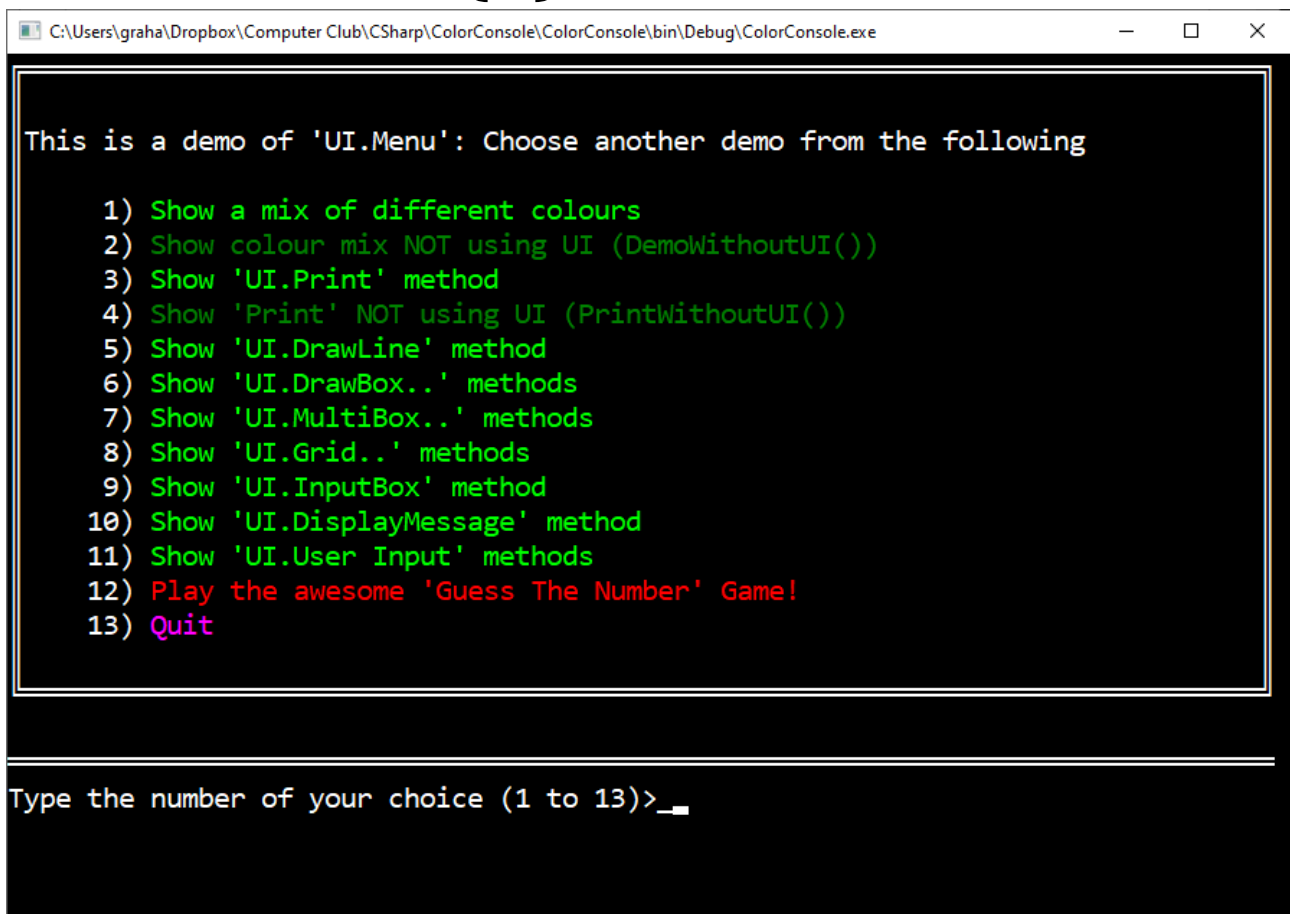


# Colo(u)r Console C#



```
C:\Users\graha\Dropbox\Computer Club\CSharp\ColorConsole\ColorConsole\bin\Debug\ColorConsole.exe

This is a demo of 'UI.Menu': Choose another demo from the following

1) Show a mix of different colours
2) Show colour mix NOT using UI (DemowithoutUI())
3) Show 'UI.Print' method
4) Show 'Print' NOT using UI (PrintWithoutUI())
5) Show 'UI.DrawLine' method
6) Show 'UI.DrawBox..' methods
7) Show 'UI.MultiBox..' methods
8) Show 'UI.Grid..' methods
9) Show 'UI.InputBox' method
10) Show 'UI.DisplayMessage' method
11) Show 'UI.User Input' methods
12) Play the awesome 'Guess The Number' Game!
13) Quit

Type the number of your choice (1 to 13)>_
```

This tutorial shows you how to make the best of the standard console in Windows by using coloured text and UTF8 characters to draw lines and boxes to create some sort of UI. The spelling of colour and grey is correct UK English, but concessions have been made in code variable names to accommodate US spelling.

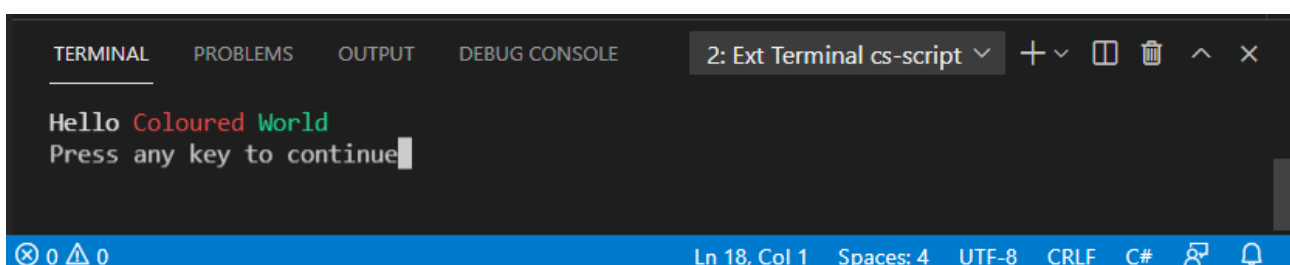
The code used has been kept as simple as possible to allow translation to Python, Lua and Java to achieve the same effect. There are probably better ways of doing it in C# that are not so easy to translate.

Coloured text is built-in to C# using the Console object. This tutorial shows how to get the best out of it.

The method is simple:

- Set the foreground colour
- Set the background colour
- Print the output
- Reset the Console colours

The following code produces a simple example of coloured output:



```
2: Ext Terminal cs-script

Hello Coloured World
Press any key to continue
```

```

using System;

static void Main()
{
    //Console.SetWindowSize(81, 25); //Not used in VSCode output
    Console.ForegroundColor = ConsoleColor.White;
    Console.BackgroundColor = ConsoleColor.Black;
    Console.Clear();
    Console.Write("Hello");
    Console.ForegroundColor = ConsoleColor.Red;
    Console.Write(" Coloured");
    Console.ForegroundColor = ConsoleColor.Green;
    Console.Write(" World\n");
    Console.ResetColor();
    Console.Write("Press any key to continue");
    Console.ReadKey();
}

```

The code as shown resizes the console to 81 columns wide by 25 rows high. The reason for 81 is to allow lines and boxes to be drawn that fill 80 columns. If the size is set to 80, the lines spill over onto the next line down. (This does not happen with Python and Lua outputs, or in VSCode.)

If you change the colours, it will only affect the next text output. The screen has to be cleared to make the new colours operate over the whole area.

The output here was just 1 line of text, but it took 10 lines of code to produce it. There has got to be a better way...

How about:

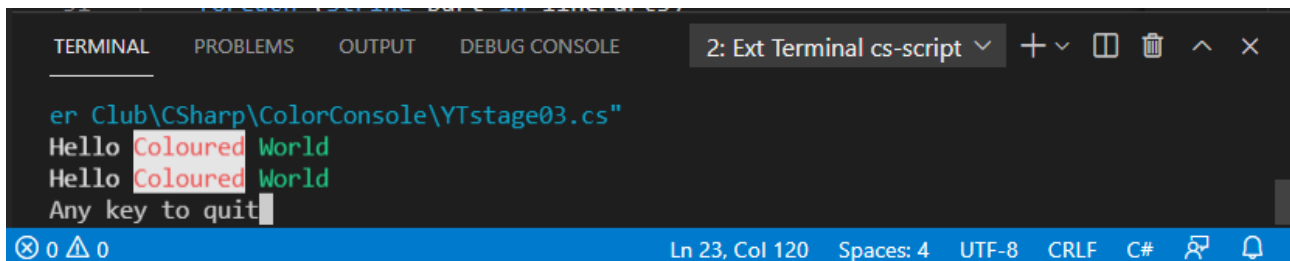
```
ColorPrint("~white~Hello ~red~~whitebg~Coloured~green~~blackbg~ World");
```

or

```
ColorPrint($"{WHITE}Hello {RED}{WHITEBG}Coloured{GREEN}{BLACKBG} World");
```

In the first example, the tilde (~) is used to tag a foreground or background colour, built in to the text. This is ideal for loading and displaying text from file, perhaps for use in a game.

The second example uses interpolated strings, starting with the \$ character so the values of the variables/constants inside {} are converted by the ColorPrint function to output colour:



Now we are getting somewhere.

The code for this is beginning to grow. The above sample needs all of this:

```

using System;
using System.Collections.Generic;

static Dictionary<string, ConsoleColor> conColors = new Dictionary<string, ConsoleColor>();
static string sep = "~";
static string WHITE = $"{sep}WHITE{sep}";
static string WHITEBG = $"{sep}WHITEBG{sep}";
static string BLACK = $"{sep}BLACK{sep}";
static string BLACKBG = $"{sep}BLACKBG{sep}";
static string RED = $"{sep}RED{sep}";
static string GREEN = $"{sep}GREEN{sep}";

static void Main(string[] args)
{
    conColors.Add("RED", ConsoleColor.Red);
    conColors.Add("GREEN", ConsoleColor.Green);
    conColors.Add("BLACK", ConsoleColor.Black);
    conColors.Add("WHITE", ConsoleColor.White);
    conColors.Add("BLACKBG", ConsoleColor.Black);
    conColors.Add("WHITEBG", ConsoleColor.White);

    ColorPrint("~white~Hello ~red~~whitebg~Coloured~green~~blackbg~ World");
    ColorPrint($"{WHITE}Hello {RED}{WHITEBG}Coloured{GREEN}{BLACKBG} World");
    Console.ResetColor();
    Console.Write("Any key to quit");
    Console.ReadKey();
}

static void ColorPrint(string text, bool newline = true, bool reset = true)
{
    string[] lineParts = text.Split(sep[0]); // sep[0] auto-converts string to char
    foreach (string part in lineParts)
    {
        if (conColors.ContainsKey(part.ToUpper())) // is 'RED' in the colors dictionary?
        {
            if (part.ToUpper().Contains("BG"))
            {
                Console.BackgroundColor = conColors[part.ToUpper()];
            }
            else if (part.ToUpper() == "RESET")
            {
                Console.ResetColor();
            }
            else
            {
                Console.ForegroundColor = conColors[part.ToUpper()];
            }
        }
        else
        {
            Console.Write(part);
        }
    }
    if (newline)
    {
        Console.Write("\n");
    }
    if (reset)
    {
        Console.ResetColor();
    }
}

```

## How does this work?

A series of variables disguised as constants in CAPS has been started, consisting of a colour name in caps assigned a value comprising a tag character (default ~ ) surrounding the colour name eg.

```
WHITE = "~WHITE~"
```

This will need to be done for all 6 colours: red, blue, green, cyan, magenta, yellow plus black and white and their dark equivalents. This combination creates 2 grey shades from the black/white/dark/bright possibilities to give Gray and DarkGray, a total of 16.

With the addition of the letters BG added to the name, 16 background colours are created, eg:

```
BLACKBG = "~BLACKBG~"
```

These can be used in the interpolated string format eg:

```
ColorPrint($"{RED}This is red text")
```

The dictionary conColors has been partially populated with the same set of colours as keys, with equivalent ConsoleColors as values. The remaining foreground and background colours need to be added.

These are used as in-line text tags eg

```
ColorPrint("~red~This is red text")
```

The colour tags can be upper/lower or mixed case eg

```
~RED~ = ~red~ = ~rEd~
```

The reason for using the variable 'sep' is to allow users to substitute the ~ character for something else if required. As sep is a string, using sep[0] will provide the char equivalent for functions requiring a (char), such as .Split()

## ColorPrint()

This takes 1 required parameter 'text' and 2 optional 'newline' and 'reset'.

The text is split into a string array using the current separator character (default ~)

The items of this array are iterated with a foreach loop, with appropriate action taken in the conditional block:

IF if an item is found in the conColor dictionary keys, it is then further checked to see if it contains "BG" indicating it is a background colour, or if it contains "RESET" (which is not present in this example code). The appropriate Console colour property is set, or Console.ResetColor() method called.

ELSE the item is output using Console.Write() which does not force a newline.

Once the array iteration is completed the Console is RESET by default, and a newline character output also by default.



# UI Library code

With these basic concepts in place, the UI library has been written to give developers a selection of methods as seen in the screenshot at the beginning of this document.

The UI library is a static class consisting of 1154 lines of code with all functions and procedures in alphabetical order. Most are public, so can be called from other classes, notably Program.cs which hosts Main()

C# static classes can have the equivalent of a constructor, which runs once only when the class is first used. This is used to initialise variables, such as populating the dictionaries:

```
#region static constructor
static UI()
{
    /// static class "constructor" runs only once on first use ///
    Initialise();
}
#endregion
```

From now on all functions and procedures will be referred to as 'functions', Whether or not they return a value is irrelevant in a general description.

The ideal way of using this library is to use it for any user input from the keyboard, and to display any output. Most program logic should be done in other classes. The program.cs and game.cs classes are used to demonstrate this principle.

## Starting in Program.Main():

```
UI.SetConsole(80, 25, "white", "black");
```

This sets the Console width to 81 characters and height to 25 rows, with white foreground on a black background. The screen is cleared within the function, and the public global variables windowWidth and windowHeight are set:

```
public static void SetConsole(int cols, int rows, string foreColor = "WHITE", string
backColor = "BLACKBG")
{
    /// Setup size and colour scheme of the console ///
    ValidateColors(ref foreColor, ref backColor);
    windowWidth = cols;
    windowHeight = rows;
    Console.SetWindowSize(windowWidth + 1, windowHeight);
    Console.BufferHeight = windowHeight;
    Console.ForegroundColor = conColors[foreColor];
    Console.BackgroundColor = conColors[backColor];
    Console.Clear();
}
```

Within this function is:

```
ValidateColors(ref foreColor, ref backColor);
```

You will note the `foreColor` and `backColor` strings are passed by reference. The reason for this is to duplicate the equivalent Python and Lua functions:

```
Python:      fore_color, back_color = validate_colors(fore_color, back_color)
Lua:         foreColor, backColor = ValidateColors(foreColor, backColor)
```

multiple return values are possible, so the use of `byRef` is a demonstration of one method of replicating this in C#. I will leave the argument over whether `ValidateColors` using `byRef` is a function or procedure to those who wish to do so on some lengthy forum post.

```
private static void ValidateColors(ref string foreColor, ref string backColor)
{
    /// Fore and back colours passed as pointers, so changing them here does not require a return ///
    string modifiedForeColor = foreColor.ToUpper(); //"red" -> "RED" "~red~" -> "~RED~"
    string modifiedBackColor = backColor.ToUpper(); //"red" -> "RED" "~redbg~" -> "~REDBG~"
    modifiedForeColor = modifiedForeColor.Replace(sep, ""); //remove separators
    modifiedBackColor = modifiedBackColor.Replace(sep, ""); //remove separators
    if (!colors.ContainsKey(modifiedForeColor)) throw new ColorValueException($"foreground colour must match
name in any format: 'white' or 'white' not {foreColor}");
    if (!colors.ContainsKey(modifiedBackColor)) throw new ColorValueException($"background colour must match
name in any format: 'black' or 'black' not {backColor}");
    modifiedForeColor = $"{sep}{modifiedForeColor}{sep}";
    if(modifiedBackColor.EndsWith("BG")) modifiedBackColor = $"{sep}{modifiedBackColor}{sep}";
    else modifiedBackColor = $"{sep}{modifiedBackColor}BG{sep}";
    foreColor = modifiedForeColor; // "~WHITE~" got through checks so re-assign original parameters by
reference
    backColor = modifiedBackColor; // "~BLACKBG~"
}
```

This function removes the `~` character and checks the upper case version of the colour is present in the Dictionary `colors`. If not a new exception is thrown to warn the developer they have made an error:

```
#region Custom exception classes
//Creating custom Exception Classes by inheriting Exception class
class ColorValueException : Exception
{
    public ColorValueException(string message) : base(message) { }
    public ColorValueException(string message, Exception innerException) : base(message, innerException) { }
}
```

Assuming the colours are validated, they are re-assembled with the `~` character front and back, and the `backColor` has `"BG"` inserted as well.

The modified colours are re-assigned to `foreColor` and `backColor`, and as they were passed byRef, the changes are immediate without needing a return. (This is a void return type, so technically it is a procedure. Plenty here to keep Forum discussions going...) (What about returning `foreColor` as a string, and `backColor` byRef, just to make life complicated?)

Continuing with `Main()`

A while loop runs the menu system shown in the screenshot on page 1.

A title for the menu is stored in string `title`.

A List of the required menu items is created using the embedded colour tags method:

```
options.Add("~green~Show a mix of different colours");
```

The title and list are passed to the `UI.Menu()` method.

The `Menu()` function will always return a valid integer, so the conditional block can be used to choose the next stage. If option 13. Quit is chosen, then `choice = 12` and the `UI.Quit()` function is called

## UI.Menu()

```
int choice = UI.Menu("d", title, ">", options);
```

"d" indicates use a double line box construction.

">" is the prompt characters where the user is expected to type their input.

The returned int value is the equivalent index of the option chosen from the list. The list display is 1 based, but the return is 0 based.

```
public static int Menu(  
    string style,  
    string title,  
    string promptChar,  
    List<string> textLines,  
    string foreColor = "WHITE",  
    string backColor = "BLACKBG",  
    string align = "left",  
    int width = 0)
```

The remaining parameters are optional.

foreColor and backColor only affect the boxes that make up the menu.

align is not currently used, but was included for further development

width can be set if less than Console.Width is required. (0 = Console.Width)

The boxes and their contents are drawn using:

```
DrawBoxOutline()  
DrawBoxBody()
```

Empty lines are added to fill the first 20 rows:

```
AddLines(5, numLines);
```

A white full width double line is drawn:

```
DrawLine("d", WHITE, BLACKBG);
```

User choice is made via the GetInteger() function

C#

```
int userInput = GetInteger(numLines, $"Type the number of your choice (1 to  
{textLines.Count})", promptChar, WHITE, BLACKBG, 1, textLines.Count);
```

Python:

```
user_input = get_integer(f"Type the number of your choice (1 to {len(text_lines)}")  
prompt_end, 'white', 'blackbg', 1, len(text_lines));
```

Lua:

```
userInput = GetInteger("Type the number of your choice (1 to "..#textLines..")",  
promptChar, foreColor, backColor, 1, #textLines)
```



If the user does not respond correctly, the `GetInteger()` method displays a message for 2 seconds, then the input part of the screen is cleared and re-drawn. It is this ability to continuously re-draw parts or the whole console that gives the best approximation of a GUI in appearance.

Most methods are self-documenting but a few notes here for any special features:

**AddLines()** is overloaded, so you can either add a specified number of blank lines, or add more lines to the current display to fill up to a desired row:

```
numLines += AddLines(2);    //Adds 2 extra lines and increments the line count
AddLines(5, numLines);     //Adds enough lines to fill console leaving 5 empty lines
```

**DisplayMessage()** is overloaded so you can supply either a string or a `List<string>`

```
GetBoolean(prompt, promptChar, textColor, backColor)
GetRealNumber(prompt, promptChar, textColor, backColor, min, max)
GetInteger(prompt, promptChar, textColor, backColor, min, max)
GetString(prompt, promptChar, textColor, backColor, withTitle, min, max)
```

All these methods are similar to Python's `input()` method, but can be configured to use coloured text, a choice of characters after the prompt, set limits on the size of returned text or magnitude of real / integer numbers accepted. They are guaranteed to return the correct `dataType` and within limits set by the developer.

They all use the private method `ProcessInput()` internally to process the keyboard input and carry out validation checks to ensure user-compliance.

**private static int FormatColorTags(ref string text, out List<string> colorTagList)**

This is the ultimate C# multi-return function!

- It has an `int` return value.
- It uses the `out List<string>` technique to add a sneaky second return. You may be familiar with this with the `int.TryParse()` or other `object.TryParse()` methods.
- It uses `byRef` on a string input, so changes its value in-situ.
- It uses `Linq` to count the number of occurrences of the `~` tag character in the supplied text

The purpose of this function is three-fold:

1. Count the number of characters used to define colour instructions eg. `~green~` = 7
2. Create a `List<string>` of colours used in the embedded text eg `~green~~blackbg~` = "`~GREEN~`", "`~BLACKBG~`"
3. Format the colour tags to upper case. e.g `~green~` → `~GREEN~`

The reason for doing all this is to enable the calculation of the length of a string in order to fit into a specified width. The number of characters used for colour formatting can then be taken into account.

A function that uses this feature is:

```
private static string GetMaxLengthString(string text, int maxLength, out int colorTagSpaces)
```

which is called by the function:

```
public static List<string> GetFormattedLines()
```

This takes a list of strings, which may contain colour formatting tags, and breaks them into sections to the nearest whole word to fit inside a box of given width.

Example: Program.UserInputDemo()

```
string description =
"The user input methods are:\n\n" +
"~green~UI.GetString ~white~to get a string typed by the user. " +
"There is an option to set minimum and maximum length," +
" and to convert to TitleCase.\n\n" +
"~cyan~UI.GetInteger ~white~to get an integer value. Minimum and max values can be specified.\n\n" +
"~magenta~UI. GetRealNumber ~white~similar to UI.GetInteger, allows for real numbers to be entered.\n\n" +
"~blue~UI.GetBoolean ~white~requires the user to type 'y' or 'n' and returns a boolean value.\n\n"+
"~green~Test ~white~each one out with ~blue~Enter ~white~only, or wrong numbers, too many characters etc.\n\n" +
"~red~Try and break it!";

string retValue = UI.GetInputDemo("string", description);

passed to  GetInputDemo(string demoType, string description)

int numLines = DrawMultiLineBox("s", description, "yellow", "black", "white", "black",
"left");
```