

# Lua with ZeroBrane Studio Tutorial

*Note code has been copy/pasted from Visual Studio 2019 and the colour highlighting does NOT match ZeroBrane Studio.*

You will be following this tutorial as one of the following:

1. I have never done any text-based coding.
2. I have used Python.
3. I have used C#, Java or other languages.
4. I am an expert in everything: Show me something new.

Lua code can be written using Notepad or any text editor, but in school environments running Lua scripts can be difficult, as you do not have access to the command line, and the chances of an association being set for .lua files to an interpreter are slim.

It is better to use an IDE (Integrated Development Environment). One of the best is a free application called ZeroBrane Studio which can be downloaded from <https://studio.zerobrane.com/>

The first thing to get used to when coding in any language is to **stop double-clicking** on your files to launch the editor. This may work for Word, Excel or simple text files, but will not work for coding. At best this will run the file, at worst you will get Windows asking you how it should handle the file type.

The second thing to get used to is organising your coding files properly. This makes finding them easier, and most IDEs in most languages allow you to select a folder and edit or run any of the files within it.

As you progress with coding skills, you will find you are using 'projects' or 'solutions' which consist of multiple files and folders, so it is a good idea to start with that principle.

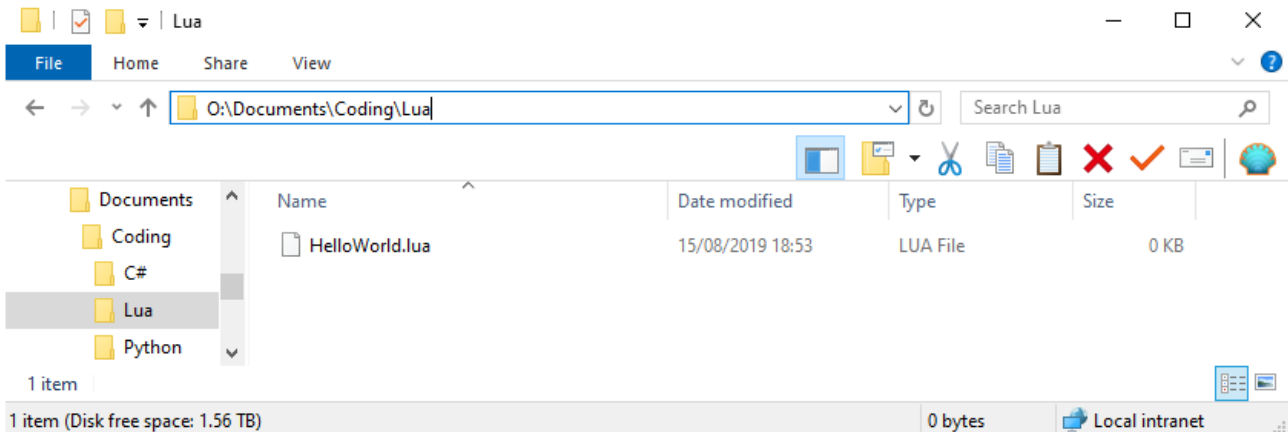
# Organising your files and folders

Suggested scheme to get started:

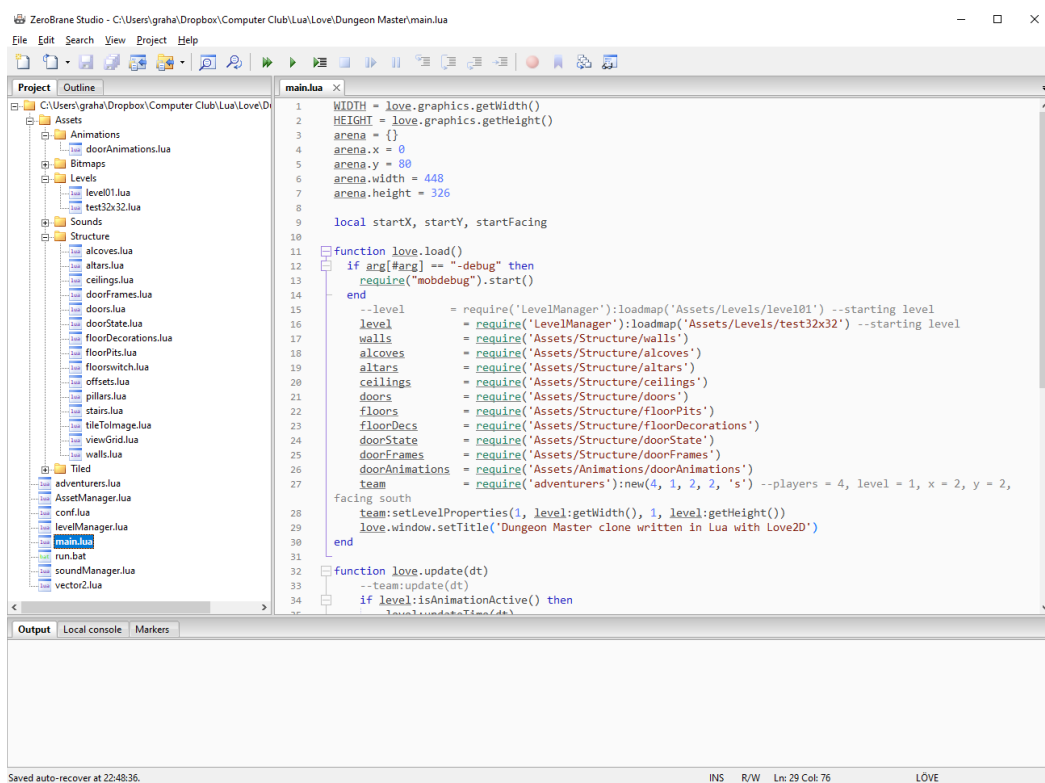
Create a new folder in Documents called 'Coding'

Create a new folder inside Coding called 'Lua'

These folders will be empty initially, until you write new files from the IDE. The "HelloWorld.lua" file seen below is an example of this.

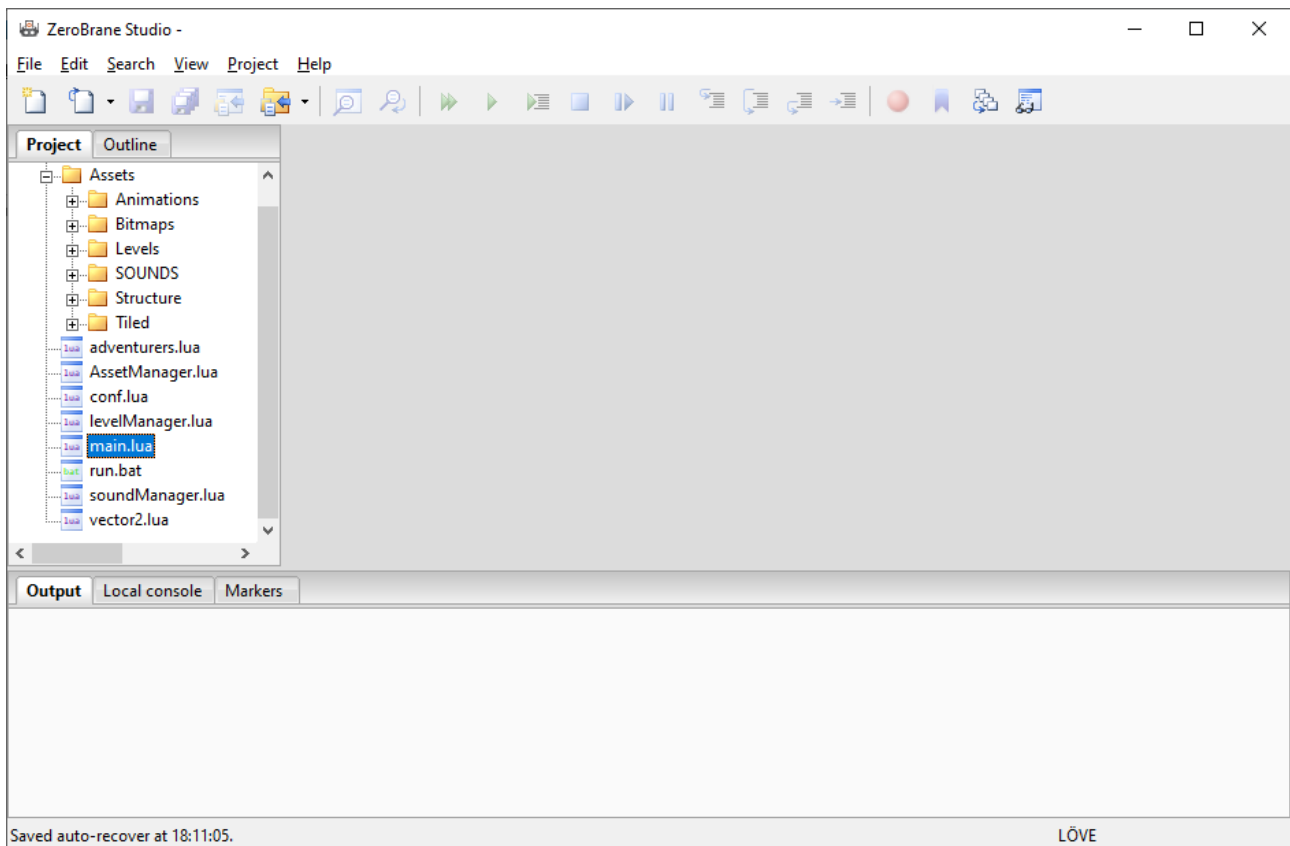


An example of a 'Project' using Love2D game engine and Lua demonstrating how complex a project can become:



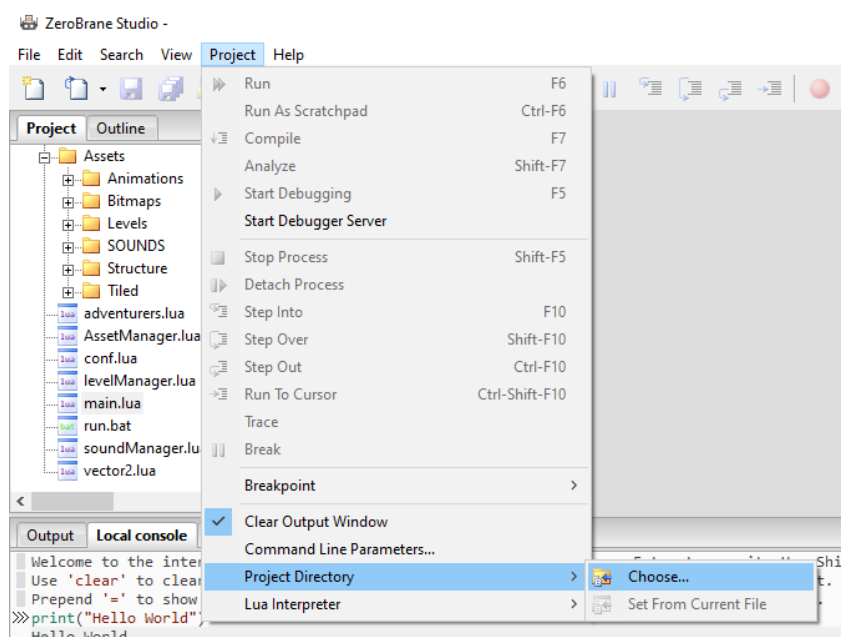
# Using ZeroBrane Studio IDE

Start Zerobrane:

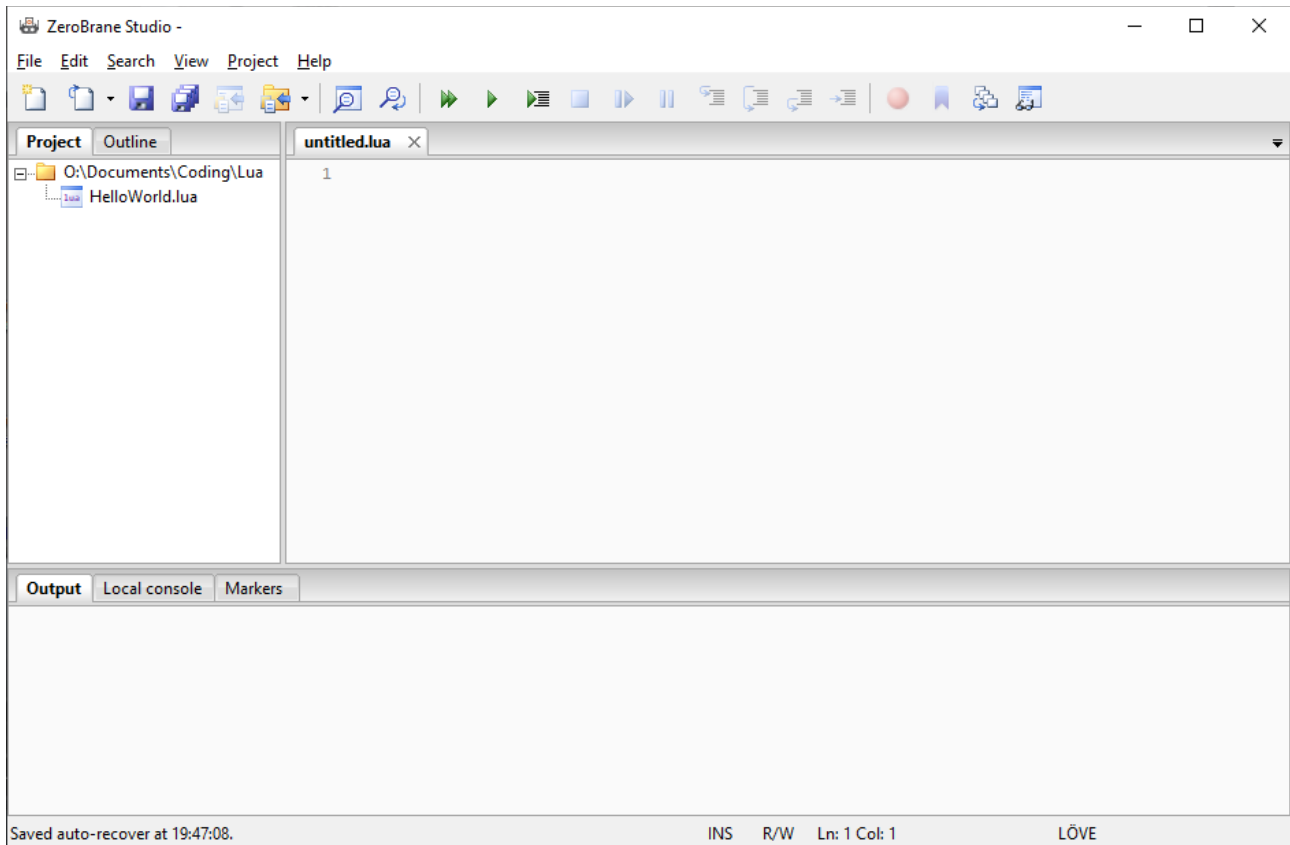


Normally, the last project worked on will be automatically loaded, but at schools this is unlikely, as the settings are found in C:\Users\<USERNAME>\AppData\Roaming\ZeroBraneStudio.ini. You would have to use the same workstation every time to ensure your settings are re-used.

Click on the Project Menu → Project Directory → Choose



Locate your newly created Lua Folder and click 'Select folder':



## Direct Lua Commands

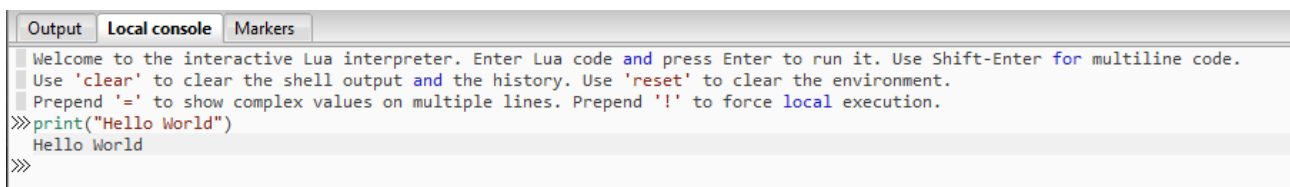
If you have used Python before, you will be familiar with the 'Shell' which allows you to type commands in one at a time.

The same can be done with Lua.

Click on the 'Local console' tab.

Type: `print("Hello World")`

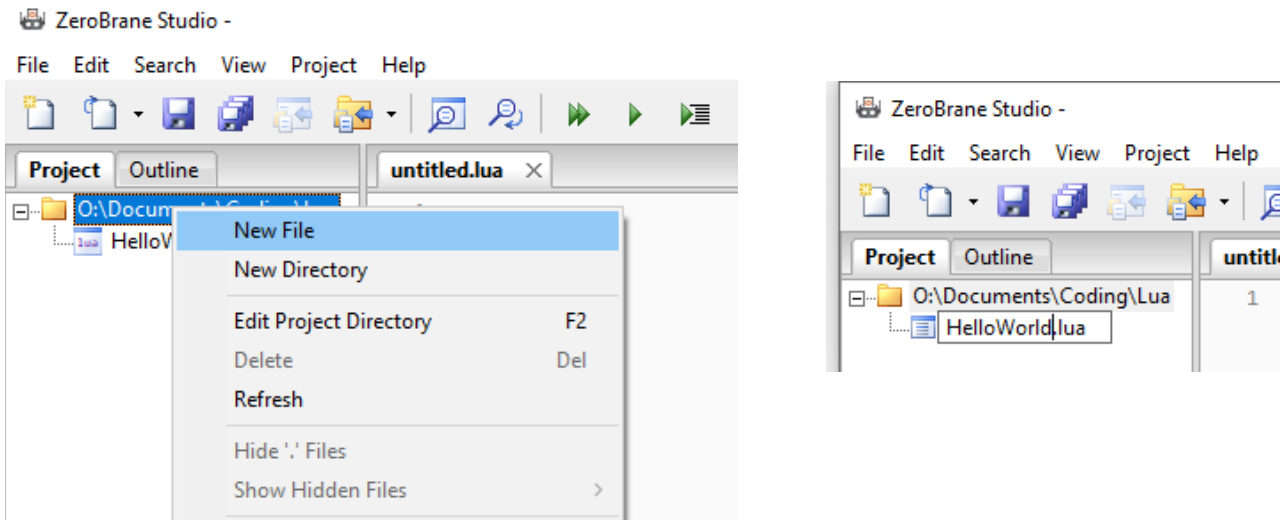
Press enter. The words "Hello World" appear in the console:



The line is temporarily saved in memory, and can be recalled using the up arrow on the keyboard, but to save it permanently it needs to be saved to a file, exactly the same as Python.

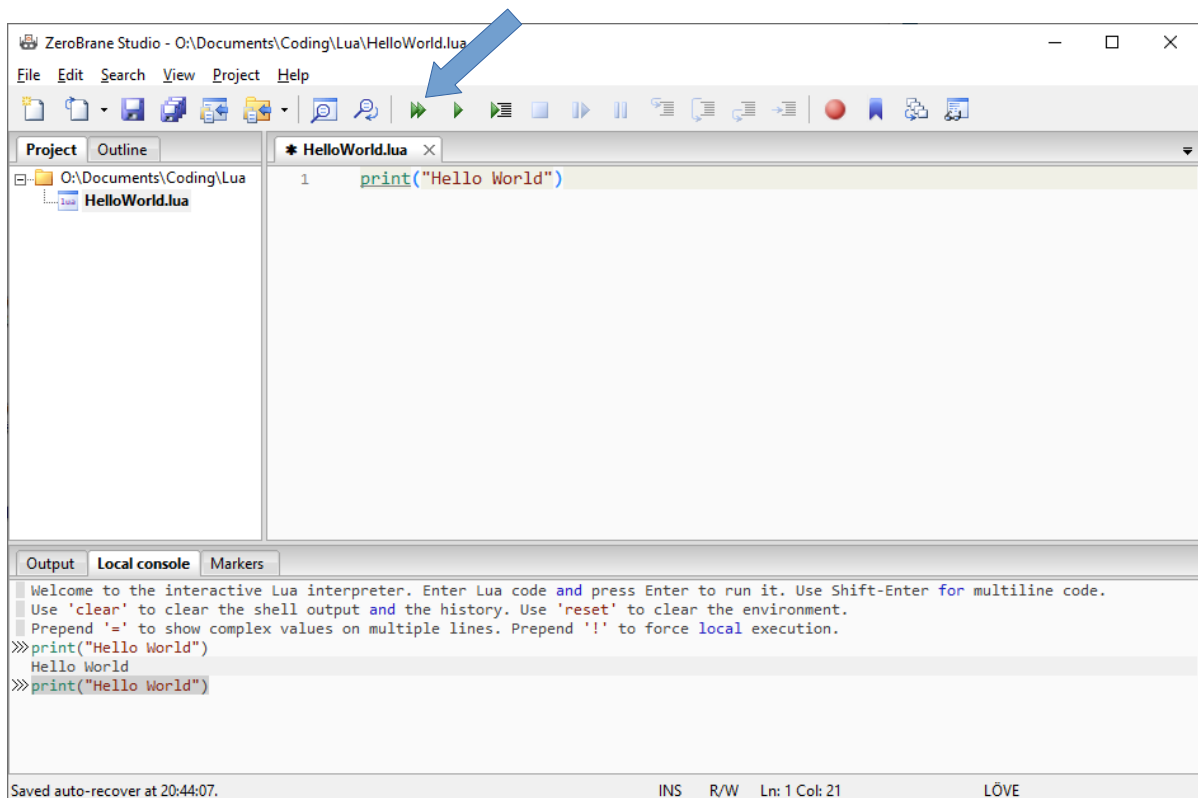
## The Statutory “Hello World” script

Right-Click on your project folder name and select ‘New File’. Type “HelloWorld.lua” into the empty box and press ‘Enter’. Don’t forget to use the .lua extension:



Double-Click on the new file to bring it into the editor.

Type: `print("Hello World")` into the Editor window:



Click the double green triangles indicated above. This will automatically save the code and run it.

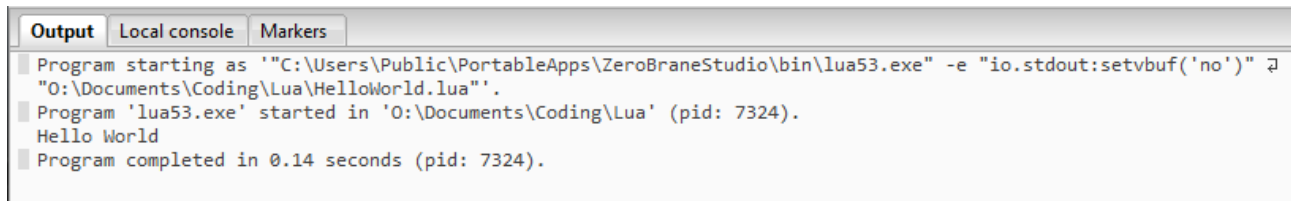
If you see a message in the Output tab similar to this:

Can't find 'main.lua' file in the current project folder: 'O:\Documents\Coding\Lua'.

Take the following steps:

Menu: Project → Lua Interpreter → Lua 5.3

Try again:



```
Output Local console Markers
Program starting as '"C:\Users\Public\PortableApps\ZeroBraneStudio\bin\lua53.exe" -e "io.stdout:setvbuf('no')"'
"O:\Documents\Coding\Lua\HelloWorld.lua".
Program 'lua53.exe' started in 'O:\Documents\Coding\Lua' (pid: 7324).
Hello World
Program completed in 0.14 seconds (pid: 7324).
```

Well done. You have run your first script.

It is only one line, but a script can be hundreds or even thousands of lines.

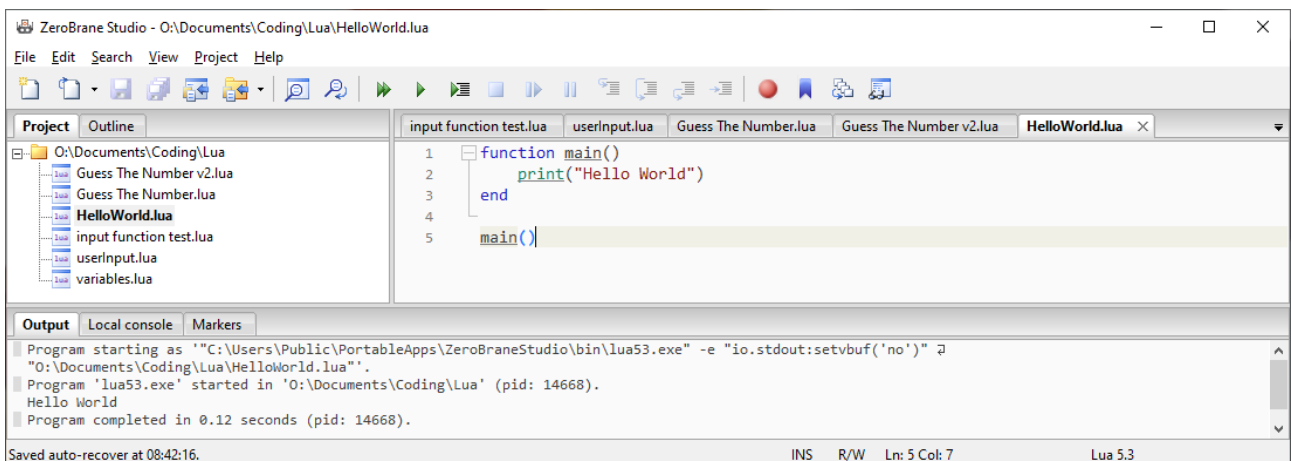
Lua (and Python) are interpreted languages. They read your script line by line, and carry out the instructions you wrote. In this case it 'print' s the text 'Hello World' to the console (screen).

ZeroBrane has a built-in interpreter, so Lua does not need to be separately installed on your system.

It is a good idea to use non-linear, or procedural programming right from the start. To get you used to that, alter your script to the lines below:

```
function main()
    print("Hello World")
end

main()
```



It works exactly the same as before, so why bother?

Procedural programming is much more efficient and can save you a lot of typing.

As your script grows in size, using functions and procedures makes it easier to maintain.

Functions carry out a block of code and return data.

Procedures carry out a block of code, but do not return anything.

As Lua only has the keyword 'function', which can also be used for procedures, I will use the word function to mean either in this document.

More advanced languages such as C# and Java start with a procedure called `main()` so it is a good idea to start using a similar version in Lua (and Python).

When the interpreter reads your modified script this time, it ignores any functions, but makes a mental note where they are, so when your script uses or 'calls' them, it knows where they are.

(Python works in the same way with 'def' and 'class')

When it reaches line 5 (`main()`), this is a call to the `main()` function found at line 1, so the script jumps from line 5, which is the starting point, to line 1, the `main()` function.

`main()` has only one line of code - `print("Hello World")`, which it executes, then control is returned back to line 5, which happens to be the end of the program.

All further examples and exercises will use a `main()` function.

If you are familiar with Python the next section will be very simple for you.

## Variables

Variables are memory locations reserved to hold some kind of data, and given a label to identify them. They must begin with a letter or an underscore character, and must not contain any spaces.

### Examples:

```
myName = "Fred"
myAge = 15
isWorking = false
```

### Non-Examples:

```
1myName (starts with a number)
!myName (starts with a !)
my Name (contains a space)
```

The first one 'myName' is the label, and it has been given the word 'Fred' as the data stored in it. Variables containing a string of letters or words are called 'string' variables

myAge contains a whole number. It is called an 'integer' variable

isWorking contains either true or false. It is called a 'boolean' variable

Lua and Python have a fairly flexible approach to variables. You can change the data stored in them from string to integer or boolean without any problem. (This is not the case with C# and Java)

## Comments

As your script gets bigger, it is helpful to write in some comments to help others understand what you are doing, or to remind yourself if you come back to it after some time.

Single-line comments are started with two hyphens: -- (no space between them)  
(Python comments start with a #, C# and Java use //)

Comments are ignored by the interpreter.

Multi-line comments start with --[[ and end with ]]  
(Python uses ''' or """ at the beginning and end, C# and Java use /\* at the start and \*/ at the end)

```
--[[ This is a multi-Line comment.
    It allows plenty of room to make notes]]

print("Hello World") -- this line prints 'Hello World'

--[[[[This comment allows the use of '[' inside it,
    By adding 2 more square brackets at the start.]]
```



# Script using variables

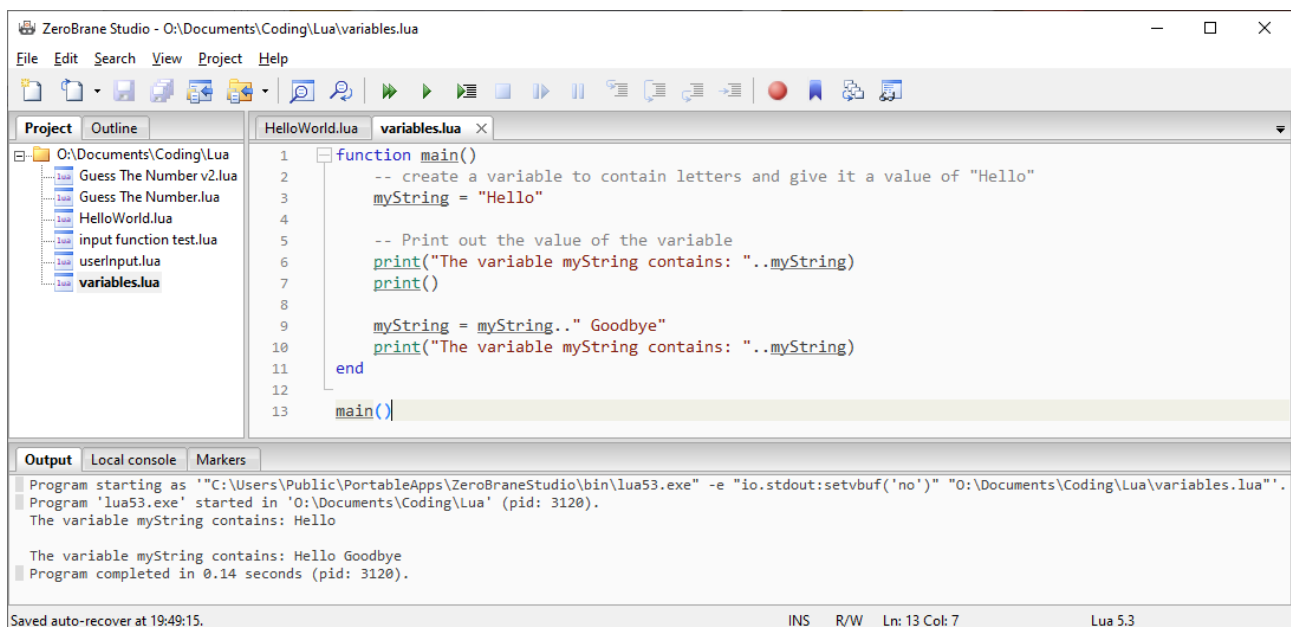
Create a new Lua file called 'variables.lua' and type the following code:

```
function main()
    myString = "Hello"

    print("The variable myString contains: "..myString)
    print()

    myString = myString.." Goodbye"
    print("The variable myString contains: "..myString)
end

main()
```



There is a lot happening here:

`myString = "Hello"`

This line creates a variable called `myString`, and gives it the value 'Hello'. It now contains a string of letters that make the word 'Hello'

`print("The variable myString contains: "..myString)`

This line prints out to the console, similar to the 'Hello World' example, except it prints out both the words "The variable myString contains: " and the CONTENTS of the variable `myString`, which was set to 'Hello' in line 1.

The joining (concatenation) is done by using the double dots between the two parts. This is equivalent to using the `+` sign in Python or C#

Lua:                    "Hello " .. "World"  
Python (and C# / Java): "Hello " + "World"

```
print()
```

This just prints a blank line

```
myString = myString.." Goodbye"
```

This uses the two dots again, but this time it takes the original value of the variable myString ('Hello') then adds ' Goodbye' to it, and finally puts the joined words back into the variable myString, so now myString contains 'Hello Goodbye'

```
print("The variable myString contains: "..myString)
```

This line prints out the new value of myString

Note the comments beginning with -- in the screenshot

## Variable Scope

This is a subject usually tackled at a much later date when learning Python, C# or Java, but it is important, so best introduced right at the beginning.

Create a new file called 'Variables with Scope.lua'

Copy/Paste all the code from 'variables.lua' to this file.

Add the word 'local' in front of the line:

```
myString = "Hello"
```

so it becomes:

```
local myString = "Hello"
```

Add this line after main():

```
print("The variable \"myString\" contains: \".. myString")
```

The complete code (without comments) is:

```
function main()
    local myString = "Hello"

    print("The variable myString contains: \"..myString")
    print()

    myString = myString.." Goodbye"
    print("The variable myString contains: \"..myString")
end

main()
print("The variable \"myString\" contains: \".. myString")
```

If you read the code through, the line at the end should print out the contents of the variable again, just like it did on the exact same line in the main() function.

Run it. Whoops!:

```
Program starting as '"C:\Users\Public\PortableApps\ZeroBraneStudio\bin\lua53.exe" -e
"io.stdout:setvbuf('no')"' "O:\Documents\Coding\Lua\Variables With Scope.lua"'
Program 'lua53.exe' started in 'O:\Documents\Coding\Lua' (pid: 13688).
The variable myString contains: Hello

The variable myString contains: Hello Goodbye
C:\Users\Public\PortableApps\ZeroBraneStudio\bin\lua53.exe: O:\Documents\Coding\Lua\Variables
With Scope.lua:14: attempt to concatenate a nil value (global 'myString')
stack traceback:
   O:\Documents\Coding\Lua\Variables With Scope.lua:14: in main chunk
   [C]: in ?
Program completed in 0.07 seconds (pid: 13688).
```

*So what went wrong?*

The script ran ok until the last line.

Delete the word 'local' and try again. It all works perfectly.

Adding the word 'local' makes the variable local to the code block it is declared in.

When the main() function was finished, the variable 'myString' was deleted, as it was local only to main(). When the last line executed, myString did not exist, so it's value is nil, and you cannot join (concatenate) a string with nil, (or add a number to nil), so the error in red above makes sense.

Removing local declared the variable 'global' by default, so all functions can use it

Good coding practice is to use variables carefully, and keep them as local as possible. The main reason for this in a large project is you may inadvertently change the value of a global variable somewhere you did not expect, and nothing works predictably as a result.

*So how can I use a local variable in main() and use it later?*

Easy:

1. add the line:

```
return myString
```

at the end of the main() code block

2. change the last 2 lines from:

```
main()  
print("The variable \"myString\" contains: \".. myString")
```

to:

```
local fromMain = main()  
print("The variable \"myString\" contains: \".. fromMain")
```

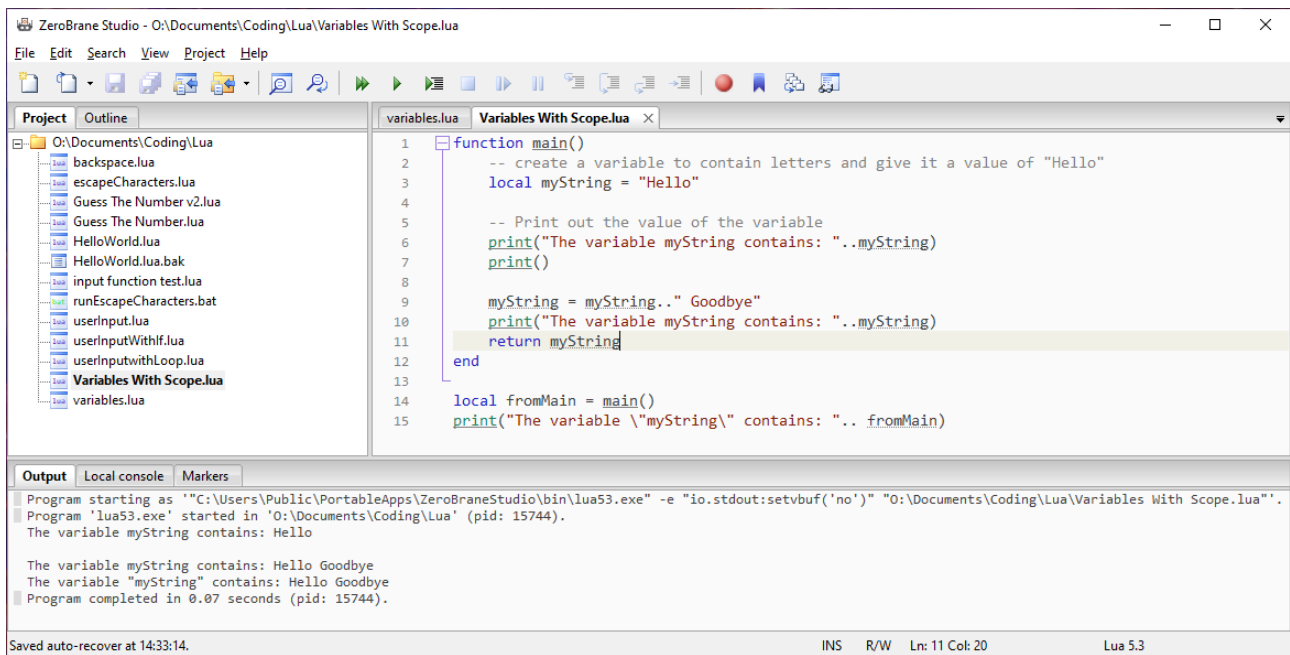
See screenshot below if this looks complicated.

These changes create a variable to store the return value from main() called fromMain

The next line prints out the value of this new variable.

You do not really need to use local in front of fromMain, as it is declared in the script body, it is still useable by all the functions in the script, but it does make it invisible to any other files that make use of this script. Later you will be making multi-file programs, so it is good to get into the habit.

This method of passing data explicitly from one function to another is good coding practice, and is extensively used in most languages.



The scope of a variable declared anywhere in the script without the keyword 'local' has **global scope** by default. It can be accessed from any function in the script and by other scripts within the project.

The scope of a variable declared 'local' in the body of the script has **module level scope**: It can be used by all functions in the script, but cannot be accessed by other scripts in the project.

The scope of a variable declared 'local' in a code block (function, loop etc) has **local scope**. It can only be accessed within that code block.

Important: Once the variable has been 'declared' (used the first time) Do NOT use the local keyword in any subsequent uses, as this will create a new variable of the same name and could over-write the existing value.

Java and C# use the keywords **private** and **public**, which work in the same way. Their variables are private by default.

## Note for Python users:

Python has got everything the wrong way round:

Python has no equivalent of 'local' or 'private': it has a 'global' keyword instead

Python variables are local by default

Python variables declared in the body of the script SHOULD have **module level scope** and be available to functions within the script. They don't. You have to re-declare the variables inside functions using the global keyword.

This is confusing and illogical.

That is Python...

# User Input

Most programs need some sort of input from the user, either from the keyboard, mouse or other device such as a joystick.

Lua and Python cannot handle anything except the keyboard without using other libraries (pre-written code modules), so the next part of the tutorial is based on keyboard input.

If using the Love2D game engine, then the mouse can be used. (Pygame or Tkinter in Python)

Python users will be aware of the `input()` function to get keyboard data.

Lua has a built-in library called `io` (input/output)

To read what the user types on the keyboard use `io.read()`

It is usual to assign the keyboard input to a variable:

```
typedInText = io.read()
```

Create a new file called 'userInput.lua'

type the following lines:

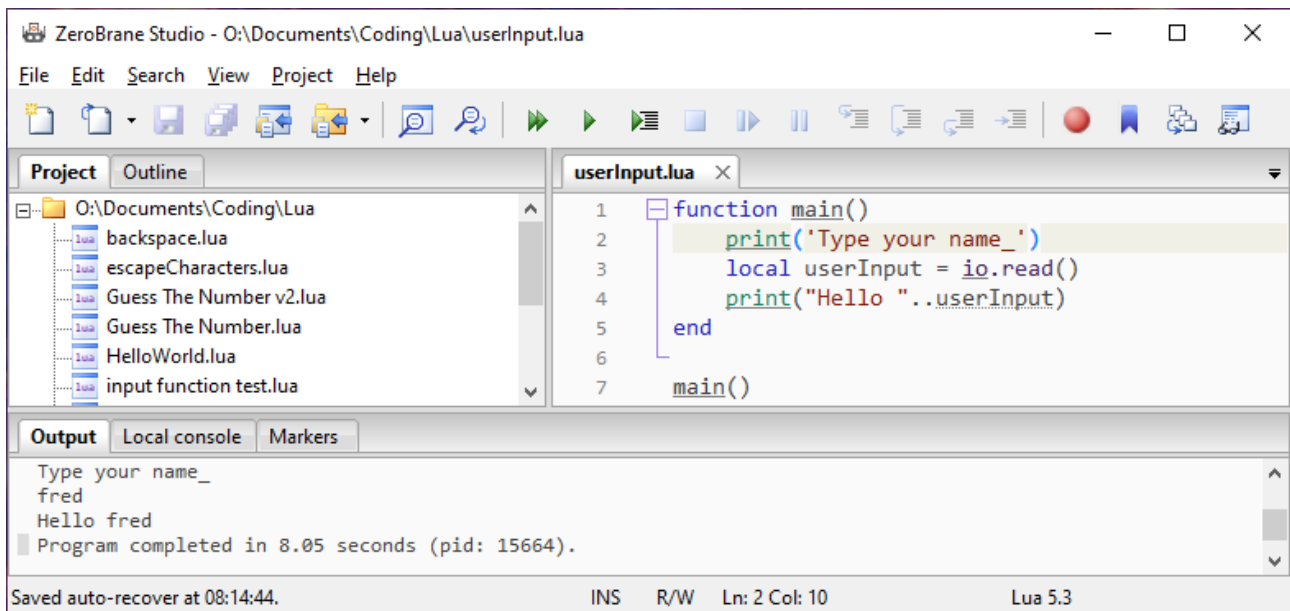
```
function main()
    print('Type your name_')
    local userInput = io.read()
    print("Hello " .. userInput)
end

main()
```

Run the script, then click inside the output window before typing your name.

When you have added your name, press the Enter key.

It is the Enter key which tells Lua you have finished typing.



ZeroBrane Studio - O:\Documents\Coding\Lua\userInput.lua

File Edit Search View Project Help

Project Outline

O:\Documents\Coding\Lua

- backspace.lua
- escapeCharacters.lua
- Guess The Number v2.lua
- Guess The Number.lua
- HelloWorld.lua
- input function test.lua

userInput.lua x

```
1 function main()
2   print('Type your name_')
3   local userInput = io.read()
4   print("Hello " .. userInput)
5 end
6
7 main()
```

Output Local console Markers

Type your name\_  
fred  
Hello fred  
Program completed in 8.05 seconds (pid: 15664).

Saved auto-recover at 08:14:44. INS R/W Ln: 2 Col: 10 Lua 5.3

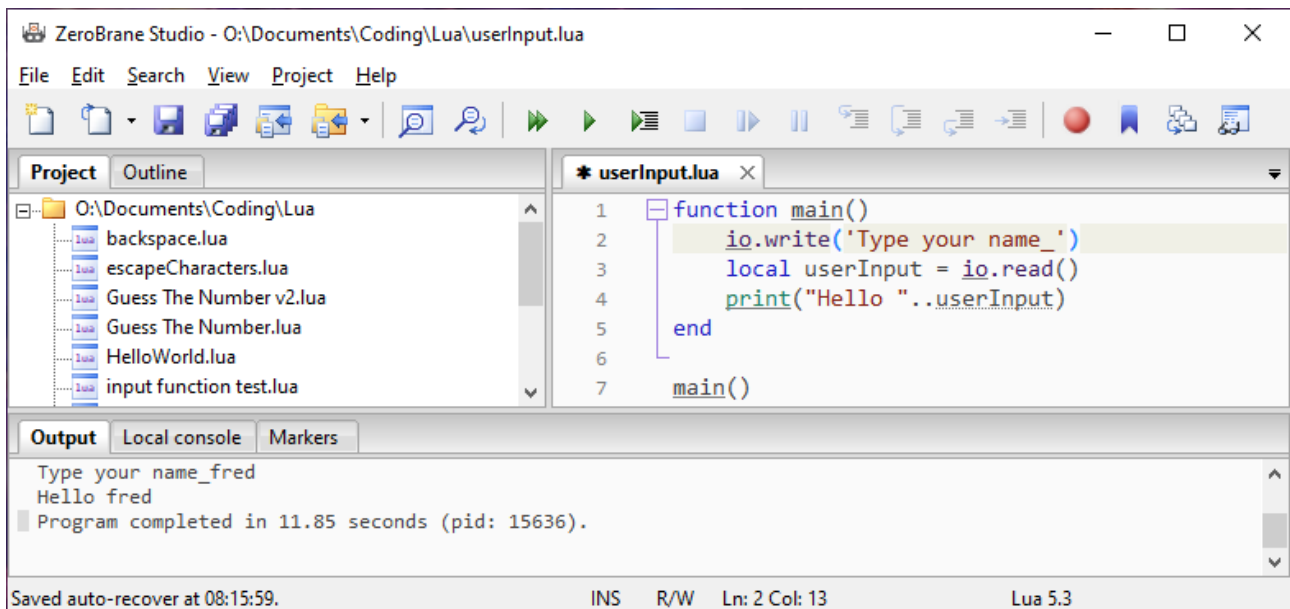
The main problem with this script is the cursor moved to the next line ready for your input. It would work better if the cursor appeared immediately after 'Type your name:'

The reason for this is the 'print' statement automatically adds a 'newline' character (which cannot be seen) that moves the cursor down to the next line.

To overcome this use the `io.write()` function:

Change line 2 in your script to:

```
io.write("Type your name:")
```



ZeroBrane Studio - O:\Documents\Coding\Lua\userInput.lua

File Edit Search View Project Help

Project Outline

O:\Documents\Coding\Lua

- backspace.lua
- escapeCharacters.lua
- Guess The Number v2.lua
- Guess The Number.lua
- HelloWorld.lua
- input function test.lua

\*userInput.lua x

```
1 function main()
2   io.write('Type your name_')
3   local userInput = io.read()
4   print("Hello " .. userInput)
5 end
6
7 main()
```

Output Local console Markers

Type your name\_fred  
Hello fred  
Program completed in 11.85 seconds (pid: 15636).

Saved auto-recover at 08:15:59. INS R/W Ln: 2 Col: 13 Lua 5.3

That's better. The cursor stayed on the same line.

You can also force `io.write()` to move to the next line by adding the hidden newline character.

Change the last line to:

```
io.write("Hello " .. userInput .. "\n")
```

the backslash before n represents the newline character. Similar hidden characters you can play with are:

- `\'` single quote: embed a single quote in the output string
- `\"` double quote: embed a double quote in the output string
- `\\` backslash: embed a backslash in the output string
- `\n` new line: move cursor to a new line
- `\r` carriage return: move cursor back to the beginning of the line
- `\t` tab: move cursor forward (usually) 4 spaces
- `\b` backspace: move cursor back one space
- `\f` form feed: move cursor down to next line

Many of these 'Escape Characters' are based on the actions of typewriters. To understand these fully, ask an old person "what is a typewriter?"

The typewriter had a tab key, which moved the 'carriage' (the roller that held the paper) to the left by 4 spaces, so the next character position was effectively moved 4 spaces forward.

There was also a 'backspace' key which moved the carriage one space to the right, so you could then insert a thin piece of white tape to over-print the incorrect character, and allow you to type the right one. The resulting mess was acceptable at the time.

The console backspace works in the same way: it moves the cursor back one space, but does NOT delete the character in front of it.

There was a prominent lever which did two things:

1. Carriage Return: which pushed the carriage fully to the right, so the left edge of the paper was under the print head
2. Line Feed (Form feed) which rotated the roller to move the paper upwards, so the print head was on a new line.

The newline character `\n` is effectively a combination of 'carriage return' (cr) and 'line feed' (lf) so it moves the virtual carriage to the left edge of the console, then moves it down by one line.

Windows uses both cr and lf to move to the next line, Unix, Linux and others use just use one of them, so 'newline' covers all operating systems.



## Exercise 1: Using Escape Characters

Create a new file called 'Escape Characters.lua'

Add an empty 'main' function

Add main() at the bottom of your script

Add some lines within main() to demonstrate the following:

1. newline: (\n) use io.write() and include \n at the end of the line on all lines in the script.
2. output this text: This line contains 'single quotes'
3. output this text: This line contains "double quotes"
4. output this text: This line contains 'single and double quotes'
5. output this text: This line contains                      2 tabs

To get you started here is a screenshot of the completed program, but you have to fill in the blanks.

Line 4 helps you with the positioning of the newline character.

The output tab shows you the expected output from your program.

The screenshot shows the ZeroBrane Studio interface. The editor displays the following Lua code:

```
1 function main()
2   io.write("This line contains ")
3   io.write("This line contains ")
4   io.write("This line contains \n")
5   io.write("This line contains ")
6   io.write("This line contains ")
7 end
8
9 main()
```

The Output console at the bottom shows the following output:

```
Program starting as "C:\Users\Public\PortableApps\ZeroBraneStudio\bin\lua53.exe" -e "io.stdout:setvbuf('no') " "O:\Documents\Coding\Lua\escapeCharacters.lua".
Program 'lua53.exe' started in 'O:\Documents\Coding\Lua' (pid: 12336).
This line contains 'single quotes'
This line contains "double quotes"
This line contains 'single and double quotes'
This line contains a backslash\
This line contains                      2 tabs
Program completed in 0.07 seconds (pid: 12336).
```

Demonstrating the backspace and carriage return is more difficult, as ZeroBranes' Output console does not follow the rules, and these characters do not work. You can use Lua53.exe directly on non-school computers and type direct commands:

The screenshot shows a command prompt window titled "C:\Users\Public\PortableApps\ZeroBraneStudio\bin\lua53.exe". The output is as follows:

```
Lua 5.3.1 Copyright (C) 1994-2015 Lua.org, PUC-Rio
> io.write("a line\n")
a line
file (76c44620)
> io.write("a line\b \n")
a lin
file (76c44620)
>
```

The line `io.write(a line\b \n)` backspaced after e at the end of line, then printed a space over the e before newline.

# Conditional Statements

These are an essential part of any programming languages

The first keyword is 'if'

You will be asked to use something called **pseudocode** during your studies. This is a code-like approach, but not in the syntax of a specific language. Here is an example:

```
if some condition is true then
    do some code
end
```

This code block will only 'do some code' if some condition checked in the line containing 'if' is true.

A more realistic pseudocode block is:

```
if userInput == "your name" then
    print("That is correct!")
end
```

The double == means 'Is the variable userInput equal to'

(The statement: userInput = would assign a value to the variable.)

Lua is the closest programming language to pseudocode. The block above is actually correct Lua syntax!

The second keyword is 'else'.

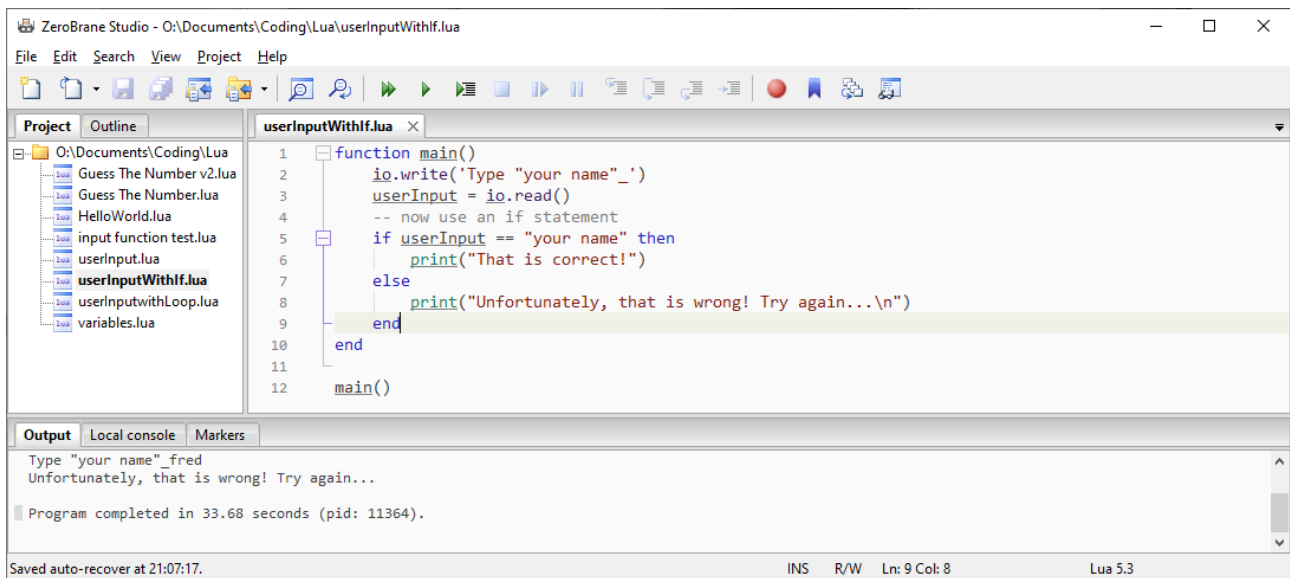
This allows an alternative block of code to run if the original condition is not true.

You can check for an alternative condition by using 'elseif' (Python uses 'elif', C# uses 'else if')

```
if userInput == "your name" then
    print("That is correct!")
elseif userInput == "name" then
    print("That is half correct")
end
```

Create a new script called 'UserInput With If.lua'. Pay attention to the use of single and double quotes on the first line. An underscore has been added to improve the way the program looks when run:

```
function main()
    io.write('Type "your name"')
    local userInput = io.read()
    -- now use an if statement
    if userInput == "your name" then
        print("That is correct!")
    else
        print("Unfortunately, that is wrong! Try again...\n")
    end
end
main()
```



## How does this work?

```
io.write('Type "your name":_')
```

By using single quotes round the whole text string, you can insert double quotes within the string to print out the quotes to the console, so the words “your name” are surrounded with double quotes in the console. This is supposed to give a clue to your user that you want them to type ‘your name’ rather than their actual name.

```
if userInput == "your name" then
```

This uses a conditional, and asks if the value typed in and stored in userInput is equal to ‘your name’.

If the test is true then the output is "That is correct!"

otherwise

output is "Unfortunately, that is wrong! Try again..."

### Exercise 2:

Modify the ‘UserInput With If.lua’ script as follows:

1. Change line 2 to use Escape Characters – `io.write(.....)` to achieve the same result as combining the single and double quotes.
2. Add a couple of lines to check if the user was partially correct and had entered ‘name’ then print an appropriate message. Hint: (elseif)

# Loops

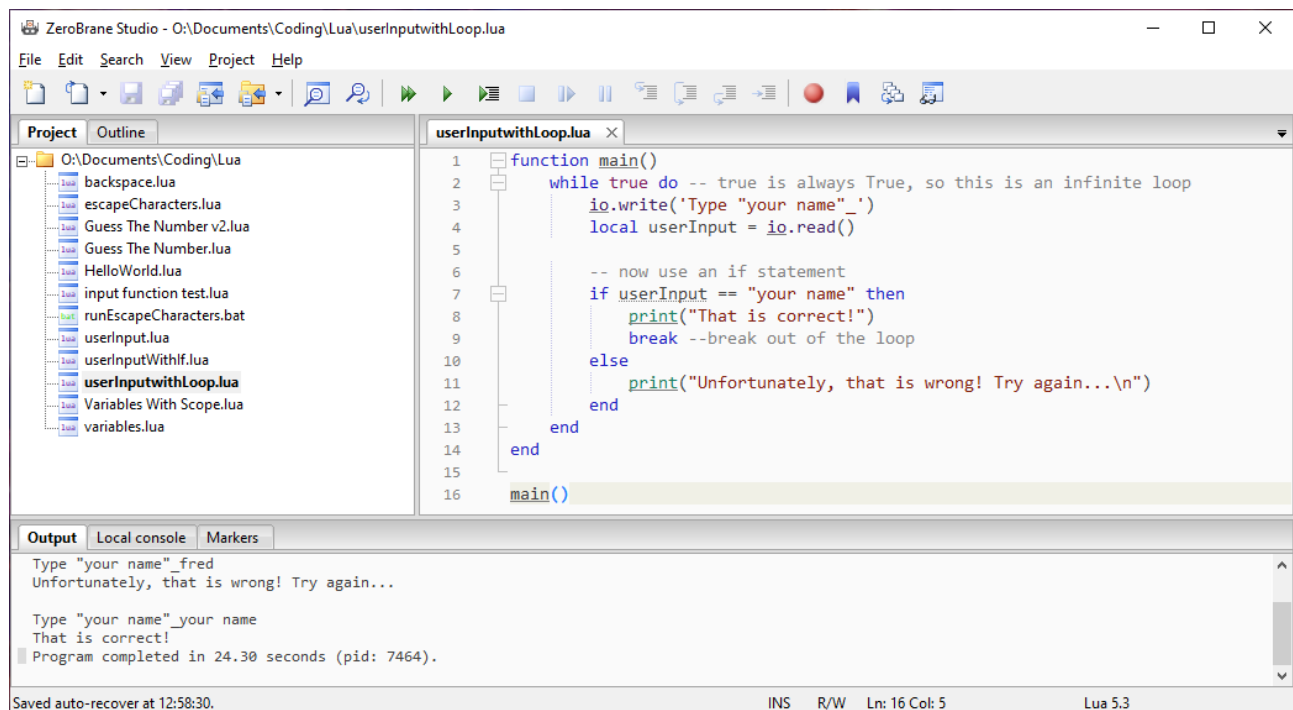
An improvement to this script would be to repeat the process of asking for 'your name' until the user realised and typed it correctly.

For this a loop is required.

Change the script to:

```
function main()
    while true do
        io.write('Type "your name"_)
        local userInput = io.read()

        if userInput == "your name" then
            print("That is correct!")
            break --break out of the loop
        else
            print("Unfortunately, that is wrong! Try again...\n")
        end
    end
end
main()
```



The screenshot shows the ZeroBrane Studio interface. The editor window displays the Lua script for `userInputWithLoop.lua`, which is identical to the code block above. The left sidebar shows a project tree with various files, including `userInputWithLoop.lua`. The bottom panel shows the 'Output' tab with the following text:

```
Type "your name" _fred
Unfortunately, that is wrong! Try again...

Type "your name" _your name
That is correct!
Program completed in 24.30 seconds (pid: 7464).
```

The status bar at the bottom indicates 'Saved auto-recover at 12:58:30.', 'INS R/W', 'Ln: 16 Col: 5', and 'Lua 5.3'.

## How does this work?

The line `while true do` will repeat all the following lines up to its closing 'end' statement (line 13) continuously or until a `break` statement is encountered.

This is called an infinite loop because the while condition cannot change. True is always true.

A more specific while loop could be used, where the condition being checked is what userInput contains:

```
function main()
    local userInput = "" -- empty variable
    while userInput ~= "your name" do
        io.write('Type "your name"_:')
        userInput = io.read()

        if userInput == "your name" then
            print("That is correct!")
        else
            print("Unfortunately, that is wrong! Try again...\n")
        end
    end
end
main()
```

The symbol `~=` means 'not equal'

The line

```
while userInput ~= "your name" do
```

translates to:

while userInput is not equal to 'your name' do

As it was set to an empty string when the loop started, this condition is true, so the loop runs at least once.

If the user types in 'your name', the message "That is correct!" is printed out, but the loop exits because its condition is no longer true. userInput IS equal to 'your name'.

A variation on the while loop is a repeat until loop:

```
function main()
    repeat
        io.write('Type "your name"_:')
        userInput = io.read()

        if userInput == "your name" then
            print("That is correct!")
        else
            print("Unfortunately, that is wrong! Try again...\n")
        end
    until userInput == 'your name'
end
main()
```

This loop always runs at least once, which is the main reason for using it.

Another loop called a 'for' loop will be covered later

# Write Your Own Input Function

If you are a Python user, the `input()` function is in constant use.

It does the same as `io.read()` but has the advantage of combining the `io.write()` function to prompt the user what to type:

#Python input example

```
userInput = input("Type your name:_")
```

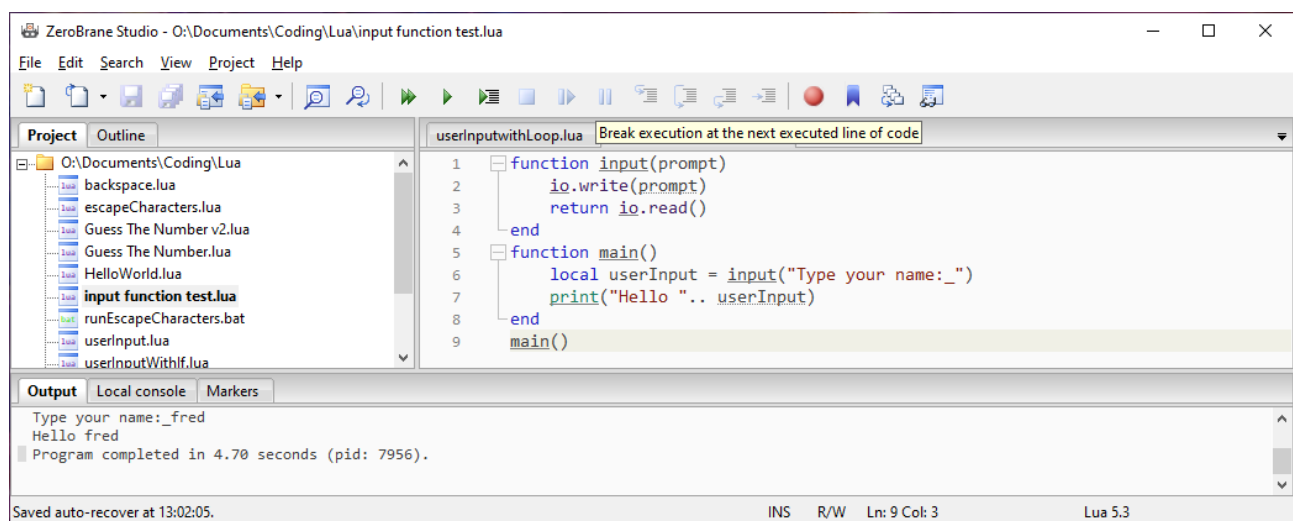
Lua allows you to create your own `input()` function very easily. All functions should be written at the top of the script, so they can be used by the main script later, when required:

```
function input(prompt)
    io.write(prompt)
    return io.read()
end
```

Create a new file called "input function test.lua".

type the following lines:

```
function input(prompt)
    io.write(prompt)
    return io.read()
end
function main()
    local userInput = input("Type your name:_")
    print("Hello " .. userInput)
end
main()
```



## How does this work?

When the Lua interpreter reads your script it will ignore anything that starts with the keyword 'function' until it passes the function's 'end' statement, so lines 1 to 4 and 5 to 8 are ignored in this example.

Line 9:

```
main()
```

This is the first instruction that is not part of a function, so the program starts here with the instruction to 'call' the function 'main', so the interpreter jumps to line 5 where 'main' starts. It moves into the function at line 6:

```
userInput = input("Type your name:_")
```

It sees the word 'input' and then looks for a function with that name, which it finds on line 1.

The words "Type your name:\_" inside the brackets are called 'parameters'. They are passed on to the code in line 1 and stored in the temporary variable called 'prompt':

```
function input(prompt)
```

The function then runs and moves to the next line

```
io.write(prompt)
```

This writes to the console, whatever is stored in the variable 'prompt', which in this case is "Type your name:\_"

The next line waits for user input and Enter key to be pressed:

```
return io.read()
```

The keyword 'return' is used in functions to send data back to the line of code that 'called' the function. In this case it was line 6, so the returned data is stored in the local variable userInput on line 6:

```
local userInput = input("Type your name:_")
```

Finally the program prints out whatever has been stored in the userInput variable:

```
print("Hello ".. userInput)
```

Control returns to line 9 where the program started.

As this is the last line, the program exits.

As soon as the function input() returned its value, the temporary variable 'prompt' was deleted and will be re-created with a new value if the function is used again.

## Random Numbers

Games often use random numbers, and Lua has a method of generating them, using it's maths library

If you want a random number between 1 and 99 use this:

```
local randomNumber = math.random(1, 99)
print(randomNumber)
```

The numbers are not truly random and the same sequence is generated each time the program runs. This can be overcome by setting a seed for the generator, based on the current time:

```
math.randomseed(os.time())
local randomNumber = math.random(1, 99)
```

Create a new lua file called 'Guess The Number.lua'

Type in the following code

```
function input(prompt)
    io.write(prompt)
    return io.read()
end

math.randomseed(os.time())
local n = math.random(1, 99)

repeat
    local guess = tonumber(input("Enter an integer from 1 to 99:_"))
    if guess < n then
        print("Too low")
    elseif guess > n then
        print("Too high")
    else
        print("you guessed it!")
    end
until guess == n
```

Note the use of the input function

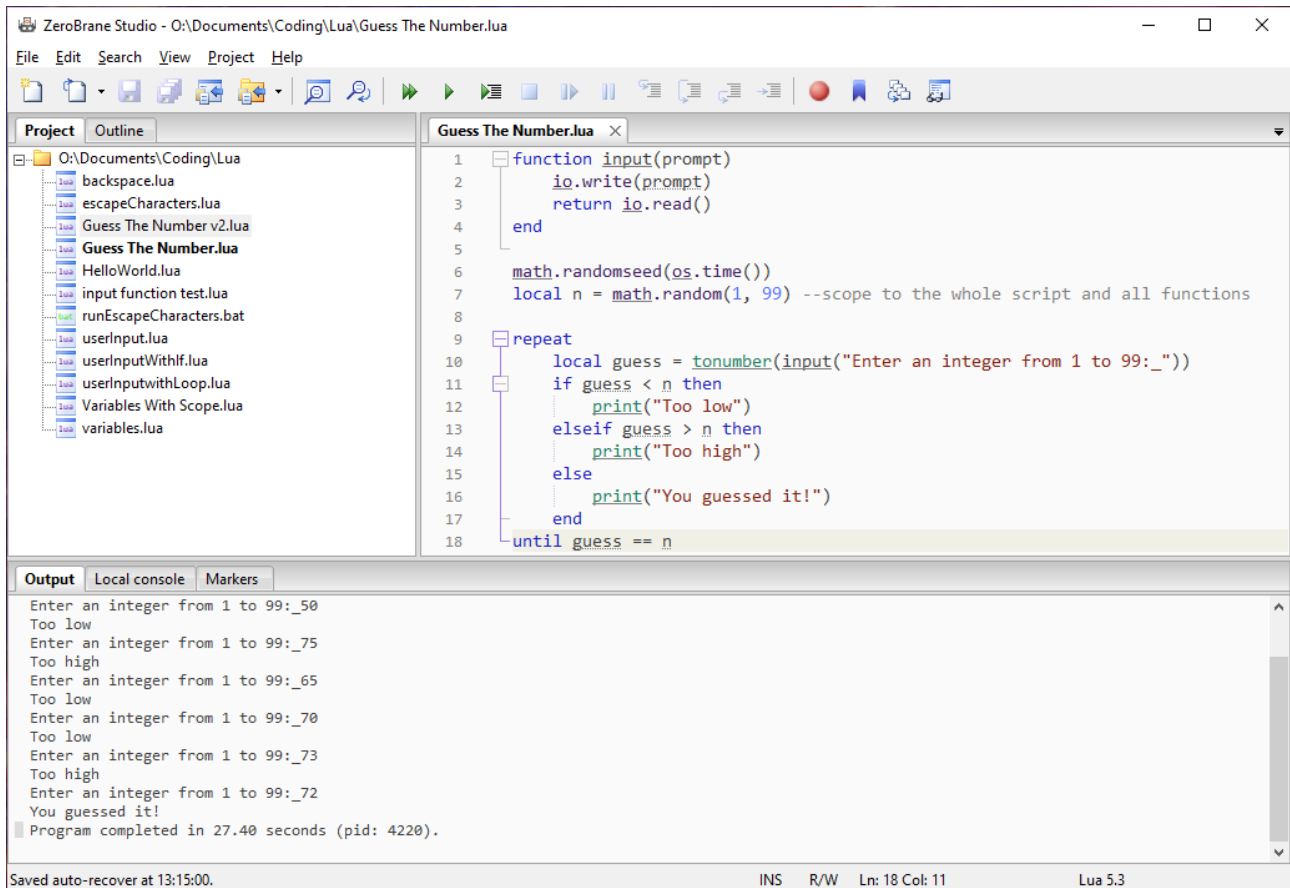
Note the use of the random number generator to give a number between 1 and 99

Note the different loop construction, which is less well known. This is the 'repeat until' loop, which forces it to run at least once. The loop breaks when the guessed number equals the one generated.

Note the `tonumber()` Lua function, which converts a string into a number. This should take the characters typed in and convert them to a number so the computer can compare the values mathematically. It could have been written as 2 separate lines:

```
local guess = input("Enter an integer from 1 to 99: ")
guess = tonumber(guess) -- only works in Python / Lua where variables can hold any type of data
```





Run it again, but this time type in a letter or spell out a number eg: 'ten' instead of a number:

Whoops! This is what appears in the Output tab. The error has been highlighted in red:

```
Enter an integer from 1 to 99: ten
C:\Users\Public\PortableApps\ZeroBraneStudio\bin\lua53.exe: O:\Documents\Coding\Lua\
Guess The Number.lua:11: attempt to compare nil with number
stack traceback:
  O:\Documents\Coding\Lua\Guess The Number.lua:11: in main chunk
  [C]: in ?
Program completed in 4.87 seconds (pid: 11128).
```

The cause of this is the function `tonumber("ten")` returned `nil`, a sign the function did not work (not surprising!)

To make your script idiot-proof, amend the code to deal with user stupidity:

```

repeat
    local userInput = input("Enter an integer from 1 to 99:_")
    local guess = tonumber(userInput)
    if guess == nil then
        print(userInput.." is not a number")
    else
        if guess < n then
            print("Too low")
        elseif guess > n then
            print("Too high")
        else
            print("You guessed it!")
        end
    end
end
until guess == n

```

The screenshot shows the ZeroBrane Studio interface. The main editor displays the Lua code for 'Guess The Number v2.lua'. The code includes a custom input function, a random number generation setup, and a loop that prompts the user to guess a number between 1 and 99. The output window shows the program's execution, including prompts and user input.

```

1  -- custom Pythonesque input function
2  function input(prompt)
3      io.write(prompt)
4      return io.read()
5  end
6  -- setup random number between 1 and 99
7  math.randomseed(os.time())
8  local n = math.random(1, 99) --scope to the whole script and all functions
9
10 repeat
11     local userInput = input("Enter an integer from 1 to 99:_")
12     local guess = tonumber(userInput)
13     if guess == nil then
14         print(userInput.." is not a number")
15     else
16         if guess < n then
17             print("Too low")
18         elseif guess > n then
19             print("Too high")
20         else
21             print("You guessed it!")
22         end
23     end
24 until guess == n

```

Output (running):

```

Program 'lua53.exe' started in 'O:\Documents\Coding\Lua' (pid: 6432).
Enter an integer from 1 to 99:_ten
ten is not a number
Enter an integer from 1 to 99:_50
Too low
Enter an integer from 1 to 99:_75
Too high
Enter an integer from 1 to 99:_72
Too high
>>>Enter an integer from 1 to 99:_

```

Saved auto-recover at 13:22:11. INS R/W Ln: 24 Col: 17 Lua 5.3

## How does this work?

Instead of trying to convert the keyboard input to a number in the same line of code, a new local variable 'userInput' is used to store exactly what is typed. Then guess is assigned to the output of tonumber()

If the user types a non-convertible string eg 'ten' then the output returned from tonumber() is nil.

The next line checks if the converted input is nil and prints the appropriate message.

Note the print statement uses the original keyboard input stored in 'userInput' NOT 'guess'

# Lua Tables and for loops

So far you have used simple variables to store a single string, number or boolean value.

But what if you wanted to store a list of strings?

Or need some kind of super-variable that could hold many different types of data?

A simple example would be to write a program to ask the user for their name, and compare it to a list of names already stored in memory. If they are on the list, they are welcome to continue, otherwise the program quits.

Python, C# and Java use Arrays, Lists and Dictionaries to hold this data:

Python List: `myFriends = ["Fred", "Alice", "Jim", "Karen"]`

C#, Java: `List<string> myFriends = new List<string>{"Fred", "Alice", "Jim", "Karen"};`

Lua: `local myFriends = {"Fred", "Alice", "Jim", "Karen"}`

Use of the curly braces `{}` is all that is needed in Lua, but it is not an array, or a list or a dictionary. It is a table.

You can use a numerical index to read/write to specific parts of the list/table:

Python, C# Java: `myFriends[0] = "Fred"`

Lua: `myFriends[1] = "Fred"`

The only difference is the start of the index. **Lua's index starts at 1, all other languages start at 0**

Lua tables can also be used like this:

```
local myFriends = {} -- empty table
```

```
myFriends["Best"] = "Fred"
```

```
myFriends.SecondBest = "Alice"
```

The closest equivalent to this in other languages is the Dictionary:

Python dictionary: `myFriends = {"Best": "Fred", "SecondBest": "Alice"}`

Python has a very useful 'in' keyword to check if an item is in a list:

Python:

```
myFriends = ["Fred", "Alice", "Jim", "Karen"]

myName = input("Type your name")
if myName in myFriends:
    print("Hello Friend")
```

Although Lua has an 'in' keyword, it does not work in the same way, so you have to loop through the whole table to check if the item is in there.

Create a new file called 'First Table.lua'

```
function input(prompt)
    io.write(prompt)
    return io.read()
end
function main()
    local found = false
    local myFriends = {"Fred", "Alice", "Jim", "Karen"}
    local userName = input("Please type your first name_")
    for i = 1, #myFriends do
        if userName == myFriends[i] then
            found = true
            break
        end
    end
    if found then
        print("Hello friend")
    else
        print("I do not know you")
    end
end
main()
```

The screenshot shows the ZeroBrane Studio IDE with the file 'first Table.lua' open. The code in the editor matches the code block above. The left sidebar shows a project tree with various Lua files. The bottom panel shows the 'Output' tab with the following text:

```
Please type your first name_fred
I do not know you
Program completed in 6.14 seconds (pid: 14696).
```

The status bar at the bottom indicates 'Saved auto-recover at 14:57:24.', 'INS R/W Ln: 22 Col: 7', and 'Lua 5.3'.

Note when running the program and entering 'fred', it did not find it in the list, but it is the first entry in the list, so what is going on?

Easy: 'fred' is not equal to 'Fred'

fix: check using lowercase conversions:

```
if string.lower(userName) == string.lower(myFriends[i]) then
```

`string.lower()` is part of the string library and can be used to convert a string to lower case. `string.upper()` is used to convert to upper case.

So how does this work?

You are already familiar with the input function, and the use of a table to hold this list of names.

The for loop is yet another type of loop.

This one runs for a specific number of iterations, which is set in the first line:

```
for i = 1, #myFriends, 1 do
```

# in front of a table name means 'number of items in' tableName

The logic of this line is: Run the following code for a fixed number of loops by using a local counter variable that starts with a value of 1, increases by one every iteration, and finishes when the value reaches the number of items in the table called myFriends.

Inside the loop the code checks each value of the myFriends table in turn:

```
if string.lower(userName) == string.lower(myFriends[i]) then
```

Note the local variable (local by default when used in a for loop) 'i' is used as the index for the myFriends table and goes through each value in turn.

If the value is found then the boolean variable is changed from its initial value of false 'to true' and the loop is stopped with the break statement.

When the for loop has finished or broken, the next lines print a message depending on whether found is true or false.

Note the line:

```
if found then
```

is a short-hand version of:

```
if found == true then
```

Similarly:

```
if found == false then
```

can be shortened to:

```
if not found then
```

Adding and removing items to a table

To add a new value "John" to myFriends use: `table.insert(myFriends, "John")`

To delete "Alice" from myFriends, (you already know her index is 2) use: `table.remove(myFriends, 2)`

## Exercise 3: Add and delete items from a table

This exercise gives you a skeleton file with the `main()` function completed and the names of other functions required. It makes use of conditional statements, for loops, while loops, repeat until loops, tables, variable scope, return values from functions.

Create a file called 'Table Methods.lua'

1. Create an empty table called 'myFriends' with Module Scope.
2. Add the `input()` function you have used before.
3. Create a function called `getTableIndex(name)` -- note parameter
4. Create a function called `addToTable()`
5. Create a function called `deleteFromTable()`
6. Create a function called `displayTable()`
7. Create a `main()` function and call it on the last line of the script

Code for the main function:

```
function main()
    -- Add 5 names to empty table
    local count = 0
    repeat
        if addToTable() == 0 then -- 0 = name added, >0 = entered name exists
            count = count + 1
        end
    until count == 5
    displayTable()
    --remove 1 name
    repeat until deleteFromTable() > 0 -- 0 = name not found >0 = name found and deleted
    displayTable()
end
```

Hints for the functions:

`getTableIndex(name)`

loop through the table with a for loop.  
compare the value in the parameter name with each item in the loop  
If there is a match, store the index of the matching item  
return either 0 for not found, or the index of the match

`addToTable()`

use the `input` function to get a name from the user  
use the `getTableIndex(name)` function to check if it has already been entered  
if already present display message and return the index  
else insert the name into the table, display message and return 0

`deleteFromTable()`

use the `input` function to get a name from the user  
use the `getTableIndex(name)` function to check if it exists  
if present display message, remove it from the table and return the index it had before deletion  
else display message and return 0

`displayTable()`

loop through the table with a for loop.  
Print the index, a couple of spaces and the value

Use the screenshot below to help

The screenshot shows the ZeroBrane Studio interface. The top menu bar includes File, Edit, Search, View, Project, and Help. The toolbar contains various icons for file operations and execution. The left sidebar shows a project tree with files like backspace.lua, escapeCharacters.lua, first Table.lua, Guess The Number v2.lua, Guess The Number.lua, HelloWorld.lua, input function test.lua, menu.lua, runEscapeCharacters.bat, Table Methods.lua, userInput.lua, userInputWithIf.lua, userInputwithLoop.lua, Variables With Scope.lua, and variables.lua. The main editor displays the code for Table Methods.lua, which includes functions for input, getting table index, adding to table, deleting from table, displaying table, and a main function. The output console shows the program's execution, including prompts for adding and deleting names, and the current state of the table.

```
1 local myFriends = {}
2 function input(prompt)
7 function getTableIndex(name)
18 function addToTable()
30 function deleteFromTable()
42 function displayTable()
49 function main()
50     -- Add 5 names to empty table
51     local count = 0
52     while count < 5 do
53         if addToTable() == 0 then -- 0 = name added, >0 = entered name exists
54             count = count + 1
55         end
56     end
57     displayTable()
58     --remove 1 name
59     repeat until deleteFromTable() > 0 -- 0 = name not found >0 = name found and deleted
60     displayTable()
61 end
62
63 main()
```

Output:

```
Type the name you want to add_Adam
Adam added to the table
Type the name you want to add_Bill
Bill added to the table
Type the name you want to add_Adam
Adam is already in the table
Type the name you want to add_Charlie
Charlie added to the table
Type the name you want to add_Anne
Anne added to the table
Type the name you want to add_Mary
Mary added to the table
Table contents:
1 Adam
2 Bill
3 Charlie
4 Anne
5 Mary
Type the name you want to delete_john
john not found.
Type the name you want to delete_anne
anne deleted from the table
Table contents:
1 Adam
2 Bill
3 Charlie
4 Mary
Program completed in 94.91 seconds (pid: 1292).
```

Saved auto-recover at 19:28:14. INS R/W Ln: 62 Col: 1 Lua 5.3

This tutorial should be enough to get you started.

Recommended Youtube:

[https://www.youtube.com/watch?v=Us46grT9wsA&list=PL0o3fqwR2CsWg\\_ockSMN6FActmMOJ70t](https://www.youtube.com/watch?v=Us46grT9wsA&list=PL0o3fqwR2CsWg_ockSMN6FActmMOJ70t)

Search for KarmaKilledTheCat, series on his Playlist