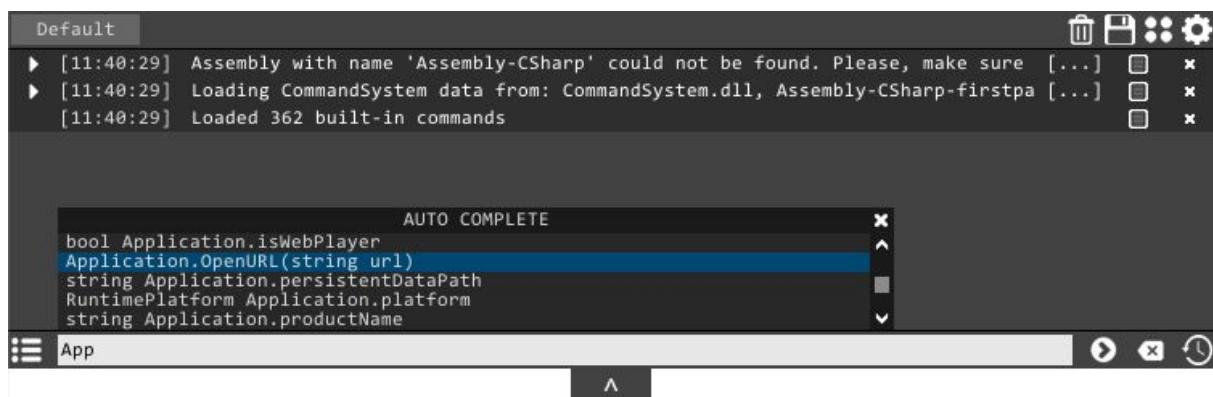


# DevConsole Documentation

## About

For the latest updated documentation, please, visit the [online docs](#).

**DevConsole** is a tool mostly aimed at the development and testing stages of a project. Through a visual interface, it allows you to communicate directly with your game by being able to **see internal logs & execute code**.



Commands management is implemented by the [CommandSystem](#), which has a MIT license.

With **DevConsole** you can:

- Execute code with different arguments
- Show an autocomplete list of commands based on your input
- Navigate through previously typed commands
- See both Unity logs and command responses
- And many more

**DevConsole** is developed and maintained by [Antonio Cobo](#).

Contact: [antoniocogo@gmail.com](mailto:antoniocogo@gmail.com)

# Version History

## Version 1.0.41.180712

- Support for Unity 2018.2

## Version 1.0.40.180627

- Fixed an issue where setting either AutoComplete or History windows open keys to "None" would cause unexpected behaviour.

## Version 1.0.29.180612

- An event called *onOpenStateChanged* is now called every time DevConsole is open or closed.
- Exceptions thrown within command calls were previously not logged. This was a bug and is now fixed (thanks to [XCVG](#)).
- Added *autoCompleteWithEnter* as a new settings to choose whether pressing enter should work with AutoComplete or not. It is enabled by default.
- Added a new setting to completely disable the History window, in case you don't want/need it. It is enabled by default.
- Added a new setting to control the way the AutoComplete window behaves. It has three possible values:
  - **Disabled:** Completely disables the AutoComplete window.
  - **Manual:** Enables the AutoComplete window, making it appear when either pressing F2 (default) or clicking its interface button.
  - **Auto:** Additionally, the AutoComplete window will also appear when typing something in the input field.

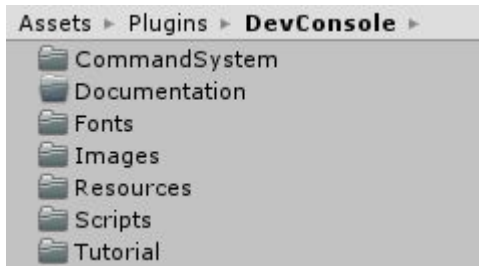
## Version 1.0.7.180506


Initial release.

# Quick start guide

## Installation

Right after importing the package, a folder named *DevConsole* should be added to your *Assets/Plugins* directory. This is what it looks like:



The package should be ready to work out of the box just after importing the package. Pressing play should greet you with a little button on top of the game window . Clicking on that button should open the **DevConsole** interface.

The *Resources* folder contains the two main assets you will be using to work with **DevConsole**.

The prefab *DevConsole* is automatically instantiated when hitting play, and contains the main functionality. It also contains all the configuration related to how each component of the interface looks and behaves. For the most part, you'll probably want to leave it as it is.

On the other hand, the asset *DevConsoleSettings* contains configuration about how **DevConsole** behaves and how it looks in general. You will most probably want to edit this to your own liking.

## Tutorial

The folder *Tutorial* contains an example scene with which you can play in the editor. Open the scene, click play and follow the instructions step by step. Once you have completed the tutorial, you can remove the folder entirely if you wish so.

## Usage

The *Tutorial* folder contains a whole bunch of examples on how to create and use commands. However that, here is a brief explanation.

For a more elaborated and in-depth explanation, please, refer to the in-depth section.

**Note:** For the majority of situations, you will need:

```
using SickDev.CommandSystem;
```

## Creating Commands

To create a command, it is sufficient to call `DevConsole.singleton.AddCommand` and pass in the type of command you want. Use `ActionCommand` for `void` methods and `FuncCommand` when the method in hand returns something.

```
public class Room3 : Room {
    void OnEnable() {
        DevConsole.singleton.AddCommand(new ActionCommand<int>(OpenRoom) { className = "Example" });
    }

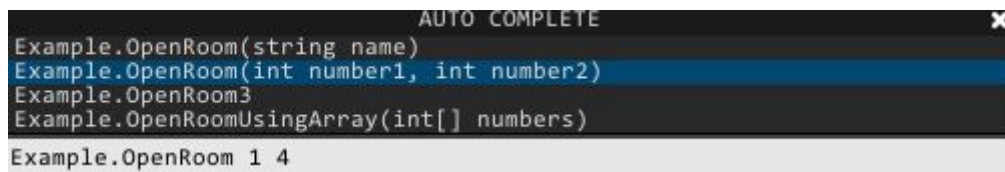
    void OpenRoom(int number) {
        if(number < 4)
            Debug.LogError("That room is already open");
        else if(number > 4)
            Debug.LogError("Do you really wanna break the tutorial? -_-");
        else if(!isComplete)
            Complete();
        else
            Debug.LogWarning("You have already opened that room. Move on!");
    }
}
```

As an alternative, you can make use of `CommandAttribute`, which automatically registers the method it is used on. This approach, however, only works with static methods.

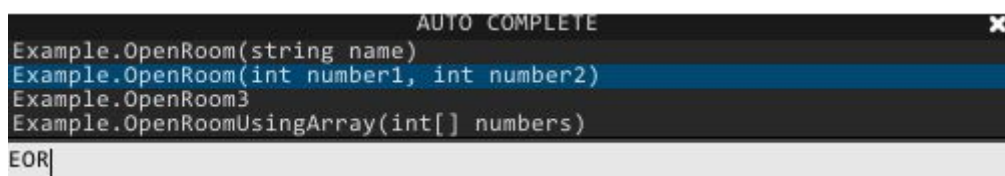
```
[Command]
static void ExampleCommandAttribute() {
    Console.WriteLine("Command called!");
}
```

## Executing Commands

With the **DevConsole** interface open, type the name of the command you want to call. If the command requires parameters, these can be appended at the end of the command name, separated by spaces.



When typing a command name, the *AutoComplete* window will appear. You can navigate the list using the arrow keys, and pressing `tab` or `enter` will select the desired command. To speed up the process, typing the capital letters of the command name will narrow the list only to the desired commands.



# In-depth guide

The following pages will exhaustively explain all the details about the workings of **DevConsole**.

## Command System

Commands management is implemented by the [CommandSystem](#), which allows to parse strings into ready-to-use commands.

Whenever a text is entered in the input field of the console, that same text is sent to a *Commands Manager*, which in turn executes the corresponding command.

The [CommandSystem](#) already contains a full in-depth guide of how it works, so feel free to read it and familiarize yourself with it.

That being said, however, I will explain here how to add and execute commands. The following is a modified abstract from [CommandSystem](#)'s readme.

### [Adding and creating commands](#)

Commands can be created in three different ways.

- Manually
- Using the *CommandAttribute*
- Using the *CommandsBuilder*

	Static Members	Instance Members	Methods	Delegates & MethodInfo	Properties	Variables	Add/Remove in Runtime
Manually	X	X	X	X			X
Command Attribute	X		X				
Commands Builder	X		X		X	X	X

### Manually

Manually adding commands can prove useful when all you need is flexibility. However, it is tedious for large amounts of commands or when you just need a quick solution. It is the only of the three methods that allows for instance methods and the usage of delegates and MethodInfo. It is limited, however, in that it does not allow to directly access properties or variables. To do that, it is advised to create a wrapper delegate.

Using this method, you first need to create the command and then add it to the **DevConsole** by calling *DevConsole.singleton.AddCommand*. A command can be created using one of the different *CommandType* classes, depending on your needs. *CommandTypes* are separated between *ActionCommands* and *FuncCommands*, just like .NET delegates.

```

public class Room3 : Room {
    void OnEnable() {
        DevConsole.singleton.AddCommand(new ActionCommand<int>(OpenRoom) { className = "Example" });
    }

    void OpenRoom(int number) {
        if(number < 4)
            Debug.LogError("That room is already open");
        else if(number > 4)
            Debug.LogError("Do you really wanna break the tutorial? ~~~");
        else if(!isComplete)
            Complete();
        else
            Debug.LogWarning("You have already opened that room. Move on!");
    }
}

```

## Using the Command Attribute

In order to use this method, just place the Command attribute on a static method and it will be available out of the box. It is great when you need something quick and don't want to bother adding the command manually.

```

[Command]
static void ExampleCommandAttribute() {
    Console.WriteLine("Command called!");
}

```

## Using the Commands Builder

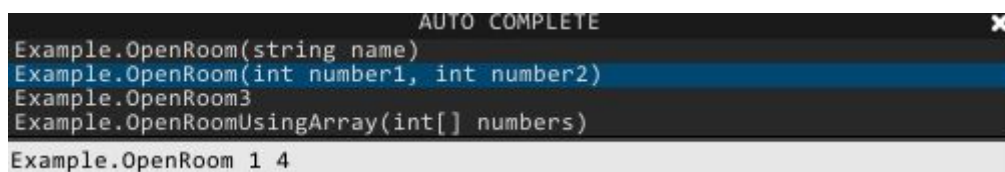
Using the *CommandsBuilder* is the perfect solution when you need to mass generate commands. It uses reflection to get members from a Type and converting them into commands. The downside is that it only work with instance members.

For a full example on how to use it, see the [BuiltInCommandsBuilder.cs](#) file.

## Executing commands

To execute a command, just type its name and its arguments and press enter. The first space marks the end of the command name, and every successive space marks the beginning of a new argument.

If an argument needs to have spaces in it (such as a string, and object or an array), encapsulate the arguments between " or '.



The screenshot shows a window titled "AUTO COMPLETE" with a close button (X). It contains a list of command suggestions:
 

- Example.OpenRoom(string name)
- Example.OpenRoom(int number1, int number2)
- Example.OpenRoom3
- Example.OpenRoomUsingArray(int[] numbers)

 Below the list, the input text "Example.OpenRoom 1 4" is shown in a separate line.

In order to determine which command to execute, if any, the commands manager looks for every overload of the command; that is, commands that have the same name as the one in the input text. After that, overloads are filtered out by those that have the same number of arguments and that successfully pass the conversion of the input arguments into their argument types. Those that

successfully pass the test are considered command matches: commands with the potential of being executed.

The following can happen:

- If no overloads are found, a *CommandNotFoundException* is thrown.
- If one or more overloads are found, but there are no matches, a *MatchNotFoundException* is thrown.
- If more than one match is found, an *AmbiguousCommandCallException* is thrown.
- If exactly one match is found, the command is invoked.

### Explicit casts

In order to prevent *AmbiguousCommandCallException* from being thrown, one can and should cast arguments into specific types. To explicitly cast an argument to a specific type, wrap the type between braces and place it before the argument, without spaces.

Examples:

- `CommandName (int)2 -->` Finds an overload that has one integer argument.
- `CommandName (float)2 3 -->` Finds an overload that has two parameters and the first one is a float. The second is converted automatically.

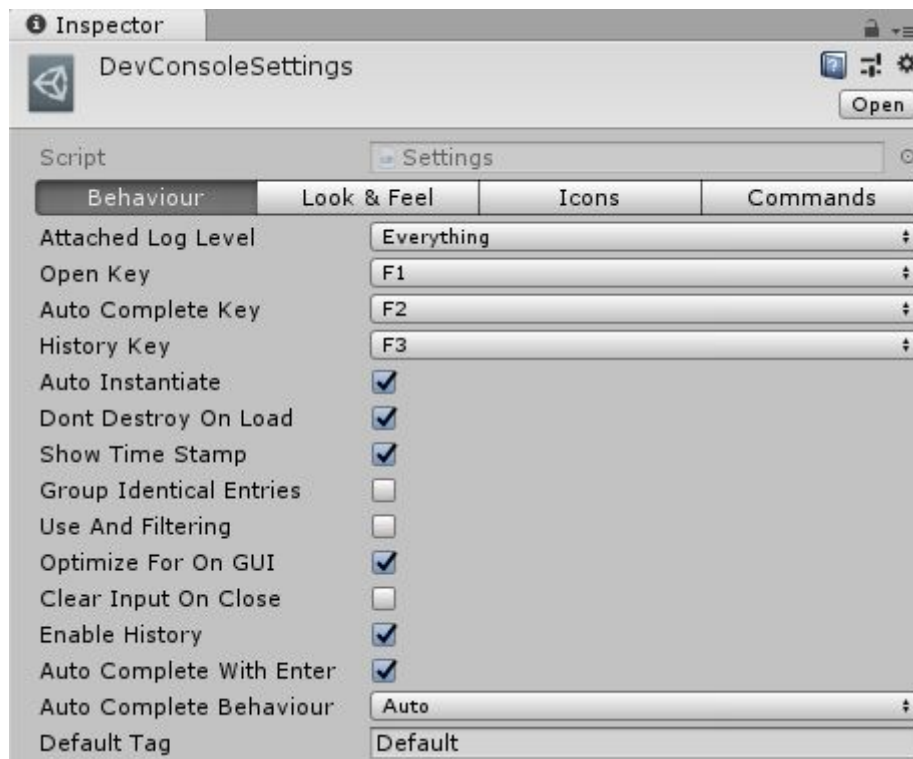
## DevConsole Settings

The *Resources* folder contains a file called “DevConsoleSettings”. That file holds the vast majority of **DevConsole**’s configuration.

Its inspector is divided into 4 tabs:

- Behaviour
- Look & Feel
- Icons
- Commands

### Behaviour



### Attached Log Level

By default, **DevConsole** will log any message that would go into the editor’s Console, such as *Debug.Log* calls, internal messages or exceptions.

You can filter what kind of messages gets logged with this setting.

### Keys

The next three parameters specify which keyboard keys should be used to open/close the **DevConsole**’s interface, the AutoComplete window and the History window, respectively.



## Auto Instantiate

If checked, **DevConsole** will be automatically instantiated in the scene right after the game starts. You'd normally want to leave it checked by default. Additionally, you might want to leave *Dont Destroy on Load* checked by default as well.

## Show Time Stamp

When checked, **DevConsole** will show a timestamp right next to each log entry. You can change this value at runtime too.

## Group Identical Entries

When checked, entries with the exact same stack trace and content will be grouped as only one entry, showing right next to it the amount of entries of that type that have been logged. This helps unflood the view, and can be toggled from within **DevConsole**'s interface at runtime too.

It is similar to what Unity's console does when activating the "collapse" option.

## Use And Filtering

If left unchecked, tabs will be treated as OR FLAGS, and so, entries will be shown as long as one of their tags is the one in the tab.

When checked, however, tabs will be treated as AND FLAGS, so that only entries containing ALL the checked flags will be shown.

## Optimize For OnGUI

Leave it checked if NO other OnGUI method is used. That will slightly improve performance.

On the other hand, if any other code in your project makes use of the IMGUI system, uncheck it.

## Clear Input On Close

If checked, the input text will be cleared every time the console closes. It is unchecked by default.

## Enable History

Unchecking it completely removes the History window functionality.

## Auto Complete with Enter

If checked, pressing *enter* will also work as the *tab* key to select entries in the AutoComplete window.

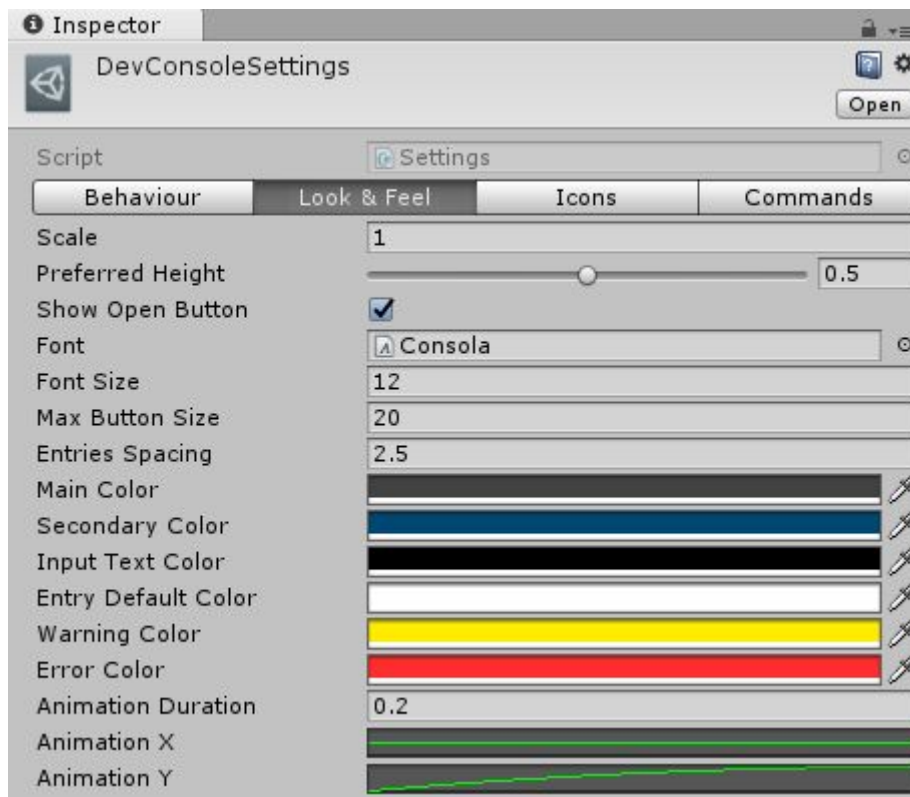
## Auto Complete Behaviour

- **Disabled:** Completely disables the AutoComplete window.
- **Manual:** Enables the AutoComplete window, making it appear when either pressing F2 (default) or clicking its interface button.
- **Auto:** Additionally, the AutoComplete window will also appear when typing something in the input field.

## Default Tag

Default Tab's name.

## Look & Feel



### **Scale**

Change this value to adjust the size of interface elements to your target resolution.

Can be changed at runtime too.

### **Preferred Height**

How much of the screen's relative height should the interface use. A value of 0.5 means it will occupy half the screen.

### **Show Open Button**

Whether to show or not the open/close button. If you are targeting touch devices, chances are you want to leave this checked.

### **Font**

Controls the font used.

### **Max Button Size**

Change this value if you have trouble pressing on some icons.

### **Entries spacing**

Space left blank between log entries. Increase this value if you feel like the interface is too clogged.

### **Colors**

You can control the color used on various aspects of the interface. It allows you to control alpha value as well.

## **Animation**

Controls the open/close animation.

## Icons

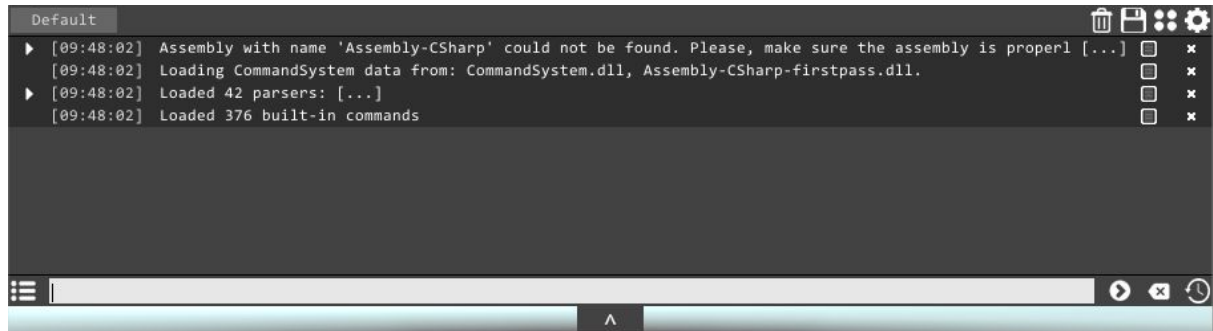
Controls which icons are used. You should leave it as it is, as it's meant for internal purposes.

## Commands

Controls which built-in commands are available by default. Each section enables/disables commands for that specific class.

**Note:** When targeting Android, enabling *Microphone* and *Location* will also add those same permissions to the final .apk.

# The User Interface



**DevConsole's** can be divided into 3 parts:

- Toolbar
- Logger
- Input field

Additionally, there are two popup windows:

- AutoComplete
- History

## The Toolbar

The toolbar is the topmost section of **DevConsole's** interface. It shows all the tabs available together with a few extra buttons.

## **Tabs**

Tabs are used to filter log entries based on their tags. For each different tag across all entries, a new tab will be shown.

Clicking on a tab will enable/disable that tag; effectively showing or hiding entries based on the filtering type used.

By default, tabs will be used as OR filtering; which means that entries will be shown as long as at least one of its tags is enabled.

On the other hand, when *Use And Filtering* is enabled, entries will only be shown when all of their tags are enabled.

Tabs will be automatically deleted when there are no entries with their tag.

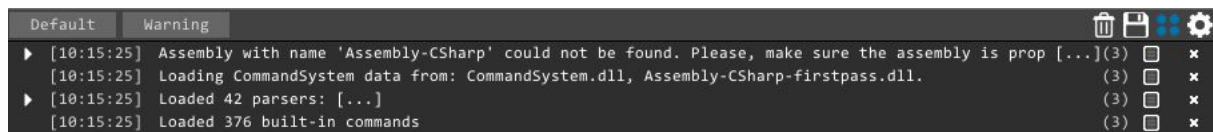
## **The Clear Button**

When the clear button is clicked, the logger will be cleared, effectively deleting all entries and tabs.

## **Save Log Button**

When clicked, the whole log will be dumped into a file at the root of *Application.persistentDataPath* with the name "DevConsole.log".

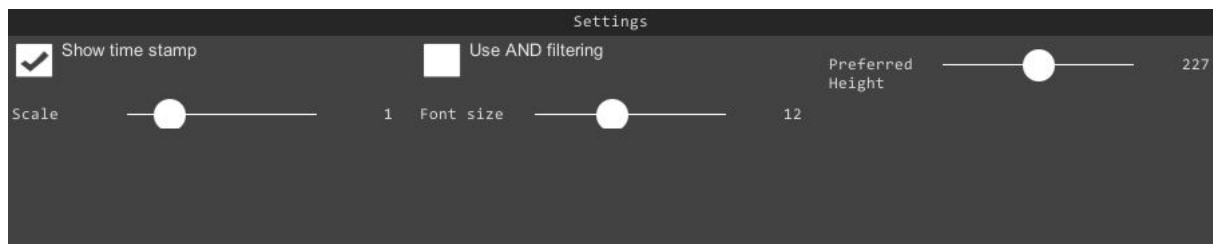
## Group Identical Entries Button



Works as a toggle. When enabled, entries will be collapsed the Unity console's way. A number will be shown right next to each entry showing how many entries have the exact same message and stacktrace.

## Settings Button

Works as a toggle, replacing the logger panel with the settings panel.



Has several widgets that allow for quick change of configuration at runtime.

## [The Logger](#)

The logger shows a list of entries in order of appearance, from top to bottom.

When an entry's text can't be fit within a single line, an arrow appears at its left. Clicking that arrow shows/hides the full text.

Additionally, entries can also be expanded to show their stack trace by clicking on the button on their right.

Finally, clicking on the X button will delete that entry (or group of entries) from the log.

## [The Input Field](#)



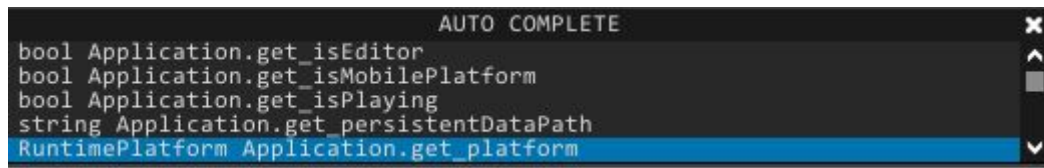
The input field in the part where commands are input. When the interface is open, it gets the keyboard focus automatically.

On the left, you have the *AutoComplete* button, with which you can open/close the window.

On the right, there is the Submit, Delete and History buttons, in that order. The Submit button, when clicked, executes the input command, while the Delete button completely clears the input field.

Lastly, the History button toggles the History window.

## The AutoComplete Window



The *AutoComplete* window, when open, allows you to see a list of commands matching the current text in the *Input Field*. Entries can be navigated using the arrow keys, and selected by either clicking on them or pressing the *tab* key.

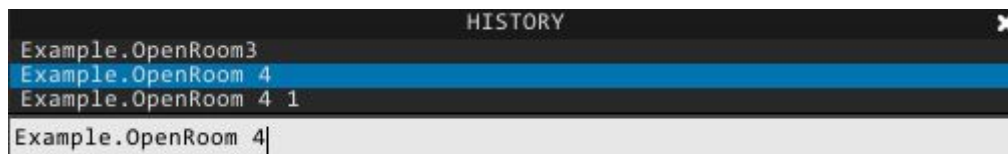
The *AutoComplete* feature separates commands by words, which are delimited by capital letters, dots and underscores.

For instance, "Application.get\_isEditor", is separated into 4 words: "Application", "get", "is" and "Editor".

Understanding that principle is the key to mastering the suggestions shown by the *AutoComplete*; as it shows commands whose words starts by the capital letters present in the input field.

For instance, for the image above, typing "AIE" in the input field will show the command "Application.get\_isEditor" in the *AutoComplete* window.

## The History Window



The *History* window shows a list of the previously entered commands, and can be navigated using the arrow keys.

Pressing the arrow keys when the *History* window is closed (and when *AutoComplete* is closed too) can also be used to navigate the history back and forth.

## API

The *DevConsole* class is the main class for using the **DevConsole**, and it rests on the *SickDev.DevConsole* namespace. The *CommandSystem*, however, lays in the *SickDev.CommandSystem* namespace; and, as you'd also have to use that a lot, I decided to make a wrapper inside the *SickDev.CommandSystem* namespace for the class *DevConsole*. What that means is that you can access both the *CommandSystem* and the *DevConsole* class by only using:

```
using SickDev.CommandSystem;
```

*DevConsole* is a *Singleton*, and so this static property is the main entry point for the rest of its public interface.