

# **Power Inspector Documentation**

Version 0.9.5b

# Table of Contents

Introduction.....	4
What Is Power Inspector?.....	4
Decoupled Design.....	4
Customizable.....	4
Installation.....	5
Supported Unity Versions.....	5
How To Install.....	5
Getting Started.....	6
Troubleshooting.....	7
Type Not Found Errors.....	7
Power Inspector Window Unresponsive.....	7
Check That Your Unity Version Is Supported.....	7
Conflict with Json.Net for Unity.....	7
Clean Reinstall.....	7
Updating.....	8
Checking For Updates.....	8
Features.....	9
List of Main Features.....	9
Toolbar.....	10
Navigation Buttons.....	11
View Menu.....	12
List of View Menu Items.....	12
Filter Field.....	15
Split View.....	17
Peeking.....	18
Keyboard Navigation.....	19
Navigation Arrow Key Navigation.....	19
Select Previous / Next Component.....	19
Select Previous / Next Of Type.....	20
(Limited) Multi-Selection Support.....	20
Improved Focused Field Indicators.....	20
Dynamic Prefix Column.....	21
Automatic Width Optimization.....	21
Manual Width Adjustment.....	21
Manual Width Adjustment In Custom Editors.....	21
Copy-Paste.....	22
Copy Anything.....	22
Paste Anywhere.....	22
Cross-Type Paste.....	22
Saved To Clipboard.....	23
Reset.....	24
Display Anything.....	25
Display Any Member.....	25
Members That Power Inspector Can Display.....	25
Types That Power Inspector Can Display.....	26
Note About Serialization.....	26
Context Menu.....	27
Opening The Context menu.....	27
Opening The Expanded Context Menu.....	27
Inspecting Static Members.....	28

<u>Debug Mode+</u> .....	29
<u>Debugging Static Members</u> .....	29
<u>Quick Invoke Menu</u> .....	30
<u>Tooltips</u> .....	31
<u>Hint Icon</u> .....	31
<u>Tooltip Attribute</u> .....	31
<u>XML Documentation Comment Support</u> .....	31
<u>Prefab Quick Editing</u> .....	32
<u>Attribute Enhancements</u> .....	33
<u>Controlling Member Visibility With Attributes</u> .....	33
<u>Exposing Non-Serialized Fields</u> .....	33
<u>Preferences</u> .....	35
<u>List of Preferences</u> .....	35
<u>Data Visibility</u> .....	35
<u>Data Visualization</u> .....	36

# Introduction

## What Is Power Inspector?

Power Inspector is full replacement for the [Default Inspector](#) window in that ships with Unity. The aim is to fully replicate the basic feature set of the default Inspector, and then to build on top of that with various features to speed up your workflow and to give you the tools to avoid some common pain points.

## Decoupled Design

A core ambition with Power Inspector has been to keep it as decoupled from your other code as possible.

The [Default Inspector](#) window, should you ever need it, also remains untouched and fully usable alongside Power Inspector.

This all ensures that the risk of integration remains minimal.

## Customizable

You can customize Power Inspector to fit your individual needs using dozens of [Preference items](#).

### ◆ BETA NOTICE

Please note that Power Inspector is still in beta and contains some bugs.

# Installation

## Supported Unity Versions

Power Inspector supports Unity version **5.6.7f1 and newer**.

The latest version of Unity tested to be working is **2019.1.0f2** (newer versions might work also).

**Beta** versions of Unity are not officially supported, so use Power Inspector with them at your own risk.

## How To Install

Installing Power Inspector is done using the [Asset Store window](#) inside of Unity.

1. Open the Asset Store window using the menu item **Window > Asset Store**.
2. Find the listing for Power Inspector in My Assets.
3. Click the Download button, then wait for the download to finish. If you don't see a Download button next to the asset listing, skip to the next step.
4. Click Import and wait until the loading bar disappears and the "Import Unity Package" dialog opens.
5. Click Import, then wait until the loading bar disappears. You should NOT tick or untick any items in the list, unless you know what you're doing, because that could mess up the installation.
6. If you see a dialog prompt about performing an API Update, click the button labeled "I made backup. Go ahead!"

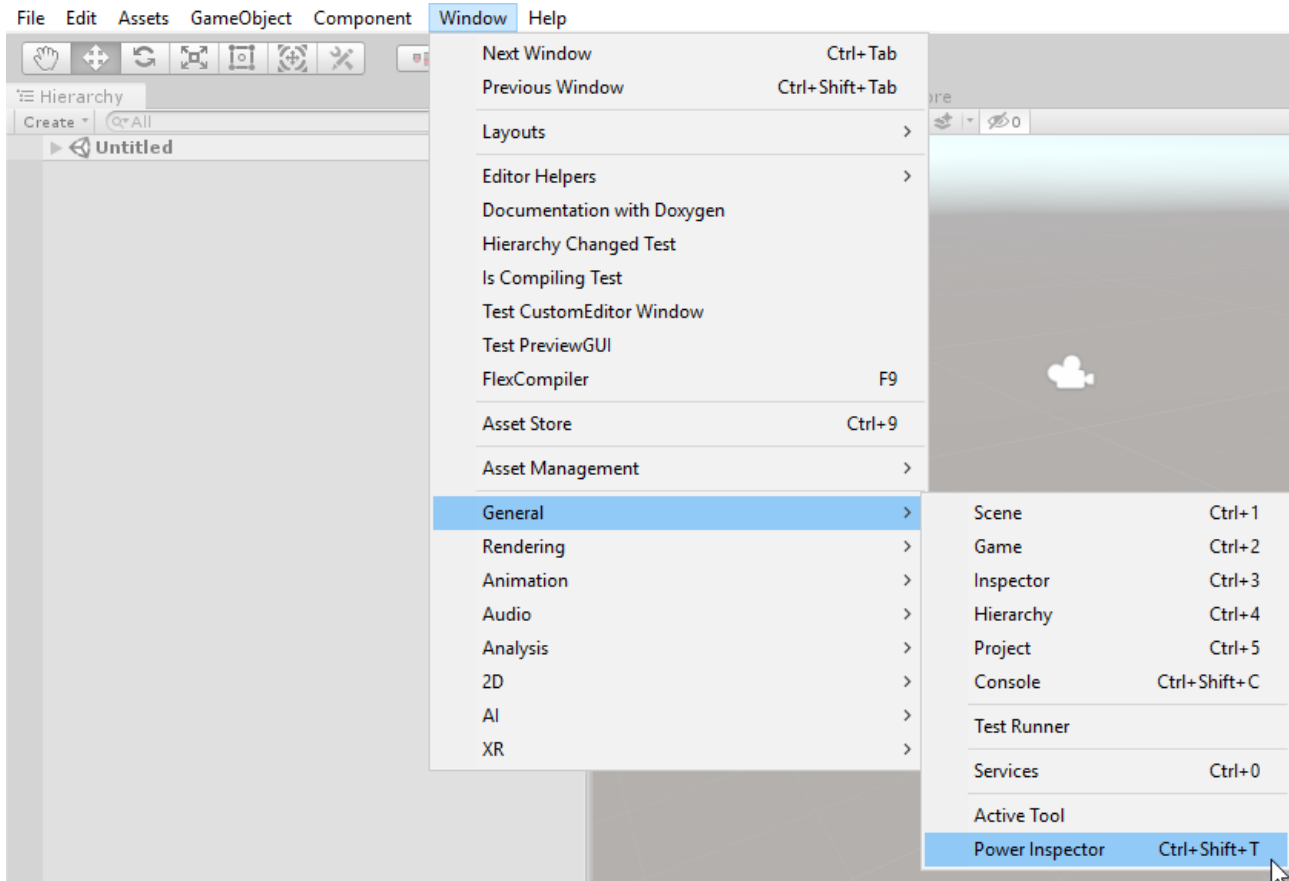
Power Inspector should now be installed and all ready to use!

If you run into any issues during installation, please refer to the [Troubleshooting page](#).

# Getting Started

Use the menu item **Window > General > Power Inspector** to open the Power Inspector Window. An alternative way to open a new Power Inspector window is using the keyboard shortcut **Ctrl+Alt+T** (Cmd+Alt+T on macOS).

It is possible to open multiple instances of the Power Inspector window.



You use the Power Inspector window like you would use the [Default Inspector](#): select an [Unity Object](#) in the [Hierarchy window](#) or the [Project window](#), and the exposed data of the [target](#) gets shown in the window, for you to view and edit as you please.

# Troubleshooting

## Type Not Found Errors

Are you suddenly seeing errors originating from Power Inspector code, possibly triggered by making changes to code? In particular, are you seeing this error in your Log window:

*“error CS0246: The type or namespace name 'x' could not be found. Are you missing an assembly reference?”*

If so, it's possible that Unity's asset loading process has failed for some reason. To fix the issue you can try the following steps:

## Power Inspector Window Unresponsive

If the Power Inspector window runs into an exception at a critical moment, it might become unresponsive. When this happens, you can try the following actions to try and fix the issue:

**NOTE:** If you have not saved your prior Layout prior to doing this step, you will need to manually reopen and reorganize your window layout.

## Check That Your Your Unity Version Is Supported

If you just installed Power Inspector, or recently updated your Unity version, and are now seeing errors in the Log window, or Power Inspector is not working properly, please check that the version of Unity you're using is among the supported versions.

## Conflict with Json.Net for Unity

Power Inspector uses Json.Net for Unity internally for thing such as copy-pasting of values. If you are already using Json.Net for Unity in your project, you might run into issues due to the two instances conflicting.

To resolve this issue, you can try deleting one of the two Json.Net for Unity instances. The one used by Power Inspect can be located inside your project folder at the path:

*Assets/Sisus/PowerInspector/Code/HelperAssets/JsonDotNet/*

## Clean Reinstall

If you're seeing errors after having tried to update Power Inspector, you can try doing a clean reinstall and see if that solves the issues.

If you are unable to solve your issues using the information provided, please contact us via the email address [support@sisus.co](mailto:support@sisus.co).

# Updating

You might want to check from time to time if Power Inspector has new updates, so you don't miss out on new features, and so that you get all the latest fixes for the latest Unity version.

Updating Power Inspector is very similar to installing it, and done using the Asset Store window inside of Unity.

## Checking For Updates

1. Open the [Asset Store window](#) using the menu item **Window > Asset Store**.
2. Find the listing for Power Inspector in My Assets.
3. If you see a button labeled "Import", that means that your installation of Power Inspector is up-to-date, and you are done. If you see a button labeled "Update", continue to the next step to update Power Inspector to the latest version.
4. Click the Update button, then wait for the download to finish.
5. Click Import and wait until the loading bar disappears and the "Import Unity Package" dialog opens.
6. Click Import, then wait until the loading bar disappears. You should **NOT** tick or untick any items in the list, unless you know what you're doing, because that could mess up the installation.
7. If you see a dialog prompt about performing an API Update, click the button labeled "I made backup. Go ahead!"

Power Inspector is now updated to the latest version.

If you see any errors, refer to the Installation Troubleshooting page.

If you're still having problems after that, please contact us via the email address [support@sisus.co](mailto:support@sisus.co).



# Features

## List of Main Features

### Filter Field

Quickly find any field in the inspector by its name, type or value.

[Find out more](#)

### Intuitive Keyboard Navigation

Arrow keys change focused field in the expected direction.

There are also several new helpful navigation shortcuts added.

[Find out more](#)

### Navigation Buttons

New navigation buttons allow you to easily jump between previously inspected targets.

[Find out more](#)

### Copy-Paste

Powerful cross-project copy-paste that works even between fields of many different types.

[Find out more](#)

### Split View

Split the inspector into two parts with the press of a button, and view two different targets right next to each other.

[Find out more](#)

### Dynamic Prefix Column

The width of field prefixes is now automatically optimized and can also be adjusted manually.

[Find out more](#)

### Debug Mode+

Easily gain access to all fields, properties and methods of Components, including non-serialized and hidden ones.

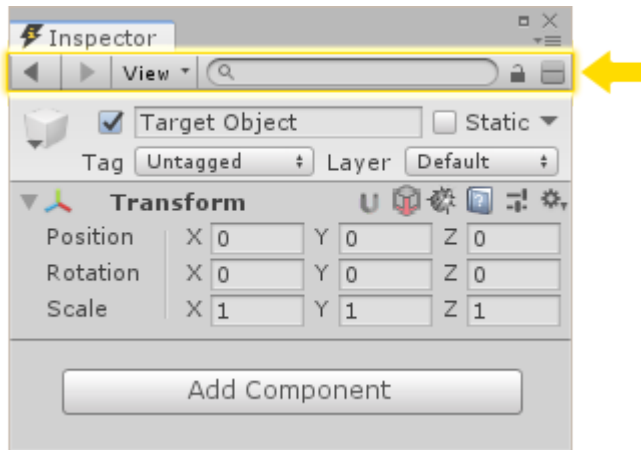
[Find out more](#)

### Quick Invoke Menu

Easily invoke any method on a Component via a dedicated header button.

[Find out more](#)

## Toolbar



The first thing you will probably notice being different in the Power Inspector Window when compared to the default Inspector, is that there's a whole new toolbar sitting on top of each view. This toolbar offers you quick and convenient access to some of the most powerful new features that Power Inspector offers.

## Navigation Buttons



The Toolbar contains back and forward buttons that enable traversing through the history of inspected Target Objects.

Right-clicking the back or the forward button opens up a context menu containing the full listing of targets in the direction in question.

This context menu also has a filter field for quickly finding a specific previously selected target.



You can also use the following shortcut keys to navigate the selection history:

### Windows Shortcut

Ctrl + Alt + Left Arrow  
Ctrl + Alt + Right Arrow

Shift + Alt + Left Arrow

Shift + Alt + Right Arrow

### macOs Shortcut

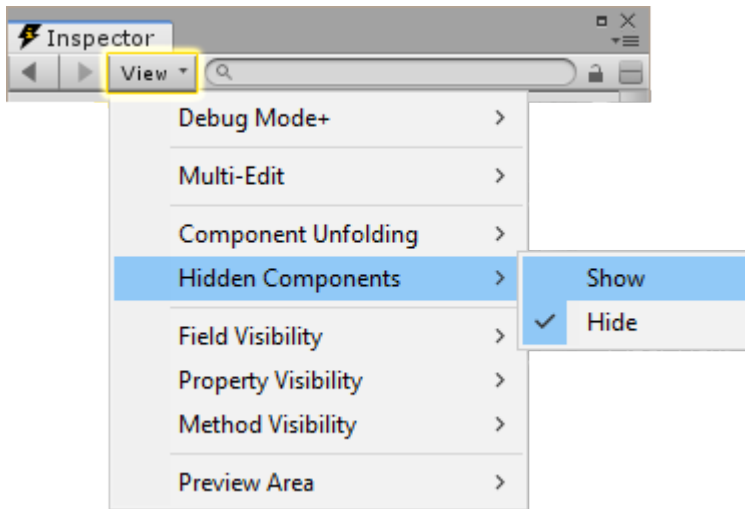
Cmd + Alt + Left Arrow  
Cmd + Alt + Right Arrow

Shift + Alt + Left Arrow

Shift + Alt + Right Arrow

Step back in inspection history.  
Step forward in inspection history.  
Open full inspection history context menu in back direction.  
Open full inspection history context menu in forward direction.

## View Menu



Clicking the View Menu button on the Toolbar opens up a context menu which allows quickly changing options related to what and how data is shown by the Power Inspector View. The contents of the View Menu can dynamically change based on what targets are currently shown in the view.

### List of View Menu Items

The contents of the View Menu can dynamically change based on what Target Objects are currently shown in the Inspector View.

Here are some of the menu items you might find in the View Menu:

#### **Debug Mode+: Off (Default)**

Disable Debug Mode+ for the view. Components, fields, properties and methods will only be shown if the current visibility settings allow it.

#### **Debug Mode+: On**

Enable Debug Mode+ for the view. All previously hidden Components and instance fields and properties are revealed, including non-serialized ones and ones hidden using attributes. Only items with the Deprecated attribute remain hidden (to avoid exceptions being thrown). The Debug Mode+ will remain active for the view even if the inspected targets are changed.

#### **Multi-Edit: Merged (Default)**

Set multi-editing mode to merged. When multiple targets of the same type are contained in the current selection, their data is merged together, so that they can be edited simultaneously. This is how the default Inspector window works.

#### **Multi-Edit: Stacked**

Set multi-editing mode to stacked. When multiple targets are selected for inspecting, they will be listed separately, one after the other. This might be a useful mode if you want to compare and/or migrate data between multiple targets.

**Component Unfolding: Unrestricted (Default)**

Allow multiple components to remain unfolded simultaneously. This is how the default Inspector window works.

**Component Unfolding: One At A Time**

Only allow one component to be unfolded at any given time. Holding down control when unfolding a Component can be used to bypass this restriction, if you e.g. need to move data between two Components. This can be a useful mode if you have GameObjects with lots of Components and want to avoid your Inspector views being cluttered.

**Hidden Components: Hide (Default)**

Components that have been hidden using HideFlags are not shown in the Inspector. This is how the default Inspector window works.

**Hidden Components: Show**

Components that have been hidden using HideFlags are still shown in the Inspector, though they will be grayed-out to indicate that they are hidden.

**Script Field: Hide (Default)**

Don't display a reference to the MonoScript asset of MonoBehaviours and ScriptableObjects as their first field. You can still right click the header of such targets and select "Edit Script" to start editing their MonoScript asset or "Select Script" to select their MonoScript asset in the Project window.

**Script Field: Show**

Display reference to the MonoScript asset of MonoBehaviours and ScriptableObjects as their first field.

This is how the default Inspector window works.

**Field Visibility: Serialized Only (Default)**

Fields that Unity can't serialize are not shown in the inspector, even if public, unless explicitly exposed using attributes like EditorBrowsable,Browsable(true),SerializeField or ShowInInspector. This is similar to how the default Inspector window works.

**Field Visibility: All Public**

All public fields are shown, whether or not Unity can serialize them, unless explicitly hidden with attributes like HideInInspector. Non-public fields are not shown unless explicitly exposed using attributes like EditorBrowsable, Browsable(true),SerializeField or ShowInInspector.

**Property Visibility: Attribute-Exposed Only (Default)**

Only show properties explicitly exposed with Attributes like EditorBrowsable, Browsable(true) or ShowInInspector.

**Property Visibility: Auto-Generated Public**

All public auto-generated properties are shown, unless explicitly hidden with attributes like HideInInspector.

Other properties are not shown unless explicitly exposed with attributes like `EditorBrowsable`, `Browsable(true)` or `ShowInInspector`.

**Property Visibility: All Public**

All public properties are shown, unless explicitly hidden with attributes like `HideInInspector` or `NonSerialized`.

**Method Visibility: Attribute-Exposed Only (Default)**

Only show methods explicitly exposed with attributes like `EditorBrowsable`, `Browsable(true)` or `ShowInInspector`.

**Method Visibility: Context Menu**

All methods with the `ContextMenu` attribute are shown, unless hidden with attributes like `HideInInspector`. This includes static methods.

Other methods are not shown unless explicitly exposed with attributes like `EditorBrowsable`, `Browsable(true)` or `ShowInInspector`.

**Method Visibility: All Public**

All public methods are shown, unless explicitly hidden using attributes like `HideInInspector`. Non-public methods are only shown if exposed using attributes like `EditorBrowsable`, `Browsable(true)` or `ShowInInspector`.

**Preview Area: Minimized**

The preview area is always minimized by default when the inspected targets change. You will always have to manually expand the preview area by clicking it.

**Preview Area: Dynamic (Default)**

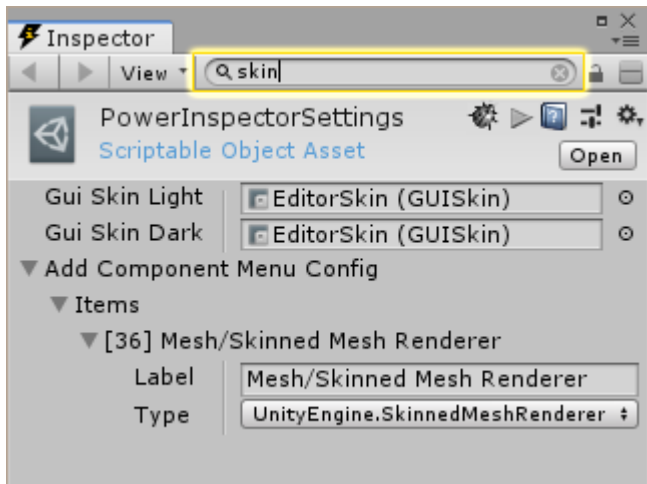
When the inspected targets change, the preview area is minimized most of the time, but expands automatically when a preview offers information that users will likely find useful - such as a visual preview of a texture or a model.

The preview area is minimized most of the time, but automatically expand it when a preview offer information that users will most likely find useful - such as a visual preview of a texture or a model. You can still always manually expand the preview area by clicking it.

**Preview Area: Manual**

The preview area is always shown if inspected targets have any previews. This is similar to how the default Inspector window works.

## Filter Field



The Filter Field can be used to filter the GUI instructions inside a Power Inspector view so that only ones that pass the filtering are shown.

There are several different methods for filtering the instructions, which you can specify by adding a special prefix before the filter text:

Prefix	Filtering Method	Examples
L:	Only show instructions whose label contains the filter.	<i>"l:position", "l:position.x"</i>
T:	Only show instruction whose type name contains the filter.	<i>"t:vector3", "t:system.obj"</i>
V:	Only show instructions whose value's string representation contains the filter.	<i>"v:null", "v:true"</i>
None	Only show instructions whose label, type OR value contains the filter.	<i>"position", "LeftArrow"</i>

### Multi-word Filtering

A filter field input may contain multiple keywords separated by a space character (' '). In this case, GUI instructions are only shown if they pass filtering tests against **all** of the keywords. E.g. "camera depth" or "enum left".

### Collapsed Instructions Searched

When a filter text is given, GUI instructions are queried from inside instructions for all Components, datasets and other collapsible instructions contained within the view, even if they are currently collapsed.

### **Custom Editors Searched**

Filtering will also work with Components and Assets that use Custom Editors. In this case the Custom Editor is temporarily bypassed, and the filtering happens against the public fields and properties of the Unity Object instead.

### **Members And Parents Shown**

When a GUI Instructions passes a search filter, all of its member instructions and parent instructions are also shown.

So e.g. if you were to filter a view with a GameObject target with *“position”*, not would the instructions for the *“position”* field be shown, but also the instructions representing it’s member fields *“x”*, *“y”* and *“z”*, as well as the instructions representing the Transform that contains the field and the GameObject that holds the Transform.

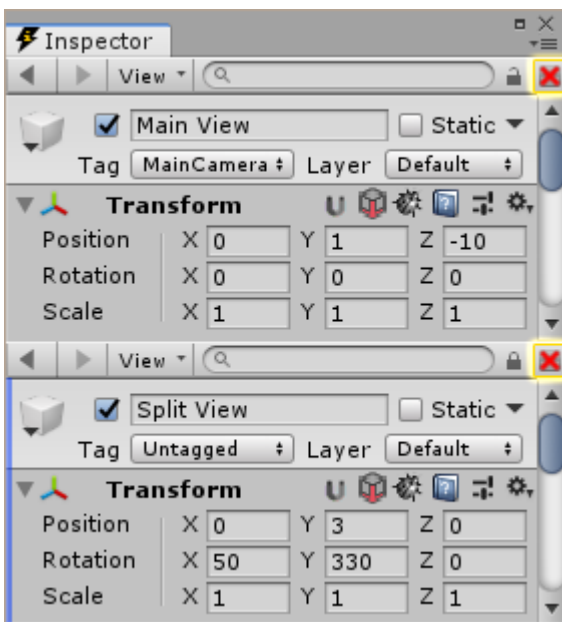


## Split View



The Split View icon found at the right edge of the Power Inspector Toolbar can be used to split the Inspector Drawer into two separate views.

This can be useful for multi-tasking purposes, for example if you want to compare or migrate data between two different targets.



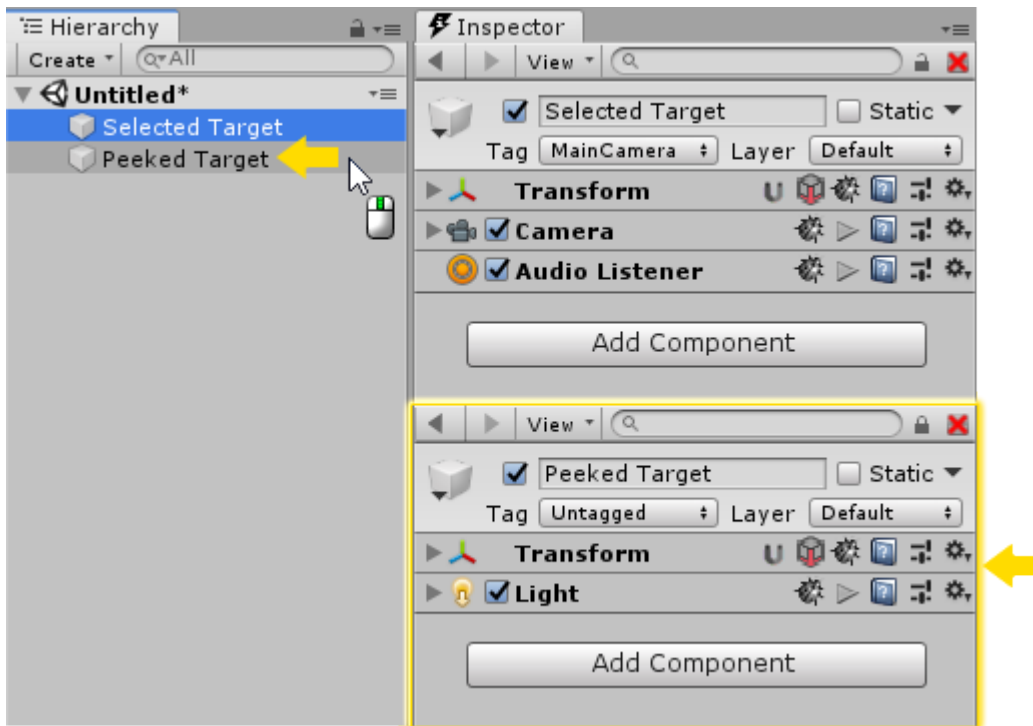
When the Power Inspector view is split, the Split View icon is replaced by a Close View icon, which can be used to close the view in question.

### Keyboard Shortcut

The keyboard shortcut **Ctrl + Space** can also be used to split the view of the selected Inspector Drawer - or close the Split View if one is currently open.

This shortcut key can be configured using the Preferences window.

## Peeking



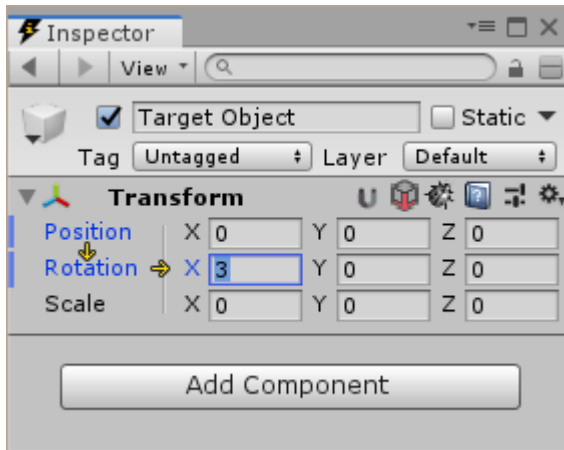
Another way to split the view is via peeking. The easiest way to peek at an Unity Object is by middle-clicking a GUI item representing one.

This works with items in the **Hierarchy** window, the **Project** window and with **Object fields** in the Power Inspector window.

## Keyboard Navigation

Keyboard navigation inside the Power Inspector window has been improved in various ways compared to the Default Inspector.

### Navigation Arrow Key Navigation

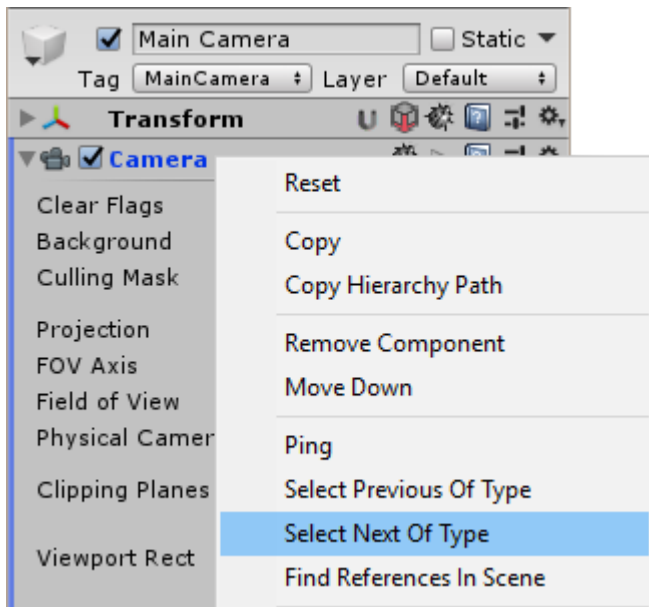


The arrow keys on your keyboard now move focus in a more intuitive manner, in the matching direction. For example, the up arrow moves focus to the GUI instructions that are spatially located above the one that is currently focused.

### Select Previous / Next Component

It is now possible to jump to the previous or next Component on the inspected GameObject.

## Select Previous / Next Of Type



It is now possible to jump to the previous or next Unity Object that is of the same type as the current Target Object.

When the target is a scene object, the next target is fetched from the hierarchy of loaded scenes.

When the target is an asset, the next target is fetched from among the assets in the project.

To select the next Object of a given type, you can:

### (Limited) Multi-Selection Support

Selecting multiple member instructions inside the instructions for collections (such as Arrays and Lists) is now possible.

When multiple instructions are selection, actions targeting the selected instructions - such as ones initiated via the context menu, or using keyboard shortcuts - are applied against all in the selection.

Actions that support multi-selection targeting include:

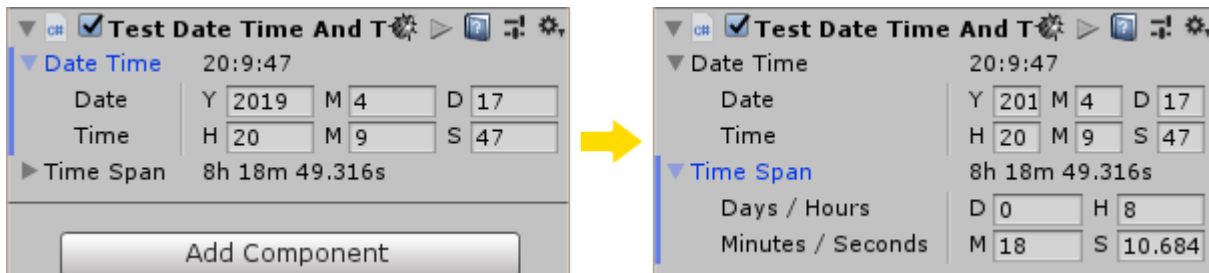
### Improved Focused Field Indicators

With Power Inspector the objective has been to make it as clear as possible to see at a glance which control is currently focused. E.g. in the Default Inspector there is no indicator at all when specific parts of a Component have keyboard focus. Power Inspector tries to fix all of these issues to make for a more seamless experience when navigating the views using the keyboard.

## Dynamic Prefix Column

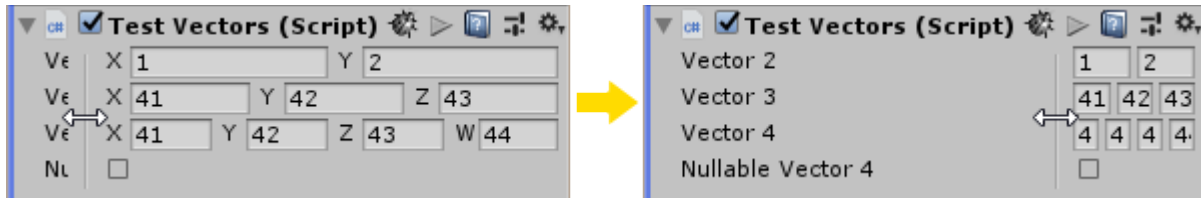
Power Inspector completely changes how the Inspector prefix label column width is determined. No longer is it just a percentage of the total width of the Inspector View, like in the Default Inspector.

### Automatic Width Optimization



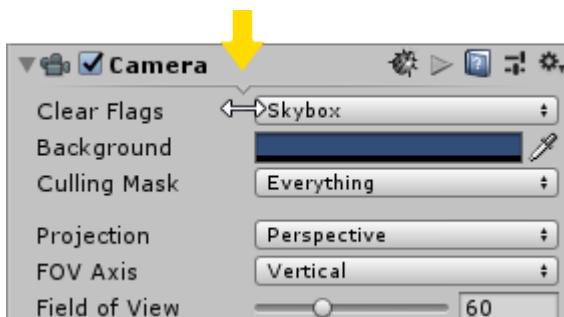
By default, whenever the contents of an Inspector View are changed, the prefix width is automatically optimized to be just wide enough, so as to fully display the labels of all instructions, while maximizing the remaining space available for the right-side controls column. This functionality can be configured in Preferences.

### Manual Width Adjustment



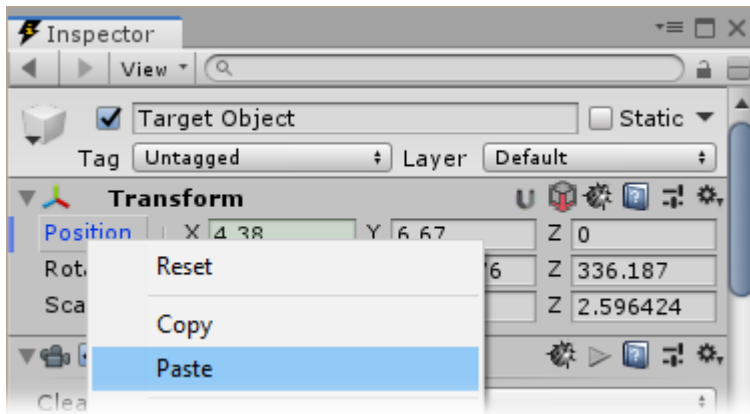
It is also possible to manually adjust the prefix column width by dragging the divider line between the columns. If you double click the divider line, the prefix width is set to an optimal width.

### Manual Width Adjustment In Custom Editors



With Custom Editors manual prefix width adjusting works the same, except instead of dragging a divider line, you drag the division point marker located at the top of the instructions.

## Copy-Paste



### Copy Anything

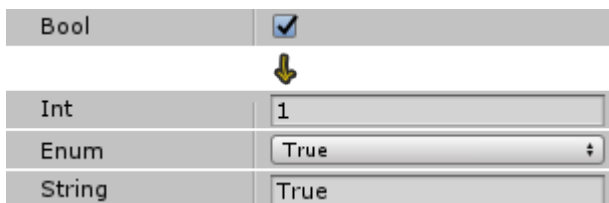
It's possible to copy the value of any GUI instructions via the right-click context menu, or using the keyboard shortcut **Ctrl+C** (**Cmd + C** on macOS).

This not only works with instructions representing simple types, but also datasets, Unity Object references, Components, Assets and even GameObjects.

### Paste Anywhere

A copied value can be pasted onto other GUI instructions via the right-click context menu, or using the keyboard shortcut **Ctrl+V** (**Cmd + V** on macOS).

### Cross-Type Paste

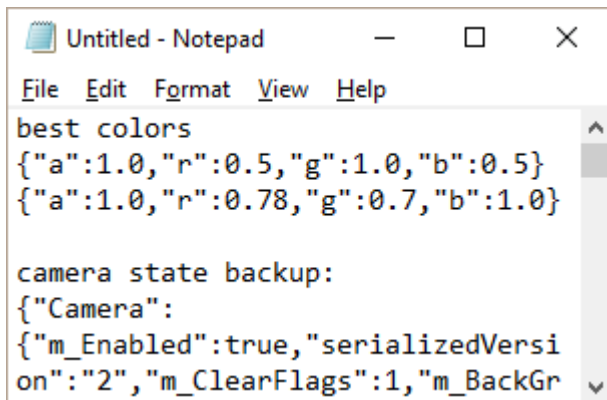


We try to support copy-pasting even between different types whenever possible.

For example you can copy-paste:

- ✓ **Int** value to **float** field.
- ✓ **Enum** name to **text** field.
- ✓ **Vector2** value to **Vector2Int** field.
- ✓ **Quaternion** value to **rotation** (Vector3) field.
- ✓ **Array** value to **List** field.
- ✓ **Asset path** to **Unity Object** field.
- ✓ Value of type **T** to field of type **Nullable<T>**.

## Saved To Clipboard



```
best colors
{"a":1.0,"r":0.5,"g":1.0,"b":0.5}
{"a":1.0,"r":0.78,"g":0.7,"b":1.0}

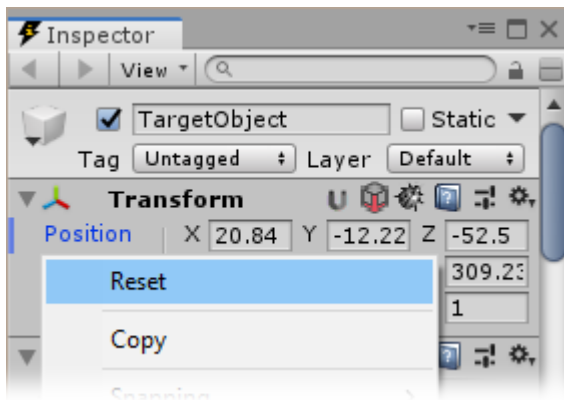
camera state backup:
{"Camera":
{"m_Enabled":true,"serializedVersi
on":"2","m_ClearFlags":1,"m_BackGr
```

The value of copied instructions is serialized into human-readable text and placed on your system clipboard.

This has a some major benefits:

- ◆ Copied values persist even if the Unity Editor is closed (or crashes).
- ◆ Copied values persist even if you switch to another open Unity Editor application. This means that cross-project copy-pasting is fully supported.
- ◆ You can paste copied values into external applications. This enables actions such as:
  - ➔ Quickly backup values to an external text editor.
  - ➔ Send the serialized state of a Component to a co-worker via a chat application.
  - ➔ Paste dialog name to your dialog authoring software.

## Reset

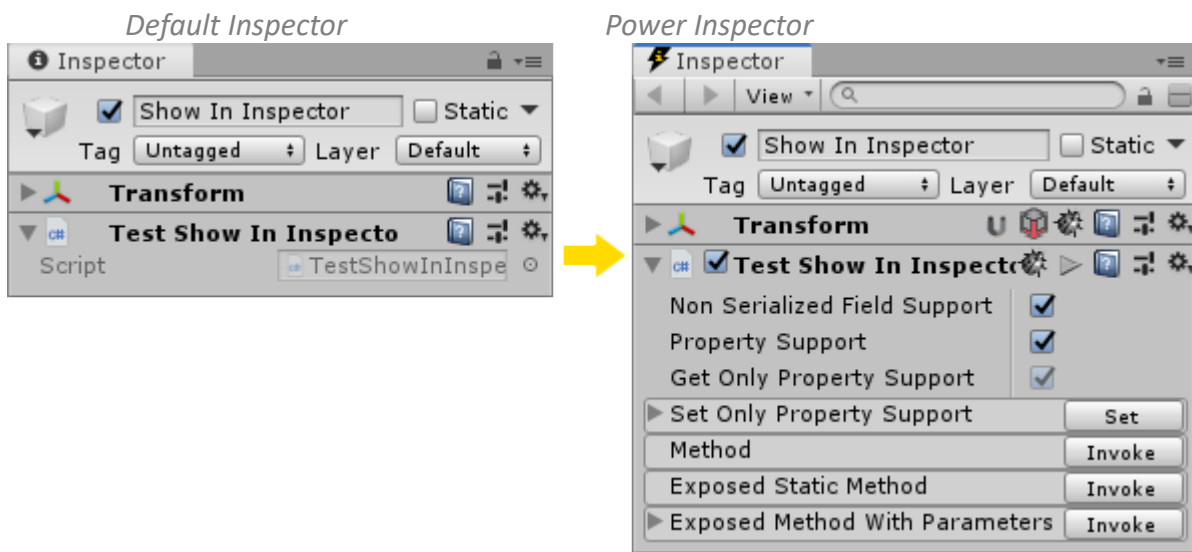


It's possible to reset the value of any GUI instructions to its default state via the right-click context menu, or by pressing the **Backspace** button on your keyboard.

This works not only with instructions representing simple types, but also datasets, Components, Assets and and even GameObjects.



## Display Anything



## Display Any Member

With Power Inspector the objective has been to be able to display anything you can think to throw at it.

Being able to see what data your every field and property has can be invaluable when debugging. And being able to invoke any method on any Component can save a lot of time when testing.

## Members That Power Inspector Can Display

### ◆ Fields

- ✓ Serialized fields.
- ✓ **Non-serialized** fields.
- ✓ Fields with the **readonly** modifier.
- ✓ Fields with the **HideInInspector** attribute (in Debug Mode+ only).
- ✓ **Static** Fields.

### ◆ Properties

- ✓ Auto-implemented properties.
- ✓ Properties with **asymmetric accessor visibility**.
- ✓ Properties with **only** the **get** accessor.
- ✓ Properties with **only** the **set** accessor.

### ◆ Methods

- ✓ Instance methods.
- ✓ Methods with **parameters**.
- ✓ **Static** methods.

### ◆ Indexers

- ✓ Single-parameter indexers.
- ✓ **Multi-parameter** indexers.

## Types That Power Inspector Can Display

Power Inspector can display just about any Type imaginable. Examples of supported types include:

- ✓ Multi-Dimensional Array
- ✓ Jagged Array
- ✓ Dictionary
- ✓ HashSet
- ✓ Type
- ✓ Delegate
- ✓ Interface
- ✓ Nullable
- ✓ System.Object
- ✓ DateTime
- ✓ TimeSpan

*and many more...*

## Note About Serialization

While Power Inspector can handle **displaying** all kinds of exotic data types, be aware that **serializing** that data is outside the scope of Power Inspector. You will still have to manually handle serializing (and deserializing) any data that Unity can't handle. For your data serialization needs you might want to look into things like the BinaryFormatter class, Json.Net or the Odin Serializer.

## Context Menu

In Power Inspector, the header or prefix label of any instructions can be right clicked to open a context menu containing many features not found in the Default Inspector, such as Copy and Paste, and Reset.

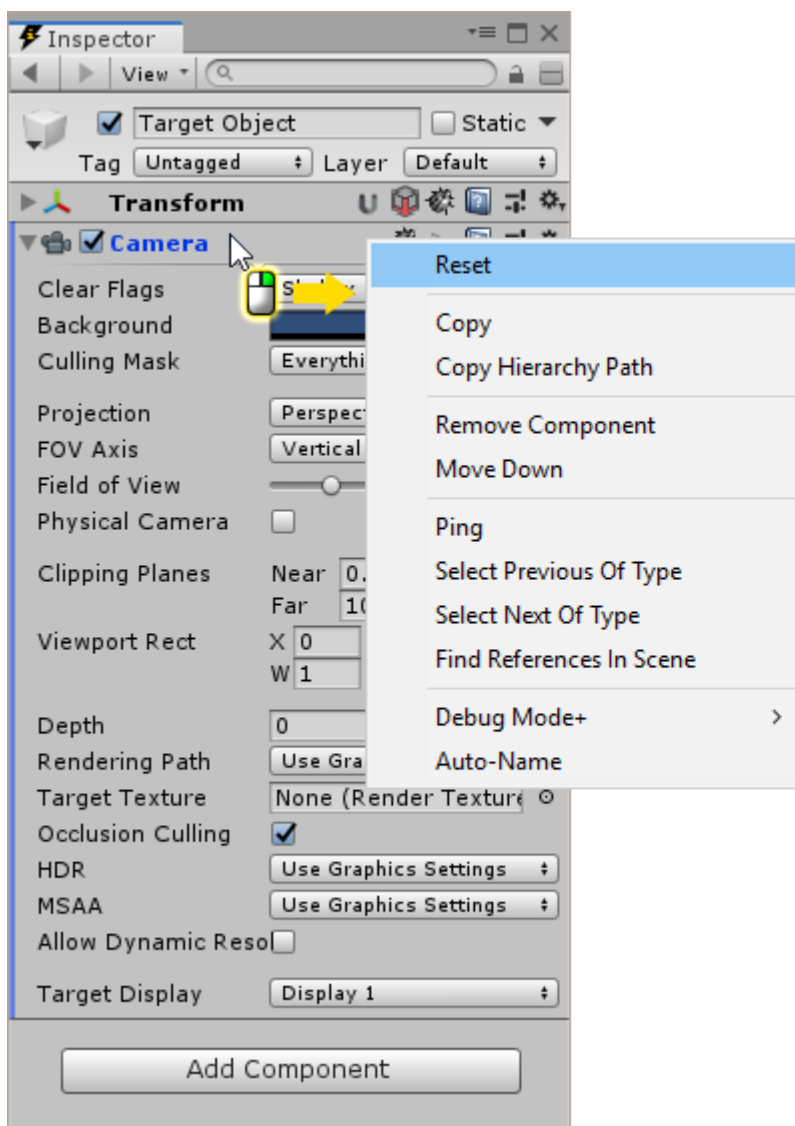
### Opening The Context menu

To open the context menu of a Component, GameObject or asset, **right-click** their header, or press the Menu key on your keyboard with the target selected.

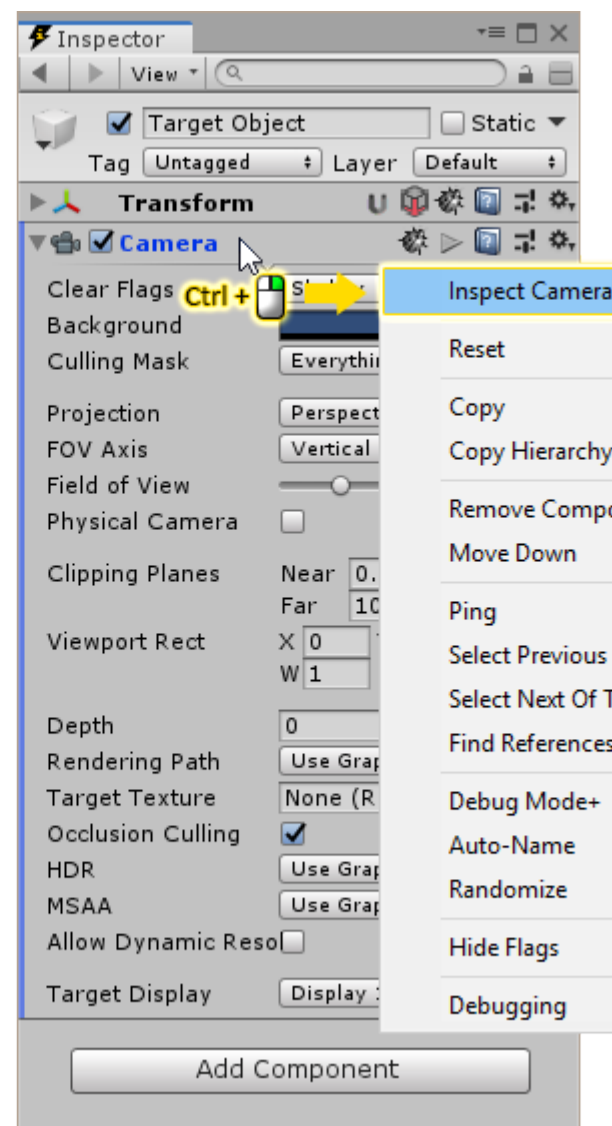
### Opening The Expanded Context Menu

If you hold down the **Ctrl** button (**Cmd** on macOS) while opening the context menu, an expanded context menu with additional entries will be opened.

Some rarely used but still useful features have been hidden away behind the expanded context menu, so that users can find the items they use most often faster in the main context menu.

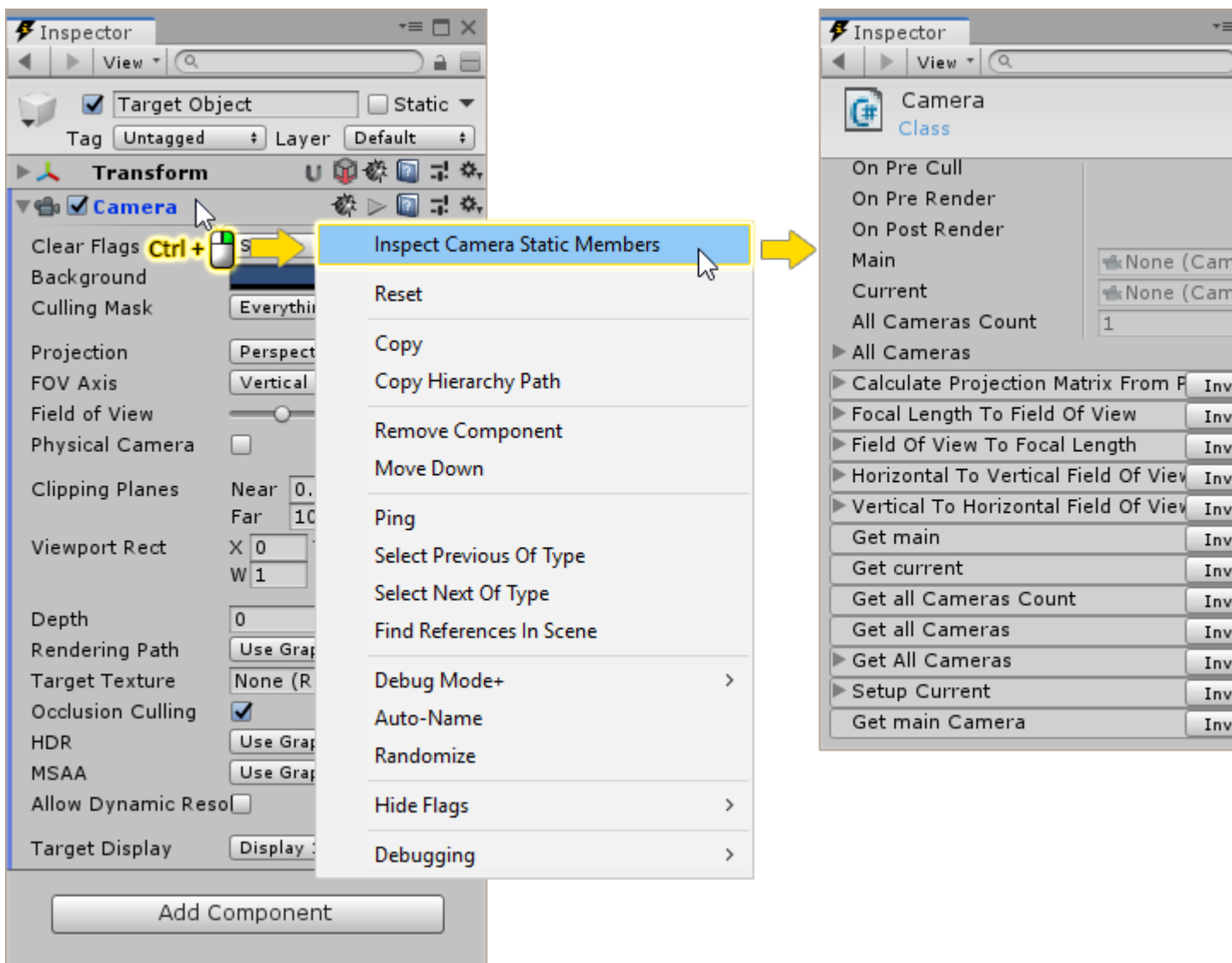


Right-Click opens the Context Menu  
Click opens the Expanded Context Menu



Ctrl / Cmd + Right

## Inspecting Static Members



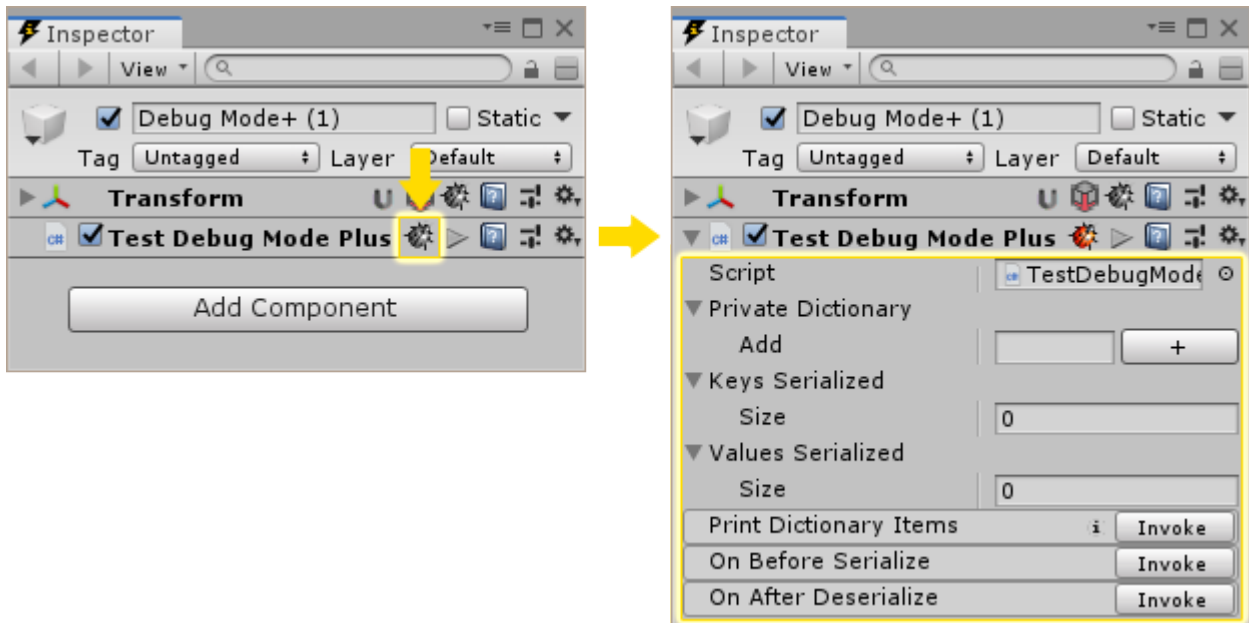
Power Inspector can also be used to view the static members of any inspected Component or asset.

To do this, [open the Expanded Context Menu](#) for the target, and select the menu item **Inspect X Static Members** (where X is replaced by the class name of the target).

### All Public Static Members Shown

By default, all public fields, properties, methods and indexers of the inspected class will be listed. If you want to display the non-public members of the class, you can do it by enabling Debug Mode+.

## Debug Mode+



Power Inspector includes an improved debugging mode called **Debug Mode+**. This mode can be enabled for a single Component or asset via the Debug Mode+ icon found amongst the header buttons, or for the whole Inspector View via the View Menu.

In Debug Mode+ all members of targets are listed (with the except of members that have the Obsolete attribute). This includes:

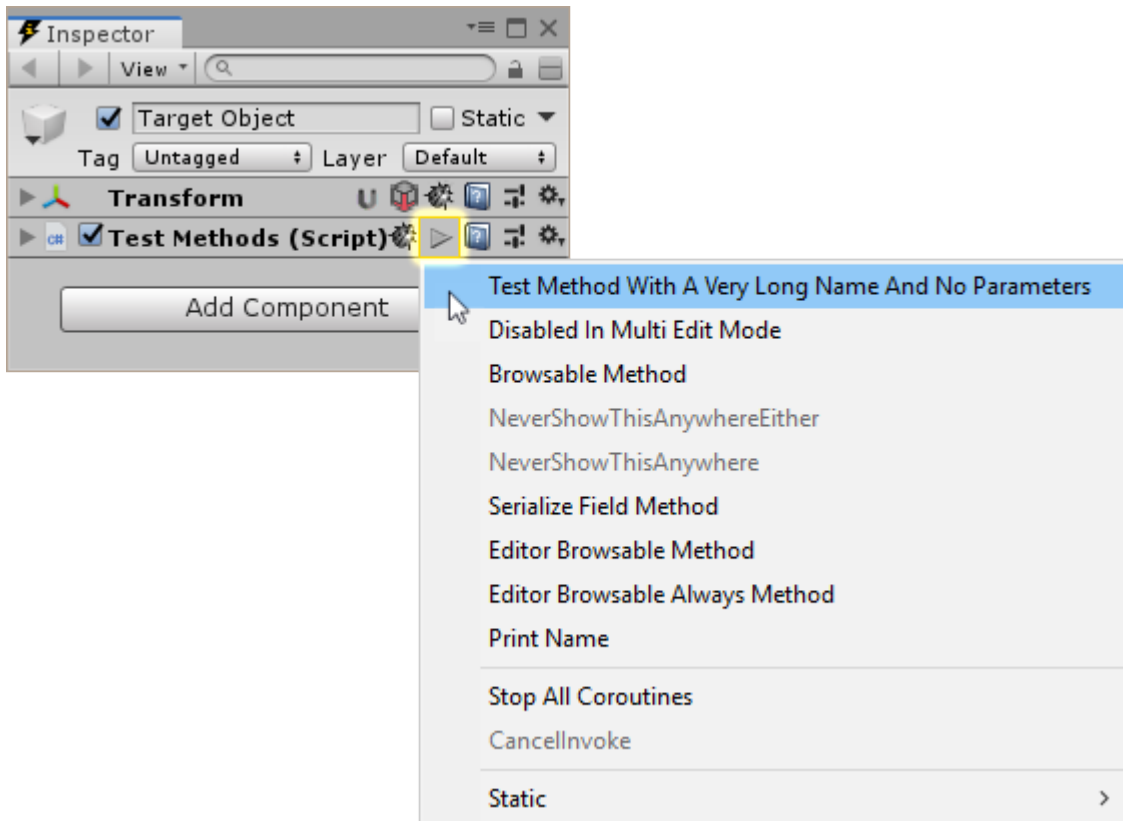
- ✓ **Non-serialized** fields.
- ✓ Fields containing the **HideInInspector** attribute.
- ✓ **Properties**.
- ✓ **Methods**.

Being able to view the full state of a target can be extremely useful when debugging.

### Debugging Static Members

Debug Mode+ also works when inspecting the static members of a target.

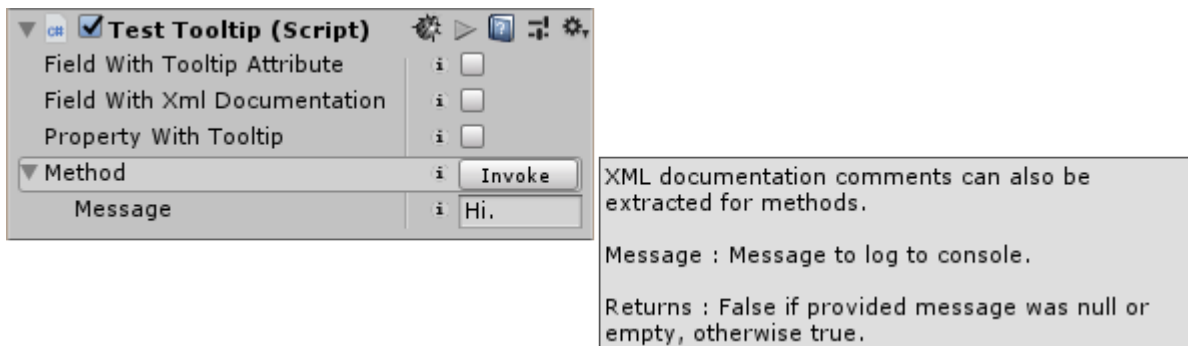
## Quick Invoke Menu



Power Inspector adds a new header button which can be used to quickly invoke methods on Components and Assets shown in the [Inspector View](#).

Which methods are accessible through this button can be customized through [Preferences](#).

## Tooltips



### Hint Icon

The way that tooltips are displayed in the Default Inspector has a couple of issues:

To solve both of these issues, Power Inspector introduces the **hint icon**. All tooltips in Power Inspector are relocated behind a hint icon. This clearly communicates to users when additional information is available, and effectively avoids tooltips popping open when prefix labels are being dragged or right-clicked.

### Tooltip Attribute

Power Inspector supports the Tooltip attribute just like the Default Inspector does. However all tooltips are shown via the hint icon by default (this behavior is customizable via the Preferences).

### XML Documentation Comment Support

Using the Tooltip attribute to generate tooltips has a couple of issues.

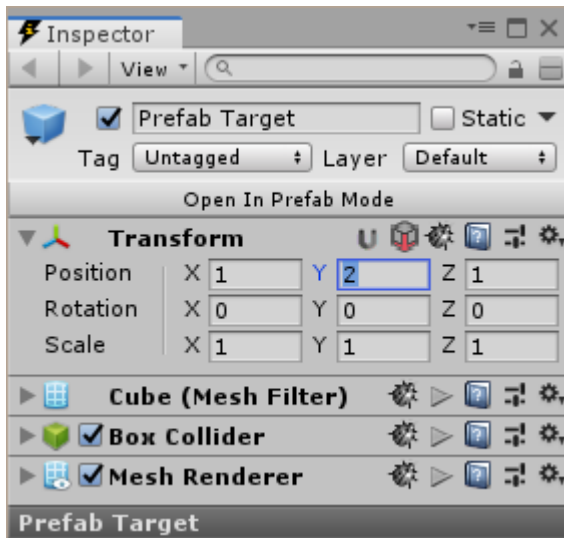
1. Only serialized fields are supported. With Power Inspector supporting the displaying members of all kinds, this can be very limiting.
2. If you also use XML documentation comments to provide tooltips to coders, it can lead to you having to write the same information twice: once for the tooltip attribute, and a second time for the documentation comment.

To solve these issues, Power Inspector can automatically generate tooltips from the XML documentation comments in your code.

This feature works with fields, properties, methods, method parameters and indexers. It supports instance members as well as static members.

XML comments can be read from the XML Documentation of compiled DLL files (when available), or even parsed straight from the MonoScript assets in your Project.

## Prefab Quick Editing



Power Inspector allows quickly editing prefab asset content directly inside the Inspector View, without having to open the prefab in Prefab Mode.

To enable this feature, **Prefab Quick Editing** should be set to **Enabled** in the Preferences.

- ✓ **NOTE:** This feature is supported even in Unity versions older than **2019.1**.



# Attribute Enhancements

## Controlling Member Visibility With Attributes

Power Inspector, using default Data Visibility Preferences, displays fields similarly to the Unity's default Inspector; by default only fields that are serialized by Unity are shown.

To be more specific, in the default Inspector, fields have been exposed according to the following logic:

1. Public fields are shown, provided that the field type is serializable by Unity (class has the `Serializable` attribute and is not generic) and the field does not have the `NonSerialized` attribute.
2. Non-public fields are only shown if they have the `SerializeField` attribute.
3. The `HideInInspector` attribute can be used to hide any field, even if it is serialized by Unity.

There have been a few major omissions in this system however, that Power Inspector takes care of.

## Exposing Non-Serialized Fields

There has been no way to expose non-serialized fields in Unity's default Inspector (outside of writing Custom Editors, which can be very time-consuming). Sometimes you may want to expose a field of a type that Unity can't serialize (such as a Dictionary!), and handle its serialization manually, but the default Inspector has provided no easy way to handle this use case.

### ShowInInspector

The only new attribute that that Power Inspector introduces for customizing how Power Inspector displays your data.

Note that if you have your own attribute named exactly "`ShowInInspector`", that will also work, and you won't need to convert to using our variant.

#### Usage:

1. Add "`using Sisus.Attributes;`" at the beginning of your script file.
2. Add "`[ShowInInspector]`" above any members to have them show up in the Inspector View.

### EditorBrowsable

The `EditorBrowsable` attribute can also be used as an alternative to `ShowInInspector`, if you don't want to add references to attributes that don't come built-in with Unity into your classes.

#### Usage:

1. Add "`using System.ComponentModel;`" at the beginning of your script file.
2. Add "`[EditorBrowsable]`" above any members to have them show up in the Inspector View.

### Browsable

The `Browsable` attribute is yet another alternative to `ShowInInspector`.

#### Usage:

1. Add "`using System.ComponentModel;`" at the beginning of your script file.
2. Add "`[Browsable(true)]`" above any members to have them show up in the Inspector View.

## **HideInInspector**

The HideInInspector attribute can still be used to hide fields like in the Default Inspector.

## **ContextMenu**

If **MethodVisibility** is set to **ContextMenu** in Preferences, then all methods with the **ContextMenu** attribute will be exposed as buttons in the Inspector View.

Additionally, if **InvokeMethodVisibility** is set to **ContextMenu** in Preferences, then the invoke button will only list methods with the **ContextMenu** attribute (unless Debug Mode+ is enabled).

## **NotNull**

Members with the NotNull attribute will be tinted red in the Inspector View if their value is null.

# Preferences

Power Inspector has been designed to be highly customizable, so that users can tweak it to fit their specific needs.

To customize the preferences for Power Inspector follow these steps:

1. Open the Preferences window by using the menu item **Edit > Preferences...**
2. Select the **Power Inspector** listing on the left-side menu.
3. Click the **Edit Preferences** button. A new Power Inspector window displaying preferences for Power Inspector should open.
4. Resize and reposition the window to fit your needs. You may even want to maximize it to make working with it easier (there are a lot of options!)

## List of Preferences

### Data Visibility

#### Show Non Serialized Fields:

- ✓ **False (Default):** Fields that Unity can't serialize are not shown in the inspector, even if public, unless explicitly exposed using attributes like `EditorBrowsable`, `Browsable(true)`, `SerializeField` or `ShowInInspector`. This is similar to how the default Inspector window works.
- ✓ **True:** All public fields are shown, whether or not Unity can serialize them, unless explicitly hidden with attributes like `HideInInspector`. Non-public fields are not shown unless explicitly exposed using attributes like `EditorBrowsable`, `Browsable(true)`, `SerializeField` or `ShowInInspector`.

#### Show Properties:

- ✓ **Attribute-Exposed Only (Default):** Only show properties explicitly exposed with Attributes like `EditorBrowsable`, `Browsable(true)` or `ShowInInspector`.
- ✓ **Auto-Generated Public** - All public auto-generated properties are shown, unless explicitly hidden with attributes like `HideInInspector`. Other properties are not shown unless explicitly exposed with attributes like `EditorBrowsable`, `Browsable(true)` or `ShowInInspector`.
- ✓ **All Public** - All public properties are shown, unless explicitly hidden with attributes like `HideInInspector` or `NonSerialized`.

#### Show Methods:

- ✓ **Attribute-Exposed Only (Default):** Only show methods explicitly exposed with attributes like `EditorBrowsable`, `Browsable(true)` or `ShowInInspector`.
- ✓ **Context Menu** - All methods with the `ContextMenu` attribute are shown, unless hidden with attributes like `HideInInspector`. This includes static methods. Other methods are not shown unless explicitly exposed with attributes like `EditorBrowsable`, `Browsable(true)` or `ShowInInspector`.
- ✓ **All Public** - All public properties are shown, unless explicitly hidden with attributes like `HideInInspector` or `NonSerialized`.

## **Data Visualization**

### **MemberDisplayOrder:**

- ✓ The order in which the Inspector should list fields, properties and methods. The default order is as follows: Fields, Properties, Methods.

### **Place Tooltips Behind Icons**

- ✓ If true, a hint icon will be placed next to members that have a tooltip as a clear indicator to end-users. The tooltip text will now also only be shown when this hint icon is mouseovered (and not e.g. when dragging the prefix of a float field).

# Extending Power Inspector

## Development Mode

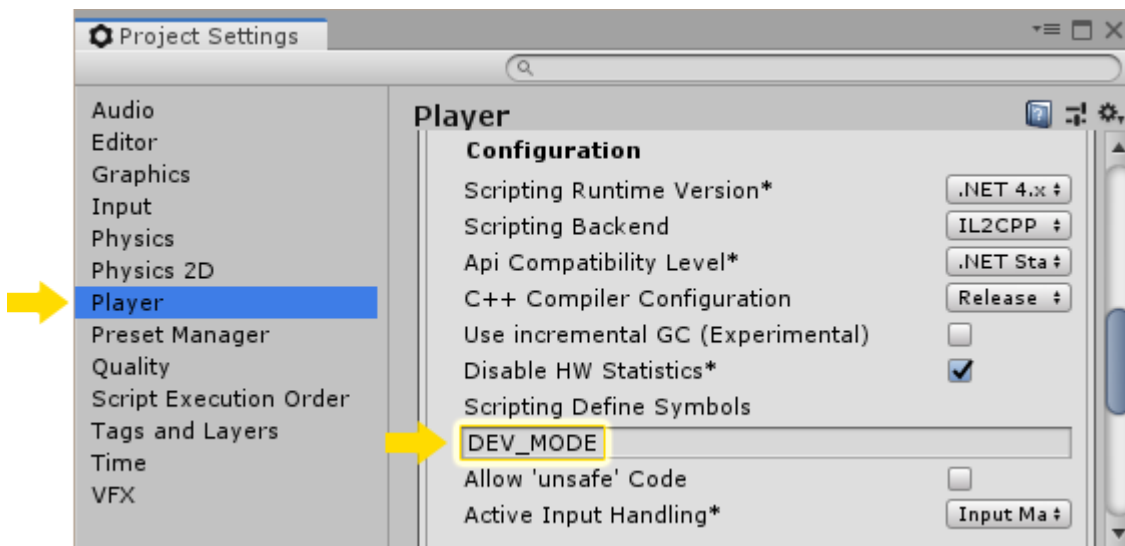
Many assertions, warning and error messages and debug logging options in Power Inspector are stripped away from the code unless a special “development mode” has been enabled.

In normal usage you do not want this mode active, to ensure that Power Inspector runs as fast as possible with as little garbage generation as possible, and to avoid messages cluttering up your Console window.

If however you’re writing new GUI instructions for your own data types, modifying existing ones, or perhaps are trying to find the source of an issue in Power Inspector, it might be useful to enable this mode.

### Enabling Development Mode

To enable development mode, follow these steps:

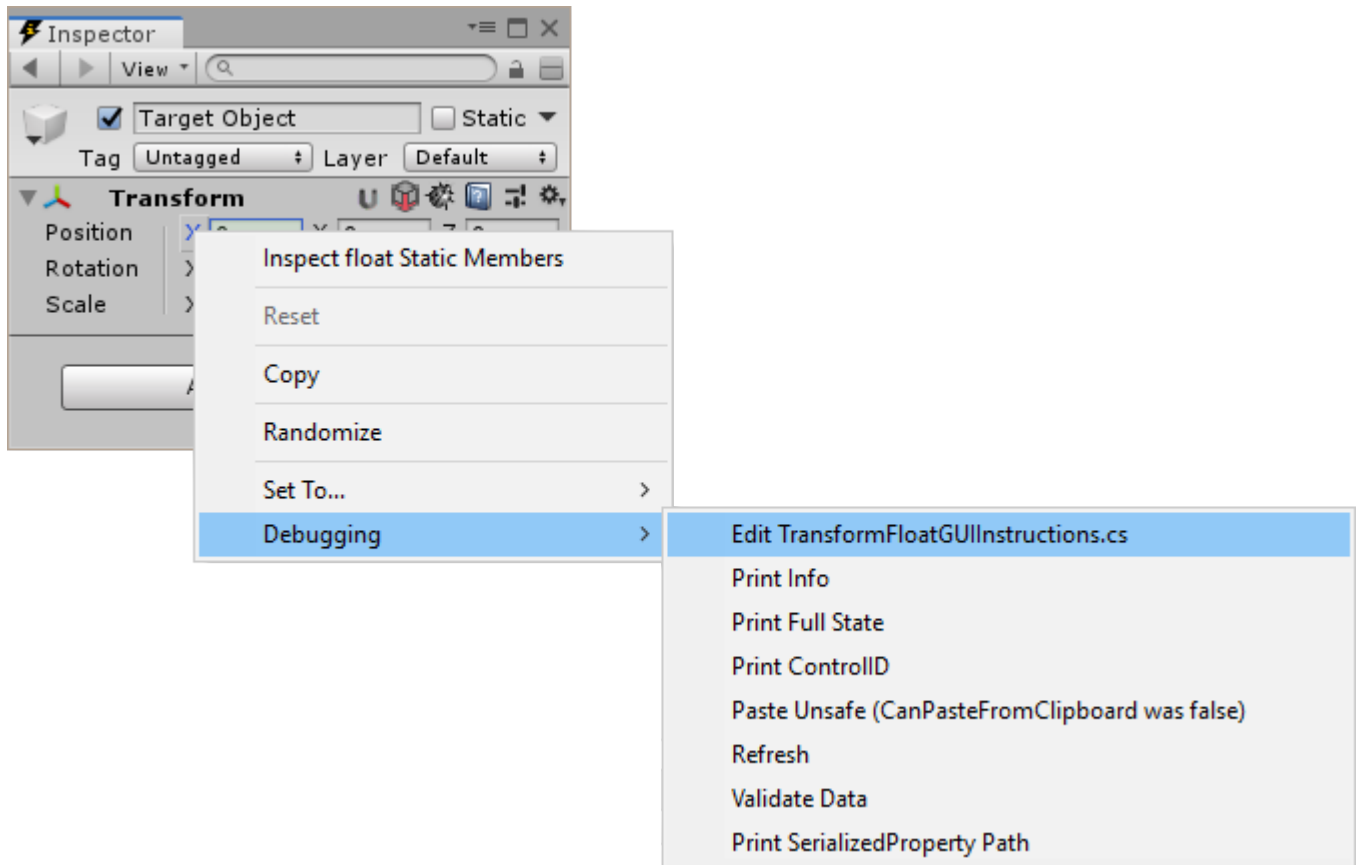


### Disabling Development Mode

To disable development mode, just remove **DEV\_MODE** from the **Scripting Define Symbols** list.

## Extended Context Menu

When development mode has been enabled, if you right click a GUI Instructions while holding down **ctrl** key (**Cmd** key on macOS), a number of additional menu items will be available behind the **Debugging** group. The first item, **Debugging/Edit NameOfInstructions.cs**, can be particularly useful if you want to figure out which GUI Instructions class is responsible for drawing a certain part of the Inspector View, or want to edit the class in question.



# List of Important GUI Instructions

Below is a list of some of the most important GUI instructions to know, if you are interested in understanding how Power Inspector works behind the scenes, or if you're interested in creating new GUI instructions for your own data types, or modifying existing GUI instructions.

## Top-Level

- **GUIInstructionsGroup** - The top-most instructions of any Inspector View, representing the inspected Target Objects as a whole.

## Core Classes

- **BaseGUIInstructions** - Abstract base class from which all instructions inherit.
- **ParentGUIInstructions** - All instructions that have members and never represent a field inherit from this abstract class.
- **ParentFieldGUIInstructions** - All instructions that have member instructions and can represent a field or a property inherit from this abstract class.
- **UnityObjectGUIInstructions** - All instructions representing Unity Objects inherit from this class, except for GameObjectGUIInstructions.

## GameObject

- **GameObjectGUIInstructions** - Represents all GameObject targets, both ones in a scene and prefab assets.

## Component

- **CustomEditorComponentGUIInstructions** - Instructions representing a Component where their body is drawn by a Custom Editor.
- **ComponentGUIInstructions** - Instructions representing a Component where their body is drawn by nested GUI Instructions.

## Asset

- **CustomEditorAssetGUIInstructions** - Instructions representing an asset type Target Object where their body is drawn by a Custom Editor.
- **AssetGUIInstructions** - Instructions representing an asset type Target Object where their body is drawn by nested GUI Instructions.
- **FormattedTextAssetGUIInstructions** - Instructions representing a text asset that can have custom syntax formatting logic attached to it.
- **MonoScriptGUIInstructions** - Instructions representing a MonoScript asset.

## Data Set

- **DataSetGUIInstructions** - A generic handler that can represent any data set that is not a Unity Object, containing any number of member fields, properties and methods.
- **TextGUIInstructions** - Instructions representing a string value.

## Simple

- **ToggleGUIInstructions** - Instructions representing a boolean value.
- **IntGUIInstructions** - Instructions representing an int value.
- **FloatGUIInstructions** - Instructions representing a float value.
- **CharGUIInstructions** - Instructions representing a character value.
- **DoubleGUIInstructions** - Instructions representing a double value.
- **LongGUIInstructions** - Instructions representing a long value.
- **ByteGUIInstructions** - Instructions representing a byte value.
- **SByteGUIInstructions** - Instructions representing a sbyte value.
- **DecimalGUIInstructions** - Instructions representing a decimal value.
- **ShortGUIInstructions** - Instructions representing a decimal value.
- **UIntGUIInstructions** - Instructions representing an uint value.

## Decorator Drawer

- **DecoratorDrawerGUIInstructions** - Instructions representing a PropertyAttribute using a DecoratorDrawer.
- **HeaderGUIInstructions** - Instructions handling the drawing of a header defined by the Header attribute.

## Property Drawer

- **PropertyDrawerGUIInstructions** - Instructions representing a field with a PropertyAttribute using a PropertyDrawer.
- **IntRangeGUIInstructions** - Instructions handling the drawing of an int field containing the Range attribute.
- **FloatRangeGUIInstructions** - Instructions handling the drawing of an int field containing the Range attribute.

## Transform

- **TransformGUIInstructions** - Instructions handling the drawing of Transform components of targets.
- **PositionGUIInstructions** - Instructions handling the drawing of position properties of Transform components.
- **RotationGUIInstructions** - Instructions handling the drawing of rotation properties of Transform components.
- **ScaleGUIInstructions** - Instructions handling the drawing of scale properties of Transform components.
- **TransformFloatGUIInstructions** - Instructions handling the drawing of float values contained inside the position, rotation and scale properties of Transform components.



# Terminology

## Inspector

The term “Inspector” is an ambiguous term that can refer to an Inspector Drawer, an Inspector View, or the combination of the two. In some instances it can also encompass the Default Inspector.

## Default Inspector

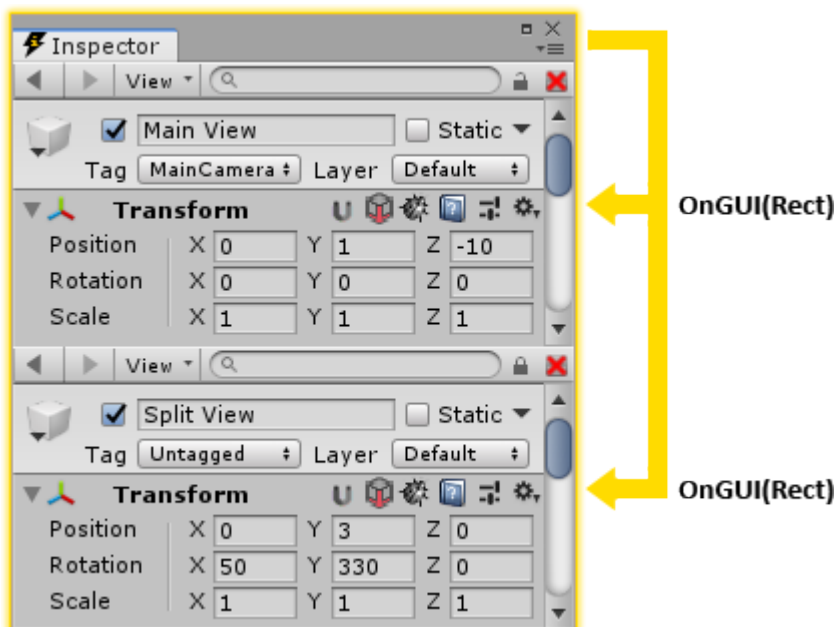
“Default Inspector” refers to the built-in Inspector window that comes bundled with the Unity editor, and which the Power Inspector window supersedes.

## Inspector Drawer

An Inspector Drawer is responsible for passing Unity’s Event Functions (such as OnGUI) to the Inspector Views it contains.

The drawer also determines where and at what size Inspector views are drawn in the screen.

An Inspector Drawer is usually an EditorWindow, but also can be e.g. a MonoBehaviour.



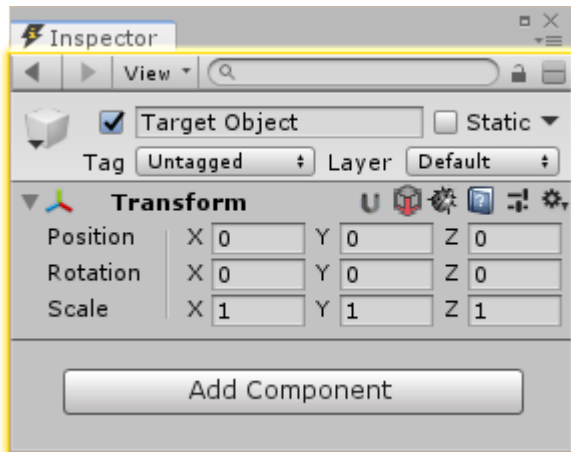
*The Power Inspector window is an Inspector Drawer.*

## Inspector View

Each Inspector Drawer consists of at least one Inspector View (the Main View), and sometimes two (adding the Split View).

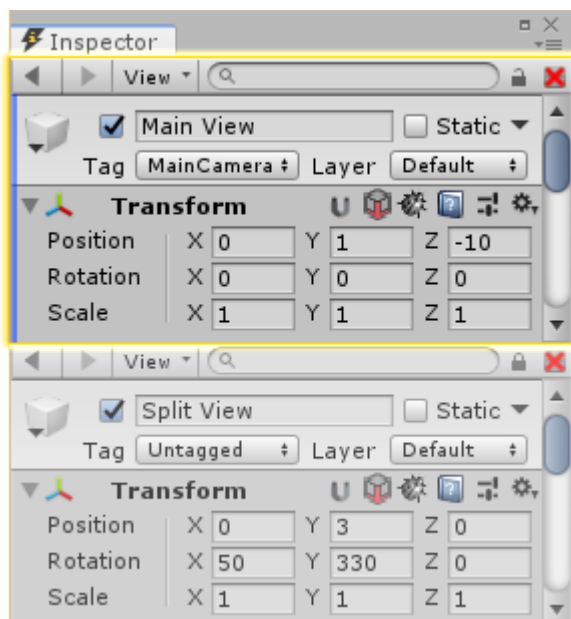
The views of an Inspector Drawer make up everything that is drawn inside it.

A view in the Power Inspector window consist of the Toolbar, the Viewport and the Preview Area.



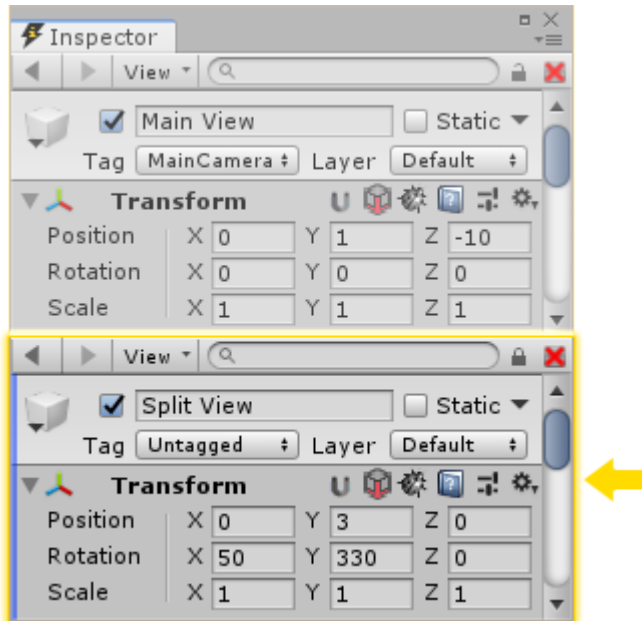
## Main View

The top-most View of an Inspector Drawer is called its “Main View”. When a Power Inspector window has only one view open, that is the Main View.



## Split View

When an Inspector Drawer has been split into two separate Views (see Peeking), the bottom View is called the “Split View”.



For more information refer to the Split View feature page.

## Peeking

“Peeking” means the act of splitting an Inspector Drawer into two separate Views, and setting an Unity Object - the target of the Peeking - as the Target Object of the bottom view, which is called the Split View.

## Unity Object

The term “Unity Object,” or sometimes just “Object” (always capitalized), is used to refer to instances of class inheriting from the Object class in the UnityEngine namespace.

The most common examples of Unity Objects are Components, GameObjects and ScriptableObjects.

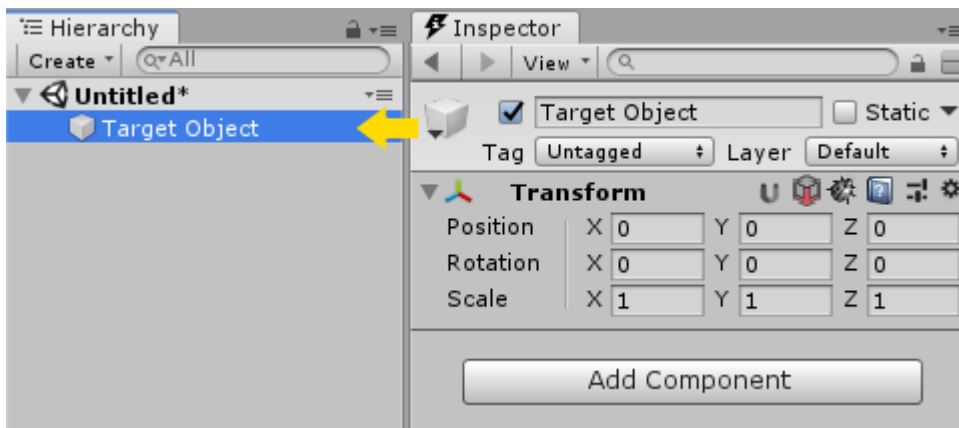
The main purpose of Inspectors is to allow the viewing and editing of data contained within Unity Objects.



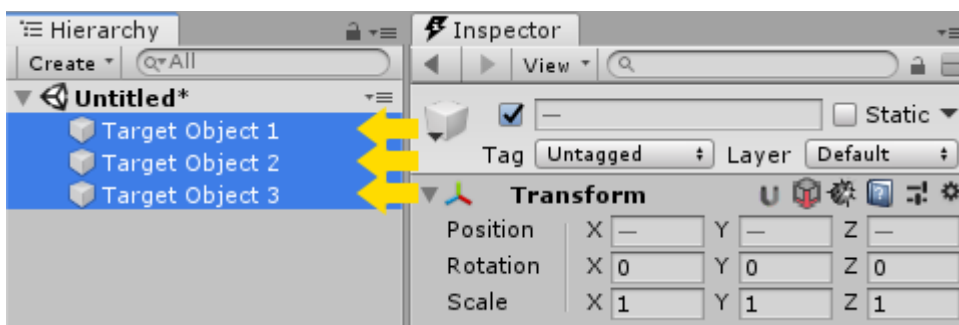
## Target Object

The term “Target Object”, or sometimes just “target”, is used to refer to the Unity Object that is currently being inspected.

The target of GUI Instructions means the Unity Object that the GUI Instructions represent, or the Unity Object that contains the field, property or method that the GUI Instructions represent.

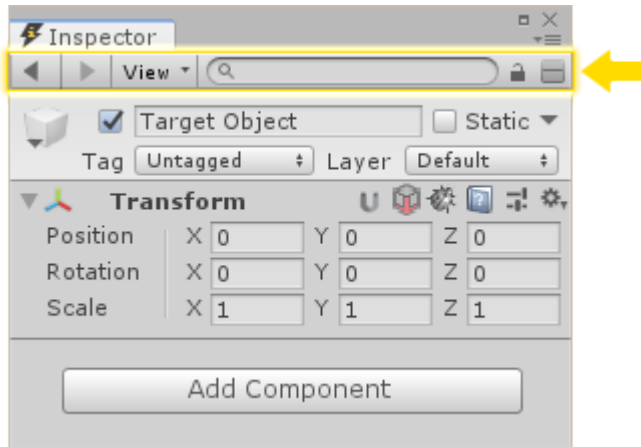


When multiple Unity Objects are inspected simultaneously in Merged Multi-Edit mode, GUI Instructions have multiple Target Objects.



## Toolbar

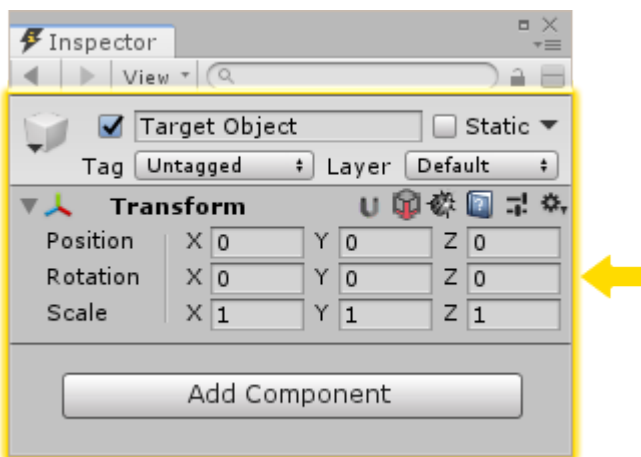
When the term Toolbar is used in this documentation, it refers specifically to the one located at the top of an Inspector View.



## Viewport

The Viewport refers to the main body of an Inspector View, which includes everything except for its Toolbar and the Preview Area.

The Viewport lists all the GUI Instructions for the Target Object of the Inspector View.



## Preview Area

Preview Area refers to the bottom section of an Inspector View, containing possible auxiliary information about the Target Object, such as the asset labels of an asset, or a visual preview of a model.

## **GUI Instructions**

GUI instructions, sometimes referred to as just “instructions”, are responsible for drawing and handling the user-interaction logic of GUI elements inside the view of a Power Inspector. They can be nested inside one another. For example there could be a GUI instruction representing a GameObject, which contains multiple GUI Instructions representing its Components, which contain instructions representing their fields, properties and methods.

## **Members**

GUI instructions nested inside other GUI instructions are referred to as the members of the latter.

In certain context this same term can also be used as an umbrella term to refer to the fields, properties, indexers and methods of a target.

## **Parent**

When GUI instructions are nested inside other GUI instructions, the latter are referred to as the parent of the prior.

E.g. the instructions representing a Vector3 field might contain three instructions representing float fields.