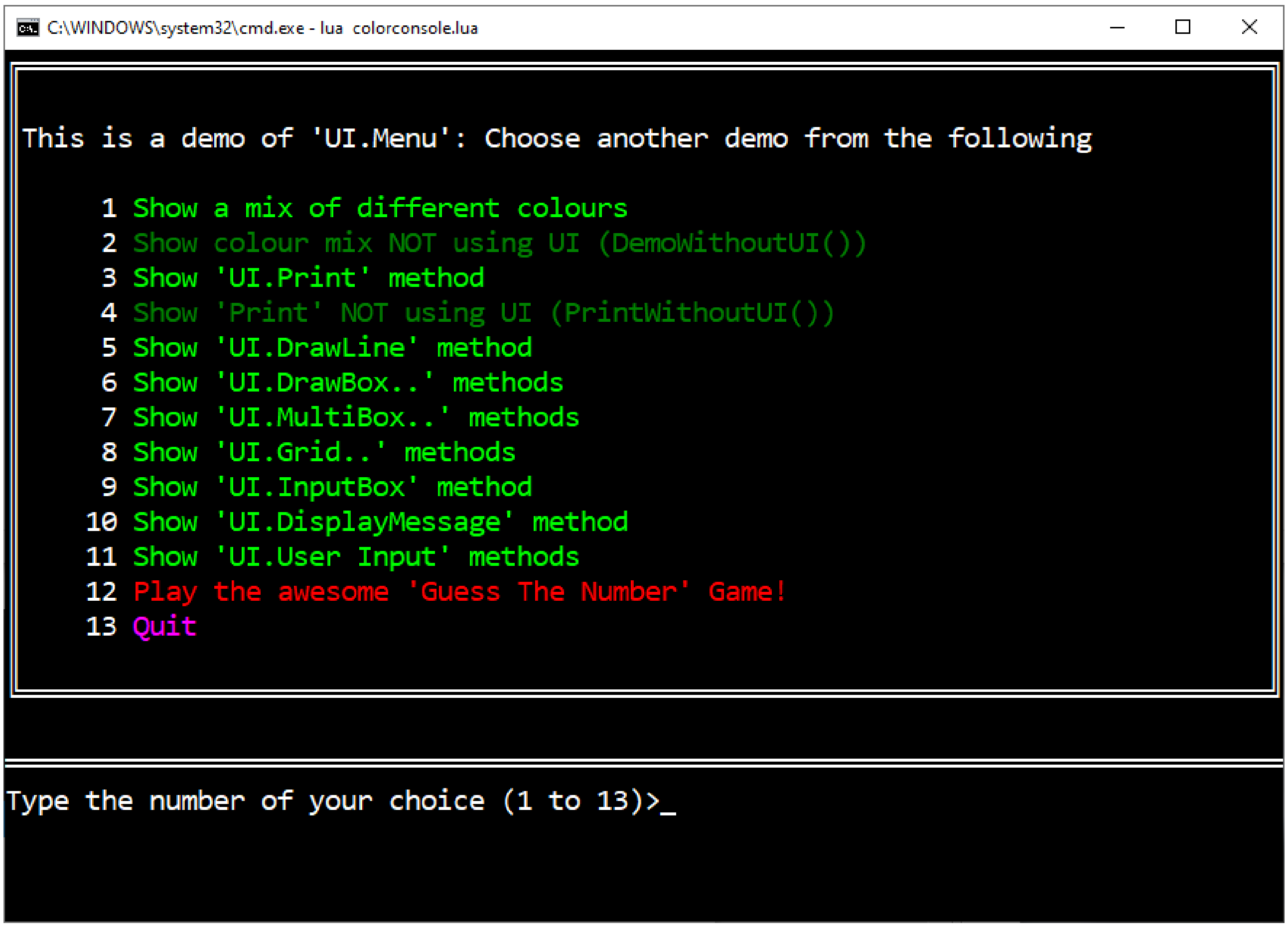
# Colo(u)r Console Lua



```
C:\WINDOWS\system32\cmd.exe - lua  colorconsole.lua

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 allow the drawing of 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 names to accommodate US spelling

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

C# has coloured text built-in. Python has the Colorama module, but the only similar library for Lua is ansicolors.lua from here: https://github.com/kikito/ansicolors.lua

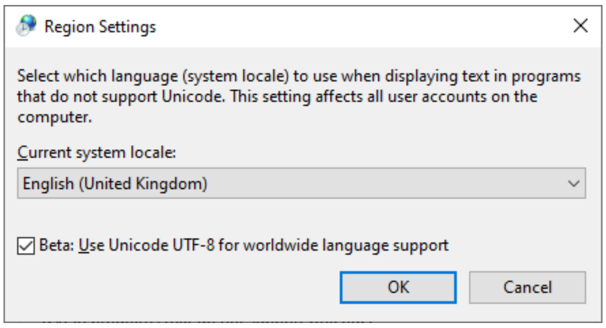
This has been used for the inspiration to get coloured output from Lua, even in Windows, but has not been included in the final code library.

## Display UTF8 and ANSI in Windows Cmd and Powershell

You will need Windows 10 version 1909 or later. At the time of writing the current version is 20H2, which has to be manually enabled. There is a method shown here:

https://stackoverflow.com/questions/57131654/using-utf-8-encoding-chcp-65001-in-command-prompt-windows-powershell-window/57134096#57134096

This page shows the setting correctly applied here:



Select which language (system locale) to use when displaying text in programs that do not support Unicode. This setting affects all user accounts on the computer.

Current system locale:

English (United Kingdom)

☑ Beta: Use Unicode UTF-8 for worldwide language support

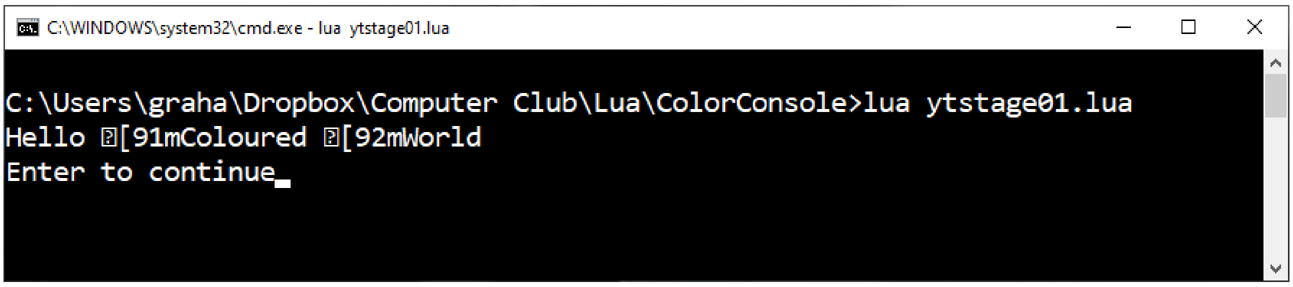
You will need a re-boot to activate this setting.

To get the coloured equivalent of “Hello World” still takes some setting up inLua code:

Let’s try:

```
print("Hello "..string.char(27)..'[91m'.."Coloured "..string.char(27)..'[92m'.."World")
```

This should give Hello in white, Coloured in red, and World in green:



```
C:\Users\graha\Dropbox\Computer Club\Lua\ColorConsole>lua ytstage01.lua
Hello [91mColoured [92mWorld
Enter to continue_
```

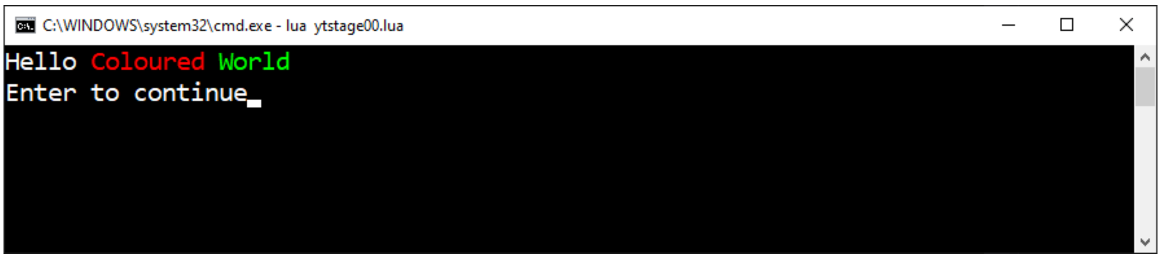
Fail.

It appears you have to initialise the console in some way, either by changing the size, or simply clearing it first, then it behaves properly.

The minimum code needed is:

```
local function main()
    os.execute("cls")

    print("Hello "..string.char(27)..'[91m'.."Coloured"..
           string.char(27)..'[92m'.."World"..
           string.char(27)..'[0m')

    io.write("Enter to continue")
    io.read()
end

main()
```

The command `os.execute("cls")` is Windows- specific and clears the console. Once that has been done, the ANSI codes work perfectly:



```
Hello Coloured World
Enter to continue_
```

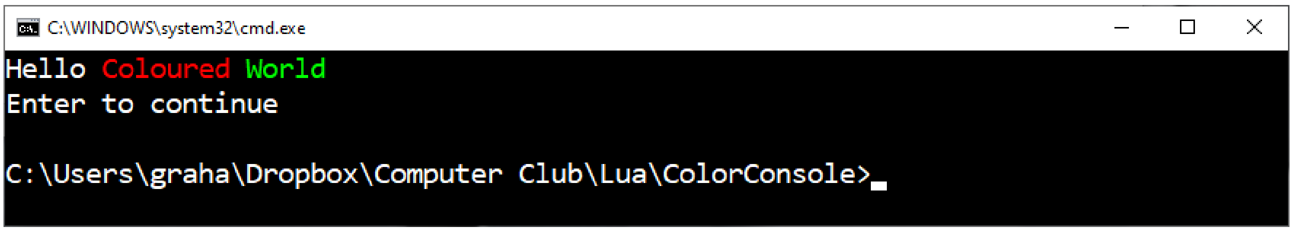
A full list of ANSI codes can be found here: https://ss64.com/nt/syntax-ansi.html

This works ok, but is going to be extremely tedious to write an epic game following this technique.

**Step 1** create some constants for the ANSI codes, and add a couple of functions to resize and clear the console first:

```lua
-- create ANSI constants
local escape    = string.char(27)..'['
local WHITE     = escape..'97m'
local RED       = escape..'91m'
local GREEN     = escape..'92m'
local BLACKBG   = escape..'40m'
local WHITEBG   = escape..'107m'
local RESET     = escape..'0m'

local function GetOS()
    -- get os type: nt = Windows
    local cpath = package.cpath
    local OS = ""
    if cpath:find('\\') == nil then
        OS = 'posix'
    else
        OS = 'nt'
    end
    return OS
end

local function Clear()
    --clear screen
    if GetOS() == 'nt' then
        os.execute("cls")
    else
        os.execute("clear")
    end
end

local function Resize(cols, rows)
    -- set console size
    if GetOS() == 'nt' then
        os.execute("mode ".. cols ..",".. rows)
    else
        local cmd = "'\\e[8;".. cols ..";".. rows .."t'"
        os.execute("echo -e "..cmd)
    end
end

function main()
    Resize(80, 5)
    Clear()
    print("Hello "..RED.."Coloured "..GREEN.."World"..RESET)
    io.write("Enter to continue")
    io.read()
end
```

```
main()
```



```
Hello Coloured World
Enter to continue

C:\Users\graha\Dropbox\Computer Club\Lua\ColorConsole>_
```

The console height was set to 5 rows inside main() by calling Resize(80, 5)
The console was cleared in main() by calling Clear()
Clear() checked the os to call the correct method “cls” or “clear”
The coloured output was achieved with:

```
print("Hello "..RED.."Coloured "..GREEN.."World"..RESET)
```

This is the equivalent of the interpolated string formats:

```
C#:          $”Hello {RED}Coloured {GREEN}World{RESET}”
Python:      f”Hello {RED}Coloured {GREEN}World{RESET}”
```

Lua does not have anything similar, but the string concatenation token `..` does a good job in its place.

Another method of outputting colour is to embed some form of colour tagging to strings. This is useful when reading text files, so the formatting is built-in. For this series on C#, Python and Lua coloured consoles, the tilde ~ character has been chosen as an ideal tag, eg:

```
“Hello ~red~Coloured ~green~World”
```

This needs a function to interpret this string, extract the colour information from it and print it out. Ideally it should be able to deal with both methods of embedding colours.

A table (equivalent to C# or Python Dictionary) is ideal for this:

```
local colors  = {}
colors.BLACK        = BLACK
colors.WHITE        = WHITE
colors.GREEN        = GREEN
colors.RED          = RED
colors.BLACKBG      = BLACKBG
colors.RESET        = RESET
```

Elements of this table can be accessed with dot notation: `colors.RED` or by index: `colors[“RED”]`

The string index refers to the pre-assigned constants WHITE, RED etc, which hold the ANSI codes for all the colours

Two functions are required to ColorPrint() both types of colour tags:

```lua
local function RemoveANSI(text)
    -- text = "Hello..RED..World" = "Hello\27[91m World"
    local retValue = ''
    local start = text:find(string.char(27))
    while start ~= nil do                          -- start = 6
       retValue = retValue..text:sub(1, start - 1) -- "Hello "
       text = text:sub(start)                      -- "\27[91m World"
       start = text:find('m')                      -- 5
       text = text:sub(start + 1)                  -- " World"
       start = text:find(string.char(27))          -- nil: end loop
    end
    return retValue
end

local function ColorPrint(value, newline, reset)
    --[[ If running from Zeobrane or other IDE isColoured = false: ignore colour tags]]
    if newline == nil then newline = true end
    if reset == nil then reset = true end
    numLines = 0
    if value:find(sep) ~= nil then -- sep default value ~ text has colour tags in it
       lineParts = value:Split(sep) -- Split is NOT a built-in function
       for i = 1, #lineParts do
           part = lineParts[i]
           if colors[part:upper()]~= nil then-- is 'red' / 'RED' in the dictionary?
              if isColoured then – global boolean value set already
                  io.write(colors[part:upper()])
              end
           else -- not a colour command so print it out without newline
              if part:find("\n") ~= nil then
                  numLines = numLines + 1
              end
              io.write(part)
           end
       end
       if reset and isColoured then
           io.write(colors.RESET)
       end
    elseif value:find(string.char(27)) ~= nil then
       -- string is in form of "Hello..RED..World"
       -- can be printed directly if isColoured
       if not isColoured or not isConsole then
           -- remove colour tags
           value = RemoveANSI(value)
       end
       io.write(value)
    else -- no colour tags in the text
       io.write(value)
    end
    if newline then
       io.write("\n")     -- Add newline to complete the print command
       numLines = numLines + 1
    end

    return numLines
end
```

## How does this work?

`ColorPrint()` takes the text to be printed, and otional boolean flags to allow choice of adding a newline character (similar to print()) and a choice of resetting the console colours back to default. Both these options are true by default

The text is checked to see if it contains the ~ character, and if found, it is assumed there is at least one colour embedded in the form "Hello ~red~World". The text is split on the ~ character and the resulting table iterated to output the correct coloured output. If running in an IDE the color tags are removed. Split is not a built-in Lua function, so you have to code your own version. I used one found here: http://lua-users.org/wiki/SplitJoin and adapted it to get rid of empty fields.

If there are no ~ characters, the text is checked to see if it contains the escape character 27. If so the text is assumed to contain at least one colour in the form "Hello..RED..World" If running in a console/terminal rather than an IDE, it is assumed colours can be displayed, so the text is printed to screen with io.write(). If in an IDE such as ZeroBrane, the ANSI codes are removed by the `function RemoveANSI(text)`

Once the table iteration is completed the Console is RESET by default, and a newline character output also by default.
The reason for making these optional is to give greater flexibility. For example:

```
ColorPrint("~green~Type your name>_~red~", false, false);
```

This will write `"Type your name>_"` in green and leave the cursor on the same line. The user's input will appear in red as they type because the colours were NOT reset.

## Lines and Boxes

The use of lines and boxes can make a console based ui look very professional. There are a set of UTF8 characters for this purpose eg:

```
╔ ═ ╗ ╦ ╚ ╝ ╩ ╠ ╣ ╬ ║

Single and double line boxes:

┌────────┬────────┐   ╔══════╦══════╗
│        │        │   ║      ║      ║
├────────┼────────┤   ╠══════╬══════╣
│        │        │   ║      ║      ║
└────────┴────────┘   ╚══════╩══════╝
```

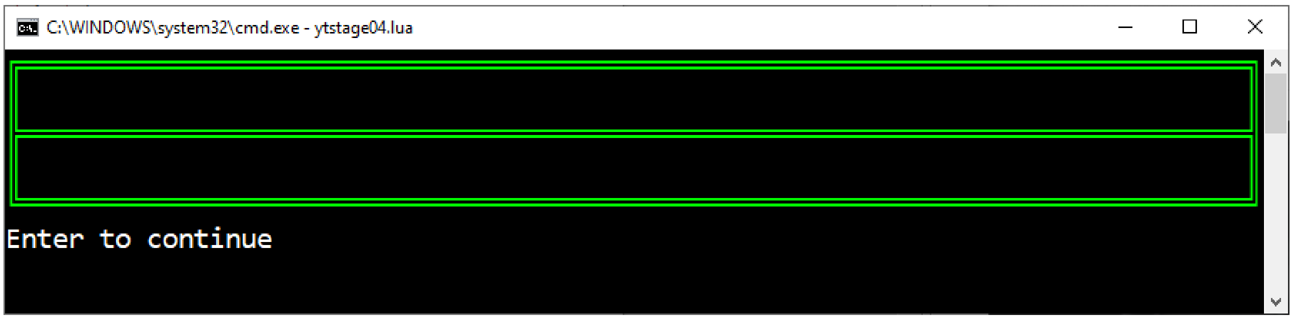
These characters are difficult to enter via the keyboard, so are best copy/pasted from specialised websites such as https://www.w3schools.com/charsets/ref_utf_box.asp

If they are kept in tables they are easier to manipulate:

```
local sSymbolsTop        = {'┌', '─', '┐', '┬'}
local sSymbolsBottom     = {'└', '─', '┘', '┴'}
local sSymbolsBody       = {'│', ' ', '│', '│'}
local sSymbolsMid        = {'├', '─', '┤', '┼'}
local dSymbolsTop        = {'╔', '═', '╗', '╦'}
local dSymbolsBottom     = {'╚', '═', '╝', '╩'}
local dSymbolsBody       = {'║', ' ', '║', '║'}
local dSymbolsMid        = {'╠', '═', '╣', '╬'}
```

The following code using the ColorPrint() functions draws a double box:

```
ColorPrint(GREEN..dSymbolsTop[1]..(""):PadRight(78, dSymbolsTop[2])..dSymbolsTop[3])
ColorPrint(GREEN..dSymbolsBody[1]..(""):PadRight(78, dSymbolsBody[2])..dSymbolsBody[3])
ColorPrint(GREEN..dSymbolsMid[1]..(""):PadRight(78, dSymbolsMid[2])..dSymbolsMid[3])
ColorPrint(GREEN..dSymbolsBody[1]..(""):PadRight(78, dSymbolsBody[2])..dSymbolsBody[3])
ColorPrint(GREEN..dSymbolsBottom[1]..(""):PadRight(78, dSymbolsBottom[2])..dSymbolsBottom[3])
```



```
Enter to continue
```

# 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 returns a table consisting of 1439  lines of code. Most functions and procedures are in alphabetical order, but unfortunatey in Lua, if a function calls another function, that one has to appear above the calling function, otherwise an error is thrown. Those functions preceded with ‘ui.’ Are effectively public. Those preceded by ‘local’ are effectively private to the table.

The table calls Initialise() at the end of the code, just before returning the entire table to the require() function in the calling script. This populates any internal tables and other variables, similar to the use of a static ‘constructor’ found in a C# static class.

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 colorconsole.lua script is used to demonstrate this principle.

## Starting in colorconsole.Main()

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

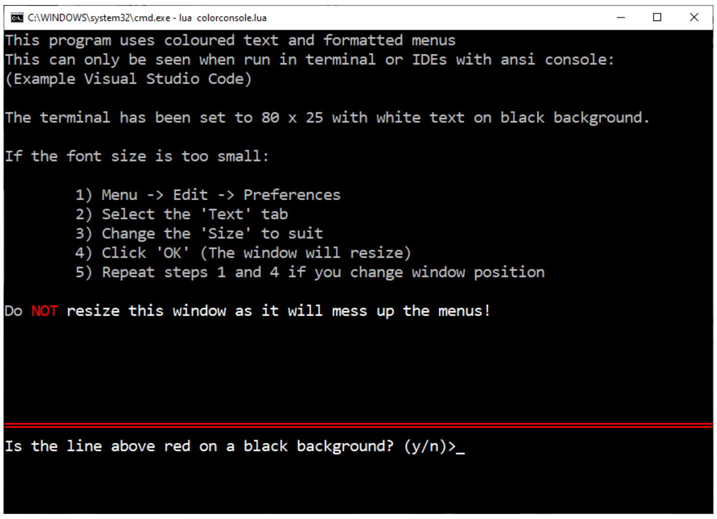
This sets the Console width to 80 characters and height to 25 rows, with white foreground on a black background. The screen is cleared within the function, and the public global variables windowWidth and windowHeight  are set:

```
function ui.SetConsole(cols, rows, foreColor, backColor, initialise)
    initialise = initialise or false
    -- Windows colour codes
       local windows =
       {
              black = '0', blue = '1', green = '2', aqua = '3',
              red = '4', purple = '5', yellow = '6',  white = '7',
              grey = '8', lightblue = '9', lightgreen = 'A', lightaqua = 'B',
              lightred = 'C', lightpurple = 'D', lightyellow = 'E', brightwhite = 'F'
       }
       -- Linux termial colour codes
       local linuxFg =
       {
              black = '30', red = '31',  green = '32', yellow = '33',
              blue = '34', magenta = '35',  cyan = '36',  white = '37',
       }
       local linuxBg =
       {
              black = '40', red = '41',  green = '42', yellow = '43',
              blue = '44', magenta = '45',  cyan = '46',  white = '47'
       }
       windowWidth = cols
       windowHeight = rows
       Resize(windowWidth, windowHeight)
       -- change colours
       if isConsole then
           if GetOS() == 'nt' then
              os.execute("color "..windows[backColor]..windows[foreColor]) -- white on black
           else
              local bc = "'\\e["..linuxBg[backColor].."m'"
              os.execute("echo -e "..bc)
              local fc = "'\\e[".."1".."m'"
              os.execute("echo -e "..fc)
              fc = "'\\e["..linuxFg[foreColor].."m'"
              os.execute("echo -e "..fc)
           end
           ui.Clear()
       end
       if initialise then
              ui.DisplaySetup(cols, rows, foreColor, backColor)
       end
end
```

The tables hold values for foreground and background colours for both Windows and Linux systems, and these are used to set the console colours. In Linux the same ANSI codes are used, but within an os.execute call.

There is an optional `ui.DisplaySetup()` call, which allows the user to cofirm if they are looking at a coloured display, and sets the isColoured flag accordingly. This will run if the initialise flag is set to true when is called:

```
ui.SetConsole(80, 25, ‘white’, ‘black’, true)
```



```
This program uses coloured text and formatted menus
This can only be seen when run in terminal or IDEs with ansi console:
(Example Visual Studio Code)

The terminal has been set to 80 x 25 with white text on black background.

If the font size is too small:

        1) Menu -> Edit -> Preferences
        2) Select the 'Text' tab
        3) Change the 'Size' to suit
        4) Click 'OK' (The window will resize)
        5) Repeat steps 1 and 4 if you change window position

Do NOT resize this window as it will mess up the menus!



Is the line above red on a black background? (y/n)>_
```

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:

```
local title = "This is a demo of 'UI.Menu': Choose another demo from the following"
local options = {}
table.insert(options, "~green~Show a mix of different colours")
table.insert(options, "~dgreen~Show colour mix NOT using UI (DemoWithoutUI())")
table.insert(options, "~green~Show 'UI.Print' method")
<continued>
local choice = UI.Menu("d", title, ">_", options)
```

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()

```
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 and the return value is 1 based.

```
function ui.Menu(style, title, promptChar, textLines, foreColor, backColor, align, width)
```

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);
```

If the user does not respond correctly, the GetInteger() method returns the validity as false, and a message is displayed for 2 seconds, then the lower part of the console 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.

Use the examples in colorconsole.lua to give you ideas on creating your own interface