PROCESO DE GESTIÓN DE FORMACIÓN PROFESIONAL INTEGRAL

FORMATO GUÍA DE APRENDIZAJE

IDENTIFICACIÓN DE LA GUIA DE APRENDIZAJE

- **Denominación del Programa de Formación:** Tecnólogo en Análisis y Desarrollo de Software.
- **Código del Programa de Formación:** *228118*.
- **Nombre del Proyecto :** construcción de software integrador de tecnologías orientadas a servicios.
- **Fase del Proyecto :**
- **Actividad de Proyecto:** .
- **Competencia:** *Desarrollar la solución de software de acuerdo con el diseño y metodologías de desarrollo.*
- **Resultados de Aprendizaje Alcanzar:** Crear componentes front-end del software de acuerdo con el diseño.
- **Duración de la Guía:** 40 horas

2. PRESENTACIÓN

3. FORMULACIÓN DE LAS ACTIVIDADES DE APRENDIZAJE

CN-GA1-AA1 – Construir componetes front-end según requerimientos del proyecto, según mapa de Navegacion, user-flow, wireframes, user interface y framework seleccionado.

- Descripción de la(s) Actividad(es):

  El equipo de trabajo basando en los requerimiento del proyecto eligira el flujo de autenticacion, registro, login, gestion de usuarios y la funcionalidad principal identificada para proyectar su MVP aplicando los siguientes procesos:

  - **Mapa de Navegación:** definir los niveles de navegación y secciones que tendrá la aplicación en los flujos elegidos.
  - **User-flow:** después de diseñado el Mapa de Navegacion diseñas los diferentes flujos que los usarios interactúan con la funcionalidad de la aplicación.
  - **Wireframe: se debe usar**
    - Diseño de los Wireframe para el flujo de autenticación, Login y registro y adminstracion de usuarios.
    - Diseño de los Wireframe del Flujo principal de la aplicación.
    - Diseñar un Inventario de los componentes en wireframe, para:



        - Navegaciones
        - Etiquetas
        - Botones
        - Links
        - Elementos de formularios
            - Labels
            - Inputs
        - Selects
        - Toggles
        - Notificaciones
        - Cards
        - Tablas
        - Graficas
        - Migas de pan
        - Steppers
        - Modals
        - Empty state
- **Generar un Moodboard y linea grafica** con referencias que inspiren el diseño de su aplicación.
- **Paleta de color**
    - Primarios: El color primario es el que caracteriza a nuestra interfaz y se identifica con nuestra marca.
    - Secundarios:
    - Terciario
    - Colores neutros
    - Colores de estado
        - Éxito
        - Aviso
        - Error
        - Informacion.

- **Tipografia**
    - Buscar una famila typografia.(usar Google fonts)
    - Definir peso tipografico
    - Escalas tipograficas
    - Definir las carateristicas de: PESO, TAMAÑO, INTERLINEADO y ESPACIADO para:
        - Títulos principales que aparecer sólo una vez por pantalla.
        - Títulos de bloques o módulos secundarios.
        - Títulos de secciones, tablas o cards.
        - Introducciones a secciones principales o dato destacado dentro de una sección.
        - Textos o datos normales y párrafos principales.
        - Textos y datos normales secundarios, información en tooltips y labels de los inputs.
        - Acciones en línea o enlaces muy destacados.
        - Enlaces normales a páginas/secciones externas o internas.
        - Texto de CTAs y botones tanto primarios como secundarios.
        - Estados y etiquetas de estado grandes.
        - Estados y etiquetas de estado pequeñas o secundarias.
    - Colores para las tipografías
        - Primarios
        - Secundarios:
        - Colores neutros
        - Colores de estado
            - Éxito
            - Aviso
            - Error
            - Informacion.
- **Iconografia**
    - Definir la galeria de iconos puden ser de librerias o SVG de uso Libre.
    - Definir los Tamaños.
- **Espacidos**



    - Definir la base de tamaño del espaciado: 4,6,8
  - **Logos, Imagenes, formas e ilustraciones**
  - **Interaction states**
    - **Default**
    - **Hovered**
    - **Focused**
    - **Pressed**
    - **Disabled**
  - **Layouts**
- Ambiente Requerido
  - Ambiente con Equipo de computo.
- Materiales
  - Para el diseño del Mapa de Navegacion usar la herramienta: https://whimsical.com/
  - Para el diseño de los wireframe y los componentes usar la herramienta: https://whimsical.com/
  - Referencias documentos e infografias: https://educated-show-144.notion.site/Roadmap-dise-o-visual-aplicaciones-1-ef68d729aed44c76a7329ae616bfea91?pvs=4
  - Todo los entregables deben se integrados en un Notion.

4. ACTIVIDADES DE EVALUACIÓN

Tome como referencia la técnica e instrumentos de evaluación citados en la guía de Desarrollo Curricular

Evidencias de Aprendizaje	Criterios de Evaluación	Técnicas e Instrumentos de Evaluación
Evidencias de Producto: • Mapa de Navegacion • Users-flows • Wireframe flujo de autenticación y flujo principal		

5. GLOSARIO DE TÉRMINOS

6. REFERENTES BILBIOGRÁFICOS

Construya o cite documentos de apoyo para el desarrollo de la guía, según lo establecido en la guía de desarrollo curricular

7. CONTROL DEL DOCUMENTO

	Nombre	Cargo	Dependencia	Fecha
Autor (es)	**Carlos Enrique Navia Torres**	**Instructor**	**CCyT**	**10/07/2023**

8. CONTROL DE CAMBIOS (diligenciar únicamente si realiza ajustes a la guía)

	Nombre	Cargo	Dependencia	Fecha	Razón del Cambio
Autor (es)					

PROCESO DE GESTIÓN DE FORMACIÓN PROFESIONAL INTEGRAL

FORMATO GUÍA DE APRENDIZAJE

IDENTIFICACIÓN DE LA GUIA DE APRENDIZAJE

- **Denominación del Programa de Formación:** Tecnólogo en producción multimedia.
- **Código del Programa de Formación:** 228101
- **Nombre del Proyecto :** Producción de aplicativos multimediales para el sector productivo.
- **Fase del Proyecto :** Planeación.
- **Competencia:** Diseñar la solución multimedial de acuerdo con el informe de análisis de la información recolectada.
- **Resultados de Aprendizaje Alcanzar:** crear el prototipo de la multimedia siguiendo el patrónestablecido en el guión, el mapa de navegación y el storyboard
- **Duración del Taller:** 27 horas

2. PRESENTACIÓN

Los casos de uso corresponden a una técnica que permite describir el comportamiento de un sistema, cada uno de ellos es una secuencia de interacciones entre un sistema y alguien o algo, que usa cualquiera de sus de sus servicios; se considera que todo sistema tiene un objetivo por cumplir, para lo cual su entorno debe interactuar con personas u otros sistemas (Hardware y Software).

3. FORMULACIÓN DE LAS ACTIVIDADES DE APRENDIZAJE

AA1 Elaborar diagrama Casos de Uso- Conceptualizacion.

Identifica con el equipo de trabajo los conceptos relacionados con los diagramas de caso de uso y mediante un documento relacionarlos teniendo en cuenta los siguientes interrogantes:

- Que es el lenguaje unificado de modelado “UML”
- Cuales son las Características de UML
- Que Tipos de diagramas UML y su uso.
- Que es el diagrama de Casos de uso.
- Que Componentes definen los casos de uso.
- Cual es su representación gráfica

- Que son y que Tipos de relaciones existen en la construccion de los casos de usos.
- Cuales son los parametros para la Documentación de un caso de uso.

El documento debe contener normas APA y la siguiente estructura:

- Presentación (Portada)
- Introducción
- Desarrollo del trabajo
- Conclusiones

Se debe realizar una **presentación en diapositivas** sobre los siguientes puntos, con ejemplos.

- Que Tipos de diagramas UML y su uso.
- Que es el diagrama de Casos de uso.
- Que Componentes definen los casos de uso.
- Cual es su representación gráfica
- Que son y que Tipos de relaciones existen en la construccion de los casos de usos.
- Cuales son los parametros para la Documentación de un caso de uso.

- **EVIDENCIAS DE APRENDIZAJE.**

Evidencia (De Producto)

a. AA1-EV1 - Documento en el procesador de texto(Word) usando las normas Apa 7th edición, donde organice los puntos del proceso de conceptualización de la actividad.
b. AA1-EV2 – Presentación de diapositivas por cada equipo de trabajo.

Evidencia (De desempeño)

c. Sustentación de la presentación, se debe tener la participación de todo el equipo.

AA2 Elaborar diagrama Casos de Uso – Elaboracion caso de uso según kata asignada.

De forma individual cada aprendiz y basado en la asignación de la kata definida por el instructor identificar los siguientes elementos necesarios para diseñar los casos de uso planteados en los requerimientos asignados.

Paso 1

Diligencie la plantilla descrita a continuación para la identificación de los objetos que componen los diagramas.

Nombre de los Actores	Empleado
Nombre de los Casos de Uso	***Gestionar citas***
Nombre del Sistema	Sistema centro medico.
Relación entre los Actores y caso de uso.	**Empleado** <<relacion de asociacion con>> ***Gestionar citas*** ***Gestionar citas <<se incluye>> crear cita***

Paso 2

Investigar y crear un cuadro comparativos de las herramientas online y escritorio especializadas para crear diagramas uml y especialmente para crear diagramas de casos de uso.

Paso 3

Diagramar el caso de uso en la herramienta online https://app.diagrams.net/ según el análisis de la kata asignada.

Paso 4

Describe los caso de uso con la herramienta correspondiente.

- **EVIDENCIAS DE APRENDIZAJE.**

Evidencia (De Producto)

a. AA2-EV1 - Documento en el procesador de texto(Word) usando las normas Apa 7th edición, donde organice los puntos del proceso de conceptualización de la actividad,

este documento debe contener los siguientes puntos: identificación de los componentes para realizar el caso de uso para la kata asignada, cuadro comparativo de herramientas para diagramas uml, Diagrama de caso de uso y las descripciones de los casos de uso identificados.

AA3-Identificar los aspectos importantes de la experiencia de usuario (UX), y los tipos de Interfaces de Usuarios existentes con el fin de garantizar producto multimediales fácilmente usables e inclusivos (6 horas).

Los aprendices de forma individual definirán los siguientes conceptos:

Conceptos de experiencia de usuario (UX).

- ¿Qué es la experiencia de usuario (UX)?
- ¿Qué aplicaciones en el diseño y desarrollo se emplea los procesos de UX?
- En que componentes para productos digitales interviene la experiencia de usuario (UX).
- ¿Cuáles son la Etapas que componen un proceso de experiencia de usuario (UX)?
- ¿Qué beneficios se pueden obtener al incluir procesos UX en tus productos multimedia y web?
- Cuál es la importancia de la evaluación de la experiencia de usuario (UX) en el desarrollo de productos multimedia y web.

Conceptos de creacion de Interfaces de usuario.

- Principios de diseño de las interfaces de usuario, crear una tabla relacinandolos.
- Tipos de Interfaz de usuario
  - Interfaces de lenguaje natural – Presente un Ejemplo.
  - Interfaces de pregunta y respuesta. – Presente un Ejemplo.
  - Interfaces con menús. – Presente un Ejemplo.
  - Interfaces con formularios (entrada/salida). – Presente un Ejemplo.
  - Interfaces de lenguaje de comandos. – Presente un Ejemplo.
  - Interfaces gráficas del usuario (GUI). – Presente un Ejemplo.

AA3-EV1: Documento de conceptualización y apropiación.

EL aprendiz presentará un documento que debe contener normas APA 7th y este debe tener la siguiente estructura:

- *Presentación (Portada)*
- *Introducción*
- *Desarrollo del trabajo*
- *Conclusiones*

AA3-EV2: Infografía.

Se propone que el aprendiz desarrolle una infografía que resuma los conceptos estudiados en la actividad.

- **EVIDENCIAS DE APRENDIZAJE.**

Evidencia (De Producto)

a. AA3-EV1: Documento de conceptualización y apropiación.
b. AA3-EV2: Infografía.
c. AA3-EV3: Cuestionario Técnico AA3.

4. ACTIVIDADES DE EVALUACIÓN

Evidencias de Aprendizaje	Criterios de Evaluación	Técnicas e Instrumentos de Evaluación
Evidencias de Conocimiento : • AA1-EV1 - Documento en el procesador de texto(Word) usando las normas Apa 7th edición, donde organice los puntos del proceso de conceptualización de la actividad. • AA3-EV3: Cuestionario Técnico AA3. • AA3-EV1: Documento de conceptualización y apropiación. **Evidencias de Desempeño** • Sustentación de la presentación, se debe tener la participación de todo el equipo. **Evidencias de Producto:** • AA1-EV2 – Presentación de diapositivas por cada equipo de trabajo.	Crea el arte de los escenarios, fondos, elementos y personajes para la producción multimedia de acuerdo con las especificaciones del guion literario, utilizando herramientas para la edición de imagen.	- Informe escrito - Lista de chequeo - Observacion directa - Lista de chequeo

• AA3-EV2: Infografía.		

5. GLOSARIO DE TÉRMINOS

- **Actividades:** acción requerida para llevar a cabo un proceso.
- **Actor:** se emplea para indicar el tipo de usuario del sistema que podrá ejecutar alguna función.
- **Asociación:** relación que se da entre elementos de los diagramas, por ejemplo entre los actores y el caso de uso o entre las clases.
- **Atributos:** características de una Clase, son datos específicos que interesa guardar de cada entidad.
- **Cardinalidad:** indica la participación que tiene una entidad en la relación.
- **Caso de uso:** indica una función que el sistema debe proveer.
- **Clases:** unidad básica que agrupa una colección de objetos.
- **Entidad:** elemento del sistema de los cuales interesa almacenar información.
- **Extend:** relación que se da entre casos de uso.
- **Métodos:** operaciones de una clase.
- **Relaciones:** también se conoce como asociaciones, sirven para interconectar las entidades.
- **Transición:** es un cambio de estado producido por un evento.
- **UML:** Unified Modeling Language, es un lenguaje estandarizado que se utiliza para visualizar los elementos de un sistema de software, compuesto por diagramas que representan elementos estáticos y dinámicos del sistema.

- **Formularios**: un formulario es un documento, diseñado con el propósito de que el usuario introduzca datos estructurados en las zonas del documento destinadas a ese propósito para ser almacenadas y procesadas posteriormente.
- **GUI**: la interfaz gráfica de usuario, conocida también como GUI (del inglés graphical user interface) es un programa informático que actúa de interfaz de usuario, utilizando un conjunto de imágenes y objetos gráficos para representar la información y acciones disponibles en la interfaz. Su principal uso, consiste en proporcionar un entorno visual sencillo para permitir la comunicación con el sistema operativo de una máquina o computador.
- **Interfaz de usuario**: la interfaz de usuario es el medio con que el usuario puede comunicarse con una máquina, un equipo o una computadora, y comprende todos los puntos de contacto entre el usuario y el equipo. Normalmente suelen ser fáciles de entender y fáciles de accionar.
- **Menús**: es una serie de opciones que el usuario puede elegir para realizar determinada tarea.
- **Prototipos**: un prototipo en software es un modelo del comportamiento del sistema que puede ser usado para entenderlo completamente o ciertos aspectos de él y así clarificar los requerimientos. Un prototipo es una representación de un sistema,

aunque no es un sistema completo, posee las características del sistema final o parte de ellas.

- **Usabilidad**: se refiere a la facilidad con que las personas pueden utilizar una herramienta particular o cualquier otro objeto fabricado por humanos con el fin de alcanzar un objetivo concreto. La usabilidad también puede referirse al estudio de los principios que hay tras la eficacia percibida de un objeto.

6. REFERENTES BILBIOGRÁFICOS

- Craig, Larman. (2007). UML y patrones. Editorial Prentice Hall.
- Martin, F., Scott, K. (1997). UML gota a gota. Pearson Educación.
- Gutiérrez, C. (2011). Casos prácticos de UML. Editorial Complutense. Recuperado de http://site.ebrary.com.bdigital.sena.edu.co/lib/senavirtualsp/reader.action?docID=10536104&ppg=1
- Rumbaugh, J., Jacobson, I. (2007). El lenguaje unificado de modelado: manual de referencia. Addison-Wesley.
- Kimmel, Paul. (2008). Manual de UML Guía de Aprendizaje. Editorial McGraw-Hill Professional Publishing. Recuperado de http://site.ebrary.com.bdigital.sena.edu.co/lib/senavirtualsp/detail.action?docID=10433806&p00=manual+uml
- Teniente López, Ernest. (2003). Especificación de sistemas software en UML. Universidad Politécnica de Catalunya. Recuperado de http://site.ebrary.com.bdigital.sena.edu.co/lib/senavirtualsp/detail.action?docID=11046224&p00=uml
- Kendall, K., Kendall, J. (2005). Análisis y Diseño de Sistemas. México: Pearson Educación.
- Mercovich, E. (2003). Diseño Iterativo de Interfaces. Buenos Aires. Recuperado de http://www. gaiasur.com.ar/infoteca/siggraph99/diseno-iterativo-y-prototipado-rapido.html
- Sommerville, I. (2005). Ingeniería del Software. Madrid: Pearson Educación.

7. CONTROL DEL DOCUMENTO

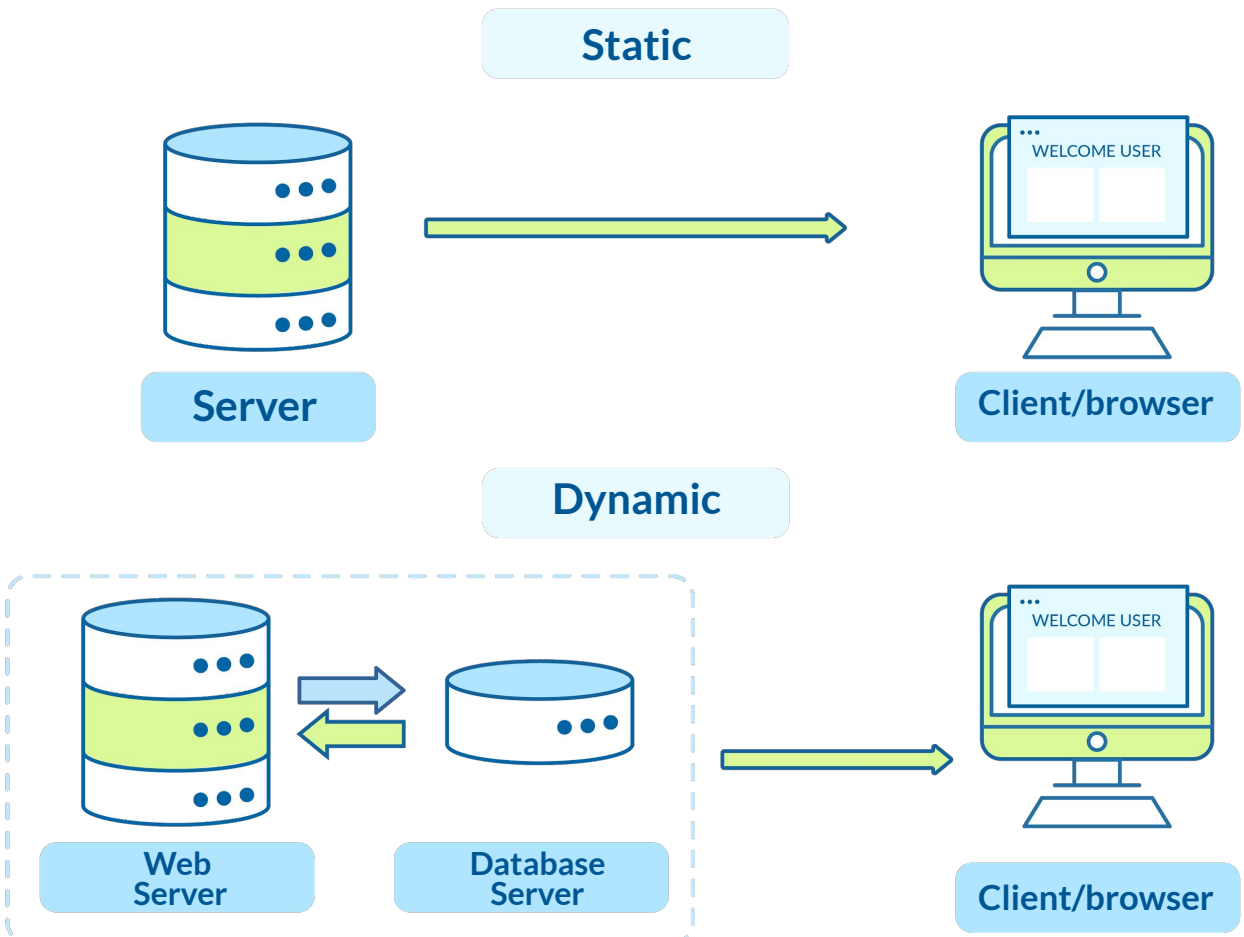
	Nombre	Cargo	Dependencia	Fecha
Autor (es)	Carlos Enrique Navia Torres	Instructor	Centro de Comercio y Turismo, Area contenidos digitales.	10/07/2023

8. CONTROL DE CAMBIOS (diligenciar únicamente si realiza ajustes a la guía)

	Nombre	**Cargo**	**Dependencia**	**Fecha**	**Razón del Cambio**
Autor (es)					

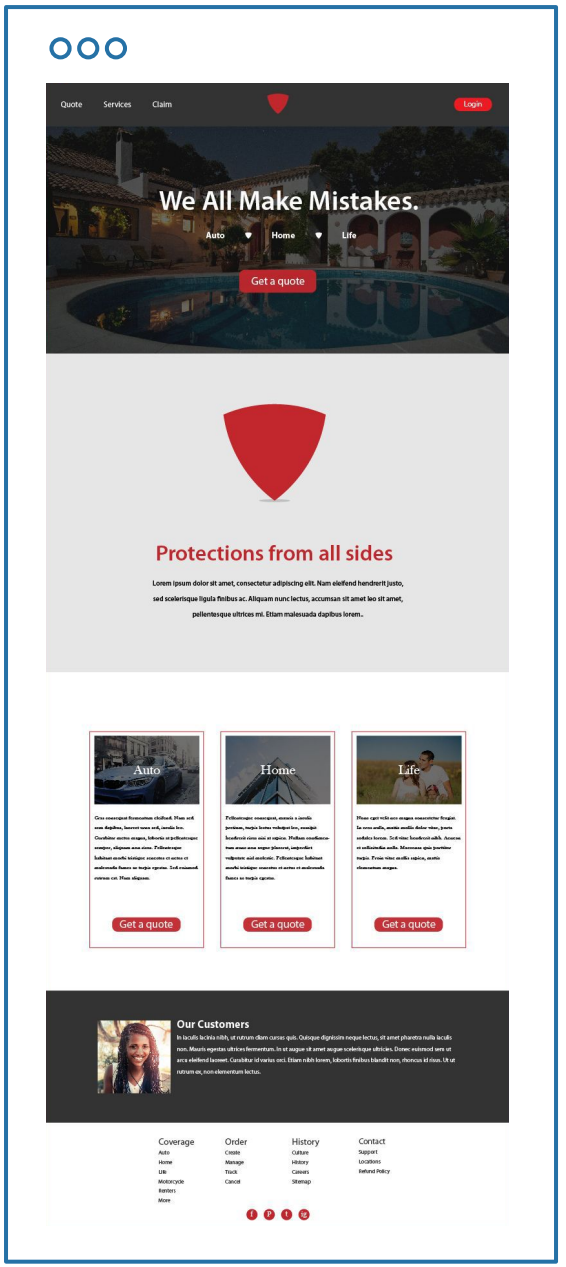
Páginas estáticas vs dinámicas

Páginas estáticas vs dinámicas



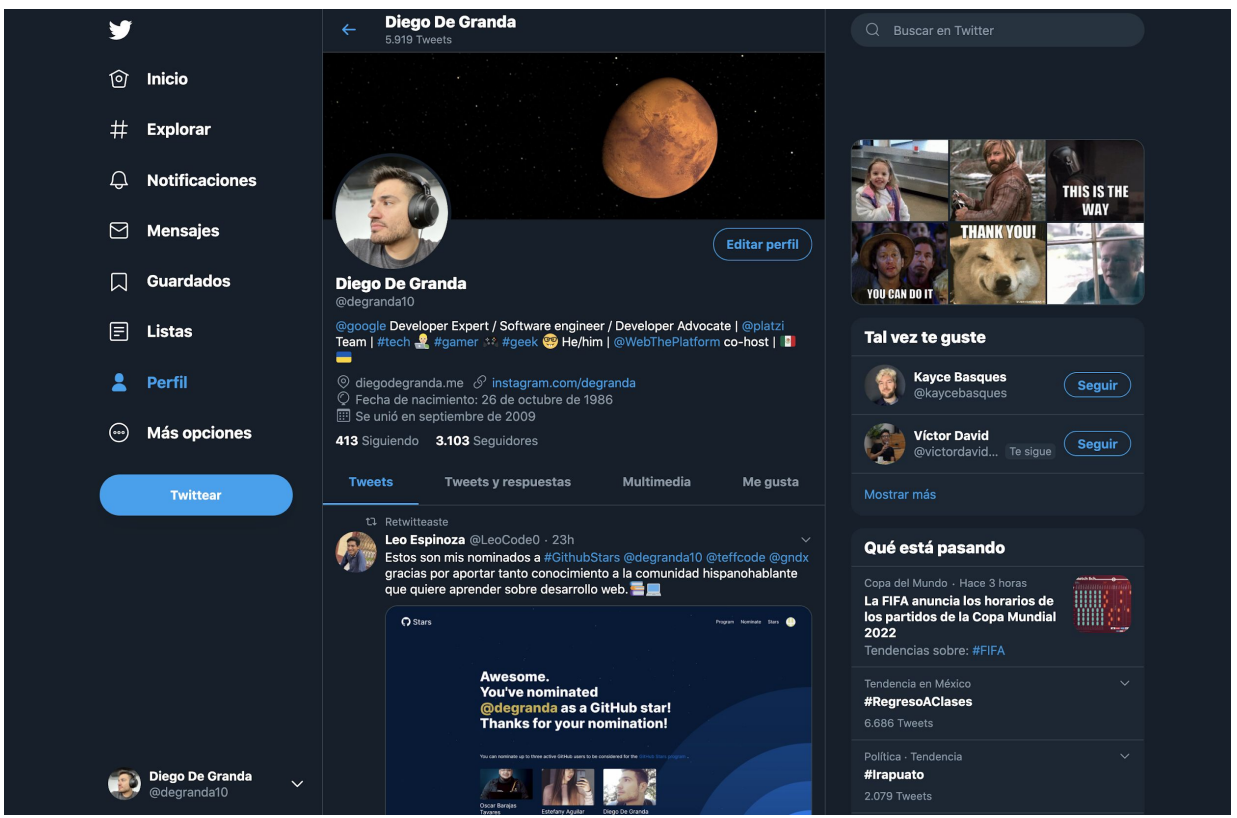
Páginas estáticas

(Landing pages)



Páginas dinámicas

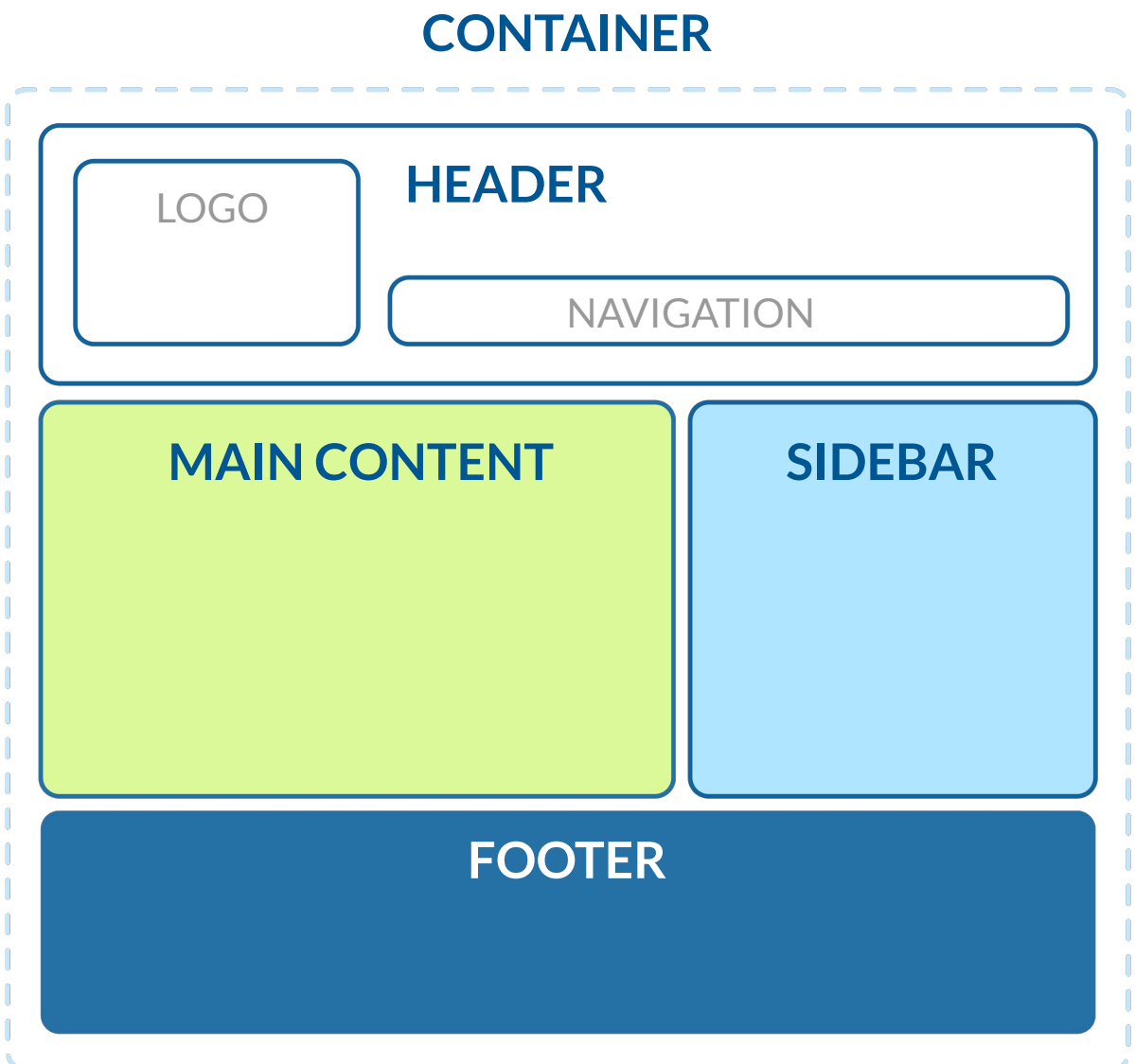
(Web Apps)



HTML

Hypertext Markup Language

Anatomía de una Página Web


CONTAINER
LOGO
HEADER
NAVIGATION
MAIN CONTENT
SIDEBAR
FOOTER

Index y su estructura básica

Anatomía de una etiqueta


Opening Tag
Attribute Name
Attribute Value
Visit DaBrook
Attribute
Element

Optimizando imágenes

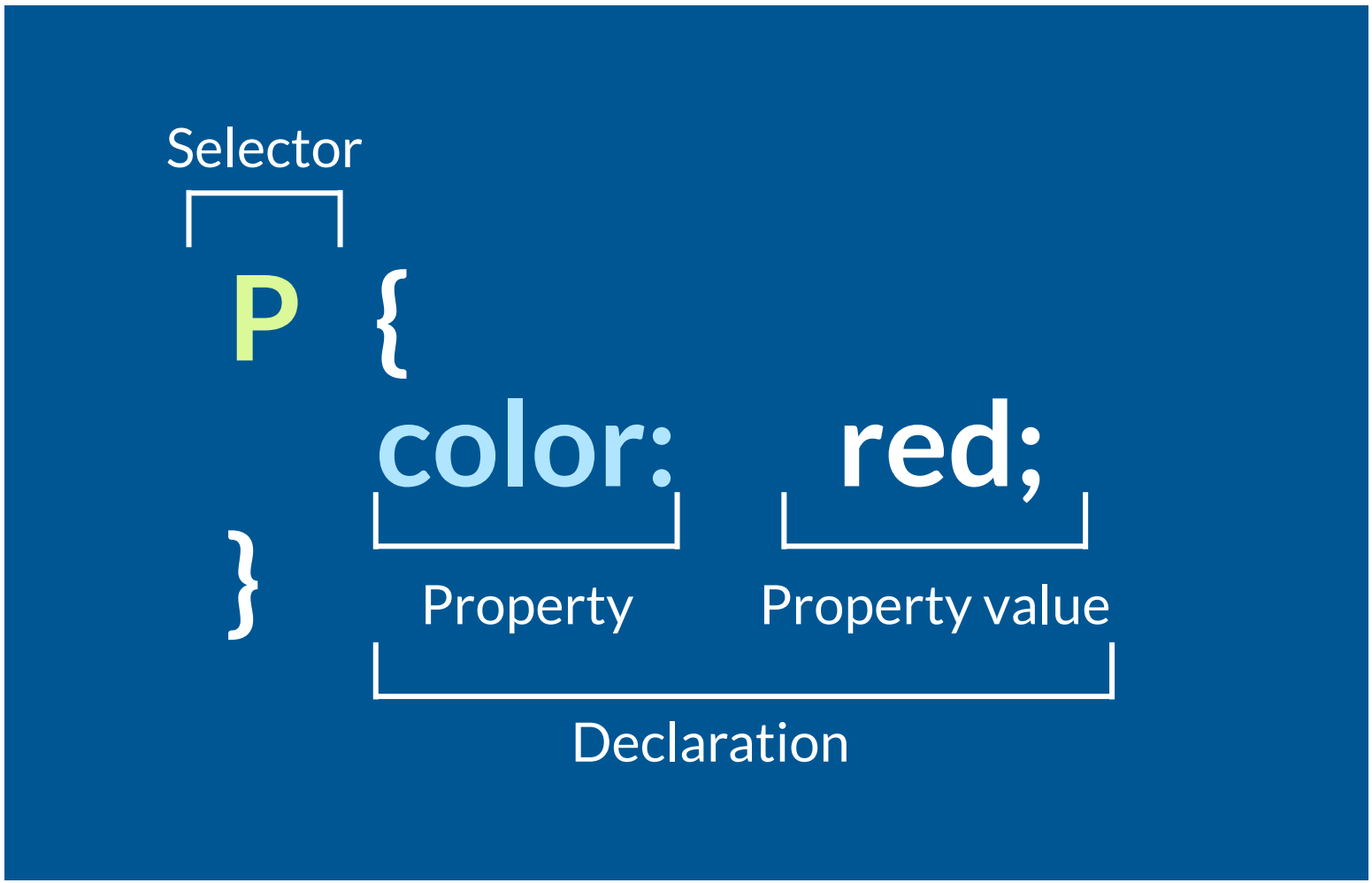
Opciones para trabajar las imágenes

- Mejora el tamaño de tus imágenes
  - Tiny PNG
- Retira metadatos de tus imágenes
  - Verexif

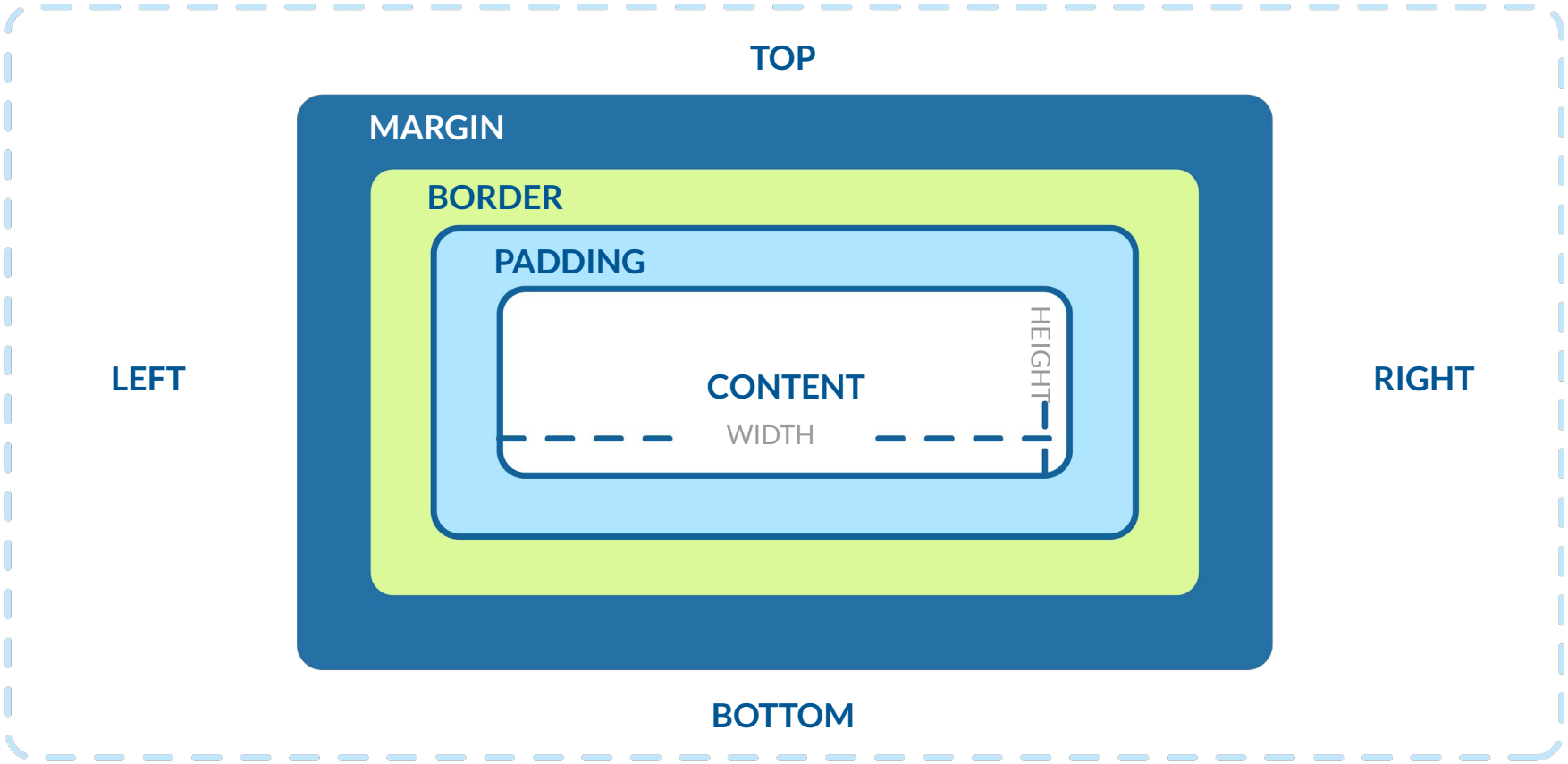
CSS

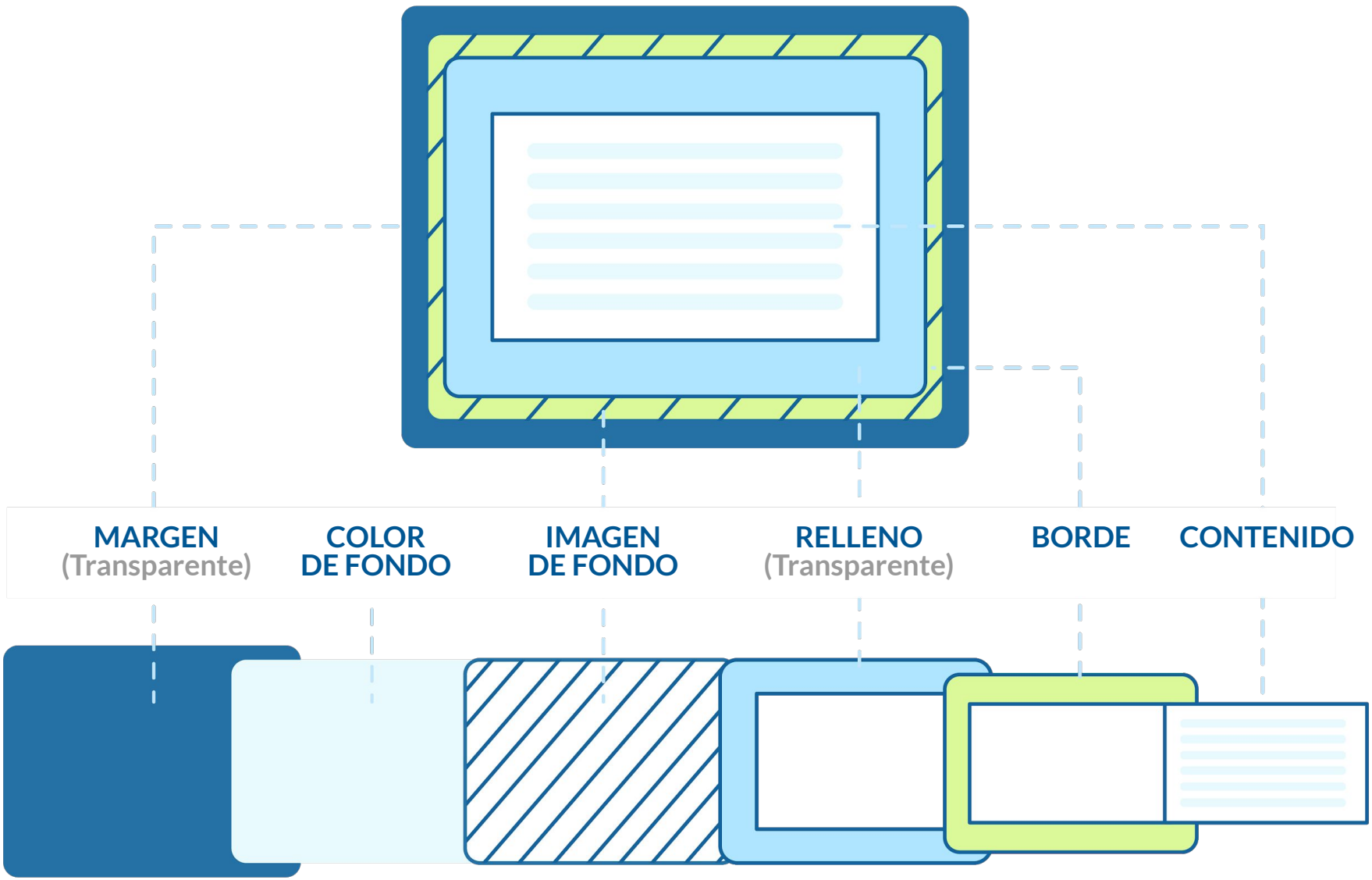
Cascading Style Sheets

Anatomía de una regla en CSS


Selector
P
{
color:
red;
}
Property
Property value
Declaration

Modelo de caja


TOP
MARGIN
BORDER
PADDING
CONTENT
HEIGHT
WIDTH
LEFT
RIGHT
BOTTOM


MARGEN
(Transparente)
COLOR
DE FONDO
IMAGEN
DE FONDO
RELLENO
(Transparente)
BORDE
CONTENIDO

ETIQUETAS HTML ESENCIALES | CHEAT SHEET

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="">
    <title>document</title>
</head>
<body>

</body>
</html>
```

Estructura de una etiqueta HTML

```
<p id="platzi">Esto es HTML</p>
```

Atributo · Valor · Cierre · Apertura · Contenido



Para la base

Etiqueta	Descripción
<!doctype>	Indica la versión de HTML
<html>	Es la raíz del archivo
<head>	Engloba los metadatos
<title>	Título en la pestaña del navegador
<link>	Enlaza archivos CSS
<style>	Escribe CSS dentro del doc HTML
<script>	Enlaza o escribe JavaScript en el doc HTML
<meta>	Características sobre la página
<body>	Elementos visibles en la página

Para el texto

<p>	Crea párrafos
<h1>,<h2>,<h3> <h4>,<h5>,<h6>	Estructura tus títulos y subtítulos, h1 es el más importante de la jerarquía

```
Esto es un h1
Esto es un h2
Esto es un h3
Esto es un h4
Esto es un h5
Esto es un h6
```

<b>	Texto en negrillas
<i>	Texto en cursiva
<u>	Texto subrayado
<a>	Agrega un link
	Texto importante de la página
<blockquote>	Texto en formato de cita
 	Crea un salto de línea

Para multimedia

Etiqueta	Descripción
<img>	Inserta una imagen
<video>	Inserta un video
<iframe>	Inserta contenido de otras páginas
<audio>	Inserta un audio

Para estructurar el contenido

Etiqueta	Descripción
	Crea una lista sin orden
	Crea una lista ordenada
	Define los artículos de las listas

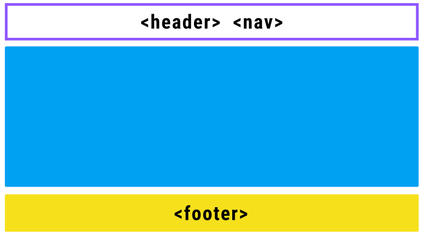
```
<ul>
   <li>Este es el primer item.</li>
   <li>Este es el segundo item.</li>
   <li>Este es el tercer item.</li>
</ul>
```

- Este es el primer item.
- Este es el segundo item.
- Este es el tercer item.

```
<ol>
   <li>Este es el primer item.</li>
   <li>Este es el segundo item.</li>
   <li>Este es el tercer item.</li>
</ol>
```

1. Este es el primer item.
2. Este es el segundo item.
3. Este es el tercer item.

	Secciona o agrupa etiquetas
<nav>	Crea el menú de navegación
<header>	Define la cabecera de la página
<footer>	Define el pie de página



Para crear tablas

Etiqueta	Descripción
<table>	Crea una tabla
<tr>	Crea las filas
<td>	Crea las columnas
<th>	Crea el título de cada columna
<col>	Especifica las propiedades de las columnas

Para crear formularios

Etiqueta	Descripción
<form>	Crea un formulario
<input>	Campo para introducir datos
<label>	El título del input
<textarea>	Campo para introducir mucho texto
<button>	Un botón

CONTACTO

Nombre

E-mail

Mensaje

Enviar Mensaje

Platzi

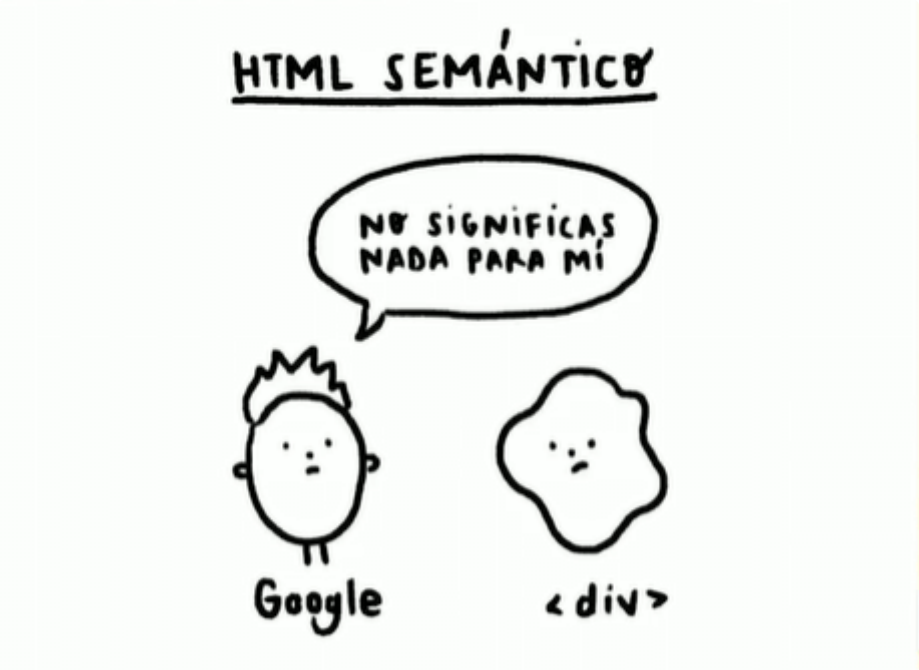
Practica HTML y haz desarrollo web profesional en **platzi.com/htmlpro**

¿Qué es HTML semántico?

El HTML semántico consiste en que cada elemento tenga su propia etiqueta que lo **defina correctamente**. Sin utilizar etiquetas muy generales, como `<div>` o `<span>` .

El problema con la etiqueta *div*

La etiqueta `div` define un **bloque genérico de contenido**, que no tiene ningún valor semántico. Se **utiliza para elementos de diseño** como contenedores.

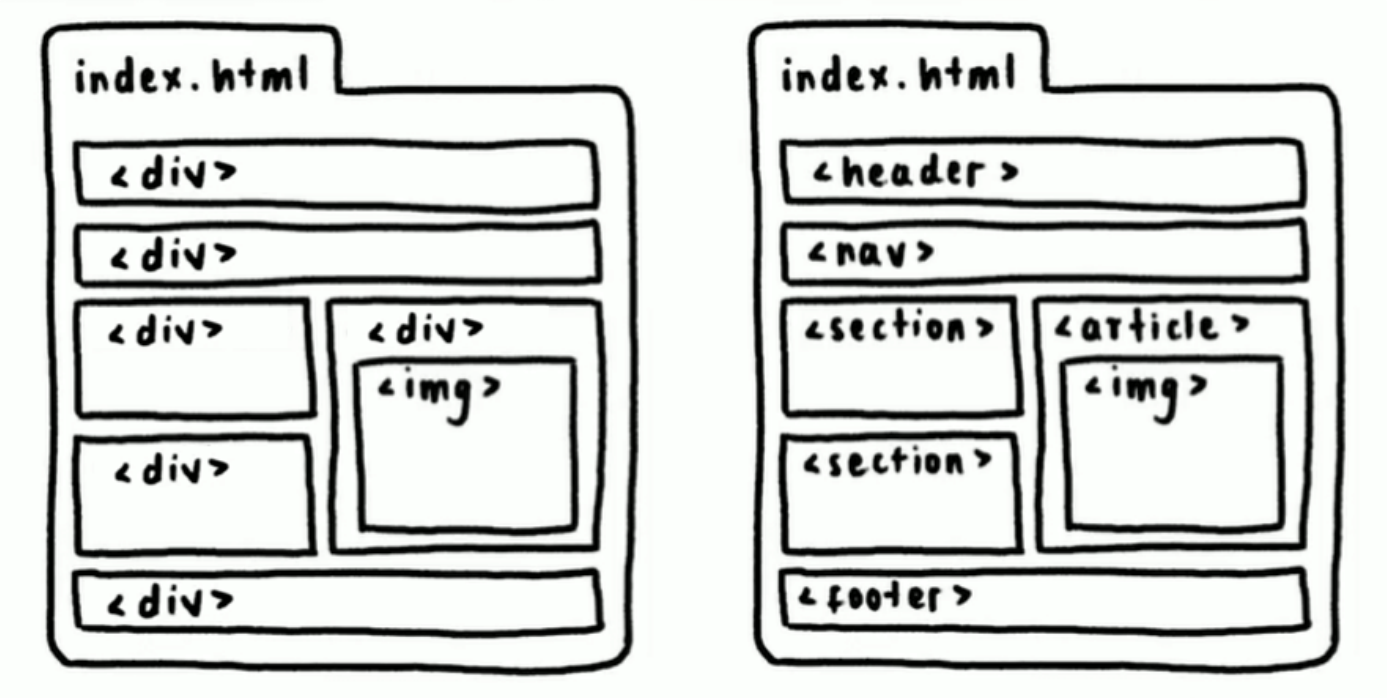


¿Cuáles son las etiquetas semánticas?

Las etiquetas semánticas para definir una interfaz de una página web son:

- `<header>` : define el **encabezado** de la página (no confundir con `<head>`).
- `<nav>` : define una **barra de navegación** que incluye enlaces.
- `<section>` : define una **sección** de la página.
- `<footer>` : define un **pie de página o de sección**.
- `<article>` : define un **artículo**, el cual puede tener su propio encabezado, navegación, sección o pie de página.

Ahora que ya conoces las etiquetas semánticas, evita el uso excesivo de `<div>` .



Ventajas de utilizar HTML semántico

Las ventajas de utilizar un HTML semántico son:

- Ayuda a tu sitio a ser accesible
- Mejora tu posicionamiento (SEO)
- Código más claro, legible y mantenible
- Ayuda a buscadores (como Google) a encontrar tu página

Desafío

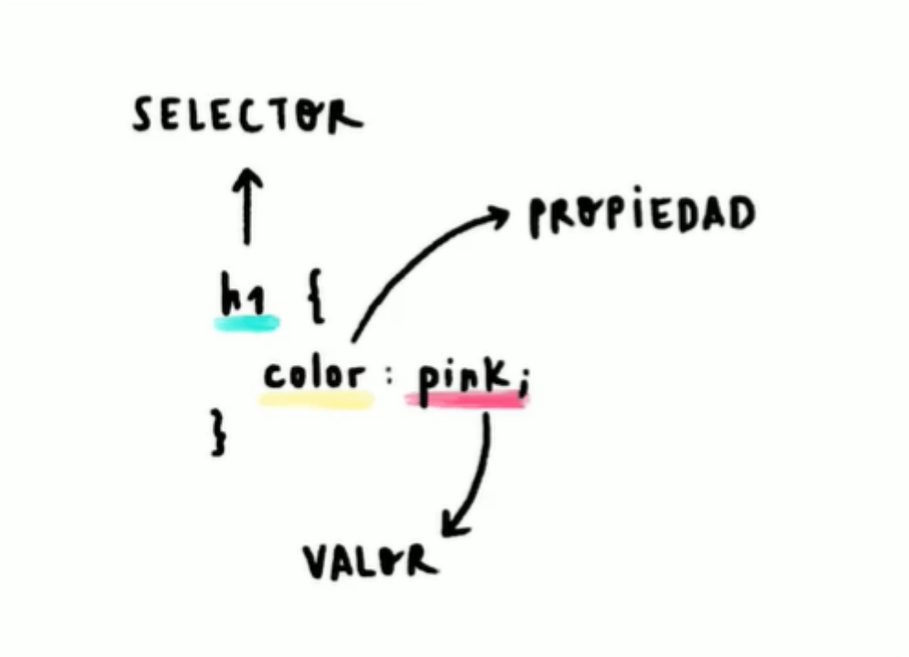
Construye la estructura semántica de un documento HTML.

Antes de empezar a escribir código CSS, debemos conocer la anatomía de una **declaración de estilos**.

Qué es una declaración de CSS

Una declaración de CSS es un bloque que especifica el conjunto de estilos que se añadirán a un elemento HTML. Su estructura contiene lo siguiente:

- **Selector**: define el elemento o conjunto de elementos a los cuales se añadirán los estilos.
- **Propiedad**: es el nombre del estilo de CSS.
- **Valor**: es el valor que tomará la propiedad.



Qué son comentarios de CSS

Los comentarios de CSS consisten en señalar algo que se ignorará. Para establecer un comentario CSS se lo envuelve entre `/*` y `*/`, independiente de la cantidad de líneas.

```
/* Este es un comentario de una línea */
/*
Este es un comentario de varias líneas
*/
```

Propiedades iniciales de CSS

Antes de empezar con CSS utilizaremos algunas propiedades de CSS.

- `color` : establece el color del texto de un elemento.
- `background-color` : establece un color de fondo al elemento.
- `font-size` : establece el tamaño de la fuente.
- `width` : establece la anchura de un elemento.
- `height` : establece la altura de un elemento.

Medidas iniciales

Estas son las medidas iniciales que debes conocer para establecer tamaños de elementos o de tipografía:

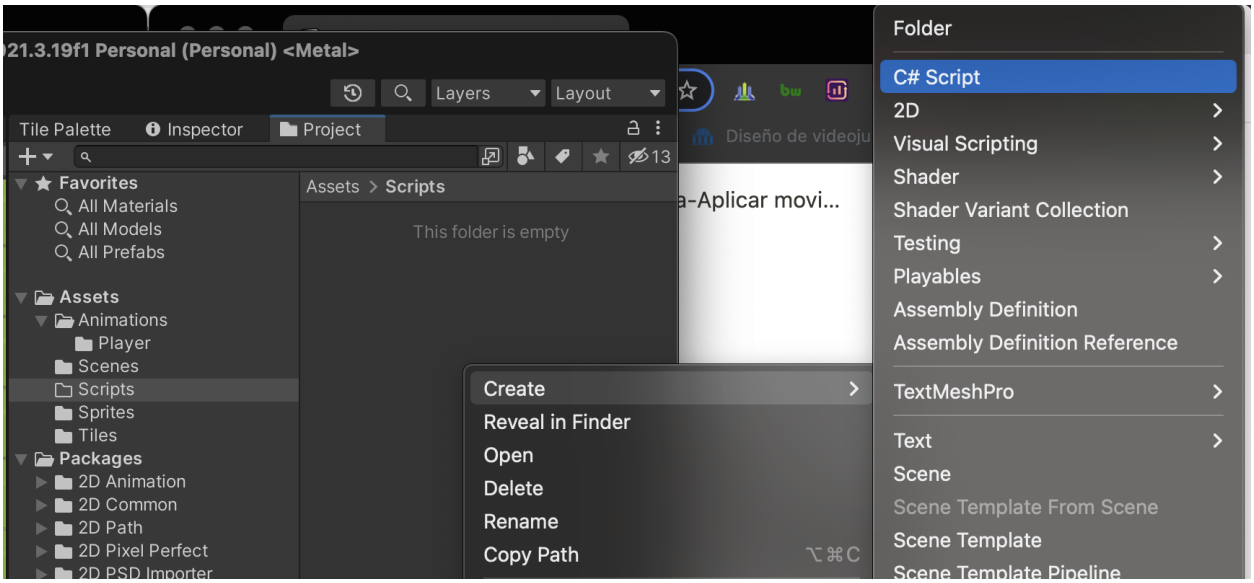
- `px` : establece una longitud de píxeles.
- `%` : establece un porcentaje con respecto a una medida base.

Profundiza más sobre el tema de medidas en:

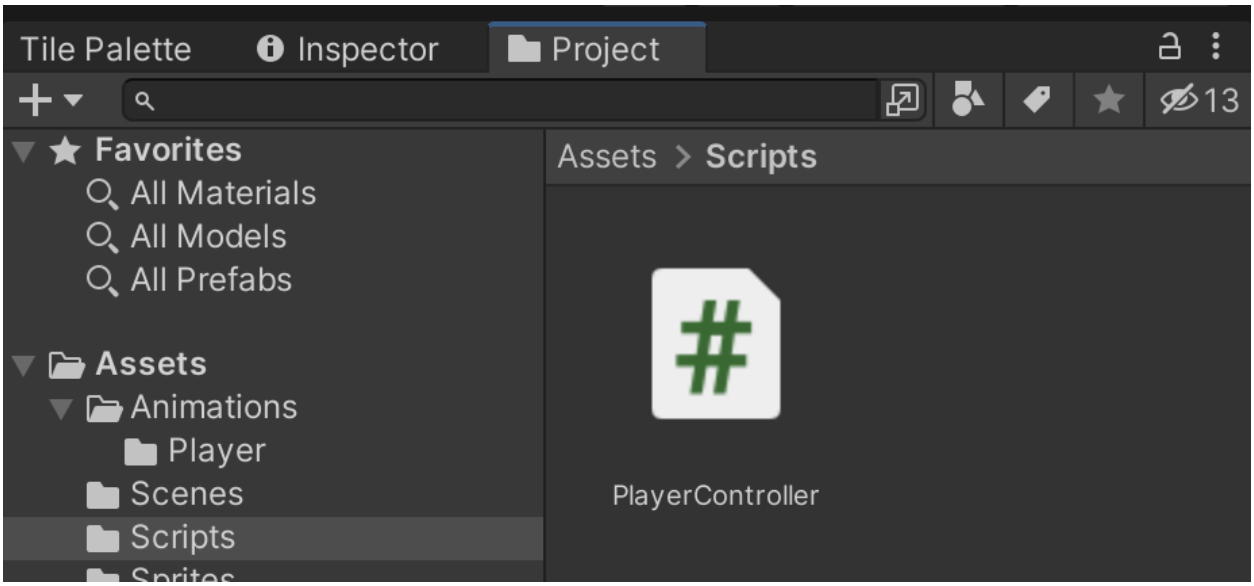
-

Practica-Aplicar movimiento al player

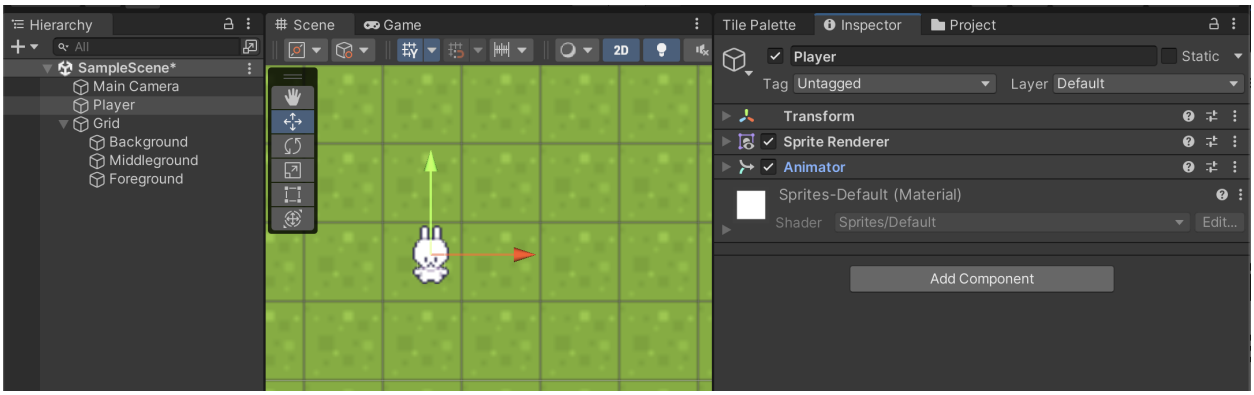
Paso de creacion de Script



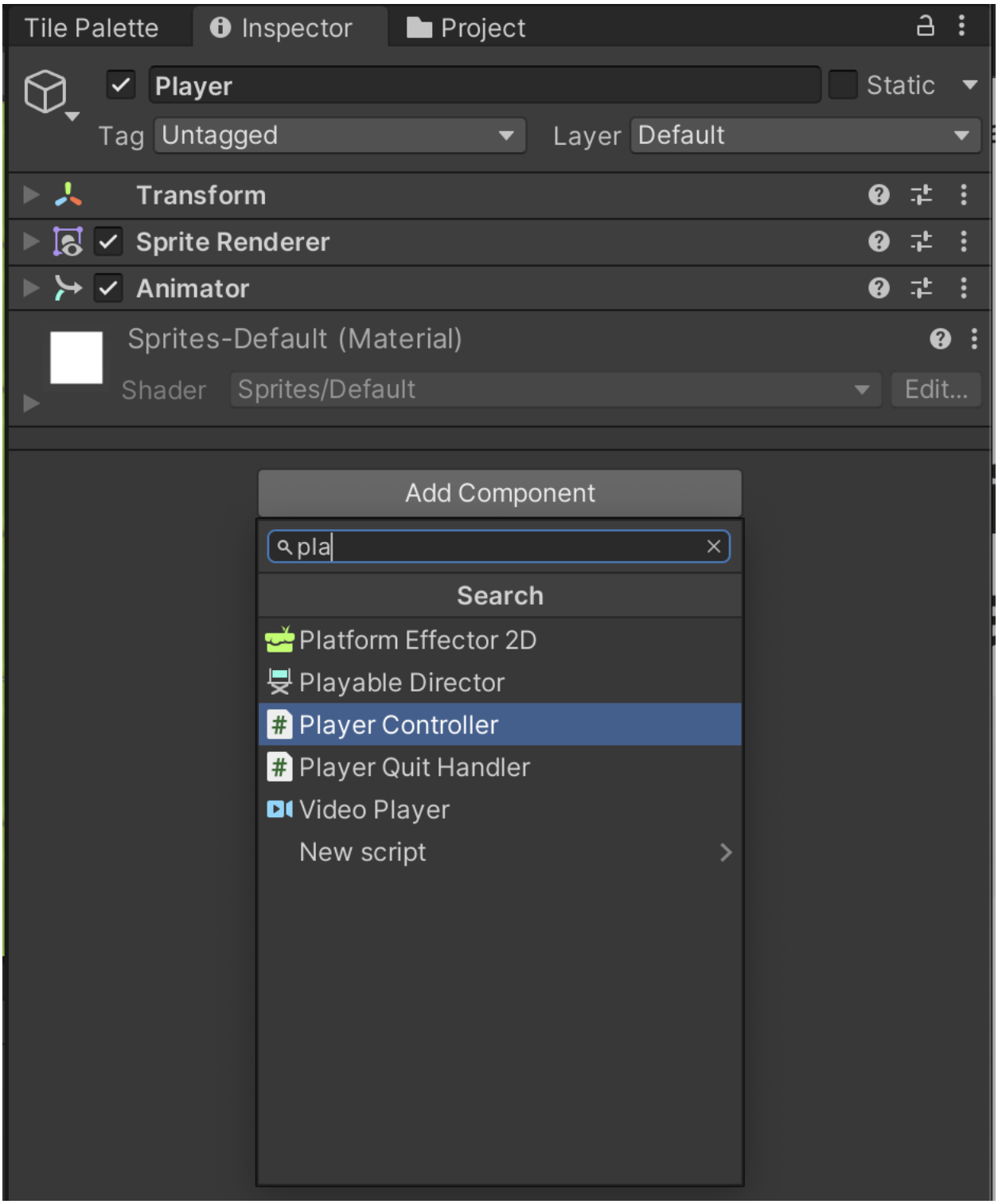
Para crear el Script, un ubicamos en `Assets` > `Scripts` > clic derecho > `Create` > `C# Script`



Nombramos el archivo como `PlayerController`

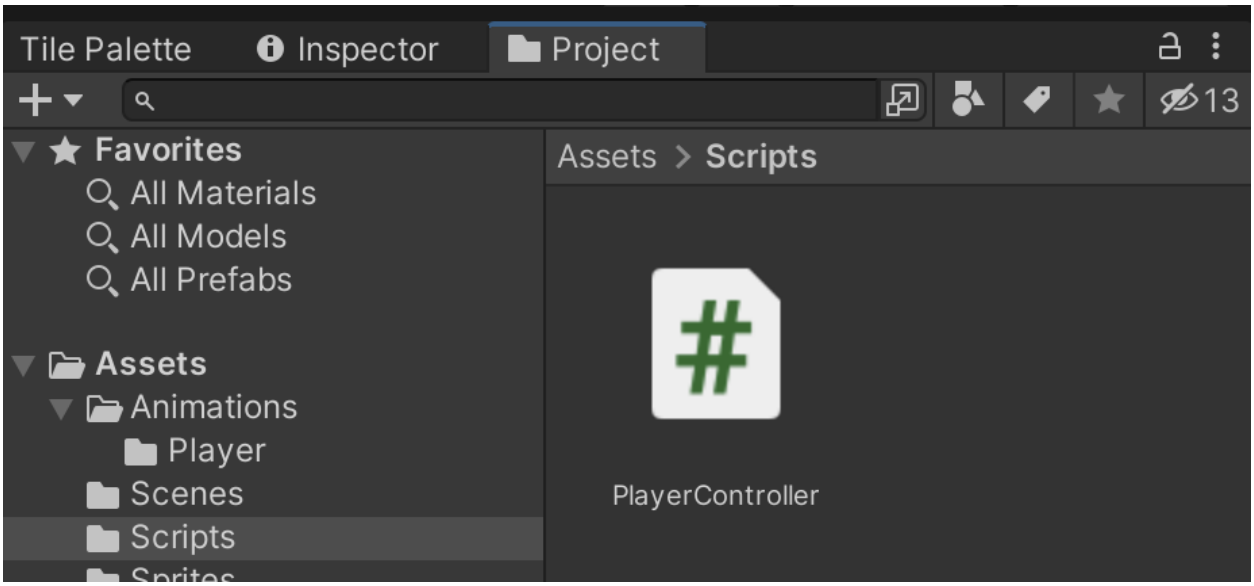


Vamos al panel `Hierarchy` > Clic > `Player` > `Add Component`



Buscamos el script creado > Player Controller > clic

Vamos a Assets > Scripts > doble clic > PlayerController eso abre visual studio.



Explicacion archivo `PlayerController.cs`

Definición de variables y constantes que inicializa la funcionalidad de movimiento.

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

public class PlayerController : MonoBehaviour
{
    public float speed = 4.0f;

    private const string Horizontal = "Horizontal";

    private const string Vertical = "Vertical";

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

    }

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

    }
}
```

Aquí tienes una explicación de cada parte:

1. `public float speed = 4.0f;`: Esta línea declara una variable pública llamada `speed` de tipo `float` y le asigna un valor inicial de 4.0. Esta variable probablemente representa la velocidad a la que se moverá el objeto controlado por el jugador.
2. `private const string Horizontal = "Horizontal";`: Esta línea declara una constante privada llamada `Horizontal` de tipo `string` y le asigna el valor `"Horizontal"`. Esta constante parece ser utilizada como un identificador para la entrada horizontal del jugador (por ejemplo, las teclas de flecha izquierda/derecha o las teclas "A" y "D").
3. `private const string Vertical = "Vertical";`: Similar a la línea anterior, esta declara una constante privada llamada `Vertical` de tipo `string` y le asigna el valor `"Vertical"`. Esta constante podría utilizarse como un identificador para la entrada vertical del jugador (por ejemplo, las teclas de flecha arriba/abajo o las teclas "W" y "S").

Estas constantes se utilizan para proporcionar un enfoque más legible y reutilizable al acceder a las entradas del jugador, en lugar de usar directamente las cadenas "Horizontal" y "Vertical" en el código. Por ejemplo, en el fragmento de código anterior que proporcionaste, la línea `Input.GetAxisRaw(Horizontal)` se refiere a la entrada horizontal del jugador utilizando la constante `Horizontal`.

Explicación código Movimiento Horizontal

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

public class PlayerController : MonoBehaviour
{
    public float speed = 4.0f;

    private const string Horizontal = "Horizontal";

    private const string Vertical = "Vertical";

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

    }

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

```
       //Movimiento horizontal

         if (Mathf.Abs(Input.GetAxisRaw(Horizontal)) > 0.5f)
         {
             this.transform.Translate(new Vector3(Input.GetAxisRaw(Horizontal)
             * speed * Time.deltaTime, 0, 0));
         }
     }
 }
```

El fragmento de código está destinado a mover un objeto en función de la entrada del jugador. Aquí está la explicación paso a paso de lo que hace este código:

1. `Mathf.Abs`: Esto es parte de la clase `Mathf` en Unity, que proporciona varias funciones matemáticas. En este caso, `Abs` devuelve el valor absoluto de un número (su valor positivo), lo cual es útil para considerar la magnitud de una entrada sin importar si es positiva o negativa.
2. `Input.GetAxisRaw(Horizontal)`: `Input` es una clase en Unity que se utiliza para leer las entradas del jugador, como teclas presionadas o movimientos del mouse. `GetAxisRaw` se utiliza para obtener el valor de una entrada del jugador, sin ningún tipo de suavizado. En este caso, `Horizontal` parece ser una cadena que se utiliza como identificador para la entrada horizontal del jugador (por ejemplo, las teclas de flecha izquierda/derecha o las teclas "A" y "D").
3. `> 0.5f`: Esto verifica si el valor absoluto de la entrada horizontal del jugador es mayor que 0.5. En otras palabras, está verificando si el jugador ha movido el joystick o presionado las teclas lo suficientemente lejos como para considerarse como un movimiento significativo.
4. `this.transform.Translate(...)`: Aquí es donde ocurre el movimiento del objeto. `transform` es una referencia al componente Transform del objeto actual al que pertenece este script. `Translate` se utiliza para mover el objeto en un cierto vector de dirección. En este caso, está moviendo el objeto a lo largo del eje x multiplicado por la entrada horizontal del jugador, multiplicado por la velocidad y por el tiempo transcurrido desde el último cuadro.

En resumen, este código se encarga de mover el objeto al que se aplica en la dirección horizontal basándose en la entrada del jugador. Si la entrada horizontal es mayor que 0.5 o menor que -0.5 (lo que indica que el jugador está moviendo el joystick o presionando las teclas de movimiento), el objeto se moverá en esa dirección con una

velocidad proporcionada. Es importante destacar que este fragmento de código debería formar parte de un bucle de actualización, como el método `Update()`, para que el movimiento se aplique constantemente en cada cuadro del juego.

Explicación código Movimiento Vertical

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

public class PlayerController : MonoBehaviour
{
    public float speed = 4.0f;

    private const string Horizontal = "Horizontal";

    private const string Vertical = "Vertical";

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

    }

    // Update is called once per frame
    void Update()
    {
        //Movimiento horizontal

          if (Mathf.Abs(Input.GetAxisRaw(Horizontal)) > 0.5f)
          {
              this.transform.Translate(new Vector3(Input.GetAxisRaw(Horizontal)
              * speed * Time.deltaTime, 0, 0));
          }

        //Movimiento vertical

        if (Mathf.Abs(Input.GetAxisRaw(Vertical)) > 0.5f)
        {
            this.transform.Translate(new Vector3(0, Input.GetAxisRaw(Vertical)
            * speed * Time.deltaTime, 0));
        }
    }
}
```

Este fragmento de código es muy similar al codigo de movimiento Horizonal, pero esta vez está orientado al movimiento vertical del objeto en función de la entrada del

jugador. Aquí está la explicación paso a paso:

1. `Mathf.Abs(Input.GetAxisRaw(Vertical)) > 0.5f`: Al igual que en el código anterior, esta línea verifica si el valor absoluto de la entrada vertical del jugador es mayor que 0.5. Esto determina si el jugador ha movido el joystick o presionado las teclas de movimiento vertical lo suficiente como para considerarlo un movimiento significativo.
2. `this.transform.Translate(...)`: Al igual que antes, aquí es donde ocurre el movimiento del objeto. Esta vez, el objeto se moverá a lo largo del eje y (vertical) en función de la entrada vertical del jugador.
3. `new Vector3(0, Input.GetAxisRaw(Vertical) * speed * Time.deltaTime, 0)`: Esto crea un nuevo vector 3D que define la cantidad en la que el objeto se moverá en cada eje. El objeto se moverá solo en el eje y (vertical), ya que se establece en 0 en los ejes x y z.
    - `0` en el eje x: No hay movimiento horizontal.
    - `Input.GetAxisRaw(Vertical) * speed * Time.deltaTime` en el eje y: El valor de la entrada vertical del jugador se multiplica por la velocidad y el tiempo transcurrido, lo que determina la cantidad de movimiento vertical.
    - `0` en el eje z: No hay movimiento en profundidad.

En resumen, este código permite que el objeto se mueva verticalmente si la entrada vertical del jugador supera un umbral determinado (0.5 en valor absoluto). El objeto se moverá en función de la velocidad proporcionada y el tiempo transcurrido, lo que asegura que el movimiento sea uniforme y no dependa de la velocidad del procesador del sistema. Recuerda que, al igual que en el código anterior, este fragmento también debería estar dentro de un bucle de actualización, como el método `Update()`, para que el movimiento se aplique en cada cuadro del juego.

Código final de la Implementación

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

public class PlayerController : MonoBehaviour
{
```

```
    public float speed = 4.0f;

    private const string Horizontal = "Horizontal";

    private const string Vertical = "Vertical";

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

    }

    // Update is called once per frame
    void Update()
    {
        //Movimiento horizontal

        if (Mathf.Abs(Input.GetAxisRaw(Horizontal)) > 0.5f)
        {
            this.transform.Translate(new Vector3(Input.GetAxisRaw(Horizontal)
            * speed * Time.deltaTime, 0, 0));
        }

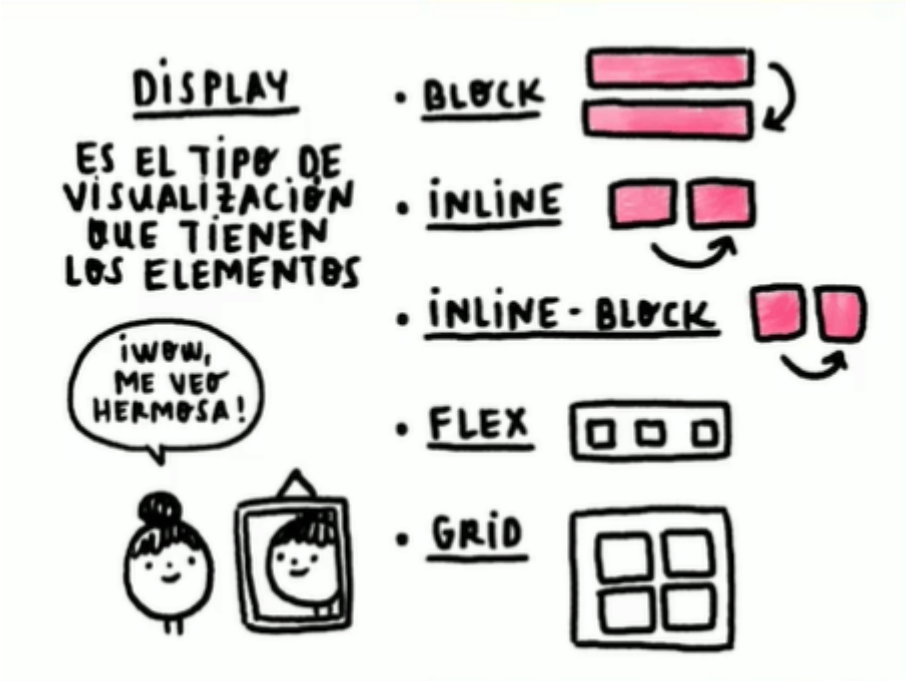
        //Movimiento vertical

        if (Mathf.Abs(Input.GetAxisRaw(Vertical)) > 0.5f)
        {
            this.transform.Translate(new Vector3(0, Input.GetAxisRaw(Vertical)
            * speed * Time.deltaTime, 0));
        }

    }
}
```

Tipos de display más usados: block, inline e inline-block

La propiedad `display` establece el tipo de visualización de los elementos HTML sin afectar el flujo normal de los elementos.



Existen etiquetas que por defecto su display ya está determinado, como la etiqueta `<div>` que tiene display `block`, `<span>` tiene display `inline` y `<button>` tiene display `inline-block`.

Abordaremos los tipos de display `block`, `inline` e `inline-block` a continuación.

Visualización en bloque (`block`)

El display `block` **establece que un elemento ocupará todo el espacio disponible por defecto y el siguiente elemento a este se situará por debajo.

Es posible añadir medidas de anchura `width` y altura `height` a estos a elementos.

También es posible agregar todas las propiedades del modelo de caja (no te preocupes de este concepto, ya lo abordaremos).

- Ejemplo de display block

Visualización en línea (`inline`)

El display `inline` establece que un elemento ocupará el espacio del contenido del mismo y el siguiente elemento se situará a la derecha.

No es posible añadir medidas de anchura `width` y altura `height` a estos a elementos.

También, no es posible agregar todas las propiedades del modelo de caja, únicamente funcionará la propiedad `margin` en el eje horizontal (no te preocupes de este concepto, ya lo abordaremos).

- Ejemplo de display inline

Visualización de bloque y línea (`inline-block`)

El display `inline-block` combina las ventajas de `bloque` de colocar medidas al elemento y propiedades del modelo de caja correctamente; con las ventajas de `inline` de color un elemento seguido de otro en el mismo espacio.

Si elemento excede el contenido total, se coloca en la siguiente línea por debajo.

- Ejemplo de display inline-block

Visualización nula (`none`)

El display `none` desactiva la visualización de un elemento, como si el elemento no existiera.

- Ejemplo de display none

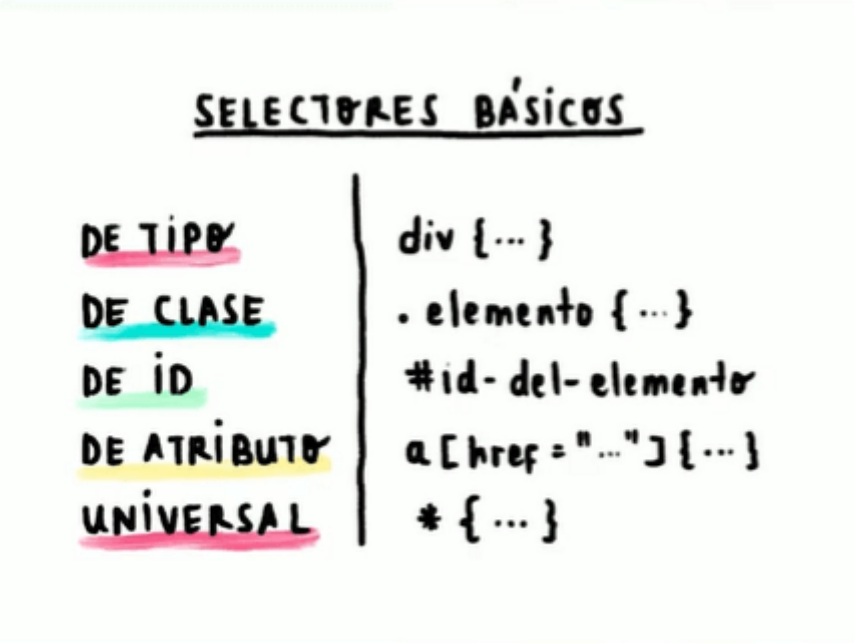
Tipos de selectores: básicos y combinadores

El selector define el elemento o conjunto de elementos HTML a los cuales se añadirán estilos. Existen n ombres de colores propios de CSS que puedes explorar. A continuación veamos más sobre selectores.

Cuáles son los selectores básicos

Un selector básico es la mínima expresión CSS para colocar estilos.

```
selector {
    /* Estilos */
}
```



1. Selector de tipo

Selecciona todos los elementos que coincidan con el **nombre de la etiqueta HTML**.

```
div {
    /* Todos los div en el documento */
}
```

Desafío de selector de tipo

Intenta dar un color de fondo a 10 etiquetas `<div>` con un solo selector, utiliza la `background-color`.

- Desafío selector de tipo
- Solución al desafío

2. Selector de clase

Selecciona todos los elementos que coincidan con las etiquetas HTML que **contengan el atributo** `class`.

```
<!--archivo HTML-->
<div class="card"> Soy una carta </div>
```

Para seleccionar estos elementos, se empieza por un punto `.` y seguido el **valor exacto** del atributo `class` de la etiqueta. Puede ser cualquier valor que desees colocar.

```
/* archivo CSS */
.card {
    /* Todas las etiquetas con la clase "card" */
}
```

Puede existir más de un valor dentro del atributo `class` separados por espacios.

```
<!--archivo HTML-->
<div class="card card1"> Soy una carta </div>
<div class="card card2"> Soy una carta </div>
```

```
.card {
    /* Todas las etiquetas con la clase "card" */
}

.card1 {
    /* Todas las etiquetas con la clase "card1" */
}

.card2 {
    /* Todas las etiquetas con la clase "card2" */
}
```

Desafío de selector de clase

De un conjunto de etiquetas `<div>` intenta dar un color de fondo a las etiquetas que contengan la clase `"card"` con un solo selector. Después, intenta dar un color de letra diferente para las etiquetas que contengan `"card1"` y `"card2"`

- Desafío selector de clase
- Solución al desafío

3. Selector de identificador único (id)

Selecciona el único elemento que coincida con la etiqueta HTML que **contenga el atributo** `id`. Solo puede existir un valor `id` para todo el documento.

```
<!--archivo HTML-->
<button id="eliminar"> Eliminar  </button>
```

Para seleccionar el elemento, se empieza por el símbolo de *hashtag* `#` y seguido el **valor exacto** del atributo `id` de la etiqueta. Puede ser cualquier valor que desees colocar.

```
/* archivo CSS */
#eliminar {
```

```
    /* La única etiqueta con el id "eliminar" */
}
```

Desafío de selector de ID

De un conjunto de etiquetas `<button>`, existe un único botón para eliminar. Intenta colocar un color de fondo rojo a este elemento.

- Desafío selector de ID
- Solución al desafío

4. Selector de atributo

Selecciona los elementos que coincidan con la etiqueta HTML que **contenga el atributo y valor** especificado.

```
<!--archivo HTML-->
<a href="https://platzi.com"> Ir a Platzi </a>
```

Para seleccionar los elementos, se empieza por el nombre de la etiqueta, seguido de corchetes `[]` que contiene el atributo y valor especificado.

```
/* archivo CSS */
a[href=""https://platzi.com"] {
    /* Todas las etiquetas <a> con una propiedad href con valor
"https://platzi.com" */
}
```

Desafío de selector de atributo

Intenta colocar un color de fondo a las etiquetas `<a>` que contengan el atributo `href` con el valor de `"https://platzi.com"`.

- Desafío selector de atributo
- Solución al desafío

5. Selector universal

Selecciona todos los elementos del documento mediante un asterisco `*`.

```
* {
    /* Todos los elementos */
}
```

Desafío de selector universal

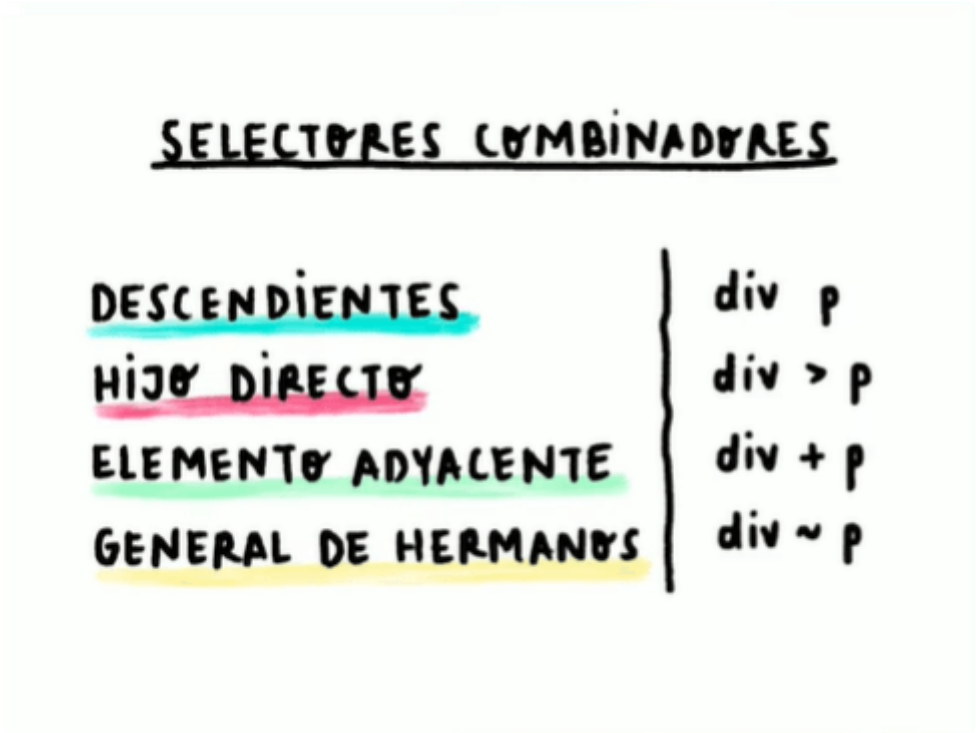
Intenta colocar un color de fondo a todos los elementos del documento.

- Desafío selector universal
- Solución al desafío

Cuáles son los selectores combinadores

Un selector combinador es la unión de dos o más selectores básicos.

```
selector1 selector2 selector3 {
    /* Estilos */
}
```



1. Combinador de descendientes

Selecciona todos los elementos del selector de la derecha que son **hijos** del selector de la izquierda, **independientemente de la profundidad**. Estos selectores están separados por un espacio.

```
padre hijos {
    /* Todos los hijos del padre */
}

div p{
    /* Todos los hijos <p> de <div>*/
}

.container img{
    /* Todos los hijos <img> de la clase "container"*/
}
```

Desafío del combinador de descendientes

Intenta colocar un color de letra a todas las etiquetas `<li>` que son hijos de la clase `"container"`.

- Desafío combinador de descendientes

- Solución al desafío

2. Combinador de hijo directo

Selecciona todos los elementos del selector de la derecha que son **hijos directos** del selector de la izquierda. Estos selectores están separados por `>`.

```
padre > hijos_directos {
    /* Todos los hijos directos del padre */
}

div > p{
    /* Todos los hijos directos <p> de <div>*/
}

.container > img{
    /* Todos los hijos directos <img> de la clase "container"*/
}
```

Desafío de combinador de hijo directo

Intenta colocar un color de letra a todos las etiquetas `<p>` que son hijos directos de la clase `"container"`.

- Desafío combinador de hijo directo

- Solución al desafío

3. Combinador de elemento adyacente

Selecciona todos los elementos del selector de la derecha que están **adyacente** al selector de la izquierda. Estos selectores están separados por `+`.

```
elemento + adyacente {
    /* Elementos adyacentes */
}

div + p{
    /* Todos los <p> adyacentes a <div>*/
}
```

```
.container + img{
    /* Todos los <img> adyacentes a la clase "container"*/
}
```

Adyacente significa que comparten el mismo padre y está situado inmediatamente hacia abajo de otro elemento. Por ejemplo, en el siguiente código, `<div>` está adyacente a `<h1>` y `<p>` está adyacente a `<div>`. Sin embargo, `<h1>` no está adyacente a `<div>` y `<div>` no está adyacente a `<p>`.

```
<!--archivo HTML -->
<h1>Soy un título </h1>
<div>Soy un div</div>
<p>Soy un párrafo</p>
```

Desafío de combinador de elemento adyacente

Intenta colocar un color de letra a todos las etiquetas `<p>` que están adyacente a las etiquetas `<div>`.

- Desafío combinador de elemento adyacente
- Solución al desafío

4. Combinador general de hermanos

Selecciona todos los elementos del selector de la derecha que son **hermanos** del selector de la izquierda. Estos selectores están separados por `~`.

```
elemento ~ hermanos {
    /* Elementos hermanos */
}

div ~ p{
    /* Todos los <p> hermanos de <div>*/
}

.container ~ img{
```

```
    /* Todos los <img> hermanos de la clase "container"*/
}
```

Hermanos significa que comparten el mismo padre y están situados hacia abajo de otro elemento. Por ejemplo, en el siguiente código, `<div>` y `<p>` son hermanos de `<h1>`, pero `<h1>` no es hermano de `<div>` y `<p>`.

```
<!--archivo HTML -->
<h1>Soy un título </h1>
<div>Soy un div</div>
<p>Soy un párrafo</p>
```

Desafío de combinador general de hermanos

Intenta colocar un color de letra a todos las etiquetas `<p>` que son hermanos de `<div>`.

- Desafío combinador general de hermanos
- Solución al desafío

Tipos de selectores: pseudoclases y pseudoelementos

Existen otros tipos de selectores, además de los selectores básicos y combinadores, capaces de cambiar un estado o añadir algo más al elemento. Estos son denominados pseudoclases y pseudoelementos.

Cuáles son las pseudoclases

Una pseudoclase define el estilo de **un estado** especial de un elemento.

- Índice de pseudo-clases estándar.

Sintaxis

```
selector : pseudoclase {
    propiedad: valor;
}
```

:hover

Representa el estado en el cual **el cursor está encima del elemento**.

- Ejemplo

:active

Representa el estado de un elemento que **no ha sido visitado**.

- Ejemplo

:visited

Representa el estado de un elemento que **ya ha sido visitado**.

- Ejemplo

:not()

Representa el estado en el cual **no coinciden los selectores que se indiquen**.

- Ejemplo

:nth-child()

Representa el estado en el cual **coinciden los hijos del elemento según el valor indicado**.

Valores de palabras clave:

- **odd:** los elementos hijos en posiciones impares.
- **even:** los elementos hijos en posiciones pares.

Fórmula matemática: `An+B` donde A y B son números enteros.

- Ejemplo

Cuáles son los pseudoelementos

Un pseudoelemento define el estilo de **una parte específica** de un elemento.

- Lista de pseudo-elementos.

Sintaxis

```
selector :: pseudo-elemento {
    propiedad: valor;
}
```

::before

Sirve para agregar un contenido **antes del elemento**. El contenido es agregado mediante la propiedad `content` de CSS.

- Ejemplo

::after

Sirve para agregar un contenido **después del elemento**. El contenido es agregado mediante la propiedad `content` de CSS.

- Ejemplo

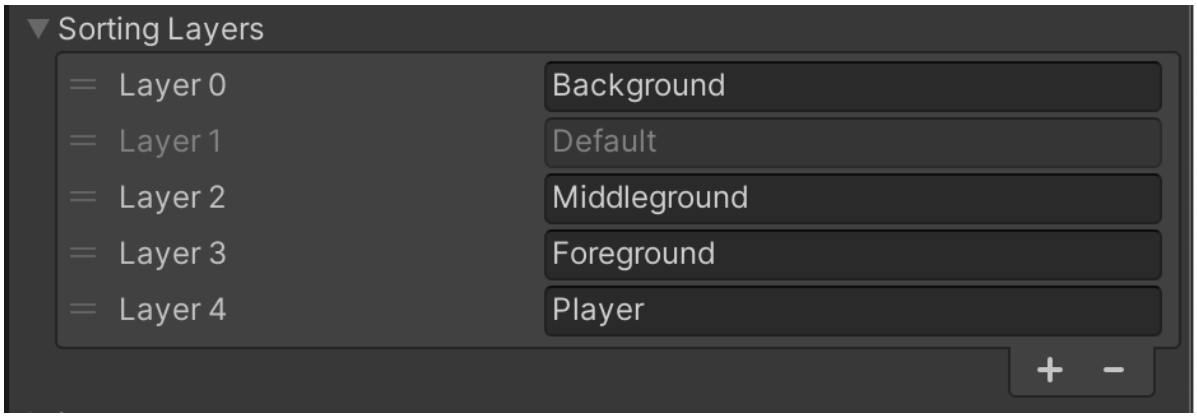
::first-letter

Sirve para añadir estilos a a la **primera letra del texto** de cualquier elemento.

- Ejemplo

Trabajo diseño de escenarios RPG2D

1. Crear 3 Tilemap en Unity, usando los recursos compartidos en la sesión anterior distribuidos en de la siguiente forma:
    a. Forebackground
    b. Middlebackground
    c. Background.
2. Crear 4 Sorting Layers que se llaman y se ordenan de la siguiente forma.

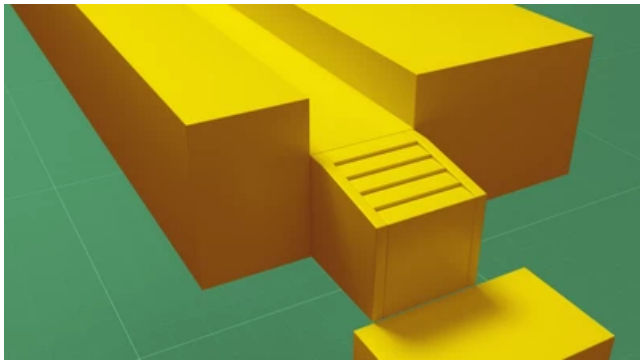


3. Ahora debe Asignar las Sorting Layer a cada GameObject (Tilemap y player) correspondiente.
4. Dibujar en cada tilemap lo siguiente:
    a. Forebackground: todo lo que colisione con el player
    b. Middlebackground: cosa sobre las que el player puede pasar y pisa como plantas, musgo etc.
    c. Background. Tierreno

💡 **Nota:** el player y demás elementos usados deben ser los entregado anteriormente. No se admite recursos diferentes, eso hace que pierda la práctica.

Aspectos básicos de la programación en Unity

Unidad 3 Planes de lecciones



3.4 Guía de estilos y buenas prácticas con C#.

PASO 1. DESCRIPCIÓN 3
PASO 2. BUENAS PRÁCTICAS EN NOMBRADO DE ARCHIVOS. 3
PASO 3. BUENAS PRÁCTICAS EN LA INDENTACIÓN DEL CÓDIGO. 3
PASO 4. BUENAS PRÁCTICAS EN ESPACIADO Y LÍNEAS EN BLANCO 4
PASO 5. CONVENCIÓN DE NOMBRES 6
PASO 6. COMENTARIOS 8
PASO 7. OTRAS RECOMENDACIONES 9

Duración

Resumen

En este tutorial, podrás hacer lo siguiente:

→ Identificar las buenas practica de creación archivos e indentación de código.
→ buenas prácticas y convenciones de nombres para las estructuras de programación.
→ Como escribir mensajes que permitan documentar el código mediante comentarios.

Habilidades

Comprensión de las reglas y buenas prácticas de escritura de código

→ Interpretar el código simple

Mejorar el código simple con las características de un IDE

Paso 1.Descripción.

El objetivo de este punto es ver una serie de reglas que nos ayuden a escribir buen código en el lenguaje C# y siguiendo buenas prácticas.

Un buen código es aquel que:

- Cumple con su función de forma eficiente.
- Está organizado y es fácil de leer.
- Es fácil detectar errores en él, y por tanto, es fácil de mantener.
-

Tener un buen código siguiendo buenas prácticas va a permitir que trabajemos de una forma más eficaz y ágil, ya que:

- Podremos revisar y entender fácilmente nuestro propio código escrito tiempo atrás.
- Podremos integrar nuestra forma de trabajar con la de un equipo de programación, permitiendo que otros programadores lean y entiendan nuestro código de forma sencilla.

Es posible que hayamos comentado ya alguna de las buenas prácticas que vamos a nombrar a continuación, pero hemos decidido recopilar todas juntas para tener un sitio donde consultar de forma global.

Paso 2. Buenas prácticas en nombrado de archivos.

Por norma general un fichero debe contener una única clase. Hay que crear el fichero de código fuente con el nombre de la clase como nombre del fichero, más la extensión correspondiente, en nuestro caso .cs.

Ejemplo: **Player.cs**

Paso 3.Buenas prácticas en la indentación del código.

- Se utiliza el tabulador para indentar el código, nunca hacerlo con espacios.
- Las líneas de código no deberían exceder de los 80 caracteres. Las líneas largas deben ser separadas en varias líneas, de modo que su lectura sea más fácil.

```
float price = 15.5f;
int units = 5;

Console.WriteLine("El precio es {0} y el número de " +
    "unidades es {1}", price, units);
```

Paso 4. Buenas prácticas en espaciado y líneas en blanco

Espaciado

- Debe de haber un espacio después de una coma o punto y coma.

```
Test(a, b, c)       // Ok
Test(a,b,c)         // No
Test( a, b, c )     // No
```

- Debe de haber un espacio rodeando los operadores (excepto los unarios)

```
a = b   // Ok
a=b     // No
```

```
for (int i = 0; i < 10; i++)    // Ok
{
    //  código
}

for (int i=0; i<10; i++)        // No
{
    //  código
}

for (int i=0;i<10;i++)          // No
{
    //  código
}
```

Líneas en blanco

Las líneas en blanco mejoran la legibilidad del código al separar bloques de código con líneas relacionadas.

Se deberían introducir líneas en blanco entre:

- Dos clases dentro de un mismo fichero. (Aunque hemos visto anteriormente que cada clase debería ir incluída en su propio fichero).
- Métodos.
- Definición de variables y el primer método.
- Dentro de un método, entre la definición de variables y la primera línea del método.
- Secciones lógicas dentro de un método.

```
using UnityEngine;

public class Vector3Script : MonoBehaviour
{
    Vector3 vector;
    Vector3 normalizedVector;

    void Update()
    {
        vector = transform.position;
        normalizedVector = Vector3.Normalize(vector);
    }

    void OnDrawGizmos()
    {
        Gizmos.color = Color.red;
        Gizmos.DrawLine(Vector3.zero, vector);

        Gizmos.color = Color.blue;
        Gizmos.DrawLine(Vector3.zero, normalizedVector);
    }
}
```

Paso 5.Convención de nombres

Convención PascalCase

Consiste en que la primera letra de la primera palabra va siempre en mayúscula, y si el nombre está formado por más palabras, cada una se añadirá sin espacios, y con la primera letra en mayúscula.

Ejemplo:

```
const int MyConstant = 10;
```

Utilizaremos esta convención para nombrar:

- Clases
- Interfaces
- Enumerados (tanto para la declaración como para los valores)
- Campos estáticos
- Campos de tipo readonly
- Constantes
- Eventos

Convención CamelCase

Consiste en que la primera letra de la primera palabra va siempre en minúscula, y si el nombre está formado por más palabras, cada una se añadirá sin espacios, y con la primera letra en mayúscula.

Ejemplo:

```
int myVariable = 10;
```

Utilizaremos esta convención para nombrar:

- Variables
- Parámetros de funciones

Reglas para los identificadores

- No se puede comenzar un identificador con un número. Sí que se pueden utilizar números al final o en la mitad de la palabra, pero nunca al inicio.

```
int 1one = 10;       //IDENTIFICADOR INCORRECTO
int one1 = 10;       //IDENTIFICADOR CORRECTO
int one1one = 10;    //IDENTIFICADOR CORRECTO
```

- No se pueden dejar espacios

```
int var spaces = 10;     // IDENTIFICADOR INCORRECTO
int varSpaces = 10;      // IDENTIFICADOR CORRECTO
```

- No se pueden utilizar palabras reservadas del lenguaje de programación. Por ejemplo, todos los tipos de variables son palabras reservadas del lenguaje C#.

```
int class = 10;        // IDENTIFICADOR INCORRECTO
int myClass = 10;      // IDENTIFICADOR CORRECTO
```

- Es altamente recomendable que utilices identificadores intuitivos para nombrar tus variables, de forma que te ayude a saber la información que almacena simplemente al verla.

```
int a = 10;      // No sé qué almacena la variable
                 // NO RECOMENDADO
int age = 10;    // La variable almacena la edad
                 // RECOMENDADO
```

Paso 6.Comentarios

Los comentarios se utilizan para el mantenimiento y legibilidad del código. Un comentario puede tener varios propósitos, por ejemplo:

- Documentar ciertas partes del código.
- Ocultar al compilador una parte del código, ya que los compiladores ignoran las entradas de comentarios y no las ejecutan. Esto puede ser útil cuando no queremos borrar una parte del código porque aún no está terminada, porque tiene errores y no deja compilar la aplicación, o simplemente porque hemos reescrito la funcionalidad y no queremos perder la anterior.

Hay dos tipos de comentarios:

- ***Comentarios de una línea***

  Para escribir comentarios de una sola línea vamos a utilizar dos barras //. Es el mismo símbolo que utilizamos para la operación aritmética de la división. Todo lo que vaya escrito detrás de las dos barras, será parte del comentario.

  Por ejemplo:

```
// Comentario de una línea
```

- ***Comentarios de varias líneas***

  Este comentario se representa al inicio con los caracteres /*, lo que indica que inicia un comentario multi línea. Y cerraremos el comentario con los caracteres */. Son los mismos que los utilizados en la división y en la multiplicación aritmética. Con esto, lograremos que el texto que se encuentre dentro de esta sección, sea ignorado.

  Por ejemplo:

```
/*
    Comentario multilínea

    En los comentarios multilínea, el
    comentario se extiende en varias líneas
*/
```

Recomendaciones para la utilización de comentarios

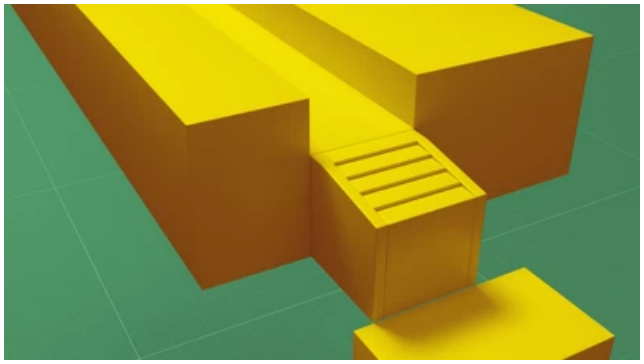
- Los comentarios tienen que ser útiles. No se debe comentar la funcionalidad del código, ni explicar cosas obvias.
- Deben ser utilizados para explicar el por qué o las limitaciones de las líneas de código. Es más útil que expliques por qué haces algo, en lugar de explicar cómo lo haces. Por ejemplo, si estás leyendo información de un array, pon un comentario de por qué lees ese array, no de cómo lo lees.
- Deben de ser concisos.

Paso 7.Otras recomendaciones

- No se debe abusar de las sentencias de salida break, continue o exit, documentando su uso cuando se considere oportuno.
- Cuando sea necesario el anidamiento de sentencias condicionales o de repetición, éste no debe superar los cuatro niveles. Si se requieren más se debe considerar la agrupación en funciones.
- Siempre que sea posible se debe reutilizar código existente, ya sean librerías o clases.
- El idioma de generación de los nombres será siempre el mismo.
- No definir campos públicos, intentar hacerlos privados. Para ellos no es necesario añadir la palabra clave private, ya que es el valor por defecto. Utilizar propiedades para hacer visibles los valores de dichos campos. La excepción a esta regla son los campos estáticos y constantes.
- No utilizar números mágicos. Los números mágicos son números utilizados directamente en el código en lugar de usar una constante o variable nombrada.

Aspectos básicos de la programación en Unity

Unidad 3 Planes de lecciones



3.5 Propiedades públicas, privadas y Proceso de serialización.

PASO 1. PROPIEDADES PÚBLICAS Y PRIVADAS EN UNITY. 3
PASO 2. PROPIEDADES PÚBLICAS. 3
PASO 3. PROPIEDADES PRIVADAS. 3
PASO 4. ATRIBUTOS SERIALIZABLES EN UNITY Y SU UTILIDAD. 4

Duración

Resumen
En este tutorial, podrás hacer lo siguiente:
→ Identificar las variables públicas y privadas y su funcionamiento.
→ Identificar la utilidad de los Serializables y su funcionamiento.

Habilidades
Comprensión de las reglas y buenas prácticas de escritura de código
→ Interpretar el código simple
Mejorar el código simple con las características de un IDE

Paso 1.Propiedades públicas y privadas en Unity.

En Unity, cuando creamos scripts para nuestros objetos, necesitamos acceder y manipular variables dentro de esos scripts. Las propiedades públicas y privadas son dos formas de definir estas variables en un script y determinar su alcance de visibilidad.

Paso 2. Propiedades públicas.

Las propiedades públicas son accesibles desde otros scripts y desde el Inspector de Unity. Esto significa que puedes modificar sus valores desde el editor sin necesidad de modificar el código. Para declarar una propiedad pública, se usa el modificador de acceso public seguido del tipo de dato y el nombre de la propiedad:

```
public class MiScript : MonoBehaviour
{
    public int miVariablePublica; // Una propiedad pública de tipo entero.
    public float otraVariablePublica; // Otra propiedad pública de tipo flotante.
    public GameObject miObjetoPublico; // Propiedad pública de tipo GameObject.
}
```

Paso 3. Propiedades privadas.

Las propiedades privadas solo son accesibles desde dentro del mismo script en el que se declaran. Es decir, no pueden ser modificadas desde el Inspector de Unity ni desde otros scripts. Se declaran con el modificador de acceso private, y también pueden ser de cualquier tipo:

```
public class MiScript : MonoBehaviour
{
    private int miVariablePrivada; // Propiedad privada de tipo entero.
    private float otraVariablePrivada; // Propiedad privada de tipo flotant
    private GameObject miObjetoPrivado; // Propiedad privada de tipo GameOb
}
```

Las propiedades privadas se utilizan cuando queremos mantener ciertos datos o lógica internos al script, sin que otros scripts puedan acceder directamente a ellos.

Paso 4. Atributos serializables en Unity y su utilidad.

Los atributos serializables en Unity son una poderosa herramienta que nos permite optimizar la organización y eficiencia en el desarrollo, al mismo tiempo que fomentan la colaboración y la comunicación efectiva en equipos de trabajo. La diversidad de atributos disponibles potencia la flexibilidad y el potencial creativo, haciendo de su uso una práctica esencial para cualquier desarrollador o diseñador que busque simplificar y mejorar su flujo de trabajo en Unity.

Algunos de los seriazable mas utilizados y utiles son los siguientes:

SerializeField:

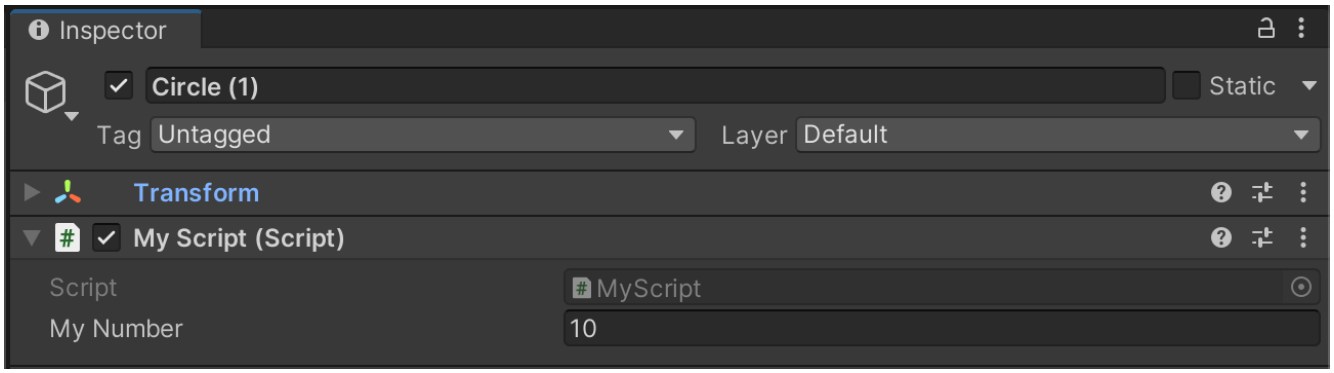
SerializeField es un atributo que se utiliza para hacer que una variable privada sea visible en el Inspector de Unity. Por defecto, las variables privadas no son visibles en el Inspector, lo que significa que no se pueden editar o asignar valores desde allí. Sin embargo, al agregar el atributo **SerializeField** encima de una variable privada, permitimos que Unity la muestre y la haga editable en el Inspector.

Ejemplo de uso:

Supongamos que tienes un script de MonoBehaviour que tiene una variable privada int que quieres modificar desde el Inspector:

```
using UnityEngine;

public class MyScript : MonoBehaviour
{
    // Esta variable privada será visible en el Inspector debido a SerializeF
    [SerializeField]
    private int myNumber = 10;
}
```



En el inspector se visualiza la el parametro disponible para actualizar.

Header:

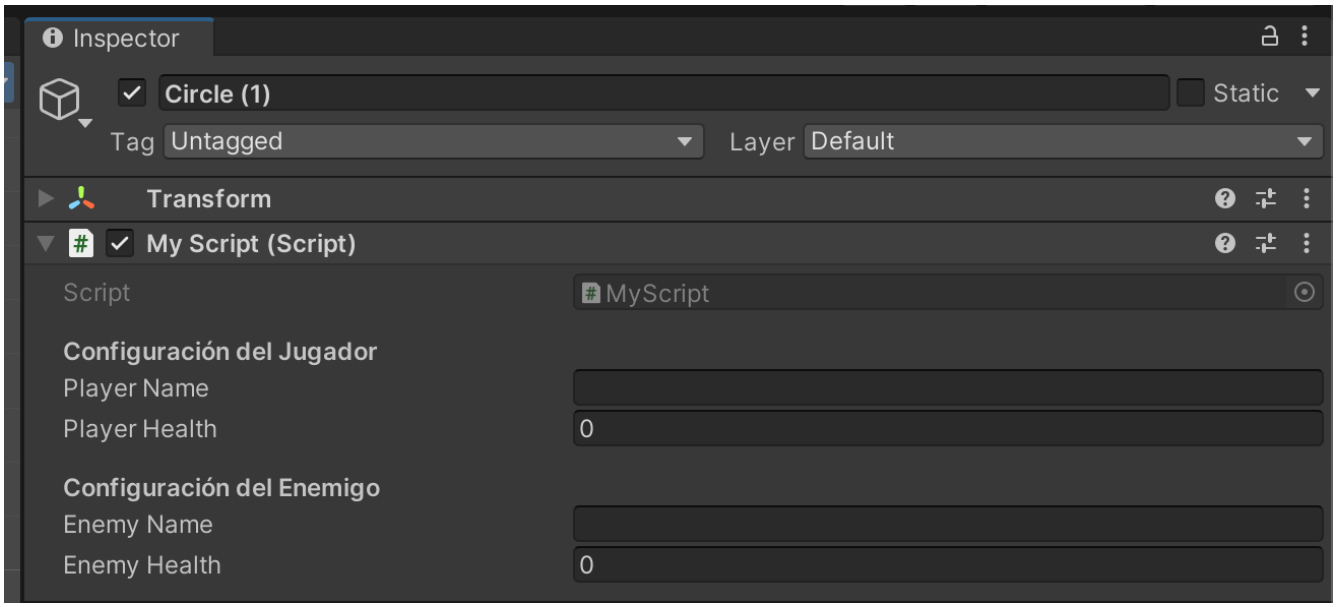
Header es un atributo que se utiliza para agregar encabezados legibles en el Inspector. Sirve para organizar y dar contexto a las diferentes variables en el Inspector. Puedes usar Header encima de una variable o antes de un grupo de variables para darle un nombre descriptivo.

Ejemplo de uso:

```
using UnityEngine;

public class MyScript : MonoBehaviour
{
    [Header("Configuración del Jugador")]
    public string playerName;
    public int playerHealth;

    [Header("Configuración del Enemigo")]
    public string enemyName;
    public int enemyHealth;
}
```



Range:

Range es un atributo que se utiliza para limitar el rango de valores que una variable numérica puede tener en el Inspector. Es útil cuando quieres asegurarte de que el valor de una variable esté dentro de un rango específico para evitar resultados inesperados o errores.

Ejemplo de uso:

```
using UnityEngine;

public class MyScript : MonoBehaviour
{
    // Esta variable solo podrá tener valores entre 1 y 100 en el Inspector
    [Range(1, 100)]
    public float speed = 50f;
}
```

Min:

Min es un atributo que se utiliza para especificar el valor mínimo que una variable numérica puede tener en el Inspector. Si se intenta establecer un valor menor que el mínimo especificado, Unity automáticamente ajustará el valor al mínimo permitido.

Ejemplo de uso:

```
using UnityEngine;

public class MyScript : MonoBehaviour
{
    // Esta variable solo podrá tener valores iguales o mayores que 5 en el
    [Min(5)]
    public int minimumValue = 10;
}
```

Space:

Space es un atributo que se utiliza para agregar espaciado visual en el Inspector de Unity. Es útil para organizar y separar visualmente las variables y secciones del Inspector para hacerlo más legible y ordenado.

Ejemplo de uso:

```
using UnityEngine;

public class MyScript : MonoBehaviour
{
    [Space]
    public int variable1;

    public float variable2;

    [Space(10)]
    public string variable3;
}
```

Tooltip:

Tooltip es un atributo que se utiliza para agregar información adicional en forma de un tooltip (información emergente) en el Inspector. Es útil para proporcionar descripciones y explicaciones sobre el propósito o el uso de una variable.

Ejemplo de uso:

```
using UnityEngine;

public class MyScript : MonoBehaviour
{
    [Tooltip("Velocidad del personaje en unidades por segundo")]
    public float speed = 5f;

    [Tooltip("Determina si el personaje puede saltar")]
    public bool canJump = true;
}
```

HideInInspector:

HideInInspector es un atributo que se utiliza para ocultar una variable en el Inspector de Unity. A diferencia de SerializeField, que muestra la variable aunque sea privada, HideInInspector se asegura de que la variable no sea visible ni editable en el Inspector, incluso si es pública.

Ejemplo de uso:

```
using UnityEngine;

public class MyScript : MonoBehaviour
{
    [HideInInspector]
    public int variable1;

    private int variable2; // Esta variable no se muestra en el Inspector
}
```

TextArea:

TextArea es un atributo que se utiliza para permitir que una cadena larga sea editada en un campo de texto más grande en el Inspector. Es útil cuando necesitas editar bloques de texto más extensos, como descripciones o notas.

Ejemplo de uso:

```
using UnityEngine;

public class MyScript : MonoBehaviour
{
    [TextArea]
    public string description;

    [TextArea(5, 10)]
    public string notes;
}
```

En este ejemplo, `description` y `notes` serán campos de texto editables en el Inspector, pero `notes` tendrá un tamaño inicial de 5 líneas y un máximo de 10 líneas.

RequireComponent:

RequireComponent es un atributo que se utiliza para asegurarse de que un GameObject que tenga un componente específico adjunto también tenga automáticamente otros componentes necesarios. Esto es útil para garantizar que un GameObject siempre tenga los componentes requeridos para que funcione correctamente, evitando errores y problemas en tiempo de ejecución.

Ejemplo de uso:

Supongamos que tienes un script que necesita un Rigidbody y un Collider para funcionar adecuadamente:

```
using UnityEngine;

[RequireComponent(typeof(Rigidbody))]
[RequireComponent(typeof(Collider))]
public class MyScript : MonoBehaviour
{
    // Tu código aquí
}
```

Con este código, si adjuntas este script a un GameObject en el Editor, Unity automáticamente agregará los componentes Rigidbody y Collider si no están presentes.

Estos atributos serializables son útiles para mejorar la eficiencia y la usabilidad en Unity al automatizar tareas, agregar funcionalidades personalizadas y mejorar la experiencia de desarrollo en general.

Execution Order en profundidad

Introducción

En esta sección vamos a profundizar en el orden de ejecución de las funciones de evento en Unity. Veremos las fases que más vamos a utilizar en nuestro desarrollo de proyectos y que son:

- Fase de inicialización (**initialization**).
- Fase de finalización (**decommissioning**).
- Fase de lógica del juego (**game logic**).
- Fase de físicas (**physics**).

Es una sección muy interesante que nos va a servir para tomar decisiones a la hora de decidir en qué función tenemos que implementar nuestro código para obtener el resultado deseado.

Fase Inicialización - Awake

Vamos a empezar a estudiar el orden de ejecución con la fase de inicialización y, dentro de la fase de inicialización, con el método Awake.

El método **Awake** es llamado:

- Cuando un gameObject activo que contiene el script es inicializado al cargarse una escena. Si la escena es cargada de nuevo, Unity carga la instancia del script de nuevo, así que Awake será llamado de nuevo, tantas veces como sea cargada la escena.
- Cuando se activa un gameObject previamente inactivo al cargarse la escena. Si una vez cargada la escena desactivamos un gameObject y lo volvemos a activar, no se volverá a llamar al Awake.
- Después de que un gameObject creado con **Object.Instantiate** es inicializado.

Awake se utiliza para inicializar variables o estados antes de que la aplicación comience.

Tenemos que tener cuidado porque no está garantizado el orden de carga entre los distintos gameObjects, por tanto no está garantizado que podamos buscar referencias externas. Aunque la documentación de Unity indica que sí que está garantizado, puedes llegar a tener problemas, así que mi consejo es que entiendas el método Awake como un constructor en el que inicializar y buscar componentes del propio gameObject. Y las referencias externas las buscaremos en el **Start**, que está garantizado que se ejecuta antes de ocurra el primer frame, cuando ya está todo cargado.

Vamos a verlo con un ejemplo. Para ello vamos a crear un script llamado **ExecOrder** con el siguiente código:

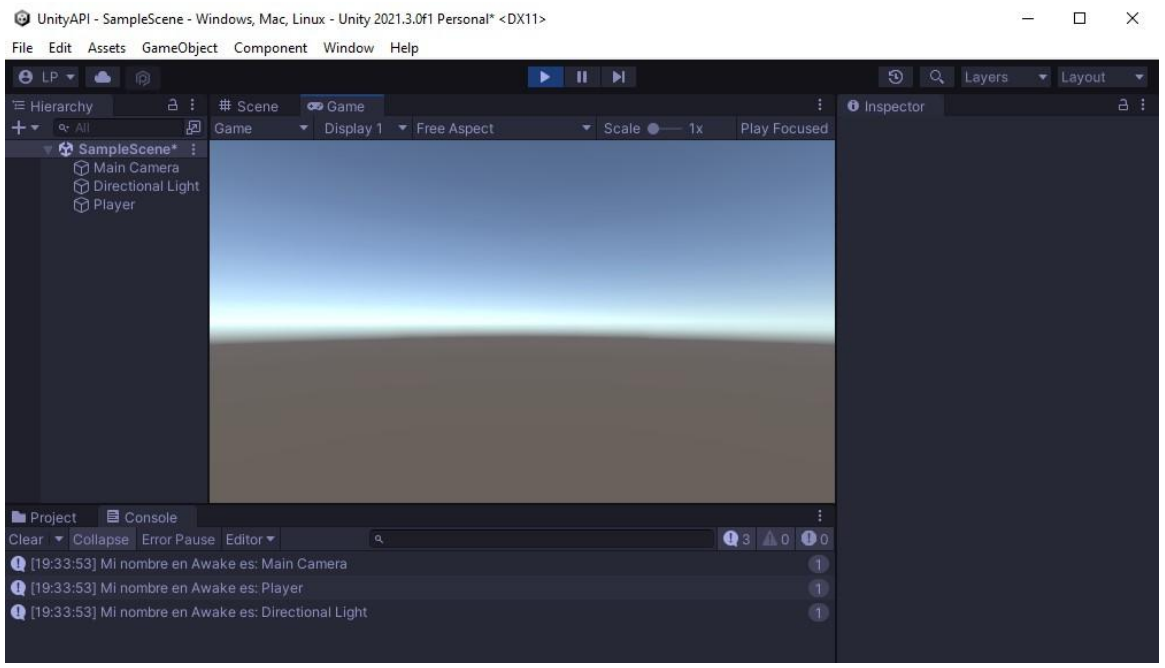
Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

```csharp
public class ExecOrder : MonoBehaviour
{
    // FASE INICIALIZATION
    private void Awake()
    {
        Debug.Log("Mi nombre en Awake es: " + name);
    }
}
```

Para ver el comportamiento del script:

- Guardamos los cambios.
- Volvemos a Unity y esperamos a que compile.
- Seleccionamos los tres gameObject que tenemos en la escena (Main Camera, Directional Light y Player) y adjuntamos el script **ExecOrder**.
- Pulsamos Play.

Los mensajes que se muestran en consola son:



Vemos que el orden en el que se han cargado los componentes es Main Camera, Player y Directional Light. Pero podría cargarlos en un orden diferente. Es por ello que no te recomiendo buscar referencias externas en el Awake, puesto que podemos intentar cargar alguna referencia de algún objeto que todavía no se haya cargado.

Fase Inicialización - OnEnable

OnEnable es llamado cada vez que se activa el componente o el gameObject.

Podemos utilizar este método por ejemplo para suscribirnos a algún evento.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

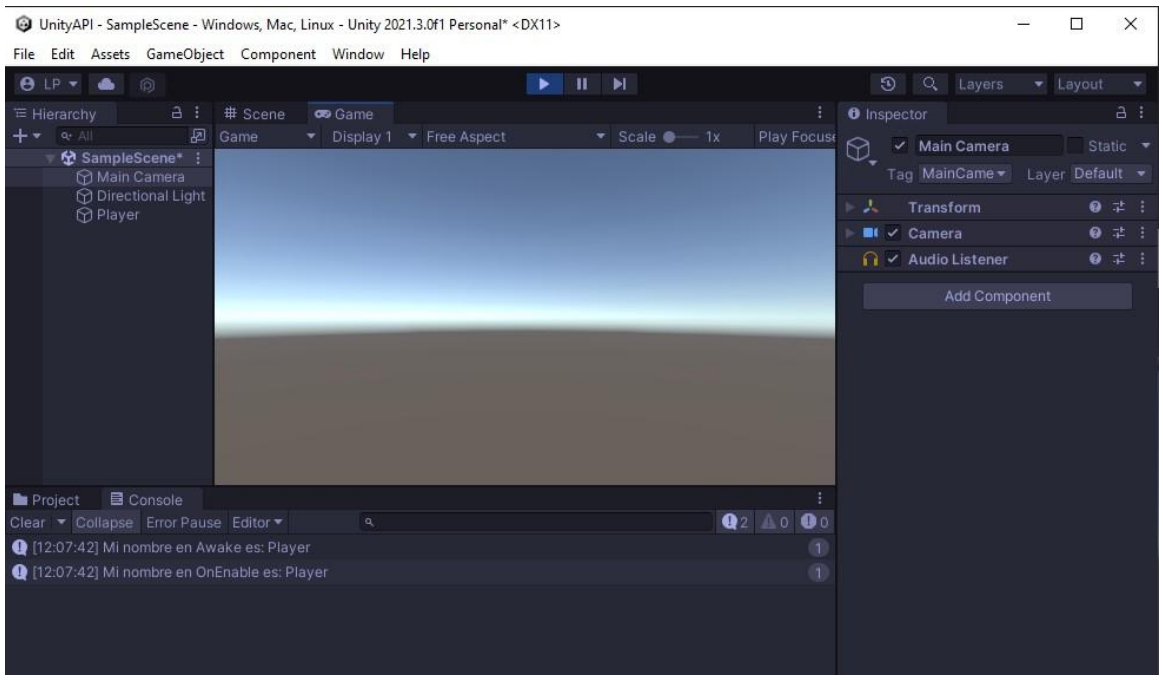
Vamos a verlo con un ejemplo. En el script **ExecOrder** añadimos la llamada al método OnEnable:

```
private void OnEnable()
{
    Debug.Log("Mi nombre en OnEnable es: " + name);
}
```

Para ver el comportamiento del script:

- Guardamos los cambios.
- Volvemos a Unity y esperamos a que compile.
- Del ejemplo anterior tenemos adjuntado el script **ExecOrder** a los gameObject Main Camera, Directional Light y Player. Vamos a eliminar el componente del gameObject Main Camera y del gameObject DirectionalLight.
- Pulsamos Play.

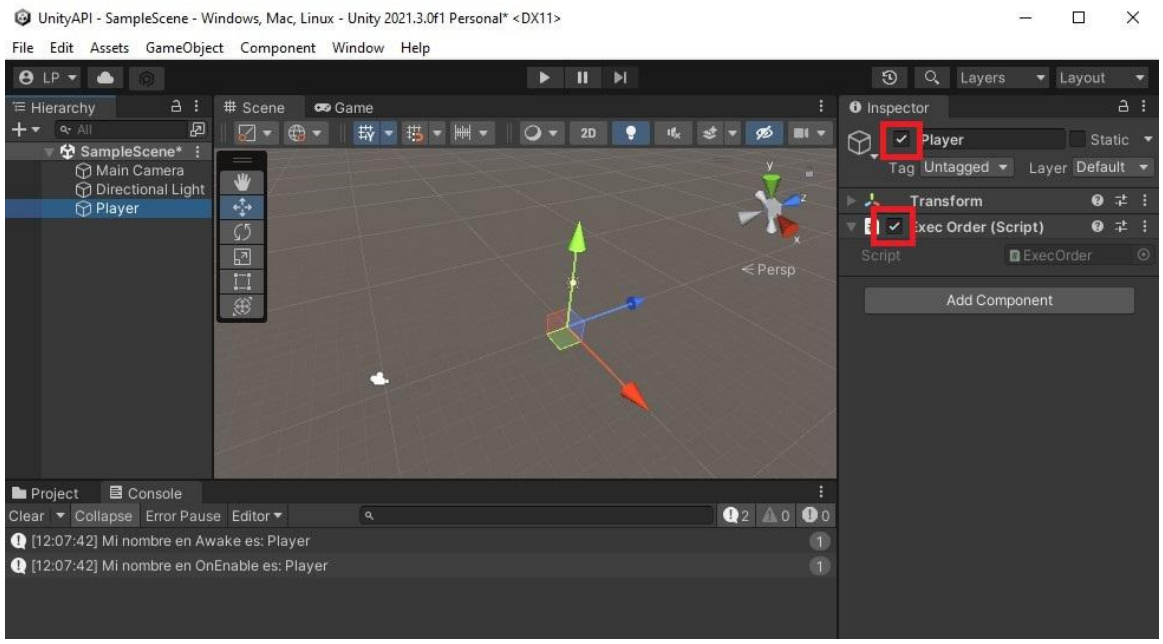
Los mensajes que se muestran en consola son:



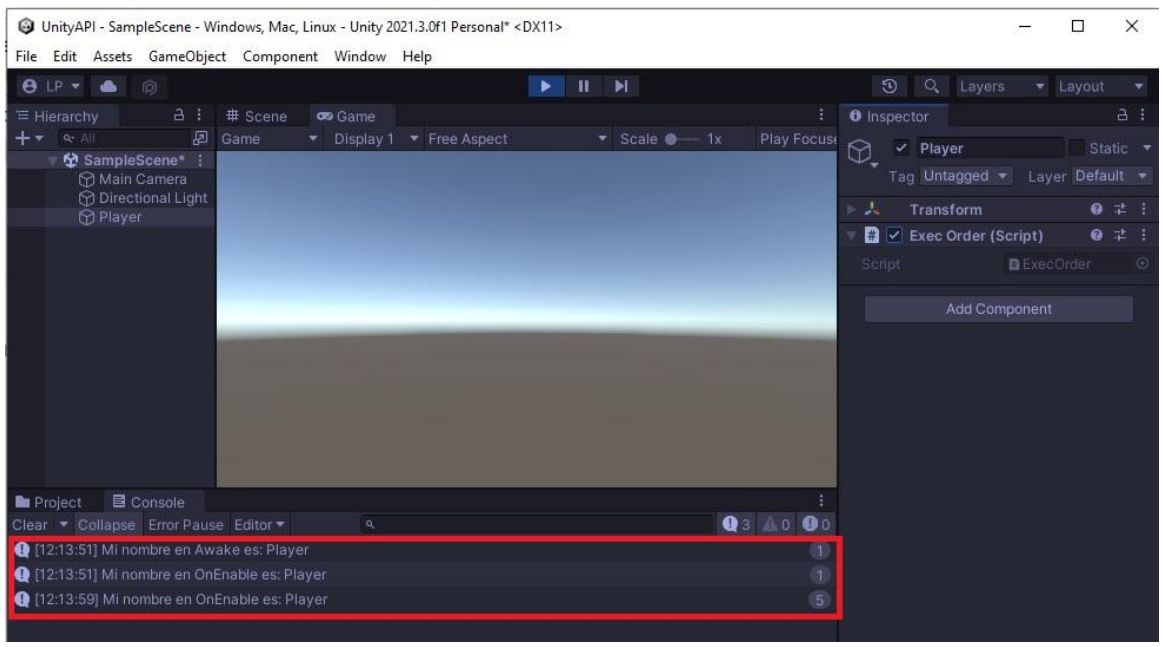
La primera vez que se carga la escena, como el gameObject Player está activo, se ejecuta el método **Awake** y el método **OnEnable**.

Si activamos y desactivamos el gameObject Player o el componente ExecOrder:

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143



Vemos que el método **Awake** no vuelve a ser llamado, pero el método **OnEnable** es llamado cada vez que se activa o el gameObject o el componente:



Fase Inicialización - Start

El siguiente método que vamos a ver es el método Start. El método **Start**:

- Se ejecuta una sola vez, como el método Awake.
- Se ejecuta cuando se han ejecutado los métodos Awake y OnEnable de todos los gameObject que hay en la escena.
- Se ejecuta antes de que se ejecute el primer frame.

Vamos a verlo con un ejemplo. En el script **ExecOrder** añadimos la llamada al método Start:

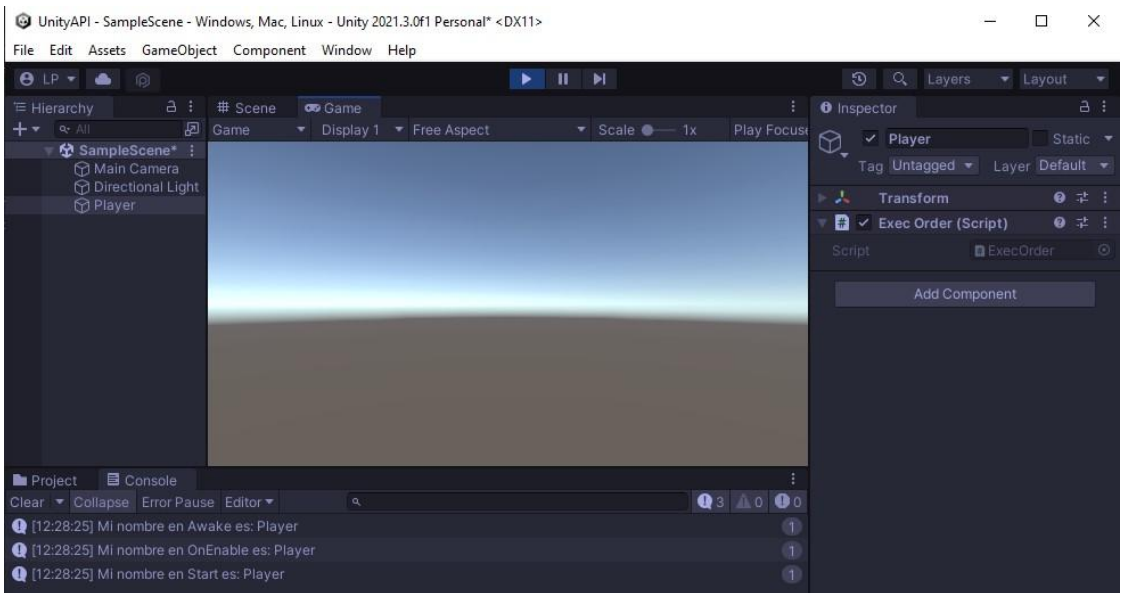
Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

```
void Start()
{
    Debug.Log("Mi nombre en Start es: " + name);
}
```

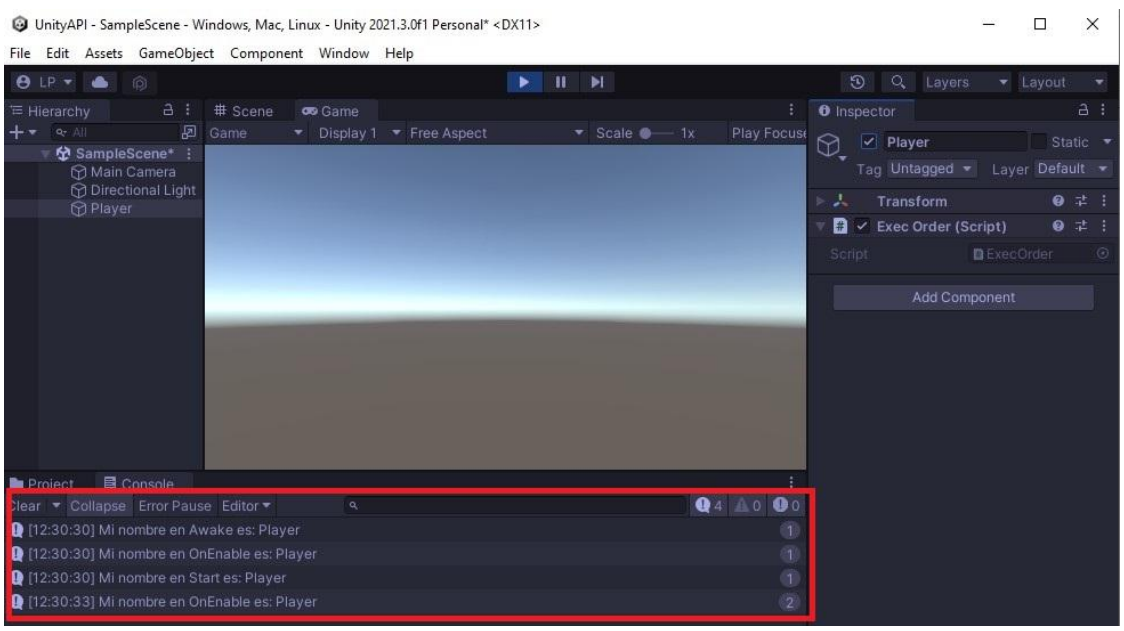
Para ver el comportamiento del script:

- Guardamos los cambios.
- Volvemos a Unity y esperamos a que compile.
- Del ejemplo anterior tenemos adjuntado el script **ExecOrder** al gameObject Player.
- Pulsamos Play.

Los mensajes que se muestran en consola son:

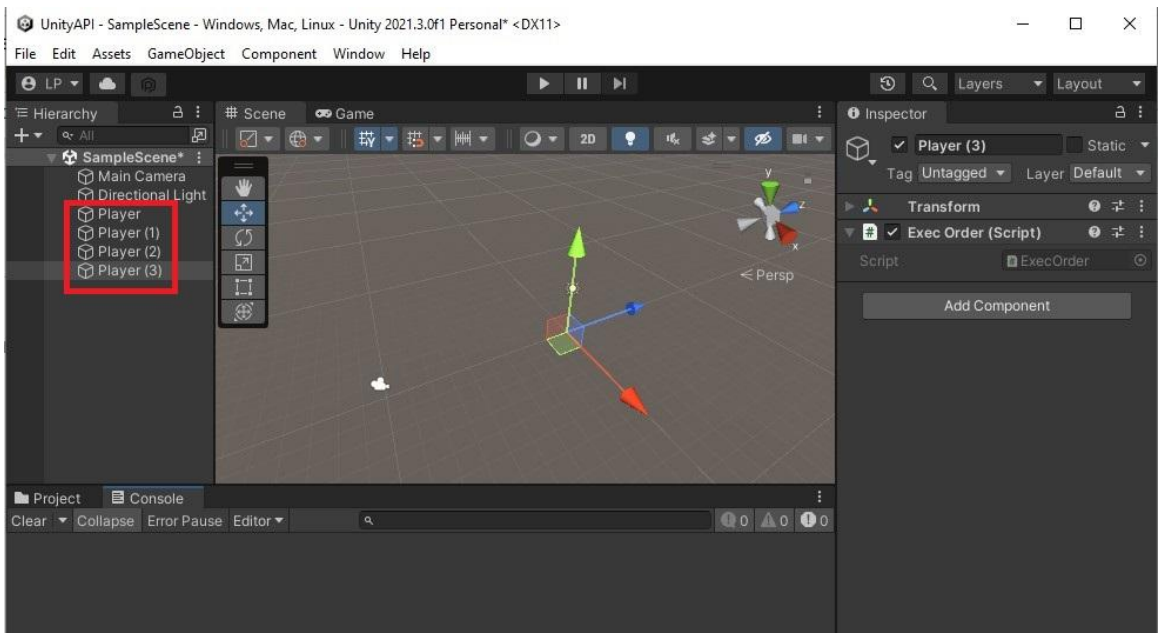


Vemos que los métodos se ejecutan en este orden, **Awake**, **OnEnable** y **Start**. Si desactivamos y activamos el gameObject Player o el componente ExecOrder, como en el punto anterior, vemos que **Start** no se vuelve a ejecutar.

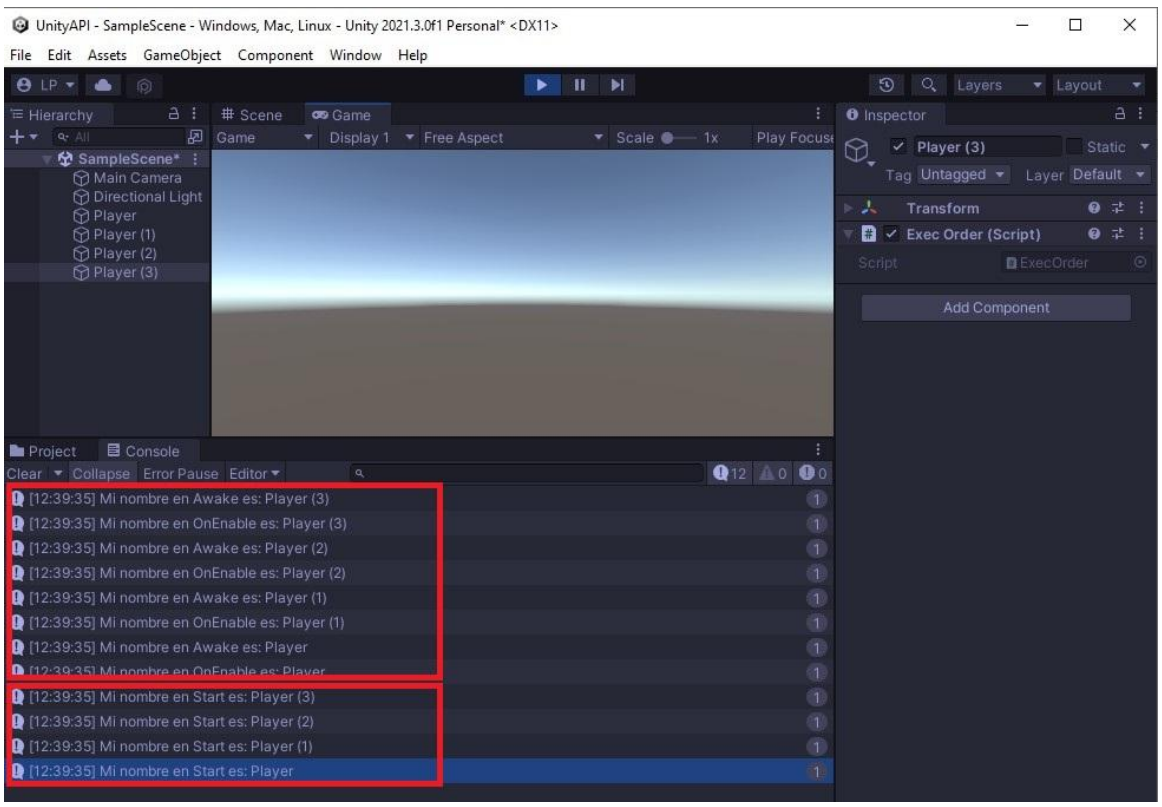


Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Para comprobar que **Start** se ejecuta después de **Awake** y **OnEnable**, vamos a crear varias copias del gameObject Player:



Si pulsamos Play, vemos que se ejecutan primero los métodos **Awake** y **OnEnable** de todos los gameObjects, y finalmente el método **Start**.



Fase Finalización - OnDisable

Este método es llamado cuando se desactiva o se destruye un gameObject.

Podemos utilizar este método por ejemplo para desuscribirnos de algún evento.

Vamos a verlo con un ejemplo. En el script **ExecOrder** añadimos la llamada al método OnDisable:

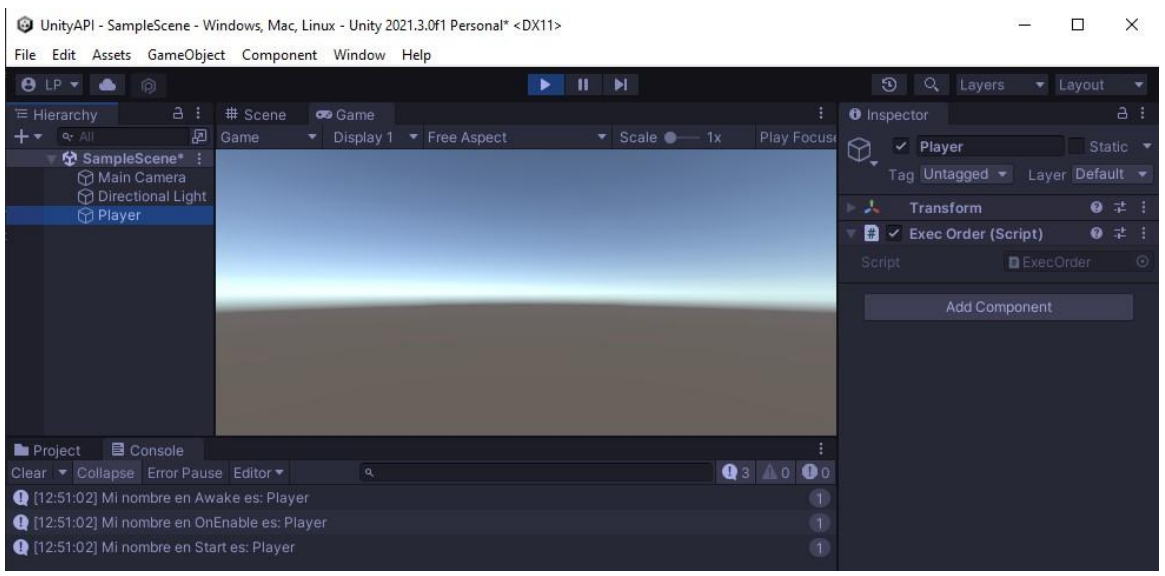
Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

```
private void OnDisable()
{
    Debug.Log("Mi nombre en OnDisable es: " + name);
}
```

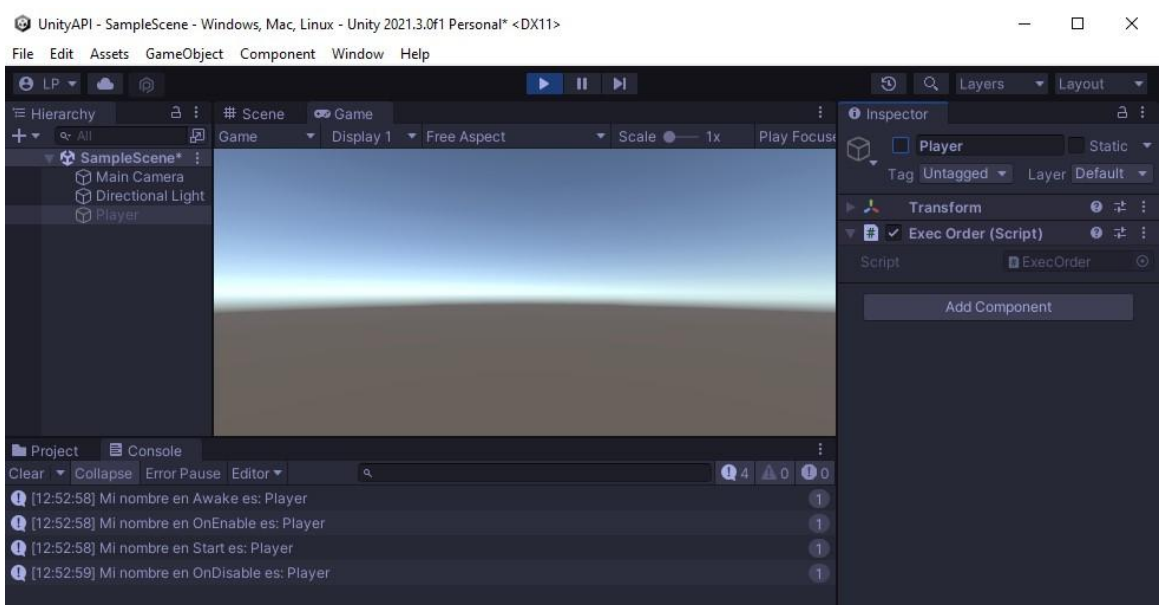
Para ver el comportamiento del script:

- Guardamos los cambios.
- Volvemos a Unity y esperamos a que compile.
- Del ejemplo anterior tenemos adjuntado el script **ExecOrder** al gameObject Player. Borramos todos los gameObject Player que habíamos duplicado.
- Pulsamos Play.

Los mensajes que se muestran en consola son:

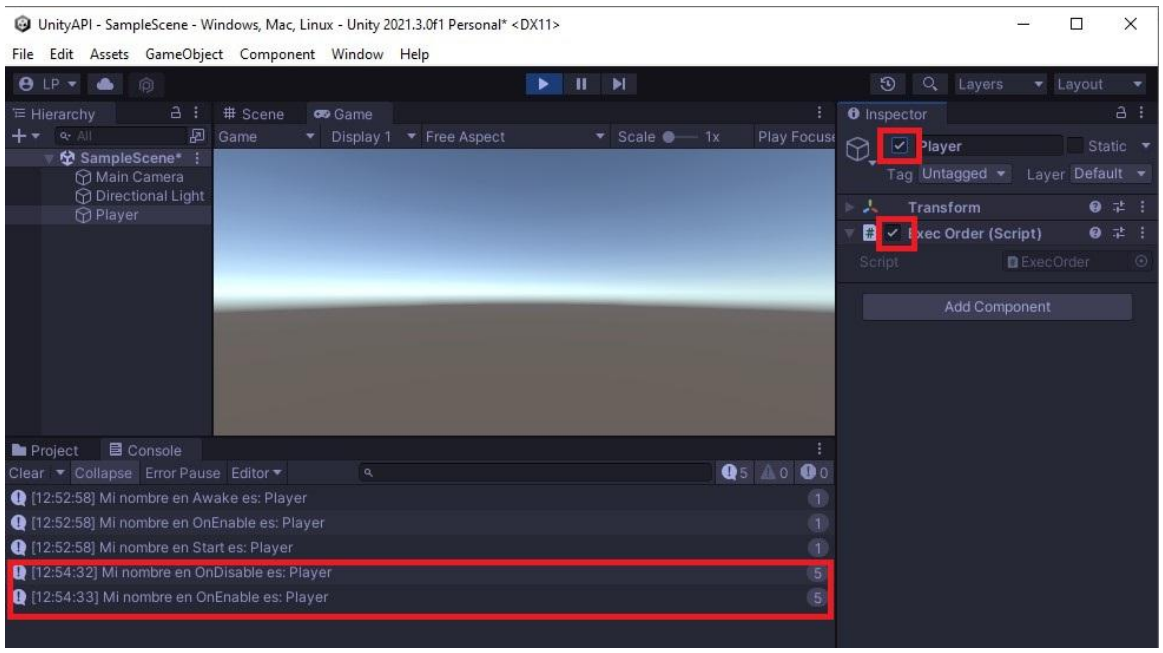


Vemos que se ejecutan los métodos **Awake**, **OnEnable** y **Start**. Si desactivamos el gameObject Player o el componente ExecOrder vemos que se ejecuta el método **OnDisable**:



Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Si lo volvemos a activar, se llama de nuevo al método **OnEnable**, y así sucesivamente cada vez que se activen y desactiven el gameObject o el componente.



Fase Finalización - OnDestroy

Este método es llamado cuando se elimina un gameObject en tiempo de ejecución o se destruye la escena que está cargada.

Vamos a verlo con un ejemplo. En el script **ExecOrder** añadimos la llamada al método OnDestroy:

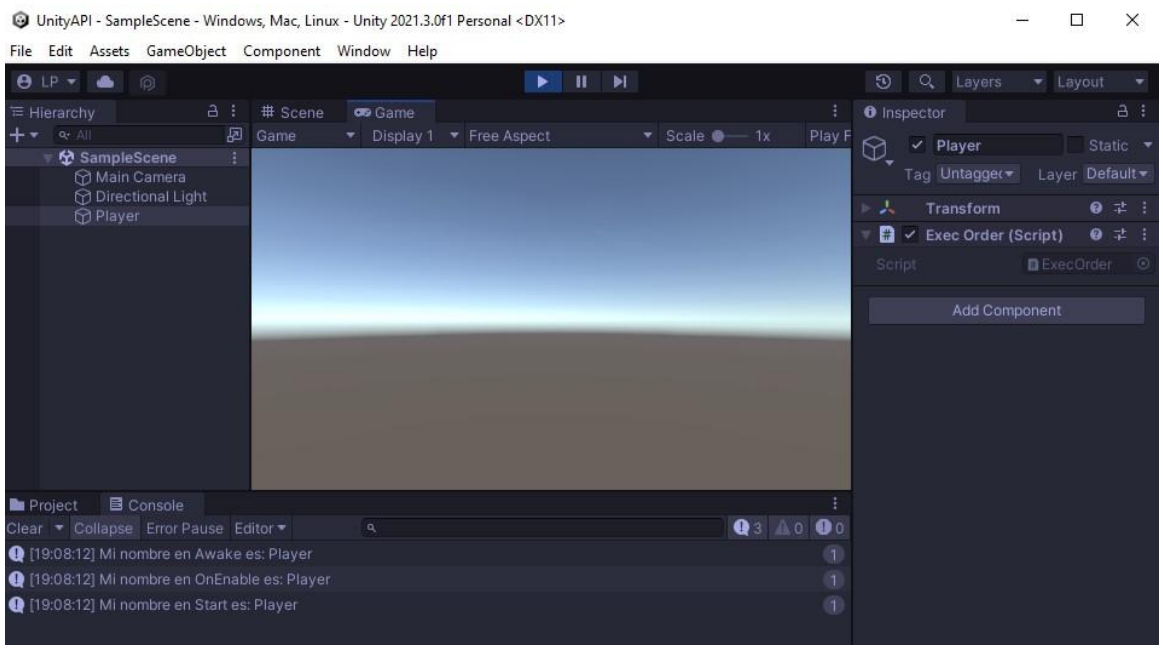
```
private void OnDestroy()
{
    Debug.Log("Mi nombre en OnDestroy es: " + name);
}
```

Para ver el comportamiento del script:

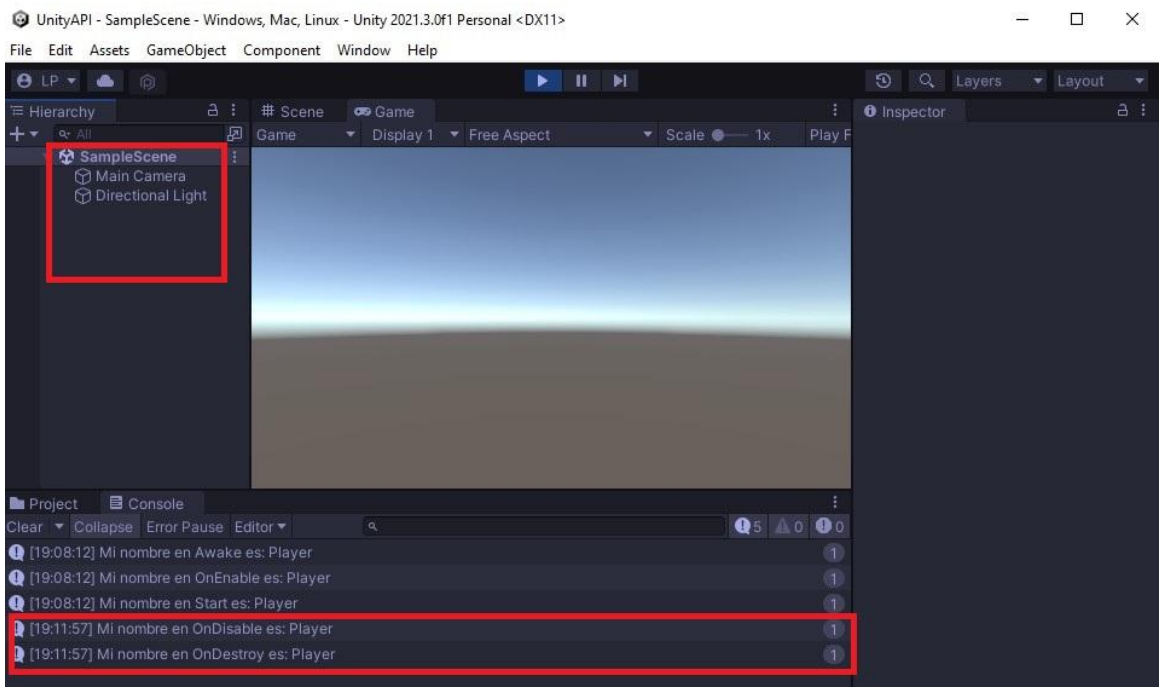
- Guardamos los cambios.
- Volvemos a Unity y esperamos a que compile.
- Del ejemplo anterior tenemos adjuntado el script **ExecOrder** al gameObject Player.
- Pulsamos Play.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Los mensajes que se muestran en consola son:



Ahora vamos a simular la destrucción del gameObject. Para ello lo borramos de la jerarquía. En el momento de borrar el player, son llamados los métodos **OnDisable** y **OnDestroy**.



Para simular la destrucción de la escena podemos ejecutar el programa y detener la ejecución. El resultado en consola sería el mismo.

Fase Finalización - OnApplicationQuit

Este método es llamado cuando se cierra la aplicación.

Vamos a verlo con un ejemplo. En el script **ExecOrder** añadimos la llamada al método OnApplicationQuit:

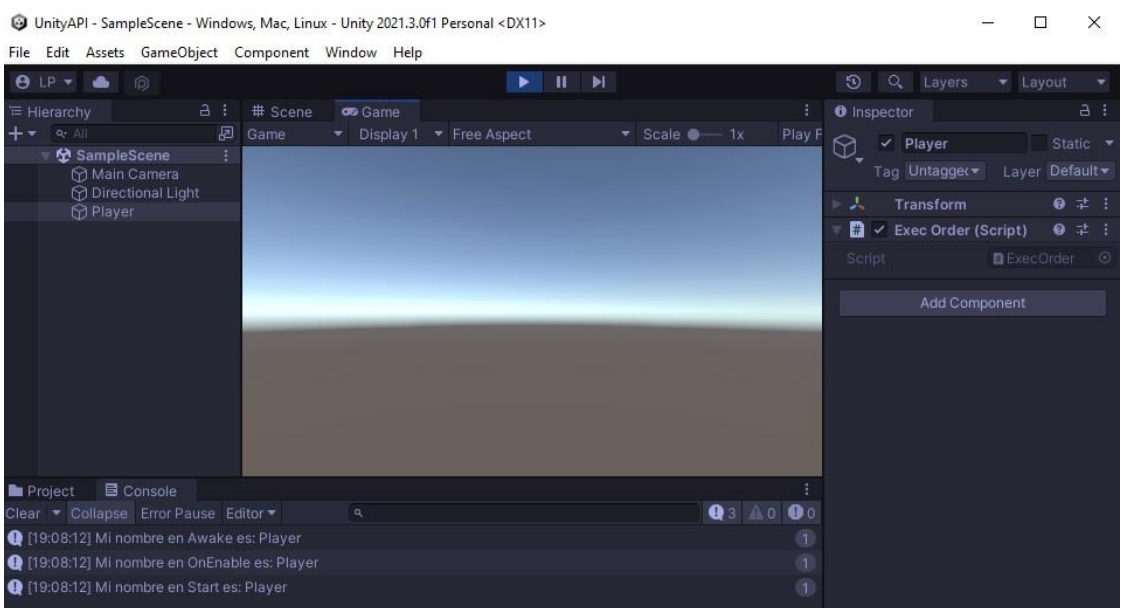
Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

```
private void OnApplicationQuit()
{
    Debug.Log("Mi nombre en AppQuit es: " + name);
}
```

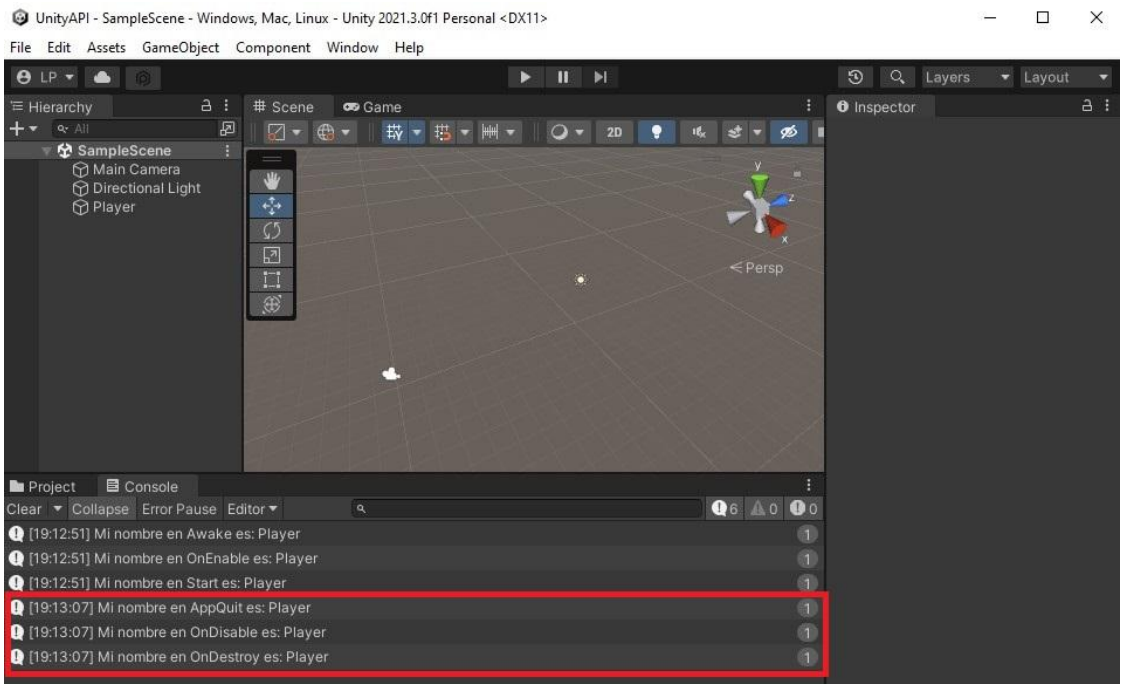
Para ver el comportamiento del script:

- Guardamos los cambios.
- Volvemos a Unity y esperamos a que compile.
- Del ejemplo anterior tenemos adjuntado el script **ExecOrder** al gameObject Player.
- Pulsamos Play.

Los mensajes que se muestran en consola son:

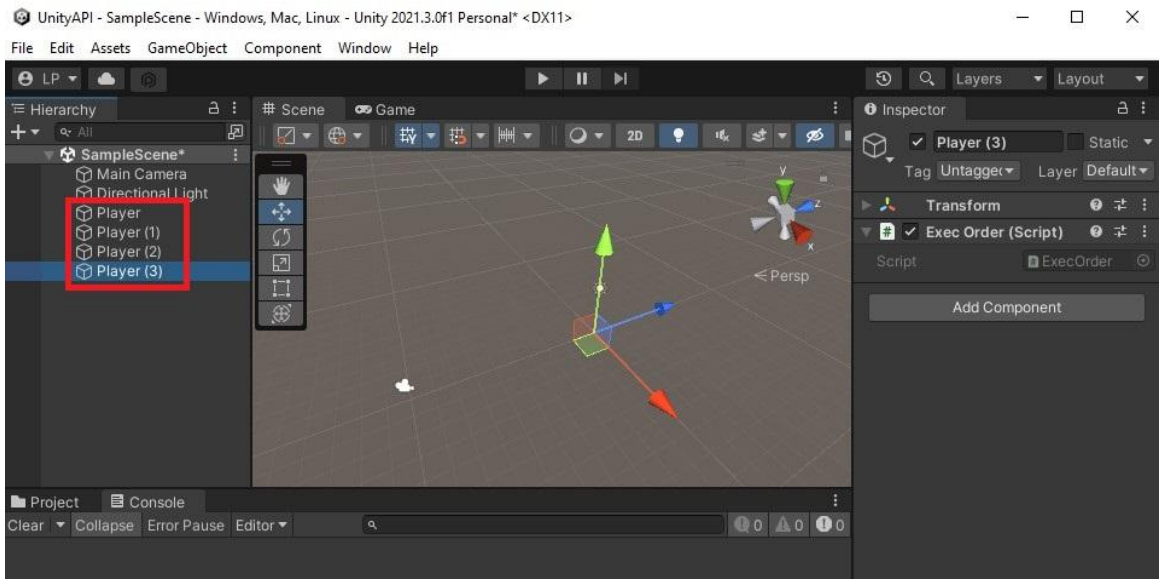


Para simular el cierre de la aplicación podemos detener la ejecución. El resultado en consola es:

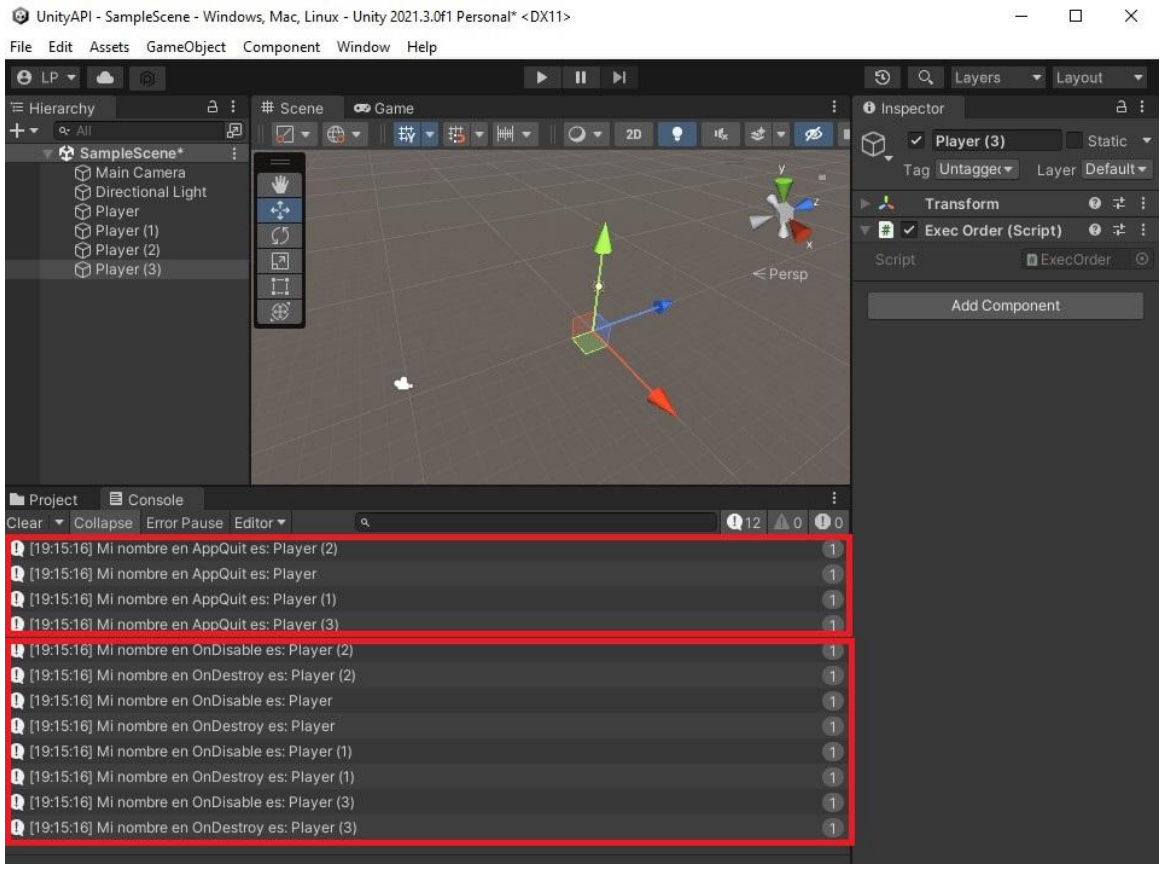


Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Para ver el orden de ejecución de los métodos en la fase de finalización, vamos a crear varias copias del gameObject Player:



Si pulsamos Play, pulsamos Clear en la consola para ver sólo los mensajes de la fase de finalización y detenemos la ejecución, vemos que se ejecuta primero el método **OnApplicationQuit** de todos los gameObject y posteriormente los métodos **OnDisable** y **OnDestroy** de cada uno de ellos.



Fase Lógica de juego - Update

El primer método de la fase de lógica de juego es el **Update**. Es uno de los más importantes porque es el primero que vamos a tener para entrar en el frame a frame, es decir, cada vez que se produce un frame del juego se ejecutará el Update.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

El Update va a procesar toda la lógica del juego. No se va a pasar al siguiente frame hasta que se ejecuten todos los Update de todos los gameObject de la escena. Por tanto, el Update es el responsable de escuchar en el frame a frame todo lo que pueda llegar a ocurrir, tomar las decisiones necesarias en función de todo ello y va a ser el que va a marcar la cantidad de frames por segundo.

Vamos a verlo con un ejemplo:

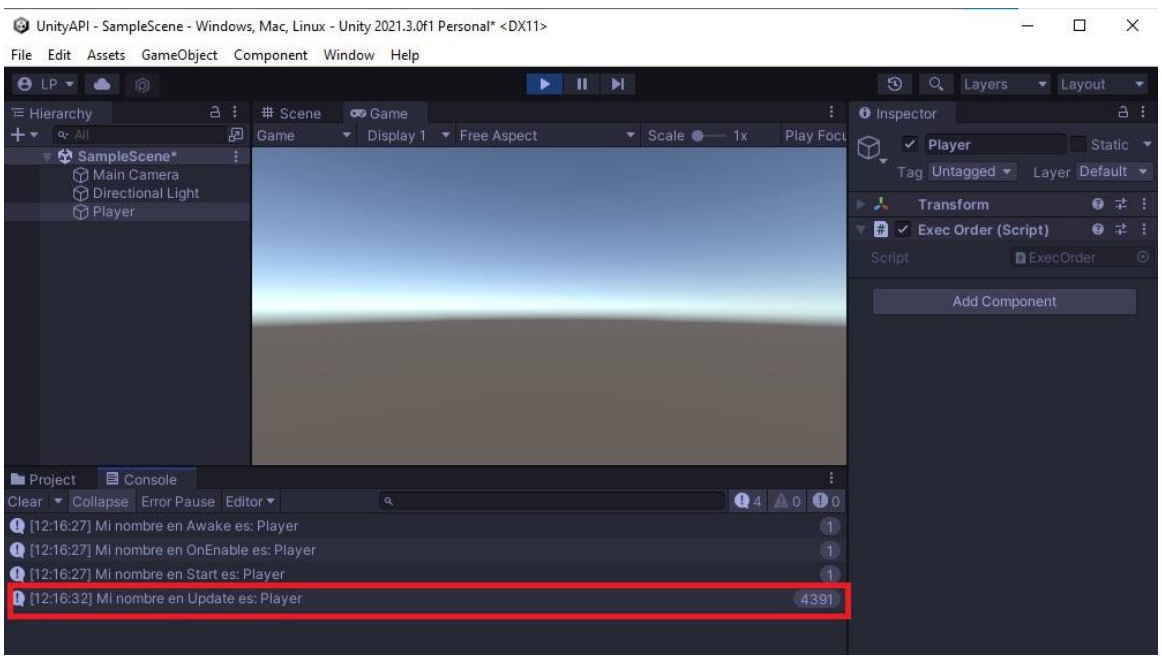
En el script **ExecOrder** añadimos la llamada al método Update:

```
void Update()
{
    Debug.Log("Mi nombre en Update es: " + name);
}
```

Para ver el comportamiento del script:

- Guardamos los cambios.
- Volvemos a Unity y esperamos a que compile.
- Del ejemplo anterior tenemos adjuntado el script **ExecOrder** al gameObject Player. Borramos todos los gameObject Player que habíamos duplicado.
- Pulsamos Play.

Los mensajes que se muestran en consola son:



Vemos que, como en cada frame se ejecuta el método **Update**, el número de mensajes es muy grande. Por ello, tenemos que ser cuidadosos con la lógica incluída en el método **Update**.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Fase Lógica de juego - LateUpdate

El siguiente método que vamos a ver en la fase de lógica de juego es el **LateUpdate**. Este método sirve para garantizar que cierta lógica se ejecuta después de otra. Vamos a necesitar esta secuenciación de código porque no podemos garantizar un orden en la ejecución de los métodos **Update** de todos los gameObject.

Un caso común en el que se trabaja con LateUpdate es cuando tenemos que hacer movimientos de cámara para seguir un objetivo. El movimiento del objetivo se hará en el Update, y el de la cámara en el LateUpdate, para asegurarnos de que es posterior.

Vamos a verlo con un ejemplo:

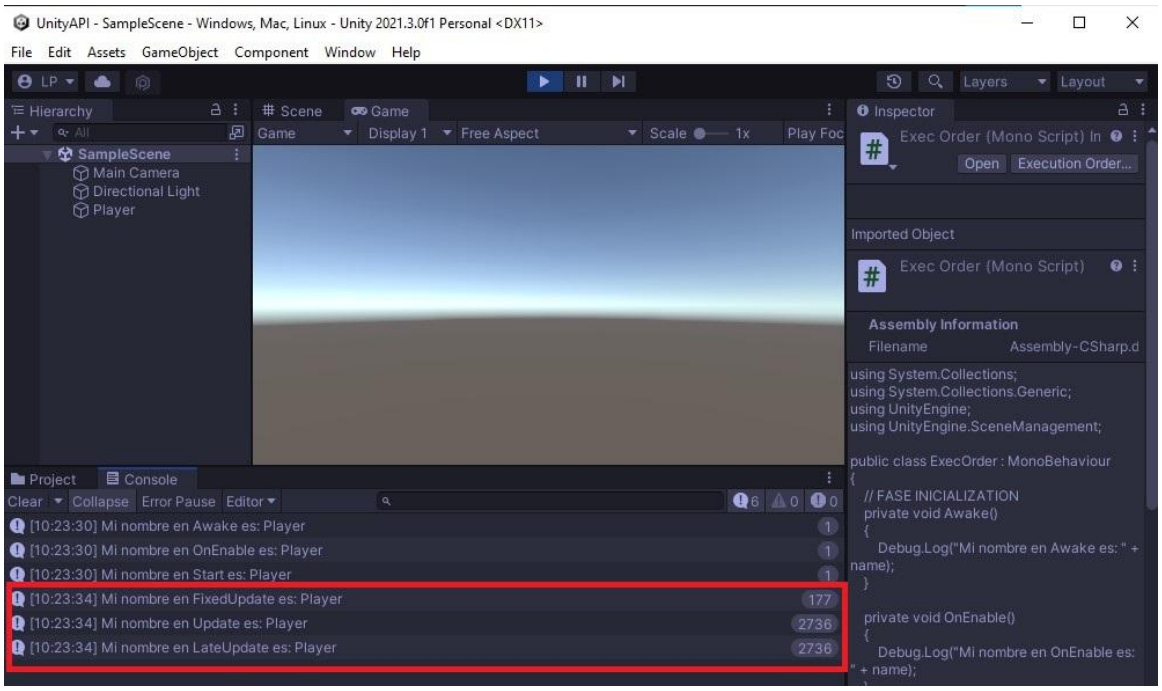
En el script **ExecOrder** añadimos la llamada al método LateUpdate:

```
private void LateUpdate()
{
    Debug.Log("Mi nombre en LateUpdate es: " + name);
}
```

Para ver el comportamiento del script:

- Guardamos los cambios.
- Volvemos a Unity y esperamos a que compile.
- Del ejemplo anterior tenemos adjuntado el script **ExecOrder** al gameObject Player.
- Pulsamos Play.

Los mensajes que se muestran en consola son:



Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Vemos que primero se ejecuta el **FixedUpdate**, posteriormente el **Update**, y por último el **LateUpdate**. **Update** y **LateUpdate** comparten el mismo número de llamadas.

Fase de Físicas - FixedUpdate

El método **FixedUpdate** está preparado para realizar todos los cálculos relacionados con un componente de físicas, ya que es llamado un número fijo de veces cada segundo, a diferencia del método **Update**, que es llamado un número variable de veces en cada segundo. El tiempo por defecto entre llamadas es 0.02 segundos (50 llamadas por segundo).

Este valor puede ser cambiado dentro de un script, o desde Unity en las opciones de menú **Editar** → **Ajustes** → **Tiempo** → **Tiempo Fijo**.

Vamos a verlo con un ejemplo:

En el script **ExecOrder** añadimos la llamada al método FixedUpdate:

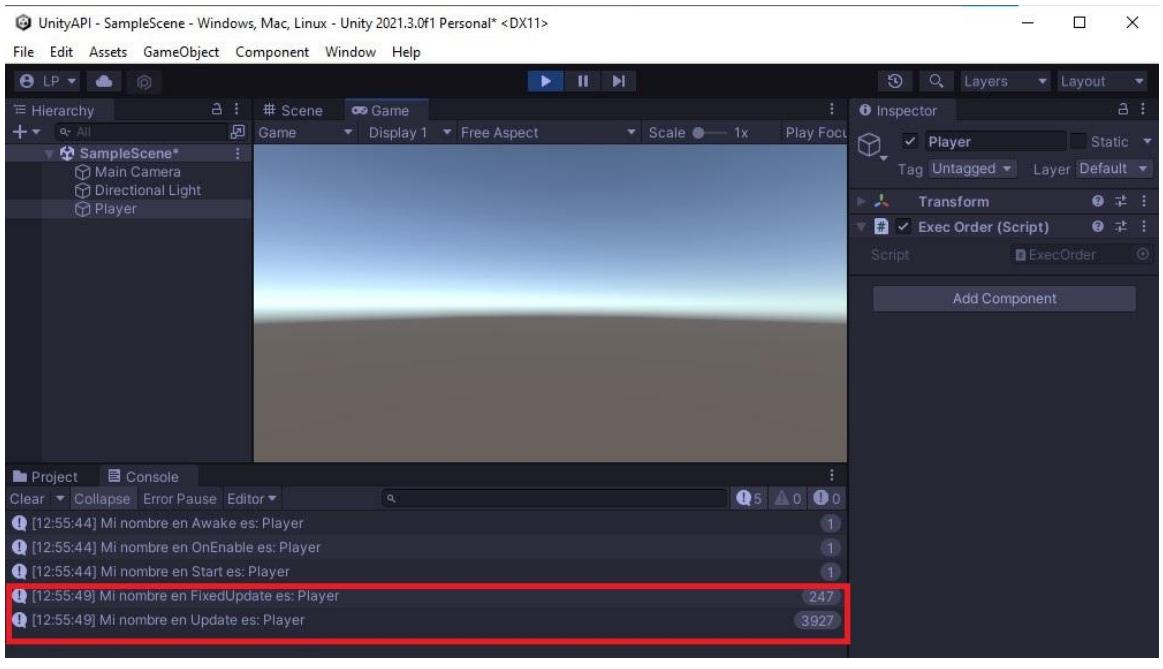
```
private void FixedUpdate()
{
    Debug.Log("Mi nombre en FixedUpdate es: " + name);
}
```

Para ver el comportamiento del script:

- Guardamos los cambios.
- Volvemos a Unity y esperamos a que compile.
- Del ejemplo anterior tenemos adjuntado el script **ExecOrder** al gameObject Player.
- Pulsamos Play.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Los mensajes que se muestran en consola son:



Vemos la diferencia de mensajes que hay entre el método **Update** y el método **FixedUpdate**. En el **FixedUpdate**, con los valores por defecto, se producen unas 50 llamadas por segundo, mientras que en el método **Update**, dependiendo del cálculo de la lógica, es un valor variable que puede ir por encima o por debajo.

En el orden de ejecución, se ejecuta antes el **FixedUpdate** que el **Update**.

Puedes ver el flujo completo del orden de ejecución en la documentación oficial de Unity:

- https://docs.unity3d.com/Manual/ExecutionOrder.html

Para recordar ...

- Las cuatro fases que hay en el orden de ejecución son "fase de inicialización (**initialization**), fase de finalización (**decommissioning**), fase de lógica del juego (**game logic**) y fase de físicas (**physics**)
- En la fase de inicialización tenemos el método **Awake**, **OnEnable** y **Start**
- El método **Awake** es llamado cuando un gameObject activo que contiene el script es inicializado al cargarse una escena, cuando se activa un gameObject previamente inactivo al cargarse la escena o después de que un gameObject creado con **Object.Instantiate** es inicializado
- El método **OnEnable** es llamado cada vez que se activa el componente o el gameObject
- El método **Start** se ejecuta una sola vez cuando se han ejecutado los métodos Awake y OnEnable de todos los gameObject que hay en la escena. Se ejecuta antes de que se ejecute el primer frame
- En la fase de finalización tenemos el método **OnDisable**, **OnDestroy** y **OnApplicationQuit**
- El método **OnDisable** es llamado cuando se desactiva o se destruye un gameObject
- El método **OnDestroy** es llamado cuando se elimina un gameObject en tiempo de ejecución o se destruye la escena que está cargada

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

- El método **OnApplicationQuit** es llamado cuando se cierra la aplicación
- En la fase de lógica de juego tenemos los métodos **Update** y **LateUpdate**
- El método **Update** va a procesar toda la lógica del juego. No se va a pasar al siguiente frame hasta que se ejecuten todos los métodos Update de todos los gameObject de la escena
- El método **LateUpdate** sirve para garantizar que cierta lógica se ejecuta después de otra. Se necesita esta secuenciación de código porque no podemos garantizar un orden en la ejecución de los métodos Update de todos los gameObject
- En la fase de físicas tenemos el método **FixedUpdate**
- El método **FixedUpdate** está preparado para realizar todos los cálculos relacionados con un componente de físicas, ya que es llamado un número fijo de veces cada segundo

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Componentes de tipo GameObject y Transform

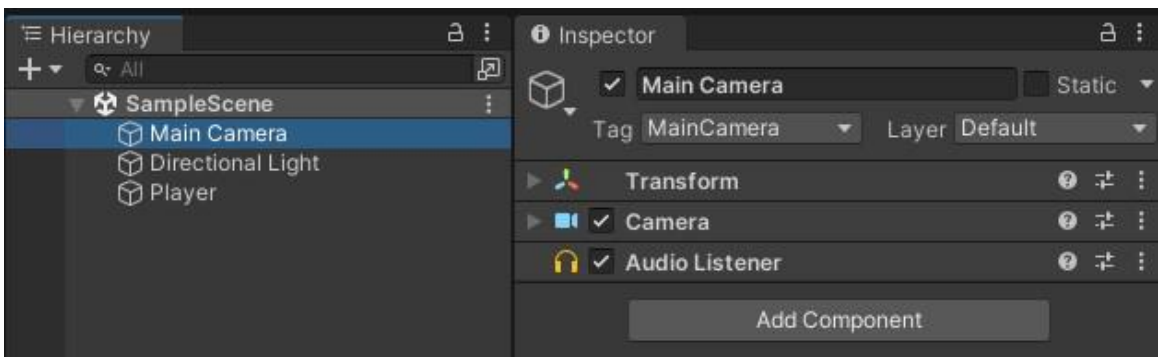
Qué son las instancias de gameObjects

Todos los gameObjects tienen dos componentes por defecto: un componente de tipo **GameObject** y un componente de tipo **Transform**. Están siempre presentes, aunque se trate de un gameObject vacío, y no se pueden eliminar. Dependiendo de la funcionalidad que le queramos dar a ese gameObject iremos añadiendo el resto de componentes.

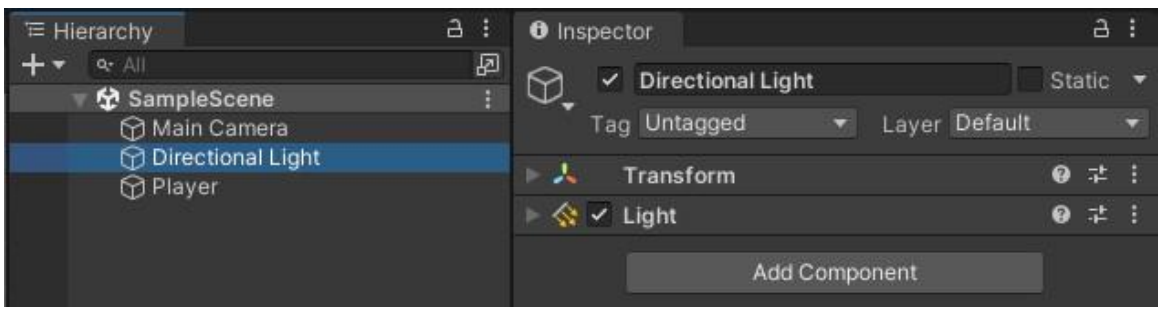
Lo que ocurre en el editor cuando ejecutamos la aplicación a través del botón Play es que, antes de empezar el orden de ejecución, se crea una instancia de los componentes de tipo **GameObject** y **Transform** para cada uno de los gameObjects que hay en la jerarquía.

Por ejemplo, en la siguiente imagen tenemos una jerarquía con tres gameObjects.

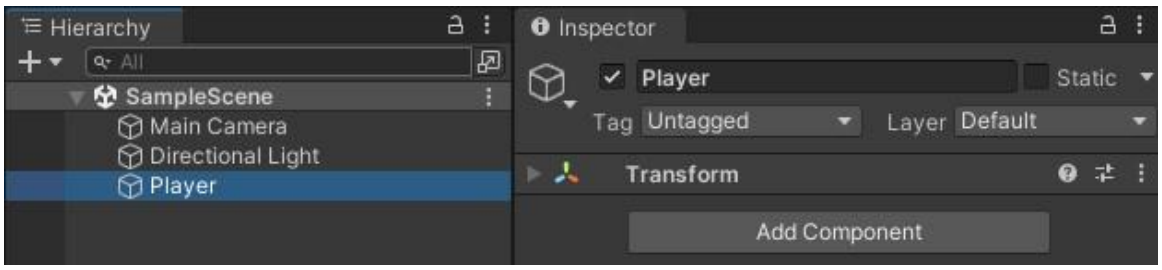
La **Main Camera** tiene un componente de GameObject, un componente de Transform, un componente de Camera y un componente de Audio Listener.



La **Directional Light** tiene un componente de GameObject, un componente de Transform y un componente de Light.



Y, por último, el **Player** tiene un componente de GameObject y un componente de Transform.



Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

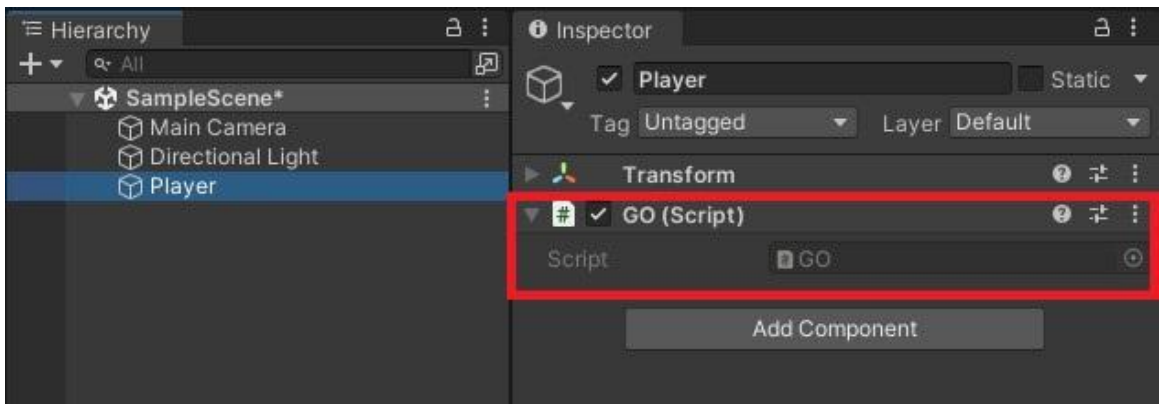
Cuando se ejecuta la aplicación, se crea una instancia de tipo **GameObject** y tipo **Transform** de cada uno de los gameObjects de la jerarquía, en nuestro ejemplo, Main Camera, Directional Light y Player. Para poder acceder al resto de componentes tenemos que acceder a la instancia de GameObject y, a través de esta instancia, podremos acceder al resto de componentes que tiene el gameObject con una de las funciones de la familia de **GetComponent**, que veremos más adelante.

Accediendo a las instancias de GameObjects

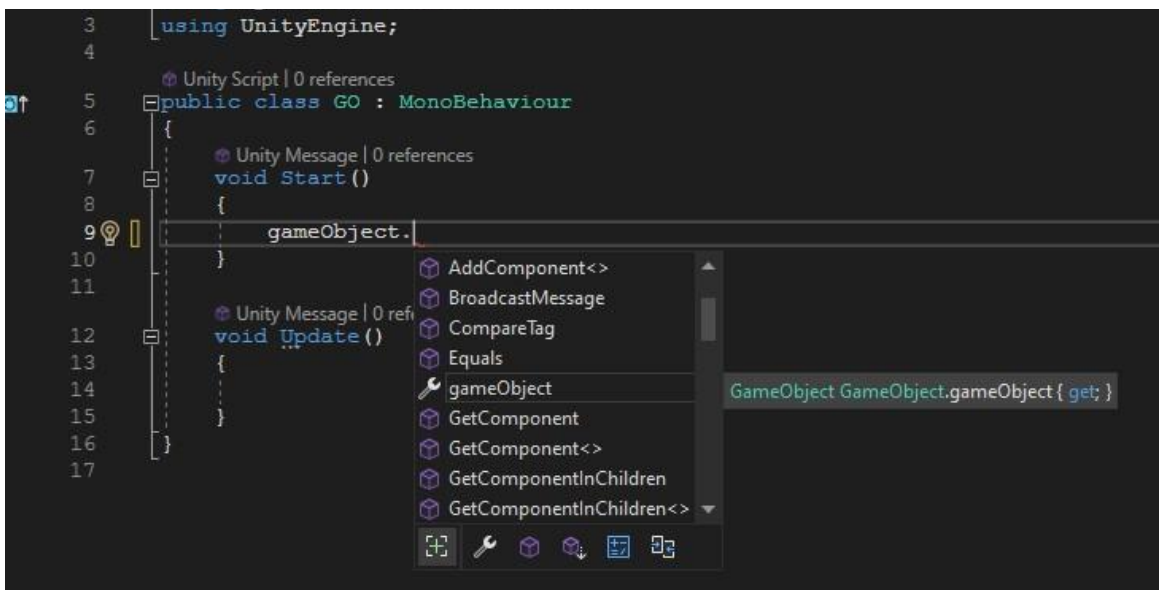
Vamos a ver cómo acceder a las instancias de GameObject y Transform que se crean de cada gameObject. También vamos a ver la diferencia entre **GameObject** - **gameObject** y entre **Transform** - **transform**.

Si creamos un script y lo adjuntamos a un gameObject de la jerarquía, este script será un componente de ese gameObject.

Por ejemplo, si creamos un script llamado GO y lo adjuntamos a un gameObject vacío de la jerarquía llamado **Player** tenemos un nuevo componente en ese gameObject.



Si en el script utilizamos **gameObject**, estamos accediendo a la instancia que se crea del gameObject al que está adjunto el script. En este ejemplo estamos accediendo a la instancia de GameObject del objeto **Player**.



Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Vamos a ver un ejemplo:

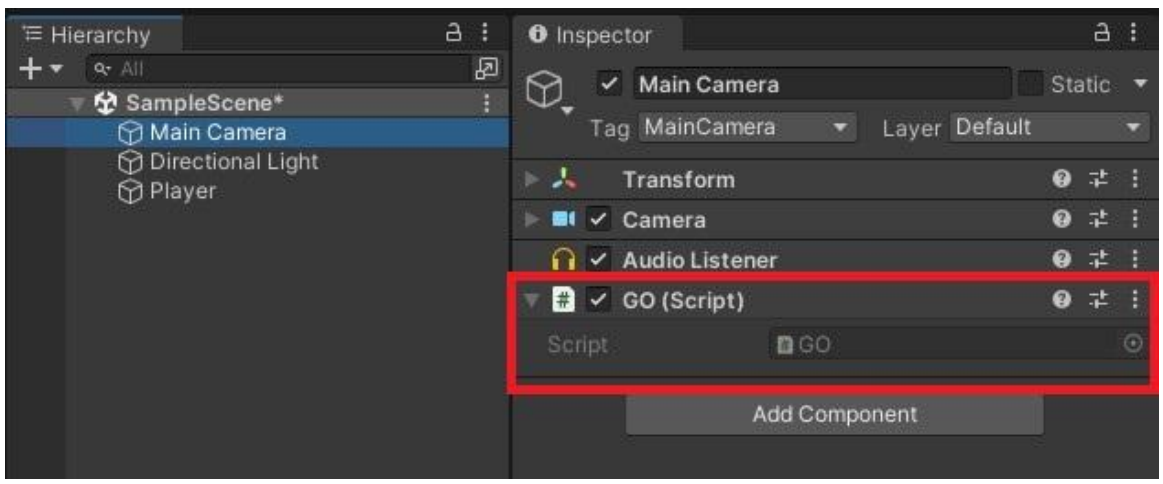
```
using UnityEngine;

public class GO : MonoBehaviour
{
    void Start()
    {
        Debug.Log(gameObject.name);
    }
}
```

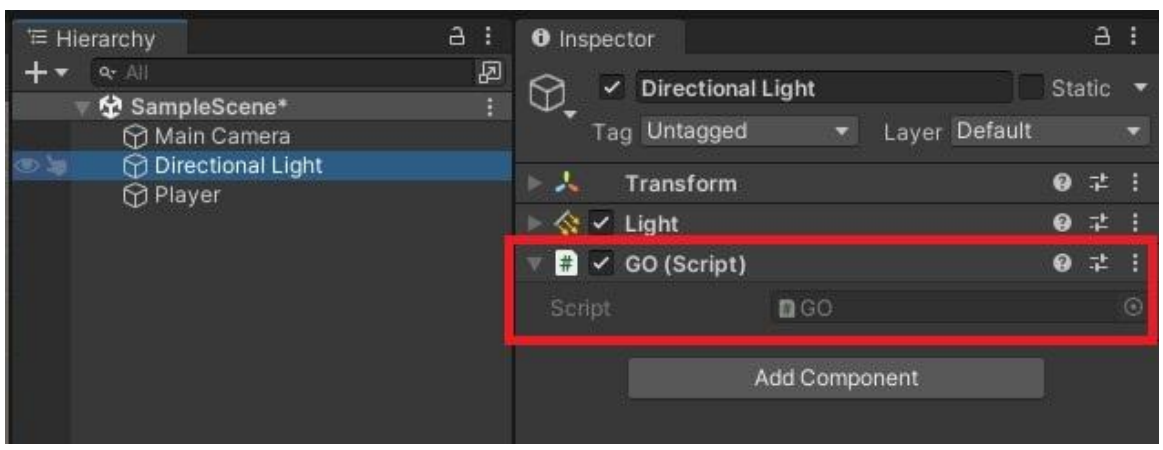
Este código muestra un script llamado GO que muestra el nombre del gameObject al que esté adjuntado el script.

Vamos a adjuntar el script a los tres gameObjects que tenemos en la jerarquía del ejemplo anterior:

Main Camera:

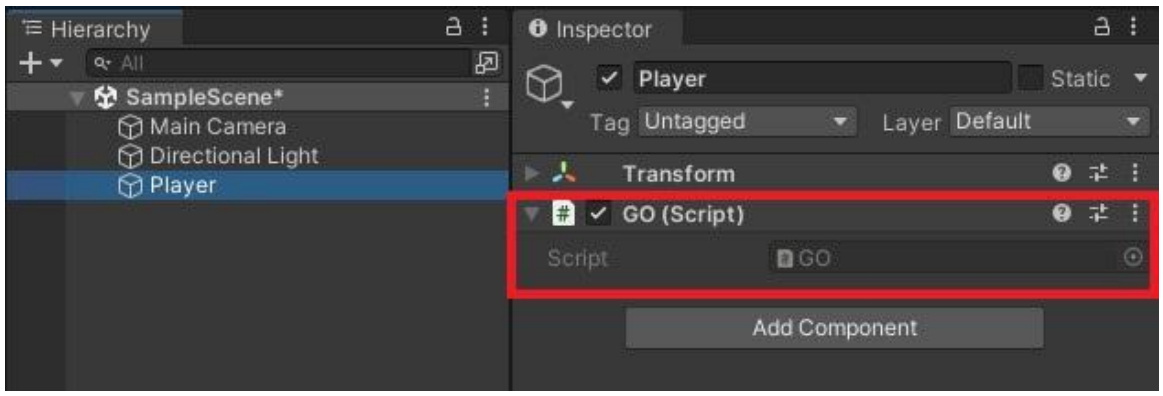


Directional Light:

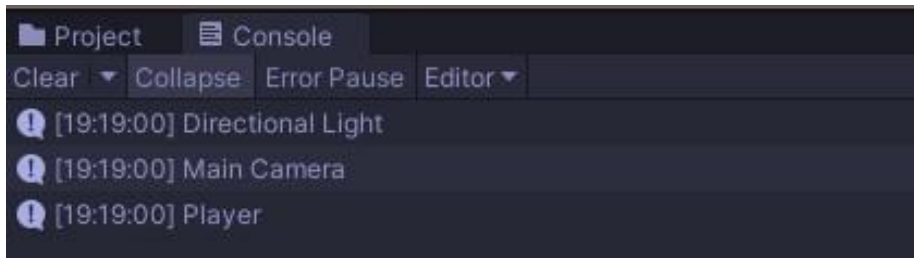


Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

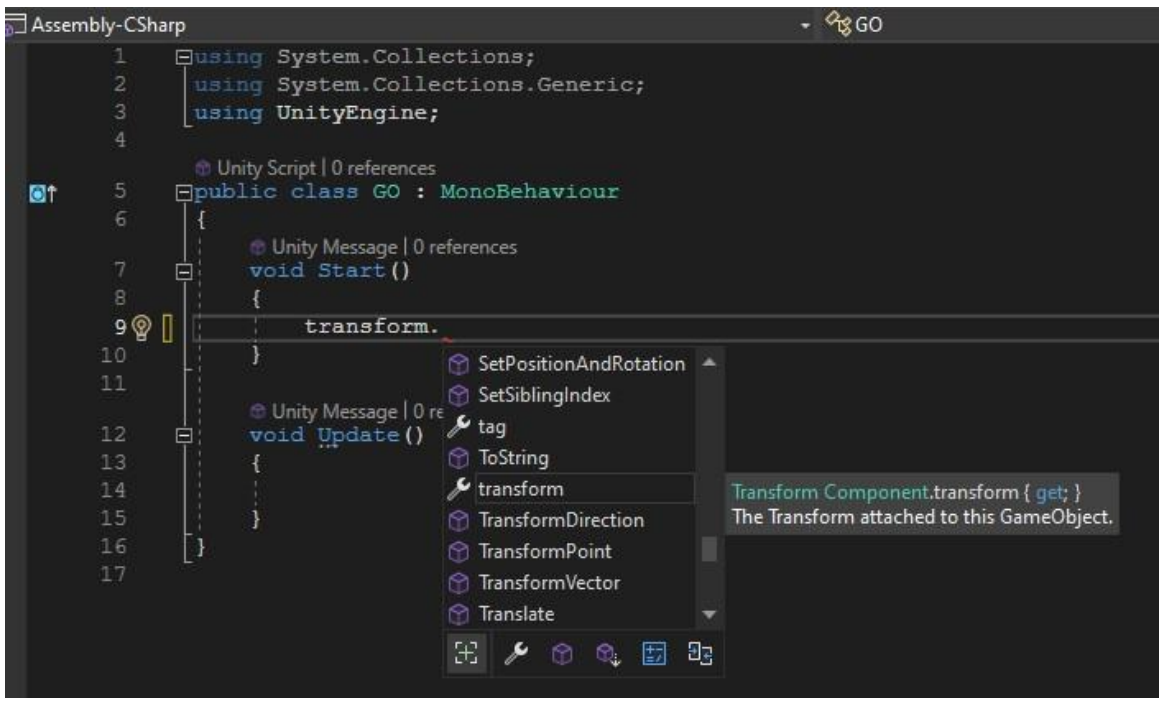
Player:



Si ejecutamos el programa anterior, obtendremos en consola el siguiente resultado:

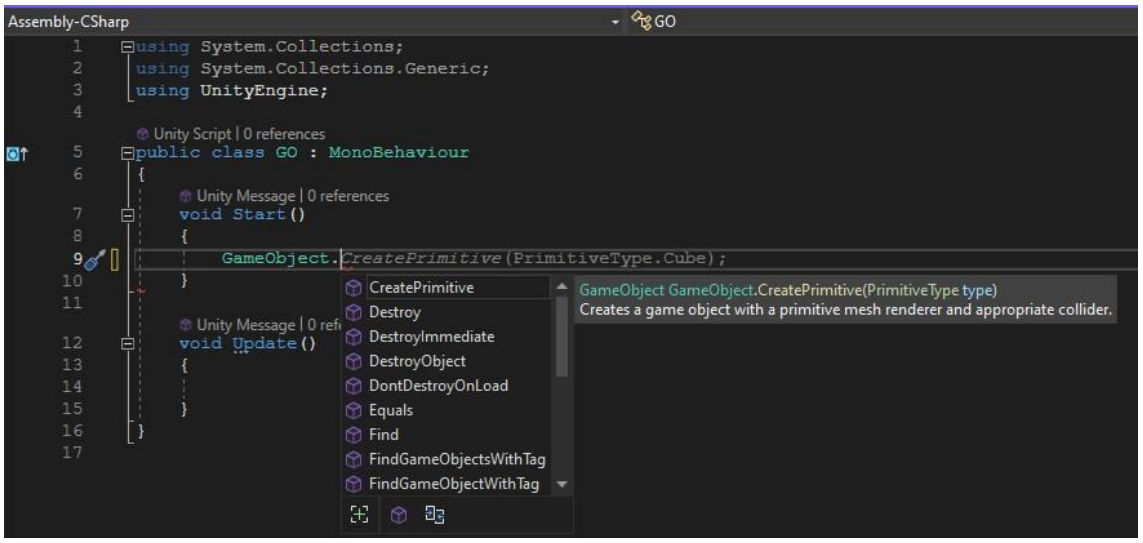


Del mismo modo, si utilizamos **transform**, estamos accediendo a la instancia del componente Transform del gameObject al que está adjunto el script. En la siguiente imagen estamos accediendo a la transform del objeto **Player**.

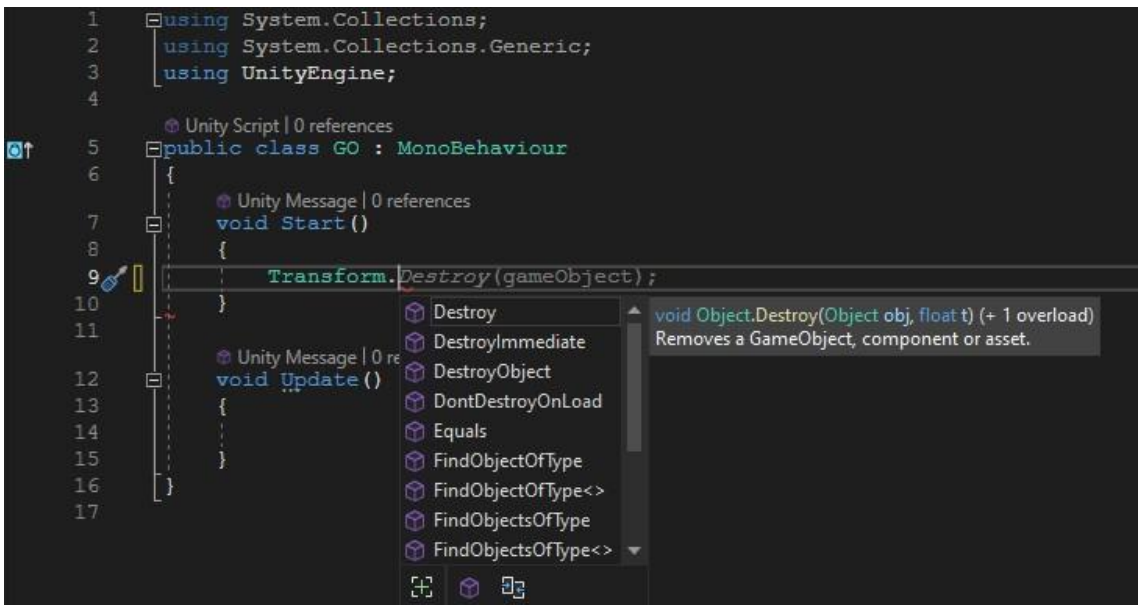


Sin embargo, si utilizamos **GameObject** estamos accediendo a la clase GameObject. La clase nos permite, además de poder crear objetos de tipo GameObject, acceder a sus métodos estáticos.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143



Lo mismo ocurre con **Transform**.



Accediendo a la clase GameObject

La clase GameObject nos permite crear objetos de tipo GameObject.

Vamos a ver un ejemplo:

```
using UnityEngine;

public class GO : MonoBehaviour
{
    [SerializeField] GameObject go;

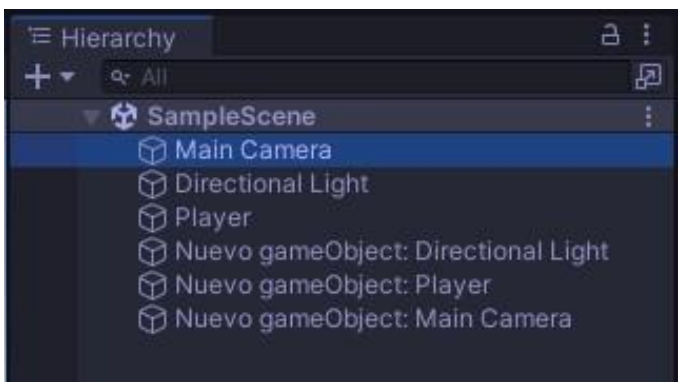
    void Start()
    {
        go = new GameObject();
        go.name = "Nuevo gameObject: " + gameObject.name;
    }
}
```


Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

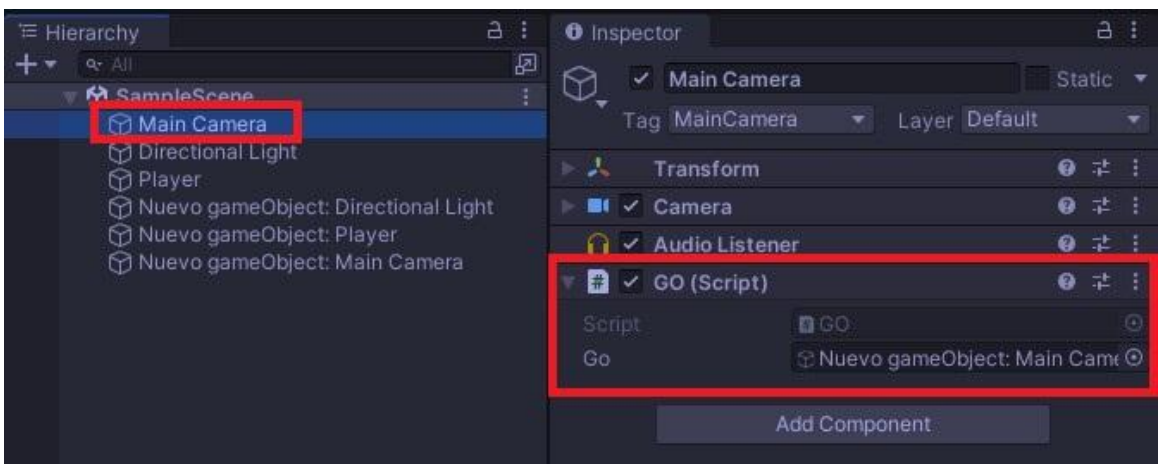
En el siguiente código:

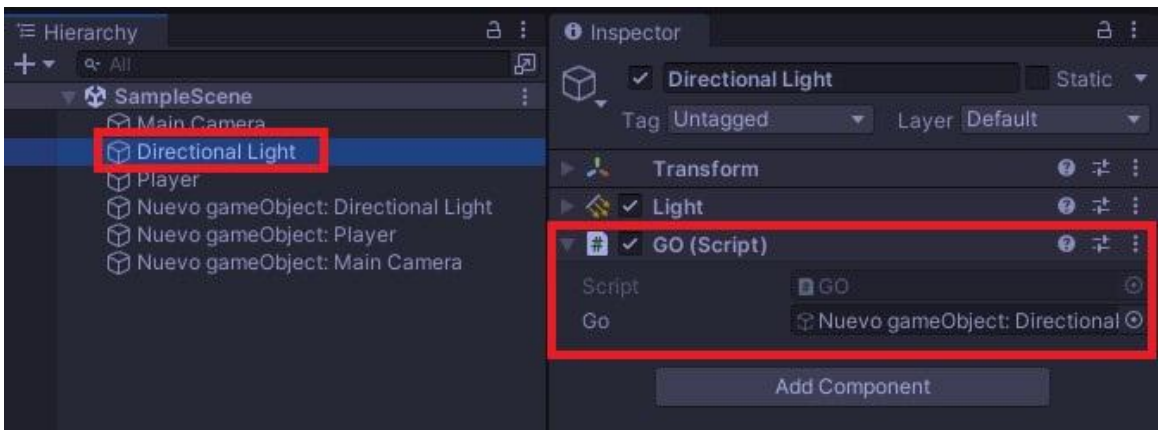
- Declaramos una variable llamada *go* de tipo **GameObject** y la serializamos con **[SerializeField]** para que aparezca en el editor.
- Con el constructor creamos un nuevo gameObject.
- Le asignamos un nombre. En este caso "Nuevo gameObject: " + el nombre del gameObject al cual esté asociado el script.

Como el script estaba asociado a los tres gameObjects de la jerarquía (**Main Camera**, **Directional Light** y **Player**), cuando ejecutemos el programa, tendremos tres nuevos gameObjects vacíos llamados "Nuevo gameObject: " + el nombre correspondiente del gameObject de la jerarquía, tal como indica la siguiente imagen:

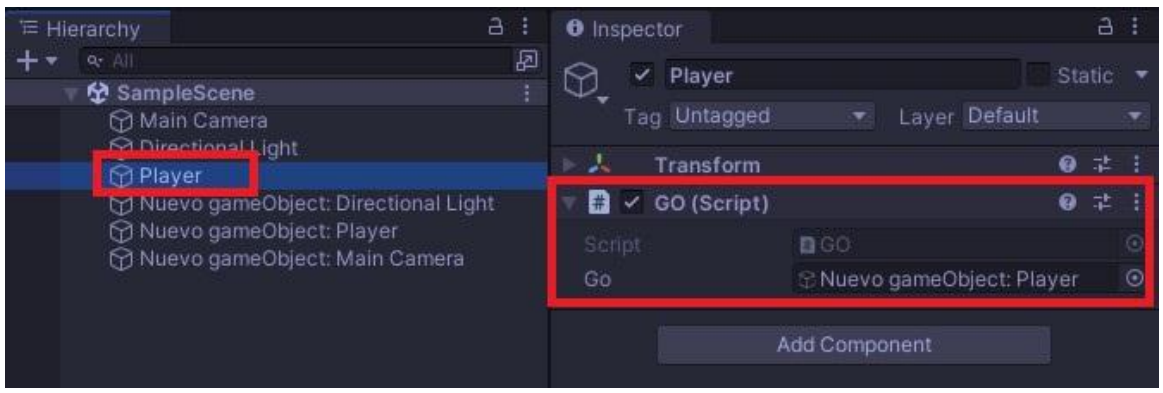


También podemos observar que cada uno de los nuevos gameObjects está asociado al gameObject que lo ha generado:





Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

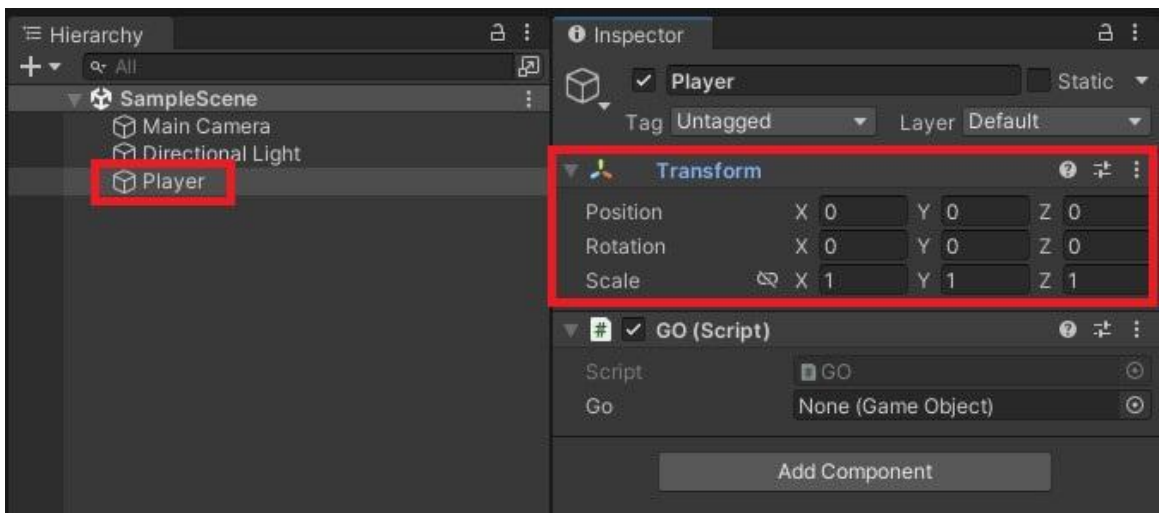


Coordenadas de componente Transform

El componente **Transform** tiene **posición**, **rotación** y **escala**. Cuando estamos trabajando con el componente Transform de un gameObject, posición, rotación y escala son coordenadas locales del gameObject. Y, si en la jerarquía hay un parentesco y el gameObject es hijo de otro gameObject, sus coordenadas serán locales con respecto al padre.

Vamos a verlo con un ejemplo:

- Tenemos el gameObject Player con sus coordenadas locales.



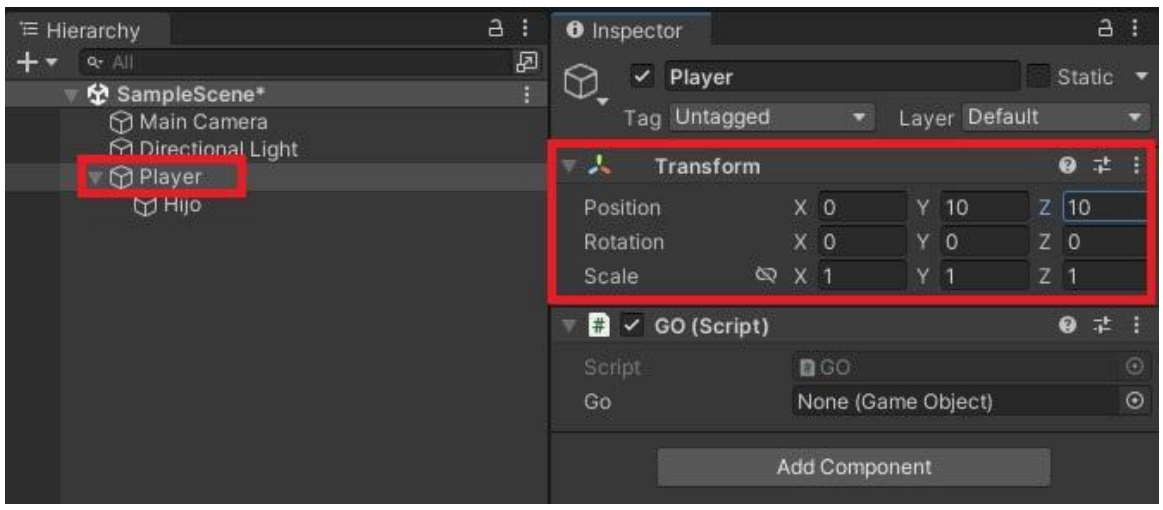
I

- Creamos otro gameObject vacío llamado Hijo y lo hacemos hijo del Player.

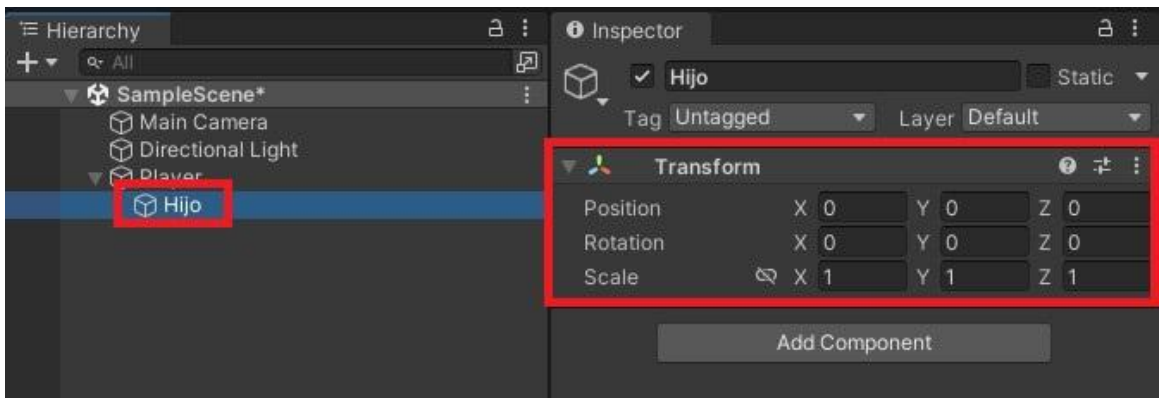


Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

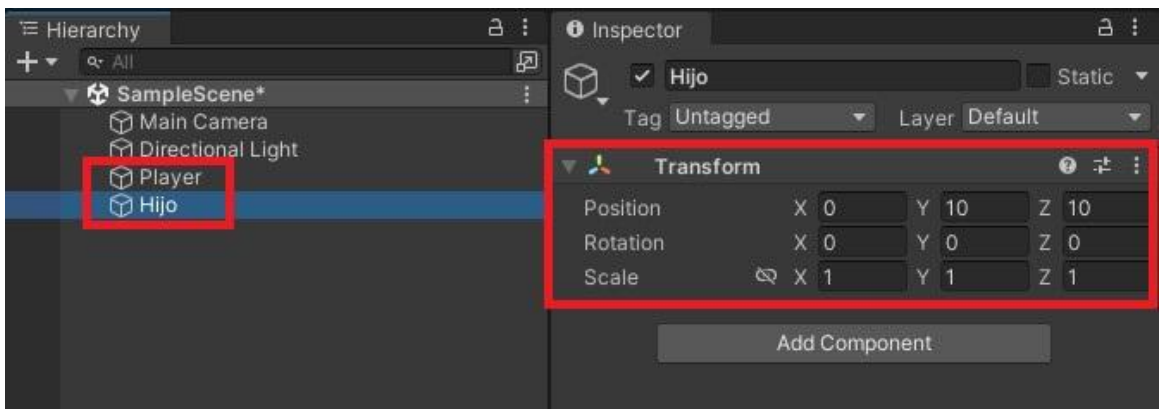
- Movemos el Player 10 unidades en el eje **y** y 10 unidades en el eje **z**.



- El objeto Hijo se ha movido con el padre pero, si lo seleccionamos, vemos que sus coordenadas siguen siendo x:0, y:0 y z:0. Estas coordenadas x:0, y:0 y z:0 son locales al padre dentro de la jerarquía.



- Si rompemos la jerarquía, el editor actualiza la nueva posición del objeto Hijo.



Por último, tienes que saber que no sólo puedes acceder a estar coordenadas a través del editor, sino que también puedes acceder a ellas a través de código dentro de un script.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

```using UnityEngine;

public class GO : MonoBehaviour
{
    [SerializeField] GameObject go;

    void Start()
    {
        go = new GameObject();
        go.name = "Nuevo gameObject: " + gameObject.name;

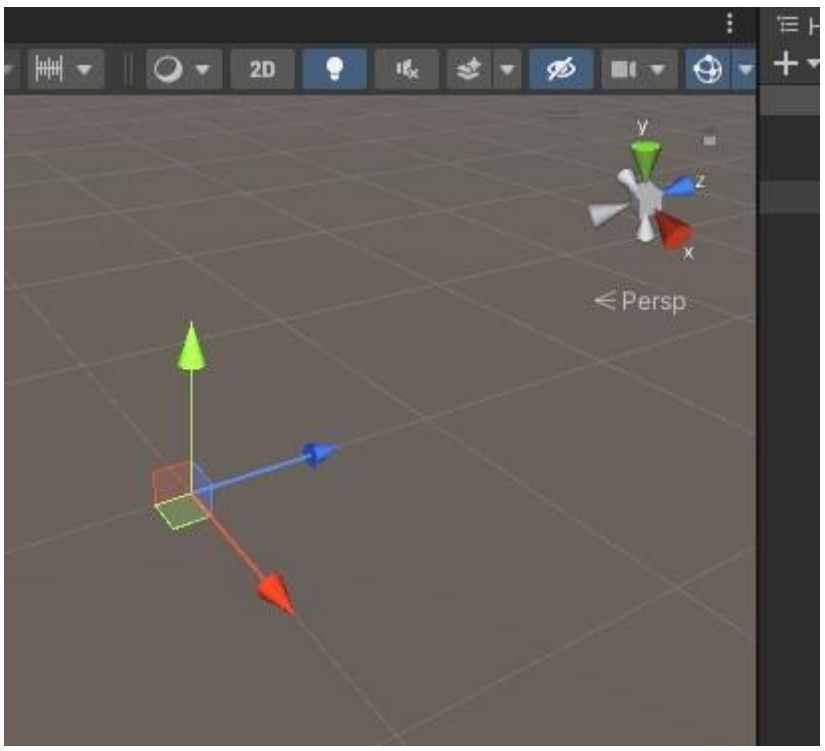
        Debug.Log(go.transform.position.x);
    }
}
```

Dirección de vector Local y valores Vector3

Hemos visto que los componentes **Transform** tienen propiedades como **posición**, **rotación** y **escala**. Cada una de estas propiedades tiene tres valores que vienen representados por decimales. Estos tres valores son puntos en el espacio que definen vectores. Por tanto, cuando accedemos a la posición, rotación o escala de un componente Transform, estamos trabajando con un tipo de dato llamado **Vector3**.

Estos tres vectores tienen definida una dirección, que es hacia dónde apuntan a nivel local con respecto a la orientación global.

Unity nos simplifica el acceso a la dirección y, a través del manejador, podemos ver que:



- La flecha **azul** equivale al **eje z** y se llama **forward**, que indica hacia adelante.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

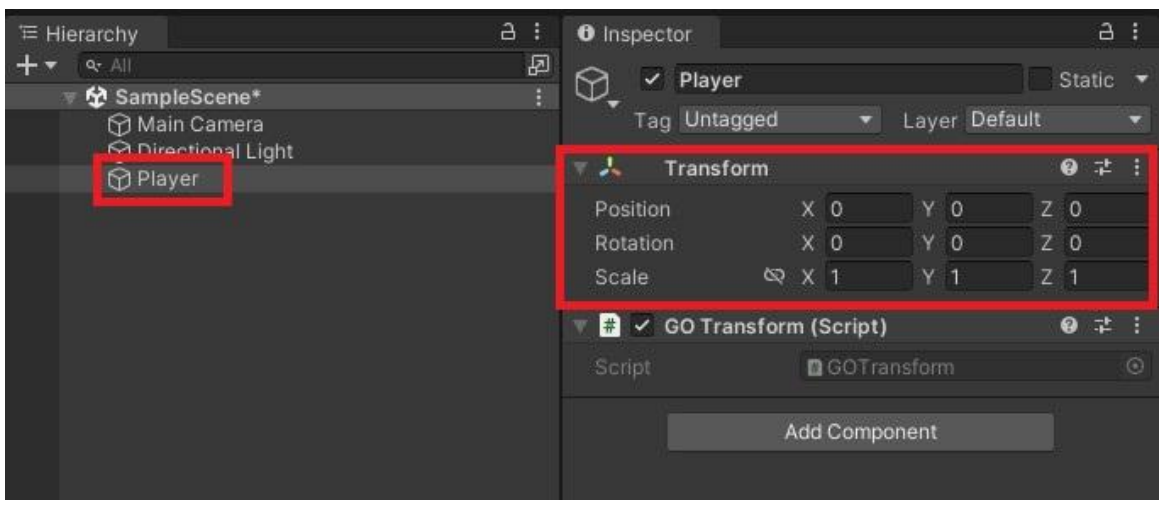
Equivale a: x = 0, y = 0, z = 1.

- La flecha **verde** equivale al **eje y**, y se llama **up**, que indica hacia arriba.
  Equivale a: x = 0, y = 1, z = 0.
- La flecha **roja** equivale al **eje x**, y se llama **right**, que indica hacia la derecha.
  Equivale a: x = 1, y = 0, z = 0.

Los tres valores x, y, z definen la dirección de cada eje.

Vamos a ver un ejemplo:

Tenemos un gameObject llamado **Player**, con el siguiente componente **Transform**:

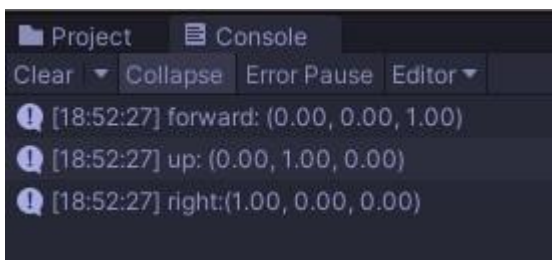


El gameObject tiene un componente llamado **GOTransform** con el siguiente código:

```using UnityEngine;

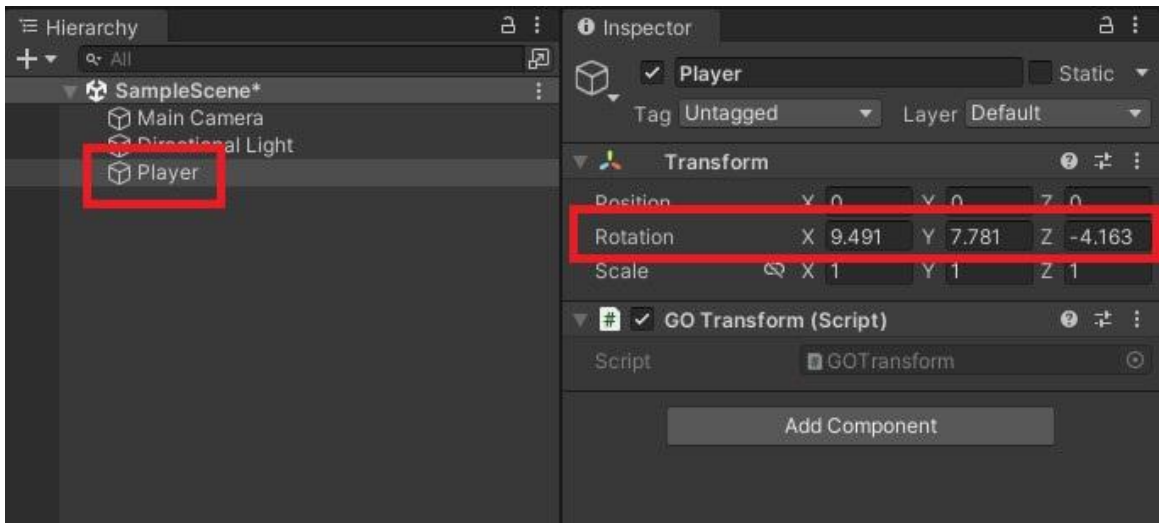
public class GOTransform : MonoBehaviour
{
    void Start()
    {
        Debug.Log("forward: " + transform.forward);
        Debug.Log("up: " + transform.up);
        Debug.Log("right: " + transform.right);
    }
}
```

Si ejecutamos el programa, obtendremos los siguientes mensajes en consola:

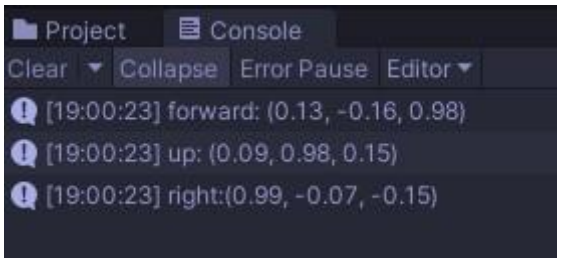


Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Ahora vamos a rotar el gameObject Player al inicio:



Cuando se ejecuta el programa, al estar rotado el gameObject, podemos ver los cambios que hay en las direcciones de los tres ejes.



Cómo crear un Vector3[1]

El constructor de la clase Vector3 tiene 3 sobrecargas.

La primera no utiliza ningún tipo de datos.

```
Vector3 myVector3 = new Vector3();
```

En la segunda, pasamos los valores **x** e **y**. Por defecto se inicializa el valor **z** a cero.

```
Vector3 myVector3 = new Vector3(2.5f, 5.4f);
```

[1] Más adelante veremos en profundidad la clase Vector3

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

En la tercera pasamos los tres valores **x**, **y**, **z**.

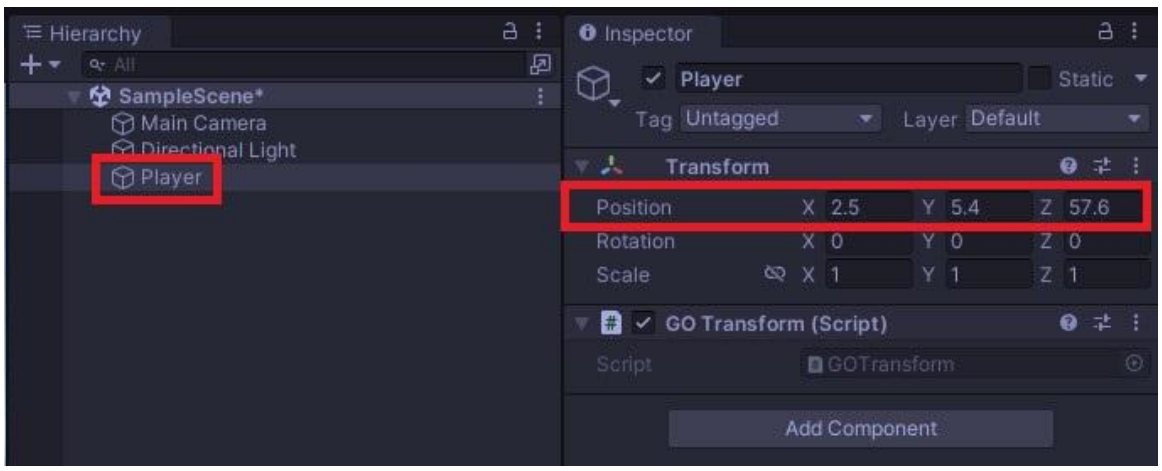
```
Vector3 myVector3 = new Vector3(2.5f, 5.4f, 57.6f);
```

Este nuevo vector es una variable más y podemos darle distintos usos. Por ejemplo podemos asignar ese valor a la posición de la **transform** del **gameObject**.

```
using UnityEngine;

public class GOTransform : MonoBehaviour
{
    void Start()
    {
        Vector3 myVector3 = new Vector3(2.5f, 5.4f, 57.6f);
        transform.position = myVector3;
    }
}
```

Si ejecutamos el programa, el gameObject **Player** cambiará su posición al definido por la variable **myVector3**.



Transform Local vs Global

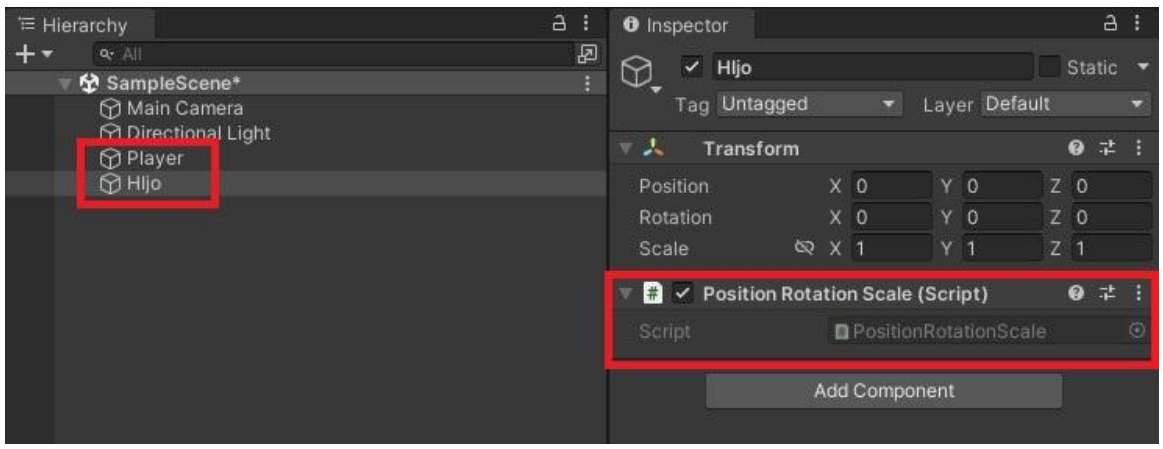
Vamos a ver la diferencia entre los valores locales y globales para cada una de las propiedades del componente Transform (posición, rotación y escala).

Si un gameObject no tiene ningún parentesco, los valores locales y globales serán iguales. Habrá diferencia cuando el gameObject tenga un parentesco.

Vamos a verlo con un ejemplo:

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Tenemos un gameObject llamado **Player** y un gameObject llamado **Hijo**, con un componente llamado **PositionRotationScale**. De momento no hay ningún parentesco entre los dos gameObjects.



El código del componente **PositionRotationScale** es:

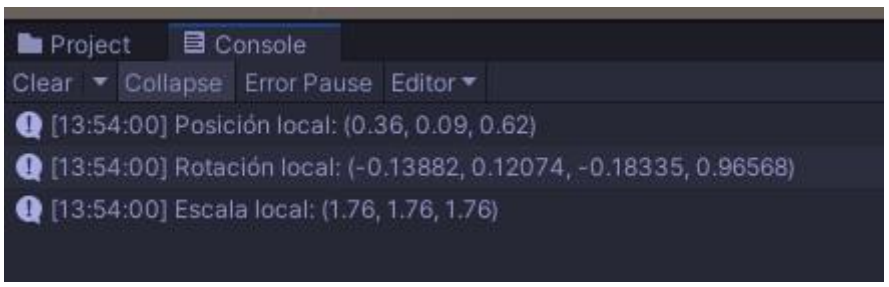
```
using UnityEngine;

public class PositionRotationScale : MonoBehaviour
{
    void Update()
    {
        Debug.Log("Posición local: " + transform.localPosition);
        Debug.Log("Rotación local: " + transform.localRotation);
        Debug.Log("Escala local: " + transform.localScale);
    }
}
```

Si ejecutamos el programa obtenemos estos valores.

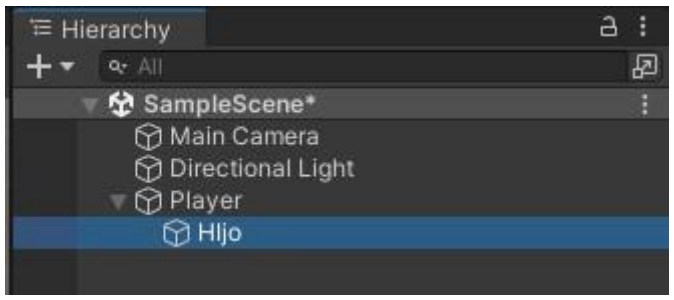


Si movemos, rotamos o cambiamos la escala del gameObject, estos valores cambian.



Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

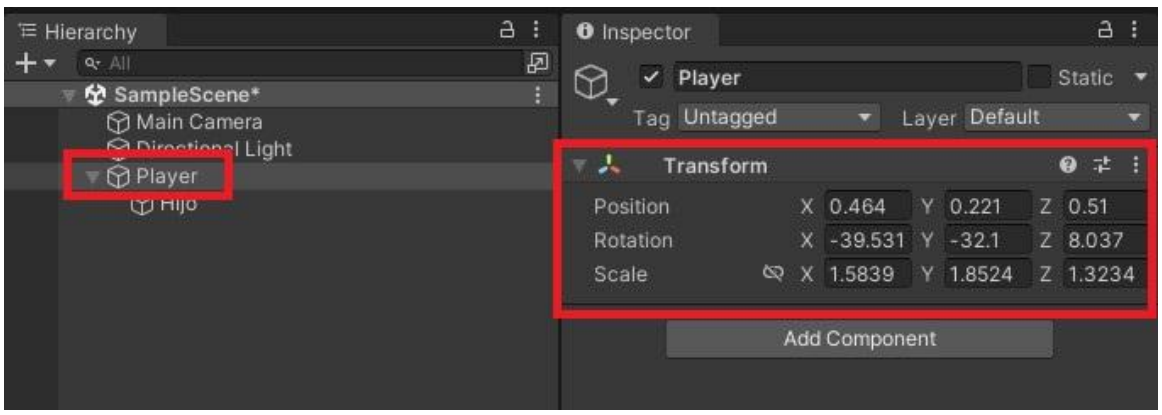
Ahora el gameObject **Hijo** va a ser hijo del gameObject **Player**.



Ejecutamos de nuevo el programa y obtenemos los siguientes valores:



Vamos a modificar la posición, rotación y escala del gameObject **Player**.



Si ejecutamos de nuevo el programa, veremos que la posición, rotación y escala locales del gameObject Hijo permanece con los mismos valores.



Esto es debido a que son los valores locales respecto al padre, que en este caso es el gameObject **Player**.

Ahora le vamos a indicar al componente **PositionRotationScale** que muestre también los valores globales.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

```cs
using UnityEngine;

public class PositionRotationScale : MonoBehaviour
{
    void Update()
    {
        Debug.Log("Posición local: " + transform.localPosition);
        Debug.Log("Rotación local: " + transform.localRotation);
        Debug.Log("Escala local: " + transform.localScale);
        Debug.Log("");
        Debug.Log("Posición global: " + transform.position);
        Debug.Log("Rotación global: " + transform.rotation);
        Debug.Log("Escala global: " + transform.lossyScale);
    }
}
```

Si ejecutamos de nuevo el programa, vemos la diferencia entre los valores locales y globales de **posición**, **rotación** y **escala** del gameObject **Hijo**.



Los valores globales son respecto al espacio global, mientras que los locales son respecto al padre.

Transform Position

Vamos a ver diferentes operativas que podemos realizar con la propiedad **position** del componente **Transform**.

- Acceder a la posición de la instancia.

```cs
using UnityEngine;

public class Position : MonoBehaviour
{
    void Start()
    {
        Debug.Log(transform.position);
    }
}
```


Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

- Utilizar la posición de la instancia como dato de una variable.

```
using UnityEngine;

public class Position : MonoBehaviour
{
    void Start()
    {
        Vector3 initialposition = transform.position;
    }
}
```

- Establecer una nueva posición creando una nueva variable de tipo **Vector3**.

```
using UnityEngine;

public class Position : MonoBehaviour
{
    void Start()
    {
        transform.position = new Vector3(50f, 25.4f, 3.9f);
    }
}
```

- Establecer una nueva posición creando una nueva variable de tipo **Vector3**. Para crear esa variable de tipo Vector3 se pueden utilizar variables de tipo float o componentes de otra variable de tipo Vector3.

```
using UnityEngine;

public class Position : MonoBehaviour
{
    float x = 15f;
    float z = -98.5f;

    Vector3 initialPosition;

    void Start()
    {
        initialPosition = transform.position;
        transform.position = new Vector3(x, initialPosition.y, z);
    }
}
```


Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

- Establecer una nueva posición utilizando una nueva variable de tipo **Vector3** ya creada.

```
using UnityEngine;

public class Position : MonoBehaviour
{
    Vector3 initialPosition;

    void Start()
    {
        initialPosition = transform.position;
    }

    void Update()
    {
        transform.position = initialPosition;
    }
}
```

- Utilizar componentes de tipo **Transform** de otros gameObjects.

```
using UnityEngine;

public class Position : MonoBehaviour
{
    GameObject wall;
    GameObject floor;

    void Start()
    {
        transform.position = new Vector3(wall.transform.position.x,
                                         floor.transform.position.y,
                                         0f);
    }
}
```


Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

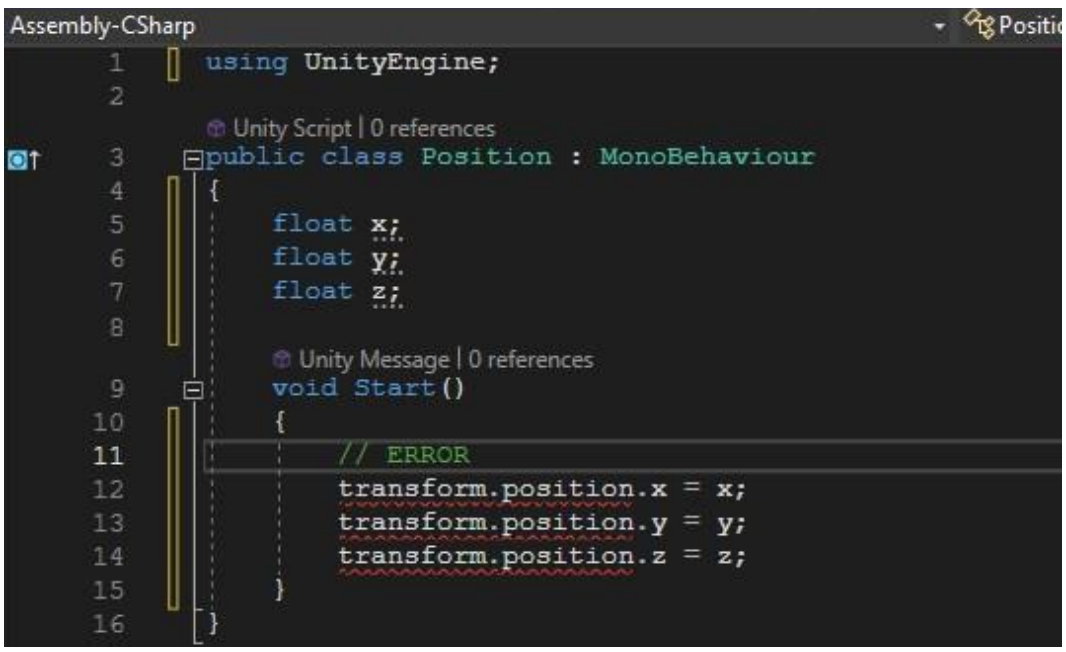
- Acceder a los componentes de la propiedad **position** del componente **Transform** y almacenarlos en variables de tipo **float**.

```
using UnityEngine;

public class Position : MonoBehaviour
{
    float x;
    float y;
    float z;

    void Start()
    {
        x = transform.position.x;
        y = transform.position.y;
        z = transform.position.z;
    }
}
```

Ten en cuenta que no vas a poder hacer la operativa contraria:



Si intentamos asignar por separado un valor a cada uno de los componentes de la propiedad **position** del componente **Transform**, vamos a tener un error en Visual Studio. Esto es porque la propiedad **position** está formada por una variable de tipo **Vector3**, NO por tres variables de tipo **float** por separado. Es la variable de tipo **Vector3** la que está formada por tres variables de tipo float.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

- Modificar un componente sin crear una variable de tipo Vector3.

```
using UnityEngine;

public class Position : MonoBehaviour
{
    void Start()
    {
        transform.position.Set(15.5f, 65.2f, 47.47f);
    }
}
```

La función **Set** rellena los valores del **Vector3** de la propiedad **position** con los valores pasados como parámetro.

Transform Scale

Todas las operativas que hemos visto en el punto anterior con la propiedad **position** del componente **Transform** son también válidas para su propiedad **localScale**.

La propiedad **loosyScale** es de solo lectura, por tanto:

- Si queremos leer la escala de un gameObject, podremos trabajar con **localScale** y **loosyScale**.
- Si queremos modificar la escala de un gameObject, tendremos que trabajar con **localScale**.

Cómo trabaja Unity las rotaciones

La rotación se trabaja de forma diferente porque, aunque se trabaja con vectores, el cálculo de la rotación se hace a través de **ángulos euler** y a través de una ecuación matemática llamada **Quaternion**.

La propiedad de las rotaciones equivalente a las propiedades **position** y **scale** que hemos visto en los puntos anteriores, es la propiedad **eulerAngles** y su correspondiente valor local, **localEulerAngles**.

Cuando indicamos la rotación en el editor, estamos informando una variable de tipo **Vector3**, pero Unity lo está pasando a una clase que se llama **Euler**. Esta clase convierte el valor de ese vector en ángulos. Unity utiliza internamente un **Quaternion** para procesar esos ángulos y generar la rotación.

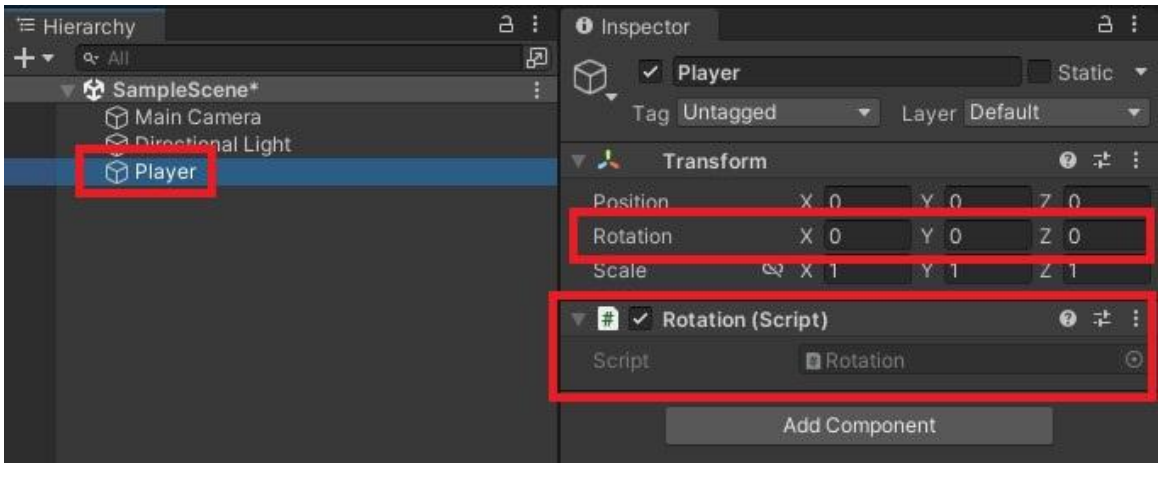
El **Quaternion** está formado por cuatro parámetros que, en ningún caso, representan directamente el ángulo de la rotación. Están basados en números complejos y no son fáciles de entender intuitivamente.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Unity utiliza Quaternions porque es una forma de garantizar que no se produce lo que se denomina **gimbal lock**[2].

Vamos a ver un ejemplo:

Tenemos un gameObject llamado **Player**, que no está rotado y que tiene un componente llamado **Rotation**.

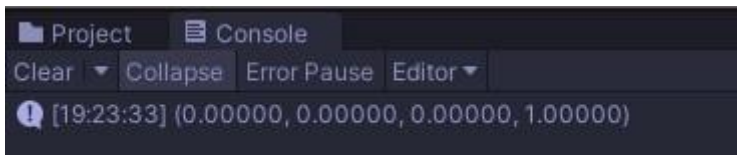


El código del componente **Rotation** es:

```
using UnityEngine;

public class Rotation : MonoBehaviour
{
    void Start()
    {
        Debug.Log(transform.rotation);
    }
}
```

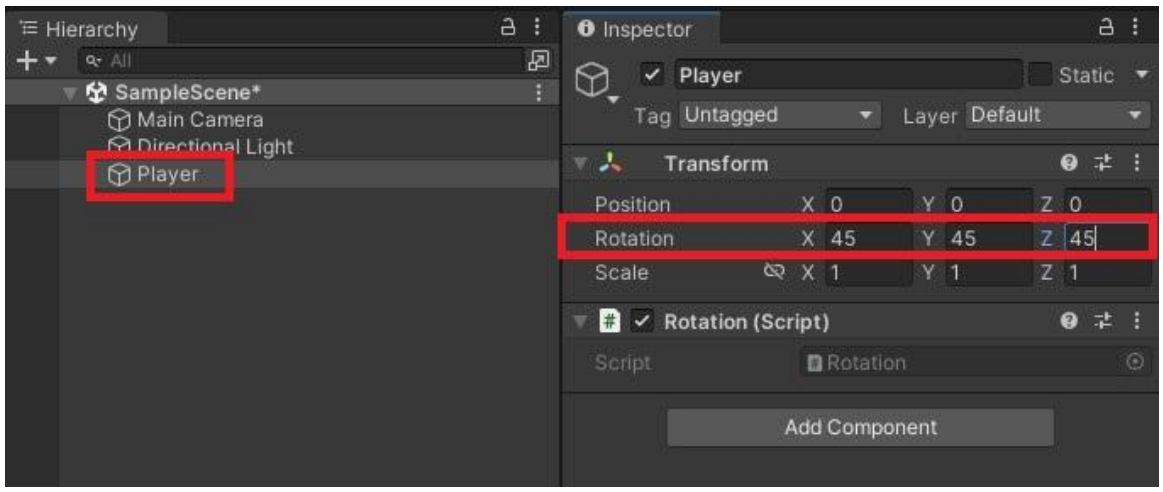
Si ejecutamos este programa, el mensaje en consola que obtenemos es:



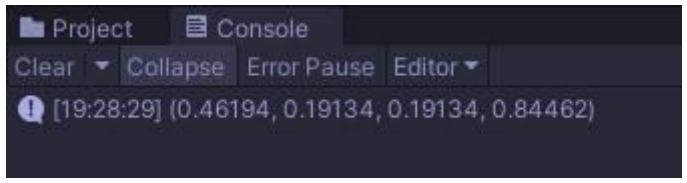
[2] **Gimbal lock** es la pérdida de un grado de libertad en un mecanismo tridimensional de tres cardanes que ocurre cuando los ejes de dos de los tres cardanes son conducidos a una configuración paralela, "bloqueando" el sistema en rotación en un espacio bidimensional degenerado.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Ahora vamos a rotar el gameObject Player:



El resultado que obtendremos ahora será:



Estos cuatro valores son con los que se construye el **Quaternion.** Podemos observar que no coinciden con los valores de la rotación que habíamos indicado en el editor.

Vamos a modificar el código del componente **Rotation** del ejemplo anterior:

```
using UnityEngine;

public class Rotation : MonoBehaviour
{
    void Start()
    {
        Debug.Log(transform.rotation);
        Debug.Log(transform.eulerAngles);
    }
}
```

Si ejecutamos ahora el programa, los mensajes que se mostrará en consola son:



Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

El primer mensaje se corresponde con el **Quaternion** que utiliza Unity para generar la rotación. El segundo mensaje se corresponde con la propiedad **eulerAngles**. Podemos ver que estos valores sí que se corresponden con los valores de la rotación introducidos en el editor.

Transform Rotation y EulerAngles

Vamos a ver los ejemplos vistos en el punto "Transform Position" aplicados a la propiedad **rotation**.

- Acceder a la rotación de la instancia.

```
using UnityEngine;

public class Rotation : MonoBehaviour
{
    void Start()
    {
        Debug.Log(transform.eulerAngles);
    }
}
```

- Utilizar la rotación de la instancia como dato de una variable.

```
using UnityEngine;

public class Rotation : MonoBehaviour
{
    void Start()
    {
        Vector3 myRotation = transform.eulerAngles;
    }
}
```

- Establecer una nueva rotación creando una nueva variable de tipo **Vector3**.

```
using UnityEngine;

public class Rotation : MonoBehaviour
{
    void Start()
    {
        transform.eulerAngles = new Vector3(5f, 4.5f, 78f);
    }
}
```


Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

- Establecer una nueva rotación creando una nueva variable de tipo **Vector3**. Para crear esa variable de tipo Vector3 se pueden utilizar variables de tipo float o componentes de otra variable de tipo Vector3.

```
using UnityEngine;

public class Rotation : MonoBehaviour
{
    float x = 5.7f;
    float z = 19.68f;

    Vector3 initialRotation;

    void Start()
    {
        initialRotation = transform.eulerAngles;
        transform.eulerAngles = new Vector3(x, initialRotation.y, z);
    }
}
```

- Establecer una nueva posición utilizando una nueva variable de tipo **Vector3** ya creada.

```
using UnityEngine;

public class Rotation : MonoBehaviour
{
    Vector3 initialRotation;

    void Start()
    {
        initialRotation = transform.eulerAngles;
    }

    void Update()
    {
        transform.eulerAngles = initialRotation;
    }
}
```


Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

- Utilizar componentes de tipo **Transform** de otros gameObjects.

```
using UnityEngine;

public class Rotation : MonoBehaviour
{
    GameObject wall;
    GameObject floor;

    void Start()
    {
        transform.eulerAngles = new Vector3(wall.transform.eulerAngles.x,
                                            floor.transform.eulerAngles.y,
                                            0);
    }
}
```

- Acceder a los componentes de la propiedad **position** del componente **Transform** y almacenarlos en variables de tipo **float**.

```
using UnityEngine;

public class Rotation : MonoBehaviour
{
    float x;
    float y;
    float z;

    void Start()
    {
        x = transform.eulerAngles.x;
        y = transform.eulerAngles.y;
        z = transform.eulerAngles.z;
    }
}
```

- Modificar un componente sin crear una variable de tipo Vector3.

```
using UnityEngine;

public class Rotation : MonoBehaviour
{
    void Start()
    {
        transform.eulerAngles.Set(15f, 0f, 0f);
    }
}
```


Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Si necesitamos calcular una rotación en tiempo real es más conveniente utilizar la clase **Quaternion**, en vez de pasar los **ángulos euler** a la **transform.eulerAngles**, para evitar el gimbal lock.

Para establecer la rotación utilizando la clase **Quaternion**:

```
using UnityEngine;

public class Rotation : MonoBehaviour
{
    void Start()
    {
        transform.rotation = Quaternion.Euler(5f, 4.5f, 78f);
    }
}
```

Quaternion.Identity implica 0 rotación en x, 0 rotación en y y 0 rotación en z.

Para recordar ...

- Todos los gameObjects tienen dos componentes por defecto: un componente de tipo **GameObject** y un componente de tipo **Transform**
- Si en un script utilizamos **gameObject**, estamos accediendo a la instancia que se crea del gameObject al que está adjunto el script
- Si en un script utilizamos **transform**, estamos accediendo a la instancia del componente Transform del gameObject al que está adjunto el script
- Si utilizamos **GameObject** estamos accediendo a la clase GameObject. Podremos crear objetos de tipo GameObject y acceder a sus métodos estáticos
- Si utilizamos **Transform** estamos accediendo a la clase Transform. Podremos crear objetos de tipo Transform y acceder a sus métodos estáticos
- El componente Transform tiene posición, rotación y escala
- Cuando accedemos a la posición, rotación o escala de un componente de tipo Transform, estamos trabajando con un tipo de dato llamado **Vector3**
- Los valores globales de los componentes de posición, rotación y escala son respecto al espacio global, mientras que los locales son respecto al padre si hay parentesco
- El cálculo de la rotación se hace a través de **ángulos euler** y a través de una ecuación matemática llamada **Quaternion**
- El **Quaternion** está formado por cuatro parámetros que, en ningún caso, representan directamente el ángulo de la rotación
- **Quaternion.Identity** implica 0 rotación en x, 0 rotación en y y 0 rotación en z

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

OnGUI y OnDrawGizmos

Introducción

En esta sección vamos a ver los métodos OnGUI y OnDrawGizmos, que nos van a permitir hacer debugging visual tanto en la ventana de Juego (Game) como en la ventana de escena (Scene).

Trabajando con OnGUI GUI y GUI Layout

OnGUI está contenido dentro de MonoBehaviour.

El método **OnGUI** está contenido dentro del orden de ejecución de los componentes. Este método nos permite dibujar botones y otros elementos de interfaz de usuario en la ventana de Juego (Game) y utilizarlos. Hay varias clases que empiezan por GUI y que pueden ser utilizadas para este propósito dentro del método OnGUI. Nosotros vamos a ver **GUI** y **GUILayout**.

```
void OnGUI()
{
    // código con las clases GUI, GUILayout
    // o cualquiera que empiece por GUI
}
```

Hay una diferencia entre GUI y GUILayout respecto al posicionamiento de los elementos:

- La clase GUI utiliza un posicionamiento manual.
- La clase GUILayout utiliza un posicionamiento automático.

Este es el sistema antiguo que se utilizaba para crear interfaces de usuario para los juegos dentro en Unity, pero es un sistema que consume muchos recursos porque se llama **varias veces por frame**. Actualmente se utiliza para hacer debugging en tiempo real sin tener que montar un sistema de interfaz de usuario y poder dibujar textos, botones, campos de texto... de forma más rápida.

Botones y textos con GUILayout y GUI

Vamos a ver con un ejemplo cómo se crean elementos con **GUI** y **GUILayout**.

En este ejemplo vamos a crear un cubo y vamos a cambiar su posición a través de **textfields** (donde indicaremos los nuevos valores de x, y y z) y un **botón**, que será el que ejecute la orden.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

```
using UnityEngine;

public class OnGUIScript : MonoBehaviour
{
    GameObject cube;
    Vector3 pos;

    void Start()
    {
        cube = GameObject.CreatePrimitive(PrimitiveType.Cube);
    }

    void OnGUI()
    {
        GUILayout.BeginHorizontal();
        GUILayout.Label("Posición en X");
        pos.x =
            float.Parse(GUILayout.TextField(pos.x.ToString(), 10));
        GUILayout.EndHorizontal();

        GUILayout.BeginHorizontal();
        GUILayout.Label("Posición en Y");
        pos.y =
            float.Parse(GUILayout.TextField(pos.y.ToString(), 10));
        GUILayout.EndHorizontal();

        GUILayout.BeginHorizontal();
        GUILayout.Label("Posición en Z");
        pos.z =
            float.Parse(GUILayout.TextField(pos.z.ToString(), 10));
        GUILayout.EndHorizontal();

        if (GUILayout.Button("Aplicar posición"))
        {
            cube.transform.position = pos;
        }

        if(GUI.Button(new Rect(0,110,150,30), "Saludar"))
        {
            Debug.Log("Hola, has presionado un botón");
        }
    }
}
```

En este código se declara una variable de tipo **GameObject** llamada *cube* y una variable de tipo **Vector3** llamada *pos*.

En el método **Start**:

- Se crea un cubo con **GameObject.CreatePrimitive** y se almacena en la variable *cube*.

En el método **OnGUI**:

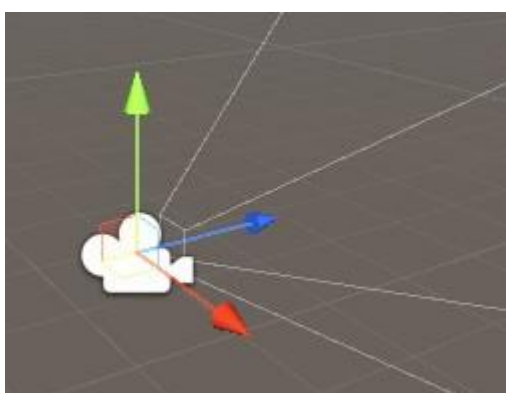
- Se crea un **label** con el texto “posición en X”.
- Se crea un **textfield** que muestra el valor de *pos.x*, y en el que el usuario podrá introducir un valor. El valor introducido se vuelve a guardar en *pos.x*.
- Tanto el label como el textfield se sitúan de forma horizontal porque están contenidos dentro de las instrucciones **GUILayout.BeginHorizontal()** y **GUILayout.EndHorizontal()**.
- Se repiten las mismas instrucciones para las posiciones **y** y **z**.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

- Se crea un botón con el posicionamiento automático (**GUILayout**) con el texto “Aplicar posición”. Si se pulsa, se asignará la nueva posición introducida por el usuario al gameObject **cube**.
- Se crea un botón con el posicionamiento manual (**GUI**) con el texto “Saludar”. Si se pulsa, se mostrará un mensaje por pantalla. Para crear este botón, se tiene que introducir de forma manual la posición en x, la posición en y, el ancho y el alto.

Trabajando con OnDrawGizmos

Un gizmo es una representación gráfica en la ventana de escena de los componentes. Por ejemplo, vamos a ver el gizmo de la cámara.

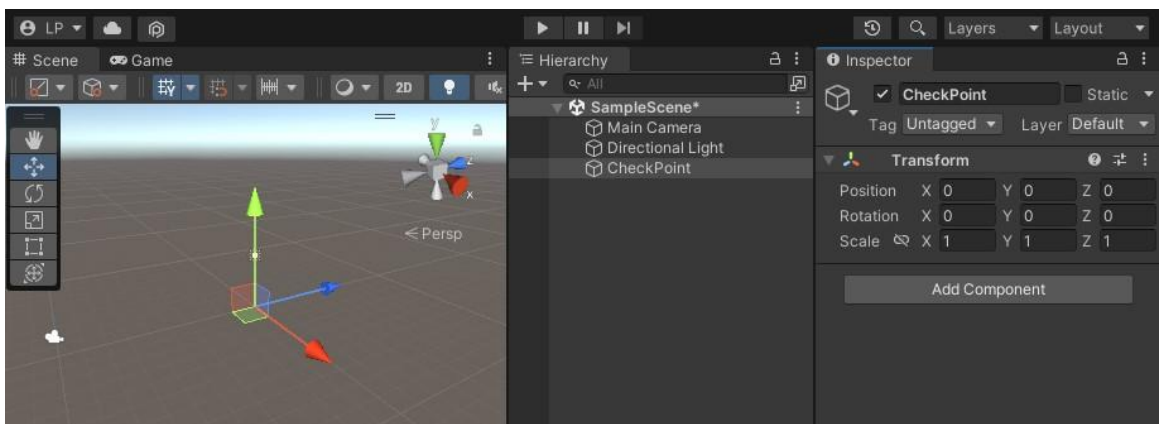


OnDrawGizmos es una función que nos va a permitir hacer debugging visual de los componentes en la ventana de escena.

```
void OnDrawGizmos()
{
    // código
}
```

OnDrawGizmos y OnDrawGizmosSelected

Vamos a ver cómo funciona **OnDrawGizmos** con un ejemplo.
Tenemos un gameObject vacío llamado *CheckPoint*.



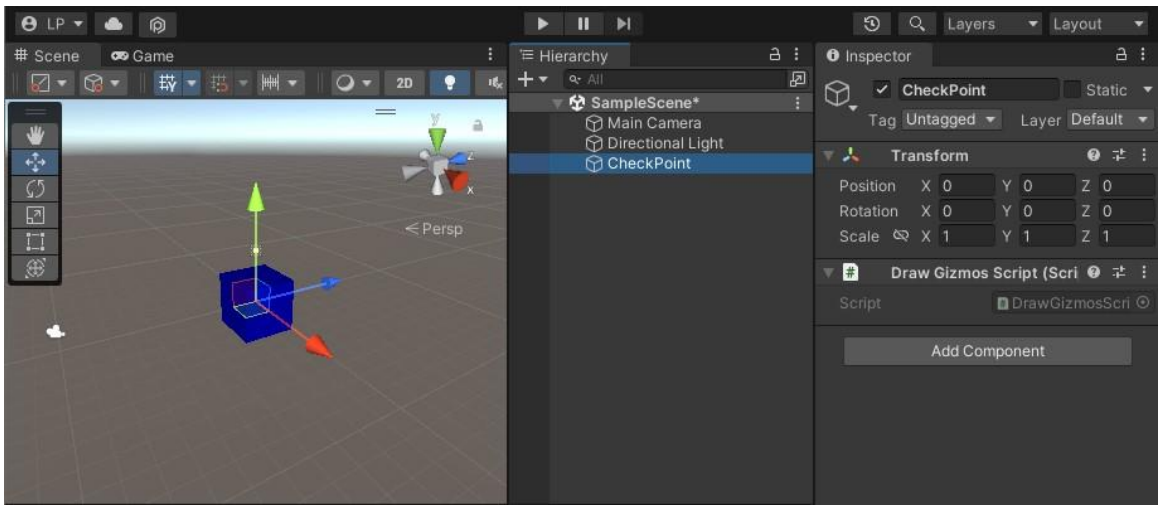
Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Creamos un script llamado **DrawGizmosScript** con el siguiente código:

```using UnityEngine;

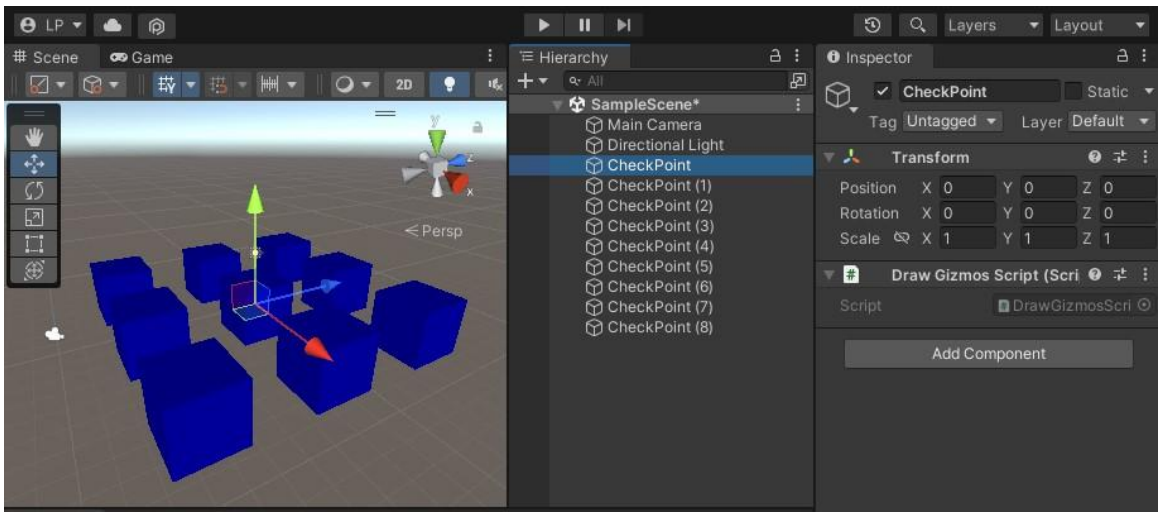
public class DrawGizmosScript : MonoBehaviour
{
    void OnDrawGizmos()
    {
        Gizmos.color = Color.blue;
        Gizmos.DrawCube(transform.position, new Vector3(3f, 3f, 3f));
    }
}
```

Este código dibujará un gizmo con forma de cubo, en la posición y del tamaño pasado como parámetro. Esto ocurrirá cuando asignemos el script al gameObject *CheckPoint*, sin necesidad de ejecutar el programa.



Esto es útil para saber visualmente dónde están situados ciertos gameObjects.

Si tenemos el script en varios gameObjects, puede resultar un poco confuso saber cuál de ellos está seleccionado.



En estos casos puede ser útil la función **OnDrawGizmosSelected**, que dibujará el gizmo sólo en el gameObject seleccionado.

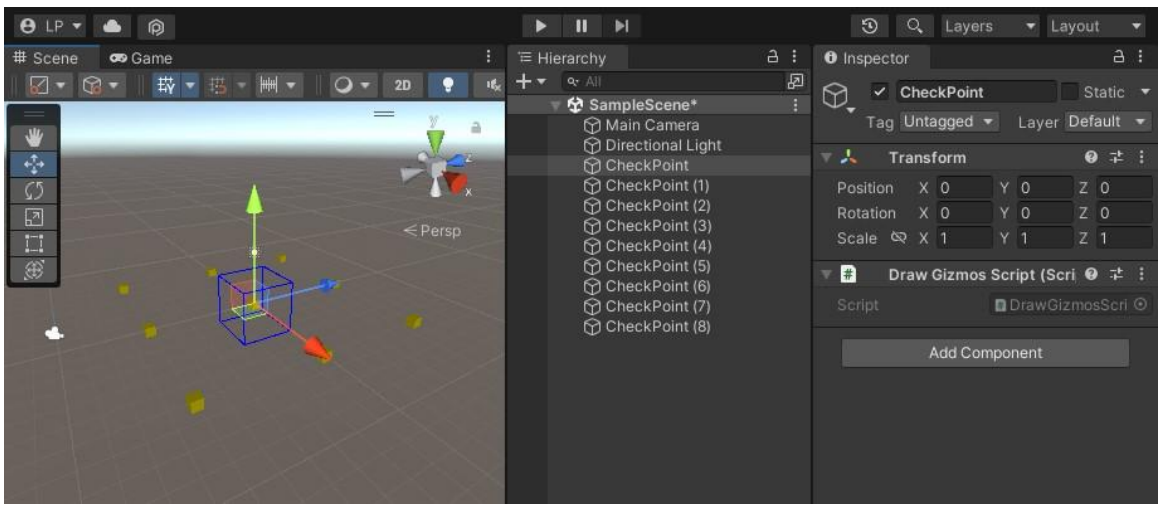
Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

```
using UnityEngine;

public class DrawGizmosScript : MonoBehaviour
{
    void OnDrawGizmos()
    {
        Gizmos.color = Color.yellow;
        Gizmos.DrawCube(transform.position,
                                    new Vector3(0.5f, 0.5f, 0.5f));
    }

    void OnDrawGizmosSelected()
    {
        Gizmos.color = Color.blue;
        Gizmos.DrawWireCube(transform.position,
                                    new Vector3(3f, 3f, 3f));
    }
}
```

Ahora se dibujará un cubo de color amarillo en todos los gameObjects, en la posición y tamaño indicados por los parámetros, y un cubo de malla de alambre en el gameObject seleccionado.

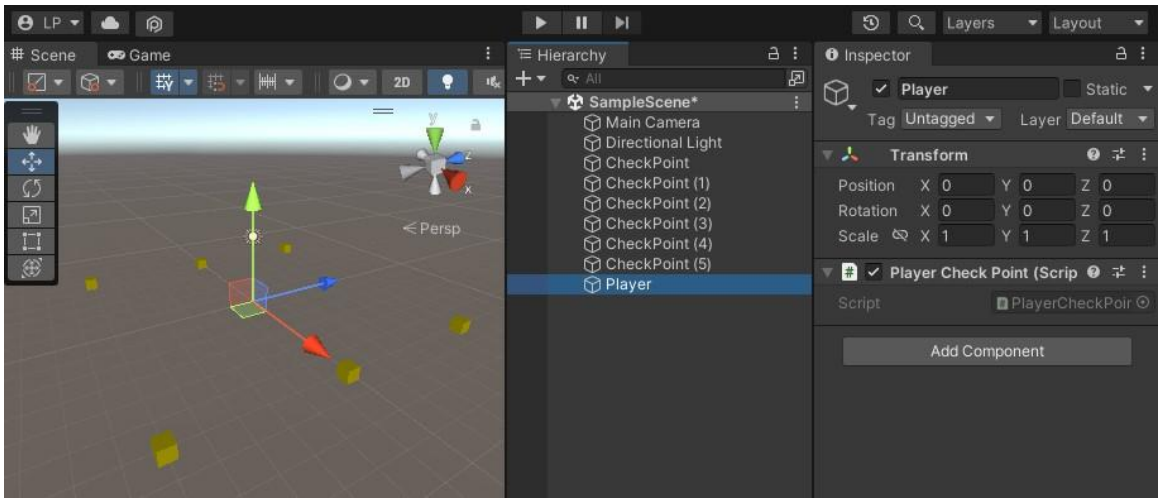


OnDrawGizmos y OnDrawGizmosSelected en tiempo de ejecución

Vamos a seguir trabajando con **OnDrawGizmos** y **OnDrawSelected**. Ahora vamos a ver, continuando con el ejemplo anterior, cómo utilizarlos en tiempo de ejecución.

Vamos a crear un gameObject vacío llamado *Player*, y vamos a hacer que se dibuje una línea al checkPoint más cercano. Para ello, vamos a crear un script llamado **PlayerCheckPoint** y se lo vamos a asignar al gameObject *Player*.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143



El script tendrá el siguiente código:

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

public class PlayerCheckPoint : MonoBehaviour
{
    List<DrawGizmosScript> checkpointList;

    void Start()
    {
        checkpointList =
            new List<DrawGizmosScript>(FindObjectsOfType<DrawGizmosScript>());
    }

    void OnDrawGizmos()
    {
        Gizmos.color = Color.green;
        Gizmos.DrawSphere(transform.position, 1);

        if (checkpointList.Count == 0 || checkpointList == null)
            return;

        DrawGizmosScript selectedPoint = null;
        float minDistance = float.MaxValue;

        foreach (var point in checkpointList)
        {
            float distance =
                Vector3.Distance(transform.position, point.transform.position);

            if (distance < minDistance)
            {
                minDistance = distance;
                selectedPoint = point;
            }
        }

        if (selectedPoint != null)
        {
            Gizmos.color = Color.red;
            Gizmos.DrawLine(transform.position, selectedPoint.transform.position);
        }
    }
}
```


Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

En este código se crea una lista de objetos de tipo **DrawGizmosScript** llamada *checkpointList*.

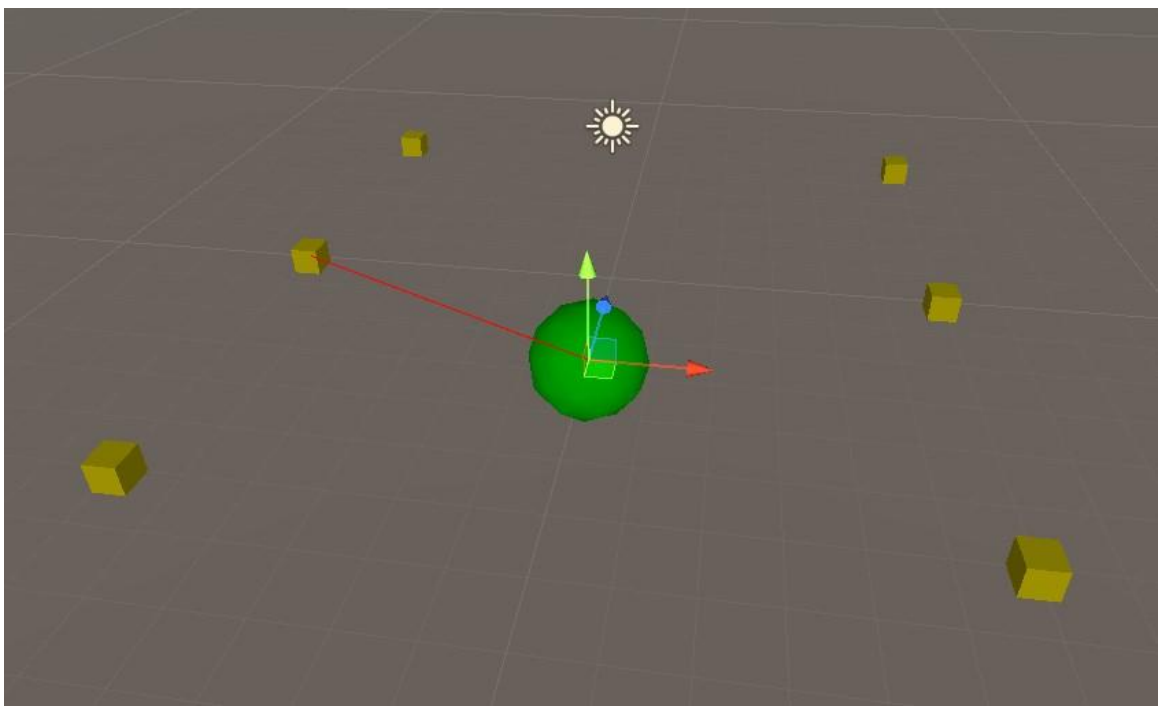
En el método **Start:**

- Se inicializa la lista utilizando el **Object initializer**. Para ello se utiliza la función **FindObjectsOfType**, que permite buscar todos los objetos del tipo indicado[1], en este caso, **DrawGizmosScript**.

En el método **OnDrawGizmos**:

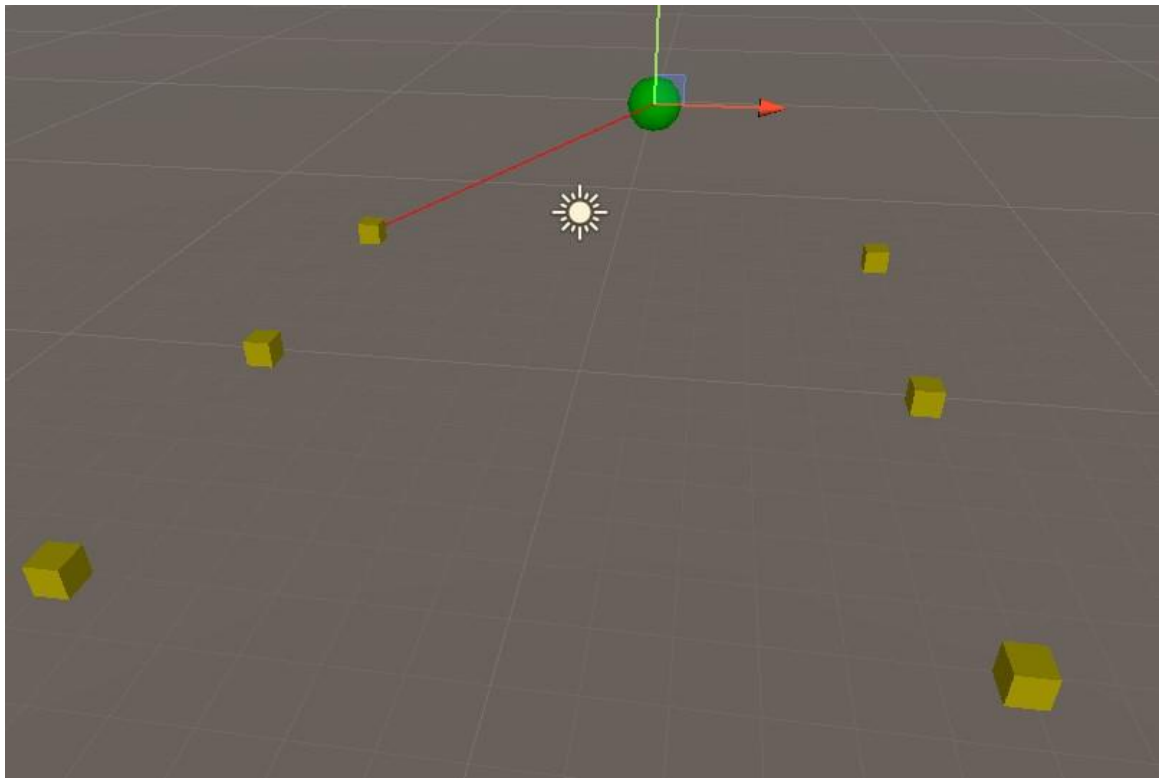
- Se define que se va a dibujar un gizmo con forma de esfera, de color verde y radio 1.
- Si la lista es nula, no se ejecuta el resto del código.
- Se declara una variable de tipo **DrawGizmosScript** llamada *selectedPoint* y se inicializa con el valor **null**. Se utilizará para almacenar el checkPoint más cercano.
- Se declara una variable de tipo **float** llamada *minDistance* y se inicializa con el valor **float.MaxValue**. Se utilizará para almacenar la mínima distancia en cada momento.
- Con un bucle foreach se recorren todos los puntos de la lista *checkpointList*. En cada iteración:
    - Se calcula la distancia desde el gameObject player hasta ese punto. Se utiliza la función **Vector3.Distance**[2].
    - Si la distancia (variable **distance**) es menor que la distancia mínima (variable **minDistance**), se actualizará el valor de la variable **minDistance** con el valor de la variable **distance** y la variable **selectedPoint** será igual a ese checkPoint.
- Si la variable **selectedPoint** no es nula, se dibujará una línea de color rojo desde el gameObject **Player** hasta ese checkPoint.

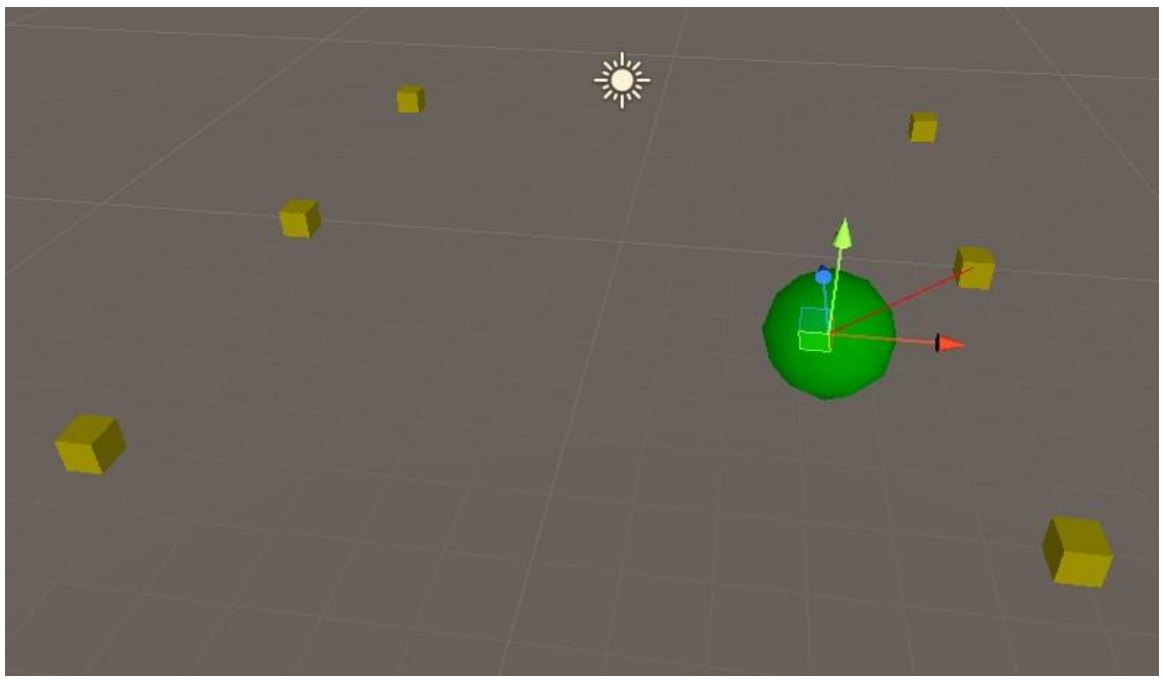
Si vamos moviendo el gameObject **Player** en tiempo de ejecución, se dibujará la línea al **checkPoint** más cercano.



[1] Lo veremos en profundidad en la clase Object
[2] Lo veremos en profundidad en la clase Vector3

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143





Atributo para ejecutar en el editor[3]

Si queremos que el código anterior funcione en el editor, sin necesidad de hacer clic en el botón Play y entrar en modo de ejecución, tendremos que utilizar el atributo **[ExecuteInEditMode]** antes de la declaración de la clase.

El código quedaría de la siguiente forma:

[3] Más adelante hay un punto entero dedicada a los atributos

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

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

[ExecuteInEditMode]

public class PlayerCheckPoint : MonoBehaviour
{
    List<DrawGizmosScript> checkpointList;

    void Start()
    {
        checkpointList =
            new List<DrawGizmosScript>(FindObjectsOfType<DrawGizmosScript>());
    }

    void OnDrawGizmos()
    {
        Gizmos.color = Color.green;
        Gizmos.DrawSphere(transform.position, 1);

        if (checkpointList.Count == 0 || checkpointList == null)
            return;

        DrawGizmosScript selectedPoint = null;
        float minDistance = float.MaxValue;

        foreach (var point in checkpointList)
        {
            float distance =
                Vector3.Distance(transform.position, point.transform.position);

            if (distance < minDistance)
            {
                minDistance = distance;
                selectedPoint = point;
            }
        }

        if (selectedPoint != null)
        {
            Gizmos.color = Color.red;
            Gizmos.DrawLine(transform.position, selectedPoint.transform.position);
        }
    }
}
```

Para recordar ...

- **OnGUI** está contenido dentro de MonoBehaviour
- El método **OnGUI** está contenido dentro del orden de ejecución de los componentes
- Este método nos permite dibujar botones y otros elementos de interfaz de usuario en la ventana de Juego (Game) y utilizarlos
- La clase **GUI** utiliza un posicionamiento manual de los elementos
- La clase **GUILayout** utiliza un posicionamiento automático de los elementos
- Un gizmo es una representación gráfica en la ventana de escena de los componentes
- **OnDrawGizmos** es una función que nos va a permitir hacer debugging visual de los componentes en la ventana de escena
- La función **OnDrawGizmosSelected** dibuja un gizmo sólo en el gameObject seleccionado
- El atributo **[ExecuteInEditMode]** sirve para que el código funcione en el editor, sin necesidad de hacer clic en el botón Play y entrar en modo de ejecución

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Buscar, obtener y crear componentes

Introducción

En esta sección vamos a ver todas las herramientas que hay para:

- Buscar componentes y grupos de componentes dentro de la jerarquía.
- Obtener los componentes de un gameObject.
- Obtener los componentes de un hijo o varios hijos de un gameObject.
- Obtener los componentes del padre de un gameObject.
- Crear componentes en un gameObject.

Diferencia entre buscar y obtener componentes - trabajando en el editor

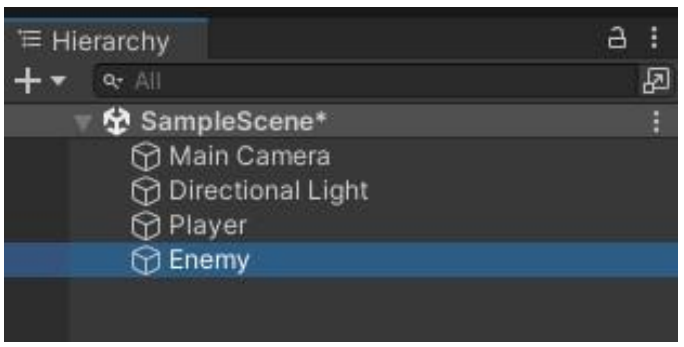
Tenemos que tener clara la diferencia entre buscar y obtener componentes:

- Buscar uno o varios componentes implica ir a la jerarquía y buscarlos por código.
- Obtener un componente implica desde un componente de un gameObject, obtener otro componente de ese mismo gameObject.

Asignar la referencia de un componente de la jerarquía desde el editor

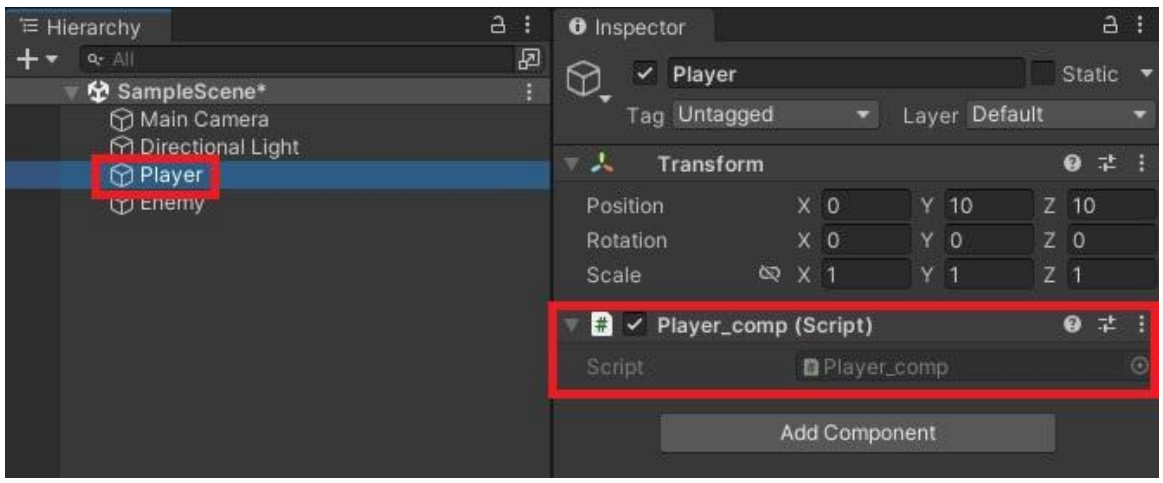
Vamos a verlo con un ejemplo:

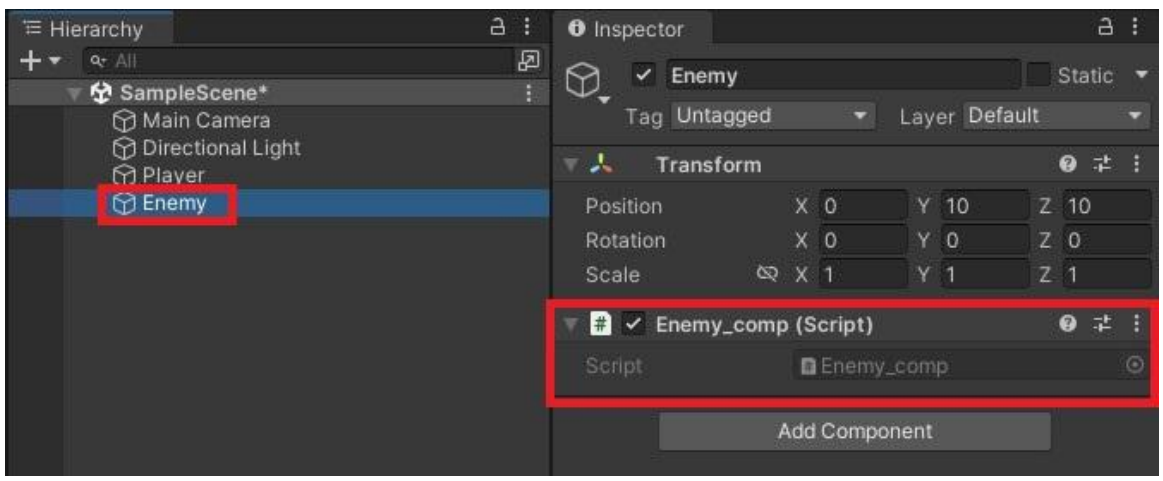
- Creamos en nuestra jerarquía dos gameObjects, uno llamado **Player** y otro llamado **Enemy**.



- Creamos también dos scripts (componentes) llamados **Player_comp** y **Enemy_comp**, y asignamos estos dos componentes a sus respectivos gameObjects.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

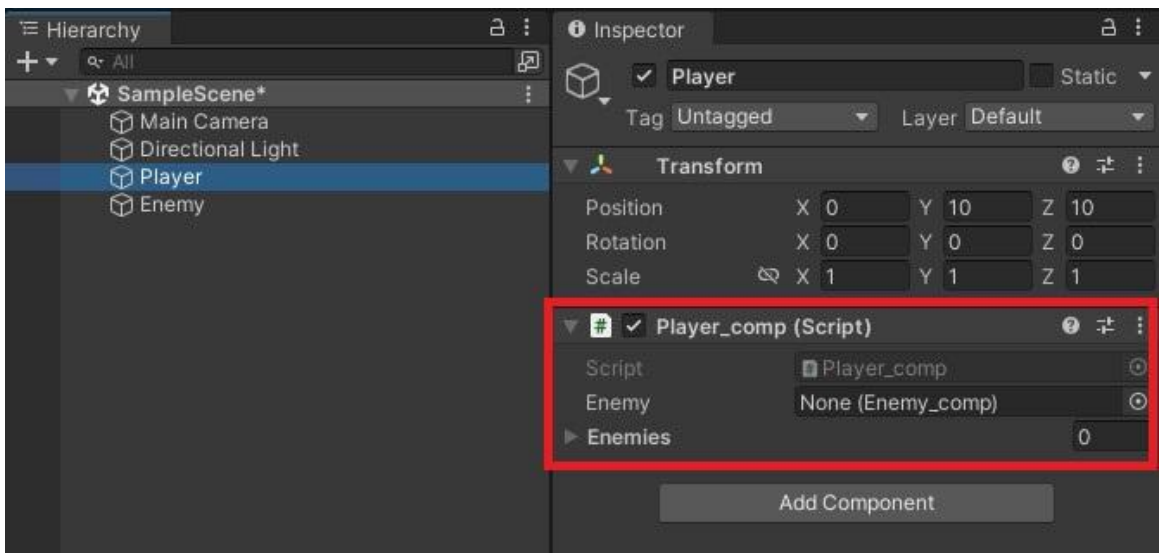




Vamos a abrir el script **Player_comp** y trabajar sobre el método **Start**. Podemos hacer búsquedas de un componente o de varios a la vez. Vamos a ver los dos casos.

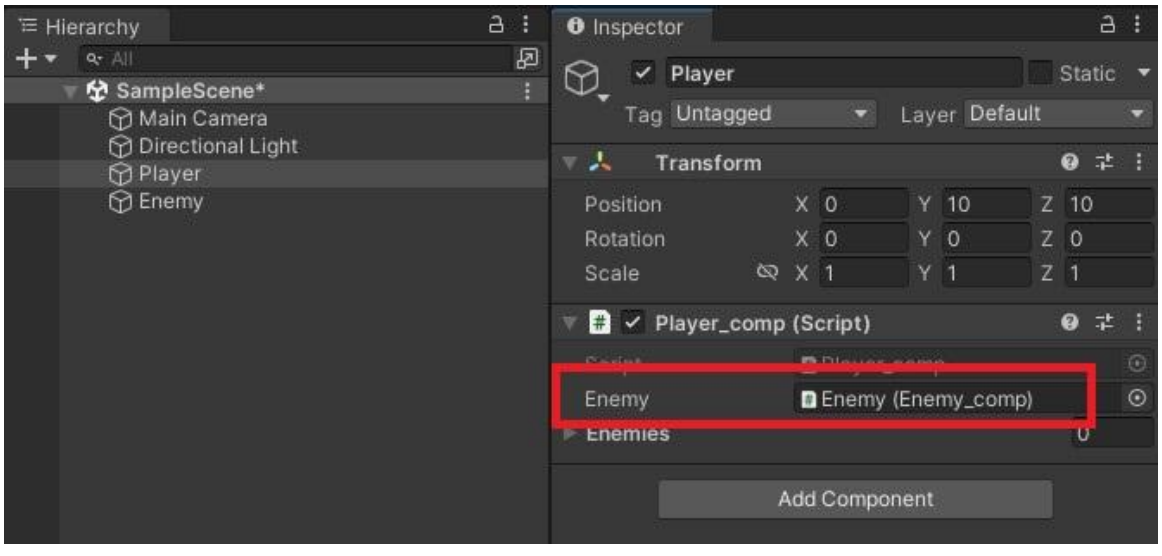
En este script hemos declarado una variable de tipo **Enemy_comp** llamada *enemy* para el caso de búsqueda de un solo componente, y una variable llamada *enemies* que es un array de tipo **Enemy_comp**, para el caso de búsqueda de varios componentes.

Si volvemos al editor de Unity y vemos las opciones del gameObject Player, vemos lo siguiente:



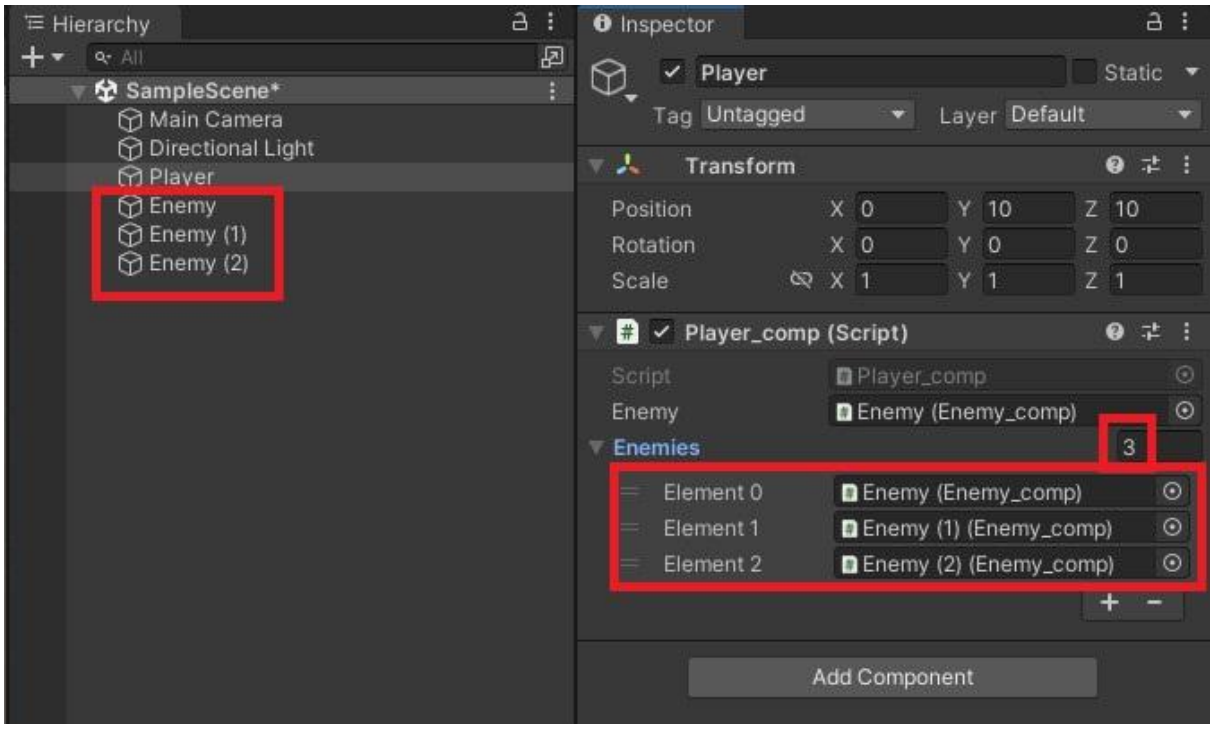
Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Para asignar la referencia de la variable *enemy*, podemos arrastrar el gameObject desde la jerarquía. Al detectar que ese gameObject tiene un componente del tipo **Enemy_comp**, se asigna correctamente.



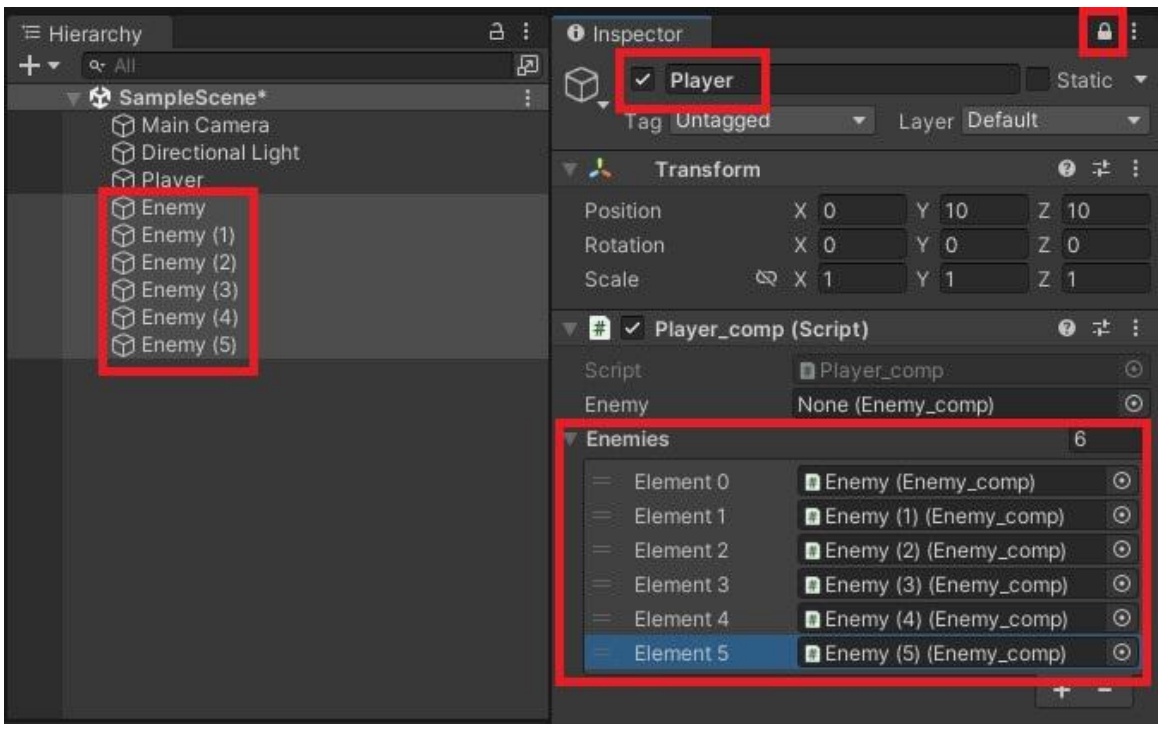
Para asignar la referencia de la variable *enemies* podemos hacerlo de dos formas.

La primera es indicar a mano el número de variables que vamos a tener. Esto habilitará tantos huecos como hayamos indicado y tendremos que arrastrar los gameObjects uno a uno desde la jerarquía.



La segunda forma es seleccionar desde la jerarquía todos los gameObjects y arrastrarlos directamente a la variable *Enemies* en el inspector. Para ello tendremos que haber bloqueado antes en el inspector la vista del gameObject *Player* para no perder la variable *Enemies* cuando hagamos la selección de todos los gameObjects a arrastrar.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143



En el siguiente punto vamos a ver cómo realizar esta búsqueda por código.

Buscando un componente en la jerarquía

Para hacer la búsqueda anterior por código tenemos que incluir el siguiente código en el script **Player_comp**:

```
1 using UnityEngine;
2
3 public class Player_comp : MonoBehaviour
4 {
5     [SerializeField] Enemy_comp enemy;
6
7     void Start()
8     {
9         enemy = FindObjectOfType<Enemy_comp>();
10
11        if(enemy != null)
12        {
13            Debug.Log(enemy.name);
14        }
15    }
16 }
```

En este código:

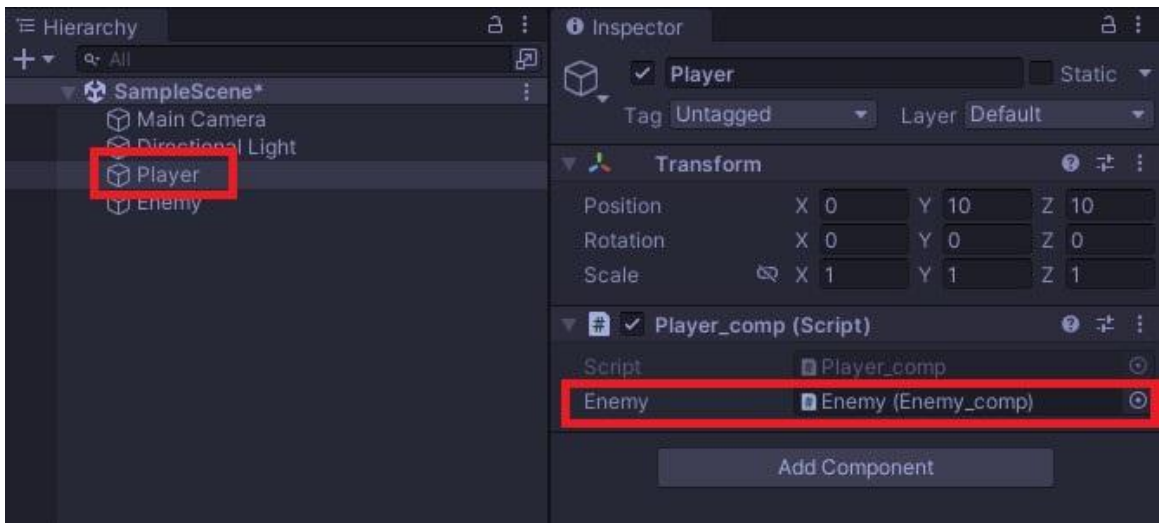
- Declaramos una variable serializada de tipo **Enemy_comp** llamada *enemy*.
- En el método **Start**, le damos valor a la variable *enemy* haciendo uso de la función **FindObjectOfType** de la clase **Object**[1] que permite buscar un objeto del tipo indicado.

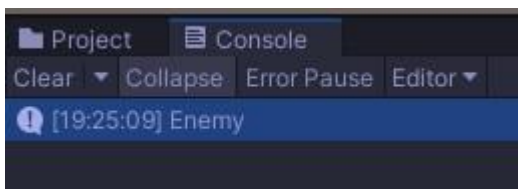
[1] Veremos en profundidad las funciones más utilizadas de la clase Object en la sección dedicada a la API de esta clase.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

- Por último hacemos una comprobación en la que, si la variable *enemy* no es nula, significa que ha encontrado un gameObject en la jerarquía que tiene un componente del tipo buscado, y se imprime su nombre en la consola.

El resultado de ejecutar este programa es que encuentra un gameObject en la jerarquía con un componente del tipo **Enemy_comp**, se asigna el componente a la variable *enemy*, y se muestra por consola el nombre del gameObject que tiene ese componente.





Hemos utilizado la función **FindObjectOfType** haciendo uso de los genéricos, pero hay dos formas más utilizando castings que ofrecen el mismo resultado.

```
enemy = (Enemy_comp)FindObjectOfType(typeof(Enemy_comp));
enemy = FindObjectOfType(typeof(Enemy_comp)) as Enemy_comp;
```

Importante: Con esta función estamos buscando un único componente. Si hay más de un gameObject en la jerarquía que tiene el tipo de componente buscado, Unity guardará la referencia del primero que encuentre en la memoria. En el siguiente punto vamos a ver cómo almacenar todas las referencias si hay más de una.

Buscando grupos de componentes en la jerarquía

Para almacenar todas las referencias a gameObjects que contengan un componente del tipo buscado, tenemos que incluir el siguiente código en el script **Player_comp**:

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

```cs
using UnityEngine;

public class Player_comp : MonoBehaviour
{
    [SerializeField] Enemy_comp[] enemies;

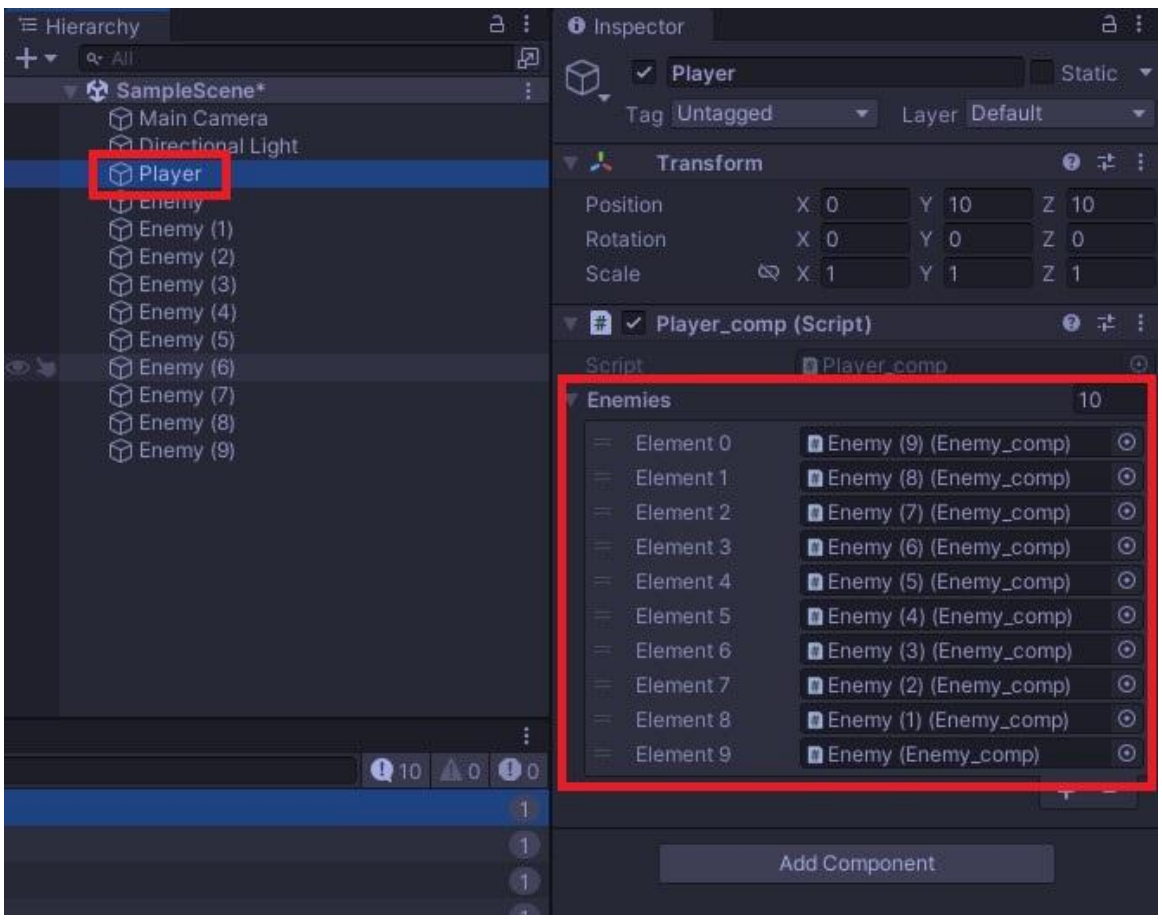
    void Start()
    {
        enemies = FindObjectsOfType<Enemy_comp>();

        foreach (var item in enemies)
        {
            Debug.Log(item.name);
        }
    }
}
```

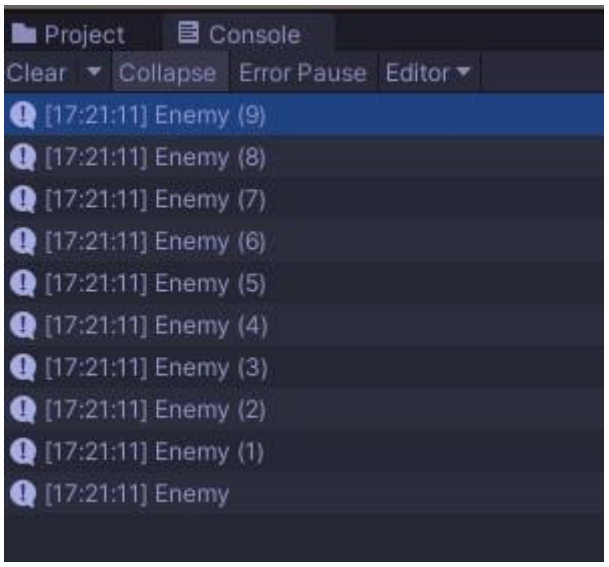
En este código:

- Declaramos una variable serializada que es un array del tipo **Enemy_comp** llamada *enemies*.
- En el método **Start**, rellenamos el array *enemies* haciendo uso de la función **FindObjectsOfType** de la clase **Object**, que permite buscar todos los objetos del tipo indicado.
- Finalmente, recorremos el array y mostramos en consola el nombre de cada gameObject que contiene el componente buscado.

El resultado de ejecutar este programa es que se encuentran todos los gameObject en la jerarquía con un componente del tipo **Enemy_comp**, se asignan a la variable *enemies*, y se muestra por consola el nombre del gameObject de cada uno de los componentes encontrados.



Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143



Hemos utilizado la función **FindObjectsOfType** haciendo uso de los genéricos, pero hay dos formas más utilizando castings que ofrecen el mismo resultado.

```
enemies = FindObjectsOfType(typeof(Enemy_comp)) as Enemy_comp[];
enemies = (Enemy_comp[])FindObjectsOfType(typeof(Enemy_comp));
```

Obteniendo un componente del gameObject

Ahora vamos a ver cómo obtener componentes en vez de buscarlos. Podemos necesitar obtener componentes en dos situaciones:

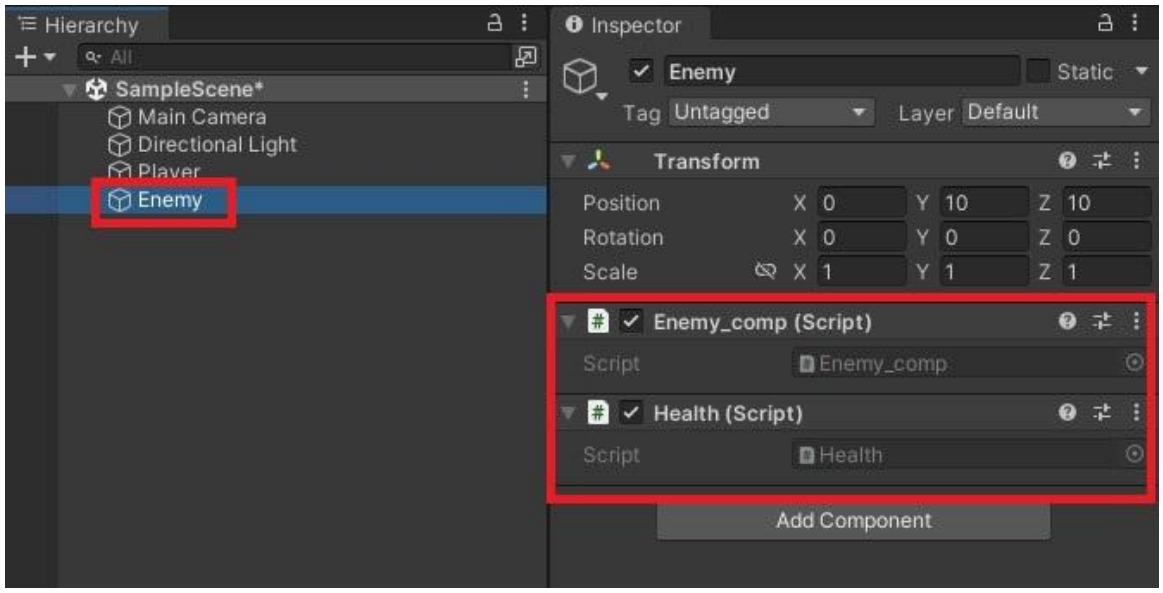
La primera es cuando estamos en un componente y obtenemos otro componente dentro del mismo gameObject.

La segunda situación es si tenemos una referencia a un gameObject y necesitamos alguno de sus componentes.

Vamos a ver un ejemplo:

Imagina que tenemos un gameObject llamado **Enemy** en la jerarquía, y éste tiene dos componentes llamados **Enemy_comp** y **Health**.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143



Si desde el componente **Enemy_comp** queremos acceder al componente **Health**, no tenemos que ir a la jerarquía a buscarlo, puesto que estamos en el mismo gameObject. Simplemente tenemos que obtenerlo. Para ello:

```
using UnityEngine;

public class Enemy_comp : MonoBehaviour
{
    [SerializeField] Health health;

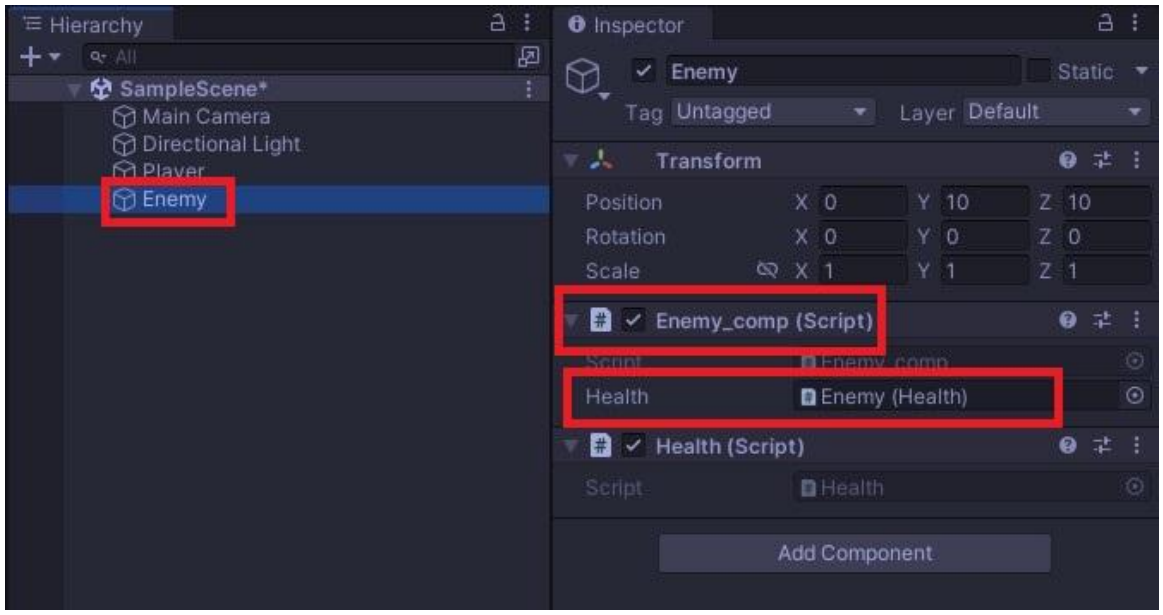
    void Start()
    {
        health = GetComponent<Health>();
    }
}
```

En este código:

- Declaramos una variable de tipo **Health** llamada *health*.
- En el método **Start**, le damos valor a la variable *health* utilizando la función **GetComponent** e indicando el tipo de componente que queremos obtener.

Si ejecutamos este programa podemos ver que, en el componente **Enemy_comp**, tenemos la referencia al componente **Health**. Ambos componentes están en el gameObject **Enemy**.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143



De esta forma, desde el componente **Enemy_comp** tenemos acceso a todas las propiedades y funcionalidades públicas que haya en el componente **Health**.

Al igual que en la búsqueda de componentes, hemos utilizado la función **GetComponent** con genéricos, pero hay cuatro formas más utilizando castings y strings que ofrecen los mismos resultados.

```
health = GetComponent(typeof(Health)) as Health;
health = (Health)GetComponent(typeof(Health));
health = GetComponent("Health") as Health;
health = (Health)GetComponent("Health");
```

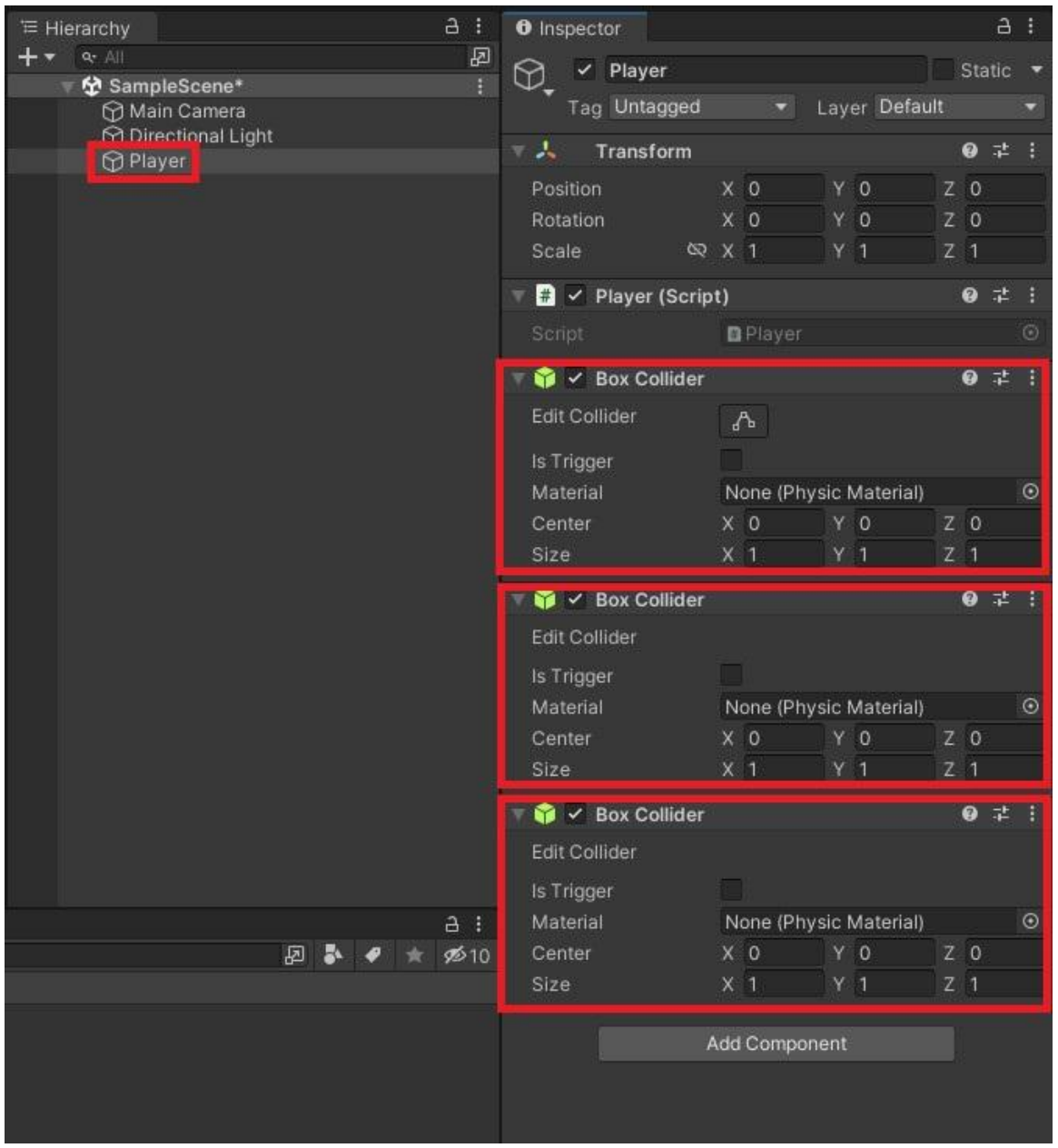
Obteniendo varios componentes del gameObject

Ahora vamos a obtener todos los componentes de un tipo determinado de un gameObject.

Vamos a verlo con un ejemplo:

Tenemos un gameObject llamado **Player** con varios componentes de tipo **BoxCollider**.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143



El código del componente **Player** será:

```
using UnityEngine;

public class Player : MonoBehaviour
{
    [SerializeField] BoxCollider[] colliders;

    void Start()
    {
        colliders = GetComponents<BoxCollider>();

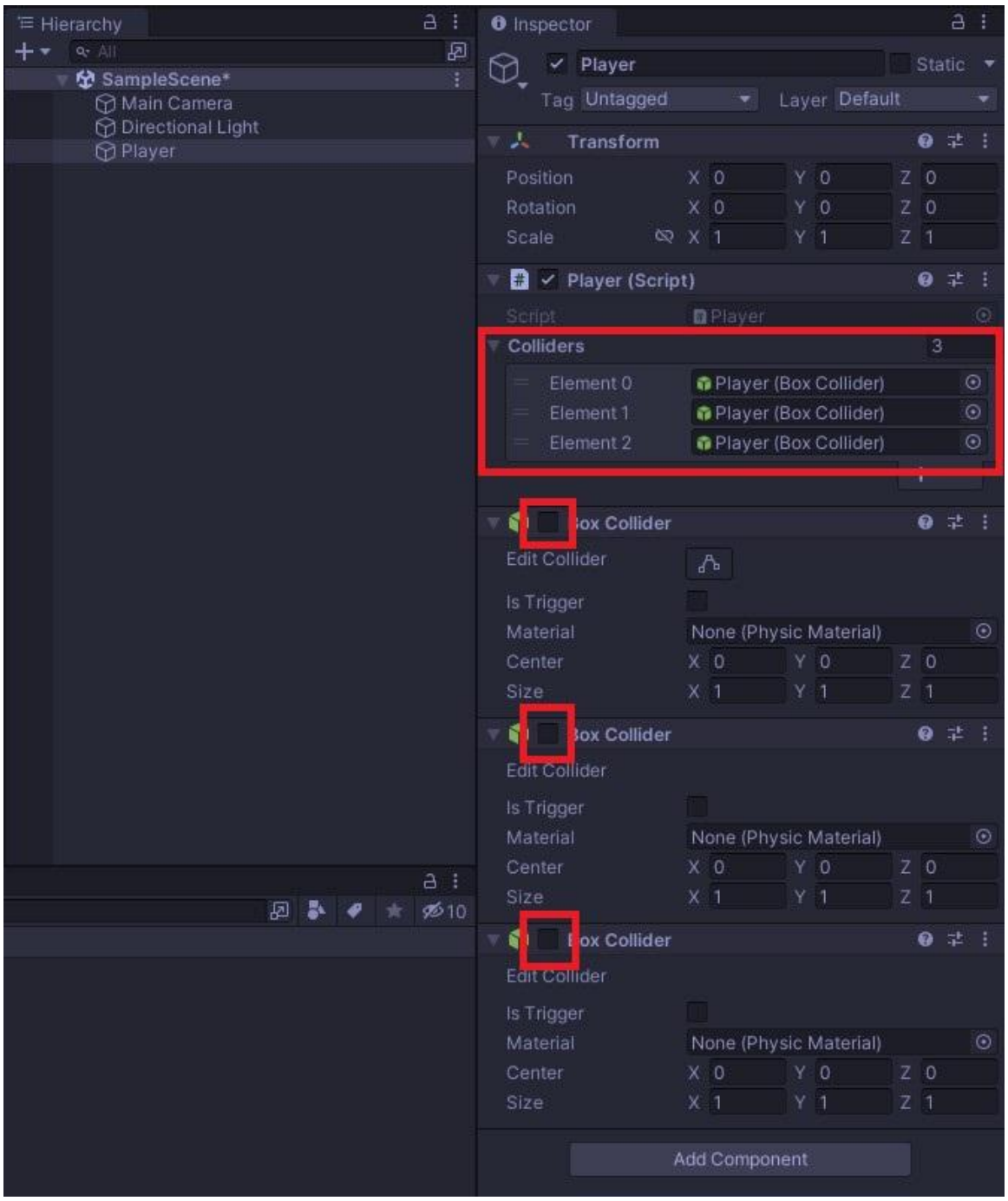
        foreach (BoxCollider collider in colliders)
        {
            collider.enabled = false;
        }
    }
}
```


Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

En este código:

- Declaramos un array de tipo **BoxCollider** llamado *colliders*.
- En el método **Start**, asignamos valor a la variable *colliders* utilizando la función **GetComponents**, que busca todos los componentes de un tipo determinado en el gameObject.
- Finalmente, recorremos el array y desactivamos cada componente encontrado.

Si ejecutamos este programa podemos ver que se encuentran todos los componentes de tipo **BoxCollider** y se desactivan.



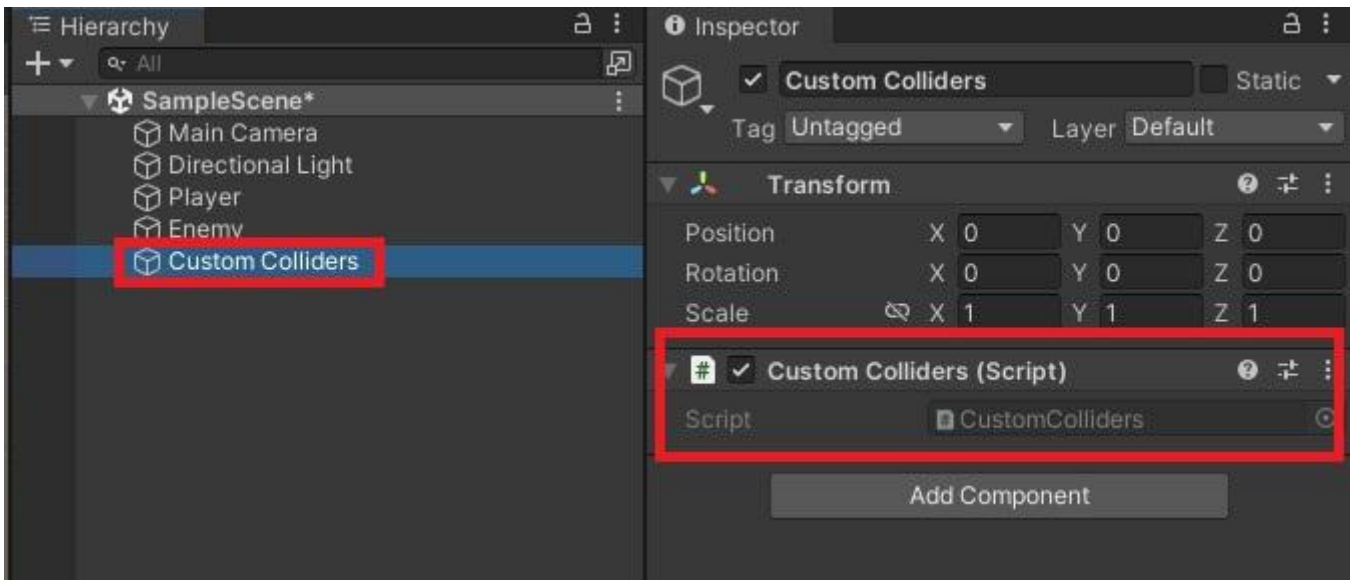
Obteniendo un componente del hijo de un gameObject

Podemos utilizar la función **GetComponentInChildren** para obtener el componente de un tipo determinado en un hijo o grupo de hijos si hay una relación de parentesco en la jerarquía.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Vamos a verlo con un ejemplo:

Creamos un gameObject en la jerarquía llamado **Custom Colliders**, y le adjuntamos un script (componente) llamado **CustomColliders**.



El código del componente **Enemy_comp** será:

```
using UnityEngine;

public class Player : MonoBehaviour
{
    [SerializeField] CustomColliders collider;

    void Start()
    {
        collider = GetComponentInChildren<CustomColliders>();

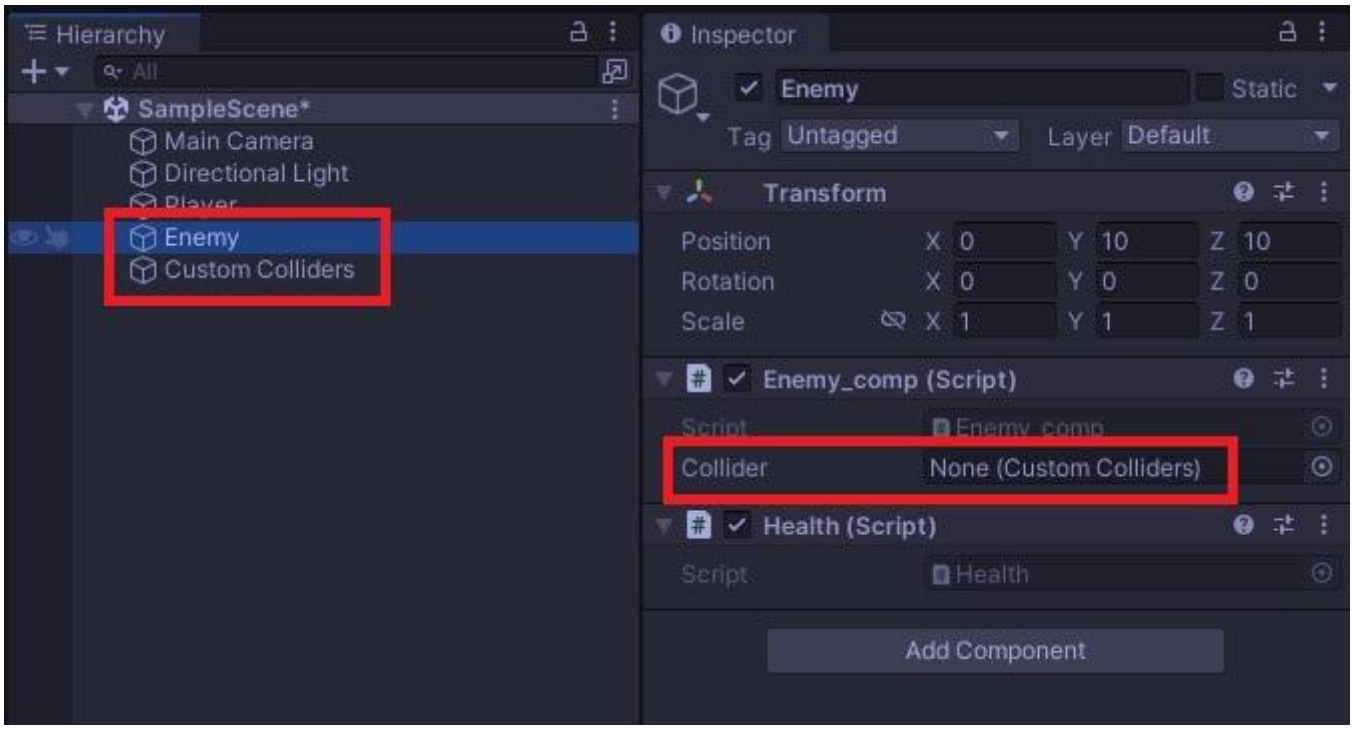
        if(collider != null)
        {
            print("Componente encontrado");
        }
        else
        {
            print("Componente NO encontrado");
        }
    }
}
```

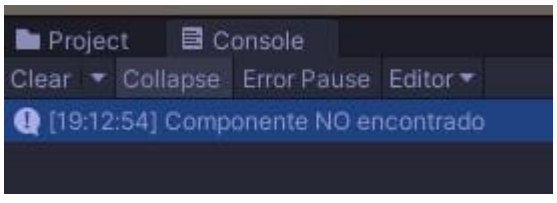
En este código:

- Declaramos una variable de tipo **CustomColliders** llamada *collider*.
- En el método **Start**, asignamos valor a la variable *collider* utilizando la función **GetComponentInChildren**, que busca un componente de un tipo determinado en el hijo o hijos del gameObject.
- Finalmente, si encuentra el componente se escribe en consola “Componente encontrado”, y si no lo encuentra se escribe en consola “Componente NO encontrado”.

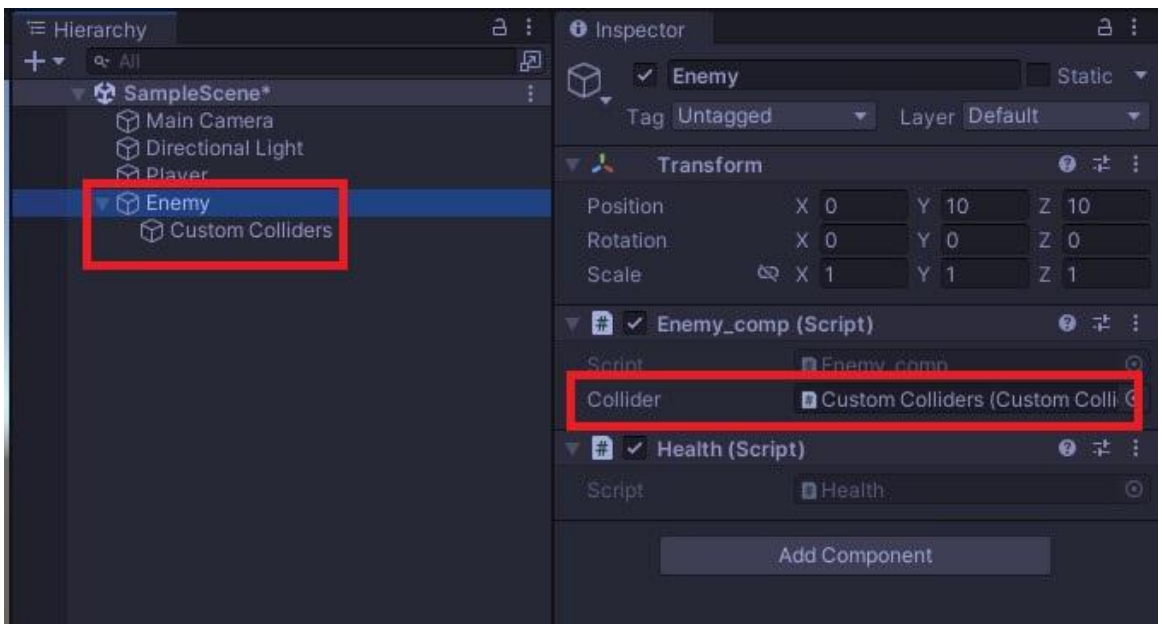
Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

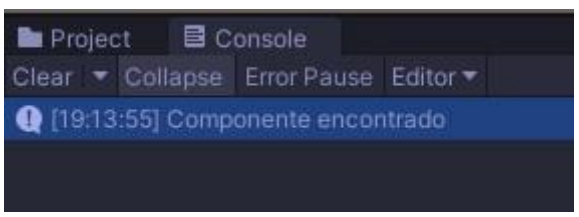
Si ejecutamos este programa tal cual lo tenemos podemos ver que no se encuentra el componente, puesto que no hay ningún parentesco y, por tanto, el componente no está en el hijo.





Mientras que si hacemos el gameObject **Custom Colliders** hijo del gameObject **Enemy**, sí que se encontrará el componente.





Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

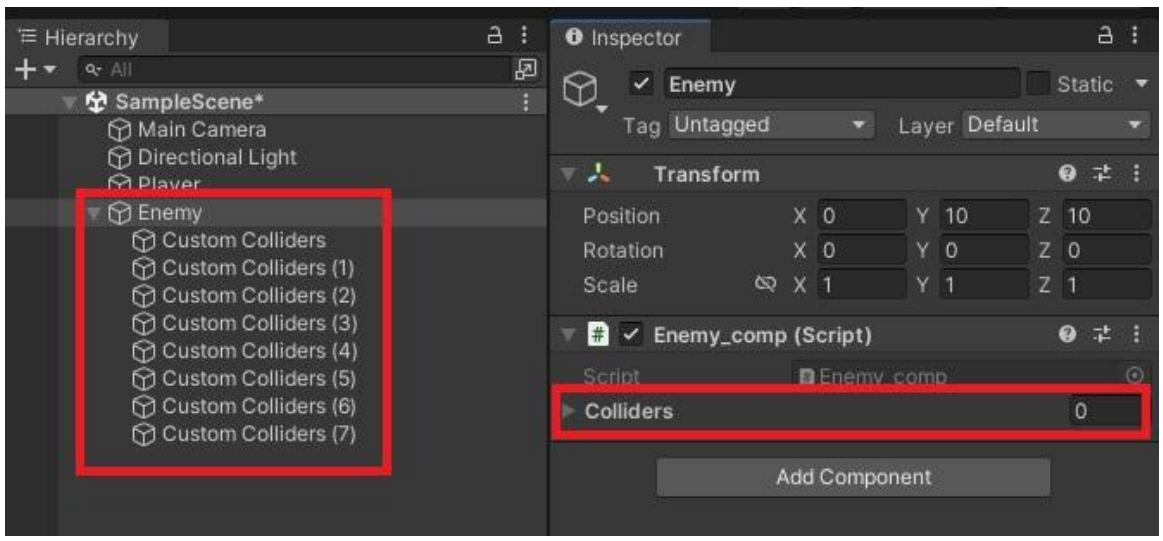
De nuevo, hemos utilizado la función con genéricos pero, para todas las funciones de la familia **GetComponent**, son válidas las formas vistas en el punto anterior.

Obteniendo varios componentes de los hijos de un gameObject

Ahora vamos a obtener todos los componentes de un tipo determinado de un hijo o grupo de hijos.

Vamos a verlo con un ejemplo.

Partiendo del ejemplo anterior, creamos varias copias del gameObject **Custom Colliders,** y hacemos todas hijas del gameObject **Enemy**.



El código del componente **Enemy_comp** será:

```
using UnityEngine;

public class Player : MonoBehaviour
{
    [SerializeField] CustomColliders[] colliders;

    void Start()
    {
        colliders = GetComponentsInChildren<CustomColliders>();

        foreach (var item in colliders)
        {
            print(item.name);
        }
    }
}
```

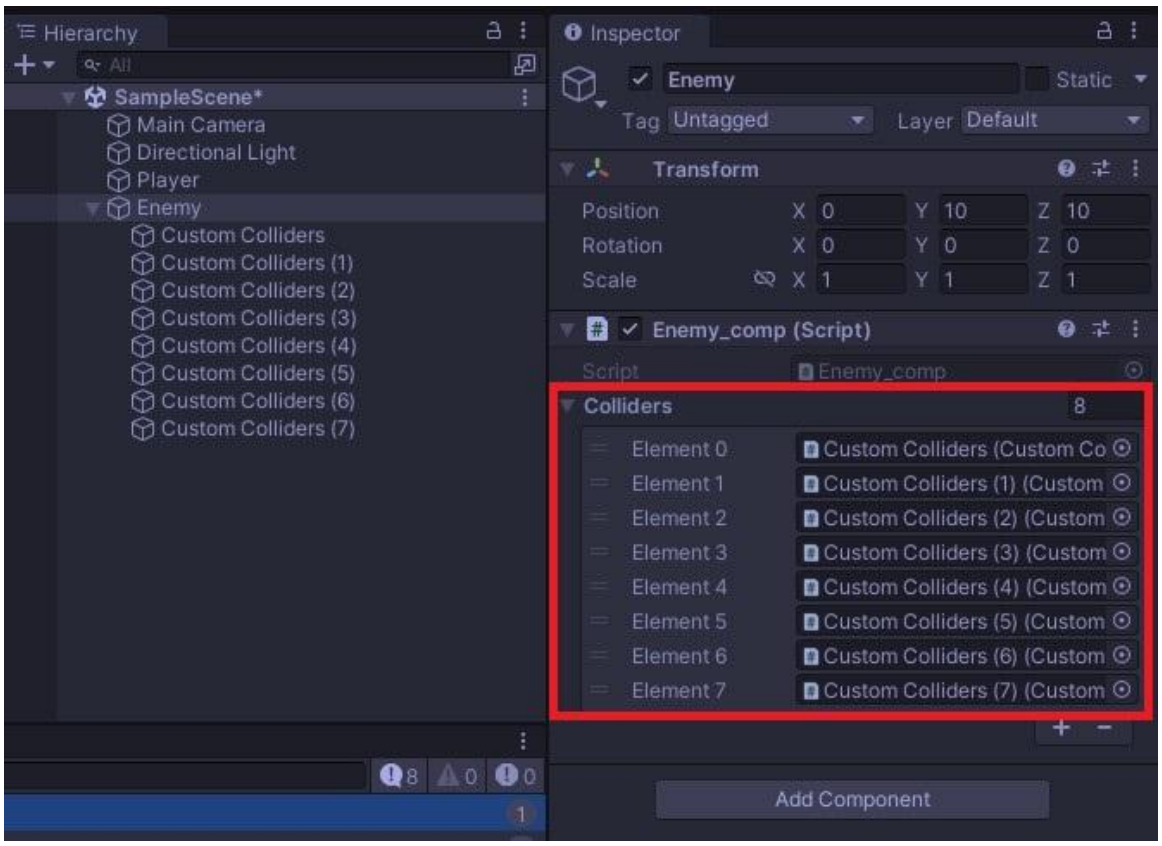
En este código:

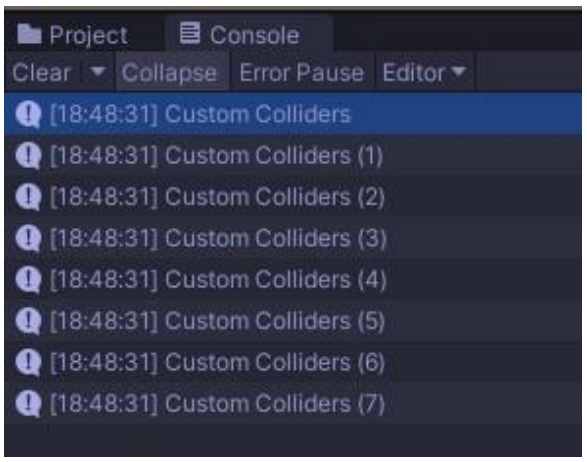
- Declaramos un array de tipo **CustomColliders** llamado *colliders*.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

- En el método **Start**, asignamos valor a la variable *colliders* utilizando la función **GetComponentsInChildren**, que busca todos los componentes de un tipo determinado en los hijos del gameObject.
- Finalmente, recorremos el array y mostramos en consola el nombre de cada gameObject que contiene el componente obtenido.

Si ejecutamos este programa podemos ver que encuentra todos los componentes y se muestra en consola el nombre de los gameObjects que poseen dichos componentes.



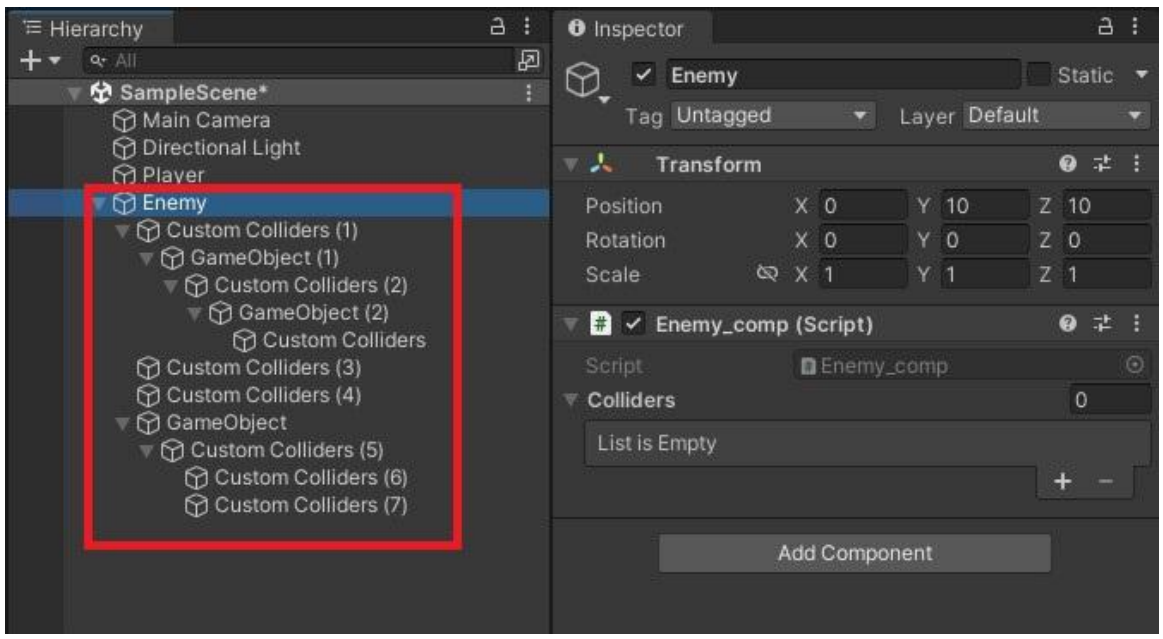


Obteniendo componentes de hijos de jerarquía compleja del gameObject

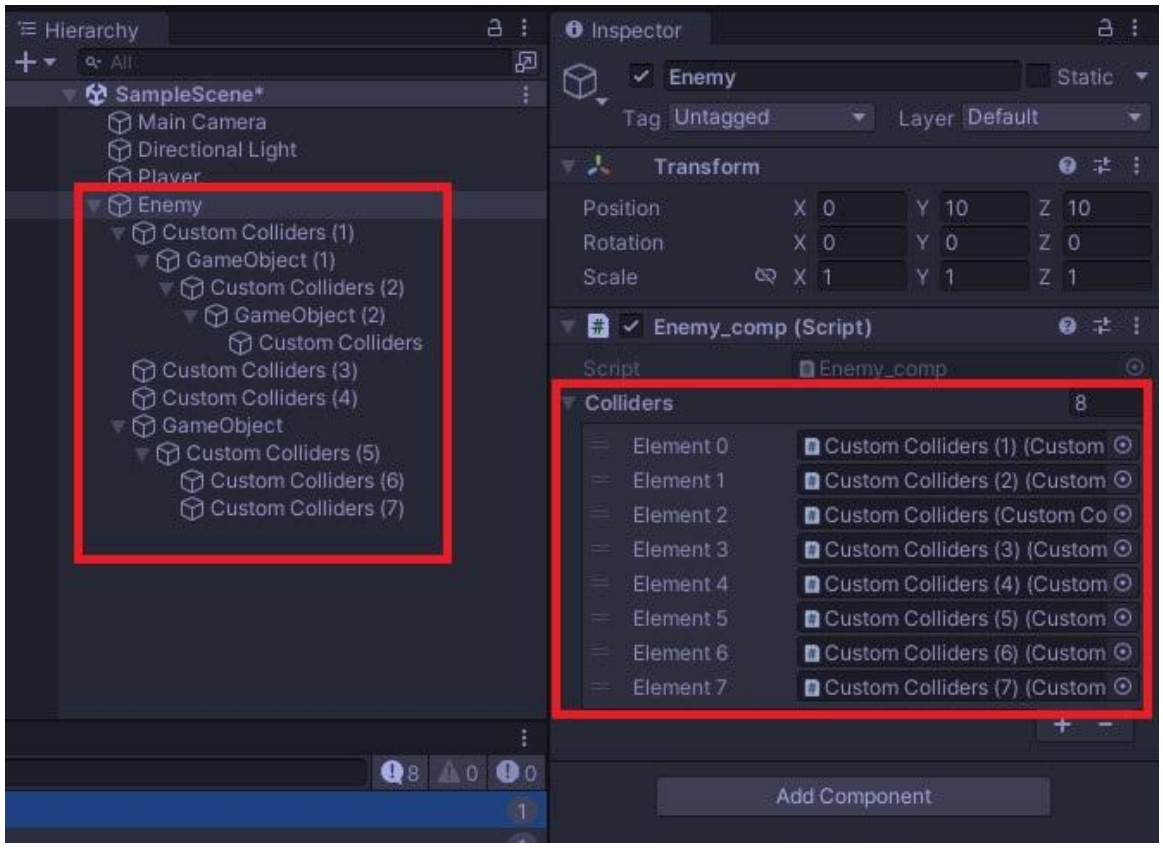
En el punto anterior estamos buscando componentes en los hijos de un gameObject en una jerarquía simple o de primer nivel. Pero si tuviésemos una jerarquía compleja, el mismo código funcionaría del mismo modo.

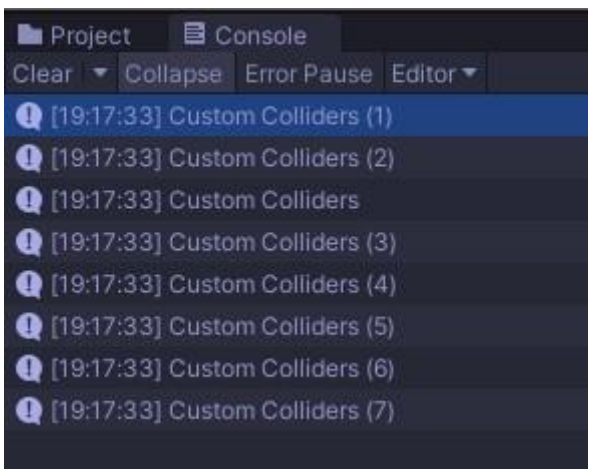
Vamos a verlo con un ejemplo. Si tenemos una jerarquía compleja:

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143



Si ejecutamos el programa, tendremos el mismo resultado que en el punto anterior.





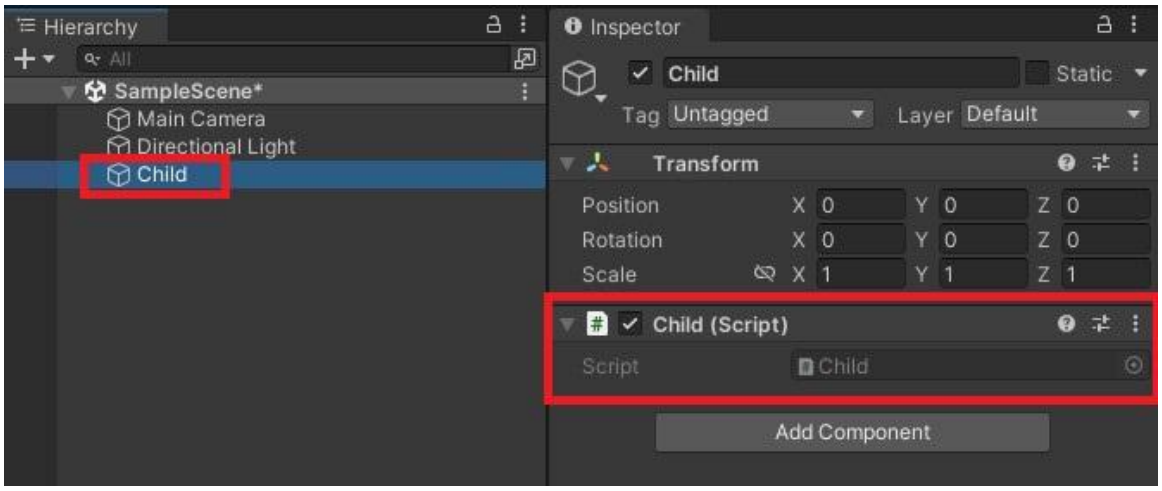
Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Obteniendo componentes de padres del gameObject

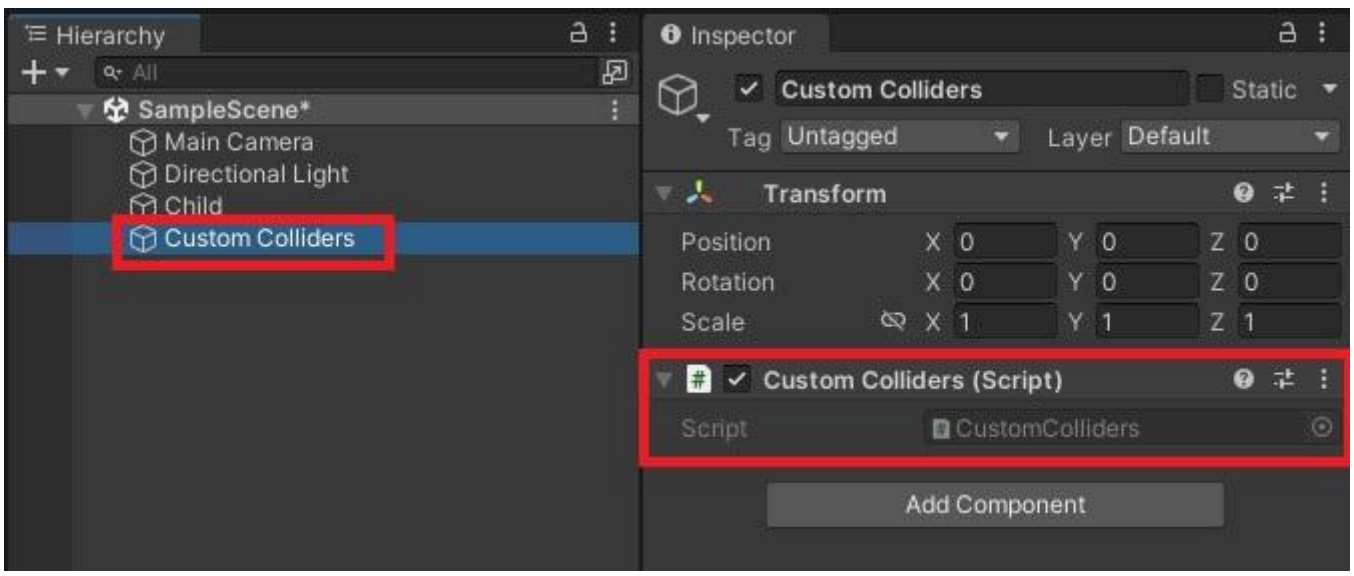
Ahora vamos a hacer la operativa inversa y, desde los hijos, vamos a obtener un componente o varios componentes de los padres.

Vamos a ver cómo obtener un componente o varios con un mismo ejemplo.

Creamos un gameObject vacío llamado **Child**. Va a ser el gameObject en la parte inferior de la jerarquía. A su vez, este gameObject tendrá un componente también llamado **Child**.

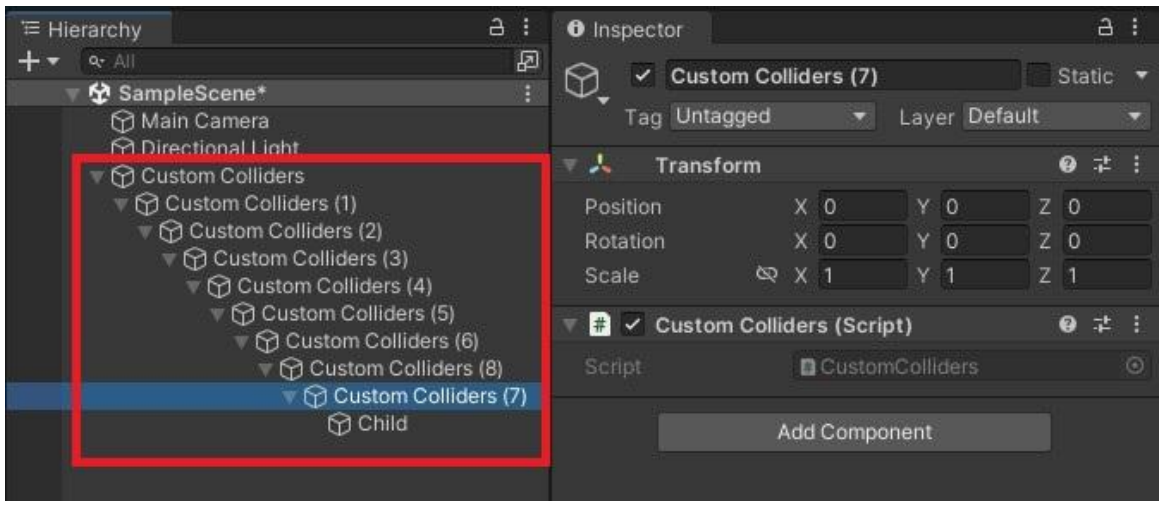


Ahora creamos un gameObject llamado **Custom Colliders** que tenga un componente llamado **CustomColliders**.



Por último, creamos varias copias de él y creamos una jerarquía compleja, poniendo en la parte inferior el gameObject **Child**.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143



El componente **Child** tendrá el siguiente código:

```
1 using UnityEngine;
2
3 public class Child : MonoBehaviour
4 {
5     [SerializeField] CustomColliders collider;
6     [SerializeField] CustomColliders[] colliders;
7
8     void Start()
9     {
10        collider = GetComponentInParent<CustomColliders>();
11        colliders = GetComponentsInParent<CustomColliders>();
12
13        if (collider != null)
14        {
15            print("Componente encontrado");
16        }
17        else
18        {
19            print("Componente NO encontrado");
20        }
21
22        foreach (var item in colliders)
23        {
24            print(item.name);
25        }
26    }
27 }
```

En este código:

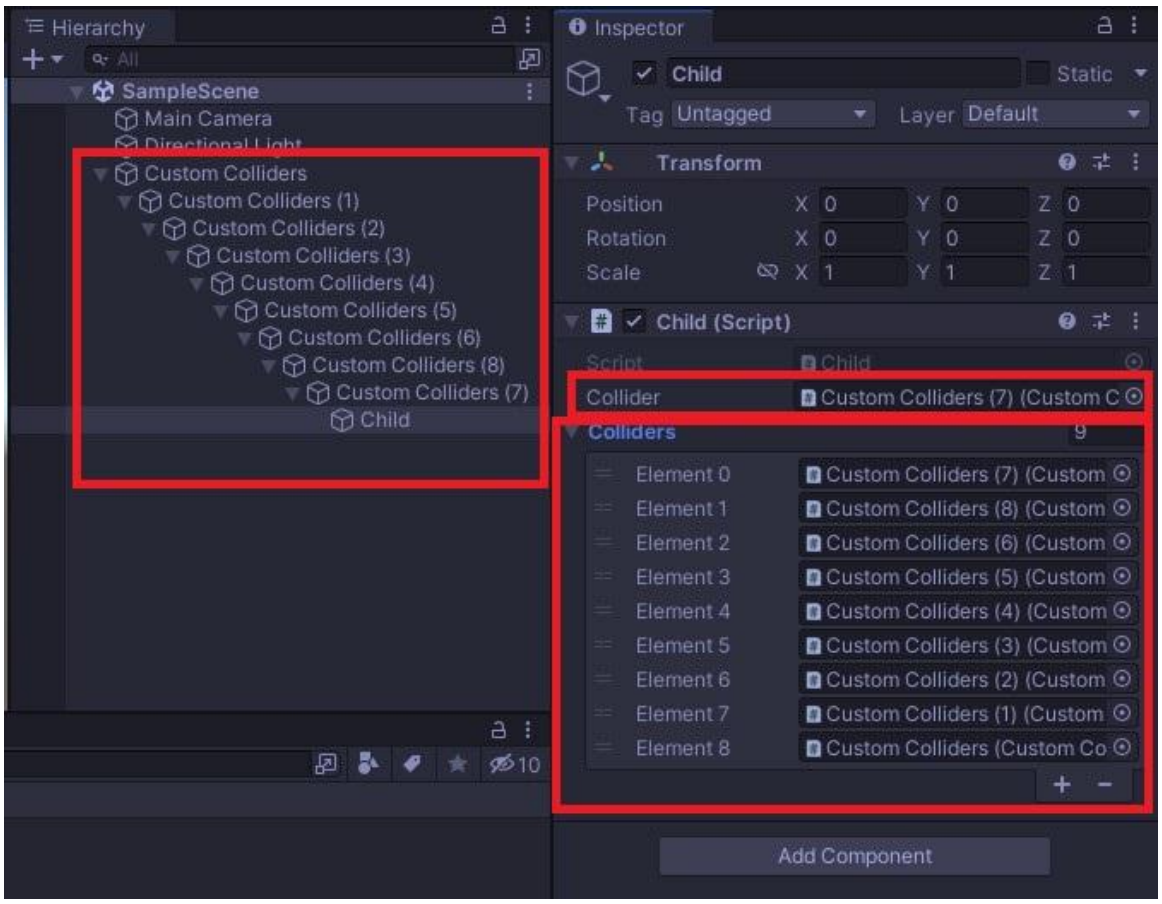
- Declaramos una variable serializada de tipo **CustomColliders** llamada *collider*.
- Declaramos una variable serializada de tipo array de **CustomColliders** llamada *colliders*.
- En el método **Start** asignamos valor a la variable *collider* utilizando la función **GetComponentInParent**, que busca un componente de un tipo determinado en el padre del gameObject.
- En el método **Start**, asignamos valor a la variable *colliders* utilizando la función **GetComponentsInParent**, que busca todos los componentes de un tipo determinado en los padres del gameObject.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

- Si encuentra el componente, es decir, si la variable *collider* no es nula, se escribe en consola "Componente encontrado", y si no lo encuentra se escribe en consola "Componente NO encontrado".
- Finalmente, recorremos el array *colliders* y mostramos en consola el nombre de cada gameObject que contiene el componente obtenido.

Si ejecutamos este programa podemos ver que:

- La variable *collider* se rellena correctamente y se muestra en consola el mensaje de "componente encontrado".
- El array de colliders se rellena correctamente y se muestra el nombre de los gameObjects que poseen los componentes encontrados.



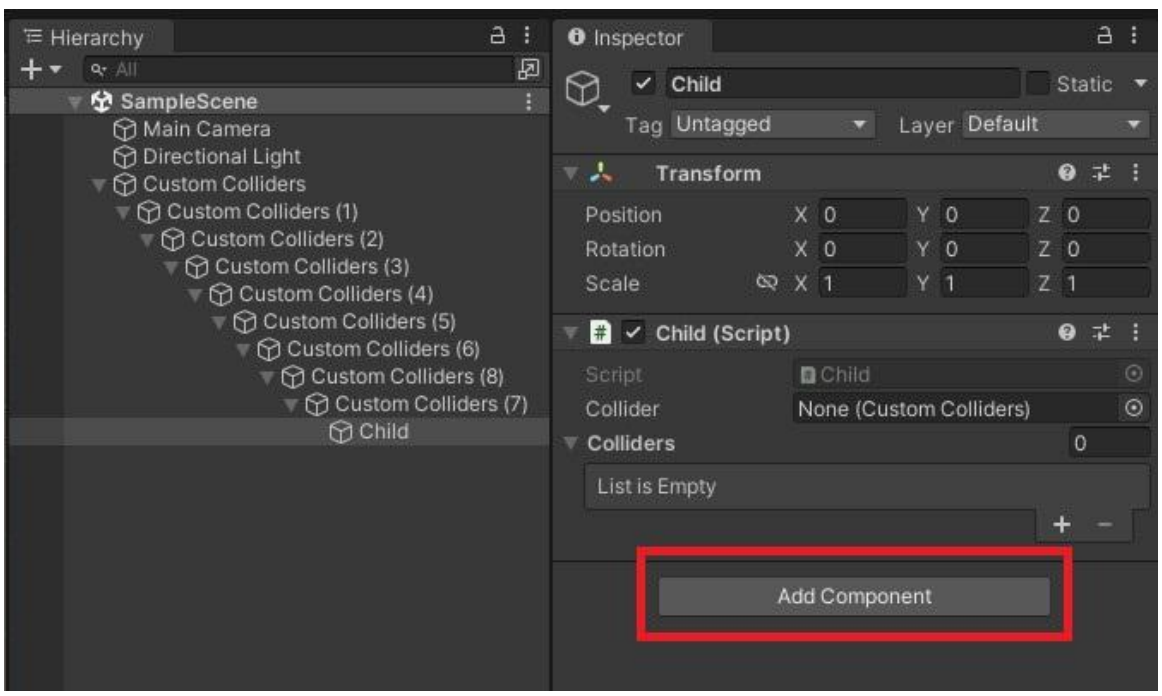


Tal como hemos visto en el punto anterior, esto funciona tanto con jerarquías simples como con jerarquías complejas.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Añadir componentes en tiempo de ejecución

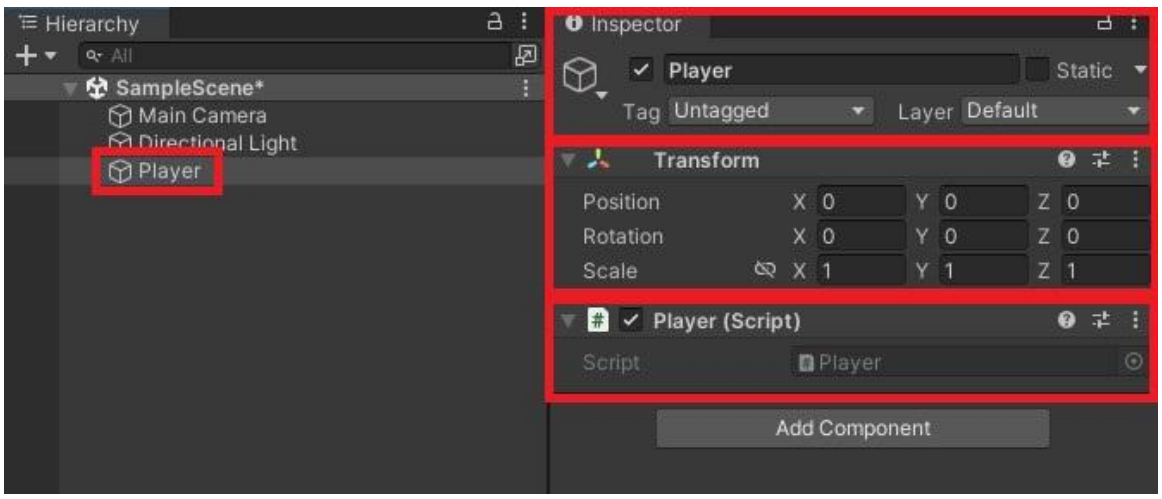
Para añadir componentes en el editor utilizamos el botón Add Component



Para añadir componentes por código en tiempo de ejecución tenemos la misma función **AddComponent**, que pertenece a la clase **GameObject**.

Vamos a verlo con un ejemplo.

Utilizaremos un gameObject vacío llamado **Player** que solo tiene los componentes por defecto (componente **GameObject** y componente **Transform**) y un componente llamado **Player**.



Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

El código del script **Player** sería el siguiente:

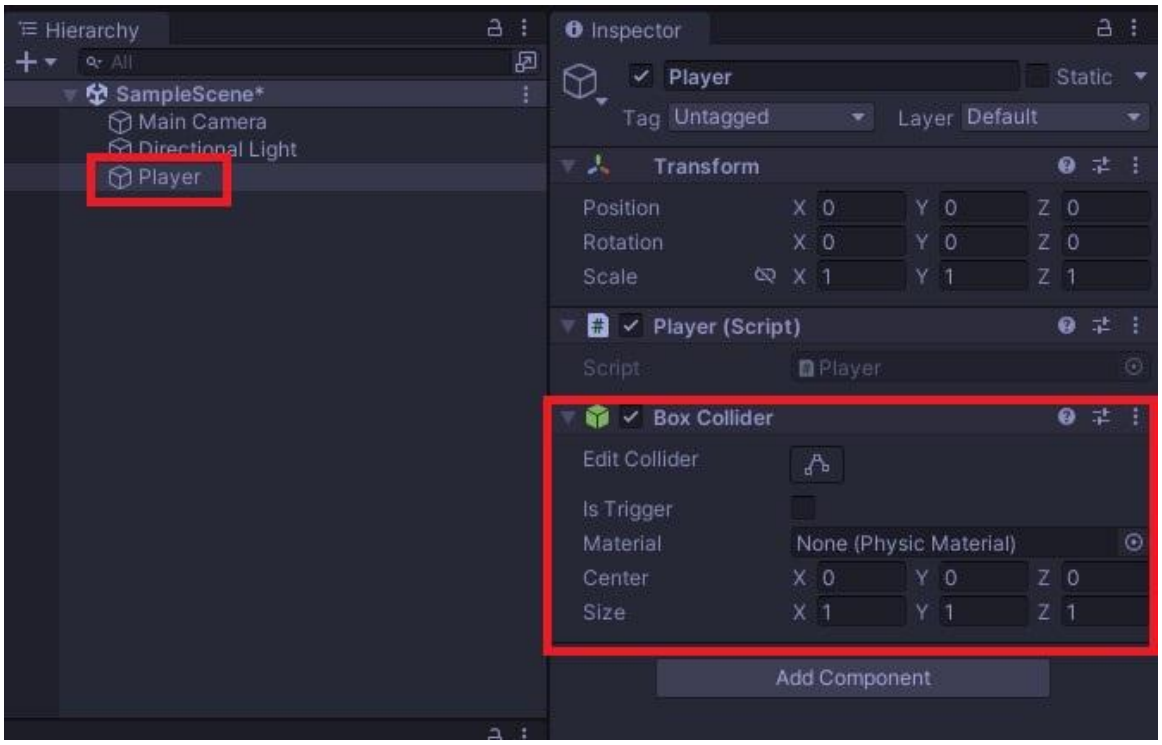
```
using UnityEngine;

public class Player : MonoBehaviour
{
    void Start()
    {
        gameObject.AddComponent<BoxCollider>();
    }
}
```

En este código:

- En el método **Start** utilizamos la instancia de gameObject con la que estamos trabajando y llamamos al método **AddComponent**. Tenemos que indicar el tipo, y se puede indicar de las mismas formas que hemos explicado en puntos anteriores (con genéricos o utilizando castings). En este caso indicamos el tipo haciendo uso de los genéricos y le añadimos un componente de tipo **BoxCollider**.

Cuando ejecutamos el programa podemos observar que al gameObject **Player** se le añade un componente de tipo **BoxCollider**.



Hemos creado el componente del tipo **BoxCollider**, pero no hemos almacenado su referencia. Para almacenarla:

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

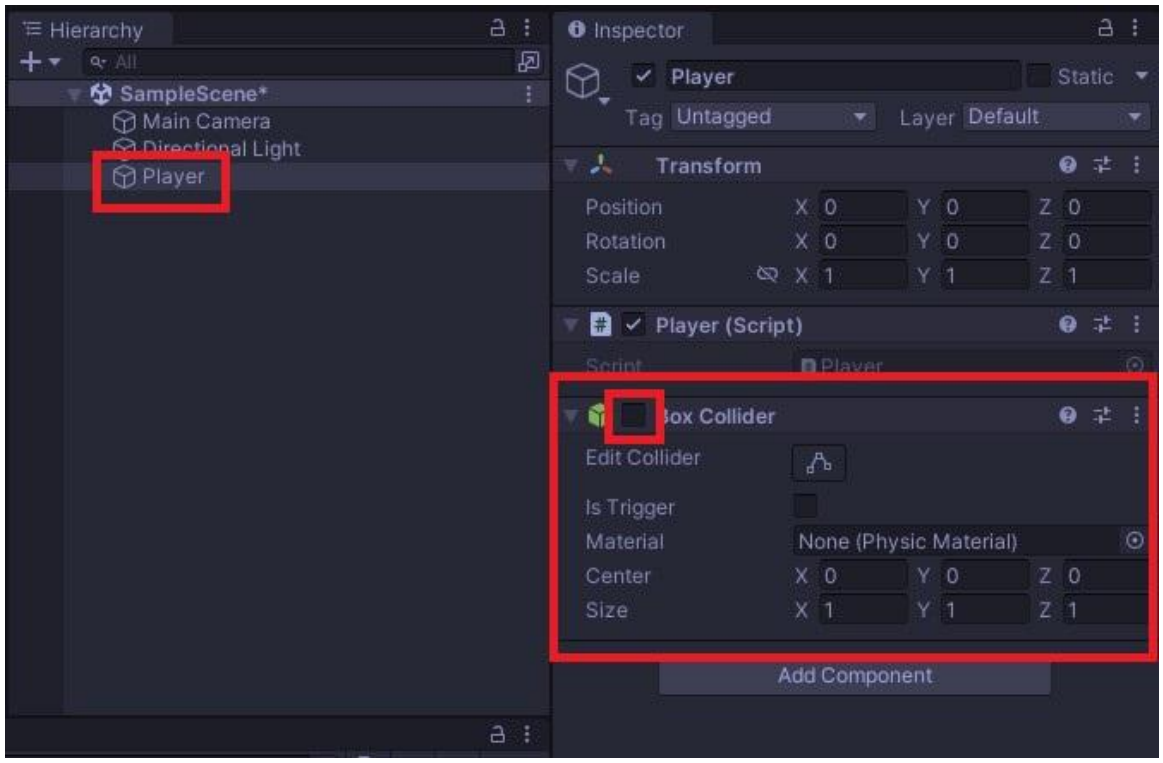
```using UnityEngine;

public class Player : MonoBehaviour
{
    void Start()
    {
        BoxCollider myBoxCollider =
                gameObject.AddComponent<BoxCollider>();
        myBoxCollider.enabled = false;
    }
}
```

En este código:

- Declaramos una variable de tipo **BoxCollider** llamada *myBoxCollider*.
- Le asignamos valor utilizando la función **AddComponent** como en el paso anterior.
- Al tener la referencia en la variable, podemos realizar modificaciones sobre ese componente. En este ejemplo lo desactivamos poniendo el valor de su propiedad **enabled** a **false**.

Cuando ejecutamos el programa podemos observar que al gameObject **Player** se le añade un componente de tipo **BoxCollider** y se desactiva, mientras que en el ejemplo anterior estaba activo.



Para recordar ...

- Es diferente buscar y obtener componentes
- Buscar uno o varios componentes implica ir a la jerarquía y buscarlos por código
- Obtener un componente implica desde un componente de un gameObject, obtener otro componente de ese mismo gameObject
- Para buscar un componente de un tipo determinado utilizaremos la función **FindObjectOfType**

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

- Para buscar todos los componentes de un tipo determinado utilizaremos **FindObjectsOfType**
- Para obtener un componente de un tipo determinado de un gameObject utilizaremos la función **GetComponent**
- Para obtener todos los componentes de un tipo determinado de un gameObject utilizaremos la función **GetComponents**
- Para obtener el componente de un tipo determinado del hijo o grupo de hijos de un gameObject utilizaremos la función **GetComponentInChildren**
- Para obtener todos los componentes de un tipo determinado de un hijo o grupo de hijos utilizaremos la función **GetComponentsInChildren**
- Para obtener el componente de un tipo determinado del padre de un gameObject utilizaremos la función **GetComponentInParent**
- Para obtener todos los componentes de un tipo determinado del padre de un gameObject utilizaremos la función **GetComponentsInParent**
- Para añadir componentes por código en tiempo de ejecución utilizaremos la función **AddComponent**

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Clase GameObject

- Clase de UnityEngine
- Hereda de Object
- Implementada en UnityEngine.CoreModule

Descripción

Clase base para todas las entidades en las escenas de Unity. Define todas las propiedades y funcionalidades que podemos utilizar con los gameObjects. Algunas son muy genéricas y otras son muy específicas.

Propiedades

- **activeInHierarchy**: Define si el gameObject está activo o no en la escena.
- **activeSelf**: Devuelve el estado activo local de este gameObject, que se establece mediante **GameObject.SetActive**. Ten en cuenta que un gameObject puede estar localmente activo pero estar inactivo en la jerarquía porque un padre no está activo.
- **isStatic**: Define si el gameObject es estático o no. Un objeto estático no se va a mover en ningún momento en modo de ejecución y va a permitir utilizar funcionalidades como almacenar la iluminación, las sombras y entrar en un grupo de cálculo llamado **static batching**.
- **layer**: Devuelve el layer en el que está el gameObject. Se pueden usar los Layers o capas para renderizar selectivamente desde cámaras o para ignorar Raycasts.
- **scene:** Devuelve la escena de la que el gameObject forma parte.
- **sceneCullingMask**: Indica el culling mask que Unity utiliza para determinar en qué escena renderizar el gameObject.
- **tag**: Devuelve la etiqueta del gameObject.
- **transform**: Devuelve el componente Transform del gameObject.

Constructores

- **GameObject**: Crea una nueva instancia de la clase GameObject. Podemos ver un ejemplo en el punto "**Accediendo a la clase GameObject**" del capítulo "**Componentes Gameobject y Transform**".

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Métodos públicos

Método AddComponent

Declaración:

```
public Component AddComponent(Type componentType);
```

Descripción:
Añade un componente de tipo *componentType* al gameObject.

Declaración:

```
public T AddComponent();
```

Descripción:
Versión con genéricos.

Hemos visto cómo utilizar esta función en el capítulo "**Buscar, obtener y crear componentes**", en el punto "**Añadir componentes en tiempo de ejecución**".

Método CompareTag

Declaración:

```
public bool CompareTag(string tag);
```

Descripción:
Devuelve si el gameObject tiene la etiqueta *tag*. Recibe como parámetro el tag o etiqueta a comparar.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Ejemplo:

```
using UnityEngine;

public class Enemy : MonoBehaviour
{
    void OnTriggerEnter(Collider other)
    {
        if(other.gameObject.CompareTag("Player"))
        {
            Destroy(gameObject);
        }
    }
}
```

En este ejemplo, si el objeto que colisiona con el **Enemy** tiene el tag **Player**, se destruye.

Método GetComponent

Declaración:

```
public Component GetComponent(Type type);
```

Descripción:
Devuelve el componente de tipo *type* que hay en el gameObject. Si no tiene ningún componente de tipo *type* devuelve nulo.

El uso de **gameObject.GetComponent** devolverá el primer componente encontrado sin ningún orden definido. Si esperas que haya más de un componente del mismo tipo, utiliza **gameObject.GetComponents** en su lugar, y puedes utilizar algún tipo de loop sobre los componentes recibidos en busca de alguna propiedad única.

Declaración:

```
public T GetComponent();
```

Descripción:
Versión con genéricos.
Hemos visto cómo utilizar esta función en el capítulo "**Buscar, obtener y crear componentes**", en el punto "**Obteniendo un componente del gameObject**"

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Método GetComponentInChildren

Declaración:

```
public Component GetComponentInChildren(Type type);
```

Descripción:
Devuelve el componente de tipo *type* en el gameObject o cualquiera de sus hijos. El componente solo es devuelto si es encontrado en un gameObject activo.

Declaración:

```
public Component GetComponentInChildren(Type type, bool includeInactive);
```

Descripción:
Si la variable **includeInactive** tiene valor **true**, el componente es devuelto incluso si el gameObject está desactivado.

Declaración:

```
public T GetComponentInChildren(bool includeInactive = false);
```

Descripción:
Versión con genéricos.

Hemos visto cómo utilizar esta función en el capítulo "**Buscar, obtener y crear componentes**", en el punto "**Obteniendo un componente del hijo de un gameObject**".

Método GetComponentInParent

Declaración:

```
public Component GetComponentInParent(Type type);
```


Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Descripción:
Devuelve el componente de tipo *type* en el gameObject o cualquiera de sus padres. El componente solo es devuelto si es encontrado en un gameObject activo.

Declaración:

```
public Component GetComponentInParent(Type type, bool includeInactive);
```

Descripción:
Si la variable **includeInactive** tiene valor **true**, el componente es devuelto incluso si el gameObject está desactivado.

Declaración:

```
public T GetComponentInParent(bool includeInactive = false);
```

Descripción:
Versión con genéricos.

Hemos visto cómo utilizar esta función en el capítulo "**Buscar, obtener y crear componentes**", en el punto "**Obteniendo componentes de padres de jerarquía compleja del gameObject**".

Método GetComponents

Declaración:

```
public Component[] GetComponents(Type type);
```

Descripción:
Devuelve todos los componentes de tipo type que hay en el gameObject.

Declaración:

```
public T[] GetComponents();
```


Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Descripción:
Versión con genéricos.

Hemos visto cómo utilizar esta función en el capítulo "**Buscar, obtener y crear componentes**", en el punto "**Obteniendo varios componentes del gameObject**".

Método GetComponentsInChildren

Declaración:

```
public Component[] GetComponentsInChildren(Type type,
                                           bool includeInactive = false);
```

Descripción:
Devuelve los componentes de tipo *type* en el gameObject o cualquiera de sus hijos. Si la variable **includeInactive** tiene valor **true**, el componente es devuelto incluso si el gameObject está desactivado.

Declaración:

```
public T[] GetComponentsInChildren();
```

Descripción:
Versión con genéricos. El componente solo es devuelto si es encontrado en un gameObject activo.

Declaración:

```
public T[] GetComponentsInChildren(bool includeInactive);
```

Descripción:
Versión con genéricos. Si la variable **includeInactive** tiene valor **true**, el componente es devuelto incluso si el gameObject está desactivado.

Hemos visto cómo utilizar esta función en el capítulo "**Buscar, obtener y crear componentes**", en el punto "**Obteniendo varios componentes de los hijos de un gameObject**".

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Método GetComponentsInParent

Declaración:

```
public Component[] GetComponentsInParent(Type type,
                                         bool includeInactive = false);
```

Descripción:
Devuelve los componentes de tipo *type* en el gameObject o cualquiera de sus padres. Si la variable **includeInactive** tiene valor **true**, el componente es devuelto incluso si el gameObject está desactivado.

Declaración:

```
public T[] GetComponentsInParent();
```

Descripción:
Versión con genéricos. El componente solo es devuelto si es encontrado en un gameObject activo.

Declaración:

```
public T[] GetComponentsInParent(bool includeInactive);
```

Descripción:
Versión con genéricos. Si la variable **includeInactive** tiene valor **true**, el componente es devuelto incluso si el gameObject está desactivado.

Hemos visto cómo utilizar esta función en el capítulo "**Buscar, obtener y crear componentes**", en el punto "**Obteniendo componentes de padres de jerarquía compleja del gameObject**".

Método BroadcastMessage

Declaración:

```
public void BroadcastMessage(string methodName,
                             object parameter = null,
                             SendMessageOptions options = SendMessageOptions.RequireReceiver);
```


Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

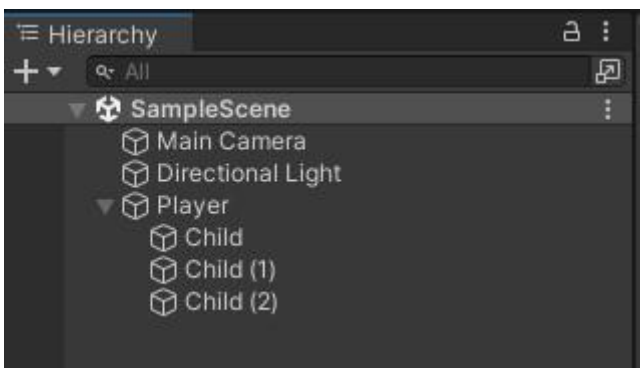
Descripción:

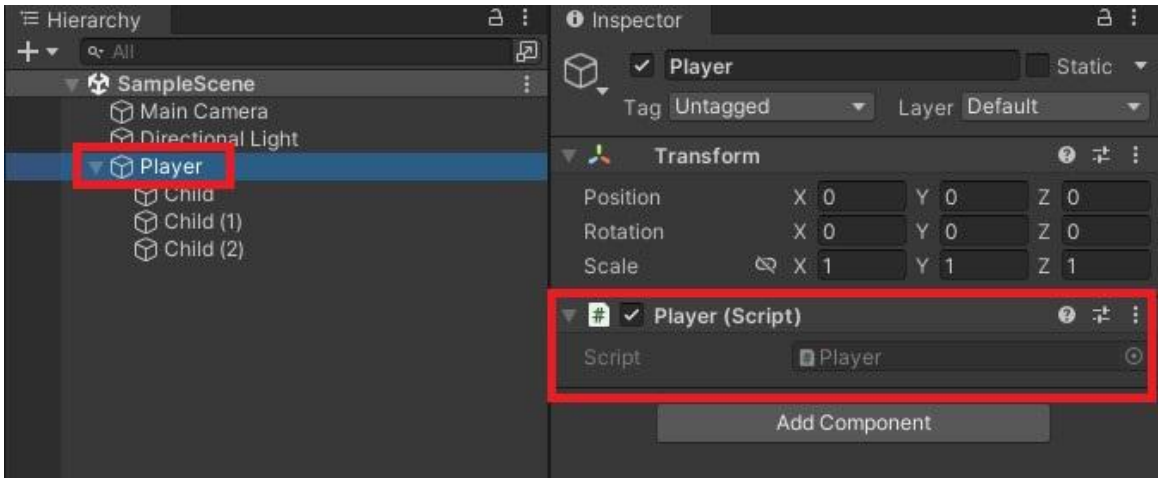
Llama al método *methodName* en cada componente que hereda de **MonoBehaviour** que haya en este GameObject o en cualquiera de sus hijos.

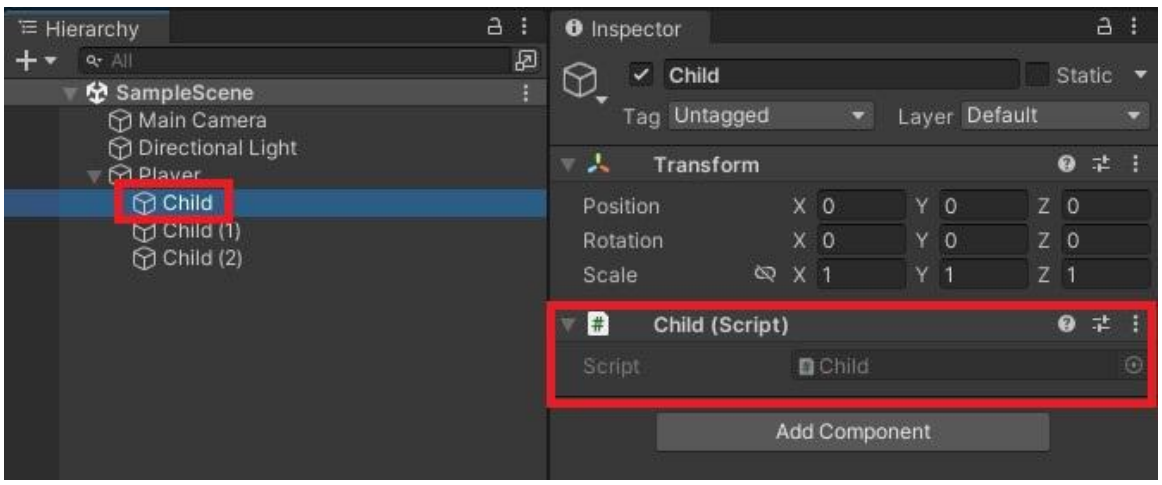
El método receptor puede elegir ignorar el parámetro *parameter* no teniendo parámetros en su declaración. Si *options* tiene el valor **SendMessageOptions.RequireReceiver** se imprimirá un error si el mensaje no es recibido por ningún componente.

Ejemplo:

Tenemos un gameObject llamado **Player** con un componente también llamado **Player**. Este gameObject tiene tres hijos llamados **Child**, **Child (1)** y **Child (2)**, que tienen un componente llamado **Child**.







Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

El código del componente **Player** es:

```
using UnityEngine;

public class Player : MonoBehaviour
{
    void Start()
    {
        BroadcastMessage("SayHello", SendMessageOptions.RequireReceiver);
    }

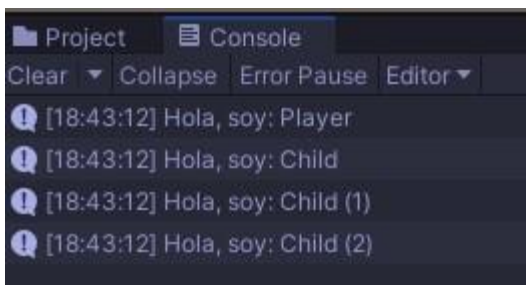
    public void SayHello()
    {
        Debug.Log("Hola, soy: " + name);
    }
}
```

El código del componente **Child** es:

```
using UnityEngine;

public class Child : MonoBehaviour
{
    public void SayHello()
    {
        Debug.Log("Hola, soy: " + name);
    }
}
```

Al ejecutar el programa, utilizando **BroadcastMessage** en el componente **Player** del gameObject **Player** se llamará a la función *SayHello* del propio gameObject Player y de los hijos.



Método SendMessage

Declaración:

```
public void SendMessage(string methodName,
                object value = null,
                SendMessageOptions options = SendMessageOptions.RequireReceiver);
```


Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

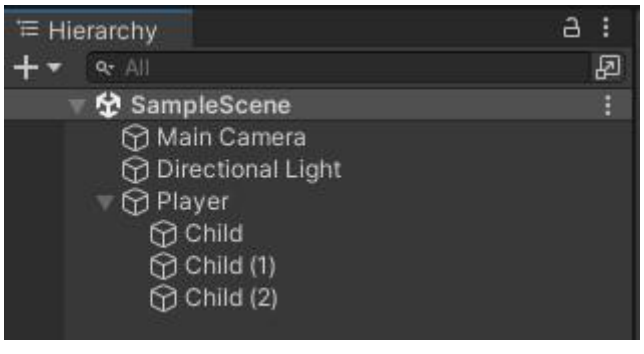
Descripción:

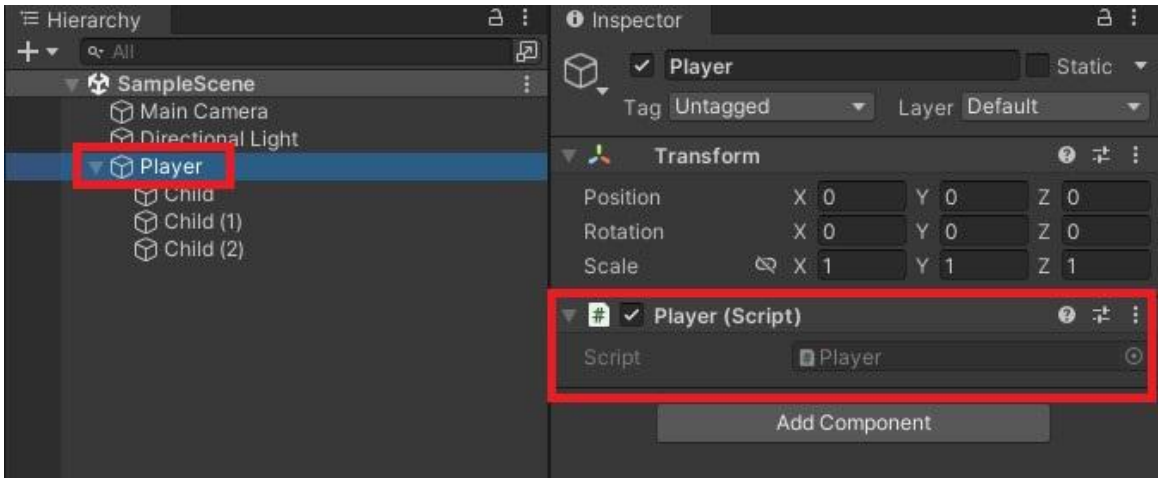
Llama al método *methodName* en cada componente que hereda de MonoBehaviour que haya en este GameObject. A diferencia de **BroadcastMessage**, el mensaje no se envía a los hijos.

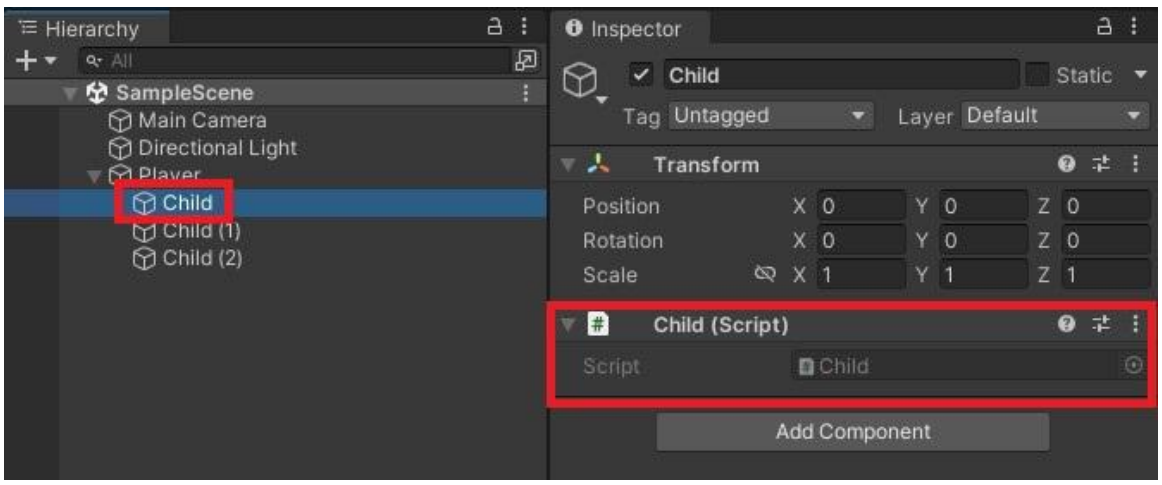
El método receptor puede elegir ignorar el parámetro no teniendo parámetros en su declaración. Si *options* tiene el valor **SendMessageOptions.RequireReceiver** se imprimirá un error si el mensaje no es recibido por ningún componente.

Ejemplo:

Tenemos un gameObject llamado **Player** con un componente también llamado **Player**. Este gameObject tiene tres hijos llamados **Child**, **Child (1)** y **Child (2)**, que tienen un componente llamado **Child**.







Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

El código del componente **Player** es:

```
using UnityEngine;

public class Player : MonoBehaviour
{
    void Start()
    {
        SendMessage("SayHello", SendMessageOptions.RequireReceiver);
    }

    public void SayHello()
    {
        Debug.Log("Hola, soy: " + name);
    }
}
```

El código del componente **Child** es:

```
using UnityEngine;

public class Child : MonoBehaviour
{
    public void SayHello()
    {
        Debug.Log("Hola, soy: " + name);
    }
}
```

Al ejecutar el programa, utilizando **SendMessage** en el componente **Player** del gameObject **Player** se llamará a la función *SayHello* del propio gameObject *Player* pero no a la función de los hijos.



Método SendMessageUpwards

Declaración:

```
public void SendMessageUpwards(string methodName,
            object value = null,
            SendMessageOptions options = SendMessageOptions.RequireReceiver);
```


Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

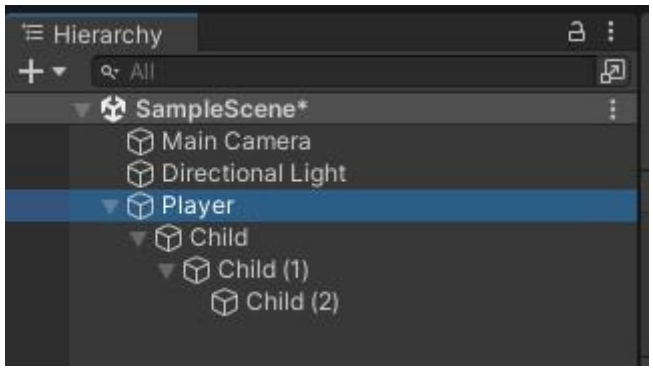
Descripción:

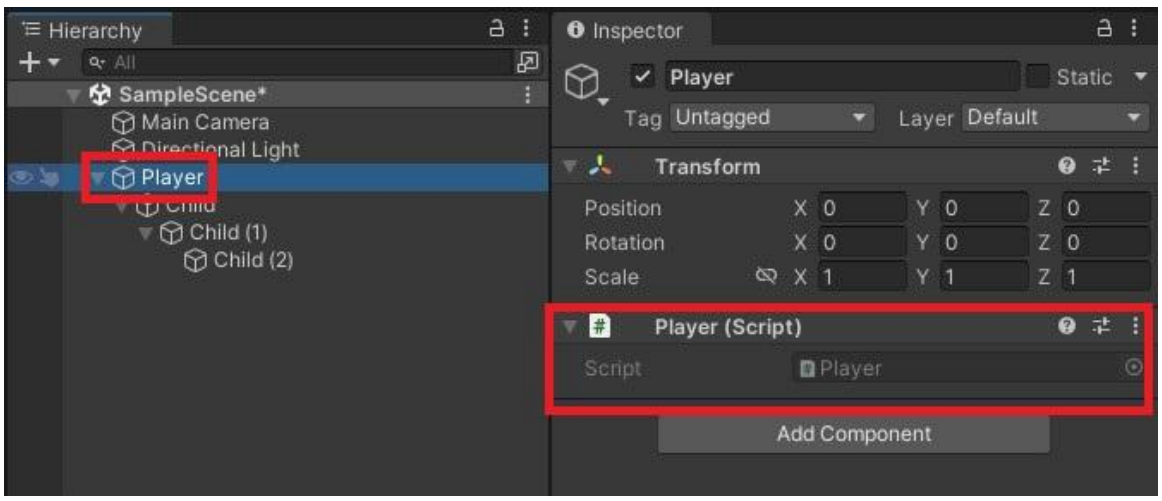
Llama al método *methodName* en cada componente que hereda de **MonoBehaviour** que haya en este GameObject o en cualquiera de sus padres.

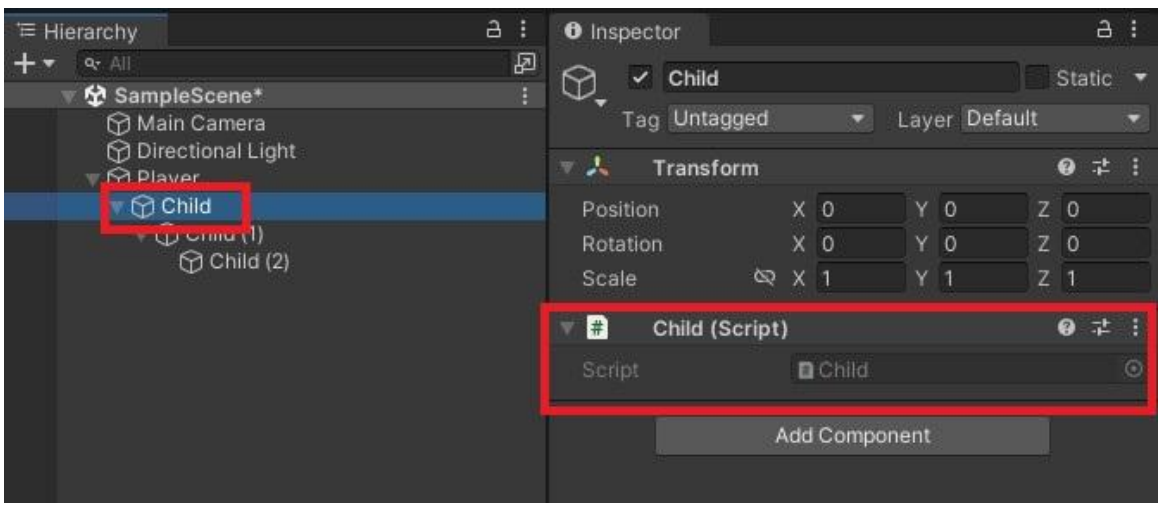
El método receptor puede elegir ignorar el parámetro no teniendo parámetros en su declaración. Si *options* tiene el valor **SendMessageOptions.RequireReceiver** se imprimirá un error si el mensaje no es recibido por ningún componente.

Ejemplo:

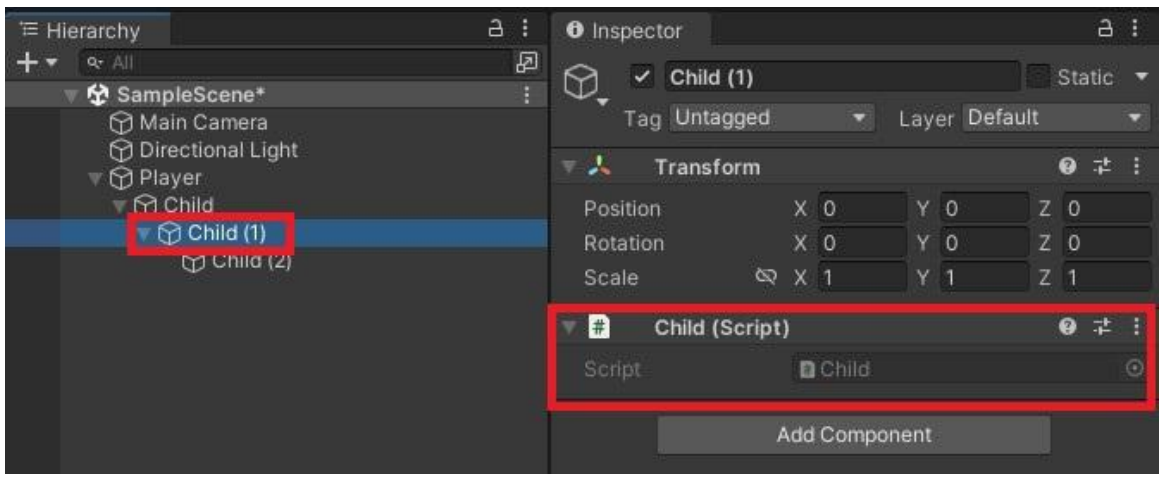
Tenemos un gameObject llamado **Player** con un componente también llamado **Player**. Este gameObject tiene un hijo llamado **Child**, que a su vez tiene un hijo llamado **Child (1)**, que a su vez tiene un hijo llamado **Child (2)**. Los componentes **Child** y **Child (1)** tienen un componente llamado **Child**, y el componente **Child (2)** tiene un componente llamado **ChildSendMessage**.

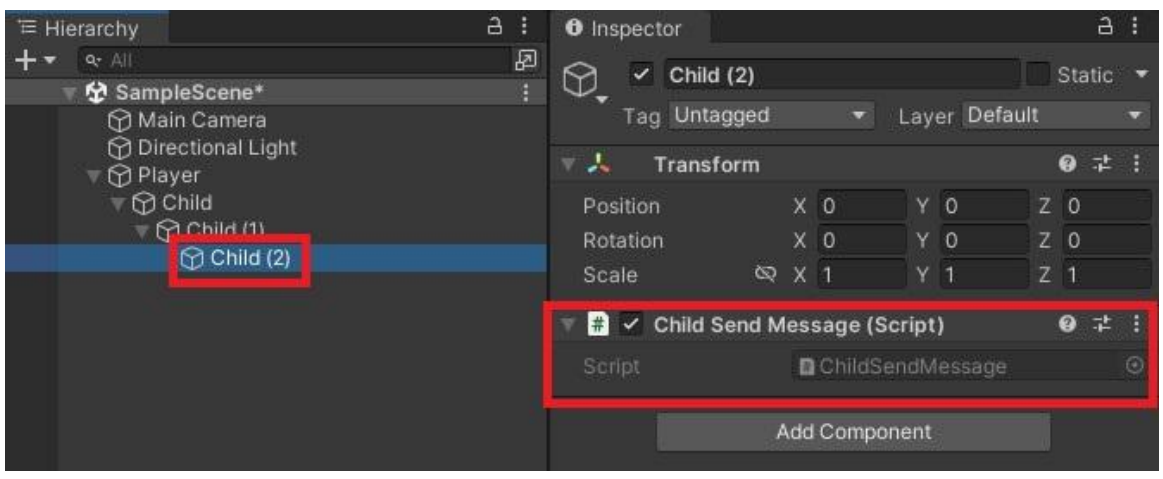






Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143





El código del componente **Player** es:

```
1 using UnityEngine;
2
3 public class Player : MonoBehaviour
4 {
5     public void SayHello()
6     {
7         Debug.Log("Hola, soy: " + name);
8     }
9 }
```

El código del componente **Child** es:

```
1 using UnityEngine;
2
3 public class Child : MonoBehaviour
4 {
5     public void SayHello()
6     {
7         Debug.Log("Hola, soy: " + name);
8     }
9 }
```


Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

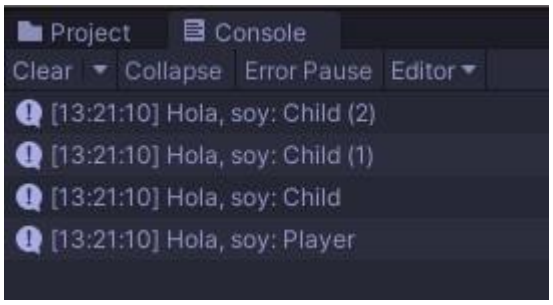
El código del componente **ChildSendMessage** es

```
using UnityEngine;

public class ChildSendMessage : MonoBehaviour
{
    void Start()
    {
        SendMessageUpwards("SayHello",
                    SendMessageOptions.RequireReceiver);
    }

    public void SayHello()
    {
        Debug.Log("Hola, soy: " + name);
    }
}
```

Al ejecutar el programa, utilizando **SendMessageUpwards** desde el componente **ChildSendMessage** del gameObject **Child (2)** se llamará a la función *SayHello* del propio gameObject *Child (2)* y de los padres.



Método SetActive

Declaración:

```
public void SetActive(bool value);
```

Descripción:
Activa o desactiva el gameObject en función del valor que contenga *value*.

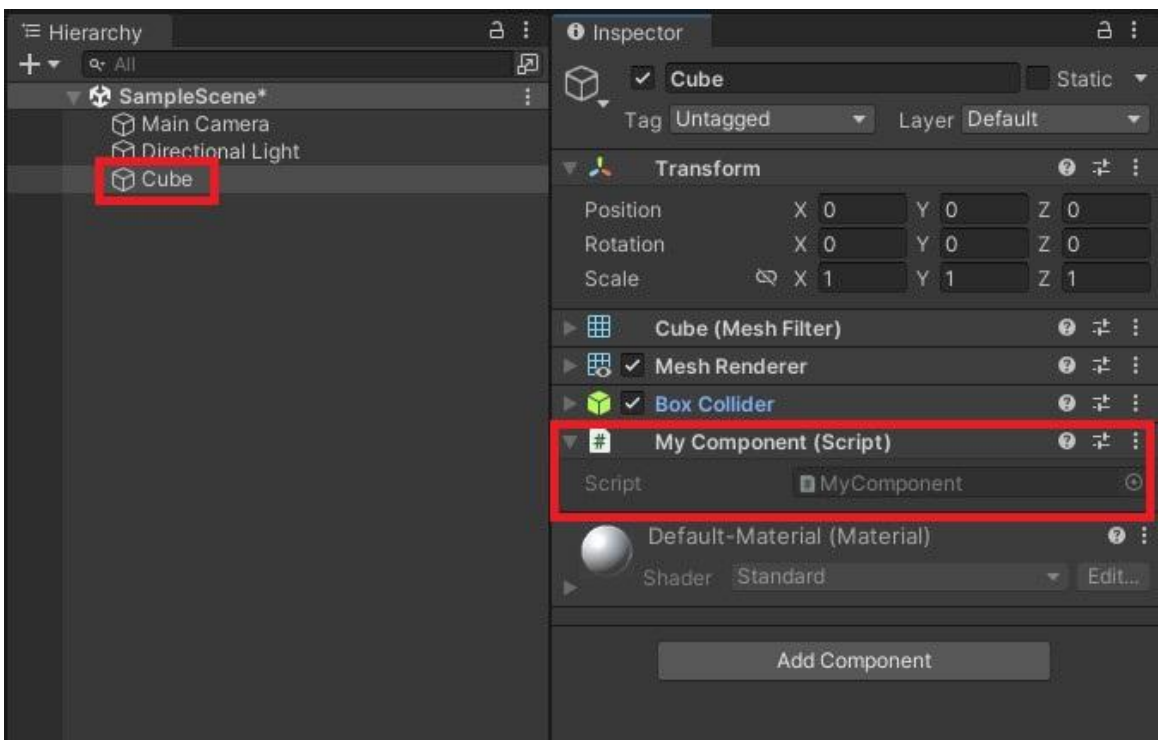
Un gameObject puede estar inactivo porque un padre no está activo. En este caso, llamar a la función SetActive no lo activará, sino que establecerá el estado local del gameObject. Este estado se puede consultar utilizando **GameObject.activeSelf**.

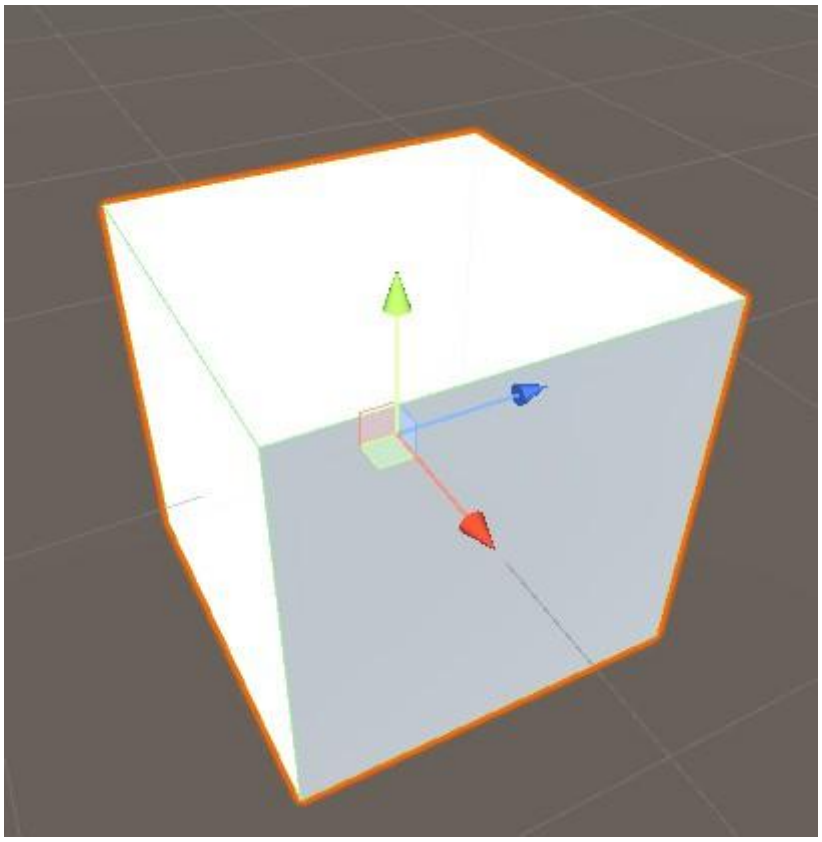
Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Cuando se desactiva un gameObject, se desactiva cada componente, incluídos renderers adjuntos, colliders, rigidbodies y scripts. Por ejemplo, Unity ya no llamará a la función Update de un script adjunto a un gameObject desactivado. Cuando el gameObject se activa con **SetActive(true)**, el método **OnEnable** es llamado. Y cuando el gameObject se desactiva con **SetActive(false)**, el método llamado es **OnDisable**.

Ejemplo:

Tenemos un gameObject de tipo cubo llamado **Cube**, con un componente llamado **MyComponent**.





Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

El código del componente **myComponent** es:

```
using UnityEngine;

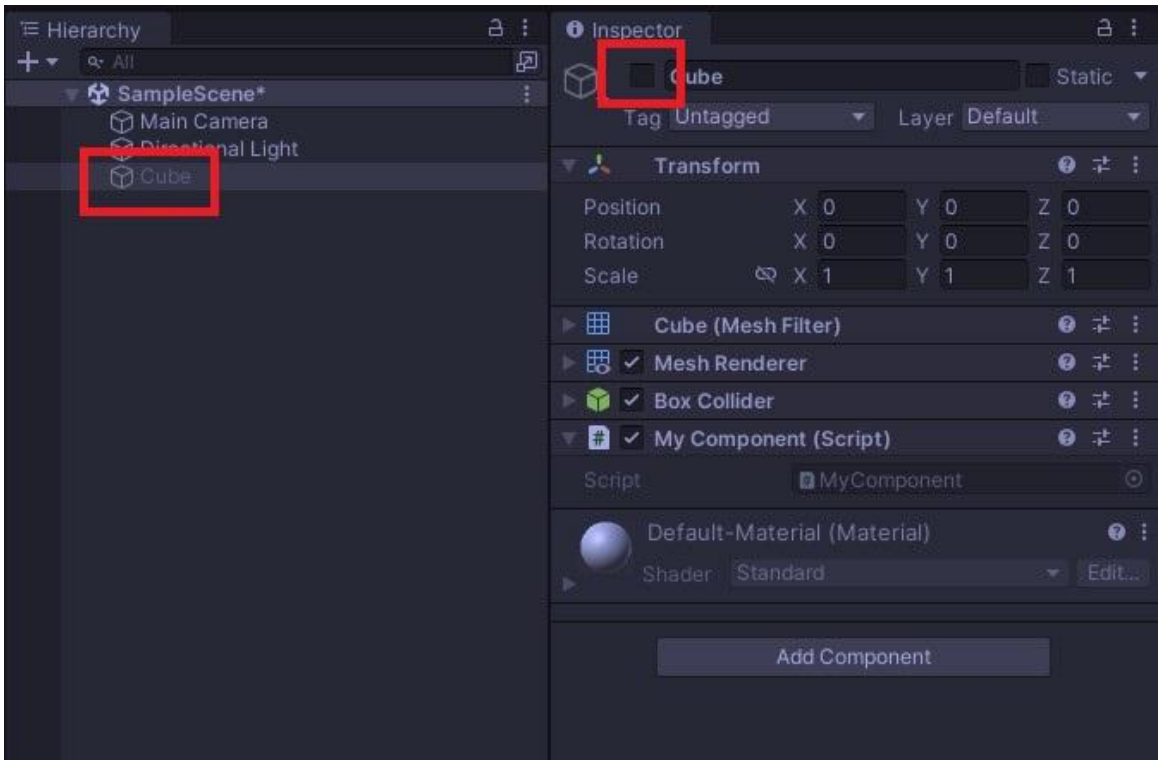
public class MyComponent : MonoBehaviour
{
    MyComponent myComponent;

    void Start()
    {
        myComponent = FindObjectOfType<MyComponent>();
        myComponent.gameObject.SetActive(false);
    }
}
```

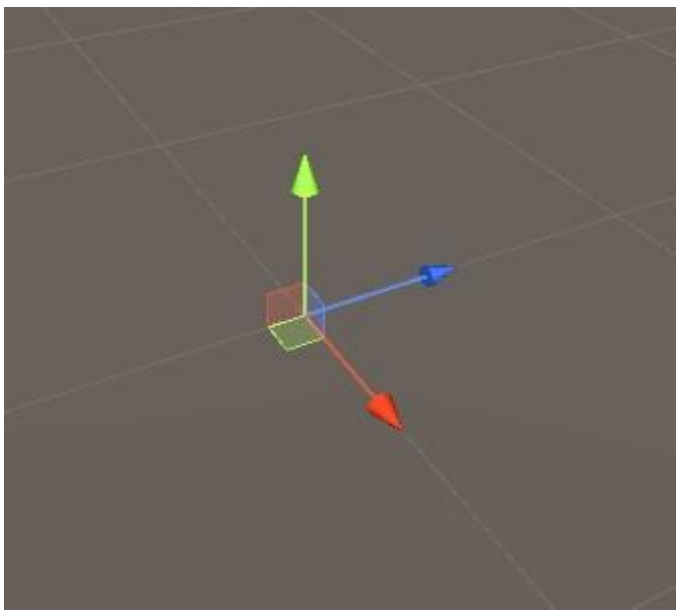
En este código:

- Declaramos una variable de tipo **MyComponent** llamada *myComponent*.
- En el método **Start**, le damos valor a la variable *myComponent* haciendo uso de la función **FindObjectOfType** de la clase **Object** que permite buscar un objeto del tipo indicado.
- Finalmente, desactivamos el gameObject asociado al componente buscado.

Cuando ejecutamos este programa, vemos que el gameObject cube se desactiva.



Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143



Método TryGetComponent

Declaración:

```
public bool TryGetComponent(out T component);
```

Descripción:

Obtiene el componente del tipo especificado, si existe.

Devuelve **true** si el componente existe y **false** si no existe.

La variable **component** tendrá el componente si existe, y si no, tendrá el valor null.

Ejemplo:

Tenemos un gameObject llamado **Enemy** con dos componentes **Enemy** y **Health**. Desde el componente **Enemy** vamos a intentar obtener el componente **Health**.

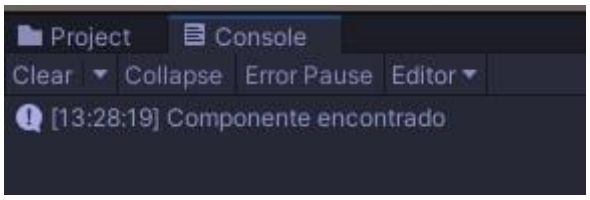
Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

```cs
using UnityEngine;

public class Enemy : MonoBehaviour
{
    Health health;

    void Start()
    {
        if (TryGetComponent<Health>(out health))
        {
            Debug.Log("Componente encontrado");
        }
        else
        {
            Debug.Log("Componente NO encontrado");
        }
    }
}
```

Cuando se ejecuta el programa, vemos que encuentra el componente.



Métodos estáticos

Método CreatePrimitive

Declaración:

```cs
public static GameObject CreatePrimitive(PrimitiveType type);
```

Descripción:

Crea un gameObject con una geometría primitiva y el BoxCollider adecuado.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

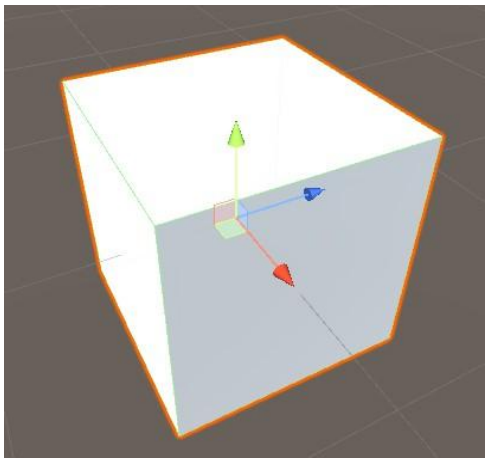
Ejemplo:

```
using UnityEngine;

public class Player : MonoBehaviour
{
    void Start()
    {
        GameObject myCube;
        myCube = GameObject.CreatePrimitive(PrimitiveType.Cube);
        myCube.name = "CuboCreatePrimitive";
    }
}
```

Al ejecutar el programa, se crea un cubo y se le asigna el nombre "*CuboCreatePrimitive*".





Método Find

Declaración:

```
public static GameObject Find(string name);
```

Descripción:

Encuentra un gameObject por el parámetro *name* y lo devuelve. Esta función sólo devuelve gameObjects activos. Si no se encuentra ningún gameObject con el nombre *name*, se devuelve null.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Es una búsqueda rápida si sabemos el nombre del gameObject que queremos buscar, pero tenemos que tener cuidado porque si cometemos alguna falta de ortografía al indicar el nombre o si cambiamos el nombre en algún momento, la búsqueda no funcionará y obtendremos el valor null.

Por razones de rendimiento, se recomienda no utilizar esta función en cada frame.

Ejemplo:

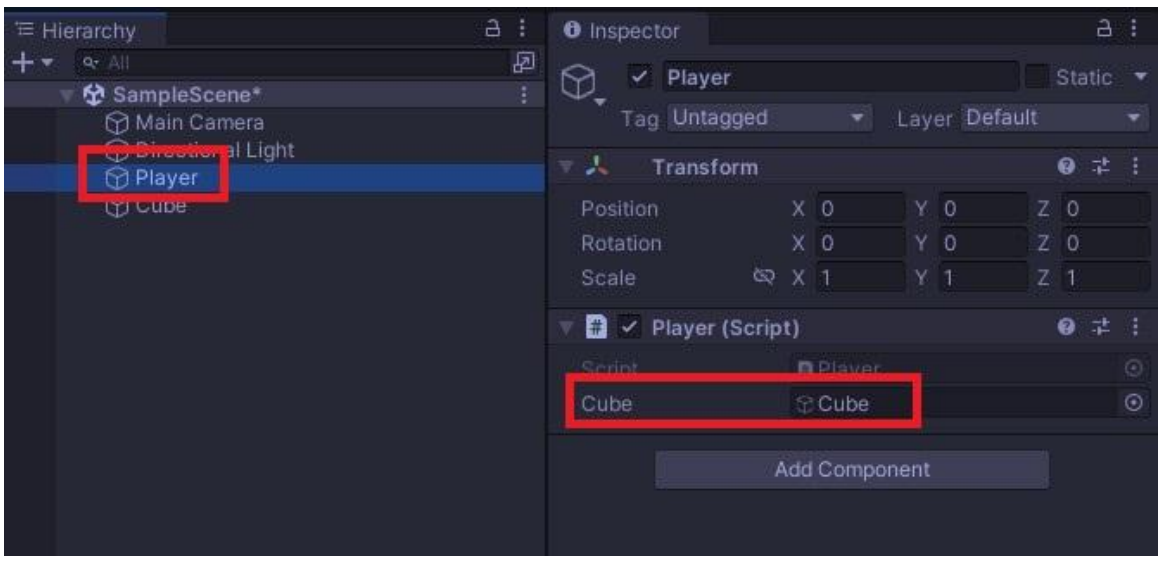
Tenemos un gameObject llamado **Player** con un componente llamado **Player** y un gameObject llamado **Cube**. Desde el **Player** vamos a buscar el **Cube**.

```using UnityEngine;

public class Player : MonoBehaviour
{
    [SerializeField] GameObject cube;

    void Start()
    {
        cube = GameObject.Find("Cube");
    }
}
```

Al ejecutar el programa, vemos que desde el gameObject Player se ha encontrado el gameObject Cube.



Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Método FindGameObjectsWithTag

Declaración:

```
public static GameObject[] FindGameObjectsWithTag(string tag)
```

Descripción:
Devuelve un array de gameObjects activos que tienen la etiqueta *tag*. Si ningún gameObject es encontrado, devuelve un array vacío.

Las etiquetas deben declararse en el gestor de etiquetas antes de utilizarlas. Se lanzará una excepción de Unity si la etiqueta no existe o se pasa una cadena vacía o null como etiqueta.

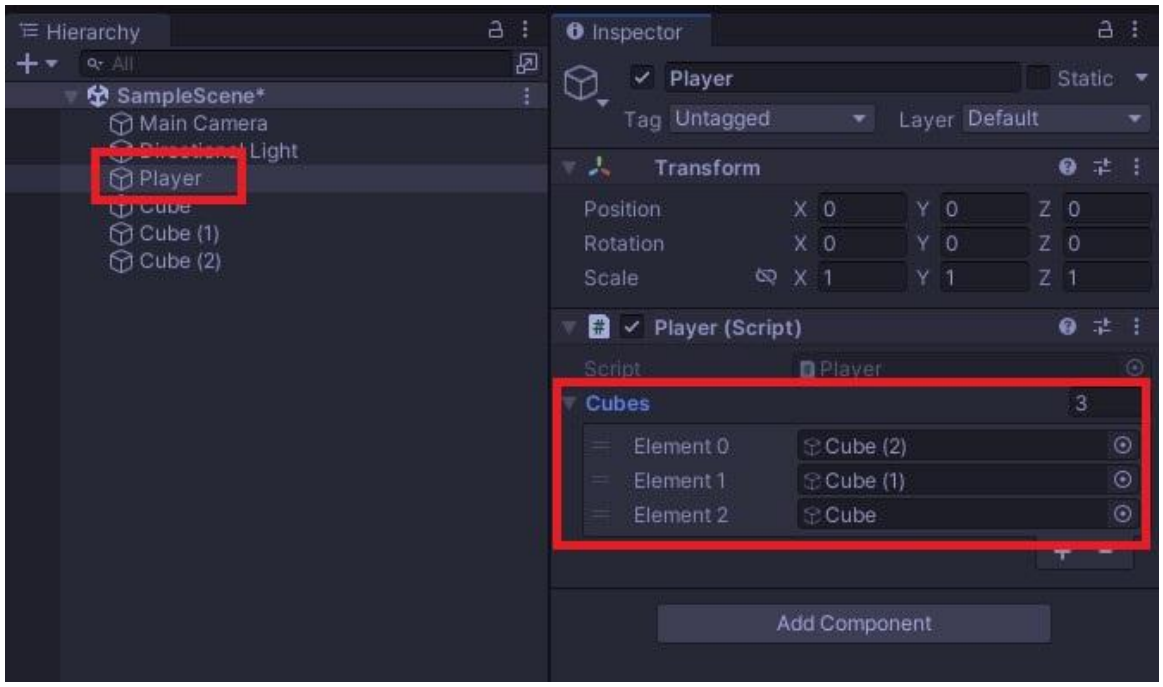
Ejemplo:

Tenemos un gameObject llamado **Player** con un componente llamado **Player** y varios gameObject de tipo **Cube**. Los gameObject de tipo **Cube** tienen asignada la etiqueta "**TestTag**". Desde el **Player** vamos a buscar todos los componentes de tipo **Cube**.

```
1 using UnityEngine;
2
3 public class Player : MonoBehaviour
4 {
5     [SerializeField] GameObject[] cubes;
6
7     void Start()
8     {
9         cubes = GameObject.FindGameObjectsWithTag("TestTag");
10    }
11 }
```

Al ejecutar el programa, vemos que desde el gameObject Player se han encontrado los gameObjects de tipo Cube.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143



Método FindWithTag

Declaración:

```
public static GameObject FindWithTag(string tag);
```

Descripción:
Devuelve un único gameObject activo que tenga la etiqueta tag. Si ningún gameObject es encontrado, devuelve null.

Las etiquetas deben declararse en el gestor de etiquetas antes de utilizarlas. Se lanzará una excepción de Unity si la etiqueta no existe o se pasa una cadena vacía o null como etiqueta.
Este método devuelve el primer gameObject que encuentra con la etiqueta especificada. Si una escena contiene múltiples gameObjects activos con la etiqueta especificada, no hay garantía de que este método devuelva un gameObject específico.

Ejemplo:

Tenemos un gameObject llamado **Player** con un componente llamado **Player** y un gameObject llamado **Cube**. El gameObject **Cube** tiene asignada la etiqueta "**TestTag**". Desde el **Player** vamos a buscar el **Cube**.

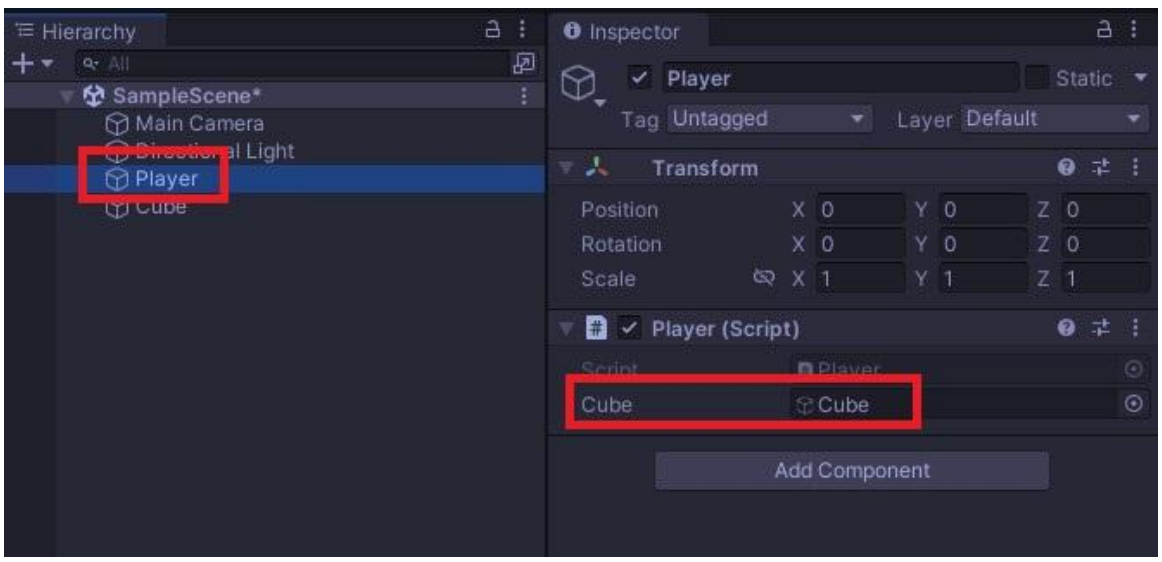
Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

```cs
using UnityEngine;

public class Player : MonoBehaviour
{
    [SerializeField] GameObject cube;

    void Start()
    {
        cube = GameObject.FindWithTag("TestTag");
    }
}
```

Al ejecutar el programa, vemos que desde el gameObject Player se ha encontrado el gameObject Cube.



Miembros heredados

Veremos el detalle de estas propiedades y métodos en la clase correspondiente. Las nombramos aquí para que sepas que, al ser miembros heredados, están disponibles en cualquier gameObject.

Propiedades

• hideFlags	heredada de la clase Object
• name	heredada de la clase Object

Métodos públicos

• GetInstanceID	heredado de la clase Object
• ToString	heredado de la clase Object

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Métodos estáticos

• Destroy	heredado de la clase Object
• DestroyImmediate	heredado de la clase Object
• DontDestroyOnLoad	heredado de la clase Object
• FindObjectOfType	heredado de la clase Object
• FindObjectsOfType	heredado de la clase Object
• Instantiate	heredado de la clase Object

Operadores

• bool	heredado de la clase Object
• operator !=	heredado de la clase Object
• operator ==	heredado de la clase Object

Link a la documentación:

https://docs.unity3d.com/2022.1/Documentation/ScriptReference/GameObject.html

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Clase Transform

- Clase de UnityEngine
- Hereda de Component
- Implementada en UnityEngine.CoreModule

Descripción

Posición, rotación y escala de un objeto.

Cada objeto en una escena tiene un componente de tipo **Transform**. Se utiliza para almacenar y manipular la posición, rotación y escala del objeto. Cada componente de tipo **Transform** puede tener un padre, lo que permite aplicar la posición, rotación y escala jerárquicamente. Esta jerarquía se ve en el panel Jerarquía.

Soporta **enumerators**, por lo que se pueden recorrer los hijos utilizando por ejemplo un bucle **foreach**:

```
using UnityEngine;

public class TransformScript : MonoBehaviour
{
    void Start()
    {
        foreach (Transform child in transform)
        {
            Debug.Log(child.position.x);
        }
    }
}
```

Propiedades

- **childCount**: Indica el número de hijos (activos o inactivos) que tiene el componente Transform. El padre no está incluído en la cuenta.
- **eulerAngles**: Indica la rotación expresada como ángulos euler en grados. Representa la rotación en el espacio global.
- **forward**: Devuelve un vector normalizado que representa la flecha **azul** del manejador de Unity y equivale al **eje z**. Indica hacia adelante y equivale a: x = 0, y = 0, z = 1.
- **localEulerAngles**: Indica la rotación expresada como ángulos euler en grados. Representa la rotación relativa al padre dentro de una jerarquía.
- **localPosition**: Indica la posición de la Transform relativa al padre dentro de una jerarquía.
- **localRotation**: Indica un Quaternion que almacena la rotación relativa al padre dentro de una jerarquía.
- **localScale**: Indica la escala de la Transform relativa al padre dentro de una jerarquía.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

- **lossyScale**: Indica la escala global del objeto. Es una propiedad de solo lectura.
- **parent**: El padre del componente Transform. Cambiar el padre modificará la posición, rotación y escala relativa al padre, pero mantendrá la misma posición, rotación y escala en el espacio global.
- **position**: Indica la posición de la Transform en el espacio global.
- **right**: Devuelve un vector normalizado que representa la flecha **roja** del manejador de Unity y equivale al **eje x**. Indica hacia la derecha y equivale a: x = 1, y = 0, z = 0.
- **rotation**: Indica un Quaternion que almacena la rotación en el espacio global.
- **up**: Devuelve un vector normalizado que representa la flecha **verde** del manejador de Unity y equivale al **eje y**. Indica hacia arriba y equivale a: x = 0, y = 1, z = 0.

Métodos públicos

Método DetachChildren

Declaración:

```
public void DetachChildren();
```

Descripción:

Rompe la jerarquía y le quita todos los hijos a un padre.

Ejemplo:

```
1 using UnityEngine;
2
3 public class TransformScript : MonoBehaviour
4 {
5     GameObject child;
6     Transform parent;
7
8     void Start()
9     {
10        child.transform.SetParent(parent);
11        parent.DetachChildren();
12    }
13 }
```

En este ejemplo, se declara una variable de tipo **GameObject** llamada *child*, y una variable de tipo **Transform** llamada *parent*.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

En el método **Start**:

- Se establece la Transform *parent* como padre del gameObject *child*.
- Posteriormente, se rompe esa jerarquía.

Método Find

Declaración:

```
public Transform Find(string n);
```

Descripción:
El método **Find** busca un hijo (aunque esté deshabilitado) con nombre *n* y lo devuelve. Si no encuentra ninguno, devuelve **null**.

Nota: **Find** no realiza un descenso recursivo por una jerarquía de Transform.

Ejemplo:

```
using UnityEngine;

public class TransformScript : MonoBehaviour
{
    GameObject child;
    Transform parent;
    Transform childFound;

    void Start()
    {
        child.transform.SetParent(parent);
        child.name = "Hijo";
        childFound = parent.Find("Hijo");
    }
}
```

En este ejemplo, se declara una variable de tipo **GameObject** llamada *child*, una variable de tipo **Transform** llamada *parent* y una variable de tipo **Transform** llamada *childFound*.

En el método **Start**:

- Se establece la Transform *parent* como padre del gameObject *child*.
- Se asigna el nombre “Hijo” al gameObject *child*.
- Se asigna valor a la variable *childFound* utilizando la función **Find**.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Método GetChild

Declaración:

```
public Transform GetChild(int index);
```

Descripción:
Devuelve el hijo del componente Transform que corresponde al índice *index*.

El parámetro *index* debe de ser menor que el valor de **Transform.childCount**, que es el número total de hijos. Si el componente Transform no tiene ningún hijo, o el argumento *index* tiene un valor superior al número de hijos, se generará un error. En este caso se producirá el error "**Transform child out of bounds**".

Ejemplo:

```
using UnityEngine;

public class TransformScript : MonoBehaviour
{
    GameObject child;
    Transform parent;

    void Start()
    {
        child.transform.SetParent(parent);
        parent.transform.GetChild(0).gameObject.SetActive(true);
    }
}
```

En este ejemplo, se declara una variable de tipo **GameObject** llamada *child*, y una variable de tipo **Transform** llamada *parent*.

En el método **Start**:

- Se establece la Transform *parent* como padre del gameObject *child*.
- Se activa el primer hijo de la Transform *parent*.

Método GetSiblingIndex

Declaración:

```
public int GetSiblingIndex();
```


Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Descripción:

Si un gameObject comparte un padre con otros gameObjects y están en el mismo nivel (es decir, comparten el mismo padre directo), estos gameObjects son conocidos como hermanos. La función **GetSiblingIndex** devuelve el índice que tiene el gameObject dentro de este parentesco.

Si el gameObject no está en ningún parentesco, esta función devolverá el índice que tiene el gameObject dentro de la lista jerárquica de la escena.

Ejemplo:

```
using UnityEngine;

public class TransformScript : MonoBehaviour
{
    GameObject child;

    void Start()
    {
        child = new GameObject();
        int index = child.transform.GetSiblingIndex();
        Debug.Log(index);
    }
}
```

En este ejemplo, se declara una variable de tipo **GameObject** llamada *child*.

En el método **Start**:

- Se crea un nuevo gameObject utilizando el constructor de la clase **GameObject**.
- Se declara una variable de tipo **int** llamada *index*, y se le asigna el valor con la función **GetSiblingIndex**. Como el gameObject *child* no está dentro de ningún parentesco, el índice devuelto será el índice que tenga el gameObject *child* dentro de la lista jerárquica de la escena.
- Se muestra por consola el valor de la variable *index*.

Método IsChildOf

Declaración:

```
public bool IsChildOf(Transform parent);
```

Descripción:

¿Es este componente Transform hijo de *parent*?

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Devuelve una variable de tipo **bool** que indica si el componente **Transform** es o no hijo de *parent*. **True** si es hijo, hijo de un hijo o el propio *parent*. **False** en caso contrario.

Ejemplo:

```
using UnityEngine;

public class TransformScript : MonoBehaviour
{
    GameObject child;
    Transform parent;

    void Start()
    {
        child.transform.SetParent(parent);
        Debug.Log(child.transform.IsChildOf(parent));
    }
}
```

En este ejemplo, se declara una variable de tipo **GameObject** llamada *child*, y una variable de tipo **Transform** llamada *parent*.

En el método **Start**:

- Se establece la Transform *parent* como padre del gameObject *child*.
- Se muestra por consola si el gameObject *child* es o no hijo de la Transform *parent*.

Método LookAt

Declaración:

```
public void LookAt(Transform target);
```

Descripción:

Rota la transformación de modo que la dirección **forward** apunte siempre a la posición actual de la Transform designada por *target*.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

Ejemplo:

```
using UnityEngine;

public class TransformScript : MonoBehaviour
{
    Transform target;

    void Update()
    {
        transform.LookAt(target);
    }
}
```

En este ejemplo, se declara una variable de tipo **Transform** llamada *target*.

En el método **Update**:

- Se establece que la dirección **forward** de la Transform apunte siempre al *target*, se mueva donde se mueva el *target* o el gameObject que tenga el script.

Declaración:

```
public void LookAt(Transform target, Vector3 worldUp = Vector3.up);
```

Descripción:

Rota la transform de modo que la dirección **forward** apunte siempre a la posición actual de la Transform designada por *target* y la dirección **up** apunte a la dirección marcada por *worldUp*.

Ejemplo:

```
using UnityEngine;

public class TransformScript : MonoBehaviour
{
    Transform target;

    void Update()
    {
        transform.LookAt(target, Vector3.left);
    }
}
```


Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143

En este ejemplo, se declara una variable de tipo **Transform** llamada *target*.

En el método **Update**:

- Se establece que la dirección **forward** de la Transform apunte siempre al *target*, se mueva donde se mueva el *target* o el gameObject que tenga el script. El movimiento estará restringido a que la dirección **up** apunte a la dirección **Vector3.left**.

Declaración:

```
public void LookAt(Vector3 worldPosition);
```

Descripción:
Rota la transformación de modo que la dirección **forward** apunte siempre a la posición *worldPosition*, que es un punto definido por un **Vector3**.

Ejemplo:

```
using UnityEngine;

public class TransformScript : MonoBehaviour
{
    void Update()
    {
        transform.LookAt(Vector3.zero);
    }
}
```

En este ejemplo, en el método **Update** se establece que la dirección **forward** de la Transform apunte siempre al punto fijo establecido por **Vector3.zero**, se mueva donde se mueva el gameObject que tenga el script.

Declaración:

```
public void LookAt(Vector3 worldPosition, Vector3 worldUp = Vector3.up);
```

Descripción:
Rota la transformación de modo que la dirección **forward** apunte siempre a la posición *worldPosition*, que es un punto definido por un **Vector3**, y la dirección **up** apunte a la dirección marcada por *worldUp*.

Licensed to Carlos Enrique Navia Torres - cenaviat@gmail.com - HP25916872390143