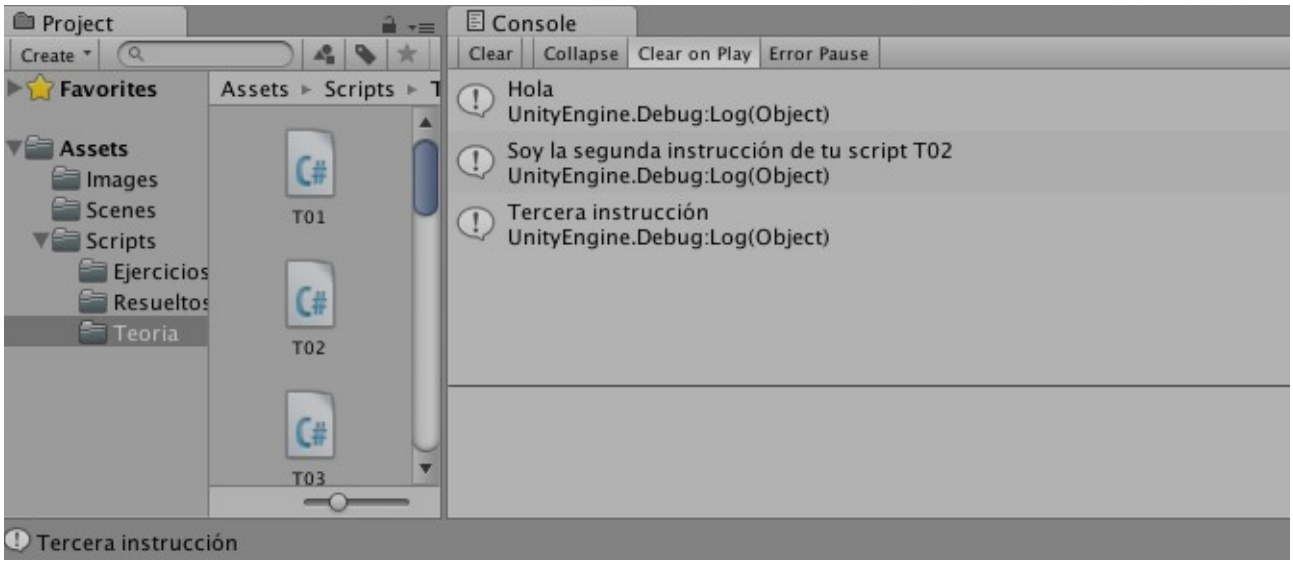
2. También sirve para que alguien que está empezando a programar pueda entender el flujo de ejecución del código y el orden en el que este se ejecuta. A veces, cuando la teoría no queda clara, con sólo depurar un ejemplo de código, se "ve" cómo este se ejecuta.

Buscar el script T02 en nuestro proyecto.

El carácter ‘;’ indica el fin de cada instrucción. Para añadir varias instrucciones, simplemente, escribimos diferentes instrucciones sin olvidar punto y coma al final de dicha instrucción.
Como solo conocemos esta instrucción, para probar, la repetimos y podemos cambiar el mensaje:

```
        Debug.Log("Hola");
        Debug.Log("Soy la segunda instrucción de tu script T02");
        Debug.Log("Tercera instrucción");
```

Guardamos el script y volvemos a Unity.
Vemos que no devuelve error, añadimos este comportamiento a la camara y si ejecuto vemos los mensajes de nuestras tres instrucciones.



Unity ha ejecutado una instrucción detrás de otra. Para mayor claridad, escribimos cada instrucción en una línea.
Podemos añadir tantas veces como queramos la instrucción.
Para ver mas claramente como se ejecutan las instrucciones, vamos a depurar el código. Va a mostrar señalada la instrucción que se va a ejecutar.

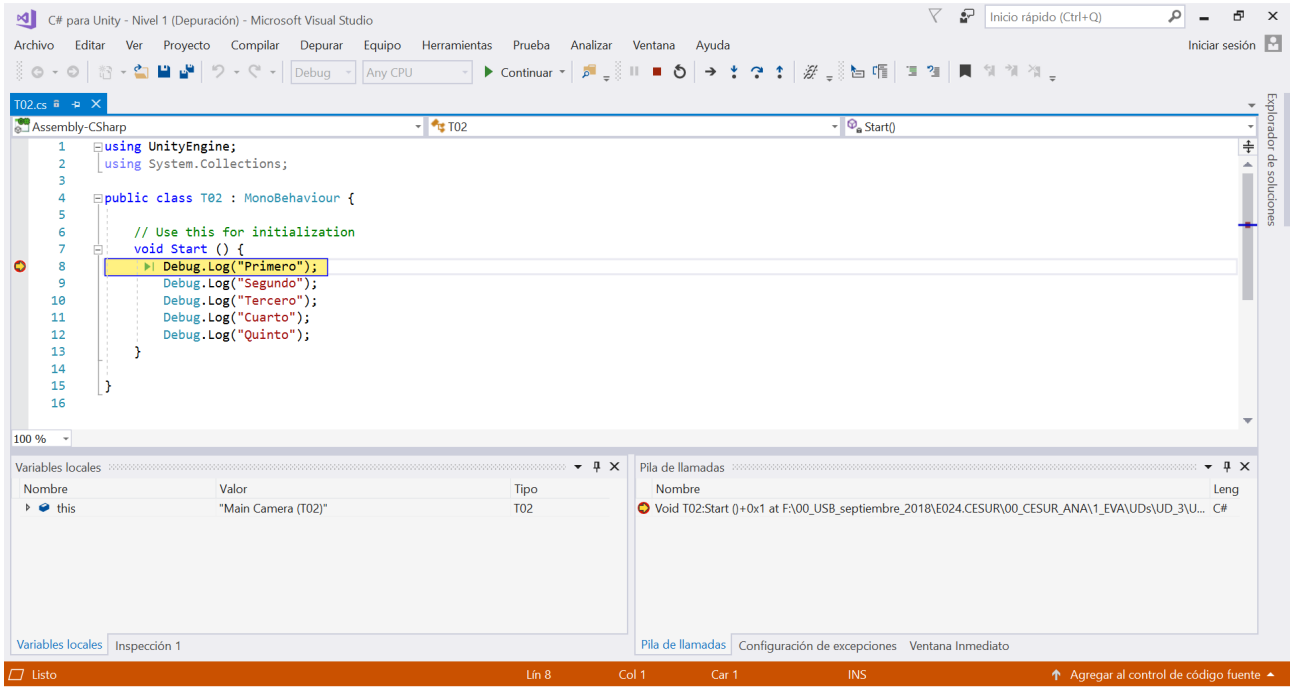
Para iniciar la depuración, vamos a Unity y, en un primer paso, nos aseguramos que la ejecución está detenida (botón pausa en parte superior de la pantalla)

Preparación del entorno para depurar código:

En Visual Studio y marcamos el punto de ruptura

Si no hemos marcado ningún punto de ruptura, la ejecución no se va a parar puesto que no existe punto indicado en el código donde debe parar la ejecución.

Al ejecutar en Unity, se detiene en la linea del punto de ruptura y el menú de depuración se habilita:



La flecha amarilla indica en que punto del script se ha detenido la ejecución.

•**Mostrar la instrucción siguiente** → Indicamos que termine la ejecución y que no se detenga hasta el fin del script o hasta el siguiente punto de ruptura.

•**Paso a paso por instrucciones - Step Over** → Ejecuta la instrucción del punto de ruptura y se detiene en la siguiente esperando a ver que es lo que queremos hacer. Pulsando este botón repetidas veces, ejecutará hasta llegar al fin del script.

La tecla F11 es el **paso a paso** comando y hace avanzar la una instrucción ejecución de aplicación a la vez. F11 es una buena manera para examinar el flujo de ejecución en el nivel más detallado

•**Paso a paso por procedimientos** → La tecla **F10** hace avanzar el depurador sin entrar en las funciones o métodos en el código de aplicación .

•**Paso a paso para salir** → Pulsando Mayús.+F10, ejecuta paso a paso hasta salir

sudo

Para mas información sobre el menú de depuración, visitar:
https://docs.microsoft.com/es-es/visualstudio/debugger/getting-started-with-the-debugger?view=vs-2017
Resumiendo:
Las instrucciones se van a ejecutar unas detrás de otras, de forma secuencial.
Cada instrucción acaba con un punto y coma.
Las llaves agrupan secuencias de instrucciones.

Ejercicio E02 C#

Crear un script con el nombre E02 en la carpeta Ejercicios que saca por consola en una linea el nº1, en otra linea el nº2 y así sucesivamente hasta el n.º5.
Solución:

```
using UnityEngine;
using System.Collections;

public class T02 : MonoBehaviour {

	// Use this for initialization
	void Start () {
		Debug.Log("Primero");
		Debug.Log("Segundo");
		Debug.Log("Tercero");
		Debug.Log("Cuarto");
		Debug.Log("Quinto");
	}

}
```

Ejecutamos y vemos la salida que muestra los número del 1 al 5.

Vamos a depurar este nuevo script:
1. Iniciamos la depuración
2. Marcamos un punto de ruptura en la instrucción que muestra el apartado nº2 haciendo click en el borde gris
3. Ejecutar paso a paso las dos instrucciones siguientes (F11) y luego pulsamos el botón de finalizar hasta salir de la ejecución (Mayús.+F11)

Eliminamos los componentes de los objetos de Unity para los próximos ejercicios.

03. AÑADIENDO COMENTARIOS

Vamos a escribir comentarios en el código. Para ello, buscar el script T03 y lo abrimos con Visual Studio.

Eliminamos el código que no usamos. Copiamos el contenido del script del ejercicio 2 ‘E02’.

Lo que tenemos en el script es un archivo de código.
Los **comentarios** son anotaciones que realizamos en el código de forma que Unity cunado las vea sabe que no tiene que hacer caso de lo que ahí se escribe. Deben tener información útil o explicaciones de lo que va a hacer un bloque de código concreto. Así si en un futuro volvemos al script con leer el comentario ya sabemos lo que hace sin tener que analizar las líneas para saber lo que hace.
Por ahora es muy sencillo pero cuando se amplíe el código será complicado saber el funcionamiento a simple vista .

Tipos de comentarios:
Comentario de una sola linea:

```
//Esto es un comentario de una sola linea
```

Comentarios multilinea

```
/* Esto es
un ejemplo
de comentario
multilinea.
Unity ignorara
todo lo que
haya a partir
de la /*, hasta
que se encuentre
con un */
```

A recordar: LOS COMENTARIOS NO SE CONSIDERAN INSTRUCCIONES

Si ponemos un punto de ruptura en una linea considerada como comentario, el punto de ruptura se establecerá en la siguiente instrucción.

Ejercicio E03 C#

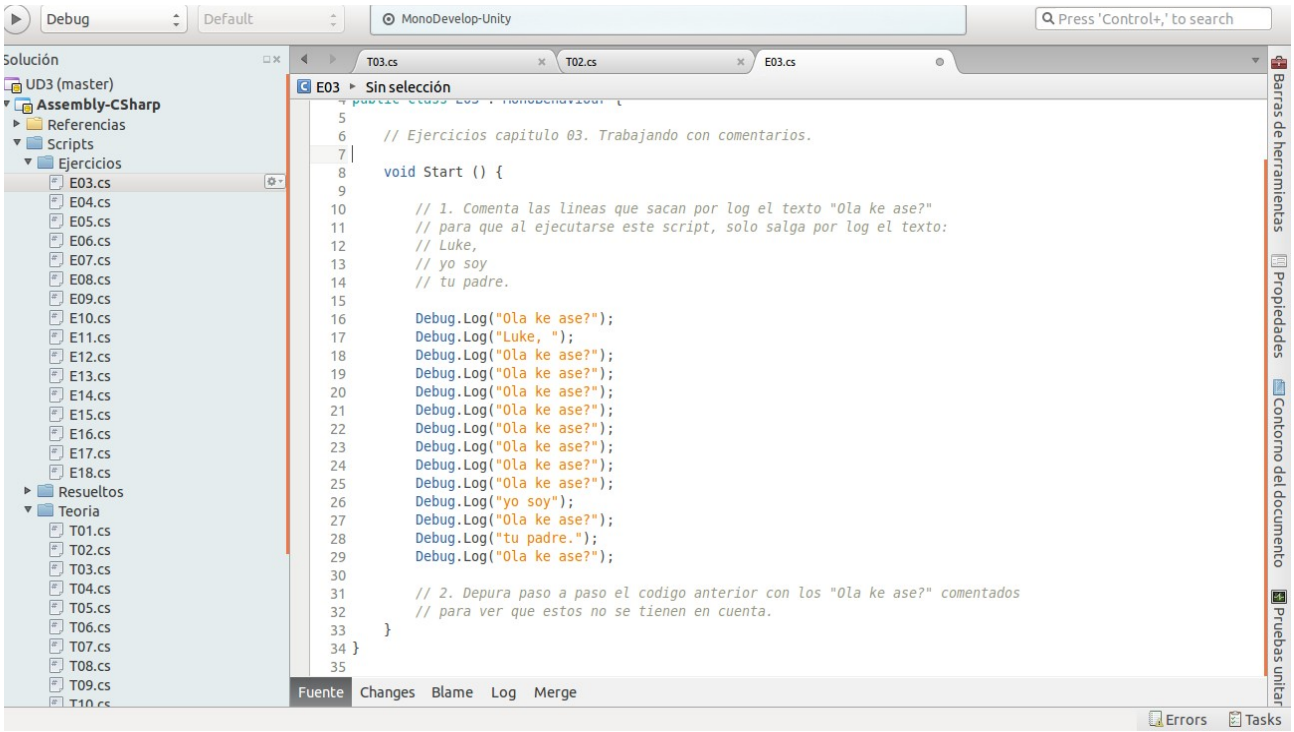
Vamos a crear el script E03 en la carpeta de ejercicios del proyecto.

Vamos a hacer dos ejercicios muy similares a lo que hemos visto ya la teoría.

El primer ejercicio consta de lo siguiente:

Comentar todas las líneas que saquen por consola el texto “ola que ase” usando comentarios de una sola línea y comentarios multi líneas cuando sea necesario.
Luego hacéis que se ejecute el script y comprobais que sólo sale por consola el texto “luke yo soy tu padre”

Solución:



Aquí tenemos la primera línea que queremos comentar y eso lo hacemos poniendo barra delante luego aquí tenemos muchas líneas seguidas que queremos comentar y, como tenemos líneas seguidas, vamos a usar un comentario multi línea, con barra asterisco y lo cerramos con asterisco barra.

Para tener el código más claro se suele hacer así: dejar la línea de separación que abre el comentario de éste en su propia línea y cerrar comentario multilinea esté en su propia línea

Luego ya nos faltan dos líneas más que comentar. Estas son sencillas puesto que se comentan con la marca de comentarios de una sola línea.

Ya todo está en gris – las instrucciones que sacan “Luke yo soy tu padre”.

Pulso “control+s” para guardar, vuelve a Unity, termina de generar y no tengo ningún error.

Aplico el script T03 a la cámara, comprueba que lo tiene como componente y ejecuta.

Debe mostrar la salida: “luke yo soy tu padre”

Lo siguiente es iniciar la depuración de este código y ejecutar Antes, poner un punto de ruptura en la primera línea y ejecutar el código paso a paso hasta el final, observando que sólo se ejecutan las líneas que se consideran instrucciones y que no se ejecutan las líneas que están marcadas como comentarios.

Recuerda:tienes que iniciar la depuración de este código, poner un punto de ruptura en la primera línea y ejecutar paso a paso cada línea hasta el final observando que solo se ejecutan las líneas que no son comentarios

Siempre ir a unity antes de depurar y comprobar que no esté ejecutando ya el proyecto.

Inicia depuración pulsando el botón play de Visual Studio selecciono unity editor

Poner el punto de ruptura en la primera instrucción, si lo pusiéramos en los comentarios, no sería a la primera instrucción.

Ejecuta el código para que se detenga la ejecución donde he puesto el punto de ruptura: vamos a Unity y pulsa reproducir.

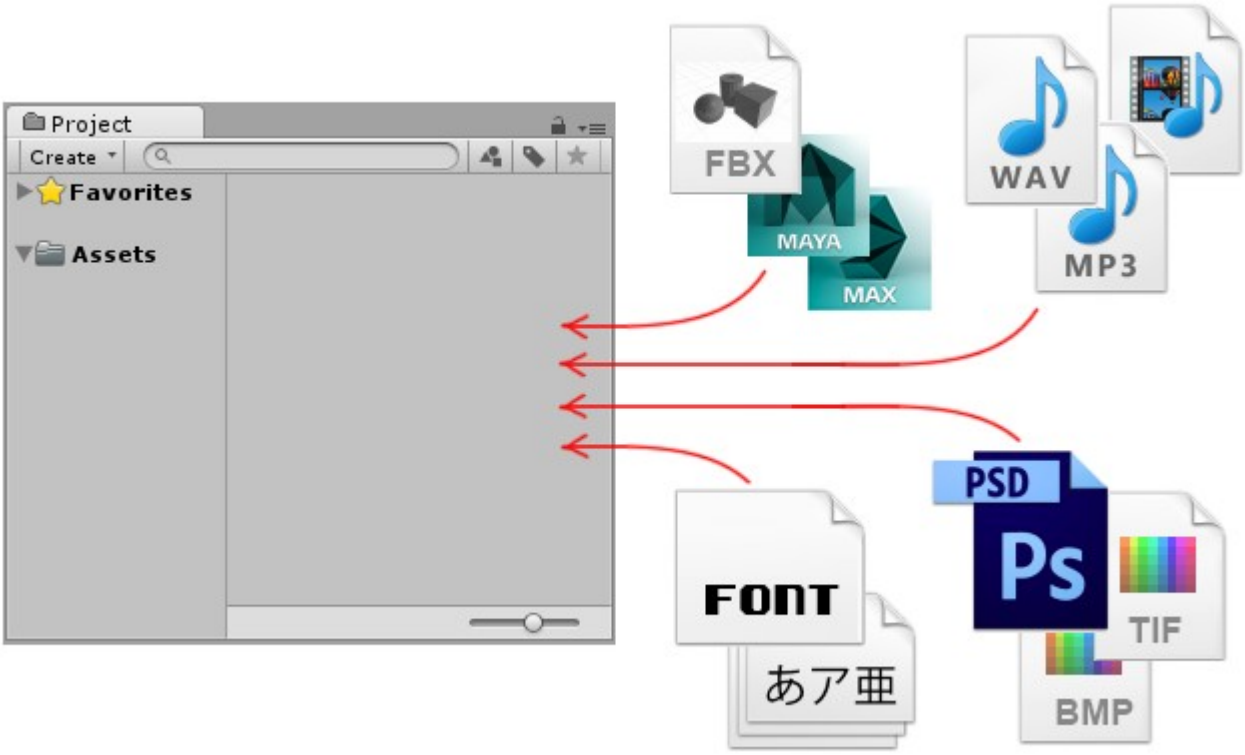
Se detendrá código y ya sólo nos queda ejecutar paso a paso cada línea hasta el fina observando que va saltando de instrucción a instrucción.

Observando que no hace caso a lo que hay marcado como comentario lo que hay de color gris voy a ejecutar paso a paso, una vez y otra vez.

Quita el punto de ruptura vuelve a unity, detiene la ejecución. Detiene la ejecución de depuración en Visual Studio.

Información de Ampliación: **Flujo de trabajo de los Assets (Asset Workflow)**

Un asset es una representación de cualquier item que puede ser utilizado en su juego o proyecto. Un asset podría venir de un archivo creado afuera de Unity, tal como un modelo 3D, un archivo de audio, una imagen, o cualquiera de los otros tipos de archivos que Unity soporta. También hay otros tipos de asset que pueden ser creados dentro de Unity, tal como un Animator Controller, un Audio Mixer o una Render Texture.



Algunos de los tipos de Asset que pueden ser importados a Unity

04 – CORRIGIENDO ERRORES

Vamos a aprender a solucionar los errores más comunes que podríamos cometer mientras programamos en C#.

Crear el script que vamos a usar: T04. Lo abro, y como siempre, borrar el bloque que no se necesita.

Coger las líneas de código del script T02 y poner en nuestro script, ya que están escritas correctamente y no dan error.

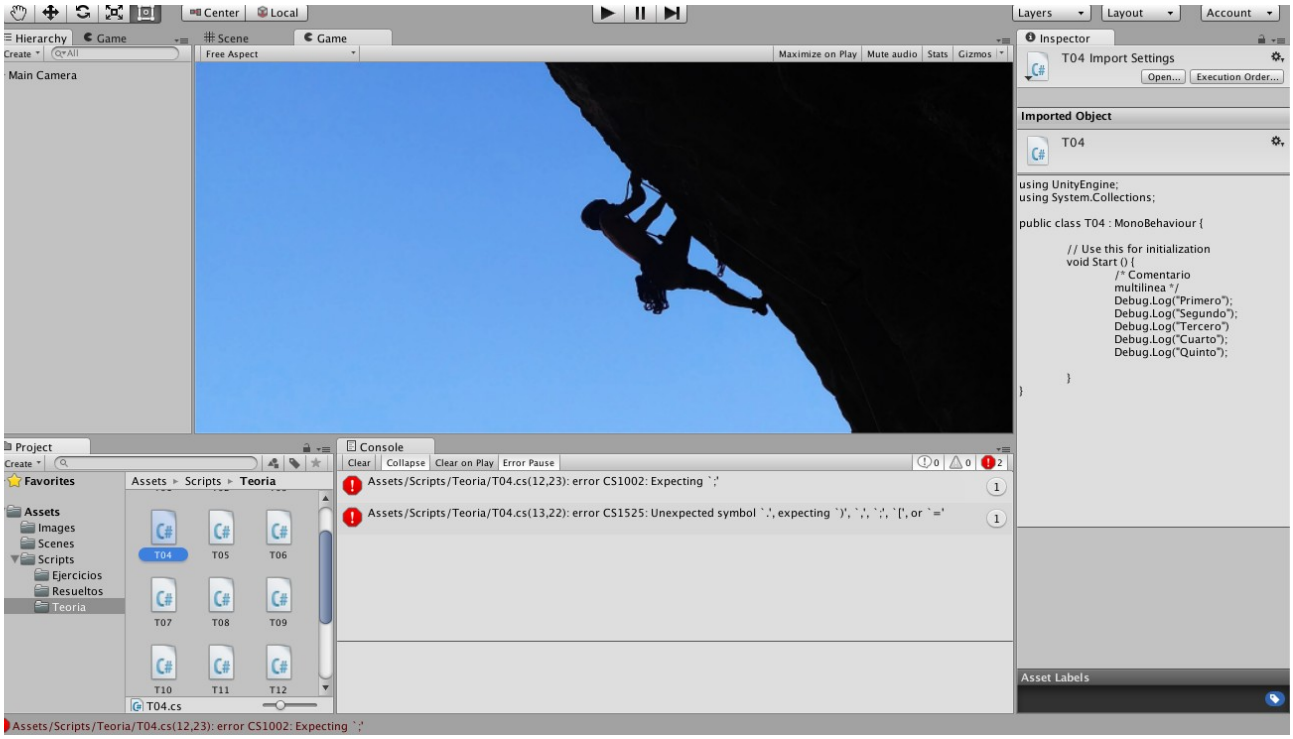
```
using UnityEngine;
using System.Collections;

public class T04 : MonoBehaviour {

    // Use this for initialization
    void Start () {
        /* Comentario
        multilinea */
        Debug.Log("Primero");
        Debug.Log("Segundo");
        Debug.Log("Tercero");
        Debug.Log("Cuarto");
        Debug.Log("Quinto");

    }
}
```

Pulsar control+s para guardar. Ir a Unity, termina de generar y no hay ningún error.

Vamos a ver qué hacer cuando tenemos errores de generación, como analizarlo y como resolverlo.

El primer error que se suele cometer tanto alguien que empieza a programar como un programador avanzado es que a veces se nos olvide poner el punto y coma al final de una instrucción o a lo mejor borrando borramos de más y borramos el punto y coma de alguna instrucción que sí lo tenía.

Imaginamos que se ha olvidado poner este punto y coma pulsa control+s para guardar y al volver a Unity ya tenemos el error.



Aquí aparecen dos errores. El truco es fijarnos en el que hay más arriba del todo, que suelen ser el que ha generado en cadena todos los demás.

Si vemos la descripción del error nos dice un montón de información útil como por ejemplo el nombre del script en el que está el error: assets/Scripts/Teoría/T04.cs.

Luego entre paréntesis esta información es muy importante: el primer número se refiere a las líneas donde ha encontrado o donde se ha dado cuenta de que había un error y luego el siguiente número es el número de columna.

Si vamos al script T04.cs y a la línea indicada veremos que es la línea en donde hemos quitado queriendo el punto y coma.

Pero estamos haciendo como que se nos ha olvidado ponerlo. Podrás leer que esperaba punto y coma y está diciendo que en esta línea se esperaba que pusiéramos un punto y coma.
Si muestra entre comillas simples un punto y coma, vas a la línea indicada de este archivo y vemos que es porque se ha olvidado poner el punto y coma al final de la instrucción.
CTRL+s, vuelve a Unity.

Otro error muy común es el no cerrar algo que abrimos o el no abrir que cerramos, como por ejemplo paréntesis, llaves o comillas o comentario multilíneas.

Vamos a empezar con las llaves ya qué el error no nos dice nada de que no nos está faltando una llave.

Cuando hago clic en cerrar llave , Visual Studio nos señala la llave que es pareja de la que hemos señalado, si señalamos la de cerrar Visual Studio nos resaltan la de abrir.

Muy importante, si se eliminara una llave se tendría en cuenta que la que cierra la llave es justamente antes y cuando está seleccionada,me está marcando justamente la que hay anterior de forma que se guarda y volver a Unity, tendría un error.

Dice la línea indicada, donde tenemos un error. Normalmente cuando indica que tenemos un error en la última línea de nuestro código es porque nos hemos dejado una llave sin cerrar. Hay que cerrarla y volver a Unity y ver que no hay error.

Igual pasa con las llaves de abrir. Las borramos por cualquier motivo y mira ahora la lista de errores que aparecen.

El truco es fijarnos en el primero que aparece. Línea indicada y ahora a cualquier cosa que me digan entre comillas simple ‘Debug’ Nos daríamos cuenta de que falta la llave de abrir. Si leemos el error dice que se no se esperaba el símbolo ‘Debug’, estaría esperando otra cosa.

Con los comentarios es igual. Es un error muy fácil de detectar. Si quitara el cerrar comentarios y guardara se trataría todo como un comentario.

Los paréntesis y las comillas tienen el mismo comportamiento.
Si no cerrara las las comillas y guardo el archivo, tenemos un montón de errores, el importante que es el que está arriba del todo. Nos marca la línea número 13 columna 0 cuando te están diciendo que hay un error en la línea 13 pero en la columna 0 lo más normal es que el error este justamente al final de la línea anterior.

Nos vendríamos a la línea anterior y veríamos que aquí este punto y coma y para empezar naranjas no deberían de estar así y eso es porque se nos ha olvidado cerrar las comillas.

Igual pasa así no las abrimos que se interpreta como las primeras comillas son las que abren las que definen la cadena de texto.

Por ahora no os preocupéis de eso, lo veremos mas adelante en la la parte de tipos de datos.

Otro tipo de error, éste es difícil de darse cuenta: cuando escribimos una letra minúscula cuando tendría que haber una letra mayúscula o a la inversa escribimos una mayúscula.

Para Unity en C#, las palabras van a ser distintas aunque cambie la mayúscula de una letra de una palabra.

CTRL+S-→ Fijaos el error : dice en la línea 12 y entre comillas que pone ‘log’ y eso está diciendo que aquí o alrededor hay algún error

Esta palabra empieza escrita en mayúscula, se debe cambiar a ‘Log’.

Si algo está escrito en mayúsculas apréndete que eso es en mayúscula si está escrito en minúsculas tendrá que estar escrito en minúsculas.

Uno de los errores que más comunes es el siguiente: el error se da cuando se crea un script, se modifica el nombre, no cambia el nombre que hay en el código no es lo suficientemente inteligente como para cambiarlo de forma que ya no encaja en Unity el

nombre que hay después de public class tiene que hacer exactamente igual que el nombre del archivo sin el punto cs.

Ejercicio E04

Para hacer el primer ejercicio de comenta este código para desconectarlo según vimos es quitar la barra asterisco y el asterisco barra luego pulsáis control+s para guardar el script y volvéis a Unity para tener este error todos los ejercicios de este vídeo van a ser así comprobar el error y tenéis que solucionarlo.

Para corregir este error:

“Assets/Scripts/Ejercicios/E04.cs(14,23): error CS0117: `UnityEngine.Debug' does not contain a definition for `log'” nos fijamos en el nombre del script es E04 y luego en la línea la línea número 14.

Volvemos al script, nos fijaríamos en lo que hay entre comillas simples que es log y me daría cuenta de que esta letra está en minúsculas cuando debería de estar en mayúscula así que pulsa control+s y al volver a Unity vemos que el error ha desaparecido.

Pasar al segundo ejercicio ejercicio 2 comenta el bloque anterior y descomenta el siguiente. Obtenemos el siguiente error:
“Assets/Scripts/Ejercicios/E04.cs(26,9): error CS1525: Unexpected symbol `}'”

Lo que haríamos para corregir el error: comprobar en el número de líneas del error que hay más arriba del todo que sería la línea 26, en este caso nos marca el cerrar llave.

Luego también leería que nos dice símbolo no esperado cerrar llave así que está diciendo que esta llave no debería estar aquí porque debería de haber algo antes. A la derecha no hay nada más que pudiera dar error así que vamos a la izquierda y a la izquierda llegamos al principio de línea e iríamos al final de la anterior y me daría cuenta de que aquí falta el punto y coma del final de la instrucción así que pongo el punto y coma.

Teclas “control+s” para guardar vuelvo a Unity y el error desaparece.
Ejercicio número 3. Descomentamos el bloque de código del ejercicio 3, quitando el abrir comentarios y el cerrar comentario. Pulso control+s, vuelvo a Unity y tenemos un chorizo de errores, así que ahora a solucionar estos errores.

Vamos como siempre al error que está más hacia arriba, me fijaría en el nombre del script que es E04.cs y línea línea 35. Me voy al script y me fijo para saber donde va a estar el error, en esa linea o alrededor, la arriba o la de abajo pero va a estar más o menos por por la linea indicada.

Error recibido: “Assets/Scripts/Ejercicios/E04.cs(35,27): error CS1525: Unexpected symbol `<internal>'”

Lo que haría sería analizar escrupulosamente la línea hasta dar con el error.

Debug está bien escrito, Log también. Por algún motivo no se ha puesto el abrir comillas de la cadena, se añade y guarda, vuelve a Unity y no deberíamos de tener ningún error porque lo acabamos de solucionar.

Ejercicio número 4 descomentamos el bloque que hay en este ejercicio. Teclas control+s para guardar y volvemos a Unity.

Error:
“Assets/Scripts/Ejercicios/E04.cs(71,1): error CS8025: Parsing error”

Como solucionaría este error: revisa el nombre del archivo E04.cs y la línea 71. Es la última del archivo, así que recordamos de la teoría que normalmente cuando se marcan al error la última línea es porque nos hemos dejado una llave sin cerrar.

Aquí se está cerrando la llave de aquí así que aquí en el bloque del ejercicio 4 vemos que se está abriendo una llave y no se está cerrando.

Así que la cerramos, pulso control+s vuelvo a Unity y el error ha desaparecido, ya hemos corregido el error

Vamos a pasar al siguiente ejercicio, comentamos el código del ejercicio 4, preparamos el ejercicio 5 pulsa control+s y vuelve a Unity para encontrarnos otra vez con un chorizo de errores. El primero es: “Assets/Scripts/Ejercicios/E04.cs(55,0): error CS1010: Newline in constant”

Como se solucionaría este error: me iría al primero que se muestra en la lista, vería que sería la línea número 55 del script E04.cs, línea 56 columna número 0, nos acordamos que cuando ocurría algo en la columna número 0 nos indicaba que el error seguramente proviene de la línea anterior, del final de la línea anterior.

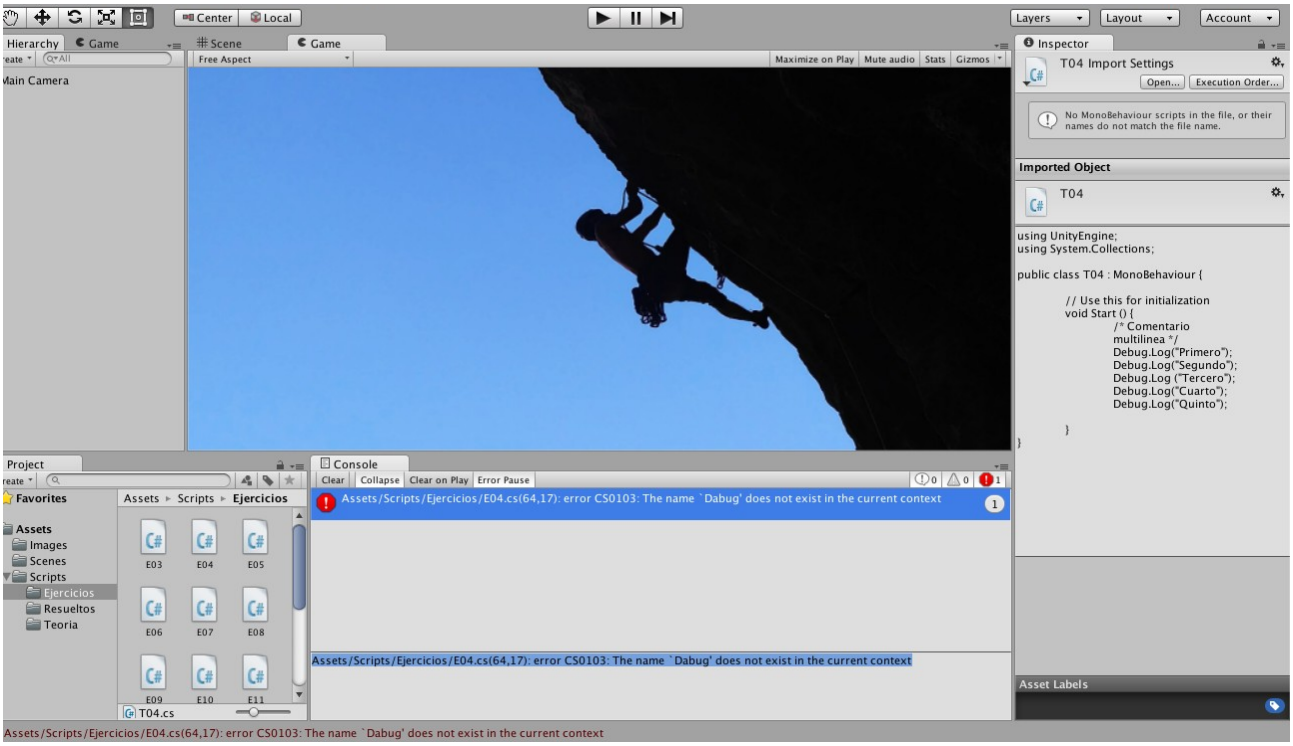
```
47
48
49      // Ejercicio 5. Idem.
50
51
52
53      void Start () {
54          Debug.Log("¡Hola mundo!);
55      }
56
57      |
58
59      // Ejercicio 6. Idem.
60
61      /*
62
63      void Start () {
64          Dabug.Log("¡Hola mundo!")
65      }
66
67      */
68
```

Así que aquí veríamos que nos estamos dejando las comillas sin poner bueno sin cerrar y ya al poner las comillas pues se control+s, se volverá Unity y los errores desaparecen.

Vamos a comentar el ejercicio 5 que ya lo hemos hecho correctamente y descomentamos el ejercicio número 6.

Quito los comentarios pulso control+s y vuelvo a Unity para tener nuevos errores:
“Assets/Scripts/Ejercicios/E04.cs(65,9): error CS1525: Unexpected symbol `}'”
“Assets/Scripts/Ejercicios/E04.cs(71,1): error CS8025: Parsing error”

Así sería como se solucionaría el script E04.cs. Vamos a la línea número 65, aquí aparecería un elemento no esperado:cerrar llave, es decir, que no se esperaba que estuviera un cerrar llave. Eso es porque antes de la llave se ha dejado algo y ya aquí al ver la instrucción se ve claramente que falta el punto y coma.
Pongo el punto y coma, pulso control+s y vuelvo a Unity. Ahora tengo otro error distinto:
“Assets/Scripts/Ejercicios/E04.cs(64,17): error CS0103: The name `Dabug' does not exist in the current context”



En este caso es el del mismo script,línea 65.Me está diciendo que el nombre ‘Dabug’ no existe en el contexto actual. Si nos fijamos entre comillas simples la palabra que hay: ’Dabug’ ya sabemos que está mal escrita. Debería ser:’Debug’,modifica, pulso control+s, vuelvo a Unity y el error ya ha desaparecido por completo.

05 - Variables

Vamos a aprender qué son las variables, qué tipos de datos almacenan, cómo declararlas y cómo ven su contenido mientras depuramos el código paso a paso.

Vamos a preparar el script que vamos a utilizar. Elimina el bloque que no vamos a necesitar.

Las variables las utilizamos para almacenar información en memoria. Qué tipo de información tendríamos en un juego: por ejemplo, usaríamos variables para llevar el número de vidas, el número de puntos, el nombre del jugador, el nivel de energía de un enemigo.

Todo esto lo gestionaremos usando variables. **Para poder utilizar variables antes las tenemos que declarar y para declararlas necesitamos indicar el tipo de dato que va a contener la variable, el nombre de la misma y el valor inicial que va a contener.**

Por partes:
Hay muchos tipos de datos, así que por ahora para no complicarlo mucho nos vamos a centrar en los tipos de datos más comunes.

El primer tipo de datos es el de los números enteros y el tipo se llama '**int**' .Números enteros son los números sin decimales como 1 y 4, el 10, el -5 el -1, el 0, etc …

Luego tenemos el tipo de los números reales, que se llama '**float'.**
Los números reales son los números con decimales, por ejemplo:
3.1415f. Para los literales en forma decimal, el separador de decimales es el punto y hay que añadir una 'f' al final. Mas ejemplos: 0.5f 99.99f,-45.75f, etc ...

Otro tipo de datos son las **cadenas de texto,** el nombre del tipo de dato de cadena de texto es '**string'** y ya hemos utilizado algunas cadenas de texto anteriormente. Como por ejemplo "Hola mundo"

Ya sabemos que el texto se escribe entre comillas dobles.

El último tipo de datos que es el tipo de los valores lógicos
el nombre es ‘**bool’**y puede tener dos valores: **true** que es verdadero o el valor ‘**false’** que es falso.
Sólo puede tener o verdadero o falso

Vamos a ver una serie de ejemplos en donde declaramos variables de cada uno de los tipos distintos.

Ejemplos de declaración de variable:

- La primera de valor entero: una que sea para llevar el total de puntos. El total de puntos será un valor entero.

Se recomienda que utilicemos solo letras del abecedario, que no utilizar en ningún carácter especial. La llamamos **puntos** y ahora la inicializamos con el valor que queremos que tenga al principio que es igual a cero.

- Definimos una variable real con decimales que tenga la última nota que saque en un examen, la llamamos notaExamen. Pues define el tipo **float** y el nombre examen y el valor que quiero que tenga:por ejemplo 9.75f

- Vamos a definir ahora una variable de tipo cadena que contenga el nombre del jugador(n**ombreJugador**) de del supuesto juego que estamos haciendo. Primero pongo el tipo **string,** luego el nombre jugador que va a ser el nombre de la variable y ahora asigno el valor que quiero que tenga esa variable. En este caso es mi nombre. Siempre que no se olvide poner el punto y coma del final de cada instrucción.

- Definir una de tipo lógica,la llamamos **personajeVivo** y voy a marcar como verdadero para indicar que el personaje está todavía jugando dentro del juego. Si estuviera muerto pues aquí pondría falso **false.**

Ya tenemos las variables definidas declaradas inicializadas cada una con su valor. Ahora sí quisiera utilizarlas pues simplemente haría uso del nombre que le haya puesto a cada una.

Por ejemplo si quiero sacarlas por consola que es la única instrucción que sabemos por ahora, haríamos lo siguiente:
Debug.Log y entre paréntesis pondría el nombre de la variable, por ejemplo nombre jugador,no lo penemos entre comillas porque si
no lo que va a salir por consola va a ser la cadena **NombreJugador.**

NO hay que poner las comillas para que Unity sepa que nos estamos refiriendo a la variable que se llama **NombreJugador**.

Entonces cuando en Debug.Log hemos puesto el nombre de esta variable lo que va a sacar por consola va a ser el contenido que tiene esta variable que va a ser el texto indicado en la inicialización.

Voy a aprovechar y sacar el contenido de cada una de las variables que hemos definido:puntos, nota examen y personaje.

Guardar, vuelve a Unity a ver, si está todo bien no tenemos errores y asigno el script a la cámara, compruebo que está asignado, pulso reproducir y veo que sale el contenido de cada
variable por consola.

Vamos a revisar una serie de cosas sobre las cadenas de texto . Definimos una variable de tipo cadena que se llame **mensaje** y voy a poner: “Este es un mensaje especial “ y voy a sacar por consola el contenido de mensaje.

Como estamos usando las comillas para definir o indicar el inicio y el final de una cadena si yo quisiera poner comillas dentro la estaríamos liando porque se interpretará que estas comillas son para indicar el final de la cadena de texto.

Cuando queremos poner comillas dentro de una cadena de textos
se hace poniendo el carácter especial barra y luego comillas **\”** de forma que si quiero que **mensaje** tenga comillas al principio al final sería así :

```
"Este es \\ un mensaje \"especial\"\nEsta es una nueva linea";
```

Guarda, vuelve a Unity y vuelvo a ejecutar para ver qué sale la frase con las comillas dentro .

Más caracteres especiales:

Si queremos que aparezca una barra, no podemos poner una sola tenemos que poner 2 y ahora ejecuto, veremos que sale una.

Otro carácter especial que podemos usar dentro de las cadenas de texto es el ‘\n’. Esta es una nueva línea. Qué significa el ‘\n’: salta de línea y empiezas a escribir en una línea nueva.

Una forma de definir muchas variables de un mismo tipo :se puede hacer para definir variables del mismo tipo en una sola linea. Poniendo el tipo de dato, el nombre de una variable **edadManolo** igual al valor que queremos que tenga, coma y el siguiente nombre de variable **edadBlanca** y su valor, el siguiente nombre **edadLuis** igual 21, ya cuando queramos terminar ponemos punto y coma. Entonces C# entenderá que quiero definir tres variables enteras cada una con su nombre y su valor inicial.

Si defino una variable que se llame **edadManolo,** no puedo declarar una variable con el nombre **edadManolo** porque tendremos un error.

Guarda, voy a ir a Unity para que veáis que Unity se nos queja diciendo una variable local llamada **edadManolo** ya está definida en este ámbito. No podéis utilizar o declarar la misma variable más de una vez. Habría que cambiarle el nombre, ponerle un número de atrás o lo que sea pero serían dos variables completamente distintas.

Vamos a **modificar el contenido de una variable**. Esto se hace
con el operador igual **=**

Como cambiamos? Por ejemplo, los **puntos.** Pues cambiamos el contenido de **puntos** de la siguiente forma: ponemos el nombre de
la variable que queremos cambiar **puntos** igual y ahora el nuevo valor que queramos que tenga por ejemplo 100. A partir de ahora valdrá 200.

Aparece un warning para decir tiene definidas variables y no las está usando pero lo vamos a ignorar. Ejecuta y veremos que sale el valor 200.

Vamos a depurar este código paso a paso para que veáis el contenido de las variables y cómo va cambiando en cada momento

Para iniciar la depuración tenemos que que Unity no está ejecutándo el proyecto y pulsamos el botón de iniciar depuración en nuestro IDE de trabajo (MonoDevelop o Visual Studio). Ejecuta en Unity y vuelve al IDE.

Nos vamos a centrar en estas dos ventanas **Watchs** y **Locals.** Si no las tuviera, se abren desde el menú View/Debug/windows, aquí podemos seleccionar **Watchs** y aparece y ya tendríamos los dos paneles que vamos a necesitar.

Vamos a continuar, ya he iniciado la depuración. Ahora necesito poner el punto de ruptura en la primera línea y hacer que se ejecute el código para que se detenga la ejecución en el punto de ruptura. Se detiene y está a punto de ejecutarse esta línea, voy a ejecutar todas una una hasta aquí donde ya se han definido o se han declarado cada variable con sus valores.

Si quiero ver el contenido de cada variable puedo hacerlo de tres formas:

- La primera es poniendo el ratón encima del nombre de la variable y vemos que **puntos** es vale 0. O encima de **notaExamen** que veo que las notas en 9.75 o desde encima de **nombreJugador** y veo que es cuando aquí es muy útil hacer clic en el pin y situarlo en donde queramos para tener a mano siempre visible en el pin el valor de cada variable.

- Otra forma es usando aquí la ventana local es local nos va a mostrar todas las variables que estén definidas en el punto o en la línea en la que esté de color amarilla es decir el momento concreto en el que estamos ejecutando el código.

Según vaya ejecutando irán apareciendo las variables según se vayan creando las variables y su valor.

- Otra es en W**atchs** deberíamos nosotros de añadir las variables que queramos. Es como **local** pero que aquí nosotros ponemos exactamente las que nosotros necesitamos. Cómo? haciendo clic y poniendo el nombre yo voy a poner **puntos** y vemos la variable y su valor. Voy a añadir **personajeVivo** de otra forma, que es poniendo el ratón sobre la variable **personajeVivo** en caso de que no queramos escribir y aquí hacemos clic con el botón derecho y seleccionamos alt watches y ya nos aparece.

De forma que en locales vemos la lista de variables con sus valores o en los pins que hemos puesto o poniendo en cada momento el ratón encima de la variable o en la lista de watches en donde hemos puesto las variables que queremos observar en cada momento.

De cualquiera de esas tres formas podemos ver cómo van cambiando el contenido de las variables.

Ahora sí voy a ejecutando paso a paso, iremos viendo cómo que en estas líneas cambiará el contenido. La linea que para ejecutarse en cuanto pulse **step over** se ejecutará la línea que está marcada y el código se detendrá en la siguiente.

Son las tres formas que tenemos de ver el contenido de las variables según se va ejecutando el código.

Errores comunes:
Uno de los errores más comunes es intentar meter un tipo de datos (información) dentro de una variable de otro tipo de datos

Por ejemplo meter una cadena de textos en una variable **int.** O meter un bool dentro de una variable de tipo **float.**

Si definimos una variable de tipo **int** tenemos que asignarle un valor que sea entero. Si definimos una variable de tipo **float** tenemos que asignar una variable de tipo real igual con la cadena que hay que asignarle una cadena. Igual con una variable lógica que hay que asignarle o verdadero o falso. Es importante.

Otro error que va relacionado con las variables en concreto con el **ámbito** de acceso a las mismas es el siguiente:
si yo tengo un bloque de código ya sabéis que las llaves agrupan una serie de instrucciones como si fueran una sola.

Cada bloque que engloban dos llaves, los vamos a llamar **ámbitos de variables. S**i yo defino estas variables en un bloque de instrucciones, solo van a existir dentro de estas llaves.

Una vez se cierra una llave todas las variables que se han definido dentro de este bloque desaparecen y la variable no puede ser usada porque no existe.

Resumiendo una vez que se cierra unas llaves todas las variables que se han definido o declarado dentro de esas
llaves desaparecen a partir de aquí la variable a no existiría

Limpiar código de T05 de la cámara en Unity.

06 - Definiendo arrays

Vamos a aprender a trabajar con **arrays.** Para qué vamos a utilizar los arrays: **para guardar en memoria más de un elemento del mismo tipo.**

Queremos tener **en memoria las 10 mejores puntuaciones** del juego que estuviéramos haciendo, podemos definir una variable para cada posición en ese ranking de los 10 mejores puntuaciones.

Necesitaríamos **definir una variable para cada elemento** y luego trabajar con todos los elementos seguidos no sería óptimo.

```
public class T06 : MonoBehaviour {

    // Use this for initialization
    void Start () {

        /*
         * Guardar en memoria las 10 mejores
puntuaciones del juego

        int puntos01 = 0;
        int puntos02 = 0;
        int puntos03 = 0;
        ...
        */
```

Para eso están los arrays, nos **permiten definir muchos elementos en memoria del mismo tipo refiriéndonos a ellos solo como una sola variable**, con un nombre.

Para definir un array:

1) **<tipo_dato_array>**
   Primero, el tipo de datos que queremos que contenga. Para almacenar las diez mejores puntuaciones usamos números enteros, no tendrían decimales.
2) **[]** Abrimos y cerramos corchete pegados al tipo sin dejar espacio dentro de los corchetes.
3) Ponemos el nombre '**puntos**' que es cómo se va a llamar el array
4) **=** Símbolo de asignación
5) **new**
   Especificar qué es un array
6) **[tamaño_array]**
   Entre corchetes la cantidad de datos o de elementos que queremos que contenga

<tipo_dato_array> [] <nombre_array> = new [tamaño_array]

```
int[] puntos = new int[10];
```

Vamos a hacer a declarar un array para cada tipo de dato: una array de números **reales**, otro para **cadenas** y otro para logicos. Cuatro arrays y cada array almacena diez elementos de el tipo que tiene definido.

```
int[] puntos = new int[10];
float[] reales = new float[5];
string[] cadenas = new string[3];
bool[] logicos = new bool[8];
```

Iniciar depuración para ver los valores de los arrays.

Las advertencias solo indican que hemos definido variables que no las estamos utilizando.

Vemos en Monodeveloper o Visual Studio la variable **puntos** que contendrá 10 elementos de tipo entero, **reales** tiene cinco elementos, **cadenas** tiene tres y **logicos** tiene 8.

Aquí no estamos indicando qué valores queremos que contenga el array **logicos**', aparecen los valores de ese tipo (bool), por defecto el valor entero se inicializa con el valor cero, los valores de los reales se inicializan también con el valor cero, las cadenas de textos se inicializan con valor nulo y los valores lógicos inicializan con el valor falso.

Importante: los arrays empiezan a contar el primer elemento en el número 0, no empezamos a contar por el número 1. Con el valor cero indicaríamos que queremos acceder a la primera posición, con el índice 1 a la segunda posición, con el índice 2 a la tercera y así sucesivamente, hasta que, si tiene 10 elementos y queremos acceder al último elemento tendremos que acceder con el índice 9.

Cómo inicializar un array de forma que podamos decir que la posición 1 tenga un valor, que la posición 2 tenga otro valor.. es decir, que podamos especificar con qué valores queremos que se inicialice.

Vamos a ver distintas formas de hacer lo mismo para inicializar los arrays:

```
int[] inicializadoA = new int[5]{1, 2, 3, 4, 5};
int[] inicializadoB = new int[]{1, 2, 3, 4, 5} ;
int[] inicializadoC = {1, 2, 3, 4, 5} ;
```

Inicia depuración para ver los valores añadidos.

Mientras estamos observando las variables, si le hacemos doble clic al valor las podemos cambiar el valor que nosotros queramos mientras depuramos, eso sí al detener la depuración, los valores que hayamos puesto mientras depuramos, se borran.

Array de cinco números **reales** inicializados:

```
float[] realesInicializado = {1f, 2f, 3f, 4f, 5f} ;
```

Con las **cadenas** inicializar entre llaves y ponemos los valores separados por comas.

```
string[] cadenasInicializado = {"Juande", "Jose", "Miguel"} ;
```

Para valores **lógicos** inicializar con verdadero, falso, verdadero, verdadero.

```
bool[] logicosInicializado = {true, false, true, true} ;
```

Para **acceder** a un valor concreto se hace poniendo la variable y justo después entre corchetes el índice al que queramos acceder.

Teniendo en cuenta que el primer elemento es el del índice 0, el segundo el del índice 1,el tercero desde el índice 2 y el último es **el total de elementos menos 1**

Vamos a sacar por consola algún valor :

```
Debug.Log(cadenasInicializado[0]);

cadenasInicializado[1] = "Pepe";

Debug.Log(cadenasInicializado[1]);

Debug.Log(cadenasInicializado.Length);
```

Si quiero acceder a la primera posición: corchete y la primera posición en una array es el valor 0, pongo un 0 y cierro corchetes . Tratamos los arrays como si fueran variables, exactamente igual, lo único que en el número se indica la posición del elemento al que queremos acceder en ese array.

Igual que podemos acceder a los elementos, **podemos modificarlos**. Con el símbolo de asignación igual.
Cambiar el nombre José por Pepe, accede al array, a la posición donde está el elemento que quiero cambiar que es la 1, igual y le añado “Pepe”.

```
cadenasInicializado[1] = "Pepe";
Debug.Log(cadenasInicializado[1]);
```

Inicia depuración y visualiza el cambio del nombre de la segunda posición del array.

Para saber el número de elementos de un array es poner el nombre del mismo, luego un punto y la plabra **length** (longitud)

```
Debug.Log(cadenasInicializado.Length);
```

Recapitulando:
Hemos aprendido a

- Definir arrays de elementos(**guardar en memoria más de un elemento del mismo tipo bajo un mismo nombre**).
- Como acceder a una posición concreta utilizando los corchetes y el índice que indica la posición a la que queremos acceder .
- Modificar sus elementos asignando valores.
- Usar la función **length** para saber cuantos elementos tiene un array

07 - Operadores y expresiones. Parte 1

Operador de asignación "="

Operadores aritméticos: +, -, *, /, %

suma (+), resta (-), multiplicación (*), división (/), módulo (%)(la operación módulo obtiene el resto de la división de un número por otro)

08 - Operadores y expresiones. Parte 2

Operadores lógicos

Operador AND (&&) "Y"

Devolverá true SI y SOLO SI los dos valores que tiene son true.

true && true => true
true && false => false
false && true => false
false && false => false

Operador OR (||) "O"

Devolverá true si alguno de los valores que tiene al lado vale verdadero

true || true => true
true || false => true
false || true => true
false || false => false

Operador NOT (!) "NO"

Invierte el valor lógico

!true => false
!false => true
!!!true => false

Operadores de comparación

== (Es igual a)
!= (Es distinto de)

Los siguientes solo para valores numéricos: int y float

> (Mayor que)
>= (Mayor o igual)
< (Menor que)
<= (Menor o igual)

09 - Operadores y expresiones. Parte 3

Operadores: +=, -=, *=, /=, %=

```
puntos = puntos + 10;
puntos += 10;
(las dos expresiones hacen lo mismo)

puntos = puntos * 10;
puntos *= 10;
(las dos expresiones hacen lo mismo)
```

Operadores: ++, –

```
vidas = vidas - 1;
vidas -= 1;
--vidas;
vidas--;
(en los cuatro casos, vidas tendrá una unidad menos)
```

Operador: ?:

evalúa una expresión y devuelve un resultado dependiendo del valor

CONCEPTOS AVANZADOS POO

FUNCIÓN Random.Range(Parámetro1, Parámetro2)

Esta instrucción, empezando a contar desde Parámetro1, devuelve un número entero aleatorio entre Parámetro1 y Parámetro2-1

Ejemplo: ***int numeroAleatorio = Random.Range(0, 5);***

// Esta linea genera un numero entero aleatorio entre 0 y 4,
//(incluyendo el 0 y el 4) y lo guarda en la variable llamada
// "numeroAleatorio".

01 – Instrucción condicional: IF

Sirve para comprobar el valor de una expresión.

```
  if(Expresión que se resuelve en un valor lógico)
{
 //Instrucción a ejecutar en caso de que la expresión sea true
}
  else
{
 //Instrucción a ejecutar en caso de que la expresión sea false
}
```

02 - Instrucción de selección : SWITCH

Controla múltiples valores de ***expresionAComprobar*** ejecutando las sentencias definidas en el bloque case que cumple el valor de ***expresionAComprobar***. Definición:

```
switch(expresionAComprobar){
        case 0:
                Debug.Log(" Caso 0 ");
                break;
        case 1:
                Debug.Log(" Caso 1 ");
                break;
        default:
                Debug.Log(" Resto de casos ");
                break;
}
```

La instrucción ***break*** hace que el programa ejecute la siguiente instrucción que hay fuera de la instrucción ***switch***.

03 – Estructura repetitiva: WHILE Y DO - WHILE

Sirve para ejecutar bloques de código repetidamente hasta que una expresión especificada se evalúa como **false.**

```
int contador = 1;
while(contador <= 10){
        // Todo el código que queramos que se repita
        // mientras la condición es true.
        Debug.Log(contador);
        contador++;
}
do{
        Debug.Log(contador);
        contador++;
}while(contador <= 10);
```

04 – Estructura repetitiva: FOR

Sirve para ejecutar bloques de código repetidamente hasta que cumpla la expresión que le indicamos para detenerse.

```
for( inicialización ; condición ; Operaciones finales ){
    // Instrucciones del bucle
    }
```

*inicialización: declaración de variable a tratar

*condición : expresión que se evalúa en cada iteración del bucle. Si es **true**: **entra**. Si es **false**: **no** entra.

*operaciones finales: define lo que sucede después de cada iteración del cuerpo del bucle

```
for( int contador = 1; contador <= 10; contador++){
        Debug.Log(contador);
    }
```

05 - Estructura repetitiva FOREACH

Sirve para procesar los elementos en orden creciente de índice, comenzando con el índice 0 y terminando con el índice Length-1.

```
foreach(int valor in valores){
        Debug.Log(valor);
    }
```

06 – BREAK

En una instrucción ***switch***, la instrucción ***break*** hace que el programa ejecute la siguiente instrucción que hay fuera de la instrucción ***switch***. Sin una instrucción ***break***, se ejecutan todas las instrucciones que hay desde la etiqueta case coincidente hasta el final de la instrucción ***switch***, incluida la clausula **default**.

En los **bucles**, la instrucción break finaliza la ejecución de la instrucción envolvente más próxima.

07 – MÉTODOS

Un método es un procedimiento o acción asociado con una clase y se compone de datos y un comportamiento. Esta es la sintaxis común para escribir un método:

```
[modificador de acceso] <tipo de retorno> <nombre>()
{
    [implementación]
}
```

Las partes marcadas con `[]` son opcionales y las que están entre <> son necesarias:

1) **modificador de acceso**, es una palabra reservada que especifica cuales de los otros objetos tendrán acceso al método, puede ser:

   `public`. Todos tienen acceso.
   `protected`. Solo las clases derivadas tienen acceso.
   `internal`. Solo clases contenidas en el mismo ensamblado tienen acceso.
   `private`. Solo la clase que lo define tiene acceso él.

   Además, también podemos añadir el modificador `static` para especificar que un método debe ser llamado desde una instancia de la clase o desde la clase misma.
   Si se omite, el valor por default es `private`.

2) **tipo de retorno**, se usa para declarar si el método retornará algo como parte de su ejecución. Los valores posibles pueden ser muchos, y dependen de la lógica del negocio. Un método puede regresar tipos por valor o referencia, pero solo puede retornar un solo tipo.
   En caso de que no exista necesidad de regresar algún valor, debemos escribir `void`.

3) **nombre**, esta es una parte esencial de la definición de un método. El nombre es un identificador que no debe comenzar por un número ni contener caracteres que no sean alfanuméricos o guiones bajos.
   Por convención, en C# los nombres de método comienzan con mayúscula, aunque tu puedes escribirlos como quieras.

4) **implementación**, es un conjunto de sentencias que definen el comportamiento del método.

08 – MÉTODOS CON PARÁMETROS

Sirven para para poder reutilizar métodos de distintas formas con datos distintos de entrada.

```
[modificador de acceso] <tipo de retorno> <nombre>([parámetros])
{
    [implementación]
}
```

Las partes marcadas con `[]` son opcionales y las que están entre <> son necesarias:

1. **modificador de acceso**, es una palabra reservada que especifica cuales de los otros objetos tendrán acceso al método, puede ser:

   `public`. Todos tienen acceso.
   `protected`. Solo las clases derivadas tienen acceso.
   `internal`. Solo clases contenidas en el mismo ensamblado tienen acceso.
   `private`. Solo la clase que lo define tiene acceso él.

   Además, también podemos añadir el modificador `static` para especificar que un método debe ser llamado desde una instancia de la clase o desde la clase misma.
   Si se omite, el valor por default es `private`.

2. **tipo de retorno**, se usa para declarar si el método retornará algo como parte de su ejecución. Los valores posibles pueden ser muchos, y dependen de la lógica del negocio. Un método puede regresar tipos por valor o por referencia, pero solo puede retornar un solo tipo.
   En caso de que no exista necesidad de regresar algún valor, debemos escribir `void`.

3. **nombre**, esta es una parte esencial de la definición de un método. El nombre es un identificador que no debe comenzar por un número ni contener caracteres que no sean alfanuméricos o guiones bajos.
   Por convención, en C# los nombres de método comienzan con mayúscula, aunque tu puedes escribirlos como quieras.

4. **lista de parámetros**, los métodos pueden recibir información del objeto que los llama a través de los parámetros, la lista de parámetros está definida por el método mediante la especificación del tipo de dato y el identificador de este, si es necesario que el método reciba más de un parámetro estos se separan por una coma.

5. **implementación**, es un conjunto de sentencias que definen el comportamiento del método.