# EasyConsole

## Quick Start

1. Import the downloaded package.
2. Drag the “Console” prefab from the “EasyConsole/Prefabs” folder into the scene
3. Press Play
4. During playing, press the backquote ` key to open the console. (on qwerty keyboards it is generally next to the 1 key.)
5. Press esc to close it again

## Contact:

Web: http://homans.nhlrebel.com/
Email: homans@nhlrebel.com
Twitter: @sanderhoman

## General Information:

Download and import the EasyConsole package from the asset store. This should create a folder called EasyConsole containing a dll and various scripts. For ease of use, it also includes a console prefab that can be dragged into the scene directly.

The console consists out of 2 parts. In the dll there is the Console backend and in a separate script is the GUI frontend. Both parts need to be in the scene for the console to work, but not necessarily on the same object. Console is a singleton, and will give an error if a second object is in the scene with the console script.

The basic GUI script uses the unity gui system, whereas the NGUI script uses NGUI as the gui system. To replace those scripts to support a different GUI system, please read the section “Replacing the GUI”.

## Usage:

The usage of the console is relatively straightforward. You can open the console with the backquote key “`”. After that, you can type in commands in the input field that has appeared. To close the console again, press “esc”. Entered commands can take 2 forms.

1st. Command parameter1 parameter2

These are simple commands that are directly linked to a method defined in your scripts. By default the gui comes with a few commands, for example, “help” and “select”. Commands are case sensitive, so be careful about that. It is also possible to get more info about these commands by using the command “help command” replacing command with the command you want to know more about. E.g. “help select”. These commands are generally used for simple test commands, such as godmode or to spawn items.

2nd.  /object/object.component.method parameter1 parameter2

This will directly call a method on the given gameobject and component. The gameobject used, is defined by the “/object/object” part of the command. This is exactly the same format as used in the GameObject.Find methods in unity. To be able to use this on objects with spaces in their names, escape the space with a “\”. For example, “/Main\ Camera/object.comp.method”.

So for example, if you have a gameobject called “Player” which has a component called “Health”, you can call the function TakeDamage on Health with “/Player.Health.TakeDamage 10”. This will call the function and use 10 as the first parameter value.

To make it easier to perform multiple functions on the same component, the select command exists. This command selects a component to perform functions on so that you don’t have to type out the whole object hierarchy everytime. Usage is “select /object/object.component”. To deselect, execute “select” without any parameters.  To call a function on the selected object, just execute “methodName parameters” similar to a normal command.

## Built-in commands:

### help [command]
Shows a help message in the console. The help message contains basic instructions and a list of registered commands

### list
Lists all active root level  gameobjects

### listComponents [object]
lists all the components of the given gameobject

### printTree [object]
prints a tree of the given gameobjects children

### select [object.component]
Selects a component to easily execute multiple methods on the same object. Execute “select” without any parameters to deselect the component.

## Functions to use from code

### Print

```
public void Print(string line)
```

Prints the given line in the console. Can be used for debugging messages.

### Eval

```
public void Eval(string line)
```

Executes the given command as if entered in the console

## Customizing the console behavior:

### Adding a custom command

You can easily add a custom command by calling the RegisterCommand on the console instance.

```
Console.Instance.RegisterCommand("customCommandTest", this, "Command");
```

The first parameter is the name of the command. The second parameter is the object you want to invoke the function on. Most of the time this will the object you are calling RegisterCommand from, but you might want to choose a different object to invoke it on. The third parameter is the function name you want to be executed when the command is entered in the console.

The function itself is just a normal function.

C#

```
void Command(string param1)
```

JS

```
function Command (param1: String)
```

It can have return values, which will be printed in the console, and it can also have various parameters. To support your own types, read the "Adding a custom parser" section.

The function can also have an optional Help attribute.

C#

```
[Help("Usage: customCommandTest param\nA simple custom command test")]
```

JS

```
@Help("Usage: customCommandTest param\nA simple custom command test")
```

This help line is shown when the user does "help commandname" in the console. You can use the normal escape characters in there to format the text. E.g. "\n" for a newline and "\t" for a tab.

### Example 1a: a complete monobehaviour with a custom console command(C#)

```
using UnityEngine;
using Homans.Console;

public class CustomCommandTest : MonoBehaviour
{
    void Start()
    {
        Console.Instance.RegisterCommand("customCommandTest", this, "Command");
    }

    [Help("Usage: customCommandTest param\nA simple custom command test")]
    void Command(string param1)
    {
        Console.Instance.Print("Called with value " + param1);
    }
}
```

### Example 1b: a monobehaviour with a custom command(JS)

```
import Homans.Console;

// Use this for initialization
function Start () {
	Console.Instance.RegisterCommand("customCommandTest", this, "Command");
}

@Help("Usage: customCommandTest param\nA simple custom command test")
function Command (param1: String) {
	Console.Instance.Print("Called with value " + param1);
}
```

## Adding a custom parser

You can add custom parsers to the console to support custom types. By default EasyConsole has parsers for int, float and string. It also comes with an example parser for Vector3. Parsers always parse a string into your desired type. To add a parser to the console, call RegisterParser on the Console instance.

**Note:** Custom parsers can only be written in C# because Unityscript does not support ref/out parameters.

```
Console.Instance.RegisterParser(typeof(TestObject), ParseTestObject);
```

RegisterParser takes 2 parameters. The first needs to be the type the parser is going to parse to. The second is the parse method. The method definition should be:

```
bool ParseTestObject(string line, out object obj)
```

The return value determines if the parse action was successful. Return false when you are unable to parse the given string and true if the parsing was successful. The second parameter is an out parameter and should be set to resulting object, or null in case of failure.

## Example 2a: parsing a custom type(C#)

```
using UnityEngine;
using Homans.Console;

public struct TestObject
{
    public int x;
    public int y;
}

public class CustomParseTest : MonoBehaviour
{
    void Start()
    {
        Console.Instance.RegisterCommand("customParseTest", this, "Command");
        Console.Instance.RegisterParser(typeof(TestObject), ParseTestObject);
    }

    [Help("Usage: customParseTest param\nA simple custom parse test")]
    void Command(TestObject param1)
    {
        Console.Instance.Print("Called with value " + param1.x + ", " + param1.y);
    }

    public bool ParseTestObject(string line, out object obj)
    {
        string[] s = line.Split(',');
        if (s.Length != 2)
        {
            obj = null;
            return false;
        }

        int x;
        if (!int.TryParse(s[0], out x))
        {
            obj = null;
            return false;
        }
        int y;
        if (!int.TryParse(s[1], out y))
        {
            obj = null;
            return false;
        }

        TestObject result = new TestObject();
        result.x = x;
        result.y = y;
        obj = result;
        return true;
    }
}
```

## Replacing the GUI

The GUI used in EasyConsole can be easily customized and replaced if desired. The Console class provides various functions to make interact with it.

The most important function is:

```
public void Eval(string line)
```

This function will execute the given line. Normally, the input from a textfield is directly send to this function.

To display the data that is currently in the console log, you can use the "Lines" property. This property returns a CircularBuffer that contains the log history .

The command history is stored in the "Commands" property.

The console also provides a few convenience methods. To parse a line into its separate components, you can use

```
public static void parseCommand(string line, out string command, out string[]
gameobjectPath, out string componentName, out string methodName, out string[]
parameters)
```

This method splits out every component in the given line.

To only parse a gameobject string, you can use

```
public static void parseGameObjectString(string line, out string[] gameobjectPath,
out string componentName, out string methodName)
```

This method takes a string in the format "/Parent/Child/ChildChild.Component.Method". This can be used for example, to use a gameobject as a parameter for a function.

### Querying the scene

To get info about the scene, you can use one of the following methods:

```
public string[] GetGameobjectsAtPath(string path)
```

Returns children of the gameobject defined by the hierarchy in the path parameter, where the last entry in the path will be used as a filter. E.g. path "/Player/Blob/loc" will return all children of "/Player/Blob" that start with "loc".

```
public string[] GetComponentsOfGameobject(string path)
```

Returns all components of the gameobject defined by the hierarchy in the path parameter, where the last bit after "." will be used as a filter. E.g. path "/Player/Blob.Trans" will return all components of "/Player/Blob" that start with "Trans".

```
public string[] GetMethodsOfComponent(string path)
```

Returns all methods of the component defined by the hierarchy in the path parameter, where the last bit after the last "." will be used as a filter. E.g. path "/Player/Blob.Transform.get" will return all components of "/Player/Blob.Transform" that start with "get".