

# Overview

This plugin will help you with debugging and prototyping various functionality in game. By creating simple commands you can run scripts or modify state of game much faster.

# Support

In case of questions or problems, you can contact me on:

<http://www.procedurallevel.com/>

<https://twitter.com/ProceduralLevel>

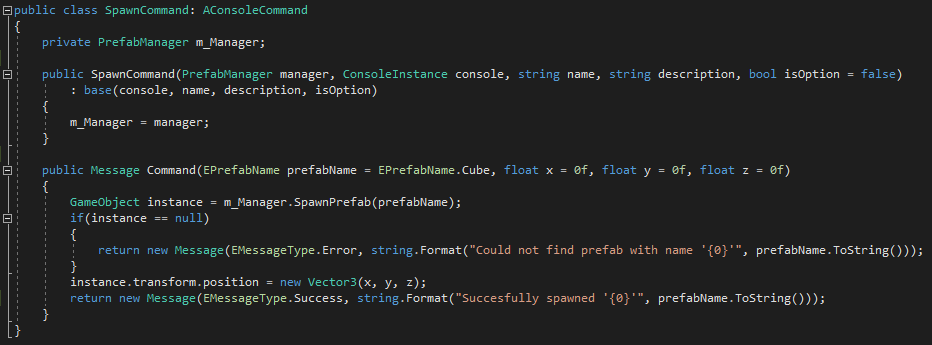
procedurallevel@outlook.com

# Setup

In order to start using Power Console in your project you need to place a “**PowerConsole**” prefab on scene. You can find it in “**ProceduralLevel/PowerConsole/Prefab**” folder. That’s it! While the prefab is on scene, you can enable console with “**~**” key or by toggling “**Active**” property on Console object.

# Quick Start

To Create a command you need to create a **class** that **inherits** from **AConsoleCommand.** In it, create a method “**Command**”. This will should contain logic of your command, and accept parameters that you want it to use.

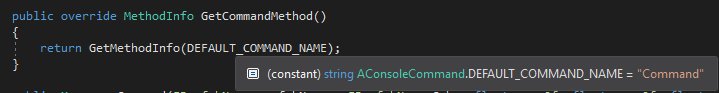


When It’s ready, you have to add it to **ConsoleInstance**. And that’s all, your command is ready to use!

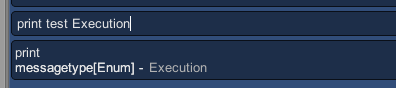


# Using different method name

You can use different command name if you don’t like default **“Command**”. Simply override **GetCommandMethod** and return reflection **MethodInfo** object with your method.



# Custom Hints



All primitive types and enums have hint support by default, but you can add your own, or even override existing one in per-command basis.

There are **2** types of hints. **ACollectionHint** and **ADynamicHint**.

You can create your own hint by inheriting any of those 2 classes.

**ACollectionHint** is used when all possible options are known before hand, for example list of string values or an enum. All you need to do is implement **GetAllOptions** method which should return an array with options.



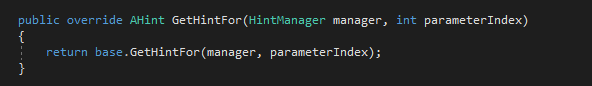
**ADynamicHint** is used when values may change, or there are simply too many to create a list. For example numbers. For this one you will need to implement two methods.



Value is equal to current value, so for example if current hint is “**1**”, this will be a value, and **PrevHint** should return “**0**” and **NextHint** should return “**1**” in case of **Integer** **hint**.

# Overriding Hints

To use custom hints for types, you command that will use it, will have to override **GetHintFor**. This should return a default hint object if not overridden (return call base implementation for this), or your custom object. Hints should be reused between all commands for performance reason!

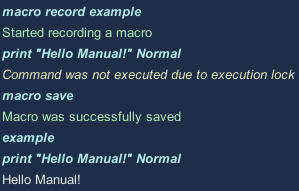


# Macro

Macro is a special command that enables recording of other commands and replaying them later on.

You start by typing “**macro record [name]**”. This will disable execution of commands and set console in record state. All commands you type from now on will be a part of macro.

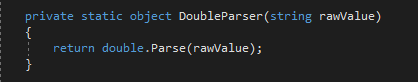
When you are done simply type “**macro save**”, and from now on, you can use macro like a normal command by calling it’s **[name]**.



# Adding supported variable types

To add parsing support for a new type, like a struct or class you will need to create a parser.

Parser receives a string input and should return desired type. Exceptions are handled on higher level so you don’t have to worry about parsing errors.



To add a parser simply call “**AddParser**” method on **ValueParser** which you can find in **ConsoleInstance**.



For reference please check “**ValueParser.cs**” class.

# Input

To change default keys for various input actions, or add new, modify “**ConsoleInput.cs**” class. You will find there a list of possible actions with associated keys.

