Custom Attributes

Documentation

Creator: Lucas Gomes Cecchini

Pseudonym: AGAMENOM

Overview

This document will help you use **Custom Attributes** for **Unity**.

With it you can customize your **Inspector**.

Just like Unity's **Attributes**, it adds functions that make it easier to use.

Such as easier selection of **Tag** or **Scene name**, **viewing properties**, etc.

Instructions

You can get more information from the Playlist on YouTube:

https://www.youtube.com/playlist?list=PL5hnfx09yM4I_6OdJvShZ0rRtYF9jv6Cd

Script Explanations

Tag Dropdown Attribute

General Overview

This script defines a **custom attribute** and a **custom property drawer** for the Unity Editor.
Its purpose is to allow a **string variable** in a Unity script to display a **dropdown menu of all available Unity tags** in the Inspector.
This is useful when you want to assign or compare Unity tags (for example, when detecting collisions or triggers by tag name) but want to avoid **typing tag names manually**, which helps prevent typos and missing-tag errors.

1. TagDropdownAttribute

Purpose
This is a simple **attribute marker** that you can place above a string variable to tell the Unity Editor that this field should display as a **dropdown list of tags** instead of a plain text box.
Implementation Details

- It inherits from Unity's base **PropertyAttribute** class.
- It has **no internal variables or methods**, because its only purpose is to **flag** fields that should use the custom drawer.

Usage Example
You would use it like this in another script:
[TagDropdown]
public string tagName;
When viewed in the Inspector, tagName will appear as a dropdown containing all the project's tags.

2. TagDropdownDrawer

Purpose
This class defines how the Unity Editor **renders** any string field that has the [TagDropdown] attribute.
It replaces the normal text field with a dropdown containing all available Unity tags, plus a warning message if the current value doesn't match any known tag.
Internal Behavior
This class is only compiled when the Unity Editor is active, thanks to the preprocessor check (#if UNITY_EDITOR).
This prevents the editor-only code from being included in the game build.

Method 1: OnGUI

Purpose
This method **draws the field** in the Inspector. It determines how the dropdown looks and how it behaves when a user interacts with it.
Parameters

- **position**: The rectangular area in the Inspector where this field will be drawn.
- **property**: The serialized version of the variable being edited.
- **label**: The label that appears next to the field in the Inspector.

Step-by-Step Explanation

1. **Type validation**
   It first checks if the variable is actually a string.
   If not, it simply displays a label saying that [TagDropdown] can only be used with string variables.
2. **Retrieve available tags**
   It asks Unity for the list of all tags defined in the project using an internal Unity Editor utility.
3. **Handle missing tag situations**
   - It stores the current string value of the property (the tag currently assigned).
   - If this current value doesn't exist in Unity's tag list, it creates a "Tag Missing" option and places it at the top of the dropdown list.
   - It then builds a complete list of tags, adding the "Tag Missing" entry if needed.
4. **Determine the current selection index**
   It finds which tag in the list corresponds to the current string value.
   If none match, it defaults to index 0 (the "Tag Missing" entry).

5. **Display the dropdown**
   It draws a dropdown in the Inspector, showing all the tags in the list.
6. **Update the string value**
   If the user selects a valid tag (not the “Tag Missing” option), it updates the variable’s string value to the selected tag.
7. **Display warning box**
   If the selection corresponds to “Tag Missing”, it draws a small warning box below the dropdown, informing the user that the current string does not match any existing tag.

📏 Method 2: GetPropertyHeight

Purpose

This method determines **how much vertical space** the field should take up in the Inspector.
Normally, a field takes one line of height, but if there is a warning message, it needs extra space.

Step-by-Step Explanation

1. **Retrieve all tags again**
   It gets the current list of tags from Unity.
2. **Check if the property’s value matches any tag**
   If the stored string value does not exist in the tag list, it increases the height to make room for the warning box.
3. **Return the correct height**
   - If there’s no issue, it returns the standard single-line height.
   - If there’s a missing tag warning, it returns a height roughly three times larger, to fit the dropdown plus the warning box.

⚙️ Variables Inside the Drawer

- **tags** → A list of all Unity tags retrieved from the internal editor utility.
- **currentString** → The current string value stored in the property (the tag currently assigned).
- **currentText** → The formatted text shown when the tag is missing, e.g., “Tag Missing (MyOldTag)”.
- **missingTagText** → The actual “Tag Missing” entry added to the list only if the tag does not exist.
- **tagList** → A list combining the “Tag Missing” entry (if needed) and all valid Unity tags.
- **currentIndex** → The position of the currently assigned tag in the dropdown list.
- **newIndex** → The position selected by the user after interacting with the dropdown.
- **helpBoxRect** → The rectangular space defined to draw the warning box below the dropdown.

🧪 Usage Summary

1. Attach the [TagDropdown] attribute above a string field in any MonoBehaviour or ScriptableObject.
2. When you select the GameObject in the Unity Editor, the field will appear as a dropdown containing all Unity tags.
3. If the string value doesn’t match any tag in the project, a “Tag Missing” entry will appear at the top, and a warning box will show below the field.
4. Selecting a valid tag automatically updates the string value in the serialized property.

✅ Benefits

- Prevents typing mistakes in tag names.
- Ensures that string fields always contain a valid Unity tag.
- Gives visual feedback when the tag is invalid or missing.
- Cleanly integrates with Unity’s built-in Inspector without needing custom windows or editors.

Scene Tag Dropdown Attribute

🧩 General Overview

This script extends the Unity Editor to display a **dropdown list of scenes** (from the Build Settings) for any string field marked with a special attribute.
Instead of typing a scene name manually (which can cause errors if the name changes or the scene is disabled), the user can simply choose it from a dropdown list in the Inspector.
The drawer also shows **warnings or errors** when the chosen scene is missing or disabled, and it automatically adjusts the height of the Inspector field to make space for those warnings.

🏷️ **1. SceneTagDropdownAttribute**
Purpose
This is a **marker attribute** that you can attach above a string variable.
It signals to Unity that the variable should show a **dropdown menu of scene names** in the Inspector.
It doesn't contain any logic or data — it simply marks the field so that the custom drawer knows to modify how it's displayed.
Example Usage
In any script, you can declare a string variable and place the attribute above it.
When that script is shown in the Inspector, Unity will render it as a dropdown listing all available scenes in the Build Settings.

🧭 **2. SceneTagDropdownDrawer**
This is the **custom editor drawer** responsible for rendering the dropdown and handling all the logic when the attribute is applied.
It only exists inside the Unity Editor (not in the final game build), because it's wrapped in an editor-only conditional directive.

🖼️ **Method: OnGUI**
Purpose
This method controls **how the field appears** and **what happens when the user interacts** with it in the Inspector.
It replaces the normal text field with a dropdown showing the list of build scenes and provides visual feedback for missing or disabled entries.

Step-by-Step Breakdown
1. **Validate the field type**
   The method first checks that the variable is actually a string.
   If it's not, it simply displays a label telling the user to use this attribute only on strings.
2. **Collect all scenes from Build Settings**
   It retrieves a list of all the scenes added in the *Build Settings*.
   Each scene's name is extracted from its file path (the part before the ".unity" extension).
3. **Identify enabled scenes**
   From that list, it filters out only the scenes that are currently enabled in the Build Settings.
   This distinction is important so it can later warn if the chosen scene is disabled.
4. **Prepare missing or disabled scene labels**
   - It reads the current value stored in the string variable.
   - If the variable is empty, it uses the label "Missing Scene".
   - If the current scene name is found but is disabled, it adds a "[Disabled]" suffix to the name.
   - If the name doesn't match any scene at all, it keeps a "Missing Scene (name)" entry at the top.

This creates a clear context for the user, showing when the variable doesn't correspond to a valid or active scene.

5. **Build the dropdown list**
   A list is built starting with a possible "Missing Scene" entry (if applicable), followed by all the enabled scene names.
   This list becomes the dropdown menu shown in the Inspector.
6. **Determine the selected index**
   It finds where the current string value appears in that list.
   If it doesn't exist, it defaults to the "Missing Scene" entry.
7. **Draw the dropdown**
   The method then renders the dropdown at the correct position with the label provided.
   The user can select any valid scene name from the list.
8. **Update the property value**
   If the user selects a valid option (not "Missing Scene"), the underlying string variable is updated to match the selected scene name.
9. **Show warnings or errors**
   - If the scene is present in Build Settings but is disabled, a **warning box** appears below the dropdown saying "The scene is in the build but is disabled."

        - If the scene does not exist at all in Build Settings, an **error box** appears saying that the value doesn't match any scene.
10. **Adjust layout for messages**
    When a warning or error appears, an extra rectangular area is defined below the dropdown to display the help message.
    This extra space ensures that the UI layout doesn't overlap or clip.

Key Variables in OnGUI
- **allScenes** → Contains every scene name from the Build Settings.
- **enabledScenes** → A filtered list of only the scenes that are enabled in the Build Settings.
- **currentString** → The actual string value stored in the property (the name currently assigned).
- **currentText** → The fallback text shown when the current scene value is missing.
- **missingScene** → Represents the label used when the current scene exists but is disabled.
- **missingText** → Represents the label used when the current scene doesn't exist or is empty.
- **sceneList** → The full list of dropdown options, including the missing entry and enabled scenes.
- **currentIndex** → The position of the current value within the list.
- **newIndex** → The index chosen by the user from the dropdown.
- **helpBoxRect** → The rectangle that defines where to draw the warning or error message.

📏 Method: GetPropertyHeight
Purpose
This method defines **how tall** the field should appear in the Inspector.
It adjusts the vertical space depending on whether the property requires extra room for warnings or errors.

Step-by-Step Breakdown
1. **Get the list of all scenes again**
   It retrieves the same list of scene names from the Build Settings.
2. **Check if the current value is valid**
    - If the stored string value is empty or doesn't exist among the scenes, extra height is added to make space for the error message.
    - If the scene exists but is disabled, extra height is also added for the warning.
    - Otherwise, only the default single-line height is returned.

This ensures that warning or error messages have space below the dropdown without overlapping other UI elements.

🧪 Usage Summary
1. In any script, declare a string variable and tag it with [SceneTagDropdown].
2. Open the Unity Editor and inspect the object containing that script.
3. The string field will now appear as a dropdown menu listing all the **enabled scenes** in Build Settings.
4. If the stored scene name is missing or disabled, a message will appear below the dropdown:
    - **Warning** → Scene exists but is disabled.
    - **Error** → Scene not found or empty.
5. Selecting a valid scene updates the variable automatically.

✅ Advantages
- Prevents typos or invalid scene name references.
- Helps visually detect missing or disabled scenes.
- Dynamically adapts to changes in Build Settings.
- Integrates cleanly with Unity's native Inspector system.

Read Only Attribute

🧩 General Overview
This script adds a feature to Unity's Inspector that allows certain fields to be **visible but not editable**.
When you apply the special marker [ReadOnly] to a variable in a Unity component, that variable will still appear in the Inspector, but the user will not be able to change its value.

This is very useful when you want to **display runtime information, debug data, or internal state variables** without risking accidental edits that could break functionality.

🏷️ 1. ReadOnlyAttribute

Purpose

This is a **custom attribute** used to tell Unity that a field should be **displayed but locked** in the Inspector.
It doesn't contain any code logic or data.
It acts purely as a **marker**, meaning that its presence tells another script (the custom drawer) how to handle the field's display behavior.

How It's Used

You apply it above a field in any MonoBehaviour or ScriptableObject.
For example, if you want to show a variable's value during play mode but not allow editing, you mark it with [ReadOnly].
Then, when Unity draws this field in the Inspector, it will appear greyed out and uneditable, but still visible.

🧭 2. ReadOnlyDrawer

This is a **custom drawer** that defines what happens in the Inspector when Unity sees a field marked with the ReadOnly attribute.
It exists only inside the Unity Editor, meaning it's not included in the final game build.
The drawer takes control of how the field is drawn on the screen.

🖼️ Method: OnGUI

Purpose

This method controls **how the property is displayed** in the Inspector.
When Unity is about to draw a field that has the [ReadOnly] attribute, it calls this method instead of its default drawing behavior.

Step-by-Step Explanation

1. **Disable editing in the graphical user interface**
   - Before drawing the field, the script tells Unity's interface system that the next controls should be **disabled**.
   - This means the user can see the field's value but cannot type or change anything inside it.
   - The field becomes visually greyed out to indicate that it is locked.
2. **Draw the field normally**
   - The drawer then instructs Unity to render the field as usual, with its label and current value.
   - Because editing is disabled, the user can only observe the data.
3. **Restore the GUI state**
   - After drawing, it immediately re-enables the user interface.
   - This is important because the Inspector might draw many other fields afterward, and we only want the [ReadOnly] field to be locked — not everything else.

Parameters Explained

- **position** → The rectangular area in the Inspector window where the field will appear. It defines its size and placement.
- **property** → The serialized representation of the variable being shown. It contains the value and metadata that Unity uses to draw it.
- **label** → The text label that appears next to the field in the Inspector.

⚙️ Internal Logic Summary

When Unity processes a field with [ReadOnly]:

1. It disables editing.
2. Draws the field in a non-interactive state.
3. Re-enables the interface for subsequent elements.

This simple sequence ensures that only the chosen variable is locked.

🧪 Usage Example (Conceptual)

You can apply the [ReadOnly] marker above any public or serialized field.
In the Inspector, the variable will appear greyed out and cannot be modified.
For instance, developers often use this to:

- Show the current health or position of a character during runtime.
- Display calculated values that are updated by scripts but shouldn't be changed manually.
- Protect internal data that should only be modified through code logic.

Variables Involved
There are **no internal variables** stored in this drawer because it doesn't need to remember or manipulate data — it only changes the *display state* of the Unity Editor temporarily while drawing.
However, three parameters are used during drawing:

- **position** (visual layout)
- **property** (the serialized data)
- **label** (the text caption next to the field)

Additionally, Unity's built-in **GUI.enabled** flag is toggled to disable and re-enable interactivity.

Benefits

- Prevents accidental changes to critical or runtime-only data.
- Makes debugging easier by displaying values directly in the Inspector.
- Keeps code safe and consistent by separating "viewable" and "editable" data.
- Works seamlessly with all serialized field types (numbers, strings, objects, etc.).

Summary of the Script's Flow

1. The [ReadOnly] attribute marks specific fields.
2. The Unity Editor detects that attribute and invokes the custom drawer.
3. The drawer temporarily disables editing for that field.
4. It renders the field's value in a non-editable (greyed-out) state.
5. The drawer then re-enables normal editing for all other fields.

Highlight Empty Reference Attribute

General Overview
This script extends Unity's Inspector so that when a variable marked with a special tag ([HighlightEmptyReference]) has **no assigned object**, it gets visually highlighted with a **red background** and a **warning message**.
It helps developers quickly identify **unassigned references** in the Editor, preventing common runtime errors such as missing components, prefabs, or script references.

1. HighlightEmptyReferenceAttribute
Purpose
This is a **marker attribute** that you attach above a field in a Unity script.
Its role is to tell the Unity Editor that this variable should be visually checked for null or missing references.
This attribute doesn't contain any internal logic or data — it only serves as a tag that the Unity Editor recognizes through the matching drawer.
Example Usage
When placed above an object reference (for example, a reference to a GameObject, Transform, or ScriptableObject), Unity will automatically highlight it in the Inspector if nothing is assigned to it.
Example use case:
You might mark fields that must always be assigned, such as "player reference", "enemy prefab", or "UI manager".
If you forget to assign something, the field turns red and a clear message appears underneath, reminding you to fill it in.

2. HighlightEmptyReferenceDrawer
This is the **custom property drawer** that defines how Unity draws fields marked with the [HighlightEmptyReference] attribute.
It only runs inside the Unity Editor, never in the game build.

This drawer modifies how the field looks and adds logic to detect and warn about unassigned object references.

🖼 Method: OnGUI

Purpose

This method controls **how the field appears and behaves** in the Unity Inspector.
It is called every time Unity renders the Inspector window for an object that uses the [HighlightEmptyReference] attribute.

Step-by-Step Breakdown

1. **Store the original background color**
   Before changing any visual settings, the script saves the current background color of the Inspector interface.
   This ensures that the interface can be restored to normal afterward.

Variable involved:

- **previousColor** → Holds the Inspector's color before modification.

2. **Check if the field is an object reference and whether it's empty**
   The drawer verifies that the variable type is an object reference (meaning it can hold things like components, prefabs, or other assets).
   Then it checks if the current reference value is null or missing.

Variable involved:

- **isEmpty** → Becomes true if the field is an object reference and nothing has been assigned.

3. **Change the background color if the field is empty**
   If the reference is missing, the field's background color changes to red, visually alerting the user. Otherwise, it keeps the default color.
4. **Define the position for the field**
   A rectangular area (called a "rect") is created to tell Unity where on the screen to draw the field. It specifies horizontal position, vertical position, width, and height.

Variable involved:

- **propertyRect** → Defines the space where the reference field is drawn.

5. **Draw the property field**
   The field is drawn as usual, but if it's empty, the background appears red.
6. **Restore the previous background color**
   After drawing the field, the background color is set back to what it was before.
   This prevents the red color from affecting any other elements in the Inspector.
7. **If the field is empty, display an error message**
   When the reference is missing, a help box appears below the field with a message that says what type of object should be assigned.

To do this:

- The script tries to find the **type of the field** using reflection (a system that allows the script to look at the field's metadata).
- If the field type can't be found, it defaults to showing "Unknown".

Variables involved:

- **typeName** → Stores the type of object that should be assigned (for example, "Transform", "AudioSource", or "ScriptableObject").
- **helpBoxRect** → Defines the position and size of the warning box below the field.

8. **Draw the warning box**
   The warning box shows a red message such as:
   "Put an item of type 'Transform' here!"
   This gives the user a clear indication of what is missing.

Parameters Explained

- **position** → Defines where in the Inspector this element is drawn (its location and size).
- **property** → Represents the serialized version of the variable being edited. It holds both the data and metadata Unity uses to display and modify fields.
- **label** → The field's display name or caption in the Inspector.

📏 Method: GetPropertyHeight

Purpose

This method determines **how tall the field's space** should be in the Inspector.
If the field is missing a reference, extra vertical space is added to make room for the help box below it.

Step-by-Step Breakdown

1. **Check if the field is empty**
   It performs the same logic as before — checking if the variable is an object reference and if it has no value assigned.
2. **Return the correct height**
   - If the reference is missing, the method adds extra height (approximately triple the normal field height) so the help box can fit below the field.
   - If the field is correctly assigned, it returns the default single-line height.

This ensures that the warning message doesn't overlap with other elements in the Inspector.

Variables Involved in Height Calculation

- **isEmpty** → Boolean that indicates whether the field has no reference assigned.
- The method then returns either a larger or smaller height value depending on that condition.

🧪 **Usage Summary**

1. Add the [HighlightEmptyReference] attribute above any object reference field (e.g., a GameObject, Transform, ScriptableObject, or Component reference).
2. In the Unity Editor, open the object containing this script.
3. If the field is **not assigned**, it will be highlighted with a **red background** and a **warning box** telling you what type of object should go there.
4. Once you assign a valid reference, the highlight and warning automatically disappear.

✅ **Benefits**

- Makes it **visually obvious** when an important reference is missing.
- Prevents runtime errors caused by unassigned objects.
- Improves debugging and scene setup speed.
- Provides clear type information about what is expected in the field.
- Automatically adapts its height to keep the Inspector layout clean and readable.

🧭 **Summary of Logic Flow**

1. The attribute [HighlightEmptyReference] marks a field for special handling.
2. When Unity draws the Inspector, it detects this attribute and uses the custom drawer.
3. The drawer checks if the field is an object reference and whether it's empty.
4. If empty:
   - The field background becomes red.
   - A help message appears below it with the expected object type.
5. If filled:
   - The field appears normal.
   - No warning is shown.
6. The drawer automatically adjusts the visual height to fit any messages.

Conditional Hide Attribute

🧩 **General Overview**

This script allows you to **hide or show fields in the Unity Inspector dynamically**, depending on the state of other variables in the same object.
It defines a **custom rule tag** called ConditionalHide, which you can attach to variables. When Unity draws the Inspector window, it checks whether the specified conditions are met. If the conditions are not satisfied, the field disappears from view.
This system is especially helpful when building complex Inspector interfaces — for example, showing advanced settings only when a toggle is enabled or hiding optional parameters when not needed.

🏷️ **ConditionalHideAttribute (the rule definition)**

This part defines the behavior rules. It's the **tag** you attach to your fields to tell Unity *when* they should appear or disappear.

🔷 **Purpose**
It stores the **names of other fields** that determine whether a specific property should be visible.
It also defines **how** those conditions should be combined — whether *any* false condition hides the property, or whether it hides only if *all* conditions fail.

🔷 **Variables inside**

- **ConditionalSourceFields**
  This holds one or more text values that represent the names of the fields Unity will check.
  Each of these source fields must be a true/false type in the Inspector.

Example: if you write [ConditionalHide("isActive")], it means “show this field only if isActive is true”.

- **HideIfAnyFalse**
  This determines the logic used when multiple conditions are listed.
  If it is true, the target field hides whenever **any** of the conditions are false.
  If it is false, the field hides only when **all** conditions are false.

🔷 **Constructors (setup methods)**
The script provides two ways to set up the rule when you apply it:

1. **Simplified usage** –
   You can pass one or more condition names directly.
   If you list multiple names, the rule automatically hides the field if any of them are false.

Example:
[ConditionalHide("isVisible", "canEdit")]
→ The field disappears if either condition is false.

2. **Detailed usage** –
   You can explicitly define whether the property should hide when *any* or *all* conditions fail.
   The first parameter is a true/false value that controls this behavior.

Example:
[ConditionalHide(false, "option1", "option2")]
→ The field will hide only when both conditions are false.

🧭 **ConditionalHidePropertyDrawer (the Inspector behavior)**
This section tells Unity **how to draw** the fields that use the ConditionalHide tag.
It runs only inside the Unity Editor and controls both **visibility** and **space allocation** in the Inspector.

🖼️ **Method: OnGUI (controls the visual display)**
This part decides **whether to show or skip** the field during Inspector rendering.

Step-by-step logic:

1. It first retrieves the rule information from the field (the conditions and how they should be evaluated).
2. It checks whether the rule should hide the field.
   - If there is **one** condition, it checks whether that condition is false.
   - If there are **multiple** conditions, it checks all of them using the rule’s HideIfAnyFalse setting.
3. If the result of that check means the field should be hidden, it stops there and skips drawing that field.
4. Otherwise, the field is displayed normally in the Inspector.

Parameters explained:

- **position** → Defines where on the screen Unity should draw the element.
- **property** → Represents the specific variable that Unity is currently drawing.
- **label** → The field’s visible name or caption inside the Inspector.

📏 **Method: GetPropertyHeight (controls layout space)**
This part ensures that when a field is hidden, it also **doesn’t occupy any vertical space** in the Inspector.
Unity’s Inspector layout engine asks how tall each field should be.

- If the field is hidden, this method returns **zero**, so nothing is drawn and no gap appears.
- If the field is visible, it returns the normal height.

This keeps the Inspector layout clean and compact.

⚙️ **Hidden logic: Condition checking**
Several internal checks determine whether the field should appear.
They work together to interpret the rules stored in the attribute.

1. CheckSingleCondition
- Looks up one specific field (the condition) by its name.
- If that field cannot be found or is not a true/false value, it assumes the condition is false.
- If the field is found and its value is false, the target field will be hidden.

2. CheckAllConditions
- Used when there are multiple condition names and the rule says “hide if any condition fails”.
- It loops through all listed condition fields.
- If any of them are missing or false, it decides that the property should be hidden.
- Only when *all* conditions are true does the property remain visible.

3. CheckAnyCondition
- Used when there are multiple condition names but the rule says “hide only if all are false”.
- It loops through all condition fields, looking for at least one that is true.
- If it finds one true condition, the field stays visible.
- If all conditions are false, the field hides.

🔍 **Method: GetConditionProperty (finding the target fields)**
This helper part is responsible for **locating the field** that controls visibility.
It takes the name provided in the ConditionalHide tag and tries to find the corresponding variable inside the same object.
If it doesn’t find it immediately, it supports **nested searches** — for example, if the condition is written like myClass.myToggle, it will look for the inner property within another object.
It returns the matching condition data or nothing if the path doesn’t exist.

🧪 **How to Use It**
You attach the ConditionalHide tag above a variable in your script, referencing one or more true/false fields in the same component.
Example Scenarios:
1. **Single toggle control**
2. [ConditionalHide("showAdvanced")]
3. public int advancedSetting;

→ The “advancedSetting” field appears only when “showAdvanced” is true.

4. **Multiple controls (any false hides)**
5. [ConditionalHide("isVisible", "isActive")]
6. public GameObject target;

→ The “target” field appears only if both “isVisible” and “isActive” are true.

7. **Multiple controls (hide only if all are false)**
8. [ConditionalHide(false, "optionA", "optionB")]
9. public GameObject optionalPanel;

→ The “optionalPanel” is visible if *at least one* of the options is true.

10. **Nested reference**
11. [ConditionalHide("settings.enableSound")]
12. public AudioClip backgroundMusic;

→ The field will only be visible if enableSound inside settings is true.

✅ **Summary of How It Works**
1. The [ConditionalHide] tag is attached to a variable.
2. The tag stores a list of condition names and how to interpret them.
3. When Unity draws the Inspector:
   - It checks those condition fields.
   - Based on their values, it decides whether the field should appear.
4. If hidden, the field is skipped visually and occupies no space.
5. If visible, it’s drawn normally as if no condition existed.

🎯 **Benefits**

- Keeps the Unity Inspector clean and context-sensitive.
- Prevents users from changing irrelevant settings.
- Makes editor interfaces more intuitive and professional.
- Supports multiple conditions and nested properties.
- Works dynamically — updates instantly as conditions change in the Inspector.

Button Attribute

🧩 **General Overview**

This system adds **interactive buttons inside the Unity Inspector** that directly call specific actions (methods) from a component.
Normally, Unity's Inspector only shows fields (numbers, text, toggles, etc.), but not buttons tied to your script's logic.
With this setup, any method marked with a special **custom tag** becomes a clickable button.
Example usage inside a component:
You can write [Button(nameof(MyMethod))] above a method, and when viewing that component in the Inspector, a button labeled **"My Method"** will appear. Clicking that button immediately runs the method.
This is extremely useful for **testing**, **debugging**, or **quickly triggering behaviors** in the Editor without running the full game.

🏷️ **Custom Tag Definition — "ButtonAttribute"**

This is the **marker** that tells Unity which methods should appear as clickable buttons.
When Unity loads the Inspector for an object, it scans all its methods looking for this marker.

🔷 **Purpose**

It identifies methods that should become buttons in the Inspector.
It also allows you to optionally define a **custom label** for each button.

🔷 **Variable inside**

- **Label**
  This holds the name that should appear on the button.
  If it is left empty, the button automatically uses the name of the method itself (but formatted nicely for readability).

🔷 **How it works conceptually**

When the Editor scans a script:

1. It looks for methods marked with [Button].
2. It checks if that method has **no parameters** (since buttons can't pass arguments).
3. For each valid one, Unity draws a button in the Inspector.
4. Clicking the button calls that method directly in the editor environment.

🔷 **Recommended usage**

You can safely link the button's label to the method name using Unity's built-in *nameof* feature.
This ensures that if you later rename the method, the label automatically updates and never breaks.
Example idea:
[Button(nameof(ResetPosition))]
→ Creates a button labeled "Reset Position".

🧭 **Custom Editor — "ButtonDrawerEditor"**

This part is what actually **draws the buttons** inside Unity's Inspector window.
It replaces the default inspector drawing process for all components (because it extends Unity's base editor class).
This custom editor only works **inside the Unity Editor**, not in a built game.

🖼️ **Method: OnInspectorGUI**

This is Unity's main process for displaying the Inspector interface.
Here, it does two things in sequence:

1. **Draws all custom buttons** that correspond to methods using the [Button] marker.
2. **Draws the regular inspector fields** (the normal variables and properties).

It also adds a small vertical gap between the custom buttons and the standard fields to keep the interface clean and readable.

⚙️ Method: DrawButtons

This is the **core** of the system — it searches for methods with the button tag and creates visual buttons for them.
Here's what happens step by step:

1. **Finds all suitable methods** inside the component being viewed.
   It filters them by the following rules:
   - The method must belong to the component being inspected.
   - It can be public or private.
   - It must not require parameters.
   - It must be marked with the [Button] tag.
2. **For each matching method**, it:
   - Reads the tag information to get the label text (if one was provided).
   - If no label is set, it automatically formats the method name into a readable button title.
3. **Draws a clickable button** in the Inspector using Unity's layout system.
4. **When clicked**, the button does the following:
   - Records the action for Unity's undo system (so the change can be reversed).
   - Calls (executes) the method directly on the component instance.
   - Marks the object as "changed" so Unity updates it visually in the editor.

This allows you to safely run editor-only actions — for example:

- Resetting values
- Creating or clearing objects
- Triggering setup or cleanup routines
- Testing AI or animation logic directly in the editor

✨ Method: FormatMethodName

This helper converts raw method names into a **readable label format** for the buttons.
It takes a name like GenerateLevelMap and turns it into **"Generate Level Map"**.
How it does this conceptually:

1. It detects transitions from lowercase to uppercase letters and inserts spaces.
   Example: "ResetPosition" → "Reset Position".
2. It separates sequences of uppercase letters from normal words to improve readability.
   Example: "GetIDText" → "Get ID Text".
3. It trims any extra spaces and returns the clean, formatted version.

This ensures that all buttons look consistent and user-friendly in the Inspector.

🧱 Supporting Concepts and Tools

The script also uses Unity's built-in systems to make everything robust and editor-safe:

- **Undo System**
  Records the state before running a button action, allowing you to revert changes easily using the Undo command.
- **Dirty Flag System**
  Marks the component as "modified" after the button is pressed, ensuring the editor properly saves any changes made by the method.
- **Reflection**
  The system looks through the script's methods at runtime to identify which ones have the [Button] marker.
  This allows it to find and display them automatically without manual configuration.

🧪 Example of Usage in Unity

Imagine you have a script controlling a player:

```
[Button(nameof(ResetStats))]
void ResetStats() {
   // logic to reset values
}
```

In the Inspector, you will now see a button labeled “Reset Stats”.
Clicking it will immediately run that method on the selected object, even when the game is not playing.
You can also add multiple buttons for different purposes:

```
[Button(nameof(SpawnEnemies))]
void SpawnEnemies() { ... }

[Button(nameof(ClearEnemies))]
void ClearEnemies() { ... }
```

Both buttons will appear stacked in the Inspector and trigger the respective methods.

✅ Summary of Behavior

1. You mark one or more methods with [Button].
2. The editor scans your component and finds those methods.
3. It draws corresponding buttons in the Inspector, one per method.
4. When you click a button:
    - The method is immediately called on the selected object.
    - Unity records and tracks changes made by that method.
5. The rest of the inspector content is drawn normally below the buttons.

🎯 Benefits and Uses

- **Improves workflow:** lets you test or reset values without entering Play mode.
- **Reduces repetitive setup:** automate initialization or debugging tasks.
- **Keeps editor clean:** no need for special menus or editor windows.
- **Safe to use:** supports Undo and change tracking.
- **Automatic formatting:** turns method names into readable button labels.