The Ingame Developer Console

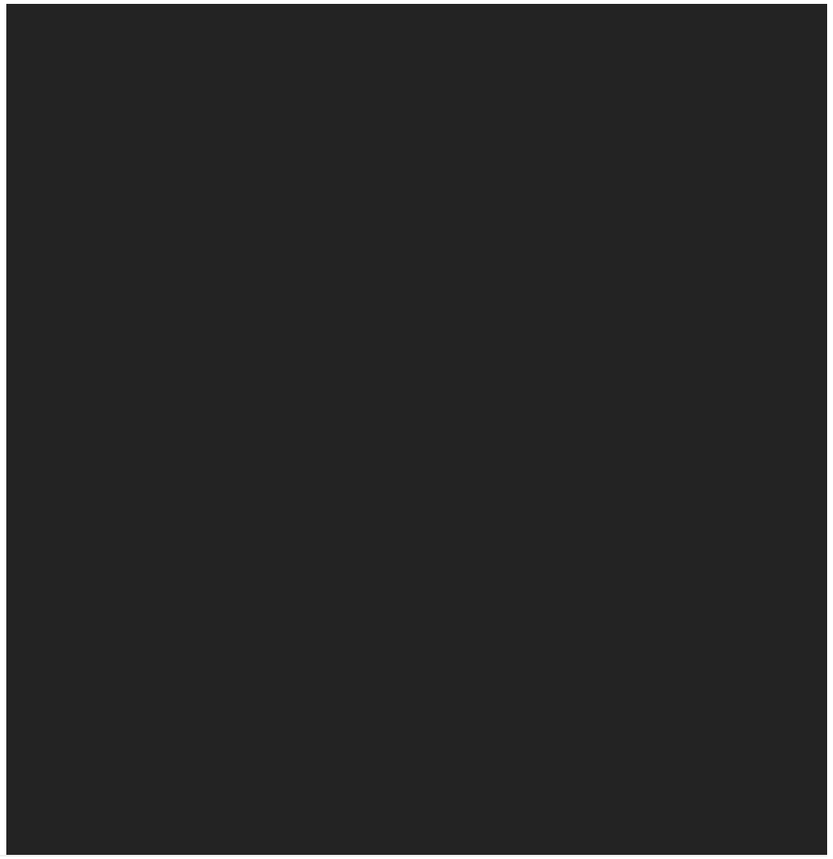
Introduction

The Ingame Developer Console (IDC) is a super powerful and easy to use ingame console/terminal, which allows you to speed up your game's development, help with debugging, and even add cheats to your game!.

After registering your C# classes with the IDC, you can turn methods into new console commands (called 'cmds') with full autocomplete and type checking using a single line of code!

Your cmds can have any number of parameters and any parameter types, including classes and structs. On top of this, the IDC allows you to track and even change the values of variables through the 'Vars Window'.

You can find a setup video https://www.youtube.com/watch?v=zmzn-F0V6OA



Adding a Command

How to add your own cmd to the IDC

The IDC allows you to add any **method** as a command you can run from the terminal in-game. Methods you use as cmds can be public or private, static or not.

Just keep in mind that if your cmd is not static then it will be called once per class instance, while static cmds will only be called once. So if you have 10 enemies each with a `Kill` method, then running the `Kill` cmd will call it 10 times, once on each enemy.

Your new cmds will be added to the IDC when your class is registered (for example on `Start()`). Registered classes can be MonoBehaviours or normal C# classes, and normal C# classes can also be static.

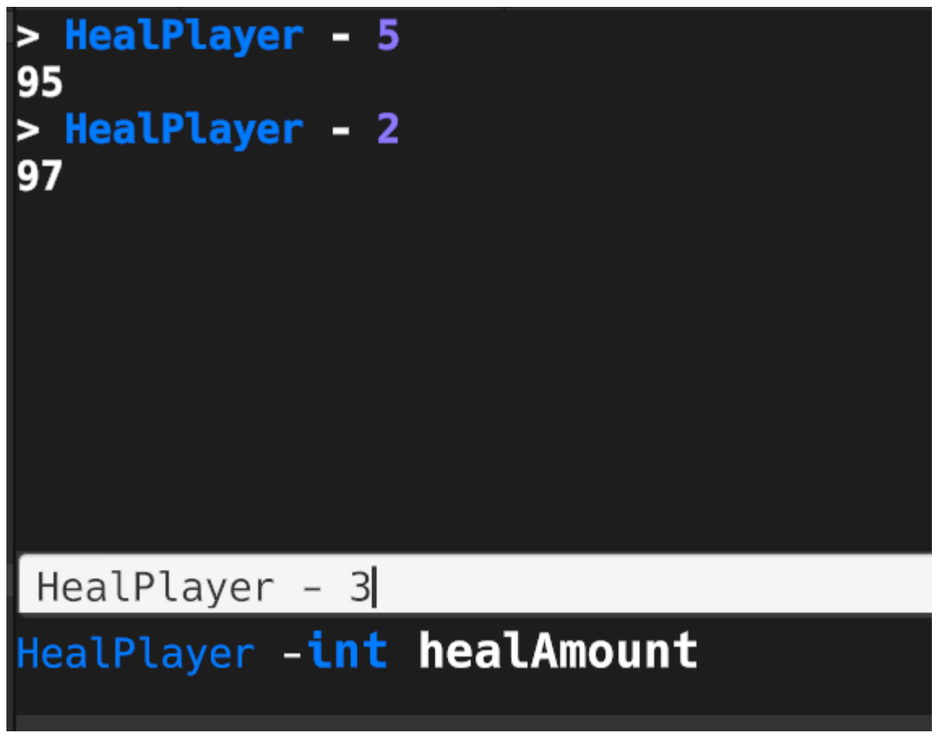
There is no need to unregister classes as The IDC will automatically detect when a

Example 1 Example 2 Example 3 Example 4

```
using IDC;  //The IDC namespace is always
required

class Player : MonoBehaviour
{
    int health;
    public int maxHealth = 100;

    void Start()
    {
        health = maxHealth;

        //Remember to register your classes!
        IDCUtils.IDC.AddClass(this);
    }

    //Since no cmd name is given, the IDC
```

class is no longer used in the game and will remove it when the Garbage Collector runs.

The following cmd changes the player health and returns the new value, which is then printed to the IDC.



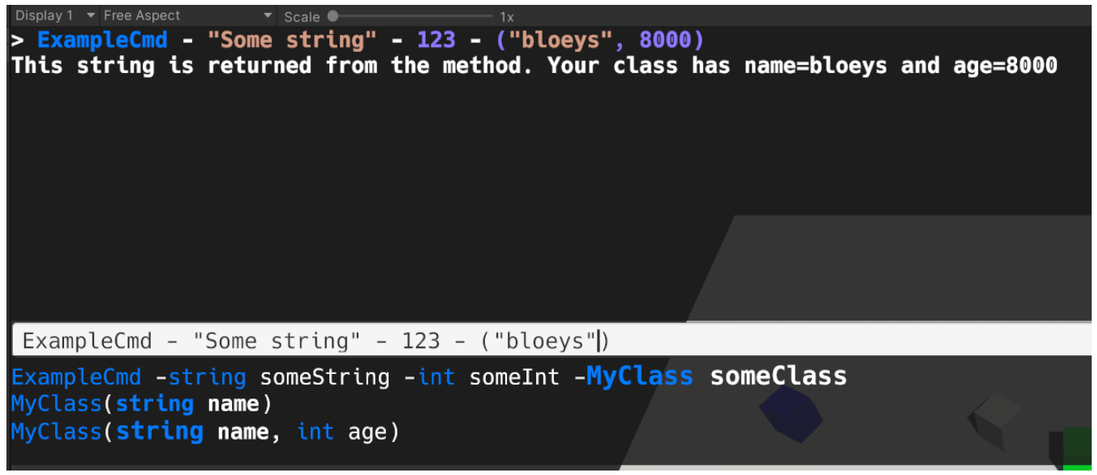

```
    //will use the method name 'HealPlayer' as
the cmd name
    [IDCCmd]
    public int HealPlayer(int healAmount)
    {
        health += healAmount;

        if (health > maxHealth)
            health = maxHealth;

        //We return the new health. This will be
shown on the console!
        return health;
    }
}
```

While typing a command's name you will see suggestions and can choose one by highlighting it with the arrow keys then pressing '**TAB**' to autocomplete it. Once a command's name is fully typed you will see all the parameters of the command that you need to pass.

Each parameter is entered by putting a dash '**-**' followed by a space and then the value you want. If the parameter is a string you can either put double quotes or not. If you want to put double quotes inside your string then escape it with '\'. For example **"My special "string" that has double quotes"**.

If your parameter is a class or a struct then you need to use brackets '**()**'. If you want to pass parameters to the class/struct constructor then put them between the brackets, separated by commas.

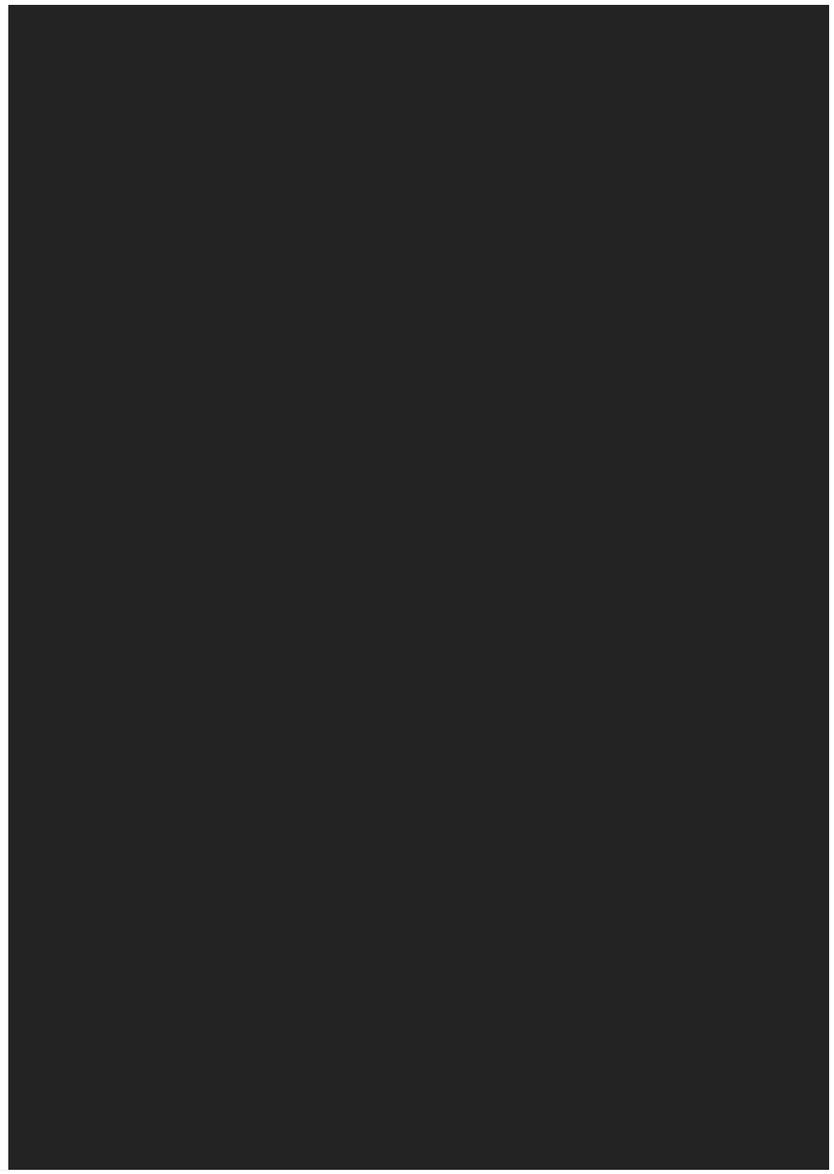
Returned values from your commands (if any) are printed to the console.



Arrays and arrays of classes are also supported by using square brackets '**[]**', and within the square brackets enter the value as if it was single parameter.

> Always remember to register your classes, otherwise your IDC cmds and variables will not be picked up.

> Cmd names **must** contain only letters, numbers, and underscore.



Adding a Variable

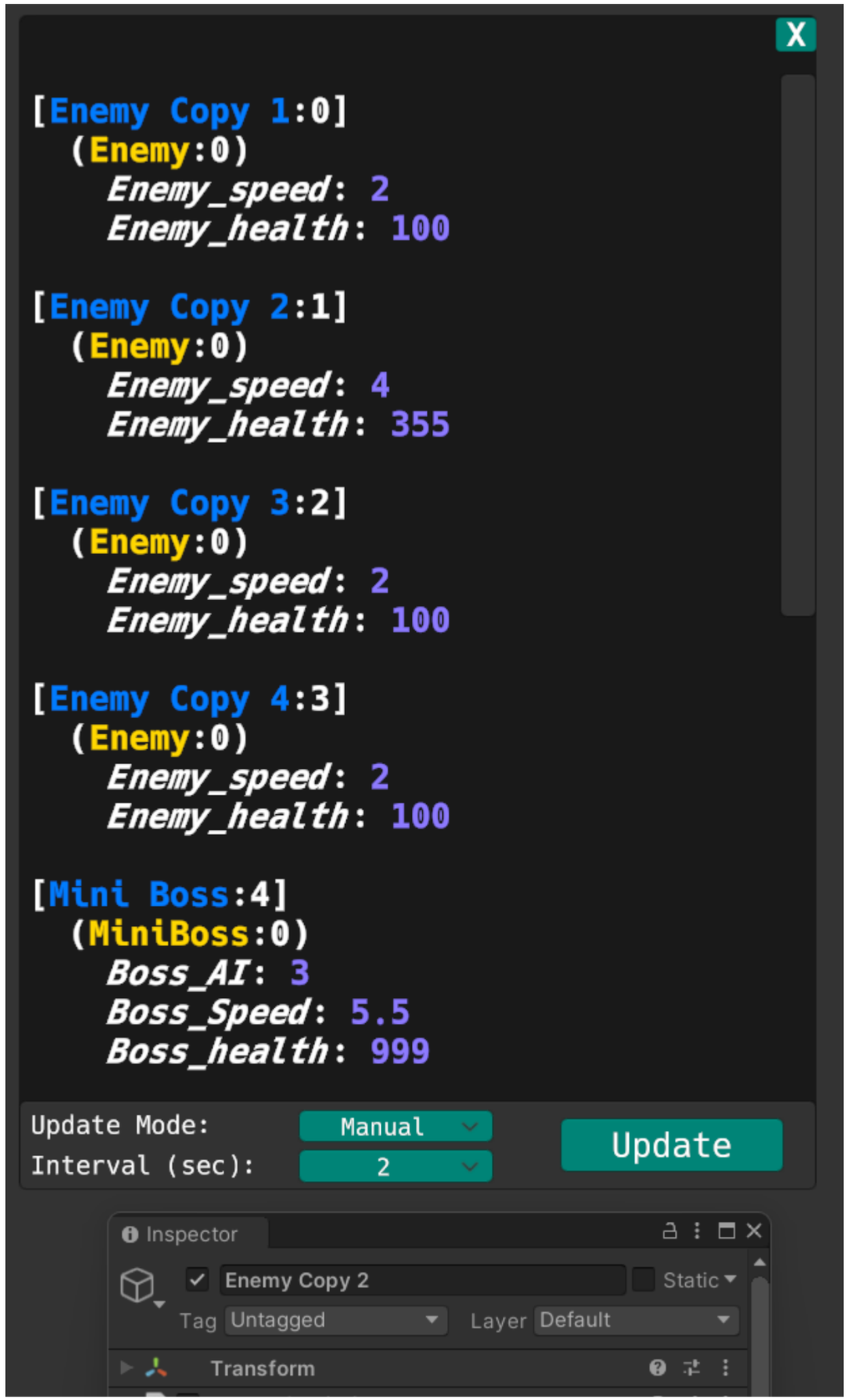
How to track your variables with the IDC

The IDC allows you to both track your variables and to edit their values through the 'Vars Window'. To open the vars window run the `ShowVarsWindow` IDC cmd.

The window shows you a list of game objects, and under each object the registered classes that are attached to that game object, and under each class a list of registered variables.

Example 1 Example 2 Example 3

```
using IDC;

class Player : MonoBehaviour
{
    public int maxHealth = 100;

    //Just like IDCCmd, this will use the
    //variable's name if no name is given
    [IDCVar]
```

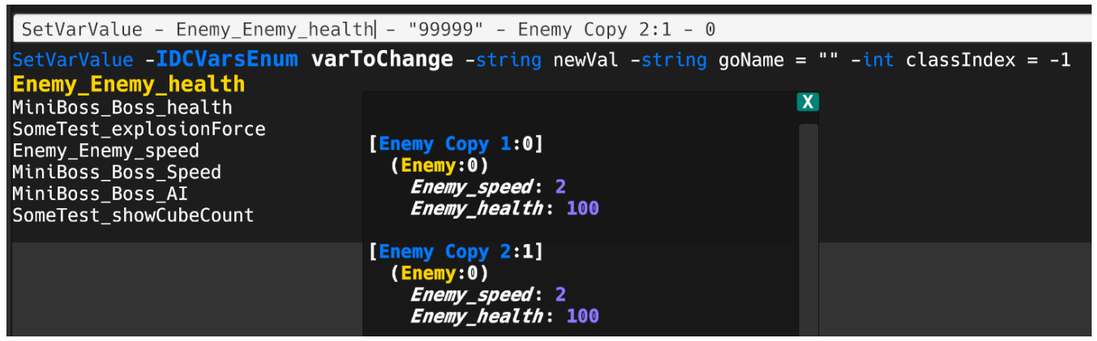



```
    int health;

    void Start()
    {
        health = maxHealth;

        //Always remember to register
        IDCUtils.IDC.AddClass(this);
    }

    [IDCCmd]
    public void HealPlayer(int healAmount)
    {
        health += healAmount;

        if (health > maxHealth)
            health = maxHealth;
    }
}
```

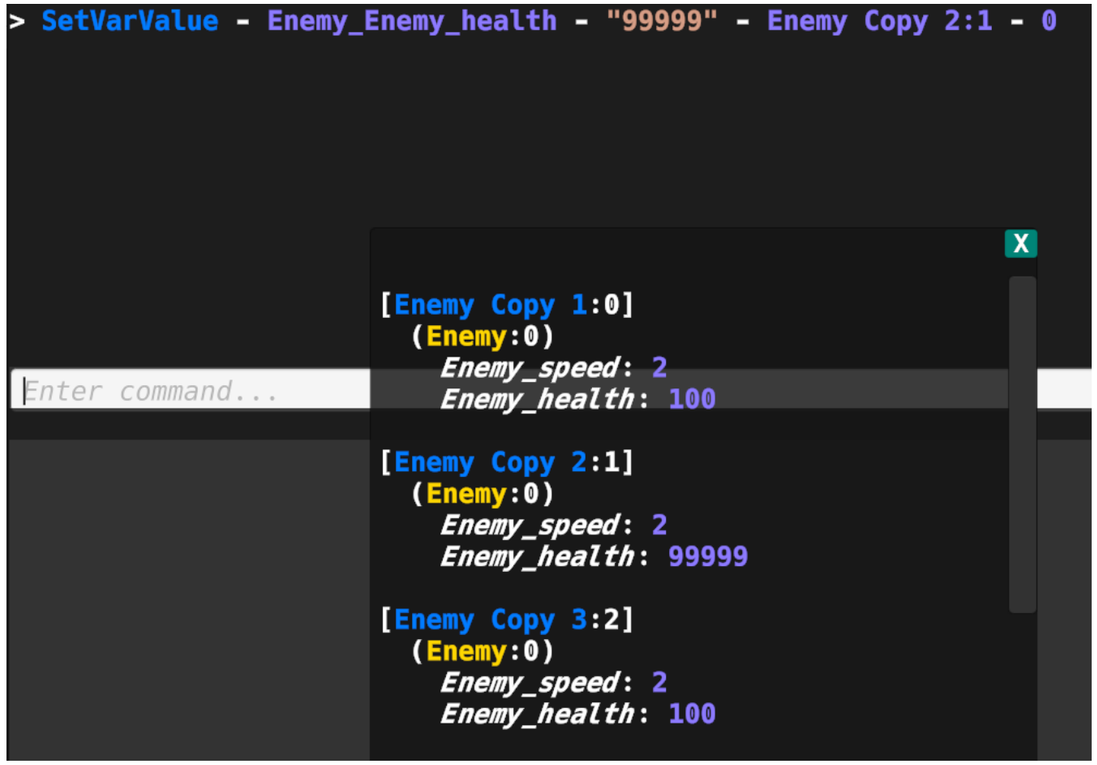
> The vars window can be moved by dragging it, and resized from the bottom-right corner of the window.

> Just like cmds, variable names **must** contain only letters, numbers, and underscore.

You can also change the value of an IDC Var by using the `SetVarValue` cmd. This cmd takes the name of the variable and its new value, along with the game object and class to change the variable on.



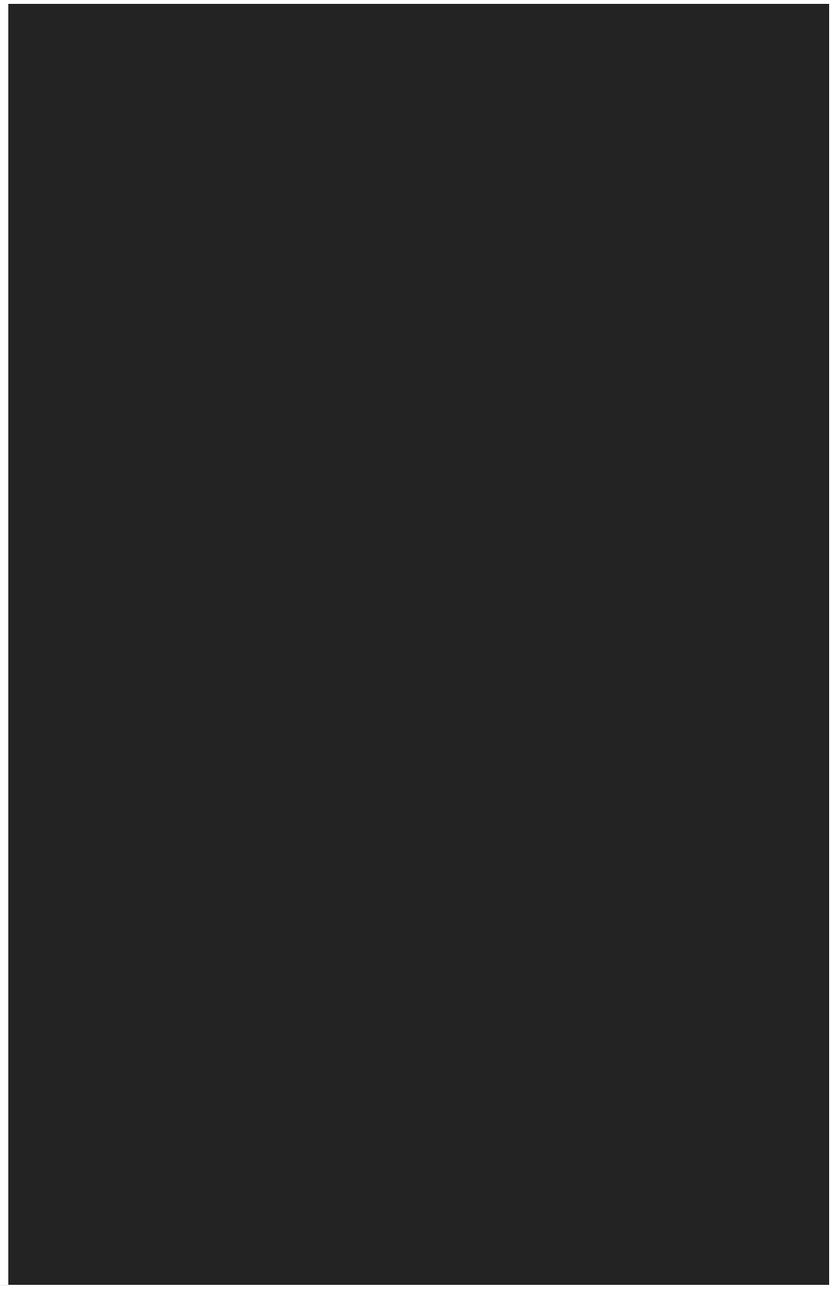
After running the previous cmd the health on ‘Enemy Copy 2’ will become 9999.



If you don't select a game object and a class then all instances of variable are changed.

> The class index is the number after the class name. In the above image, '(Enemy:0)' means the class called 'Enemy' and an index of zero.

> If your class is used on multiple game objects, you must use the index on the game object you are choosing



IDCParam Attribute

Adding custom suggestions for a cmd parameter

You can use the `IDCParam` attribute on method parameters to show additional suggestions in the console.

Example 1 Example 2

```
using UnityEngine;
using IDC;

public class NPC : MonoBehaviour
{
    string NPCType;
```

These suggestions also support Rich Text, and so properties like color can be controlled if wanted. The list of supported tags can be found here.

> User provided suggestions get shown before IDC provided ones


```
    void Start()
    {
        IDCUtils.IDC.AddClass(this);
    }

    //These three options will be shown as
suggestions
    [IDCCmd]
    void SetNPCType([IDCParam("Friendly",
"Passive", "Enemy")] string npcType)
    {
        NPCType = npcType;
    }
}
```

IDC Enums

IDCCmdsEnum and IDCVarsEnum

The IDC generates two enums, ‘IDCCmdsEnum’, which contains the names of all your IDC cmds, and ‘IDCVarsEnum’, which contains the names of IDC variables and their classes.

They are used to show you better suggestions and can be used to run IDC cmds from code. For example you can run an IDC cmd through the `IDCUtils.IDC.RunCmd` method, which requires the IDCCmdsEnum.

> If you disabled auto-regenerate in the settings then you should click the ‘Update IDC Enums’ button in the IDC prefab whenever you add/remove an IDC cmd or var.

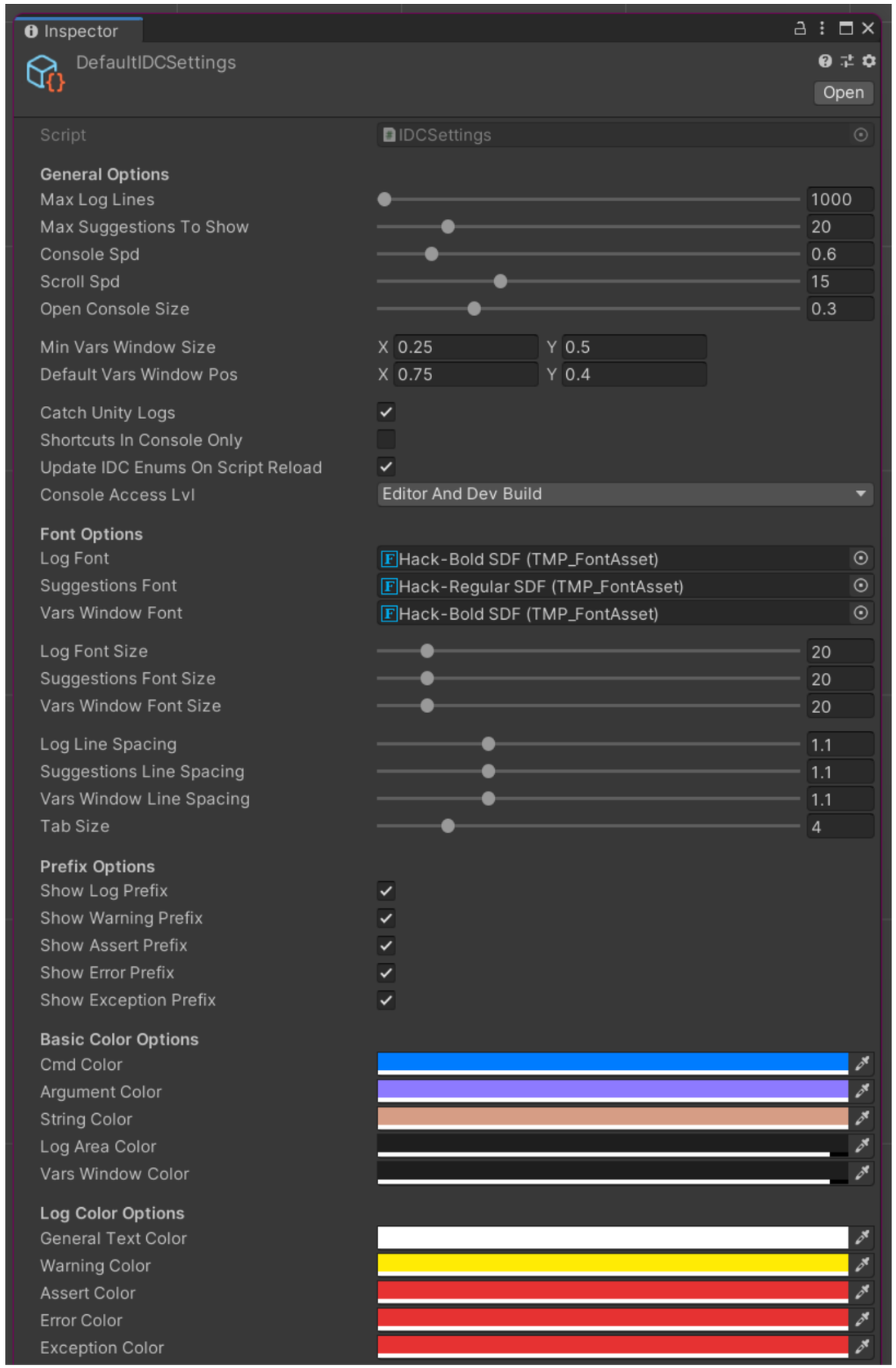
IDC Settings

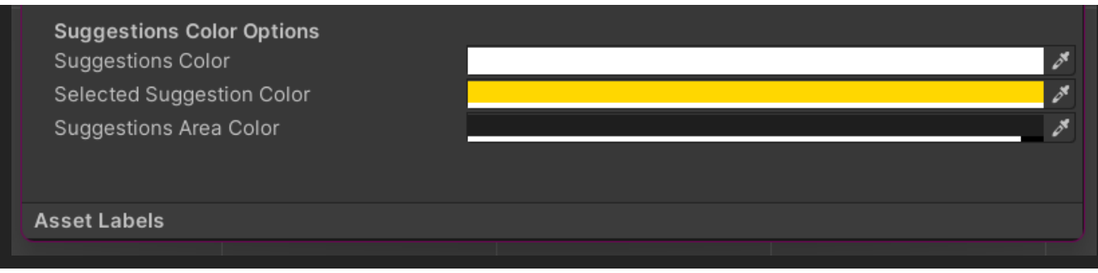
Configuring the function and look of the IDC

All of the possible IDC configuration, including fonts, colors and functional things like speed and toggle keys are controlled by an ‘IDCSettings’ scriptable object.

The default settings asset is stored in the ‘Ingame Developer Console/Settings’ folder, which you can modify to your liking. Additionally, you can create more settings assets by clicking the ‘Create New IDC Settings’ button on the IDC prefab.

Multiple setting assets allow you to easily have different configurations for different use cases or preferences (e.g. IDC on mobile devices). Settings can either be placed on the IDC prefab directly or changed ingame by using the ‘SetIDCSettings’ cmd.





Shortcut Profiles

Assigning shortcuts to your cmds

Shortcut profiles are very similar to settings, they are simply scriptable objects that contain shortcut assignments. Again, this allows you to have different sets of shortcuts that you can simply swapout.

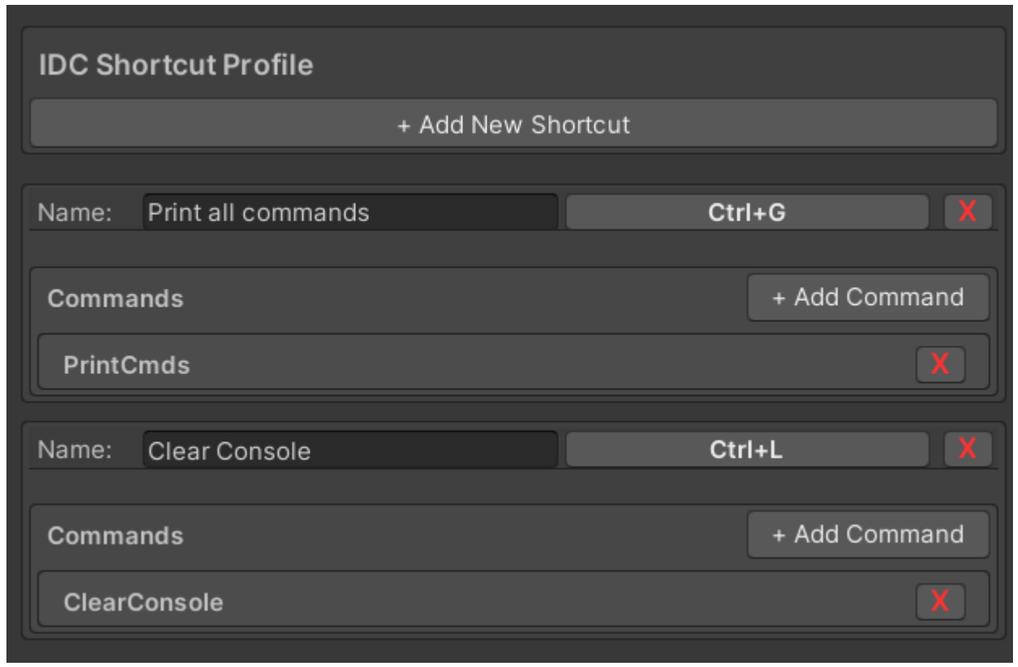
Shortcut profiles can be changed by assigning them directly to the IDC prefab or by using the ‘SetIDCShortcutProfile’ cmd.

By default, shortcut profiles are stored in ‘Ingame Developer Console/Shortcut Profiles’. New profiles can be created by clicking the ‘Create New IDC Shortcut Profile’ button on the IDC prefab.

In the image below, you can see an example of a single cmd attached to the ‘LeftCtrl + L’ shortcut.

> Multiple cmds can be attached to one shortcut.

> Shortcut cmds **must** take **zero** arguments.



AddClass

Registers the class instance with the IDC

Parameters

object *classInstance* The class instance to register

Adding a class registers all of its commands and variables with the IDC.

> The class will automatically be removed from the IDC when it's garbage collected (e.g. When its GameObject is destroyed)

> Adding a class in the **Awake** method of a GameObject that is loaded at scene start might fail. If your object is available from scene start then you should add it in the **Start** method.

Example

```
void Start()
{
  IDCUtils.IDC.AddClass(this);  //Adds the current class to the IDC
}
```

AddStaticClass

Registers a static class with the IDC

Parameters

Type *classType*	The type of the class to register

Adding a class registers all of its commands and variables with the IDC.

> Adding a class in the **Awake** method of a GameObject that is loaded at scene start might fail. If your object is available from scene start then you should add it in the **Start** method.

Example

```
void Start()
{

IDCUtils.IDC.AddStaticClass(typeof(MyStaticClass
));
}
```

RunCmd

Runs an IDC command from a passed string

Parameters

IDCCmdsEnum *cmd*	The command to be executed
string *args*	The args passed to the cmd

The args string **must** be in the same format as commands entered in the IDC UI while in-game.

Example 1 Example 2

```
IDCUtils.IDC.RunCmd(IDCCmdsEnum.SetCmdColor, " -
red");
```

SetParamSuggestions

Sets the user suggestions for a command parameter

Example

```
//Show the following values as suggestions for
the 'lineSpacing' parameter of the
```

Parameters

IDCCmdsEnum *cmd*	The name of the command that contains the wanted parameter
string *paramName*	The name of the parameter
string[] *newSugg*	An array of suggestions to show for this parameter when using the UI

This is essentially a runtime version of the `IDCParam` attribute, which allows you to override the suggestions of `IDCParam`.

Just as with the attribute, suggestions support Rich Text and the list of supported tags can be found here.

> User provided suggestions get shown before IDC provided ones

```
'SetLogAreaLineSpacing' command
string[] mySuggestions = new string[] { "1",
"1.2", "1.4", "1.6" };

IDCUtils.IDC.SetParamSuggestions(
  IDCCmdsEnum.SetLogAreaLineSpacing,
  "lineSpacing", mySuggestions);
```

SetParamSuggestionsFunc

Takes a function that will be called when suggestions are needed for a parameter

Parameters

IDCCmdsEnum *cmd*	The name of the command that contains the wanted parameter
string *paramName*	The name of the parameter
Func<string[]> *func*	The function to be called when the IDC needs suggestions

Similar to **SetParamSuggestions**, but instead of hard-coding suggestions you provide a function to the IDC that gets called automatically by the IDC when it

Example

```
void Start()
{
  IDCUtils.IDC.SetParamSuggestionsFunc(
    IDCCmdsEnum.SetLogAreaLineSpacing,
    "lineSpacing",
    GetLogLineSpacingSugg);
}

string[] GetLogLineSpacingSugg()
{
  //Return a random linespacing between 1-2 as a
suggestion
  string lineSpacing = Random.Range(1f,
2f).ToString();
  return new string[] { lineSpacing };
}
```

needs suggestions for a param. This function returns an array of suggestions that get shown along with the automatic ones.

This is useful when you want to provide dynamic suggestions that change depending on the conditions of your game.

> The passed function is only called when the user is being shown the suggestions of the parameter targeted by this method.

Log

Logs the passed message to the IDC console

Parameters

string *msg*	The message to log to the console

Logs a string to the IDC console directly and does not show on the Unity console. Supports Rich Text.

IDC VAR

A variable that provides access to the running IDC instance

The **IDC** variable is a static variable in the IDCUtils class that has a reference to the IDC instance, and is the main way to use the IDC asset.

> Trying to use this variable at **Awake** when the scene first starts is *undefined* as the reference might not have been loaded yet.

Example Example 2 Example 3 Example 4

```
IDCUtils.IDC.Log("My log message");
```

ColorString

Surrounds the string with the rich text tag of the wanted color

Parameters

string *s*	The string to surround with the rich text tag
Color *c*	The color to use for the passed string

Converts the passed RGBA color to the equivalent Hexadecimal representation and uses that result in the rich text tag. Returns the original string but surrounded with the appropriate color tag.

Example

```
string myColoredString =
IDCUtils.ColorString("Hello World!",
Color.green);
```

BoldString

Surrounds the string with the 'b' rich text tag

Parameters

string *s*	The string to surround with the rich text tag

Returns the original string but surrounded with the **b** tag.

Example

```
string myBoldString = IDCUtils.BoldString("Hello
World!");
```

ItalicString

Surrounds the string with the 'i' rich text tag

Parameters

string *s*	The string to surround with the rich text tag

Example

```
string myItalicString =
IDCUtils.ItalicString("Hello World");
```

Template by CloudCannon

Returns the original string but surrounded with the **i** tag.

SetStringSize

Surrounds the string with the 'size' rich text tag

Parameters

string *s*	The string to surround with the rich text tag
float *newSize*	The new size of the passed string

Returns the original string but surrounded with the **size** tag set to the passed size.

Example

```
string myLargeString =
IDCUtils.SetStringSize("Hello IDC!", 25.5f);
```