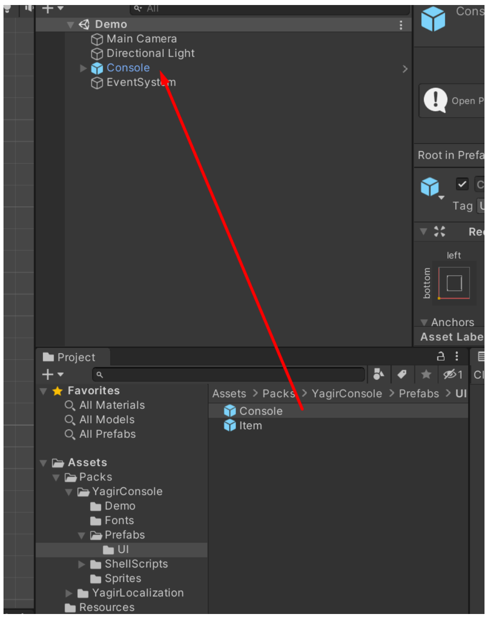
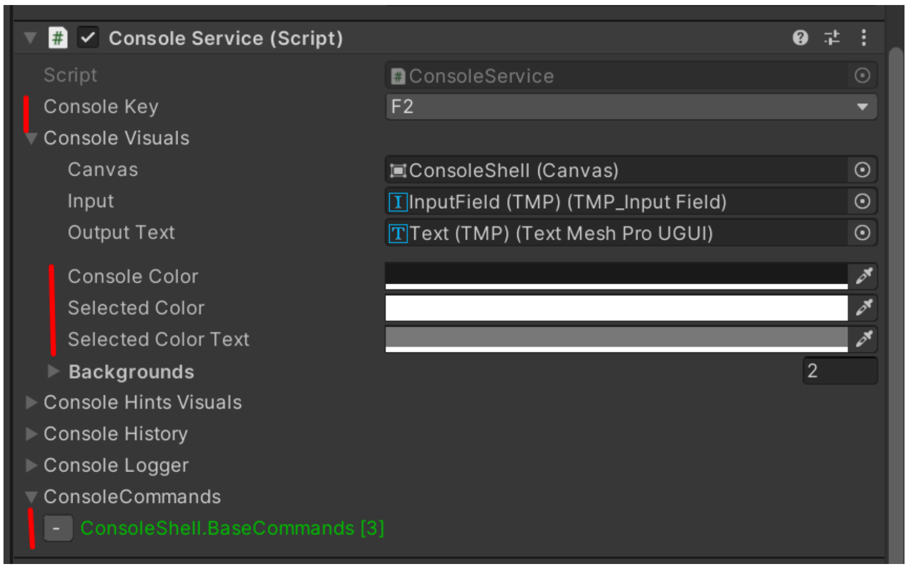
Initialization

In order to initialize the console, you need to drag the Console prefab onto the stage.



After opening the prefab, you can select the namespaces of the commands that are relevant to you in the ConsoleCommands section.
+ adds a list of commands from namespace
- removes all namespace commands from the console.

You can also change the color of the console and the activation button according to the standard - **F2**



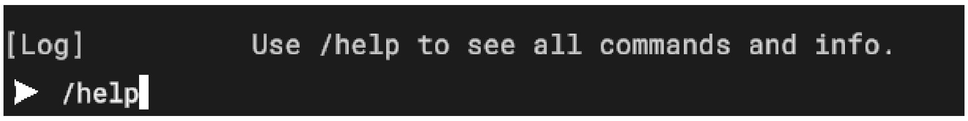
Console Service (Script)

Script	ConsoleService
Console Key	F2
Console Visuals	
Canvas	ConsoleShell (Canvas)
Input	InputField (TMP) (TMP_Input Field)
Output Text	Text (TMP) (Text Mesh Pro UGUI)
Console Color	■
Selected Color	□
Selected Color Text	▨
Backgrounds	2
Console Hints Visuals	
Console History	
Console Logger	
ConsoleCommands	

- ConsoleShell.BaseCommands [3]

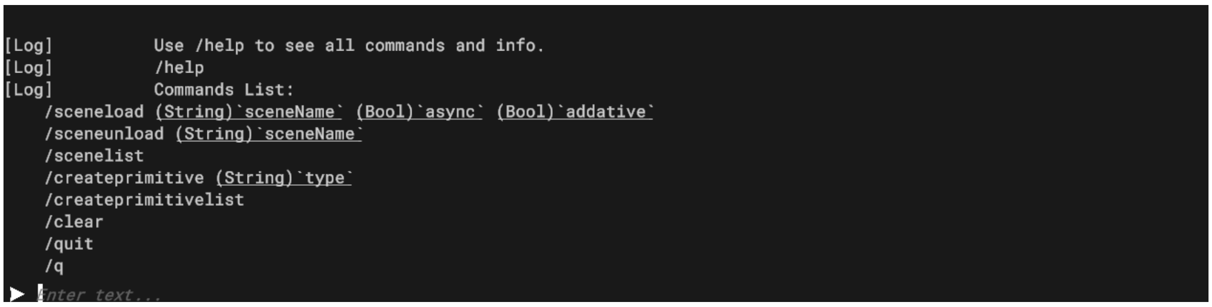
Usage

In order to find out the list of commands and their arguments, you need to enter the command /help
All commands begin with a "/" character



```
[Log]          Use /help to see all commands and info.
▶ /help
```

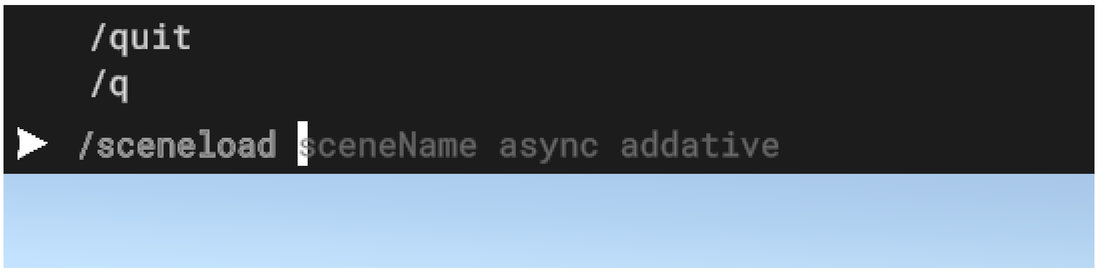
After you press Return/Enter button:



```
[Log]          Use /help to see all commands and info.
[Log]          /help
[Log]          Commands List:
    /sceneload (String)`sceneName` (Bool)`async` (Bool)`addative`
    /sceneunload (String)`sceneName`
    /scenelist
    /createprimitive (String)`type`
    /createprimitivelist
    /clear
    /quit
    /q
▶ Enter text...
```

Each command will be output here. Even your commands that you add. This command generates this text.

Console highlights command arguments:

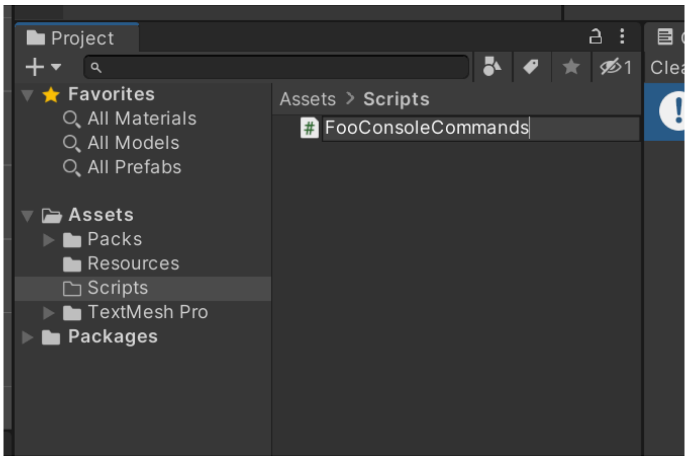


```
    /quit
    /q
▶ /sceneload sceneName async addative
```

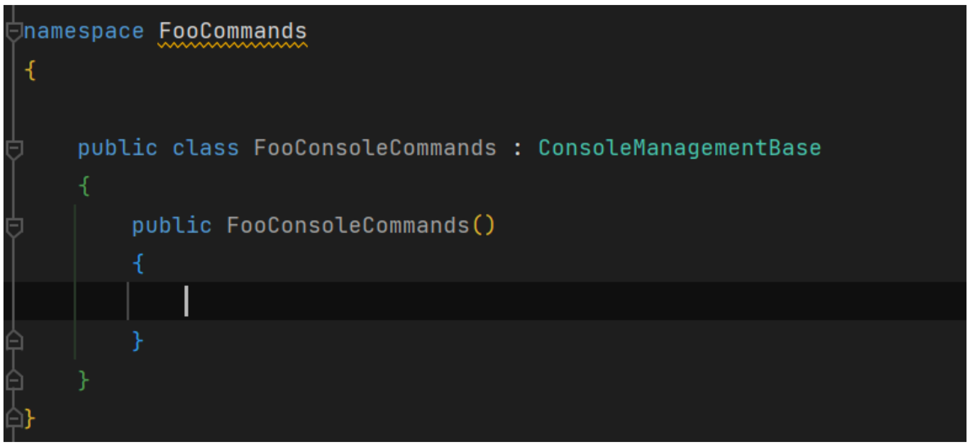
You can view the types of input arguments in /help.

Creating your own console commands

1. Create a class or open a ready-made one

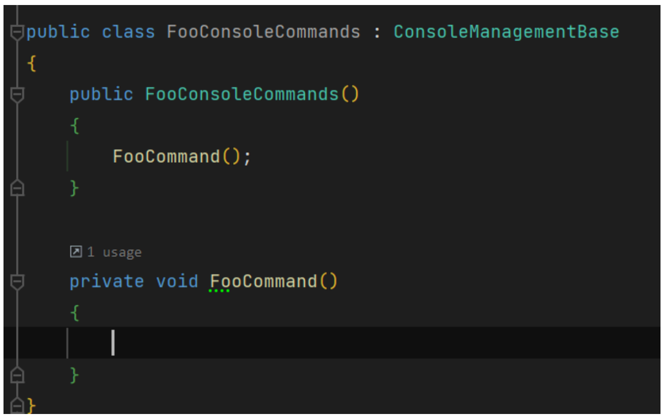


2. We inherit our class from the ConsoleManagementBase type and create an empty constructor. Write the type into namespace **FooCommands**



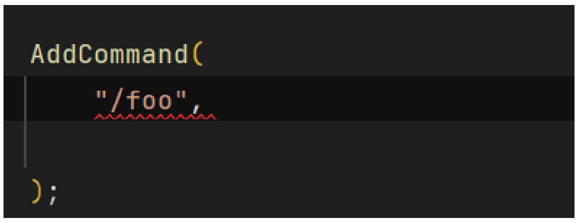
```
namespace FooCommands
{

    public class FooConsoleCommands : ConsoleManagementBase
    {
        public FooConsoleCommands()
        {

        }
    }
}
```

3. In the constructor we call the method for creating a console command.



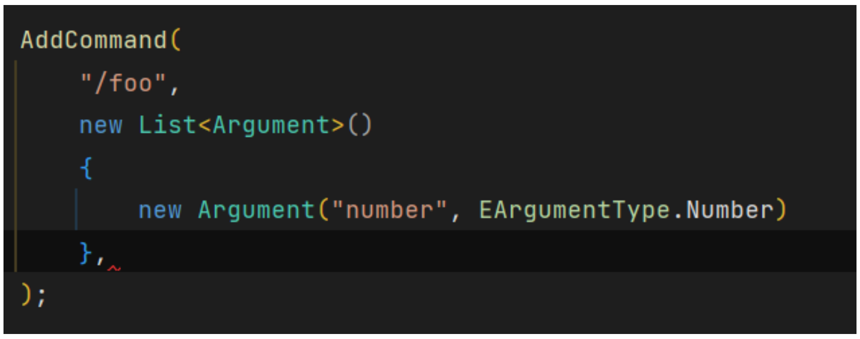
```
public class FooConsoleCommands : ConsoleManagementBase
{
    public FooConsoleCommands()
    {
        FooCommand();
    }

    1 usage
    private void FooCommand()
    {

    }
}
```

4. We enter the command as the first argument.

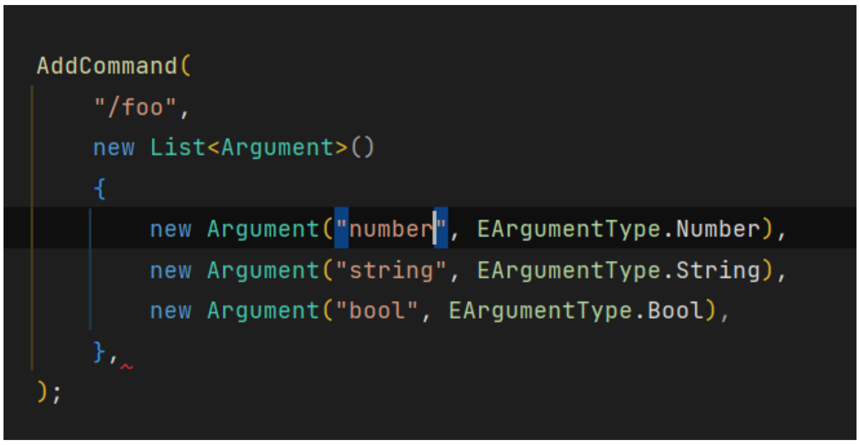


```
AddCommand(
    "/foo",

);
```

5. Enter a list of arguments. The name of the argument, the type of the argument, whether it is required, and also set the standard values if the argument is not required for input.

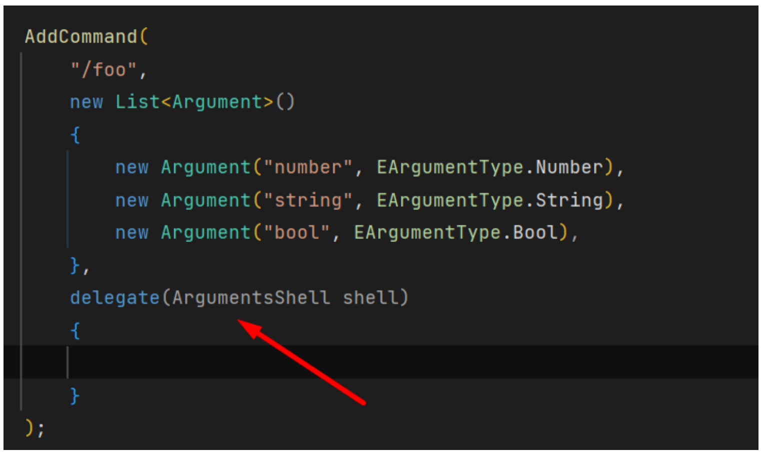


```
AddCommand(
    "/foo",
    new List<Argument>()
    {
        new Argument("number", EArgumentType.Number)
    },
);
```



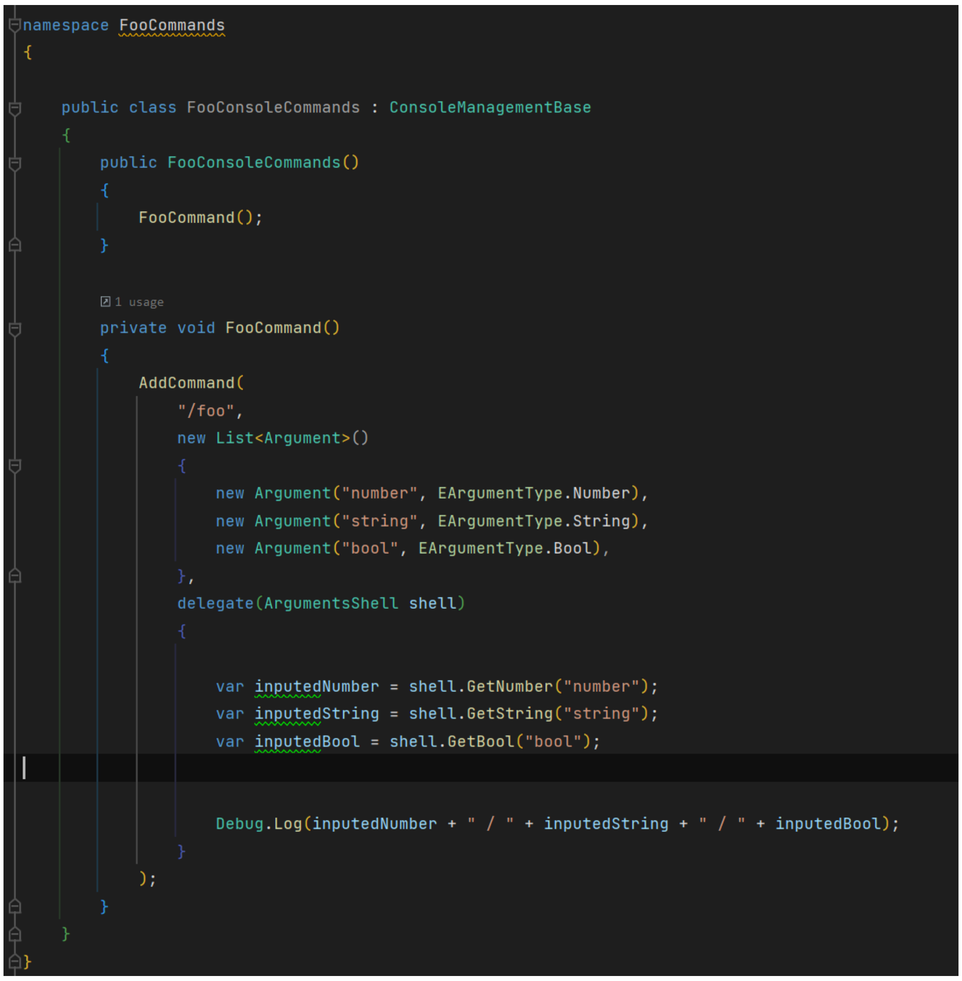
```
AddCommand(
    "/foo",
    new List<Argument>()
    {
        new Argument("number", EArgumentType.Number),
        new Argument("string", EArgumentType.String),
        new Argument("bool", EArgumentType.Bool),
    },
);
```

6. We create a delegate, and in it, we write the logic that will occur during the execution of the command



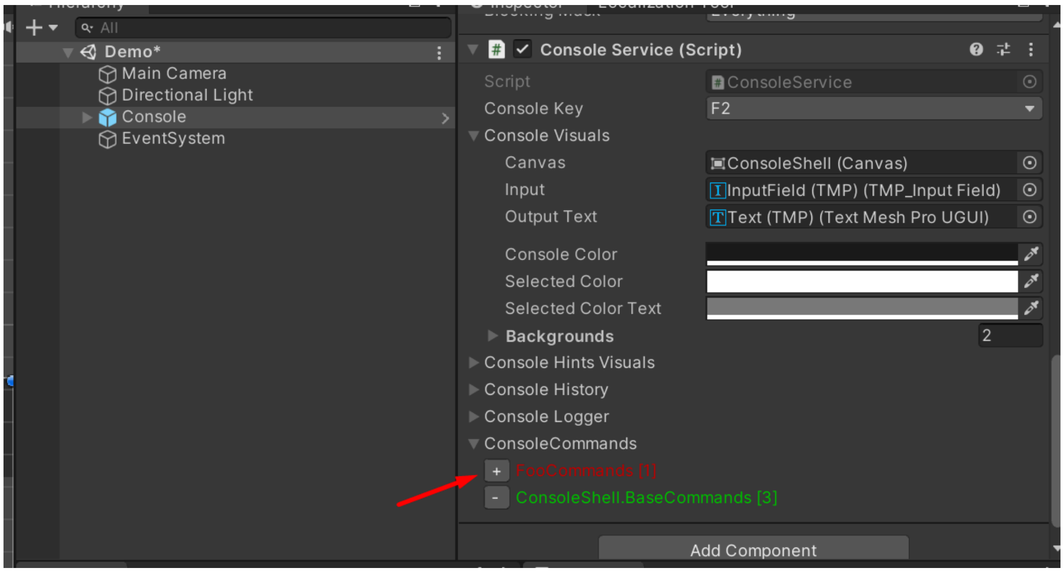
```
AddCommand(
    "/foo",
    new List<Argument>()
    {
        new Argument("number", EArgumentType.Number),
        new Argument("string", EArgumentType.String),
        new Argument("bool", EArgumentType.Bool),
    },
    delegate(ArgumentsShell shell)
    {

    }
);
```

7. Now through Shell we can get arguments by name and output them to the console via Debug.Log



```
namespace FooCommands
{

    public class FooConsoleCommands : ConsoleManagementBase
    {
        public FooConsoleCommands()
        {
            FooCommand();
        }

        1 usage
        private void FooCommand()
        {
            AddCommand(
                "/foo",
                new List<Argument>()
                {
                    new Argument("number", EArgumentType.Number),
                    new Argument("string", EArgumentType.String),
                    new Argument("bool", EArgumentType.Bool),
                },
                delegate(ArgumentsShell shell)
                {

                    var inputedNumber = shell.GetNumber("number");
                    var inputedString = shell.GetString("string");
                    var inputedBool = shell.GetBool("bool");


                    Debug.Log(inputedNumber + " / " + inputedString + " / " + inputedBool);
                }
            );
        }
    }
}
```

8. Now add a file with commands to the console

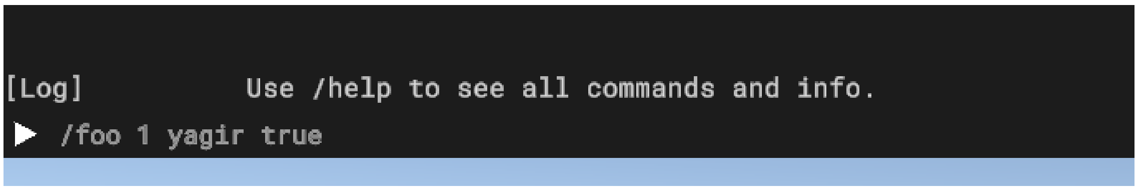


Demo*
Main Camera
Directional Light
Console
EventSystem

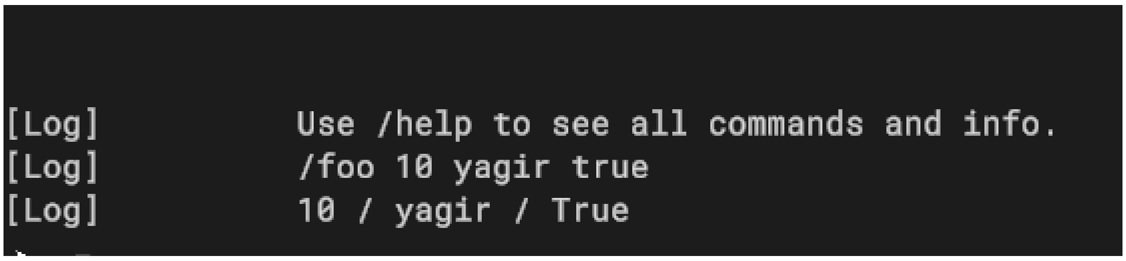
Console Service (Script)

Script: ConsoleService
Console Key: F2
Console Visuals
Canvas: ConsoleShell (Canvas)
Input: InputField (TMP) (TMP_Input Field)
Output Text: Text (TMP) (Text Mesh Pro UGUI)
Console Color
Selected Color
Selected Color Text
Backgrounds: 2
Console Hints Visuals
Console History
Console Logger
ConsoleCommands
+ FooCommands [1]
- ConsoleShell.BaseCommands [3]

9. Using



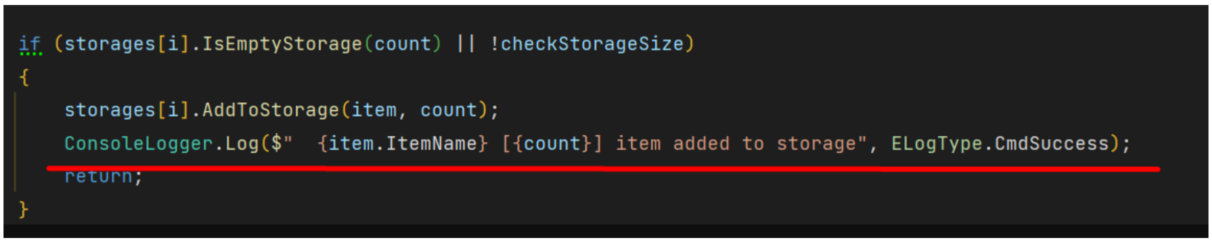
```
[Log]          Use /help to see all commands and info.
▶ /foo 1 yagir true
```



```
[Log]          Use /help to see all commands and info.
[Log]          /foo 10 yagir true
[Log]          10 / yagir / True
```

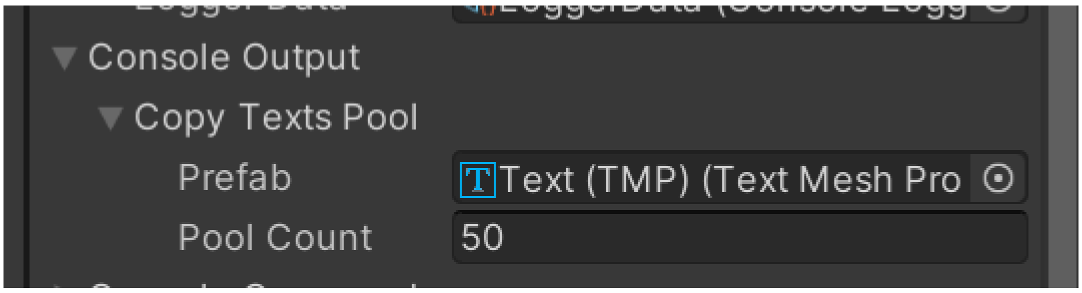
ConsoleLogger

A simple class that wraps UnityEngine.Debug.
It allows you to log to the console with custom types.



```
if (storages[i].IsEmptyStorage(count) || !checkStorageSize)
{
    storages[i].AddToStorage(item, count);
    ConsoleLogger.Log($"  {item.ItemName} [{count}] item added to storage", ELogType.CmdSuccess);
    return;
}
```

ConsoleOutput



```
▼ Console Output
    ▼ Copy Texts Pool
        Prefab          Text (TMP) (Text Mesh Pro
        Pool Count      50
```

Pool Count - number of texts to output to the console.