Smart Console

Unity Package

User Manual

Version 2.2.1

June 14, 2023

Table of content

1. Introduction

1. Welcome

Thank you for expressing interest in this package! I am confident that it will prove to be a **valuable asset** to help you realize your games. If you encounter any issues or require assistance with anything related to this package, please don't hesitate to reach out to me at edgarcovarel@yahoo.fr or via the discord server.

Additionally, if you have any feedback or ideas about this package, I would be delighted to hear them.

2. About

Elevate your game development experience with Smart Console - a flexible and intuitive in-game command console that lets you easily create custom commands by simply adding **[Command]** to your code.

Say goodbye to tedious debugging and hello to seamless development with Smart Console.

3. Use cases

Smart Console is ideal for projects that require **cheat codes** or **debugging assistance**, as well as those that need to implement **QA tools**.

You could also use it to **adjust game settings** in real-time or more simply to **test gameplay mechanics**.

Its versatility makes it an excellent choice for any game development project!

2. Getting Started

1. Getting Started

To get started, simply follow these steps:

- Make sure that the **Text Mesh Pro** package is installed (it is installed by default for new projects).
- Make sure that you have an **EventSystem** in your scene to ensure the functionality of the UI (go to GameObject > UI > EventSystem).
- Add the Smart Console prefab to your scene (located at Assets/Smart Console/Prefabs).

Note that the input to toggle the Smart Console is the **input key Escape** by default.

Voilà, you are now ready to use Smart Console!

2. Adding Command

To write your own commands to the console you will need to use the **[Command]** method attribute.

By writing **[Command]** above a function, it will identify this as a command to load on the Smart Console initialization.

Example

```
using ED.SC;
...

[Command]
public static void Add(int n1, int n2)
{
    SmartConsole.Log((n1 + n2).ToString());
}
```

*In this example, **Add** function is loaded in the Smart Console which means that it can be used in the console as is:*

```
> Add 2 2
4
```

At the top the user command invocation and under the Smart Console response.

3. Commands

1. Command

The **[Command]** method attribute is used to identify your functions as commands in the Smart Console.

Some parameters can be used to change the command behaviour:

- name: the name of the command in the Smart Console. By default the function name.
- description: the description of the command to help understanding it. By default empty.
- monoTargetType: the monobehaviour target type on which command should be invoked, please refer to the *Targeting process* tab for more information. By default MonoTargetType.Single.

Smart Console **does not** support commands with the same name.

The **[Command]** method attribute works with:

- Public/private function
- Static function
- Function with int/float/bool/string or any Enum parameter(s)
- Function with optional parameter(s)
- Coroutine

Example

```
using ED.SC;
...

[Command("get_name", "Prints gameObject name", MonoTargetType.All)]
public void GetName()
{
    SmartConsole.Log(gameObject.name);
}
```

In this example, GetName command's name is "get_name", its description is "Prints gameObject name", and its monoTargetType is all.

2. Targeting process

When invoking a command, Smart Console will look for its target(s) in the gameobjects of your loaded scenes. To control the behaviour of this process, you need to change the monoTargetType parameter on your function's **[Command]** attribute.

There is multiple MonoTargetType:

- Single: Targets the first active instance found of the MonoBehaviour.
- Active: Targets all active instances found of the MonoBehaviour.
- All: Targets both active and inactive instances found of the MonoBehaviour.

Note that MonoTargetType do not concern static functions.

4. Autocompletes

1. Command autocomplete

Smart Console includes a contextual command autocomplete system. It will provide commands suggestions depending on your text input.

To copy first suggestions, press the **input key Tab**.

To copy the last suggestions, repeat the process until the last suggestion is copied.

For further explanation, each time the **input key Tab** is pressed, it will copy the next autocomplete suggestion if there is one, if it hits the last suggestion, it will go back to the first suggestion.

In order to know which suggestion is copied, the copied suggestion is **coloured in orange**.

2. Parameter autocomplete

Once you have copied the command you wanted to use, you can use the contextual parameter autocomplete system. It will provide parameters suggestions depending on the parameter type.

Firstly, you need to **select the parameter** you want to autocomplete.

To select the first parameter, press the **input key Shift**.

To select the last parameter, repeat the process until the last parameter is selected.

For further explanation, each time the **input key Shift** is pressed, it will select the next parameter if there is one, if it hits the last parameter, it will reset the parameter selection.

In order to know which parameter is selected, the selected parameter is **underlined**.

Secondly, you need to press the **input key Tab** to autocomplete the selected parameter.

Two cases of behaviour:

- Your input parameter was empty: it will copy the default parameter (as examples: default int is 0, default bool is false, default Enum is the first value).
- Your input parameter was filled: it will copy the next parameter, for int and float it depends on the Parameter Modifiers, please refer to the *Parameter Modifiers* tab for more information (as examples: next int is incrementation, next bool is inverse, next Enum is the next Enum value).

To add your custom types, please refer to the *Custom types* tab.

5. Customizations

1. Inputs

You can customize the console inputs going on the Smart Console prefab (located at Assets/Smart Console/Prefabs) and doing a modification of the **InputHandler** component at the root of the prefab.

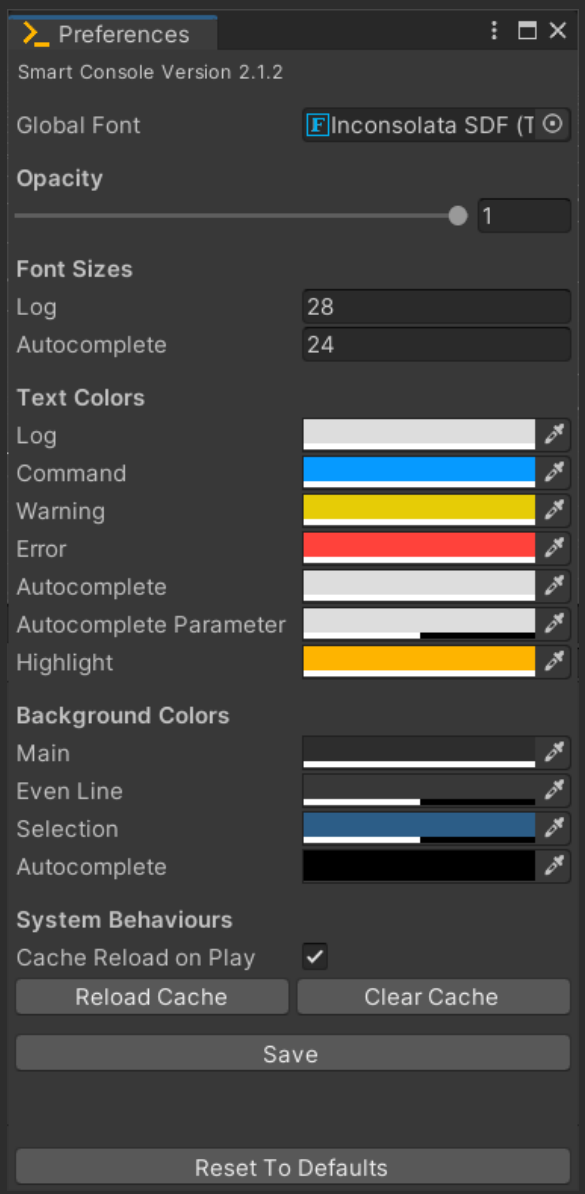
Modifications are possible on both Input System and Legacy Input System.

2. Themes

You can customize the console UI without modifying the prefab using the Smart Console Preference window (go to Tools > Smart Console > Preferences)

You will be able to modify the font, opacity, font sizes, text colors, background colors, cache behaviours...

Smart Console Preference window



3. Parameter modifiers

The parameter modifiers are modifiers for int and float parameters, it is used to determine the next parameter that will be autocomplete using the parameter autocomplete, please refer to the *Parameter autocomplete* tab for more information.

If the int parameter modifier **is 2**, each use of parameter autocomplete will increase the parameter int **by 2**. Same for the float parameter modifier.

By default, Int parameter modifier is **1** and float parameter modifier is **0.1**.

You can modify these values any time at the top of the **TypeUtil** script (located at Assets/Smart Console/Utils).

4. Custom types

You can add your custom types going into the **TypeUtil** script (located at Assets/Smart Console/Utils) and doing modifications on the spots provided for this purpose.

The 2 spots with the comment “write your custom types here” are must-haves to make it usable in the console.

The 2 other spots with the comment “write your custom types here (optional - parameter autocomplete usage)” are optional to make it usable with the autocomplete.

6. Extra

1. Cache

To improve performance as its maximum, Smart Console uses a cache to store all command data.

When you add new commands, you will need to reload this cache but don't worry, there is an **automatic reload option** that allows you to do it automatically on play in the editor.

Good to know: **50 commands** take **15 to 30 ms** to be loaded in the cache on my computer. So, most of the time you will leave that option ticked.

For bigger project where it can be annoying to have that automatic reload, you can untick it on the tool's preferences and reload the cache **manually**.

By default, automatic cache reload on play is **ticked** in the preferences.

2. Group

To package commands into groups, you can use the **[CommandGroup]** class attribute. It is useful when you have a lot of commands to organize them.

When you want to get the commands from a specific group, you can use the command **GetCommandsFromGroup** that takes a string as parameter.

Note that default group is named "Default".

```
using ED.SC;
...

[CommandGroup("Test")]
public class TestCommands()
{
    [Command]
    public void PackagedCommand()
    {
        // command action
    }
}
```

*In this example, the class **TestCommands** packages the command **PackagedCommand** to the group "Test" using the class attribute **CommandGroup**.*

3. Refresh autocomplete

Sometimes you need to instantiate a new GameObject on your scene that might contain scripts with commands, to see these new commands in the autocomplete suggestions, you need to add them to the autocomplete system.

- To add a type to the autocomplete → use the function **AddAutocomplete** with the type you want to add as parameter.
- To remove a type from the autocomplete → use the function **RemoveAutocomplete** with the type you want to remove as parameter.
- To refresh all types of the autocomplete → use the function **RefreshAutocomplete** (heavy).

Note that commands which are not in the autocomplete still can be used if the instance is loaded on one of your scenes.

Example

```
using ED.SC;
...

public void Awake()
{
    SmartConsole.AddAutocomplete(this);
}

[Command]
public void NewRuntimeCommand()
{
    // command action
}

public void OnDestroy()
{
    SmartConsole.RemoveAutocomplete(this);
}
```

*In this example, we use the **AddAutocomplete** function at **Awake** and the **RemoveAutocomplete** at **OnDestroy** to ensure autocomplete suggestion of **NewRunetimeCommand** command during type lifetime.*

4. Console log

To print logs into the console, you can use these functions depending on the type of the message that you want to print:

- Log → SmartConsole.Log("your log");
- Warning → SmartConsole.LogWarning("you warning");
- Error → SmartConsole.LogError("your error);

Note that the console prints all application logs if you tick the variable **ShowApplicationLogs** in the **ConsoleSystem** component of the Smart Console prefab.

5. Command request

It may happen that you want to use a command by code.

To do so, you need to use the function **Send** from the **SmartConsole** static class.

Send function does work exactly like if you would have submitted an input text from the UI input field.

I **do not** recommend using that because it is the best way to have a bad project architecture. But it can be useful in very specific situation.

6. Extra Commands

Extra commands are included in this package to have some basic commands useful for any kind of projects.

Smart Console includes:

- **CoreCommands** → implements Smart Console core commands help, version, quit.
- **ScreenCommands** → implements screen related commands like toggle fullscreen, capture screenshot, set resolution…
- **GraphicsCommands** → implements graphics related commands like set max fps, set vsync…
- **PlayerPrefsCommands** → implements player prefs related commands like get player prefs key, set players prefs key, delete player prefs key…
- **SceneCommands** → implements scene related commands like load scene, unload scene…
- **TimeCommands** → implements time related commands like get time scale or set time scale.

For more informations about theses commands, please use the command **help**.