1. Inspector Attributes (namespace MVsToolkit.Dev)

All these attributes are meant to be used on **serialized fields or methods** in your components (public fields, or private fields with [SerializeField]).

1.1 TabAttribute

Purpose
Group your fields into **tabs** in the inspector. A tab starts at the field where you put the attribute and includes all following fields until the next tab (or end of the script).

Usage

```
using MVsToolkit.Dev;

using UnityEngine;


public class PlayerSettings : MonoBehaviour

{

  [Tab("Movement")]

  public float speed;

  public float jumpForce;


  [Tab("Combat")]

  public int maxHealth;

  public int damage;

}
```

- In the inspector you'll get a **toolbar** with tabs: "Movement", "Combat".
- All fields after [Tab("Movement")] appear in the *Movement* tab, up until the next [Tab].
- Tabs can be combined with FoldoutAttribute to add sub-groups inside a tab (see below).

Notes

- Only the **first field** that has [Tab("Name")] defines where the tab starts; the tab continues until another tab or until there are no more fields.
- You can have any number of tabs in the same component.

1.2 FoldoutAttribute

Purpose
Create a **collapsible foldout group** in the inspector. All fields after the one marked with [Foldout] will be grouped under that foldout, until the next foldout (or new tab).

Usage

```
using MVsToolkit.Dev;

using UnityEngine;


public class EnemySettings : MonoBehaviour

{

  [Foldout("Stats")]

  public int health;

  public int attack;

  public int defense;


  [Foldout("Loot")]

  public int gold;

  public GameObject[] possibleDrops;

}
```

- The inspector shows two foldouts: **Stats** and **Loot**.
- All fields after [Foldout("Stats")] are inside the *Stats* group until [Foldout("Loot")].

With Tabs

```
[Tab("Combat")]

[Foldout("Stats")]

public int health;

public int attack;
```

- Here, *Stats* foldout appears **inside** the *Combat* tab.

Notes

- Foldouts work both **globally** (no tab active) and inside a specific tab.
- State (open/closed) is remembered per component type & foldout name.

1.3 InlineAttribute

Purpose
Draw a **struct or class inline**, using as little vertical space as possible, instead of the default big foldout.

Usage

```
using MVsToolkit.Dev;
using UnityEngine;


[System.Serializable]
public struct Stats
{
   public int health;
   public int attack;
   public int defense;
}


public class Character : MonoBehaviour
{
   [Inline]
   public Stats baseStats;
}
```

What you get

- Instead of a single foldout for baseStats, the inspector lays out the **child fields side by side** (wrapping onto multiple lines if needed).
- Great for small data containers (structs/classes) you want to see at a glance.

Notes

- Works best with small to medium numbers of simple fields (bool, float, int, etc.).
- The underlying data & serialization are unchanged; only the layout changes.

1.4 ButtonAttribute

Purpose
Expose **methods as clickable buttons** in the inspector, so you can trigger actions directly from the editor (reset values, spawn things, run tests, etc.).

Where to put it
On **methods**, not on fields.

Basic usage

```
using MVsToolkit.Dev;

using UnityEngine;


public class Spawner : MonoBehaviour

{

  public GameObject prefab;


  [Button]

  void SpawnOne()

  {

    Instantiate(prefab, transform.position, Quaternion.identity);

  }

}
```

- In the inspector, a button named **“Spawn One”** appears (nicified from the method name).
- Clicking the button calls the method **on the inspected instance**.

Passing parameters

ButtonAttribute takes a params object[] list of parameters. These can be:

- **Literal values** (numbers, strings, etc.), or

- **Names of fields** on the same component (as strings). These names are automatically resolved to the current field values at call time.

ButtonAttribute

Example – constant parameters:

```
[Button(3)]
void SpawnMany(int count)
{
   for (int i = 0; i < count; i++)
      Debug.Log("Spawn " + i);
}
```

Example – using field values as parameters:

```
public int spawnCount;

[Button("spawnCount")]
void SpawnUsingInspectorValue(int count)
{
   Debug.Log("Will spawn: " + count);
}
```

- Here, "spawnCount" is treated as the **name of a field**, not as the string "spawnCount".
  The system finds that field on the component and passes its **current value**.

Limitations

- You **cannot** apply [Button] more than once on the same method (AllowMultiple = false).

ButtonAttribute

- Make sure your method signature matches the parameters (number & types), otherwise you'll get runtime reflection errors in the editor.

1.5 DropdownAttribute

Purpose
Draw a **dropdown in the inspector** instead of a free-edit field, using either:

- A **hard-coded list** of values, or
- A **reference to a field/collection** that contains the options.

Supported field types

- string
- float
- int

1.5.1 Hard-coded values

```
using MVsToolkit.Dev;
using UnityEngine;

public class Example : MonoBehaviour
{
   [Dropdown("Easy", "Medium", "Hard")]
   public string difficulty;
}
```

- The inspector shows a dropdown with **Easy / Medium / Hard**.
- Selecting a value sets the underlying difficulty string.

For numbers:

```
[Dropdown(0f, 0.5f, 1f)]
public float volume;

[Dropdown(1, 5, 10, 20)]
public int enemyCount;
```

1.5.2 Reference-based values

Use this when the choices come from another field (array/list) either on the same component or from a referenced object (accessed by a dot path).

public string[] difficultyOptions = { "Easy", "Medium", "Hard" };

[Dropdown("difficultyOptions")]

public string difficulty;

- "difficultyOptions" is a **path** to a field that contains the options.
- At runtime in the inspector, the dropdown is filled from the current contents of that array.

You can go through references with dot notation, e.g.:

public DataHolder dataHolder;

[Dropdown("dataHolder.availableNames")]

public string selectedName;

The path is resolved via reflection, so invalid names will simply result in an empty dropdown.

Notes & limitations

- Only one [Dropdown] per field (AllowMultiple = false).

DropdownAttribute

- If the referenced data cannot be resolved or is empty, the field falls back to a normal inspector field (but visually disabled).
- Changing options at runtime (in edit mode) will affect what appears in the dropdown next time the inspector is redrawn.

1.6 ShowIfAttribute

Purpose
Show a field **only when a condition is met**. If the condition is not satisfied, the field is hidden in the inspector.

Supported conditions

- **bool** field

- **enum** field

Usage

```
using MVsToolkit.Dev;
using UnityEngine;

public class ConditionalExample : MonoBehaviour
{
    public bool showAdvanced;

    [ShowIf("showAdvanced", true)]
    public float advancedValue;


    public enum Mode { Normal, Debug }
    public Mode mode;

    [ShowIf("mode", Mode.Debug)]
    public string debugText;
}
```

- advancedValue appears only when showAdvanced == true.
- debugText appears only when mode == Mode.Debug.

Parameters

[ShowIf(string conditionField, object compareValue)]

- conditionField: name of the **bool** or **enum** field on the same component.
- compareValue: value that the conditionField must equal for the field to be shown (e.g. true, or an enum value).

Multiple conditions

- The attribute is declared with AllowMultiple = true, so you can stack several conditions on the same field if needed.

ShowIfAttribute

Notes

- If the condition field cannot be found or is of an unsupported type, the field is **shown** and a warning is logged in the console (so you don't accidentally hide things forever).

1.7 HideIfAttribute

Purpose
Hide a field **when** a condition is met. If the condition is not satisfied, the field is displayed normally.

Supported conditions

- **bool** field
- **enum** field

Usage

```
using MVsToolkit.Dev;
using UnityEngine;

public class HideExample : MonoBehaviour
{
    public bool isReadOnlyMode;

    [HideIf("isReadOnlyMode", true)]
    public string editableText;



    public enum AIState { Idle, Patrol, Attack }
    public AIState currentState;

    [HideIf("currentState", AIState.Idle)]
```

```
    public float patrolRadius;
}
```

- editableText is hidden when isReadOnlyMode == true.
- patrolRadius is hidden when currentState == AIState.Idle.

Parameters

[HideIf(string conditionField, object compareValue)]

- conditionField: name of bool/enum field to check.
- compareValue: value that hides the property when equal to the field.

Multiple conditions

- Also AllowMultiple = true, so you can stack several [HideIf] on the same field.

HideIfAttribute

Notes

- Same as ShowIf: if the condition is misconfigured (field not found, wrong type, etc.), the field is shown and a warning is logged.

1.8 ReadOnlyAttribute

Purpose
Make a field **non-editable** in the inspector (grayed out) while still showing its value and still serializing it.

Usage

```
using MVsToolkit.Dev;
using UnityEngine;

public class ReadOnlyExample : MonoBehaviour
{
    [ReadOnly]
    public int runtimeValue;

    void Update()
```

```
  {
    runtimeValue++;
  }
}
```

- runtimeValue will update visually in the inspector at runtime, but you **cannot edit** it manually.
- Useful to display computed or debug values that should not be changed in the editor.

Notes

- The attribute works for both simple types and complex types (arrays, structs, nested objects): everything under that field is drawn disabled.

1.9 SceneNameAttribute

Purpose
Turn a string field into a dropdown listing the **names of all scenes enabled in the Build Settings**.

Usage

```
using MVsToolkit.Dev;
using UnityEngine;

public class SceneLoader : MonoBehaviour
{
  [SceneName]
  public string sceneToLoad;
}
```

- In the inspector, sceneToLoad is a **dropdown** populated with all scenes that:
    - Are listed in **File → Build Settings... → Scenes In Build**, and
    - Are **enabled** there.

Notes

- The field must be a string. If not, a help message appears saying you should use it on a string.

SceneNameDropdown_PropertyDrawer

- At runtime, you can call SceneManager.LoadScene(sceneToLoad); without worrying about typos.

1.10 WatchAttribute

Purpose
Display fields and their **live values on screen at runtime**, without needing a custom UI or debugger. It creates a simple on-screen HUD listing all watched values.

Usage

```
using MVsToolkit.Dev;

using UnityEngine;


public class Player : MonoBehaviour

{

  [Watch]

  public int health;


  [Watch]

  [SerializeField] float speed;

}
```

What happens at runtime

- On game start, the system scans **all MonoBehaviour instances** in the scene for fields marked with [Watch].

WatchDisplay

- It creates a hidden WatchDisplay GameObject that persists across scene loads.
- During play, a small panel appears (bottom-left of the screen) showing lines like:
- Player.health = 100
- Player.speed = 5

and updates every frame.

WatchDisplay

Notes & limitations

- Targets: **fields** only (AttributeTargets.Field, AllowMultiple = false).

WatchAttribute

- This runs in both the editor **Play mode** and in builds (since it uses RuntimeInitializeOnLoadMethod).
- Values are displayed using ToString(). For complex objects, override ToString() if you want a more readable output.
- To stop watching a variable, simply remove the [Watch] attribute or the component.

---

2. Editor Windows & Tools (Better Interface)

Here's how to **use** the editor windows & tools you uploaded, from a user point of view.

2.1 Single Component Window (SingleComponentWindow)

What it is
A **floating inspector** that shows a single component in its own mini-window. Very handy when:

- You want to **pin** a component while selecting other objects.
- You want several inspectors side by side for comparison.

SingleComponentWindow

How to open it

You have two entry points:

1. **From the component context menu**
    - In the **Inspector**, right-click the header of any component (Transform, custom script, etc.).
    - Choose **"Open in new Tab"** from the context menu.
      This menu item is registered on CONTEXT/Component/Open in new Tab.
2. **From the header icon**

- In the Inspector, look at the component's header bar (where its name is).
- A small icon is injected there with the tooltip **"Open in new Tab"**.
- Click it to open the same window for that component.

Both ways call SingleComponentWindow.Show(component) under the hood.

How the window behaves

- Opens as a **popup window** centered on the screen, with a default size (~400×200).

SingleComponentWindow

- Title shows the **GameObject name**.
- Inside, you see the **exact same inspector** you'd see in the main Inspector for that component – including all your custom attributes.
- You can **scroll** if the content is taller than the window.

SingleComponentWindow

Moving & Resizing

- **Move**: click & drag on the gray header (but not the close button).
- **Resize (horizontal)**: drag the right edge of the window; width is clamped between a minimum and maximum value.

SingleComponentWindow

- Height auto-fits to content within [30, 800] px bounds.

Closing

- Click the **"×"** button in the top-right of the header.

SingleComponentWindow

Typical use cases

- Keep one component's inspector open while selecting other objects (e.g. a manager, camera controller, global settings, etc.).
- Compare settings from two different objects side by side by opening two windows.
- Have a "debug inspector" window pinned on a second monitor.

2.2 Scene Browser Popup (SceneBrowserPopUp)

What it is
A small **popup window with a searchable list of scenes** in your project, with support for favorites and quick loading.

How to open it

The popup is opened from the **Hierarchy** window:

- When **not in Play mode**, click with the **left mouse button** on a Hierarchy row that doesn't correspond to a GameObject (for example, an empty row/scene header area).
- When that row is clicked and the underlying object is null, the tool calls:
- PopupWindow.Show(popUpRect, new SceneBrowserPopUp());

In practice:

Try clicking on the empty space or header line in the Hierarchy (below or around the scene name); the popup will appear right under your cursor.

What you see in the popup

- A **search bar** at the top (toolbar style).
- A scrollable list of scenes, split into:
    1. **Favorite scenes** (top section, with a horizontal separator).
    2. **Other scenes** (the rest).
- Each scene is drawn as:
    - A **button** with the scene name (left-aligned).
    - A star icon on the right (★ or ☆) when you hover the row, to toggle favorite status.

SceneBrowserPopUp

The scenes list is built from **all .unity scenes under the Assets/ folder**, not just those in Build Settings.

SceneBrowserPopUp

Filtering & favorites

Search

- Type in the search bar to filter scenes **by name** (case-insensitive).
- Only scenes whose name contains the typed text are shown.

Favorites

- Hover a row:
    - For a favorite scene, you can click the star button to **remove it from favorites**.
    - For a non-favorite scene, you can click to **add it to favorites**.

SceneBrowserPopUp

- Favorite state is **persisted** using EditorPrefs (key "SceneBrowser_Data"), so your favorites survive after closing Unity.

Undo favorite changes

- While the popup is focused, press **Ctrl+Z** to undo the last favorite/unfavorite action.
- Changes are tracked on a small internal stack.

Opening scenes

- Click on a scene button to open that scene.

If the current active scene has **unsaved changes**, you'll get a dialog:

SceneBrowserPopUp

"The scene has been modified. Would you want to save it?"
Buttons: **Yes / No / Cancel**

- **Yes** → Save all open scenes, then open the selected one.
- **No** → Open the selected scene **without saving** changes.
- **Cancel** → Do nothing; stay on the current scene.

If the scene is **not modified**, the tool just opens the selected scene immediately.

Window size behavior

- The popup calculates its height based on:
    - Search bar height,
    - Number of visible scenes (after filtering),
    - Some padding and dividers.
- Height is clamped to a **maximum** (defaults around 300 px); width is slightly larger when clamped (250–275 px).