MVsToolkit Attributes Documentation

Table of Contents

Introduction

MVsToolkit provides a collection of custom attributes that enhance the Unity Inspector experience. These attributes allow you to create more intuitive and organized Inspector interfaces, add runtime debugging capabilities, and improve your development workflow.

All attributes are in the `MVsToolkit.Dev` namespace. Make sure to add `using MVsToolkit.Dev;` at the top of your scripts to use them.

Inspector Display Attributes

ReadOnly

Description: Makes a field read-only (grayed out) in the Inspector, preventing editing while still displaying the value.

Usage:

```
[ReadOnly]
public int currentHealth = 100;
```

Use Cases:

- Display calculated values
- Show runtime state without allowing modification
- Prevent accidental changes to important values

Inline

Description: Draws a struct or class inline in the Inspector without a foldout, displaying all nested fields directly.

Usage:

```
[System.Serializable]
public class PlayerStats
{
    public int health;
    public int mana;
}

[Inline]
public PlayerStats stats;
```

Use Cases:

- Display simple data structures compactly
- Reduce Inspector clutter for small classes
- Make related fields easier to see at a glance

Conditional Display Attributes

ShowIf

Description: Shows the field in the Inspector only when a specified condition is met. Works with boolean fields and enum comparisons.

Parameters:

- `conditionField` (string): Name of the field to evaluate
- `compareValue` (object): Value that triggers visibility

Supported Types:

- Boolean conditions
- Enum comparisons

Multiple Conditions: Can be used multiple times on the same field (all conditions must be true).

Usage:

```
public bool enableSpecialAbility;

[ShowIf("enableSpecialAbility", true)]
public float abilityPower = 10f;

public enum WeaponType { Melee, Ranged, Magic }
public WeaponType weaponType;

[ShowIf("weaponType", WeaponType.Magic)]
public int manaCost;
```

Use Cases:

- Show weapon-specific settings
- Display options only when a feature is enabled
- Create context-sensitive Inspector layouts

HideIf

Description: Hides the field in the Inspector when a specified condition is met. Opposite of ShowIf.

Parameters:

- `conditionField` (string): Name of the field to evaluate
- `compareValue` (object): Value that triggers hiding

Supported Types:

- Boolean conditions
- Enum comparisons

Multiple Conditions: Can be used multiple times on the same field (any matching condition will hide the field).

Usage:

```
public bool isInvincible;

[HideIf("isInvincible", true)]
public int armor;

public enum GameMode { Normal, Debug, Testing }
public GameMode mode;

[HideIf("mode", GameMode.Debug)]
public float difficulty;
```

Use Cases:

- Hide irrelevant options based on configuration
- Simplify Inspector when certain features are disabled
- Create mutually exclusive field groups

Organization Attributes

Tab

Description: Creates a tab in the Inspector that groups fields until the next Tab attribute is encountered. Helps organize large inspectors into logical sections.

Parameters:

- `tabName` (string): Label for the tab section

Usage:

```
[Tab("Movement")]
public float speed;
public float jumpHeight;

[Tab("Combat")]
public int damage;
public float attackRate;

[Tab("Audio")]
public AudioClip jumpSound;
public AudioClip attackSound;
```

Use Cases:

- Organize complex components with many fields
- Separate different aspects of gameplay (movement, combat, visuals)
- Improve Inspector readability for large scripts

CloseTab

Description: Closes the current tab section. Used to explicitly end a tab group.

Usage:

```
[Tab("General")]
public string playerName;
public int level;

[CloseTab]
[Tab("Skills")]
public float strength;
```

Use Cases:

- Explicitly end a tab section before starting another
- Create clear boundaries between tab groups

- Better control over tab organization

Foldout

Description: Groups multiple fields under a collapsible foldout section in the Inspector. Fields following this attribute will be grouped until a CloseFoldout or another organizing attribute is encountered.

Parameters:

- `foldoutName` (string): Label for the foldout section

Usage:

```
[Foldout("Player Settings")]
public string playerName;
public int playerLevel;
public float experience;

[Foldout("Graphics Settings")]
public int resolution;
public bool fullscreen;
```

Use Cases:

- Group related fields together
- Reduce visual clutter in the Inspector
- Create collapsible sections for optional settings

CloseFoldout

Description: Explicitly closes the current foldout section.

Usage:

```
[Foldout("Basic Stats")]
public int health;
public int mana;

[CloseFoldout]
public string notes; // Not in foldout
```

Use Cases:

- Explicitly end a foldout section
- Control precisely which fields are grouped
- Create clear boundaries between sections

Selection Attributes

Dropdown

Description: Displays a dropdown menu in the Inspector with predefined values. Can use hardcoded values or reference another field/property.

Parameters (Reference Mode):

- `path` (string): Name of the field or property containing dropdown values

Parameters (Hardcoded Mode):

- `objects` (params object[ ]): Hardcoded values to display

Supported Types:

- string
- float
- int

Usage:

```
// Hardcoded values
[Dropdown("Easy", "Medium", "Hard")]
public string difficulty;

[Dropdown(16, 32, 64, 128, 256)]
public int textureSize;

// Reference to another field
public string[] availableWeapons = { "Sword", "Bow", "Staff" };

[Dropdown("availableWeapons")]
public string selectedWeapon;
```

Use Cases:

- Provide predefined options for settings
- Ensure valid values are selected
- Reference dynamic lists of options

SceneName

Description: Displays a dropdown with all scene names from the Build Settings. The field must be a string.

Usage:

```
[SceneName]
public string nextSceneName;

[SceneName]
public string mainMenuScene;
```

Use Cases:

- Select scenes for level transitions
- Reference scenes without hardcoding names
- Avoid typos in scene names

TagName

Description: Displays a dropdown of all tags defined in the project. The field must be a string.

Usage:

```
[TagName]
public string enemyTag;

[TagName]
public string playerTag;
```

Use Cases:

- Select tags for collision detection
- Reference tags without typos
- Ensure valid tag names are used

Scene View Attributes

Handle

Description: Draws an interactive handle in the Scene view for Vector3 or Vector2 fields, allowing you to manipulate positions visually.

Parameters:

- `spaceType` (Space): Coordinate space - `Space.Self` (Local) or `Space.World` (Global). Default: `Space.Self`
- `drawType` (HandleDrawType): Handle shape - `Default`, `Sphere`, or `Cube`. Default: `Default`
- `preset` (ColorPreset): Handle color - `White`, `Red`, `Green`, `Blue`, `Yellow`, `Cyan`, `Magenta`. Default: `White`
- `size` (float): Handle size. Default: `0.2`

Usage:

```
// Simple handle with defaults
[Handle]
public Vector3 targetPosition;

// Handle with global coordinates
[Handle(Space.World)]
public Vector3 worldPosition;

// Colored sphere handle
[Handle(Space.Self, HandleDrawType.Sphere, ColorPreset.Red)]
public Vector3 spawnPoint;

// Large cyan cube handle in world space
[Handle(Space.World, HandleDrawType.Cube, ColorPreset.Cyan, 1f)]
public Vector3 patrolPoint;
```

Handle Types:

- **Default**: Standard position handle
- **Sphere**: Spherical gizmo
- **Cube**: Cubic gizmo

Use Cases:

- Position waypoints visually in the Scene view
- Set spawn points or target locations
- Adjust positions without typing coordinates
- Visualize important positions with different colors

Method Attributes

Button

Description: Creates a button in the Inspector that calls the decorated method when clicked. Can pass parameters to the method.

Parameters:

- `parameters` (params object[ ]): Values or field names to pass as arguments

Parameter Passing:

- Direct values: `[Button(10, "PlayerName")]` - passes these exact values
- Field references: `[Button("healthValue")]` - passes the value of the field named "healthValue"

Usage:

```
// Simple button with no parameters
[Button]
void ResetPlayer()
{
    health = 100;
    Debug.Log("Player reset!");
}

// Button with direct string parameter
[Button("Test Message")]
void LogMessage(string message)
{
    Debug.Log(message);
}

// Button with field reference
public int damageAmount = 50;
```

```
[Button("damageAmount")]
void TakeDamage(int damage)
{
    health -= damage;
    Debug.Log($"Took {damage} damage");
}

// Button with multiple parameters
[Button(10, 5.5f, "Bonus")]
void ComplexMethod(int count, float multiplier, string label)
{
    Debug.Log($"{label}: {count * multiplier}");
}
```

Use Cases:

- Test methods directly from the Inspector
- Provide designer-friendly controls
- Debug and iterate quickly without writing editor scripts
- Execute common operations during development

Runtime Debugging Attributes

Watch

Description: Displays the field's name and current value on screen during Play Mode. Useful for debugging and monitoring values in real-time.

Limitations:

- Can only be used once per variable
- Only visible during Play Mode

Usage:

```
[Watch]
public int currentScore;

[Watch]
public float playerSpeed;
```

```
[Watch]
private bool isGrounded; // Works with private fields too
```

Display:

- Values appear on screen as: `ClassName.fieldName = value`
- Positioned in the lower-left corner
- Updates automatically during gameplay

Use Cases:

- Monitor values during gameplay without the debugger
- Track changing values in real-time
- Debug gameplay without pausing
- Quick visibility into game state

Technical Note: Uses the `WatchDisplay` component which is automatically created at runtime. Values are displayed using OnGUI in the lower-left corner of the screen.

Type Reference Attributes

InterfaceReference

Description: A serializable wrapper that allows you to reference Unity objects by their interface type in the Inspector. Normally, Unity cannot serialize interface references, but this class provides that capability.

Type Parameter:

- `T` : The interface type (must be a class type)

Key Property:

- `Value` : Returns the referenced object cast as the interface type

Supported Object Types:

- MonoBehaviour
- ScriptableObject
- Any UnityEngine.Object that implements the interface

Special Features:

- Supports `IEnumerable<T>` when the stored object is a GameObject

- Returns all components on a GameObject that implement the specified interface
- Implicit conversion operator for cleaner syntax

Usage:

```
// Define an interface
public interface IInteractable
{
    void Interact();
    string GetDescription();
}

// Implement the interface
public class Door : MonoBehaviour, IInteractable
{
    public void Interact()
    {
        Debug.Log("Door opened!");
    }

    public string GetDescription()
    {
        return "A wooden door";
    }
}

// Use InterfaceReference in another script
public class Player : MonoBehaviour
{
    public InterfaceReference<IInteractable> currentInteractable;

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.E) && currentInteractable.Value != r
        {
            // Access the interface through .Value
            currentInteractable.Value.Interact();

            // Or use implicit conversion
            IInteractable interactable = currentInteractable;
            Debug.Log(interactable.GetDescription());
        }
    }
}
```

Advanced Usage - IEnumerable:

```
public interface IDamageable
{
    void TakeDamage(int damage);
}

// Get all IDamageable components on a GameObject
public InterfaceReference<IEnumerable<IDamageable>> damageableComponents;

void DealAreaDamage()
{
    if (damageableComponents.Value != null)
    {
        foreach (var damageable in damageableComponents.Value)
        {
            damageable.TakeDamage(10);
        }
    }
}
```

Inspector Behavior:

- Drag and drop any Unity Object that implements the interface
- Editor will show a dropdown filtered to valid types
- Validation ensures only compatible objects can be assigned

Use Cases:

- Create flexible, interface-based systems
- Reference objects by their capabilities, not concrete types
- Build modular, loosely-coupled architectures
- Enable polymorphic behavior in the Inspector
- Implement design patterns like Strategy or Observer

Best Practices:

1. Always check if `Value` is null before using it
2. Use interfaces to define clear contracts
3. Prefer composition over inheritance
4. Use InterfaceReference for cross-system communication

Complete Example

Here's a comprehensive example demonstrating multiple attributes working together:

```
using UnityEngine;
using MVsToolkit.Dev;

public class CharacterController : MonoBehaviour
{
    [Tab("Basic Info")]
    [ReadOnly]
    public string characterID = "CHAR_001";

    public string characterName;
    public int level = 1;

    [Tab("Movement")]
    [Dropdown(1f, 2f, 3f, 5f, 10f)]
    public float moveSpeed = 3f;

    public float jumpForce = 5f;

    [Watch]
    public bool isGrounded;

    [Tab("Combat")]
    [Foldout("Stats")]
    public int maxHealth = 100;

    [Watch]
    public int currentHealth = 100;

    public int attackDamage = 10;
    [CloseFoldout]

    public bool hasMagic;

    [ShowIf("hasMagic", true)]
    public int manaPoints = 50;

    [ShowIf("hasMagic", true)]
    [Dropdown("Fire", "Ice", "Lightning")]
    public string magicType;
```

```
    [Tab("Configuration")]
    [SceneName]
    public string respawnScene;

    [TagName]
    public string enemyTag = "Enemy";

    [Handle(Space.World, HandleDrawType.Sphere, ColorPreset.Green)]
    public Vector3 checkpointPosition;

    [Tab("Debug")]
    [Button]
    void ResetCharacter()
    {
        currentHealth = maxHealth;
        transform.position = checkpointPosition;
        Debug.Log("Character reset to checkpoint!");
    }

    [Button("attackDamage")]
    void TestDamage(int damage)
    {
        currentHealth -= damage;
        Debug.Log($"Took {damage} damage. Health: {currentHealth}");
    }

    [Button(999, 999)]
    void MaxOut(int health, int mana)
    {
        currentHealth = health;
        if (hasMagic) manaPoints = mana;
        Debug.Log("Stats maxed out!");
    }
}
```

Tips and Best Practices

Organization

- Use **Tab** for major sections of complex components
- Use **Foldout** for grouping related fields within a section

- Keep related settings together for better workflow

Conditional Display

- Combine **ShowIf**/**HideIf** to create dynamic Inspectors
- Use enum-based conditions for state machines
- Multiple conditions can create complex visibility rules

Debugging

- Add **Watch** to critical values you need to monitor
- Use **Button** attributes for testing and iteration
- Combine **ReadOnly** with calculated values for visibility

Scene Tools

- Use **Handle** with different colors for different position types
- Use **Space.World** for absolute positions
- Use **Space.Self** for positions relative to the object

Type Safety

- Use **Dropdown** to prevent invalid values
- Use **SceneName** and **TagName** to avoid typos
- Use **InterfaceReference** for flexible, type-safe references

---

Namespace Reference

All attributes (except CloseFoldout and CloseTab) are in the `MVsToolkit.Dev` namespace:

```
using MVsToolkit.Dev;
```

The following attributes are in the global namespace:

- `CloseFoldout`
- `CloseTab`

Compatibility

- **Unity Version**: Compatible with Unity 2020.3 and later
- **Platform**: All platforms supported by Unity
- **Editor Only**: Most attribute drawers only affect the Editor Inspector
- **Runtime**: Only the **Watch** attribute affects runtime display

---

Troubleshooting

Attribute Not Working

- Ensure you've added `using MVsToolkit.Dev;`
- Check that the field is serializable (public or has `[SerializeField]`)
- Verify attribute parameters are correct type

ShowIf/HideIf Not Updating

- Ensure the condition field is spelled correctly (case-sensitive)
- Check that the condition field is before the dependent field
- Verify the compare value matches the field type

Button Not Appearing

- Attribute must be on a method, not a field
- Method should be private or public (not static)
- Check that parameter types match the attribute arguments

Handle Not Visible

- Field must be Vector3 or Vector2
- Ensure the object is selected in the Scene view
- Check that Gizmos are enabled in the Scene view

Watch Not Displaying

- Ensure you're in Play Mode
- Check that the field is not null
- Verify the WatchDisplay GameObject is created (automatic)

Version History

- **v0.7.3**: Current version with all documented attributes
- Features include conditional display, runtime debugging, scene tools, and interface references

---

Support

For issues, feature requests, or questions about MVsToolkit attributes, please refer to the main MVsToolkit documentation or repository.

This documentation covers all attributes available in MVsToolkit v0.7.3. Always refer to the latest version for up-to-date information.