# Documentation

## Console Command & Monitoring System


**Author:** Jonathan Lang


**Unity-Version:** 2019.4.10f1
**Version:** WIP
**Date:** 2021.02.06

# Content

# Introduction . . . . . 1.0

This project / bundle contains two autonomous systems and some fundamental utility assets. First the console / command system which provides functionality to access static methods, properties, and fields during runtime via console input. The console asset is customizable and provides a simple way to create versatile and flexible commands with a few intuitive attributes. The second system is a module based monitoring system that provides an easy way to verify and review important systems and values during runtime. Both systems make use of some universally applicable utility classes.

## Disclaimer . . . . . 1.1

This asset is not the work of a professional. I am a student, and I am planning to use and improve the systems continuous without providing regular updates. If you want a production ready amount of polishing and support, you should use alternatives. However, if you would like to use, improve, or build on top of the systems feel free to do so. My only requirement would be that you provide an appropriate way to credit me. (See license for more information). Also feel free to contact me if you have any questions.

Especially the contents of the console command system were partly inspired by a tutorial from the channel Dapper Dino on YouTube. ( https://www.youtube.com/watch?v=usShGWFLvUk ) link to the video. Although his approach is essentially different from this, his tutorial helped me to build a fist prototype and gave me a good start to get a feel for the basic concept.
I would also like to provide a link to an alternative (production ready) asset: QFSWs “Quantum Console” ( https://assetstore.unity.com/packages/tools/utilities/quantum-console-128881 ) This asset is a polished version of my approach.

There are some things I would like to change / improve in the future that I would like to name beforehand. First, I would like to reduce the usage of singletons. I would also like to rework the canvas elements of both assets for better performance, customizability, and clarity. I am going to use both systems in an upcoming multiplayer project (most likely using Mirror) and will use this opportunity to add networking compatibility to the project.

## License . . . . . 1.2



## Contact & Feedback . . . . . 1.3

I appreciate feedback but please remember that this project and documentation are a study. Just for context: At the time of writing this I have less than 2 years of experience with c#, programming in general and Unity.
Mail: WIP

# Console . . . . . 2.0

The console / command system provides functionality to access static methods, properties, and fields during runtime via console input. The console asset is customizable and provides a simple way to create versatile and flexible commands with a few intuitive attributes. Its intended use is to provide a debugging tool with an easy interface to invoke certain methods and get or set the values of specific member during runtime.

## Quick Start . . . . . 2.1

If you do not have the console GameObject in your scene use `(GameObject => Ganymed => Console)` to instantiate the console prefab automatically. Use `'#'` to activate / deactivate the console and `'/'` as a prefix to invoke a command. `"/commands"` should provide you with a list of available console commands.
The console requires a configuration file. If you do not have a configuration file you can create one using: `(Rightclick => Create => Console => Configuration).` To select an active configuration file, drag it into the configuration slot on the console component in the console prefab.

## User Input . . . . . 2.2

You can use any input system you like. By default, the input is handled by the Console-Input-Standalone component on the consoles GameObject. This system can be replaced by any other input system. The console implements the IConsoleInput interface. This interface provides access to input relevant methods.

Note that the input field is a TextMeshPro Input field and not a pure vanilla input field.

## Console.Log . . . . . 2.2

The console implements two static methods to that will take in an object, convert it, and then output it in the console (strings will skip the conversion). The `Console.LogRaw()` method will log the message without any formatting. The `Console.Log()` method offers multiple overloads and optional parameter for additional formatting options. Both methods are good for logging simple methods / formatting.

`Console.LogRaw()`

| Index | Type and Name | Description (Main Signature) |
|---|---|---|
| 0 | `object message` | The message that will be logged. |

`Console.Log()`

| Index | Type and Name | Description (Main Signature) |
|---|---|---|
| 0 | `string message` | The message that will be logged. |
| 1 | `Color? color` | Set the text color. If null, the default color will be used. (declared in configuration) |
| 2 | `int? lineHeight` | Set the line height of the message. If null, the default line height will be used. (declared in configuration) |
| 3 | `LogOptions options` | Bitmask instructions with additional options. You can find the enum @ (Ganymed.Console.LogOptions) |

```
private const LogOptions logOptions = LogOptions.None;

public static void Log(object message, LogOptions options = logOptions)
    => Log(message?.ToString(), null, null, options);

public static void Log(object message, int lineHeight, LogOptions options = logOptions)
    => Log(message?.ToString(), null, lineHeight, options);

public static void Log(object message, Color? color, LogOptions options = logOptions)
    => Log(message?.ToString(), color, null, options);


public static void Log(string message, LogOptions options = logOptions)
    => Log(message, null, defaultLineHeight, options);

public static void Log(string message, int lineHeight, LogOptions options = logOptions)
    => Log(message, null, lineHeight, options);

public static void Log(string message, Color? color, LogOptions options = logOptions)
    => Log(message, color, null, options);
```

| | | |
|---|---|---|
| `LogOptions.None` | No options but previous formatting will be removed. | 0 |
| `LogOptions.IgnoreFormatting` | No previous formatting will not be removed. | 1 |
| `LogOptions.DontBreak` | No line break will be used to separate from previous output. | 2 |
| `LogOptions.IsInput` | Format the message like an input. | 4 |
| `LogOptions.EndLine` | Add a break after the message. | 8 |
| `LogOptions.Tab` | Add a 4px indent to the message. | 16 |
| `LogOptions.Cross` | ~~The message will be crossed out.~~ | 32 |
| `LogOptions.Bold` | **The message will be bold.** | 64 |

## Formatting . . . . . 2.2

If you require more versatile formatting, you can use the `Transmission` class. This class offers methods that will provide advanced formatting options. Transmissions work according to the following pattern:

1. Start a transmission using `Transmission.Start()`
2. Add lines, breaks etc. to the transmission using any `Transmission.AddXXX()`
3. Release the transmission using `Transmission.Release()` or `Transmission.ReleaseAsync()`

`Transmission.Start()` will tell the class that a new transmission has been started. This method must be invoked before any message can be added.

| Index | Type and Name | Description |
|---|---|---|
| 0 | `TransmissonOptions options = None` | Arguments that will apply to every message of the transmission. |
| 1 | `object sender = null` | The object that sends the transmission. |

`Transmission.AddLine()` will add a new line to the transmission. Lines can contain multiple columns that are passed one after the other.

| Index | Type and Name | Description |
|---|---|---|
| 0 | `params object[] messages` | `object` array containing the content of each individual column. |
| 0 | `params MessageFormat[] messages` | `MessageFormat` array containing the content of each individual column with individual formatting options. |

`MessageFormat` is a custom struct that you can use instead of an `object` if you want to add additional formatting options for an individual column of a line. The first signature (`object[]`) will try to cast each object to a `MessageFormat` so you can use a combination of both signatures.

| Index | Type and Name | Description |
|---|---|---|
| 0 | `object message` | The message contained in the struct. |
| 1 | `Color color` | Optional text color. |
| 1 / 2 | `MessageOptions options = None` | Bitmask instructions with additional formatting options. |

| | | |
|---|---|---|
| `MessageOptions.None` | No formatting options are applied. | 0 |
| `MessageOptions.Bold` | **Content is formatted in bold** | 1 |
| `MessageOptions.Italics` | *Content is formatted in italics* | 2 |
| `MessageOptions.Strike` | ~~Content is crossed out.~~ | 4 |
| `MessageOptions.Underline` | Content is underlined | 8 |
| `MessageOptions.UpperCase` | CONTENT IS UPPERCASE | 16 |
| `MessageOptions.LowerCase` | content is lowercase | 32 |
| `MessageOptions.Smallcaps` | Content Is Smallcaps | 64 |
| `MessageOptions.Brackets` | [Content is surrounded in brackets] | 128 |

`Transmission.AddBreak()` will add a new break after the last transmitted line.

| Index | Type and Name | Description |
|---|---|---|
| 0 | `int? lineHeight = null` | Add a break. If null, the default break line height of the console configuration will be used. |

`Transmission.AddTitle()` is an easy way to format a line to look like a title.

| Index | Type and Name | Description |
|---|---|---|
| 0 | `string title` | The text that will be displayed as a title. |
| 1 | `TitlePreset preset = TitlePreset.Main` | Formatting preset. (Main Title or Subheading) |

`Transmission.Release()` will compile, format, and then log the previously sent messages.

`Transmission.ReleaseAsync()` will compile, format, and then log the previously sent messages asynchronous. Compiling and formatting of the sent messages will be handled by another task. When finished an optional callback is invoked.

| Index | Type and Name | Description |
|---|---|---|
| 0 | `Action callback = null` | Optional callback if the transmission is released asynchronous. |

You can use the **`TransmissionExample`** script at `Assets/Ganymed/Examples` for examples to try out yourself.

## Configuration . . . . . 2.3

If you want to know how to setup the configuration file of the console see: 2.1 Setup. The configuration file is an easy way to customize settings for commands, performance, visuals etc. You can receive information regarding the configuration and its settings during runtime by using the command "/configuration".

| Setting | Type | Description |
|---|---|---|
| Enabled | bool | Is the console enabled. |
| Active | bool | Is the console active. |
| Command Prefix | char | The prefix required to access commands. |
| Info Operator | char | Adding this character at the end of a command input will log additional information about parameter, signature etc. instead of executing the command. example: "/configuration?" |
| Allow Command Pre Processing | bool | When enabled, the command processor can process inputs before they are entered. This will enable the console to make suggestions and enable the equivalent of syntax highlighting. |
| Log Commands Loaded On Start | bool | when enabled, additional information about loaded commands will be logged at the start of the game. |
| Allow Numeric Bool Processing | bool | when enabled numbers can be used as input arguments for Boolean parameter. 1 = true / 0 = false. formula:(value = argument > 0? True : False) |
| Activate Console On Start | bool | When enabled, the console will be activated automatically at the start of the game. |
| Enable Cursor On Activation | bool | When enabled, the cursor will be activated, and its lock state will be set to confined / none (depending on its original state) when activating / opening the console. |
| Log Configuration On Start | bool | When enabled, the configuration will be logged at the start of the game. |
| Log Time On Input | bool | When enabled, the current time will be added as a prefix to every log. |
| Input Cache Size | byte | The number of past inputs that are cached. Previous inputs cam be selected by using the arrow keys. Identical inputs will not be cached multiple times. |
| Bind Consoles | bool | Links the unity console with this console. |
| Allowed Unity Messages | enum | (if consoles are linked) Select what types of unity logs can be received. |
| Log Stack Trace On | enum | Select what types of unity logs will show their stack trace. |
| Allow Shader And Animations | bool | When enabled small animations like min max ease as well as shaders like the blur background effect are allowed. Deactivate for performance improvements. Shaders are not optimized. |
| Allow Shader | bool | Enable / disable shader |
| Allow Animations | bool | Enable / disable animations |
| Show Rich Text | bool | Use for debugging. If enabled, the raw rich text will be shown. |
| Input Font Size | int | The font size of the input field. |
| Font Size | int | The font size of the console. |
| Break Line Height | int | The default line height after a break. |
| Default Line Height | int | The default line height of the console. |
| Console Color | Color | Colors of the console (background etc.) |
| Text Color | Color | General colors that are related to the text of the console. |
| Validation Color | Color | Colors in which the text of the input field will be displayed to communicate information about the validation of the input. |
| Unity Console Color | Color | Colors in which unity console logs will be displayed. |
| Custom Color | Color | Custom colors that can be used by the user. |

## Console Commands . . . . . 3.0

Every static method can be declared as a console command by adding the `[ConsoleCommand]` attribute. Those methods are not required to have a public modifier. Each command requires a unique accessor (key). To invoke a command via the console, use the prefix `'/'` followed by the key. e.g. `"/ExampleKey"`. Note: The accessor of a console command is not case-sensitive.

You can also use the `CommandExample` script for a variety of examples to try out yourself. You can find the script here: **`Assets/Ganymed/Examples`**

```
[ConsoleCommand(key: "Key")]
public static void Cmd()
{
    Debug.Log("Example");
}
```

### Properties . . . . . 3.1

`Key (string):` Unique accessor for the command. This property is required.

`Description (string):` Custom description for the command. The description is shown in listings or when logging information about the command.

`Priority (int):` Higher priorities will be preferred by autocompletion and are shown higher up in listings. The default value for each command is 0.

`DisableNBP (bool):` Determines if numeric input for Boolean parameter for this command is disabled. Note that nbp (numeric Boolean processing) can also be controlled via global configuration. Use this property to disable nbp for specific commands. The default value of this property is false. In other words, nbp is active by default.

`BuildSettings (enum):` Bitmask containing instructions for alternative handling of commands in builds. If you do not want alternative command behavior in your build leave this property as it is. Default value of this property is 0: CmdBuildSettings.none

```
[ConsoleCommand("Key", Description = "Description", Priority = 0, DisableNBP = false)]
public static void Cmd()
{
}
```

```
private const CmdBuildSettings Settings
    = CmdBuildSettings.DisableListings | CmdBuildSettings.DisableAutoCompletion;

[ConsoleCommand("Key", BuildSettings = Settings)]
public static void Cmd()
{
}
```

## Parameter . . . . . 3.2

Commands support parameters. Everything after the key in the input string will automatically be separated (split by spaces) and parsed into multiple arguments that will be converted into the type of the current parameter. Primitives, strings and enums as well as some structs are viable parameter types. Reference types are always passed as null. Some types either require some additional attention or have some altering behavior.

| Type | Category | Note |
|---|---|---|
| bool | primitive | Booleans can be passed as "true" / "false" or additionally "1" / "0" if NBP (numeric bool processing) is enabled in the console configuration as well as the attribute. By default, NBP is enabled. |
| char | primitive | If multiple characters are passed as an argument. The first will be used if it is not null or whitespace. |
| byte | primitive | The input of every primitive numeric parameter is filtered. Every character that is not a number will be removed. |
| byte | primitive | |
| int | primitive | |
| uint | primitive | |
| long | primitive | |
| ulong | primitive | |
| short | primitive | |
| ushort | primitive | |
| decimal | primitive | Floating points can be used with "." or "," |
| double | primitive | |
| float | primitive | |
| enum | primitive enum | Enums can either be entered by their name (string) or value (number) The command processors autocompletion will also suggest names and values of the enum. |
| string | string | Strings might require some additional attention. Arguments are split by spaces which will cause words in strings to be individual arguments. As a workaround it is required to wrap string inputs with multiple words in quotation marks to label them as one related argument. This is not a prerequisite for single word strings but is recommended because the marks will also tell the autocompletion when a string argument is completed and the hint for the next parameter can be shown. |
| Vector2 | struct | Arguments for supported structs are filtered automatically for numeric values. This means that you character that are not a number, "," \|\| "." or space will be ignored. e.g., "x:20,00 y:30.5 z:1" would be a valid input for a Vector3 and result in: (20f, 30.5f, 1f) |
| Vector3 | struct | |
| Vector4 | struct | |
| Vector2Int | struct | |
| Vector3Int | struct | |
| Color | struct | |
| Color32 | struct | |
| class, interface etc. | Reference | Nullable types (except strings) are not supported and will always be passed as null. If a nullable type is used a custom warning will be logged. Because command methods can potentially be used like normal methods you can still use reference types. In this case you can add the attribute [AllowUnsafeCommand] to suppress the warning messages. |

Default parameters values are supported and will be suggested by autocompletion.

```
[ConsoleCommand("Key")]
public static void Cmd(bool param1 = true, int param2 = 20 , string param3 = "example")
{
}
```

## Multiple Signatures . . . . . 3.3

If the same key is used multiple times, the command processor will automatically add additional methods with the same key as overloads to the already existing command. Be careful when using multiple signatures with the same key. If used careless this might produce some unintended behavior.

```
[ConsoleCommand("Key")]
private static void CmdA(int param1, string param2)
{
    Debug.Log($"Signature 1 : Integer: {param1} String: {param2}");
}

[ConsoleCommand("Key")]
private static void CmdB(string param1)
{
    Debug.Log($"Signature 2 : String: {param1}");
}
```

Another option if you want to have several related commands is to add a `'.'` between individual words in the key. Autocompletion will only suggest a key up to the `'.'`

```
[ConsoleCommand("Add.Int")]
private static void CmdA(int param)
{
}

[ConsoleCommand("Add.Float")]
private static void CmdB(float param)
{
}

[ConsoleCommand("Add.Bool")]
private static void CmdC(bool param)
{
}

// "/a"  (when preprocessing is enabled) will suggest "/add.".
```

## Custom Attributes (Console Commands) . . . . . 3.4

### [HintAttribute]

The [Hint] attribute can be used to create a custom description for the parameter of a command. Hints are displayed right before typing in the first character of an argument of the related parameter. The attribute can also be used to determine if and what additional information like name, type and default value of the parameter will be displayed. By default, with or without the attribute, the name and type of the parameter is shown. The attribute has two properties:

`Description (string)`: Custom description for the command. The description is shown in listings or when logging information about the command.

`Show (HintConfig (enum))`: Bitmask enum that determines which if value, type, or name of the parameter is displayed.

```
[ConsoleCommand("Example")]
private static void Cmd([Hint("Example C", Show = HintConfig.ExcludeValue)] string param)
{
}
```

### [SuggestionAttribute]

Second the [Suggestion] attribute can be used for string parameter to add custom suggestions for the autocompletion of a string to compensate for the lack of an adequate default value for strings.

`Suggestions (string[])`: Array containing a collection of suggestions. Suggestion strings can contain multiple words. This property is required and set by the constructor.

`IgnoreCase (bool)`: Value determines if case should be ignored when comparing input strings with suggestion strings.

```
[ConsoleCommand("Example")]
private static void Cmd([Suggestion("Example", "Multiple Words", IgnoreCase = true)] string param)
{
}
```

## Getter and Setter . . . . . 4.0

These attributes will expose any static property or field to be accessible via console input. If you want to get the value of a property / field, you can use the `[Getter]` attribute. If you want to set the value of a property / field, you can use the `[Setter]` attribute. Getter and setter can be combined using the `[GetSet]` attribute. Note: In contrast to a command, the accessor of a getter / setter is case-sensitive.

You can also use the `GetSetExample` script for a variety of examples to try out yourself. You can find the script here: **`Assets/Ganymed/Examples`**

```
[Getter] private static int PropertyGetter { get; }
[Getter] private static int fieldGetter;

[Setter] private static int PropertySetter { set; }
[Setter] private static int fieldSetter;

[GetSet] private static int PropertyGetSet { get; set; }
[GetSet] private static int fieldGetSet;
```

### Properties . . . . . 4.1

The following properties apply to all `[Getter]`[`Setter`][`GetSet`] attributes

**`Shortcut (string):`**
Shortcuts can be used as an abbreviation to access a getter/setter.

**`Description (string):`**
Custom description of the member.

**`Priority (int):`**
Higher priorities will be preferred by autocompletion and are shown higher up in listings. The default value is 0.

**`HideInBuild (bool):`**
Determines if the getter/setter should be excluded from builds.

```
[Getter(Shortcut = "shortcut", Description = "description", Priority = 0, HideInBuild = false)]
```

The following properties only apply to `[Setter]`[`GetSet`] attributes (`ISetter imp`)

**`Default (object):`**
Default value for the autocompletion of the member. (like default parameter)

```
[Setter(Shortcut = "n.a", Description = "n.a", Priority = 0, HideInBuild = true, Default = "e.g.")]

[GetSet(Shortcut = "n.a", Description = "n.a", Priority = 0, HideInBuild = true, Default = 30)]
```

### Getter 4.2

Because getter will only log the value without altering it and because every object can be converted to a string by default, the value of an exposed member will be logged via `ToString()` method. Classes and structures can implement the `IGettable` interface that will replace the default `ToString()` method.

```
[Getter] private static ExampleClass Interface { get; } = new ExampleClass();

private class ExampleClass : IGettable
{
    public string GetterValue() => "example";
}
```

### Setter 4.3

Like the parameter of  a console commands, the value of a set attribute is only viable for certain types.

| Type | Category | Note |
|---|---|---|
| bool | primitive | Booleans can be passed as "true" / "false" or "1" / "0" |
| char | primitive | If multiple characters are passed. The first will be used if it is not null or whitespace. |
| byte | primitive | The input of every primitive numeric setter value is filtered. Every character that is not a number will be removed.<br>This includes decimal, double and float |
| byte | primitive | |
| int | primitive | |
| uint | primitive | |
| long | primitive | |
| ulong | primitive | |
| short | primitive | |
| ushort | primitive | |
| decimal | primitive | Floating points can be used with "." or "," |
| double | primitive | |
| float | primitive | |
| enum | primitive enum | Enums can either be entered by their name (string) or value (number) The command processors autocompletion will also suggest names and values of the enum. Additionally. Setter support [Flags] bitmap enums. You can combine values using the \| operator. |
| string | string | Strings do not require to be wrapped in quotation marks. |
| Vector2 | struct | Arguments for supported structs are filtered automatically for numeric values. This means that you character that are not a number, "," \|\| "." or space will be ignored. e.g., "x:20,00 y:30.5 z:1" would be a valid input for a Vector3 and result in: (20f, 30.5f, 1f) |
| Vector3 | struct | |
| Vector4 | struct | |
| Vector2Int | struct | |
| Vector3Int | struct | |
| Color | struct | |
| Color32 | struct | |
| class, interface etc. | Reference | Nullable types (except strings) are not supported and will always be passed as null. Because command methods can potentially be used like normal methods you can still use reference types. In this case you can add the attribute [AllowUnsafeSetter] to suppress the warning. |

### Attributes 4.4

The custom attribute [DeclaringName] determines a custom "DeclaringName" for Getter and Setter declared in the target class.

[Getter] are by default accessed by the following pattern: `"/get classname.membername"`
[Setter] are by default accessed by the following pattern: `"/set classname.membername value"`

```
[DeclaringName("System")]
private class ExampleClass
{
    // This property is accessible by "/get System.ExampleProperty"
    [Getter] private static string ExampleProperty { get; set; } = "Hello World";
}
```

# Monitoring Modules . . . . . 5.0

Monitoring Modules are a custom system, providing an easy way to verify and review important systems and values during runtime.

## Quick Start . . . . . 5.1

WIP

## Creating Modules . . . . . 5.1

WIP

| | |
|---|---|
| `virtual void Tick()` | Tick is called every Unity-Update if the module is enabled. Override this method if your module requires logic that needs to be executed every frame. E.g. calculating frames per second. |
| `virtual void OnInspection()` | Method is called repeatedly if auto inspection and the module are enabled. Delay between calls can be set in the inspector. Override this method if the value of the module might require to be validated occasionally. You might want to check a value that could be altered from a variety of different sources. E.g. Check the state of the cursor without creating and subscribing to every method that might alter the state of the cursor. This way we can check the state of the cursor and are not required to follow up on every influence that might alter it. |

| | |
|---|---|
| `virtual string ParseToString(T currentValue)` | Override this method if the style of the value is dynamic (e.g. alter the fps color depending on new value.) |

| | |
|---|---|
| `abstract void OnInitialize()` | OnInitialize is called when the module is initialized which will happen during the Unity Start method. (including editor) |

| | |
|---|---|
| `virtual void ModuleEnabled()` | Called when the module gets enabled |
| `virtual void ModuleDiabled()` | Called when the module gets disabled |
| `virtual void ModuleActivated()` | Called when the module gets activated |
| `virtual void ModuleDeactivated()` | Called when the module gets deactivated |
| `virtual void ModuleVisible()` | Called when the module canvas element gets activated |
| `virtual void ModuleInVisible()` | Called when the module canvas element gets deactivated |

| | |
|---|---|
| `virtual void OnBeforeUpdate(T currentValue)` | OnBeforeUpdate is called before the value was processed. |
| `virtual void OnAfterUpdate(T currentValue)` | OnAfterUpdate is invoked on every module update event (GUI, UPDATE, etc.) |

| | |
|---|---|
| `virtual void OnQuit()` | OnQuit is called either when exiting Play-Mode or when quitting the application. |

WIP



Every module has some basic settings. WIP

| | | |
|---|---|---|
| Is Enabled | bool | If disabled, the modules Tick method will not be called. |
| Is Active | bool | If inactive, the modules will be invisible. |

| | | |
|---|---|---|
| Use Custom Style | bool | If enabled, the custom style will be used if provided |
| Custom Style | Style | Custom style for the module providing settings like font size, color etc. |

# Utilities . . . . . 6.0

The Utility Assembly contains extensions, Helper etc. some are worth noting here because other systems might require these resources. Feel free to look at the resources of this assembly yourself to see if you can find anything that might be of use for you.

## Hierarchy Optimization . . . . . 5.1

Optimizing the hierarchy is an important step when it comes to increasing performance. There are a lot of good articles on why it is important and how to do it. Having an optimized hierarchy and one that is sorted are two things that are hard to combine. A few custom components can be used for generic hierarchy optimization.

First the `UnfoldObjectOnLoad` class. This component will set a new root parent for its children and then destroy itself during initialization.

Second the `SetRootOnLoad` class. This component will set a new root for its GameObject and then destroy itself during initialization.

Third the `DestroyOnLoad` class. This component will destroy the GameObject during initialization.

All components offer various settings. You can also use the Hierarchy Optimization Configuration in `Assets/Ganymed/Utils/Configurations` to access global settings for those components. If no object is available, you can create a new using: `(Rightclick => Create => Hierarchy Optimization => Configuration)`

## Unity Event Callbacks . . . . . 5.1

This utility class provides a variety of callbacks for unity events, messages, and editor callbacks. Subscribed listener are not required to check the application state for editor callbacks. Both the Console and the Monitoring systems require this helper. You do not have to manually create an instance of this object because both systems validate its integrity automatically. The gameobject is hidden by choice. If you would like to unhide it for any reason you can do so: `(Help => Hide Flags => Show All Objects).` You can also manually create an instance of this object (if you do not use the console or monitoring): `(GameObject => Ganymed => UnityEventCallbacks).`

Note: Callbacks from this class will be executed first in their category. This means that listener subscribed to the Awake callback of this class will be executed before any other awake function. This is due to the script's low script execution order.

To get a list of available callbacks / event types you can also check the UnityEventType enum. In the Ganymed.Utils namespace. The UnityEventCallbacks class contains two static methods as an access point. Both methods have multiple parameter and signatures.

### `UnityEventCallbacks.AddEventListener()`

Use this method to subscribe a listener to a unity event callback. This Method has multiple Signatures.

| Index | Type and Name | Description (Main Signature) |
|---|---|---|
| 0 | `Action<UnityEventType> listener / Action listener` | The listener you would like to add. |
| 1 | `bool removePreviousListener` | Check assure that the listener is not subscribed multiple times. |
| 2 | `ApplicationState callbackDuring` | Set if callbacks can be invoked during playmode, editmode or both. |
| 3 | `params UnityEventType[] callbackTypes` | What events types the listener should be subscribed to. |

| Index | Type and Name | Note (alt Signature) |
|---|---|---|
| 0 | `Action<UnityEventType> listener / Action listener` | Previous listener will be removed.<br>`bool removePreviousListener = true` |
| 1 | `ApplicationState callbackDuring` | |
| 2 | `params UnityEventType[] callbackTypes` | |

| Index | Type and Name | Note (alt Signature) |
|---|---|---|
| 0 | `Action<UnityEventType> listener / Action listener` | Listener will be subscribed to both Editmode and Playmode callbacks.<br>`ApplicationState callbackDuring = ApplicationState.EditAndPlayMode` |
| 1 | `bool removePreviousListener` | |
| 2 | `params UnityEventType[] callbackTypes` | |

| Index | Type and Name | Note(alt Signature) |
|---|---|---|
| 0 | `Action<UnityEventType> listener` | Previous listener will be removed. And the listener will be subscribed to both Editmode and Playmode callbacks. |
| 1 | `params UnityEventType[] callbackTypes` | |

### `UnityEventCallbacks.RemoveEventListener()`

Use this method to remove a listener from a unity event callback. This Method has multiple Signatures.

| Index | Type and Name | Description (Main Signature) |
|---|---|---|
| 0 | `Action<UnityEventType> listener / Action listener` | The listener you would like to remove. |
| 1 | `ApplicationState applicationState` | Can the listener be removed form playmode, editmode or both callbacks. |
| 2 | `params UnityEventType[] callbackTypes` | What events types the listener should be unsubscribed from. |

| Index | Type and Name | Note (alt Signature) |
|---|---|---|
| 0 | `Action<UnityEventType> listener / Action listener` | The listener will be removed from both playmode and editmode callbacks. |
| 1 | `params UnityEventType[] callbackTypes` | |

## Attributes . . . . . 5.2

The utilities assembly provides a variety of custom attributes, Extension methods etc. Feel free to use everything as you see fit. Every custom attribute class is commented. Most of the following attributes are reflection helpers that will provide custom warning messages.

### `[AttributeTarget]` . . . . .

This attribute can should only be applied to other attributes. It lets you determine specific types to which the target attribute can be applied to. You could compare it to the `[AttributeUsage]` attribute in the System namespace, only that it will let you limit the target to specific types instead of categories.

**`Inherited (bool):`**
Are types permitted that are a subclass of or are derived from the specified types.

`[PropertyAccessRequired]`

A very niche attribute that allows you to specify access requirements for the target of an attribute. It can only be applied to other attributes. If the target of the target attribute is a property and does not meet the required access specification a warning will be logged.

`RequiresRead (bool):`
Does the attribute require the property to have read (get) access?

`RequiresWrite (bool):`
Does the attribute require the property to have write (set) access?

`[RequiredAccess]`

Just like the property-access-attribute, This attribute allows you to specify requirements for the target of an attribute. It can only be applied to other attributes. If the target of the target attribute is a member and does not have the required modifier a warning will be logged. Note: both properties have nullable backfields. This means that only requirements will only be set when using the properties. If left alone no requirement will be set.

`Static (bool):`
Are targets required to be static or required to be non-static?

`Public (bool):`
Are targets required to be public or required to be non-public?

`[RequiresAdditionalAttributes]`

RequiresAdditionalAttributes states that instances of the specified target attribute require instances of the passed attribute/s type/s to be valid. With other words an attribute with this attribute cannot be applied to anything without at least a second attribute of the specified type.

`Inherited (bool):`
Determines whether the instances of the required types can be a subclass of the types or not.

`RequiredAttributes (Type[]):`
An array containing types required by the attribute. This array can only contain types that are a subclass of System.Attribute.

`RequiresAny (bool):`
Value indicates whether instances of the specified target require instances of every specified type / attribute to be valid or if the attribute is valid as soon as there is any instance of the specified types / attributes alongside. True if only one instance is required. False if instances of every type are required.

### [TargetParamRestrictions] And [TargetTypeRestriction]

The target-param-restrictions attribute restricts the parameter types of the methods to which the target attribute can be applied to. If the parameter of an affected methods does not fit the specified type/s a warning will be logged. The target-type-restriction attribute limits the types to which a target attribute can be assigned. Both individual types and categories can be specified. Because both attributes operate in a similar way, they share the same properties.

**ValidTypes (Type[]):**
An array containing valid types.

**Inherited (bool):**
Determines whether the types of the (parameter or target) types can be a subclass of the valid types or not.

**ValidTypeAffiliations (enum TypeAffiliations):**
Bitmask enum that contains categories of valid types.

**AllowXXX (bool):**
This is not one but a collection of properties that lets you set individual type affiliations. E.g., AllowStruct will set every struct to be valid. XXX => (Primitives, Strings, Enums, Class, Generic, Interface, Struct)

### [ScriptOrder]

A simple attribute that will automatically set the target behaviors script execution without having to set it manually in the editor.

**order (int):**
The value of the script execution order.

### [HideFlags] & [GameObjectHideFlags]

Two simple attributes that will automatically set the target behaviors Instance(script) or GameObjects Hide Flags. This Attribute can only be applied to MonoBehaviors. Hide Flags are validated OnLoad and during Awake.

**hideFlags (HideFlags):** The state that will be applied to the Component / GameObject

**[HideInHierarchy] & [GameObjectHideInHierarchy]**
This attribute is an abbreviation for `[HideFlags(HideFlags.HideInHierarchy)]`
Or `[GameObjectHideFlags(HideFlags.HideInHierarchy)]`