UNITY TUTORIALS | UNITY | UNITY KNOWLEDGE

11 Useful Unity Attributes You Can Use To Customize The Inspector

By Pavee August 11, 2022

Learning how to customize the looks of Unity's inspector window is important. It makes adjusting various values of your game easier.

While the SerializeField attribute is commonly used to display variables in the inspector window, there are other attributes you can use such as the Range attribute which displays a slider for the variable in the inspector, or the Multiline attribute which makes the input field a multi-line text box.

Knowing how and when to use these attributes can be useful, both to you and the people you are working with. I'll list some of the useful ones with examples below so make sure you read through them!

Table of Contents

SerializeField

Perhaps one of the most common and most well-known unity attributes, the **SerializeField** attribute makes a private variable visible in the inspector and allows it to be edited. This is the base of other Unity attributes and you will be using it quite often.

To serialize a variable, simply declare **SerializeField** above or in front of the variable's declaration line:

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class TestScript : MonoBehaviour
6  {
7      // You can put the attribute above
8      // or in front of the declaration line
9      // Both work
10     [SerializeField]
11     private int stepCount = 95;
12
13     [SerializeField] private string testString = "Hi!";
```

```
}

```

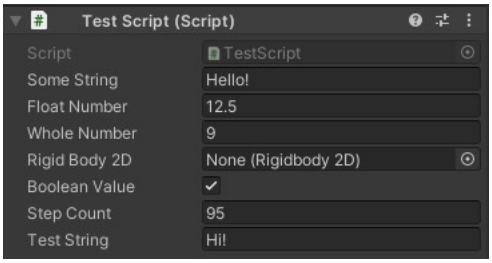
The field visible in the inspector will vary based on the type of the variable serialized.

So if it's a string, you get a text box with a string value in it. If it's a boolean, you get a checkbox. And so on.

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

public class TestScript : MonoBehaviour
{
    [SerializeField]
    private string someString = "Hello!";
    [SerializeField]
    private float floatNumber = 12.5f;
    [SerializeField]
    private int wholeNumber = 9;
    [SerializeField]
    private Rigidbody2D rigidBody2D;
    [SerializeField]
    private bool booleanValue = true;
}

```



Displayed fields

Some of you might have noticed that this attribute behaves similarly to when you set a variable as public.

"Then why don't we just set every variable we want to use in the inspector as public? Isn't that shorter to write?"

> It is generally not recommended to set a variable as public unless you absolutely have to as it can be accessed by other classes which can potentially cause issues. Using the SerializeField attribute to display private variables in the inspector is more secure.

Header

The **Header** attribute is used to create a bold header text. This is useful for separating each group of variables displayed in the inspector into sections to make them easier to read or find.

To add a header to the inspector, simply put the **Header** attribute above any serialized variable:

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

```
using UnityEngine;

public class TestScript : MonoBehaviour
{
    [Header("This is a header")]
    [SerializeField]
    private string someString = "Hello!";
    [SerializeField]
    private float floatNumber = 12.5f;

    [Header("This is another header")]
    [SerializeField]
    private int wholeNumber = 9;
    [SerializeField]
    private Rigidbody2D rigidBody2D;
    [SerializeField]
    private bool booleanValue = true;
}

```

This will appear in the inspector like this:

Fields with headers

Space

The **Space** attribute creates a small gap between 2 fields in the inspector. This is useful if you want to create groups of fields but don't want to use the **Header** attribute.

To use the **Space** attribute, put it between serialized variables, like so:

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

public class TestScript : MonoBehaviour
{
    [SerializeField]
    private string someString = "Hello!";
    [SerializeField]
    private int wholeNumber = 75;
    [SerializeField]
    private bool isAvailable = false;

    [Space]
    [SerializeField]
```

```
    private string characterName = "Jane Doe";
    [SerializeField]
    private string nickname = "Jane";
}
```

This will create a tiny gap between the **Is Available** field and the **Character Name** field.

There's a blank space

Range

The **Range** attribute gives a numeric variable a value slider in the inspector. This is useful when you want to limit the value range of a variable to a specific range.

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

public class TestScript : MonoBehaviour
{
    [Range(0f, 1f)]
    [SerializeField]
    private float floatNumber = 0.25f;

    [Range(0f, 10f)]
    [SerializeField]
    private int wholeNumber = 9;
}
```

Value sliders

Note that the **Range** attribute only works with numeric values. Using it with any other type of variable will result in a warning being displayed in the inspector.

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

public class TestScript : MonoBehaviour
{

    // This won't work
    [Range(0f, 1.5f)]
    [SerializeField]
```

```
    private string someString = "Hello!";
}
```

Strings can't use the **Range** attribute.

TextArea

The **TextArea** attribute turns a string input field into a text box that supports inputting multiple lines of text. This is commonly used when you want a long string.

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

public class TestScript : MonoBehaviour
{
    [SerializeField]
    private string nickname = "Jane";

    [TextArea]
    [SerializeField]
    private string characterDescription;
}
```

a text box

Multiline

The **Multiline** attribute is similar to the **TextArea** attribute in that it turns an input field into a text box. The difference is that the **Multiline** attribute's text box is displayed next to the variable name and the input field is not scrollable.

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

public class TestScript : MonoBehaviour
{
    [SerializeField]
    private string nickname = "Jane";

    [Multiline]
    [SerializeField]
    private string characterDescription;
}
```

Multiline attribute

Notice that the **Multiline** text box doesn't display scroll bars even when the text overflows.

Tooltip

The **Tooltip** attribute displays a floating tooltip text to a variable when you hover the mouse over the variable's name in the inspector.

This is usually used when you want to add a description to a variable. It is quite useful when you work with multiple people.

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

public class TestScript : MonoBehaviour
{
    [Tooltip("The name of the player-controlled character.")]
    [SerializeField]
    private string characterName = "Super Boy";
}
```

A tooltip

ContextMenu

The **ContextMenu** attribute adds a context menu to the script that, when selected, will execute a function.

This attribute is very useful when you want to quickly do something to the game object from the inspector, like resetting all of its properties back to the default value or setting the position of the object to a certain coordinate for quick debugging.

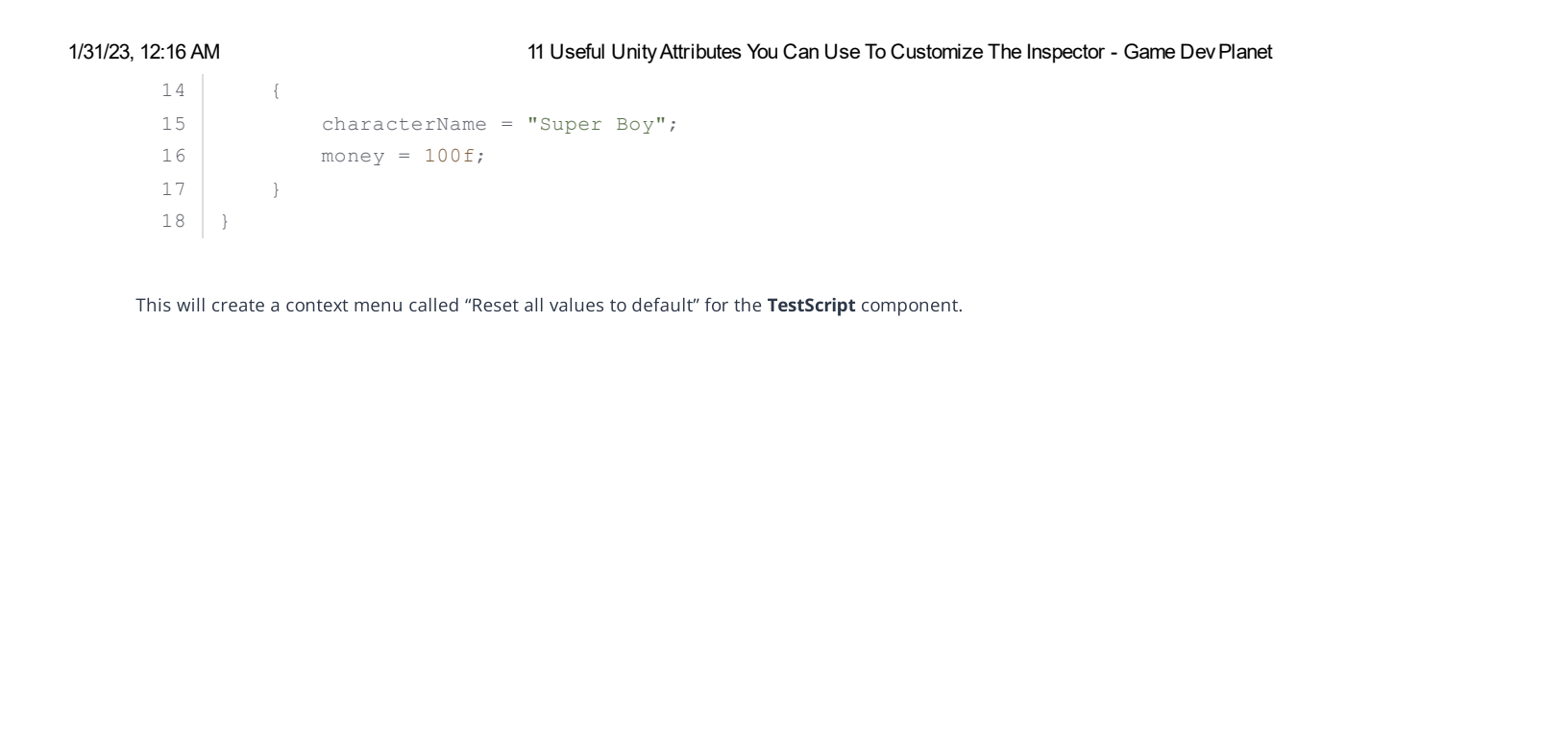
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class TestScript : MonoBehaviour
{
    [SerializeField]
    private string characterName = "Super Boy";
    [SerializeField]
    private float money = 100f;

    [ContextMenu("Reset all values to default")]
    private void ResetToDefault()
```

```
    {
        characterName = "Super Boy";
        money = 100f;
    }
}
```

This will create a context menu called “Reset all values to default” for the **TestScript** component.



A new context menu

When selected, the function **ResetToDefault** will be executed, setting the values of all variables to the default values.

Before:

After:

ContextMenuItem

The **ContextMenuItem** attribute is similar to the **ContextMenu** attribute but, instead of adding a context menu to the script component, it adds a context menu to a variable.

This is useful when you want to run a function that does something to a variable.

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

public class TestScript : MonoBehaviour
{
    [ContextMenuItem("Double the value", "ResetToDefault")]
    [SerializeField]
    private float money = 100f;

    private void ResetToDefault()
    {
        money *= 2f;
    }
}
```

Right-clicking on the variable name will bring up the context menu.

From the code example above, selecting “Double the value” will result in the current value of the money variable getting doubled.

RequireComponent

The **RequireComponent** attribute forces the game object that the script is attached to to add a component and prevent the component from being removed as long as the script is attached to it.

This attribute is used when a script requires certain components to be present in a game object it is attached to.

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

[RequireComponent(typeof(AudioSource))]
[RequireComponent(typeof(CapsuleCollider))]
public class TestScript : MonoBehaviour
{
    // ...
}
```

From the example above, the Audio Source component and the Capsule Collider component will be added to the game object when this script is attached to the object.

And you're not allowed to remove those components as long as the attribute or the script itself still exist within the game object.

Removing the component is not allowed

SelectionBase

The SelectionBase attribute marks the game object as the selection base in the scene view, meaning when a child object of the base object is selected in the scene view, the base object will be selected instead.

This is useful for when you have an object with multiple children that are just there for decoration or when you don't want to accidentally click and move child objects of an object in the scene view.

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

[SelectionBase]
public class TestScript : MonoBehaviour
{

}
```

These are the useful Unity attributes I use regularly. If you haven't tried them, I recommend you start using them. They will make your life a lot easier.

And If you're just starting out and don't know what kind of game you want to make, I've got a nifty list for you.

Cheers!

Attribution

Programming icon made by Icongeek26 from www.flaticon.com

#Game Development #Unity #Unity Editor

Pavee

Hi, I'm Pavee. I'm a software developer, an aspiring pixel artist, and the owner of Game Dev Planet. I love learning new things and have a passion for game development.

← PREVIOUS
Which Version of Unity Should I use? LTS vs Latest Version

NEXT →
Ways To Log And Customize Debug Messages For Unity Console

Similar Posts

Pros and Cons of Early Access: What Game Developers Should Know

By Pavee September 15, 2022

Adding URP to an Existing Unity Project Without Breaking It

By Pavee March 6, 2022

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment *

Name *

Email *

Save my name, email, and website in this browser for the next time I comment.

Post Comment

This site uses Akismet to reduce spam. Learn how your comment data is processed.

One Comment

Lukas Thompson says:
April 23, 2022 at 01:31

Thank you so much for this, instant bookmark.

This is going to help me make my scripts way nicer and more easily accessible to non-programmers.

Reply

LATEST POSTS

6 Easy To Learn Game Engines for Beginners and Non-Programmers

TCP vs UDP: Why UDP is Preferred Over TCP For Online Multiplayer Games

Pros and Cons of Early Access: What Game Developers Should Know

Why Are Pixel Art games So Popular Even Now?

The Fascinating History and Evolution of Platformers

AFFILIATE DISCLOSURE

When possible, Game Dev Planet uses affiliate links (at no additional cost to you). Game Dev Planet also participates in affiliate programs with Cloudways, Unity Asset Store, and other sites. Game Dev Planet is compensated for referring traffic and business to these companies.

CATEGORIES

- General Game Development
- Unity
- Unity Knowledge
- Unity Tutorials

Animation Art Style Camera Cinemachine Effector 2D Game Assets Game Development Game Jam Godot History Knowledge Marketing Mobile Game Developement Multiplayer Networking Optimization Pixel Art Platformer Quixel Bridge Quixel Megascans RPG Maker Time Unity Unity 2D Unity Assets Unity Editor Unity Tutorial Universal Render Pipeline Unreal Engine Update Loop