

# Unity - Tema 7

## Herramientas

Scripting III

Javier Alegre Landáburu  
[javier.alegre@u-tad.com](mailto:javier.alegre@u-tad.com)



# Resumen del tema

- ¿Qué posibilidades tenemos al crear herramientas?
- ¿Cómo se crea una ventana?
- ¿Cómo se puede automatizar el proceso de importación de assets?



# [SerializeField]

- Permite que las variables que son privadas se vean en el inspector.



# [HideInInspector]

- Permite que las variables que son publicas no se vean en el inspector.

```
[HideInInspector]  
public int m_amm0;
```

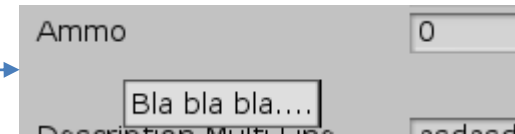
- Es conveniente usar la directiva [NonSerialized] para que Unity no tenga en cuenta posibles valores que hayamos puesto en la variable cuando si se veía en el inspector.

```
[HideInInspector]  
[NonSerialized]  
public int m_amm0;
```

# [Tooltip]

- Crea un tooltip cuando ponemos el ratón sobre la variable.

```
[Tooltip("Bla bla bla....")]  
public int m_ammo;
```



# [Range]

- Crea un slider en el editor para que la variable sólo pueda tomar valores en el rango seleccionado.

```
[Range(-5f, 5f)]  
public int m_amm0;
```



# [TextArea]

- Crea un campo de texto de más de una línea y crea un scroll en caso de que el número de líneas sea mayor que las que caben.

Weapon Description



```
[TextArea]  
public string m_weaponDescription;
```

Weapon Description

asd  
dasd

# [Multiline]

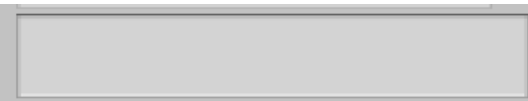
- Al igual que [TextArea] crea un campo de texto en el que se pueden meter multiples líneas, pero no crea el scroll en caso de que el número de líneas sea mayor que el tamaño del campo.

Weapon Description



```
[Multiline]  
public string m_weaponDescription;
```

Weapon Description

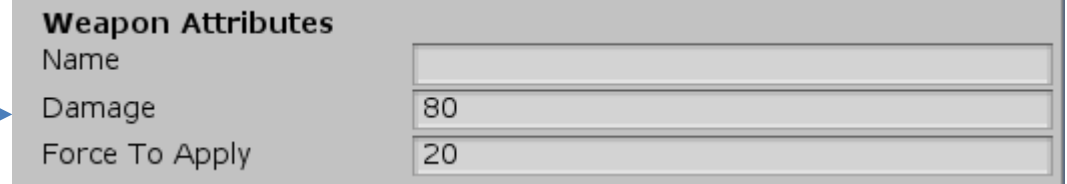




# [Header]

- Me permite crear un título en el inspector que puedo usar para diferenciar conjuntos de variables que tengan una funcionalidad parecida (o según el criterio de cada uno).

```
[Header("Weapon Attributes")]  
public string    m_name;  
public float    m_damage      = 80.0f;  
public float    m_forceToApply = 20.0f;
```



The image shows a Unity Inspector window with a custom header titled "Weapon Attributes". Below the header, there are three public variables from the script: "Name" (a string field), "Damage" (a float field with a value of 80), and "Force To Apply" (a float field with a value of 20). A blue arrow points from the code on the left to the "Damage" field in the inspector.

Weapon Attributes	
Name	
Damage	80
Force To Apply	20

# [Space]

- Crea un pequeño espacio entre variables.

Weapon Attributes	
Name	<input type="text"/>
Damage	<input type="text" value="80"/>
Force To Apply	<input type="text" value="20"/>



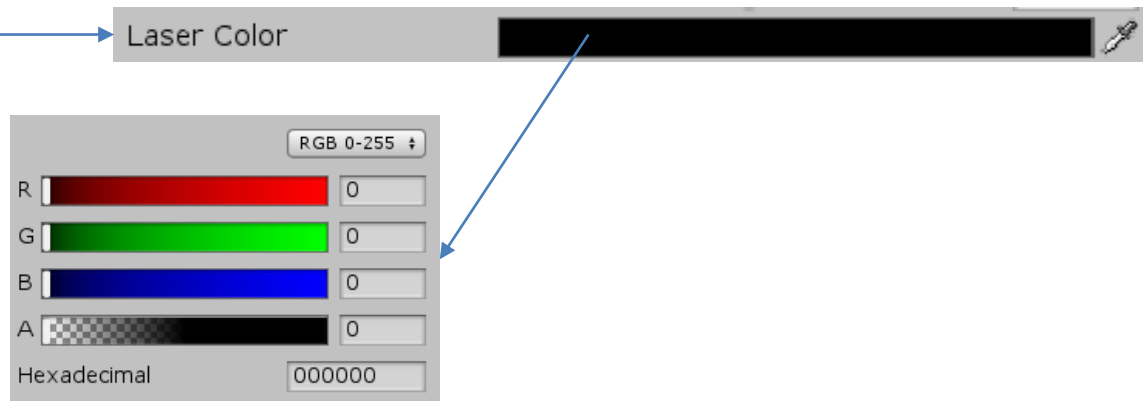
```
[Header("Weapon Attributes")]  
[Space]  
public string    m_name;  
public float    m_damage      = 80.0f;  
[Space]  
[Space]  
public float    m_forceToApply = 20.0f;
```

Weapon Attributes	
Name	<input type="text"/>
Damage	<input type="text" value="80"/>
Force To Apply	<input type="text" value="20"/>

# [ColorUsage]

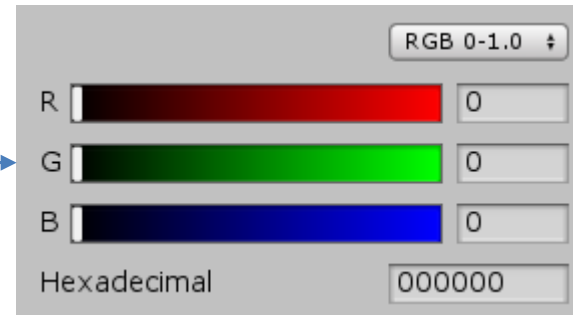
- Nos permite darle características al selector de color del inspector.
- Si hacemos la variable publica o privada con [SerializeField], vemos el selector de color en el inspector.

```
[SerializeField]  
private Color m_laserColor;
```



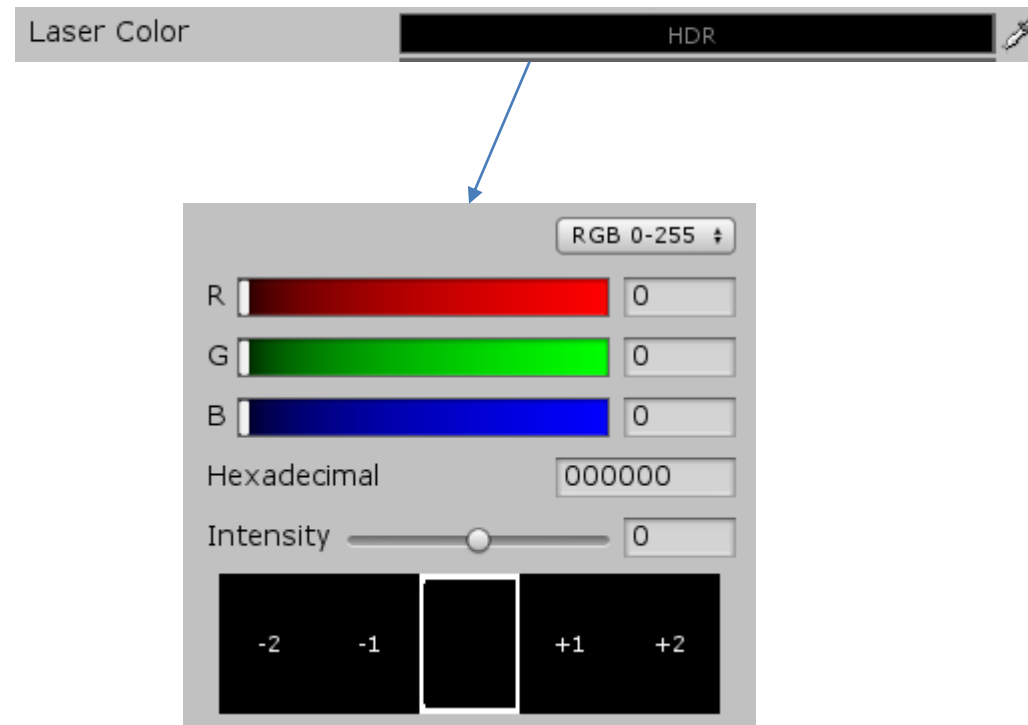
- Si añadimos [ColorUsage(false)] podemos hacer que el color no tenga selector de Alpha (el parámetro que hemos puesto a false se llama showAlpha).

```
[SerializeField]  
[ColorUsage(false)]  
private Color m_laserColor;
```



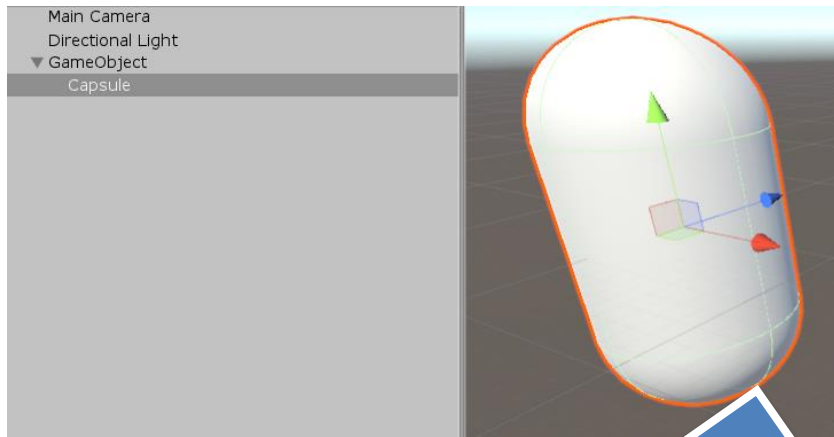
# [ColorUsage]

- ColorUsage tiene un segundo parámetro llamado “hdr” que nos indica si queremos que el color sea HDR.
- Si hacemos [ColorUsage(LoQueSeaElAlpha, true)]

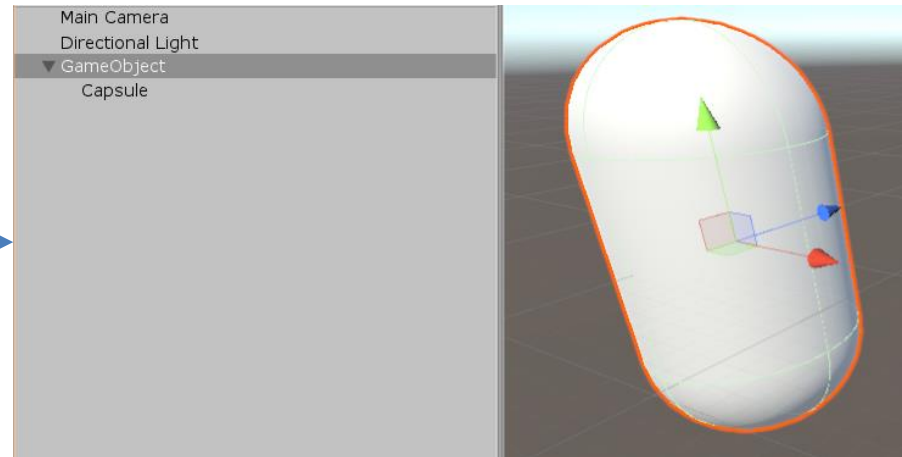


# [SelectionBase]

- Si añadimos esta directiva antes del nombre de la CLASE!! (no de una variable), hacemos que cuando se pincha en la escena en un hijo del objeto que tiene el componente que tiene esta directiva, se seleccione el padre.



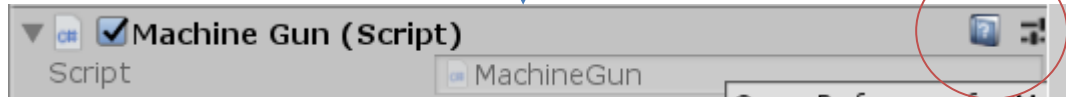
```
[SelectionBase]
public class Weapon : MonoBehaviour
{
```



# [HeaderHelpUrl]

- Cambia la dirección que se abre cuando pulsamos en la ayuda del script.

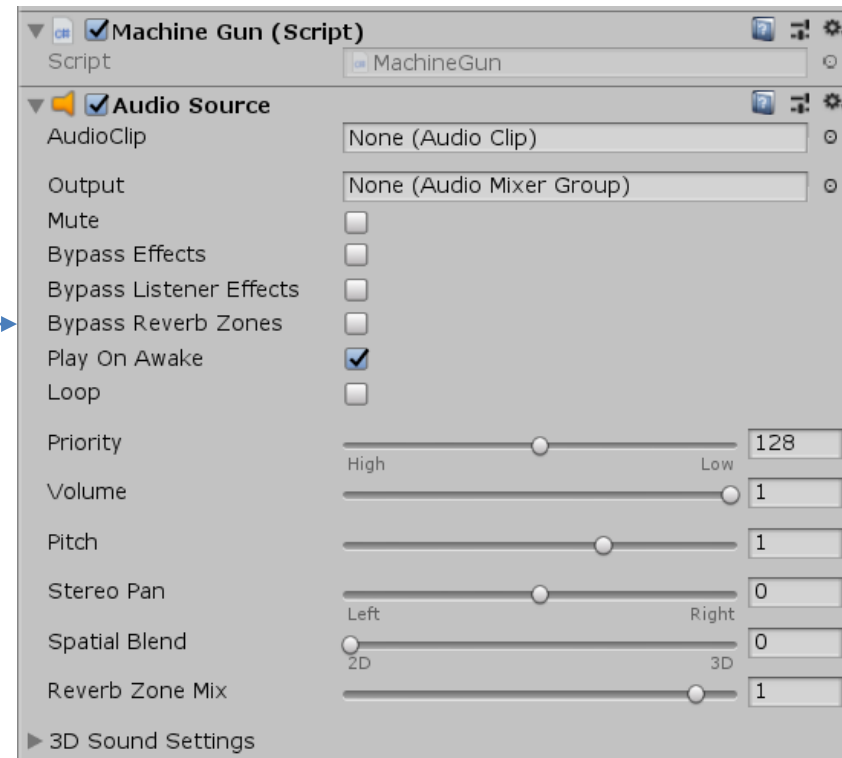
```
[HelpURL("http://example.com/docs/MyComponent.html")]  
public class MachineGun : MonoBehaviour {
```



# [RequireComponent]

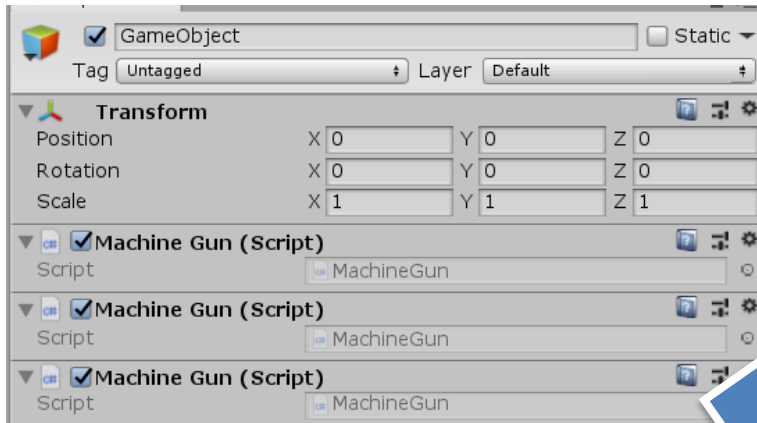
- Usando esta directiva conseguimos que se añadan automáticamente en el *gameobject* que tiene este componente la lista de scripts que definamos.

```
[RequireComponent (typeof (AudioSource))]  
public class MachineGun : MonoBehaviour {
```

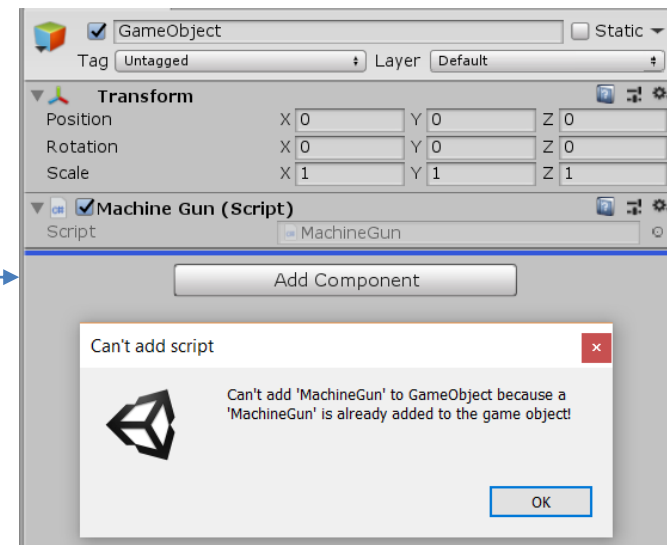


# [DisallowMultipleComponent]

- Impide que se pueda añadir un script varias veces al *gameobject*.



```
[DisallowMultipleComponent]  
public class MachineGun : MonoBehaviour {
```





# [ContextMenuItem]

- Me permite crear una opción al pulsar con el botón derecho sobre una variable.



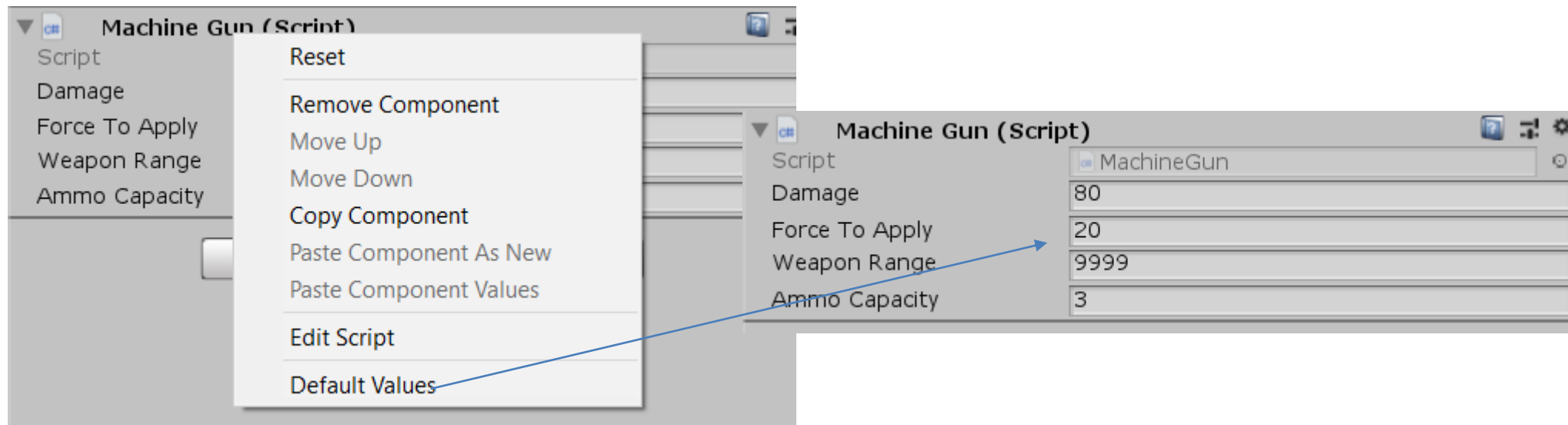
- Lo que hace la opción, lo tengo que definir en un método dentro del script.

```
[ContextMenuItem("Random Damage", "SetRandomDamage")]  
public float m_damage = 80.0f;  
  
private void SetRandomDamage()  
{  
    m_damage = Random.Range(0, 100);  
}
```

# [ContextMenu]

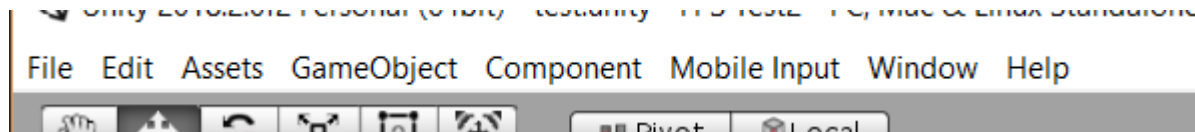
- Me permite crear una opción al pulsar con el botón derecho sobre el script. En este caso la hemos llamado "DefaultValues".

```
[ContextMenu("Default Values")]  
private void SetDefaultValues()  
{  
    m_damage      = 80;  
    m_forceToApply = 20.0f;  
    m_weaponRange  = 9999.0f;  
    m_ammoCapacity = 3;  
}
```



# [MenuItem]

- Nos permite añadir una opción donde queramos en los menus superiores de Unity.



```
[MenuItem("Assets/Create/Weapon List")]  
public static WeaponList Create()
```

# AssetDatabase

- Clase que permite tratar con assets.
- Por ejemplo podemos usarlo para crear nuevos assets con ciertas características.

```
[MenuItem("Assets/Create/Weapon List")]
public static WeaponList Create()
{
    WeaponList asset = ScriptableObject.CreateInstance<WeaponList>();
    AssetDatabase.CreateAsset(asset, "Assets/WeaponList.asset");
    AssetDatabase.SaveAssets();
    return asset;
}

[MenuItem("GameObject/Create Material")]
static void CreateMaterial()
{
    Material material = new Material(Shader.Find("Specular"));
    AssetDatabase.CreateAsset(material, "Assets/MyMaterial.mat");
}
```

# AssetDatabase

- Tiene muchas opciones posibles, se pueden buscar assets, crear carpetas, etc.

```
[MenuItem("GameObject/Print AssetsPath")]
static void SearchByname ()
{
    string[] results;

    results = AssetDatabase.FindAssets("_sprite");
    foreach (string guid in results)
    {
        Debug.Log("Path: " + AssetDatabase.GUIDToAssetPath(guid));
    }

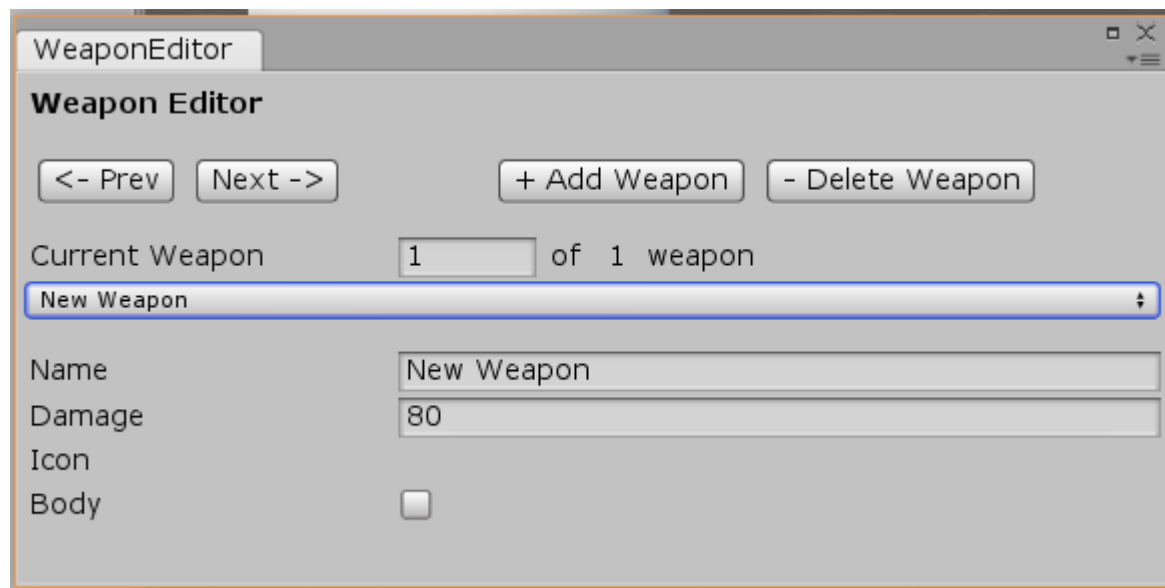
    results = AssetDatabase.FindAssets("t:Texture2D");
    foreach (string guid in results)
    {
        Debug.Log("path: " + AssetDatabase.GUIDToAssetPath(guid));
    }
}
```

```
inventoryItemList = AssetDatabase.LoadAssetAtPath (objectPath, typeof(SkillList)) as SkillList;
```

```
string relPath = AssetDatabase.GetAssetPath(inventoryItemList);
```

# Creando una herramienta para los ScriptableObject

- Vamos a crear una herramienta en una Pestaña nueva.
- La herramienta nos servirá para manejar comodamente la lista de scriptable objects que creamos en el tema anterior.



# Creando una herramienta para los ScriptableObject

- Para hacer la herramienta tenemos que crear un script que herede de EditorWindow.
- Tenemos que añadir un [MenuItem] y el método static que será llamado cuando pulsemos en la opción correspondiente.
- Dentro de este método, tenemos que usar EditorWindow.GetWindow para crear la Ventana.

```
public class WeaponEditor : EditorWindow
{
    [MenuItem ("Window/Weapon Editor")]
    static void Init ()
    {
        EditorWindow.GetWindow (typeof (WeaponEditor));
    }
}
```

# Creando una herramienta para los ScriptableObject

- En el método OnEnable comprobamos si ya está creado el archivo que contiene el scriptableObject y en caso de que no, lo creamos:

```
void OnEnable ()
{
    if(EditorPrefs.HasKey("ObjectPath"))
    {
        string objectPath = "Assets/Resources/Data/WeaponList.asset";
        inventoryItemList = AssetDatabase.LoadAssetAtPath (objectPath, typeof(WeaponList)) as WeaponList;
    }

    if (inventoryItemList == null)
    {
        viewIndex = 1;

        WeaponList asset = ScriptableObject.CreateInstance<WeaponList>();
        AssetDatabase.CreateAsset(asset, "Assets/WeaponList.asset");
        AssetDatabase.SaveAssets();

        inventoryItemList = asset;

        if (inventoryItemList)
        {
            inventoryItemList.weaponList = new List<WeaponItem>();
            string relPath = AssetDatabase.GetAssetPath(inventoryItemList);
            EditorPrefs.SetString("ObjectPath", relPath);
        }
    }
}
```



# Creando una herramienta para los ScriptableObject

- En el método OnGUI pintamos la Ventana y su contenido.

```
void OnGUI ()
{
    GUILayout.Label ("Weapon Editor", EditorStyles.boldLabel);
    GUILayout.Space(10);

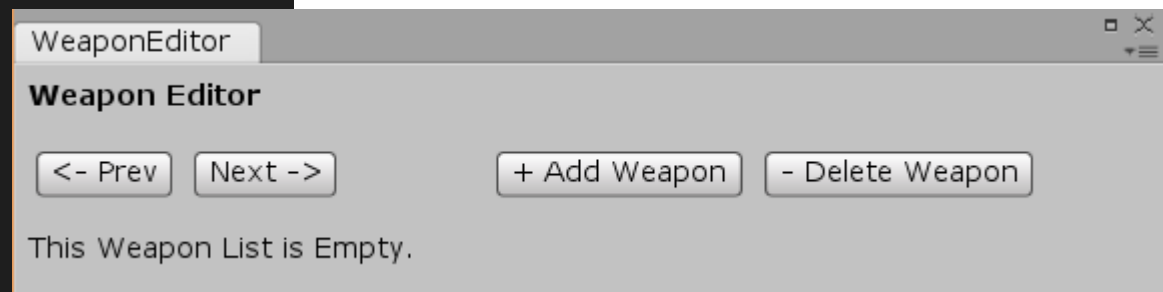
    if (inventoryItemList != null)
    {
        PrintTopMenu ();
    }
    else
    {
        GUILayout.Space(10);
        GUILayout.Label ("Can't load weapon list.");
    }

    if (GUI.changed)
    {
        EditorUtility.SetDirty(inventoryItemList);
    }
}
```

- En el PrintTopMenu que llamamos desde OnGUI pintamos los botones superiores y llamamos al metodo WeaponListMenu para pintar el elemento actual..

```
void PrintTopMenu ()
{
    GUILayout.BeginHorizontal ();
    GUILayout.Space(10);
    if (GUILayout.Button("<- Prev", GUILayout.ExpandWidth(false)))
    {
        if (viewIndex > 1)
            viewIndex --;
    }
    GUILayout.Space(5);
    if (GUILayout.Button("Next ->", GUILayout.ExpandWidth(false)))
    {
        if (viewIndex < inventoryItemList.weaponList.Count)
        {
            viewIndex ++;
        }
    }
    GUILayout.Space(60);
    if (GUILayout.Button("+ Add Weapon", GUILayout.ExpandWidth(false)))
    {
        AddItem();
    }
    GUILayout.Space(5);
    if (GUILayout.Button("- Delete Weapon", GUILayout.ExpandWidth(false)))
    {
        DeleteItem(viewIndex - 1);
    }
    GUILayout.EndHorizontal ();

    if (inventoryItemList.weaponList.Count > 0)
    {
        WeaponListMenu ();
    }
    else
    {
        GUILayout.Space(10);
        GUILayout.Label ("This Weapon List is Empty.");
    }
}
```



# Creando una herramienta para los ScriptableObject

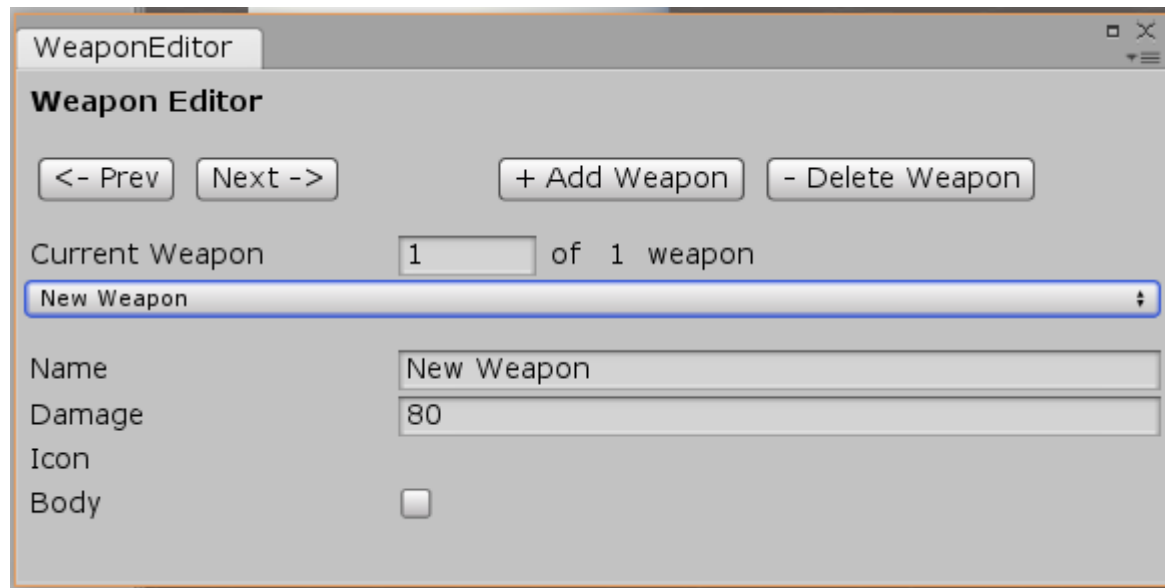
- Añadimos los métodos para crear y borrar elementos en la lista.

```
void AddItem ()
{
    WeaponItem newItem = new WeaponItem();
    newItem.m_name = "New Weapon";
    inventoryItemList.weaponList.Add (newItem);
    viewIndex = inventoryItemList.weaponList.Count;
}

void DeleteItem (int index)
{
    inventoryItemList.weaponList.RemoveAt (index);
}
```

# Creando una herramienta para los ScriptableObject

- Por último tenemos que añadir el poder seleccionar un elemento de la lista y editar sus características.



# Creando una herramienta para los ScriptableObject

```
void WeaponListMenu ()
{
    GUILayout.Space(10);
    GUILayout.BeginHorizontal ();
    viewIndex = Mathf.Clamp (EditorGUILayout.IntField ("Current Weapon", viewIndex, GUILayout.ExpandWidth(false)), 1, inventoryItemList.weaponList.Count);
    EditorGUILayout.LabelField ("of " + inventoryItemList.weaponList.Count.ToString() + " weapon", "", GUILayout.ExpandWidth(false));
    GUILayout.EndHorizontal ();
    string[] _choices = new string[inventoryItemList.weaponList.Count];
    for (int i = 0; i < inventoryItemList.weaponList.Count; i++)
    {
        _choices[i] = inventoryItemList.weaponList[i].m_name;
    }
    int _choiceIndex = viewIndex - 1;
    viewIndex = EditorGUILayout.Popup(_choiceIndex, _choices) + 1;

    //EditorUtility.SetDirty(someClass);
    GUILayout.Space(10);
    inventoryItemList.weaponList[viewIndex-1].m_name = EditorGUILayout.TextField ("Name", inventoryItemList.weaponList[viewIndex-1].m_name as string);
    inventoryItemList.weaponList[viewIndex-1].m_damage = EditorGUILayout.FloatField ("Damage", inventoryItemList.weaponList[viewIndex-1].m_damage);
    //inventoryItemList.weaponList[viewIndex-1].mode = (ModeSkillType)EditorGUILayout.EnumPopup ("Mode", inventoryItemList.skillList[viewIndex-1].mode);

    GUILayout.Label ("Icon");
    //inventoryItemList.skillList[viewIndex-1].icon = (Sprite)EditorGUILayout.ObjectField (inventoryItemList.skillList[viewIndex-1].icon, typeof(Sprite), false);
    inventoryItemList.weaponList[viewIndex-1].m_isAMachineGun = (bool)EditorGUILayout.Toggle ("Body", inventoryItemList.weaponList[viewIndex-1].m_isAMachineGun, GUILayout.ExpandWidth(false));
}
```

# AssetsPostProcessor

- Nos permite entrar en el proceso de importación de assets y cambiar las propiedades que tienen los assets al importarlos.
- Para poder hacer esto, tenemos que crear un script que herede de AssetPostProcessor y situarlo en Assets/Editor. Tenemos que añadir “using UnityEditor” para que no cause error.

```
using UnityEditor;  
using UnityEngine;  
  
public class CustomImporter : AssetPostprocessor
```

- Para que nos avise que se ha importado un elemento y poder tratarlo tenemos que añadir un método al script (parece raro, pero es lo mismo que cuando añadimos el método OnCollision si queremos que nos avise de que se ha producido una colisión).
- El método es OnPreprocess + NombreDelElemento para actuar antes de importar el elemento o OnPostprocess + NombreDelElemento para actuar una vez importado el elemento.



# AssetsPostProcessor

- Posibles valores de OnPreprocess:
  - OnPreprocessAssets
  - OnPreprocessAnimation
  - OnPreprocessAudio
  - OnPreprocessModel
  - OnPreprocessSpeedTree
  - OnPreprocessTexture

# AssetsPostProcessor

- En el preprocess es útil usar las siguientes variables:
- assetPath: Contiene la ruta del elemento que estoy importando. Viene bien para cambiar propiedades del elemento dependiendo de su nombre.
- assetImporter: Es el importador que se esta llamando (AudioImporter, TextureImporter, ModelImporter, etc).





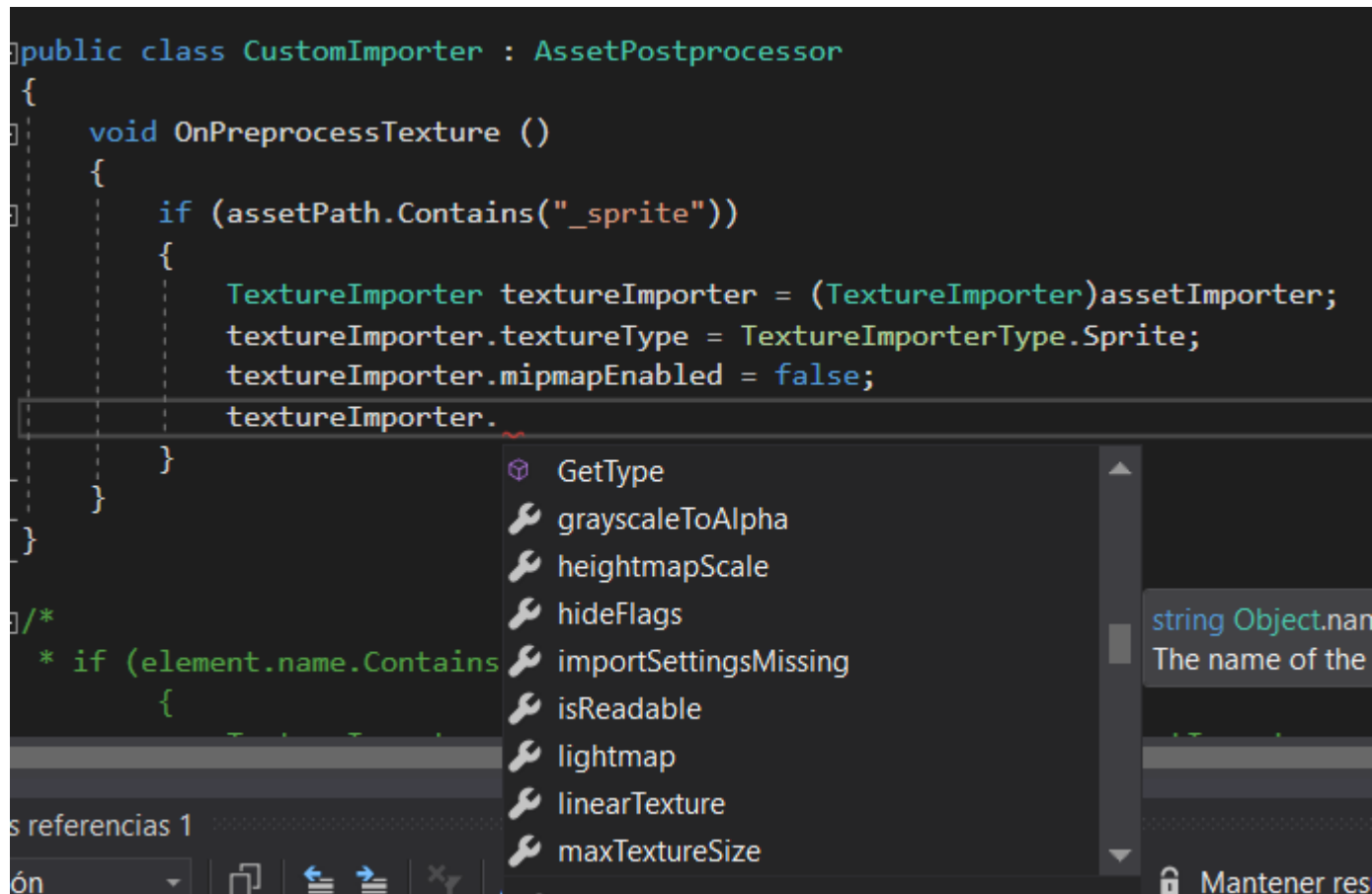
# AssetsPostProcessor

- Ej: Para cambiar las propiedades al importar una textura, tenemos que hacerlo en el preprocess. En este caso, cambiamos todas las propiedades de las texturas que se importen nuevas que tengan “\_sprite” en el nombre.

```
public class CustomImporter : AssetPostprocessor
{
    void OnPreprocessTexture ()
    {
        if (assetPath.Contains("_sprite"))
        {
            TextureImporter textureImporter = (TextureImporter)assetImporter;
            textureImporter.textureType = TextureImporterType.Sprite;
            textureImporter.mipmapEnabled = false;
            textureImporter.
        }
    }
}

/*
 * if (element.name.Contains
 {

```



# AssetsPostProcessor

- Posibles valores de OnPostProcess:
  - OnPostprocessAllAssets
  - OnPostprocessAudio (AudioClip)
  - OnPostprocessCubemap (CubeMap)
  - OnPostprocessGameObjectWithUserProperties
  - OnPostprocessMaterial (Material)
  - OnPostprocessModel (GameObject)
  - OnPostprocessSpeedTree (GameObject)
  - OnPostprocessSprites (Texture2D, Sprite[])
  - OnPostprocessTexture (Texture2D)



# AssetsPostProcessor

- OnPostprocessModel (GameObject)

```
void OnPostprocessModel (GameObject element)
{
    if (element.name.ToLower().Contains("weapon"))
        element.gameObject.AddComponent<WeaponBehaviour>();
}
```