Jeroen P. Broks

A beginner's guide to:

Table of Contents

1. Introduction to Lua and this book

1.1 What is Lua?

Lua is a scripting language, which works completely in an interpreted environment. It has been coded in C and has been set up to be easily implemented with any program written in C. It's been used for many kinds of application. For example for creating addons for utilities, and most of all for games.

Lua was designed by Roberto Ierusalimschy, Waldemar Celes and Luiz Henrique de Figueiredo in a university in Rio de Janeiro in Brazil, and is completely free. It was named after the moon as “lua” means “moon” in Portuguese. Despite Lua itself being set up in C, you don't need any knowledge of C in order to use it, due to language quite often being implemented in tools in which you may need it.

Lua is very extremely symplistic in its syntax and general setup, making it very easy to learn and understand, even when you never touched a programming language before. Still Lua has a kind of low-level approach and is despite being interpreted quite fast. Since it's specifically designed to work in a C environment, it's very popular in the professional industry and knowing the basics of Lua can get you some extra tickets into the professional gaming industry.

I will go into the deep of how Lua works later in this book, but the famous “Hello World” program is only one line: “print('Hello World!')”.

Lua has also been designed to not bother about platform specific issues. So basically a Lua script should work in Windows, MacOS and Linux and basically any platform you can think of.

1.2 What do you need to install for this course.

When it comes to Game programming one of the most complete Lua engines to make anything is LÖVE (http://love2d.org) but I advise you not to start with LÖVE right away. Before we get onto a complete engine like that it's essential that you first understand how Lua itself works. For that you need a Lua program you can run from the CLI. This program can only perform the core features, but it's good enough for testing what you've learned. If CLI tools are too difficult for you to understand, you can also use the Lua playground (https://www.lua.org/cgi-bin/demo). You can just enter the Lua code and the result will appear as soon as you click “run”. For training this will do.

To Test the playground just type “print('Hello World!')” in the textarea and then click “Run”, and “Hello World!” should appear below. When you use the cli tool type “cat > hello.lua” in the terminal when you use Linux, Mac or BSD and “copy con hello.lua” on Windows and type

“print('Hello World!')” and press enter and press ctrl-D in Linux/Mac/BSD and ctrl-z on Windows then type “lua hello.lua” and if you see “Hello World!”, you can see you installed the CLI tool correctly.

When you really get serious into Lua scripting you will need an editor that can handle Lua Scripting. I use the Lua Devolopment Tools for Eclipse, but that can be a rather complex to understand. Atom, Geany, SublimeText, NotePad++ should all be able to allow you to work with Lua scripts.

1.3 How to use this book

I will try to take you step by step into the secrets of Lua. I will show you many scripts. It could be wise to copy these and try what they do.

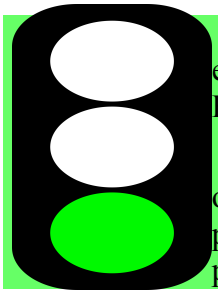
Code will look like this:

```
-- Test
a="Hello World"
print(a)
```

The colors you see is called “Syntax Highlight”. This will help you a lot keep some overview over your code.

When you see this icon and the text beside it with a blue background, it means that I got something very important to tell. Like all programming languages Lua can be full of trapdoors you can easily fall through, and you should therefore be aware of those. I make these warnings for a reason.

When you see this icon on an amber background, it means that I am about to tell you some typical nerdy stuff, that is just nice to know, but most of all aimed to people who can look beyond the regular stuff. If you have no understanding of what I tell here, then don't you worry one bit, as it's not really important to move on. Perhaps if you got more understanding of Lua and you want to read this over, hey, why not ;)



And lastly when you see this icon on a green background, I'll be giving you some exercise. The best way to learn to code, no matter if you do that in Lua, Pascal, BASIC or even in C is by doing it. And thus a few assignments never hurt.

These assignments may sometimes have some code snippets, but mostly you are on your own, but you area allowed to look things up in the previous chapter or paragraphs or sections to see the solution. Sometimes I may even give you some pointers of the expected output, so you can check a few things for yourself.

And with this all discussed, I wish you good luck with this book.

2. Hello World! Your first real program.

Most traditionally the first program you'll be writing in whatever language you're gonna learn is Hello World! It's been said that the first time Hello World was written was to demonstrate a prototype of the C programming language.

In C Hello World looks like this:

```
#include <stdio.h>
int main(void){
     printf("Hello World\n");
}
```

Not really much to it, even in C, although in C you need 4 lines and in Lua, just one.

```
print("Hello World")
```

Now “print” is a function. I'll go into the deep about what functions are later in this book, but for now it's important to know that a function can be used to make Lua do something. In the case of “print” that is to put something on the screen. “Hello World” is was we call a string. A string is a series of characters, usually used to form text. So in English we said to Lua *put the text “Hello World” on the screen.*

Let's test this out. Copy that line into a program or onto the playground and run it and see what happens. It should show “Hello World”.

When it comes to strings in nearly any language you will need quotes. In the example I used double quotes. Lua will also understand single quotes. It doesn't really matter if you use single or double quotes, as long as you are consequent in using the same quotes to end the quotes as you started them. I however advice double quotes. If you will ever consider moving on to C, Go, C# and some other languages in this family single quotes have a different meaning, and thus you can confuse yourself when you use single quotes (big exception is Pascal that requires single quotes).

Now I'm gonna take you into something important. Especially when you coded in BASIC or Pascal before you can really suffer here. Lua is case sensitive.

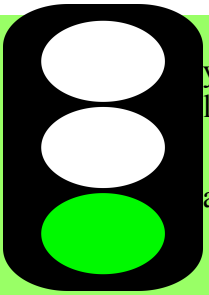
```
PRINT("Hello World")
```

This code will therefore not work. Try it and you will see Lua complain about a trying to call a “nil value” or something like that. I shall go into the deep about what a “nil value” is later, but for now let's suffice to say that a nil value means that you are trying to do something with something that does not exist, and therefore Lua can't handle that. Lua does make a difference between upper and lower case letters. So “print” and “PRINT” are therefore not the same. Especially when you move on to more complex Lua scripting you should be very strict on yourself on how you use upper and lower case or your scripts can become a big mess.

Now with this knowledge you can take this a little bit further. “print” will not only show text on screen. It will also move to the next line. And thus we can also do this:

So without trying this out in the playground or a cli tool, you must be able to tell what this program does:

```
print("USA stands for United States of America")
print("Its first president was George Washington")
print("It borders to Canada and Mexico")
```



Now write a program that shows your name and your age, your date of birth and your city of birth onto the screen. All data fields I asked for should have their own line in the output.

If you can do this, you have understood the basic of the Hello World sequence and then we can get on the move for the next chapter.

3. Variables

Like any programming language Lua handles data most of all through variables.

Unlike compiler based languages such as C, Go and Pascal, Lua does not require any variable declarations. A variable is technically created on the moment it's first asked for.

Variables can be defined, and read out. Read data can be put on the screen, or be used for creating new data, or be checked.

In this particular chapter I'll limit stuff to definitions of variables, reading and some nice manipulations.

Strictly speaking Lua only has 5 data types for variables: nil, boolean, string, number, table and function. I will not discuss function in this chapter. I will get to that once we'll really go into the deep of functions, and for tables same story.

Variables can also be global and local. I will get to the difference between the local and global in a later section. For now we'll only use globals. Global variables are available in your entire script, and that is all you need to know for now.

Definining variables is as easy as this:

```
a = 1 -- number
b = true -- boolean
c = "Hello World" -- string
d = { 3, 4, 5, 10 } -- table
e = function() print("Yo!") end -- function
n = nil -- nil
```

Now “--” in Lua means a comment. Lua will ignore everything that comes after that.

3.1 The nil type

Nil is a word that you'll soon hate when you get into Lua coding. It will haunt you wherever you go. Any variable you did not yet define will automatically be of the nil type, and the nil type only has one “value”. Nil.

Nil just means your variable contains no data at all. In most cases nil cannot be used and thus an error will pop up if you try.

Nil can also be enforced like you can see in the code in the intro section of this chapter by simply saying “myvar = nil”. Especially when working with tables this can get you far as this may cause Lua to automatically clean up all the memory taken by the table. More about that when I actually come to tables.

3.2 The number type

The number type is like the name suggests for storing numbers. These can be integers as well as floats. Now number variables are pretty important as they allow you to perform mathematical calculations, and even the simplest of programs (well except maybe Hello World) will often require you to do some calculations. Scoring points, deducting hitpoints, but also determining coordinates of the player or the enemies. The number type makes it all possible.

3.2.1 Definition and showing numbers

```
a = 1
print(a)
```

Yeah, well it's really that simple. This program will define 1 into variable a. And print will then show the content of a.

Note that now that we are using a variable and not a string we do not quote it. Due to that Lua understands this is a variable and will thus just show the value of variable a

```
a = 1
b = 5
print(b)
print(a)
```

Now 1 will be put in a and 5 will be put in b. Note that I first asked to show variable b and then variable a. This will cause Lua to first output 5 and then 1. It is very important that you put the commands in the correct order to prevent bugs. In this case it didn't matter in which order I defined the variables, but the order of the print instructions do.

3.2.2 Mathematics with variables

Now we get into actual programming.

For this I'll need to discuss operators. For mathematic formulas Lua has 6 operators at your disposal.

Addition	+
Subtraction	-
Multiplication	*
Division	/
Modulo	%
Empower	^

Lua does take in calculations the official order into account when multiple operators are used, so first empowering, then multiplication and division and lastly additions and subtractions, but just like in normal mathematics you can alter this order with parents ( ).

Now I'm sure this all sounds a bit like abracedabra. So let's explain by means of some examples.

When doing math in Lua you can use both numbers and variables in your formulas and you can use them in both your definitions as in function arguments, so you can do this:

```
print( 2 + 5 )
```

And you can also do this:

```
a = 2
print(a + 5)
```

Both examples will put 7 on the screen.

But this works too:

```
a = 2 + 5
print(a)
```

And yeah, you can also go this way:

```
a = 2
b = 5
c = a+b
print(c)
```

Anything is possible.

Now that's interesting, but can you also make a variable increment itself? Sure!

```
a = 1
a = a + 1
print(a) -- outputs 1
a = a + a
print(a) -- outputs 4
```

And basically you can go all the way with this with the other operators.

The “%” operator or “modulo” (in some programming languages this is the “mod” keyword) is an operator that can confuse some people. It contains the remainder value of a division.

8 % 4 = 0

9 % 4 = 1 (8:4=2 and 1 remains).

If you have been using C/C++/Go/C# before you will not like to hear this, but Lua has no shortcut incrementors or manipulation support. So a++ or a-- do not exist in Lua and there is also no variant for a+=4 or anything like that. I'm afraid this is something you'll have to deal with.

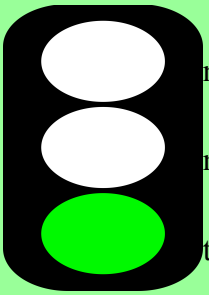
I've not been informed by any plans of support for this in the future. I wouldn't be surprised if it will be implemented, but for the good of this book, I'll do without them. :(

Now like I said before, Lua can handle more complex formulas by working with parents, in which like in real-life mathematics the stuff between parents takes priority

```
a = 4
b = 6
c = 2
d = (a+b)/c
print(d) -- outputs 5
```

Since a is 4 and b is 6 and c is 2 the formula send to d is basically *(4+6)/2*, so first calculate what is between the parents, well 4+6=10, so that makes 10:2 which is 5 and thus the output.

3.2.3 Assignment



Write a program in which you have variable a and b are defined with random numbers of your choice, as long as it's anything higher than zero.

Then let it output the result of adding the two variables, then subtract then multiplied by each other and then divided.

Then make the same program make the two variables double themselves and do the same sequence of mathematic results again.

Please make sure you read the assignment well and that you've understood it well, as this may be one of the more complex ones to understand at once.

3.3 The string type

3.3.1 An introduction to strings

Strings are in Lua pretty easy to use. Like I said strings are a chain of characters, usually used for text. Technically strings can have any length and you can do various things with them.

Defining a string variable is as simple as this:

```
mystring = "Hello World"
```

It's that easy. Now you can just use functions like print to put the content onto the screen.

```
mystring = "Hello World"
print(mystring)
```

Copying a string to another variable does not require special functions (like strcpy in C), but can just be done as easily as this:

```
mystring = "Hello World"
mystring2 = mystring
print(mystring2)
```

3.3.2 Concatenation

Now in Lua you cannot do any mathematic things on strings (seems logical, but there are some languages (like JavaScript) that can under certain circumstances, so that's why I explicitly note it), but there are ways to do some manipulations on strings.

One of the most common actions you can do with strings is concatenation, which is, simply appending a string to a string. Maybe that didn't make much sense, so let's demonstrate.

```
h = "Hello"
w = "World"
hw = h..w
print(hw)
```

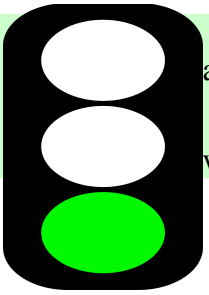
So we placed “Hello” in h and “World” in w. In the third line we concatenate these strings and the result is stored in hw.

Now when you copied this program to the playground or a cli tool you will see something went “wrong”. The output is “HelloWorld” and not “Hello World”. This is not Lua's fault, but mine. I didn't put a space in the original variables. Spaces are just characters just like letters, so if you want them you must put them in, and if you don't want them, leave them out. Unwanted spaces or spaces not appearing when they should can easily happen in careless handing of strings, and sometimes the results are pretty downhearting and heart to find when debugging.

Now the funny part to concatenation is that Lua even allows numbers to be concatenated into strings.

```
s = "I am "..40.." years old"
print(s)

s = "I was born in "
y = 1975
s2 = s..y
print(s2)
```

Both examples work just fine.



Create a program in which one variable contains your name as a string and another variable that contains your age (in years) as a number.

Create a third variable that concatenates the variables in a string where the out put will be “My name is <name> and I am <age> years old.”

3.3.3 C formatted strings in Lua

Now the C junkies reading this will very likely prefer C-formatting for concatenation, and therefore it's good to know that Lua has full support for this. There are basically 2 ways to do this, and it just comes down to what you think is best.

There is no variant to printf, but rather to sprintf and the result is just returned as a string. Two examples to do this:

```
name = "Jeroen"
year = 1975
mystring = string.format("My name is %s and I was born in %d",name,year)
print(mystring)
```

This is the simple way to do this. Of course to a real C programmer it goes without saying that you can use functions as parameters, so you can also do print(string.format(blah blah)).

There is also a kind of OOP way to do this and that'll look like this:

```
name = "Jeroen"
year = 1975
mystring = ("My name is %s and I was born in %d"):format(name,year)
print(mystring)
```

Now this way to go is also completely valid. Please note the (and the) in which the format string is set *is* required, or Lua will throw an error.

3.3.4 String Slicing

String slicing has always been an important thing you can do with strings. And that's why Lua too has support for this. Lua uses the string.sub() function for this.

The basic syntax for string.sub is as follows:

`string.sub(`*`string`*`,`*`startsubstring`*`,`*`endsubstring`*`)`

Now this is pretty vague, I know.

When you are new to programming let me first tell you what string slicing is. It's nothing more or less than cutting or copying a part out of a string and make a new string out of that.

I guess that made even less sense, so let's demonstrate with some examples:

```
s = "I am Jeroen"
print(string.sub(s,6)) -- outputs "Jeroen".
print(string.sub(s,1,4)) -- outputs "I am".
print(string.sub(s,6,8)) -- outputs "Jer".
print(string.sub(s,6,6)) -- outputs "J".
```

Of course, the question is if you have understood what this all does.

Let's ignore the first print line for now, I'll get back to that later. The second line I did request the

characters on spots 1 till 4. Well:

1. Spot #1 contains “I”
2. Spot #2 contains a space
3. Spot #3 contains “a”
4. Spot #4 contains “m”

And the combined result is returned and thus the string “I am” was formed this way. Now all the two lines below that line basically perform the same trick.

In the first print line you may have seen that I did not give a number for the ending spot. And yet the line outputs “Jeroen”. When no ending spot is given Lua will assume that the last spot of the string will be the ending spot, and since the first letter of “Jeroen” was on spot #6, it therefore puts in all letters coming next.

Now there's another trick... Negative numbers in string.sub()

```
s = "I am Jeroen"
print(string.sub(s,-6)) -- Outputs "Jeroen"
print(string.sub(s,-6,-2)) -- Outputs "Jeroe"
print(string.sub(s,-6,8)) -- Outputs "Jer"
```

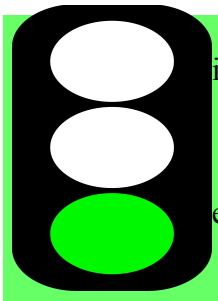
Well do you have any idea what just happened here?

Indeed when negative numbers are used, it will count from the right in stead of the left. So -6 means 6th spot from the right. Since the last word “Jeroen” contains 6 letters, you can easily guess why my name was outputted. And I did not give an ending number so Lua did just assume I wanted to go to the end of the string.

Well the -6 -2 combination is now easy to guess. From the 6th spot from the right (J) to the 2nd spot from the right (e) and thus “Jeroe” was outputted. Easy, huh?

The last one is put in to show you that negative numbers and positive numbers can just be combined. From the 6th spot from the right (J) to the 8th spot from the left (r) and thus “Jer” was outputted.

Now Lua will not moan or whine or anything when you come up with an impossible combination like from 6 to 5 (from the left). Whenever you come up with impossible combinations, string.sub will just return an empty string. As a programmer you are fully responsible for giving correct digits to string.sub for proper output!



Now create a program with a string variable containing the string “Donald Trump is the 45th president of the United States”.

Then make it output “Donald” by slicing out that name.

Then make it output “States” simply by slicing out from the right without an ending point.

Then make it output “Trump” by slicing out that name from the left.

And lastly (and that may be a tricky one) make it output “United” simply by slicing from the right only.

3.4 Boolean type

Now the last type I will cover in this chapter (as I will go into the deep of tables and functions types later) is the boolean type. Now “boolean” is not really coming from an existing word like the names of the other types. It was named after the English mathematician and philosopher George Boole, who lived from 1815 until 1864.

Boolean types can be pretty abstract when you are new to programming, but they might well be one of the most important types in the history of programming.

A boolean type can only have two values. “true” and “false”.

Now at first sight, this may seem pretty pointless, however boolean values are often generated through expressions. When I'm gonna explain about the 'if' and the 'while' commands you will find out how important boolean expressions can be, and then it's good to know that you can store the output in a variable.

Now this is getting quite abstract, so let's throw in an example:

```
s1 = "Donald Trump"
s2 = "Donald Trump"
s3 = "Barack Obama"

b1 = s1 == s2
b2 = s1 == s3

print(b1) -- Outputs "true"
print(b2) -- Outputs "false"
```

Now the first three lines don't need any explanation, I think.

Now in Lua “==” means that Lua needs to check if the value in both variables are the same. If so the outcome is “true” if not the outcome is false. Now this makes it very easy to see why b1 contains “true” since both s1 and s2 contained “Donald Trump”. As s3 contains “Barack Obama” which is of course not “Donald Trump” b2 became false.

```
s1 = "Donald Trump"
s2 = "Donald Trump"
s3 = "Barack Obama"

b1 = s1 ~= s2
b2 = s1 ~= s3

print(b1) -- Outputs "false"
print(b2) -- Outputs "true"
```

Now I've altered the code a bit. You can see I replaced both “==” operators with “~=”. In Lua “~=” means “is not”. And hence now b1 became false as I made it check if “Donald Trump” is not “Donald Trump”, but as that is both the same string “is not” is therefore not the case (you follow as double negative is always hard to understand), and thus the outcome “false”. And I wanted “Donald Trump” not to be “Barack Obama” in b2, well since that is not the case, the outcome is “true”.

Now boolean expressions too are in for combinations.

```
s1 = "Donald Trump"
s2 = "Donald Trump"
s3 = "Barack Obama"

b1 = s1 == s2 and s1 == s3
b2 = s1 == s2 or  s1 == s3

print(b1) -- Outputs "false"
print(b2) -- Outputs "true"
```

“and” and “or” are keywords reserved for usage in boolean checks. Now can you see why the output is the way it is?

Basically the code explains itself.

For b1 in wanted “Donald Trump” to be “Donald Trump” AND “Donald Trump” to be “Barack Obama”. Due to the “and” keyword both statements must be true, but as the latter is false the entire expression is false.

For b2 either one of the two statements or both had to be true, well since the first statement is true so the entire statement is true and that the latter is not, is no longer relevant.

Then there is one final keyword to keep in mind and that is “not” and it does exactly what the name implies.

```
s1 = "Donald Trump"
s2 = "Donald Trump"
s3 = "Barack Obama"

b1 = not ( s1 == s2 )
b2 = not ( s1 == s3 )

print(b1) -- Outputs "false"
print(b2) -- Outputs "true"
```

Since I put in “not” this time, it basically may not be “true”. Since s1 *has* the same value as s2 that is true, but as it may not be true, “false” is the result, and since s1 did not have the same value we did see what we want and get “true”.

Lua can be a very strange beast in parse checkings, and if you are using an IDE that supports parse checking then “b1 = not s1==s2” will be taken as a valid instruction, but the outcome may surprise you as what Lua will do is “b1 = (not s1)==s2”. I therefore advice to never use 'not' without parents, but to always include them like shown above.

Now it is also cool to note, and that is where boolean expressions can be powerful is that they can also be used on mathematic operations and concatenations, and all this in real time. Let's write a

small program to demonstrate this:

```
b1 = 5+7 == 12
b2 = 5+8 == 15
d1 = "Donald"
d2 = "Trump"
d3 = "DonaldTrump"
b3 = d3 == d1..d2
print(b1)
print(b2)
print(b3)
```

Well since 5+7 is indeed 12, b1 will be true. Since 5+8 is 13 and not 15, b2 will be “false”, and the concatenation of “Donald” and “Trump” is indeed “DonaldTrump”, so that outcome also fits and thus “true”.

Lastly there are a few more operators for you to keep in mind:

- Greater than >
- Lower than <
- Greater than or equal >=
- Lower than or equal <=

They can be used in stead of == or ~= but only when you are using number values.

```
b1 = 5+7 == 12
b2 = 5+8 <= 15
d1 = "Donald"
d2 = "Trump"
d3 = "DonaldTrump"
b3 = d3 == d1..d2
print(b1)
print(b2)
print(b3)
```

When I change the code to this, you will see 3x “true” as output, and that fits since 5+8 is 13 which happens to be lower than 15.

4. The “if” keyword

4.1 if … then … end

Now the “if” keyword is important to understand. This is one of the commands in which you'll be able to make your program actually respond to what the user is doing.

Basically you say “*if* something is the case *then* perform all instructions until the *end*”.

The basic syntax is like this:

```
if <boolean-expression> then
<all instructions to perform when the boolean expression happens
to be true>
end
```

Now the “end” keyword pops up for the first time in this book, and it's the most used keyword in Lua. “if” creates what we call a scoop. A scoop is just a collection of commands. Every scoop ends with the “end” keyword (few exceptions, but we'll get to that later). For C users, it's basically what you use accolades for in C, but in Lua “if” and other commands like that will always open a scoop.

Now let's demonstrate this:

```
name = "Neil Armstrong"
print(name)
if name=="Neil Armstrong" then
    print("Yup, that's the name of the first man on the moon.")
end
```

Let's break this down a bit. The first two lines should be known by now. With “if” we start the “if”-statement and as you can see the name is checked in the same way as we create a boolean variable. The “then” keywords lets Lua know the end of the expression is reached and starts the scoop with all the commands that should be executed then, which is now only one, but there is no limit how far you can go with then and lastly the “end” commands closes the “if” scoop.

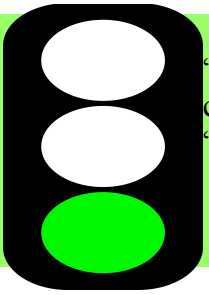
If you run this program you can see that the program confirms that Neil Armstrong is the first man on the moon. If you put anything else in the “name” variable nothing will happen aside from showing the name, after all that confirmation command should only be executed if the expression is true.

NOTE: You can see that I put in some leading spaces for all commands within the “if” scoop. We call this “indenting”. It is good practice to follow my example on this, as this keeps your code more readable. It's then easier to see what belongs to what scoop. If you want to plan to learn Python in the future all scoops end when you stop indenting in Python. Although Lua doesn't care about this, you can spare yourself some misery when you also learn Python, and also for good overview it's better.

4.2 if...then...else... end

```
name = "Neil Armstrong"
print(name)
if name=="Neil Armstrong" then
    print("Yup, that's the name of the first man on the moon.")
else
    print("No, the name is not the name of the first man on the moon.")
end
```

Perhaps you can see a little where I'm going here. If the variable name contains “Neil Armstrong” then the first scoop is performed. Else will end the first scoop and open a new one and there all the commands are shown performed when the expression was false.



Now you should be able to make a program that checks if a variable called “name” (string) actually contains your name, and if a variable called “age”(number) contains your correct age and make the computer say “Correct” if it's correct and “Incorrect” when it's incorrect.

All the information you need to do so is in the last two subsections.

4.3 if … then … elseif … then … else … end

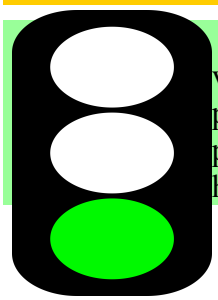
```
name = "Neil Armstrong"
print(name)
if name=="Neil Armstrong" then
    print("That's the name of the first man on the moon.")
elseif name=="George Washington" then
    print("That's the first president of the United States")
elseif name=="Margaret Thatcher" then
    print("That's the first female prime minister of the United Kingdom")
else
    print("I don't know who that is.")
end
```

Now this looks a bit more complicated, but it's not as hard as it looks.

“elseif” simply means “if any of the previous statements was wrong check this one and execute if true”. You can throw in as many elseif statements as you want, and the last else is also optional. Now you can get a real show on the road, eh?

This note only matters when you have experience with C, Go, C#, Pascal or any other programming language.

Lua has no case-support, meaning that if you ever need casing you are basically condemned to use if-elseif-else structures like above. It's a very serious downside about Lua, and I do not know if future versions will ever support it. Sorry about that!



Now make a program in which you put in the name of a person you know in a variable called “name” and use ifs and elseifs to check the name and state that person's profession and in the end an else statement stating that you don't know that persons profession. Now change the name variable with several tests to see what happens.

5. Loops

Loops allow your program to do things over and over, again and again.

Now basically there are three ways in which Lua can loop. “For”, “While” and “Repeat/Until”.

5.1 For

For can be used in multiple ways, but for now, we will only use it for countdowns and countups. Let's demonstrate the for routine, eh?

```
for i=1,10 do
    print(i)
    if i%2==0 then
       print("Even")
    else
       print("Odd")
    end
end
```

The “for” command creates a variable just for itself and that is in this case “i”. This variable will only exist within the “for”-scoop, not outside of it. It will be defined as 1 and then all commands in the for-scoop will be executed and after that “i” will be increased with 1 and then the scoop will go on again. Once “i” has reached 10 at the end of the for-scoop, the loop will end and the first command after the “end” will then execute.

Now it's nice to note that if has its own scoops. Lua will know which end belong to which scoop, as basically end closes all scoops in reverse order as they are created, so the parser (the program inside Lua that checks and interprets your code) will not get confused.

Now basically any value can do, as long as you first put in the start value and the end value at the end. The instruction as shown above however, can only count up, but like I hinted, counting down is possible if you enter a third number, which we call the “step value”

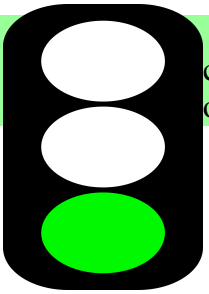
```
for i=10,1,-1 do
    print(i)
    if i%2==0 then
       print("Even")
    else
       print("Odd")
    end
end
```

Due to the step value being negative Lua knows it should now count backwards.

Now this is fun, but we can basically go into any direction here with any kind of values

```
for i=0,50,5 do
    print(i)
    if i%2==0 then
       print("Even")
    else
       print("Odd")
    end
end
```

This example works the same as the programs before, however now “i” will be incremented with 5 every cycle.



Now make a program and use the for-command in order count from 1 till five and combine it with the if-elseif commands to make it print the number in word in stead of digits.

5.2 While

Now proper understanding of the “while” command is very important. The working of the “while” command is similar to “if” command however while keeps repeating the scoop until the boolean expression it comes with returns false.

Now while should be handled with care and perhaps this examples show you why:

```
name = "Jeroen"
while name=="Jeroen" do
    print("Looping")
end
```

Yup, the loop goes on forever, as the expression will always be true.

But we can do something else here:

```
a = 0
b = 1
while b<10 do
     a = a + 1
     if a>b then
        a = 1
        b = b + 1
     end
     print(a..",",..b)
end
```

Well as the variable being checked by while now we don't have an infinite loop. The variable b can be changed in every cycle, but that doesn't necessarily happen (as would be the case in a for-

loop). This is one of the most common loops in most of your programming work. Since we've not yet gone into the deep of functions yet, I cannot demonstrate much more, but this does show the basic idea of what to expect on while loops.

5.3 Repeat Until

Repeat until work pretty similar to while, although it is different on a small yet essential point. Where while checks as soon as the loop starts, repeat until checks when the loop ends. Due to this it is possible with while the loop gets skip altogether, as there could be a boolean expression that is false from the start. Since Repeat Until checks at the end the loop will always be executed at least once.

```
a = 0
b = 1
repeat
      a = a + 1
      if a>b then
         a = 1
         b = b + 1
      end
      print(a..","..b)
until b>=10
```

This program is a bit of the repeat/until variant of the program I wrote earlier with while. Now if you'd make b to value 10 in the while-version of this program, you'd see nothing happens, if you do that here, you'll see the loop is performed once. In this original setup you should see any difference. Whether you should go for while or repeat/until is always a matter of judging the situation you're in as a programmer.

Please note that repeat until does not require an “end” to end the scoop the loop is in. Since until always comes at the end of the scoop, it's simply not needed.

Now here's a trapdoor, and especially when you are used to coding in C (or any of its variants) and therefore used to the do{}while() loops you can easily fall for this.

Where while keeps looping as long as the boolean expression is true, the repeat/until loops will loop like the word “until” implies, loop until the expression is true. In other words the expression must be false in order to keep looping. (It may seem a bit strange for a language developed in C to take over the standard of Pascal for this).

5.4 Break

This is a command I do not want to put too much attention on, as you can better avoid it, but when you need it, it's there. The break command will immediately terminate a loop and continue the program with the next command after it.

```
while true do -- start infinite loop
   a = math.random(1,10) -- generate random number from 1 till 10
    print(a)
    if a == 5 then break end
end
print("I have now left the loop")
```

Normally I'd use repeat until for this, but this terrible code demonstrates how break works.

The math.random() function just generates (pseudo-)random numbers and as soon as number 5 was generated the break command is executed causing the loop to end and go to the “print” command after the “end” (or until in case of a repeat/until loop).

Please note that break only works inside a loop. It has no effect on other kinds of scoops and without a loop it may even throw an error.

6. Tables

Tables is where working with Lua becomes fun. Tables are a very powerful instrument. There is a lot a programming language should support which Lua does not, however tanks to the table system you can cheat on that. Even OOP (object oriented programming) becomes possible, although Lua officially doesn't support that.

6.1 Using tables as arrays

Arrays in programming languages are variables containing a series of values. In Lua you can use tables to make this happen. Best is maybe to show this in example code.

```
a = {} -- create table
a[1] = "One"
a[2] = "Two"
a[3] = "Three"
a[4] = "Four"
a[5] = "Five"
for i=1,5 do
    print(a[i])
end
```

When you ever programmed in another language you'll be used to 0 being starting index of an array. Lua however uses 1 as starting index on an array. Never forget this or you can mess up dearly!

Although 0 can be used as an index, in array usage it will be ignored.

Well the variable “a” is thus the table and the numbers between the brackets [and] are called the indexes. Now now you can see maybe why understanding arrays is so important and how much power they can give. The for-loop at the end of the code above shows that you can use a variable to determine the index number.

The fun doesn't end there. The example above can be done with less code. Like this:

```
a = {"One","Two","Three","Four","Five","Six","Seven","Eight","Nine","Ten"}
for i=1,#a do
    print(a[i])
end
```

You can, as you can see, easily define the entire array in one line. The first entry will be index #1, the 2nd will be index #2 and so on. And #a means “length” of “a”. This sign can be used for strings and array-based tables to get the length. If it's a string #a will contain the number of characters, and in a table the number of indexes.

Now in Lua any undefined variable will be considered “nil”. Table elements are no different from this point. So a[12] will be nil, as I didn't define that index. #a will basically count from index 1 until the first 'nil'. Since a[11] is the first nil in the example above #a will therefore be 10. When you put in the line a[6]=nil, then #a will be 5 and despite 7, 8, 9 and 10 having values. They will be ignored. I'll get later on how you can properly remove values from an array.

The for loop I used in the example above is considered dirty in Lua. Many programming languages contain a workabout that we often call “foreach”. Now “foreach” is not a keyword in Lua, but the name is based on the fact that much programming languages either have “foreach” or “for” and an expression with “each” in it. Forget about that for now, let's do this the Lua way.

```
a = {"One","Two","Three","Four","Five","Six","Seven","Eight","Nine","Ten"}
for i,n in ipairs(a) do
    print(n)
end
```

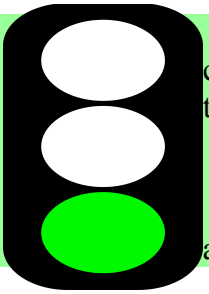
The keyword “ipairs” is one you'll find a lot in Lua scripts. The “for” command will loop as always, but a bit differently than in the examples we had up until now. It will now take all indexes in order and loop the scoop tied to this until the first “nil” arrives. In the example above “i” will contain the index number during each loop cycle and “n” the value stored in that specific element.

Is the fun over of using arrays in Lua. No! Lua can do something more. Tables can be altered after their definitions. In the first example you saw you can do that one by one so adding “a[11]='Eleven'” is just a simple way to go, but in an OOP set up game, you can go even further than this, as then you can just add objects as they go. Imagine a space shooter in which enemies just fly in at random, becoming more and more? Arrays can do this.

```
a = {}
-- method #1 for addition
table.insert(a,"One")
table.insert(a,"Two")
table.insert(a,"Three")

-- method #2 for addition
a[#a+1]="Four"
a[#a+1]="Five"
a[#a+1]="Six"

for i,n in ipairs(a) do
    print(n)
end
```

In the example above I've shown you two methods to add elements to an array. There is some debate among Lua programmers which method is best and even the creators of Lua are not too sure, but I personally prefer method #2.

Anyway, you can go as far as you want to go with this. There is no official limit (although there are limitations to your RAM, but don't worry about that when you are only an amateur. In my Sixty-Three Fires of Lung game, I've used with array with over thousands if not ten-thousands if elements).



Make a program and create a table and add three animal names in it in the creation instruction. Then add any number of other animals you like with either table.insert() or the a[#a+1] method I showed above.

Now make a for-loop that shows all these animals.

Lastly use a print instruction that shows how many animals there are in your array.

Lastly I will explain table.remove(). That can be used to remove one element from your array and all the other elements will be adapted to the new situation.

Try adding the line “table.remove(a,3)” in your program (assuming your table variable is “a”) just before the for-loop and run the program again and see what happens. ;)

6.2 Using tables as dictionaries and structures

Now as all programming languages should support arrays, Lua cheated a little on the array functionality. The true power of tables is now gonna be revealed. Or at least part of that. ;)

Where arrays could only use numbers for index, and which had to be all in order, tables in full work in key values and they can basically be any type. Most common for key values are strings though, and I will limit myself to that.

```
a = {}
a["dog"]="woof"
a["cat"]="meow"
a["chicken"]="bacock"
a["cow"]="moo"
for k,v in pairs(a) do
    print(k.." says: "..v)
end
```

Now you can see that in stead of numbers I used strings. Other than that you see the behavior is pretty similar to arrays. Now in stead of ipairs() I used pairs() in the for-loop... without the “i”. As the usage is pretty similar tables are when used as a dictionary easy to use.

Important to note though is that there's no specific order in which pairs() will give the data to the for-command. You can be sure that all keys and its respective values will be given to the for loop (as long as the table variable is not modified during the loop), and that is all. There are a few advanced tricks when the order is really important, but that is the more advanced stuff.

Of course, up until now I only used strings for values, but can numbers be values too?

Of course!

```
a = {}
a["George Washington"] = 1
a["Andrew Jackson"] = 7
a["Abraham Lincoln"] = 16
a["Grover Cleveland"] = 22
a["Herbert Hoover"] = 31
a["John F. Kennedy"] = 35
a["Ronald Reagon"] = 40
a["George H.W. Bush"] = 41
a["Bill Clinton"] = 42
a["George W. Bush"] = 43
a["Barack Obama"] = 44
a["Donald Trump"] = 45
for president,number in pairs(a) do
    print(president.." was president number "..number.." of the United States")
end
```

The downside is that all presidents can be shown in any order, but to come to the point, this

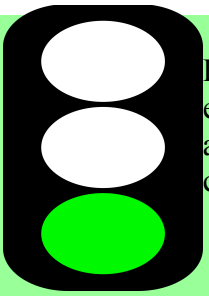
works.

And now we're gonna take this to the next level. Can you make a table of tables?

Yes!

```
presidents = {}

president["George Washington"] = {}
president["George Washington"]["Gender"] = "Male"
president["George Washington"]["Country"] = "United States"

president["Park"] = {}
president["Park"]["Gender"] = "Female"
president["Park"]["Country"] = "South Korea"

president["Vladimir Putin"] = {}
president["Vladimir Putin"]["Gender"] = "Male"
president["Vladimir Putin"]["Country"] = "Russian Federation"
```

And a table in a table in a table in a table... yeah... you get the idea.



Make a table and put in it any number of fictional people you like. Like “Harry Potter” or “Frodo Baggins” or whatever. Those names are the keys. Now every element should be a table on its own containing the characters name, their gender and the name of their lover (just add the string “none” if they have none) and of course use key names you can easily recognize.

Now comes the hard part of this assignment, but with a bit of thinking you can do it. Use the for-command (you may use it multiple times if you think you need that) that lists out all these persons and all the data inside the tables attached to them.

6.3 Reference or value

It's very important that you'll understand this part of Lua, and it unfortunately *is* a bit hard to understand.

In Lua strings, booleans and numbers are value based types. And as such a boolean check such as “JohnCleese=='comedian'” works and “five==5” can work too. Arrays and tables (which are in Lua technically the same thing) are pointers. They do not contain any values, but only a reference to the memory address where the data is stored.

Now unlike C where you have to allocate the memory yourself and to release the memory yourself, Lua has this part fully automated, and the code doing this is also pretty damn fast. So at least you don't have to worry about that part, but I will show you a few trapdoors if you are not aware of how the pointer based structure works.

```
a = {}
b = a
a["Hello"] = "Hi"
print(b["Hello"])
```

Did the output say “Hi”?

It did, didn't it?

But how is that possible when you didn't define “Hello” in b?

That is because the “b = a” instruction did not copy the data in a, it only copied the pointer, the reference to where in the memory 'a' is stored. Now since a and b have now the same memory address, everything you change in “a” will therefore automatically affect “b” and vice versa.

The same trapdoor also happens here:

```
a = {}
b = {}
a["Hello"]="Hi"
b["Hello"]="Hi"
print(a==b)
```

You could have expected “true” as the outcome since a and b have the same data, but what the boolean checker checks is the true data in the variables themselves which happen to be the pointers, and they are different.

```
a = {}
b = a
a["Hello"]="Hi"
b["Hello"]="Hi"
print(a==b)
```

This version of the code, will result in true.

This goes for the table and function type variables (and strictly speaking for the userdata type as well, but that is only important if you are going to make C-APIs for Lua and that's something I am not gonna cover in this book).

This is maybe one of the more complex parts of Lua to understand for beginning programmers, but once you do you are well on the road. Also there are many other languages in which the same kind of rules apply as I now explained for Lua. Python, BlitzMax, C# to name a few.

6.4 The dot syntactic sugar

Syntactic sugar means a notation that is not official, but the parser will still accept it as legit.

Lua is full of these sugar things. Since Lua has no support for “structs” and “classes” which are deemed essential in programming these days, Lua invented a few ways to cheat around.

Now I won't go into the dept what “structs” and “classes” are, as structs are only relevant to refer to when you have experience in C and the C programmers should know what they are, and classes will be discussed when we get to the Object Oriented Programming part of Lua (or rather how Lua cheated its way around in this with syntactic sugar as well).

Let's suffice to say that in C you can often see this kind of code

```
a.AnimalName = "Cow";
a.Sound = "Moo";
a.Food = "Grass";
b.AnimalName = "Cat";
b.Sound = "Meow";
b.Food = "Birds and mice";
```

Basically a and b are struct variables then. In Lua we can simply use tables for this.

```
a = {}
a.AnimalName = "Cow"
a.Sound = "Moo"
a.Food = "Grass"
b = {}
b.AnimalName = "Cat"
b.Sound = "Meow"
b.Food = "Birds and mice"
```

Now what good does this do?

Well, add this line to your code and be surprised:

```
 print("a "..b["AnimalName"].." says '"..b["Sound"].."' and eats
"..b["Food"]..".")
```

What did we see? What did we learn.

Yes in Lua “a.Sound” and “a[“Sound”]” are the same thing. This makes a shorter notation for tables and their keys and gives C-junkies the illusion they are actually using struct variables, so they are also happy.

What is very extremely important to note:

First of all, this only works when the key value is a string. So saying “a.1” in stead of “a[1]” will lead to a crash, and when the key value is a reserved word in Lua (also known as “keyword”) you also got a problem. “a[“for”]” is valid “a.for” is not. This because “for” with quotes is just a string, but as for without them is not, Lua won't understand what you are doing. Fortunately the number of keywords in Lua is rather small.

7. Functions

In order to make Lua communicate well with the underlying C program, or for quick programming in general, Lua heavily relies on functions. I told you in the “Hello World” lesson that 'print' is a function. And the 'pairs' and 'ipairs' I discussed in the arrays and tables chapter are functions as well.

7.1 General approach

You can think of them as a kind of 'mini-programs' in your main program, yet they can be called whenever needed and even return data.

Let's put up a simple example:

```
function Bart()
    print("Yo!")
    print("Hey! What's happenin', dude!")
end

Bart()
```

So the function keyword is used for defining functions. And the two 'print' calls are now part of the function named “Bart”. And yes, just typing Bart() calls it.

Now I see you wonder. The “print” function accepts data to put on screen. Can I do that too?

Of course, easy as pie!

```
function ShowSum(a,b)
     print(a+b)
end

ShowSum(4,5)
-- Outputs 9
```

Now I like to call these functions “void” functions (after the keyword 'void' that you use in C to define these kinds of functions), as they are not designed to return data. But in Lua you can also make functions return values, and those can be values of ANY kind:

```
function TimesFour(a)
     return a * 4
end


print ( TimesFour(2) ) -- outputs 8
```

The code is pretty self-explaining. “a” is now automatically created as a local variable. It can therefore only be called inside the TimesFour function. By calling TimesFour(2) this local variable contains the number 2, and the return command ends the execution of function TimesFour() and returns outcome of a*4 or in this case 2*4 and since that is 8, that is what the print function will display.

As you can see a function can accept no values, multiple values, but can it also return multiple values. Well, yes, in Lua it can, and it's shockingly easy to do so, also!

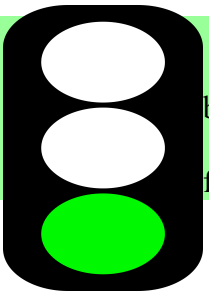
Trust me!

```
-- double, triple, quadruple
function dtq(a)
   return a*2,a*3,a*4
end

value=20
doubled,tripled,quadrupled = dtq(value)

print(value)
print(doubled)
print(tripled)
print(quadrupled)
```

So yeah, all you need the function to do is to place all the values behind the “return” keyword separated by commas and you can just get them like shown above with all variables noted all at once.

Now there's also a drill. In Lua you can expect more values than there will be given/returned. Lua will just go on, and the values that are not given will be “nil”.

```
function myfunc(a,b)
    print(a)
    print(b)
end

myfunc("Hello")
-- outputs
-- "Hello"
-- nil

function myreturn()
    return 1,2,3
end

a,b,c,d,e = myreturn()

print(a)
print(b)
print(c)
print(d)
print(e)
-- You'll get
-- 1
-- 2
-- 3
-- nil
-- nil
```



Make a program which has a function saying hello.
A function that returns the sum and the product of two values, and return them both in once, and catch them in two variables.
Now make the program say hello with your function and print the results of the function returning the sum.

7.2 Cyclic or recursive function calls

This is the magic of making functions call itself. Yes that is possible yet risky. Doing this will carelessly will cause in infinite loop and can be costly on your RAM. Now Lua has been protected against this and will eventually cause a “stack overflow” error, but you want to prevent his.

However when done right, this can be a very handy tool at your disposal.

Traditionally nearly all programming guides for all programming languages create a factorial function to demonstrate this process.

```
function factorial(n)
    if n==0 then return 1 end
    return factorial(n-1)*n
end


for i=1,9 do
    print( "!"..i.." = "..factorial(i) )
end
```

Well I only added the for-loop so you can see this program can successfully calculations the factorials of 1 till 9. Now the true nature of a factorial can be read in this wikipedia article, but is not really relevant for our Lua lessons, so I will not go too much in the deep of that.

The easiest way to calculate factorials is though to always deem factorial 0 as 1, and all next factorials as the factorial as the number before times the number you have. More complicated than it needs to be, but anyway, you can see how the factorial recalls itself in a cyclic manner in order to come to the full number.

I will not give any assignments on this one, but it's just handy you know that this routine exists.

Oh yeah, I warned you for infinite loops... let's give you one!

```
function wrong()
    print("This will go wrong")
    wrong()
end

wrong()
```

Now the playground will most likely terminate this program before the stack overflow goes off, but the cli tool and real-time Lua engines will eventually throw an error.

7.3 Treating functions as variables

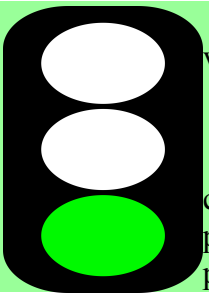
Now I told you that “function” was a variable type. So maybe you could draw the conclusion yourself already that functions are in Lua just variables, they only contain a pointer to code in stead of data. Now this will be very important when you get into OOP programming, but I will get into that part later.

Officially the function commands as I used them in the previous section are “syntactic sugar”. The official way to write “function hello()” would actually be “hello = function()” in Lua, although, in most situations not a single soul uses that notation, that does not mean it's not important to know as

it can do great things.

```
function GetMeAFunction()
   return function()
         print("Hello World")
   end
end

hello = GetMeAFunction()
hi = hello
hi()
```

I guess this might be one of the most complicated versions of “Hello World” you've ever seen, but it demonstrates how you can easily use functions themselves as variables. As soon as you don't give up the (and the), the function will be treated as any other variable, and thus I could easily turn the variable hi into a function with the “hi=hello” command.



Now create a function the way you were used to do, and copy it to another variable, and make a call to this variable.

And a little test for you to see if you have understood the way Lua works completely. You don't have to feel ashamed if you don't know the answer, but it is possible to make Lua forget about a function. Do you know how? If not, no problem, but if you know, you're on the way to become a pro.

7.4 Functions and tables, and a way to fake classes. The way to OOP.

Now a lot of programming languages work with classes, and C# in particular can't cope without them.

This is an example of a class in C#

```
class ohyeah{
    public int i=0;
    public void changeme(int newvalue){
          i = newvalue;
    }
}
```

In C# this would enable me to create a variable of the type “ohyeah”, let's name it “y” for now, and when I type y.changeme(4) then y.i will thus be 4... well this is the basic idea. Now void changeme acts here as a function tied in to this class and we call that a method, and due to that it will know that “i” belongs to the class.... Do you understand this? If not, then don't fret, as Lua has no support for this at all.

Then why did I mention this? As the example above is basically a textbook example for OOP programming, and although Lua does officially not support OOP, Lua can cheat its way around that to “pretend” an OOP environment, and to fake the class system, and that's what we're getting into today.

The cheat I'm gonna explain to you, lies in stuff I explained to you before. Tables. In section 6.4 I already presented you with the syntactic sugar using dots in stead of [], right. Well, we're gonna exploit that part a little bit more.

OOP, which is the acronym for “Object Oriented Programming”, came to be as people realized over time how important it is not to think in data in general, but but to objects, and to also tie the code to objects, as well. Now this can allow us easily for example in a space shooter game to make all the enemies move independently, and to carry all the data they need, and even all the functions they need.

This way of programming can be a bit hard for a beginner to understand as it's a rather abstract way of programming, and when you really dive into the depths of OOP, you will no longer be just doing a bunch of code line by line and hop over to the next when you're done, etc, etc. So proper thinking is required.

Don't let that all discourage you though. Once you got the hang of OOP, you will not easily want to put it away, and especially in game programming, OOP can work wonders.

Before we begin on this, I did tell you that functions were nothing more but variables, right. Let me work out that class I showed you in C# at the beginning of this chapter in a Lua workout.

```
function ohyeah()
    return {
          i=0,
          changeme = function (self,newvalue)
              self.i = newvalue
          end
    }
end
```

Yeah, that looks pretty cool, huh?

I won't blame you if you do not fully understand it. Like I said, as Lua doesn't understand classes I created this function to fake my “ohyeah” class. It returns a new table, and yes as you can see Lua allows me to immediately define keys in it. You should make sure that every field is separated by commas though, and that means that if you put in a function this way a comma will be required after the end if you have more keys to add after that function.

Now let's use add a few lines of code in order to take “ohyeah” into action.

```
a = ohyeah()
a:changeme(4)
print(a.i)
```

Now can you see what I did here?

Well “a = ohyeah()” is very likely the easy part. Now a contains a table with the number in “i” and the function in “changeme”. And “print(a.i)” should speak for itself.

But how about “a:changeme(4)”?

Yup, this is syntactic sugar. Lua will translate “a:changeme(4)” into “a[“changeme”](a,4)” (or as a.changeme(a,4) if you prefer), and I guess the call will then make more sense, won't it?

With the “:” operator after a table's name, Lua will automatically add the table itself as the function's first parameter, and add the parameters I manually added. This is as close as Lua can get to method calls. Keeping this in mind now “self.i” inside changeme will make sense.

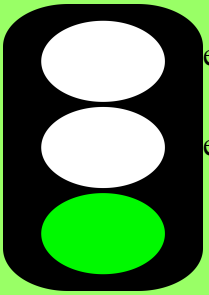
Now keep the code as you have it and add these lines, and try to understand what they do prior to checking the result on the playground.

```
function a:doubleme()
     self.i = self.i * 2
end

a:doubleme()
print(a.i)
```

Syntactic sugar?

Definitely!

If you define a function like this Lua will translate it as “a.doubleme = function(self)”, and this makes things very easy now, isn't it?



I will now do things a differently than you are used to see, but I want you to examine the following code, and try to predict the output. Write it down if you will.

Then you can copy the code in the playground to check if the output is what you expected. If so, you are well on the way.

```
template = {
     x = 0,
     y = 0,
     s = 5
}

function template:move(x,y)
     self.x = self.x + x
     self.y = self.y + y
end

actors = {}
for i=1,10 do
   newactor = {}
   actors[#actors+1] = newactor
   for k,v in pairs(template) do newactor[k] = v end
   newactor.x=i
   newactor.y=100-i
end


for cycle=1,10 do
  for i,actor in ipairs(actors) do
    print("Actor #"..i)
    actor:move(i,-i)
    print("Position ("..actor.x..","..actor.y..")", speed: "..actor.s)
  end
end
```

Now let's make a little variation to this. If you have understood the code above, you will know this code will run 10 cycles with 10 actors. If you have really understood everything, then try to find how you can make a new actor be added to the actors table prior to the for-ipairs loop.
This may be the hardest task so far, so think it out well.

7.5 Local variables

Before we move on to chapter 8 in which we will prepare for some actual game coding, first an easy section after the complicated OOP stuff. Local variables. Up until now we've only worked with global variables, except for the variables created in for-loops and function parameters.

Locals only live in the function or scoop in which they are created. Now all variables (even functions) can be local. In Lua all you have to do is to put the keyword “local” prior to their definition.

```
function lv()
   local x = 40
   print(x)
end

lv()
print(x)
-- output
-- 40
-- nil
```

Since the “local” keyword told Lua to keep “x” local to the function lv, the x=40 definition only affects the print(x) inside that function, and that's why the second print(x) outputs nil. Well, and if the local keyword is not present when a variable is created Lua will assume it's a global.

Now here are a few things to keep into account.

```
x = 70

function lv()
   local x = 40
   print(x)
end

lv()
print(x)
-- output
-- 40
-- 70
```

At the start of the program I now created x as a global, and yet inside lv x behaved as a local as told, and the print(x) outside the function returned the 70 as I originally asked for.

Now what Lua does when you use the local keyword is always create a new variable, and if a global or a “higher-level” local (more on that later) already exists, it will simply put emphasis on the local created within that scoop and ignore everything else until that scoop has ended.

Now locals can not only be used inside functions, but inside any kind of scoop:

```
function test()
  local a = 10
  if a == 10 then
     local b = 20
     print ( b )
  end
  print( b )
end

test()
```

See what I mean? Since b was defined as a local inside the “if”-scoop, the print(b) outside the “if”-scoop became “nil”...

```
function test()
  local a = 10
  local b = 80
  if a == 10 then
     local b = 20
     print ( b )
  end
  print( b )
end

test()
```

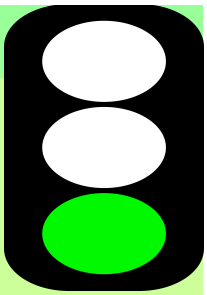
And now this, and now the output will be 80 and 20. Value 80 is in b tied to the function as a whole, and the b containing 20 to the if-scoop, hence the output the way it is.

But what about locals that are made a local in the function as a whole, but not inside the sub-scoops? Let's try it!

```
function test()
  local a = 10
  local b = 80
  if a == 10 then
     local b = 20
     print ( a )
     print ( b )
  end
  print( b )
end

test()
```

Well, as you can see, since “a” was not recreated inside the “if” scoop and the “if” scoop being part of the function scoop, a was just taken the way it was in the higher scoop.

Does this code work, and if not, why not? (try to explain before using a cli tool or the playground) :P



```
function test1()

    print("test1")

    function test2()
       print("test2")
    end

    local function test3()
       print("test3")
    end

end

test1()
test2()
test3()
```

8. LÖVE a simple engine to create games with Lua

Now there are countless of engines working with Lua, and if you can code C you can even set up a Lua engine yourself. For the course of this book, I'll go for the LÖVE engine, because it's free.

Now LÖVE uses games stored in .love files, now a .love file is only a zip file containing Lua Scripts, png pictures, mp3/ogg audo files, and so on... So any program that can create a zip file can be used to create a .love file.

Of course, setting up LÖVE can differ a bit depending on your platform. For the Lua code you write the platform should not matter.

8.1 Getting ahold of LÖVE and setting it up

Since LÖVE has been set up to included with your game we will for now put LÖVE and your game file together in one folder. This is a bit of a “dirty” approach, but one that works during development and debugging. Distributing a game is something I will get later to, although the wiki on the LÖVE website will have to suffice.

The LÖVE version that was the standard when this book was written was 11.2, but as I am only going into the basics of LÖVE, most likely any version should do, but LÖVE changes rather quickly (too quickly, actually), so I deem it possible, some sources beyond this point may not work due to version differences.

Now the LÖVE code itself will basically be the same regardless of your platform, but the outer setup differs per platform. DON'T EVER be foolish enough to download the mobile packages in this stage. If you really want a mobile version of your game, design it on a desktop or laptop and export the result to mobile later!!! I won't cover mobile export in this book, but the LÖVE website should cover that part, so no problem.

8.1.1 Setting LÖVE up in Windows

I'll go for the zip-file in this book. On the website you'll see there's a 32-bit version and a 64-bit version of LÖVE. If you are not sure your computer supports 64-bit binary code, then download the 32-bit version, which will always work, as the 64-bit version only works on 64-bit machines, but 64-bit machines can handle 32-bit code just fine.

Now let's unpack the contents to the folder we're gonna work with. Any folder will do as long as you reserve it for your LÖVE work in as far as this book in concerned. Now I advice you to create one subfolder and call it “testprojects” or something in the folder where you placed LÖVE.

Now somewhere in your Windows default apps should be a program named “Command Prompt” or something like that. Run it and you'll see a black window with “C:\Users\MyName>” in which “MyName” has been replaced with your username in Windows.

Type “cd <the name of the folder where you put LÖVE>” (and hit enter)

Now LÖVE officially works with zip files, but you can also work with regular folders (for debugging reasons). But let's first test LÖVE. Type “love” and hit enter. A window should pop-up giving a kind of flashy show stating no game has been loaded (duh!), well if that happens you know stuff works, and then you can just close the window.

For now I'll do this the quick way. Type this on the prompt (and hit ctrl-Z when I say ^Z)

```
md testprojects\helloworld
copy con testprojects\helloworld\main.lua
function love.draw()
    love.graphics.print("Hello World", 400, 300)
end
^Z
love testprojects\helloworld
```

If you now get a black window with the words “Hello World” on it, then perfect, then stuff works the way it should, and you can close this window.

Now if you take a look in your file system you may see in the folder testprojects the folder “helloworld” is there. When you tell LÖVE to run this folder it will always look up the “main.lua” file and run it. I will not yet get into the deep of what the Lua code above means... that will come once we get into actual game coding in the sections where platforms no longer matter.

Keep the window where your prompt lives open, as you're gonna need that pretty bad in the next sections.

8.1.2 Setting LÖVE up on Mac

On Mac things can be a little bit disturbed by the application bundle structure Mac uses, which makes it a blessing for distributing your game, but less practical in these learning sections, but no matter, I'll help you to create your way around it.

First of all download the Mac zip and double-click it and Mac will create the LÖVE application. Don't move this to your regular applications folder as you are used to. Create a folder where you want to do your exercise in and move the application there. Also create a folder named “testprojects” in this same folder.

Now in the “Utilities” folder of your applications folder is a program named “Terminal”. Open it (please note that the translations of MacOS are horrible, which can make it hard to find this tool if you are not using the English version of MacOS). And you'll see a window that is in the default settings white with black letters, and unix-command-prompt on it. It can look different depending on your settings, but it should likely end with a “$”.

Now you'll have to type “cd <path where you installed LÖVE>”. If you do not know how to type that in unix, MacOS (and Linux too, btw) has a dirty trick for this. Just type “cd” and press spacebar and drag the folder where LÖVE has been installed in to the terminal window (the FOLDER!! not the app itself) and you will see the proper name appears behind “cd”. Hit Enter.

Now enter the next lines:

```
cat > love
#!/bin/sh
love.app/Contents/MacOS/love "$1"
```

Now hit ctrl-D

Lastly type:

```
chmod +x love
```

You don't have to understand what you did exactly. All you need to know is that you created a quick way to run your love projects from the terminal now.

Type "./love" and hit enter, and see if you get the flashy "No Game" screen. If so, stuff works okay, and you can close the window or hit Command-Q.

Let's now test our "Hello World" program. Type this and press ctrl-D when I say ^D

```
mkdir testprojects/helloworld
cat > testprojects/helloworld/main.lua
function love.draw()
    love.graphics.print("Hello World", 400, 300)
end
^D
./love testprojects/helloworld
```

If everything works the way it should you'll now see a black screen with "Hello World" in white letters.

Leave your terminal window open as you'll need it in the next sections.

8.1.3 Setting LÖVE up in Linux

Now I must admit that Linux and I are NOT the best of friends, and I've never really done this before in Linux. Since Linux is also dependency based, and the required dependencies can also differ per distro (unfortunately) this do not get easier here. Cut short Linux is a downright disaster.

This means you should already know the true ins and outs of Linux and how your distro differs from other distros if you wanna get into this.

The only package you'll find on the website is for Ubuntu. If you have a different distro you'll have to work your way around this. I have not yet gone for the AppImage version, as I am for debugging very terminal based.

Cut short when it comes to LÖVE itself you are on your own, sorry.

What you can do is create a folder and then the terminal. Type "love" and hit enter and if you see the flashy "No Game" screen, at least you know things are working accordingly.

Now let's type this:

And when I say ^D press ctrl-D

```
mkdir helloworld
cat > helloworld/main.lua
function love.draw()
    love.graphics.print("Hello World", 400, 300)
end
^D
love helloworld
```

If you see “Hello World” on a black window, then everything works as it should.

Leave the terminal window open. You'll need it later.

8.1.4 Some final notes before we really get on the move with LÖVE

When you've paid attention in the section that was about your platform, you should know now that the “helloworld” folder was the project folder and that LÖVE will look up for the “main.lua” file and execute it. Since Hello World is not requiring anything that was for now a very simple thing to do, however when you actually get into using graphics and audio you have to put them into the same folder. You are allowed to create subfolders in your project folder as long as you do not go outside this the main folder of your project. (I hope you get the grammar behind this one).

So a folder like this is allowed:

myproject/main.lua

myproject/myimage.png

myproject/mymusic.mp3

But this is also allowed

myproject/main.lua

myproject/images/myimage.png

myproject/audio/mymusic.mp3

And THIS is NOT allowed

myproject/main.lua

images/myimage.png

/audio/mymusic.mp3

This all comes with your own insights on the matter.

Now the setup we have now is not really suitable for distribution, but for development and debugging it's (for now) perfect.

Now what you name your projects in the upcoming sections is up to you... basically you will in

the upcoming sections run them as:

Windows: love myprojects/<projectname>

Mac: ./love myprojects/<projectname>

Linux: love <projectname>

8.2 An introduction to callbacks

Now callbacks is a bit of an abstract way of programming, and therefore it requires some thinking in stranger ways. If you could master OOP, this will be peanuts, but even when you don't, the basic principal is not hard to understand. If you can do this properly you have already made a start to widening your future perspectives, as JavaScript for example heavily relies on callbacks, and JavaScript is what browsers use for scripting.

What is a callback?

In a regular program the program just starts at the top of your code and runs all instructions in order, right?

In a procedurial program the program can jump a bit around due to function calls, but overall things still look pretty regular.

In callback this is all different. And therefore it's important you understand this if you wanna master LÖVE or even to BEGIN to do something with it.

Basically every time something happens inside a computer an “event” is created. This event contains data about what happened and under what circumstances. In a regular program these events would be read out and decide for itself what to do with it. In call back the environment responds to these events and will automatically call a function with a name it expects and act to it accordingly. And that process is what we call a “callback”.

For example if you have created a Lua engine that should should pop up “Hello” every time the letter H is pressed then your could could look like this in that engine:

```
function UserPressedH()
        PopUpDialog("Hello World")
end
```

Of course this does not work in LÖVE, this code was only meant to make you understand the callback principal. You engine sees H is pressed and calls to UserPressedH and this happens.

And basically every event can have its own callback function. LÖVE has callback functions for drawing stuff on the screen and updating data. These two are repeated continuously as your game runs. Then there are callback functions for mouseclicks and keyboard presses and releases of keys, and many more kinds of events. There is also a callback that starts when you load your game, which is only called once.

In the next sections I will be focussing on these callbacks most of all, as they are most vital to

understand when you want to create simple games.

8.2.1 love.load()

The “love.load” function is automatically called when your game is being loaded. It has been specifically designed to make sure all assets you require in your game are properly loaded.

Now there has been some debate if this callback is actually NEEDED or not. I say, USE IT and DON'T EVEN THINK about doing this otherwise. The debate arises since Lua can strictly speaking load stuff during loading when you put the load instruction outside any function. However if you have complex programs requiring special functions of your own design to do all this the order in which you put your code can give you very unpleasant surprises, and by putting it all in love.load() you can prevent all that crap, and it's also considered “cleaner code”, so needed or not, best is to use it.

Well how to do this.

First of all, you may create a project folder and download this image into it:
https://raw.githubusercontent.com/TrickyTeach/LuaStartGuide/master/img/Ball.png

And then let's put in the next code in the main.lua file:

```
function love.load()
   ball = love.graphics.newImage("Ball.png")
   x = 50
   y = 50
   mx = 1
   my = 1
end
```

Be sure that you keep filenames in the same setting of lower and upper case as you do in your file system. Very important. For starters Linux is case sensitive, so that can make you suffer already if you don't do this right, but when you're about to learn to make .love files, the same issue can get to you. Either way, be punctual on this here, even when you are a Windows user.

Well, you can try to run your program now, but you won't see anything happen at the present point. After all this is just a loading instruction. When we get further on we can see the full effects of this code.

But if you run now LÖVE will execute love.load() causing the image “Ball.png” to be loaded (as you may have guessed that love.graphics.newImage() is for loading images in LÖVE), and I defined for number variables for later usage.

Let's move on, shall we?

8.2.2 love.draw() and love.update()

Like I told you, LÖVE programs basically loop forever calling love.draw() and love.update().

These two callbacks are therefore a bit of the cores of your LÖVE game.

The love.update() callback is basically for any kinds of changes that may happen every time. If you are for example making a pac-man clone, you could put the code here that makes the ghosts move around.

The love.draw() callback is where the drawing happens. Always make sure your drawing commands only happen here. You can put them elsewhere in your program, but that will cause no visual effect.

So let's now get our program we had in the previous section and expand it with a love.draw() and love.update() function.

```
function love.load()
   ball = love.graphics.newImage("Ball.png")
   x = 50
   y = 50
   mx = 1
   my = 1
end



function love.update()
   x = x + mx
   y = y + my
   if x<=0 then mx=1
   elseif x>=(800-32) then mx=-1 end
   if y<=0 then my=1
   elseif y>=(600-32) then my=-1 end
end

function love.draw()
   love.graphics.draw(ball,x,y)
end
```

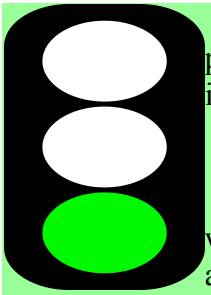
So let's break this down, shall we?

When you run the program, then LÖVE will first execute love.load() and load the Ball.png image and set the variables for the rest of the run. After that it enters an eternal loop in which love.update() and love.draw() are being called. As you can see, I put in some code to make the ball move around in love.update(), and that is simply done by changing variables, as you can well see. The love.draw() function draws the ball on the screen based on the x and y coordinate.

If you are completely new to how computers do things you need to watch out. If your mathematics teacher told you that *x* decides how far to the right a point is and *y* how high, then they are right as far as mathematics is concerned outside computers. In computers the value of the *y* coordinate decides how LOW the point is. So (0,0) is in mathematics bottom-left, but in computers top-left. Now LÖVE uses a 800x600 screen by default (you can change that, but that's not important right now) that means that in my program (799,599) is bottom-right (800x600 will just fall out of the screen, so you must in max coordinates always have the width and height minus 1).

Now before you run it, can you guess what this program does? Try to get the basic idea, and when you think you got it, run it and see if what is supposed to happen, actually happens.

(Answer is below, you can mark the text to see it or copy and paste it to an editor without color support like notepad).



Now try to cook something up yourself. You may either use my ball picture or a picture of your own and try to create a program showing this. It's up to you to make it move. Anything goes... Just try it.

If you have any knowledge about radials and sine and cosine or that stuff you may wanna try the math.sin(), math.cos(), math.tan() functions on this... Just play around. This is a pretty free task... What counts is that you understand what you are doing.

8.2.3 love.keypressed() and love.keyreleased()

Well, playing around with love.update() and love.draw() is fun, but what good is a game without interaction?

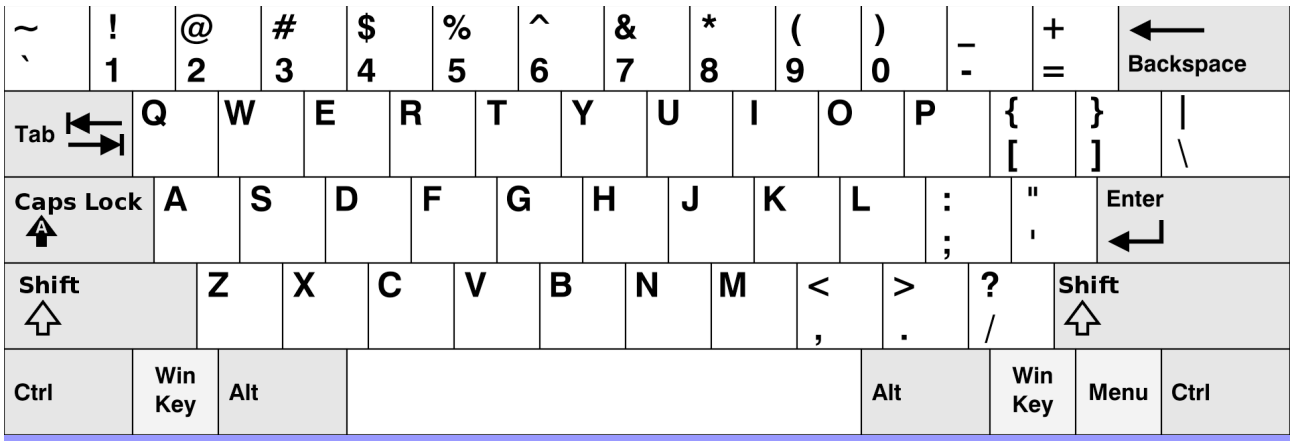
I told you before that every time something happens an event is created and that love will try to execute the corresponding call-back function if it actually exists. Well, love.keypressed() and love.keyreleased() and these functions are called exactly on the moment their names imply. When a key is either pressed or released.

Now these functions alone are of course useless when you do not know WHICH keys are actually being pressed or released, so yes, the callback feature is even able to give parameters to these functions.

The first parameter is the name of the key as a string. The scancode is the name of the code of the key (there are more, but that's for the nerds among us, and they may check the wiki).

There are many different kind of keyboards. Since the standard QWERTY setup in the US style is most common, LÖVE will assume you have that type of keyboard. This DOES mean that if you are in Belgium or France having an AZERTY keyboard that pressing the “A” will therefore result into “q” being generated as scancode string since the “A” is there where normally the “Q” should be. And when you are dealing with keyboards for the Cyrillic or even other kinds of scripts things could be even more odd.

Now this may appear illogical, however there's reasoning to this, and that is most likely due to the WASD controls modern gamers expect your game to support. This approach will make the game always have the same feeling for players going for WASD control regardless of the keyboard layout they have. This can be a bit confusing but if you wanna be sure you got everything alright THIS is the layout we're talking about:



Please note that the F-keys, the numberic keypad and the keys between that pad and the main keypad were left out as they are on normal Windows keyboards (which is also used on Linux) always the same. Mac keyboards can differ on this a little, but you can if you want just connect a normal Windows-keyboard to a Mac and then everything is the same.

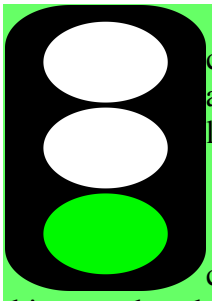
So you can use the keycode if the way the keys are set up matters, and the scancode the location of the key matters.

Now a full lists of all the scancodes can be found on the LÖVE wiki.

Let's get to code everything, ok?

It's up to you if you want to keep the old code or replace it with the new, as long as we got our Ball.png picture in the project folder and here goes:

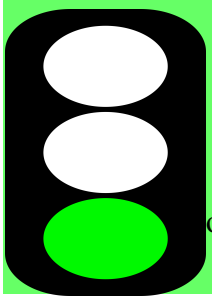
```
function love.load()
   ball = love.graphics.newImage("Ball.png")
   x = 50
   y = 50
   mx = 0
   my = 0
end

function love.update()
    x = x + mx
    y = y + my
    if x<-40 then x=801 end
    if x>832 then x=-38 end
    if y<-40 then y=601 end
    if y>632 then y=-38 end
end


function love.keypressed(key,scancode)
   if scancode=="up"    then my=-1 end
   if scancode=="down"  then my= 1 end
   if scancode=="left"  then mx=-1 end
   if scancode=="right" then mx= 1 end
end

function love.keyreleased(key,scancode)
  if scancode=="up"    and my<0 then my=0 end
  if scancode=="down"  and my>0 then my=0 end
  if scancode=="left"  and mx<0 then mx=0 end
  if scancode=="right" and mx>0 then mx=0 end
end


function love.draw()
   love.graphics.draw(ball,x,y)
end
```



Of course I could break this entire code down for you, but maybe the time has come you do that yourself this time. Now that you've come this far, you must be able to understand how the code above works and break everything down line by line.

Also, if you can play around with stuff a little, now's the time to try it out.

The code above shows some very basics you should understand if you want to create a game in general regardless of any engine, and as far as LÖVE is concerned, this one takes the deal.



And then one more puzzle for you!

You can make the ball move faster by altering this code.

The simplest way is by altering four lines in the code above.

And for this question you only need to alter 1 character in all four of these lines in order to make the ball move twice as fast as in the original code.

Can you do this?

8.2.4 Mouse control in LÖVE (and this is also usable for touchscreen)

The last callback functions, I'll be teaching you, are the ones you need for proper mouse control. (you may find out the others in the wiki when you feel you're up to it). Now LÖVE has multiple callbacks for this, and let's discuss them one by one.

8.2.4.1 love.mousefocus()

The official description is if your game window gets or loses focus.

In English for dummies, it basically means if the mouse pointer is within your game window or not. A parameter is given with the boolean value “true” if the mouse pointer is moved to a spot where it “enters” your window, and “false” when it “leaves” your window.

If you want you can try this code taken directly from the LÖVE-wiki:

```
function love.load()
 text = "Mouse is in the window!"
end

function love.draw()
  love.graphics.print(text,0,0)
end

function love.mousefocus(f)
  if not f then
    text = "Mouse is not in the window!"
    print("LOST MOUSE FOCUS")
  else
    text = "Mouse is in the window!"
    print("GAINED MOUSE FOCUS")
  end
end
```

8.2.4.2 love.mousemoved()

This callback is called whenever the user moves the mouse inside the game window.

5 parameters are given with this callback.

- *x,y* are used to just tell you on which coordinates the mouse currently is now that is has moved.
- *dx,dy* are used to tell you how much the mouse has moved since the last move event. Now this may be a bit abstract to understand, but if the mouse is now on (100,100) and I move it to (90,110) *dx* will be -10 and *dy* will be 10, as I moved the mouse 10 pixels left and 10 down.
- *touch* is a boolean variable and is true if the event was triggered by the touchscreen in stead of the mouse. This (in combination with *dx* and *dy*) are most of all set up to make LÖVE friendly for mobile devices, and swiping might be easier to code this way.

Now this will be fun if we can write a little demo on this callback. I will focus on the first two parameters now and I am not aiming for mobile devices right now, and come up with this cute little demo.

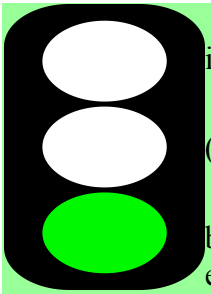
```
function love.load()
   ball = love.graphics.newImage("Ball.png")
   x = 50
   y = 50
   mx = x
   my = y
end

function love.update()
   x = x + math.ceil((mx-x)/5)
   y = y + math.ceil((my-y)/5)
end

function love.mousefocus(b)
    if not b then
       mx = x
       my = y
    end
end

function love.mousemoved(x,y,dx,dy,touch)
   mx = x
   my = y
end

function love.draw()
   love.graphics.draw(ball,x,y,0,1,1,16,16)
end
```

Now this is a nice little program in which the ball will chase the mouse pointer. For those who are not familiar with the “ceil” functions in programming, in most computer languages they are present as a function that rounds everything to integers and it always rounds up. So 1.2 and 1.4 and 1.7 are all rounded up to 2. If you wall want it rounded down you have “floor” functions which is in Lua indeed math.floor()

You can basically see that I just made the ball respond to the coordinates of the mouse which I stored in “mx” and “my” each time they were changed due to mouse movement thanks to the love.mousemoved() callback. I made the ball chase the mouse pointer due love.update() function calculating a nice way to make the ball speed dependent on the distance between the mouse pointer and the ball.

Yeah, when you were paying attention you can see that I added more parameters to my love.graphics.draw() function than I did before. This is advanced stuff you only need to worry about when you are really getting good.

The zero after the y is rotation in radians. As I didn't want to rotate the ball, I left it 0. The two ones are for scaling. 1 is normal size. 2 is twice the size and 0.5 is half the size and 0 is totally disappearing. The two 16 parameters were the reason I did all this. The ball picture is 32x32 pixels, but as 0,0 is normally the point to draw from this would make the program act ugly in the chasing of the mouse pointer. I figured it would be better if the mouse pointer would be in the middle once the ball comes to a standstill and thus by setting the “origin” coordinates to 16 (as 32:2=16) I could achieve that.



Before I explain the task itself I shall first tell you about love.graphics.line which is (like the name implies) for drawing lines (tadaaa!)

It works as easy as this love.graphics.line(10,60,40,50) will draw a line from (10,60) to (40,50).

Now your assignment is this... Make a program in which a line from the top to the bottom of your window and a line from the left to the right of your window cross each other on the position of the mouse pointer.

And if you could do that, expand it a little (hahaha) by making the lines only appear if the mouse is actually within your window (so in other words, make them disappear when the mouse “leaves” the windo).

8.2.4.3 love.mousepressed() and love.mousereleased()

Well, this should be easy, I suppose, as this works roughly in the same manner as the keyboard callbacks.

The parameters both functions accept are these:

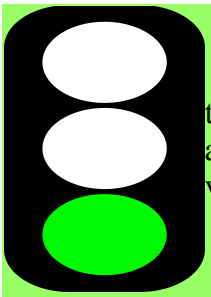
- *x,y* The coordinates of the mouse when the click or release took place.
- *button* This will contain value 1 for the left mouse button, 2 for the right mouse button and 3 if the scrollwheel was pushed (yeah, you can push the scroll wheel, did you know that?)
- *istouch* This will contain true if the press or release was caused by a touchscreen in stead of the mouse.
- *times* The number of presses in a short time. This is most of all intended in order to support double-clicking.

I hear you wonder why *x,y* when we already have love.mousemove()? That is because when you only need to check clicks, having to rely on love.mousemouse() is simply impractical. However if you want your game to have its own mouse pointer or have swipe possibility love.mousemove() is a function you won't be able to do without. That's the basic idea.

```
function love.load()
  ball = love.graphics.newImage("Ball.png")
   ballx = 400
   bally = 300
end


function love.mousepressed(x,y,button,touched,times)
    ballx = x
    bally = y
end

function love.mousereleased(x,y,button,touched,times)
    ballx=400
    bally=300
end




function love.draw()
   love.graphics.draw(ball,ballx,bally,0,1,1,16,16)
end
```

The program above can demonstrate how you can work this all out.

When you press the mouse button, the ball will be on the place where the mouse was when clicking and be back to the center when you release the button. The ball will not follow as you move around.



Make an empty table variable and call it “mytable”.

Now once somebody clicks the left button, a picture (yeah, use my ball picture for that if you want) should appear on the spot where you clicked, and remain there, and you should be able now to work out how you can do that with your table variable.

Now if you want to go one step further. Use “table.remove(mytable,1)” in order to remove the first picture that appears in the table whenever somebody presses the right mouse button.

8.2.4.4 Final thought

Well, and that's as fas as I will go on callbacks. From here on you should be able to deal with the most important callbacks, and if you really master all this, you can check the wiki for more possibilities. As my prime concern was to teach you Lua and not LÖVE (but I used LÖVE in this book as it's a free ready-to-go engine and therefore perfect to see Lua perform in actual projects).

Now what you can do with all the commands inside LÖVE should be easy to find out, with the understanding of Lua you should have by this point. Good luck.

8.3 Saving data in LÖVE (and also handy for Lua in general).

I've been asked about this explicitly when I was writing this book about saving data when you using LÖVE. Now Lua has its own io module in order to perform reading and writing files in Lua, which has been entirely based on the basic routines the C language provides, and there's documentation all over the place about that, and if you want me to cover that, tell me, but I will then cover that in a different chapter or section. ;)

Although all functions in the io module work in LÖVE, still its developers decided to come up with a safer routine, and which should also take away the hurdles of differences in system.

LÖVE will create a special directory in accordance of the rules of the underlying system and put all saved data there, and it will also be easy to retrieve it from there.

This table describing that has been copied from the LÖVE wiki and answers it all.

OS	Path	Alternative	Notes
Windows XP	`C:\Documents and Settings\user\Application Data\LOVE\`	`%appdata%\LOVE\`	-
Windows Vista, 7, 8 and 10	`C:\Users\user\AppData\Roaming\LOVE`	`%appdata%\LOVE\`	-
Mac	`/Users/user/Library/Application Support/LOVE/`	`~/Library/Application Support/LOVE/`	-
Linux	`$XDG_DATA_HOME/love/`	`~/.local/share/love/`	-
Android	`/data/user/0/org.love2d.android/files/save/`	`/data/data/org.love2d.android/files/save/`	On Android there are various save locations. If these don't work then you can use `love.filesystem.getSaveDirectory()` to check.

(NOTE! I altered the “Alternative” field for Mac, as the original wiki states Mac doesn't have that, which is a lie, it does, and I added it).

Now “user” is to be replaced with the username on your operating system and “LOVE” or “love” will be replaced with your project name. There are ways to change that, but that is for the more advanced user to look up in the wiki, I shall for now stick to the default methods.

Now you can put all data you want to save into one big string. For the good of this book we'll put that string in the variable “data”.

Now this instruction: “success, message = love.filesystem.write('mydata', data)” will save the entire string into the file “mydata” and store it in the folder in accordance to the table above. The

variable “success” will in this case contain a boolean. When it's “true” the operation was successful, and it's “false” the operation failed and the variable “message” will then contain a string containing the error message, so you can find out what went wrong.

Now you can load this data as simply as “contents, size = love.filesystem.read(“myfile”)” and the data will be loaded as a string into contents and the size contains the number of bytes read. If this operation fails contents will be “nil” and then “size” will contain a string with the error message.

Now a very popular method for saving data in Lua is by means of a serializer. This will simply store the variable and all it contains in a string containing Lua formatted code which Lua can compile, execute and return, giving you the data exactly the way it was on the moment it's stored.

A way to build a serializer yourself (or basically the code I wrote myself for this) can be found in the chapter about advanced tricks, so check it out.

8.4 Distributing a LÖVE project

Now this is covered well in the wiki of LÖVE, but as I wanted to close this chapter in a proper way, I'll cover it.

Now the first thing you wanna do is create a .love file. Now this is easy. .love files are nothing more but zip files containing everything in the project folder only with .zip replaced with .love, easy as pie. You can just try this and use the “love” run instructions I explained earlier in this section to see for yourself. Stuff will run.

Now in Windows you can use WinZip for the job, and on Mac you can just right-click the folder in the finder choose “Compress Folder” in the pop-up window. Rename your file properly and done. In Linux this method can differ per distro (dang!) but most Linux distros should come with tools to create zip files.

Now so far the easy part. How to turn everything into a good distributable game, with its own icon and that does not need any messing with the command line interface, is (dang) different per OS (we all saw that one coming, didn't we?)

8.4.1 Distributing a LÖVE project in Windows

Now this is tricky, and requires the command line....

1. Create a folder in which you want “compile” your release, and put all the files of LÖVE itself in them.
2. Use the “cd” command to go to the folder where you placed love.
3. Now it's easier if you copy your .love file into this directory as well.
4. On the prompt type: “copy /b love.exe+mygame.love mygame.exe”
5. Now mygame.exe is your run file, but you want to get rid of needless files and to give it an icon.
6. Type “del love.exe” (we no longer need this)
7. Type “del mygame.love”) (we no longer need this either)

8. Now create an icon with your favorite icon creation tool
9. Now download and install Resource Hacker.
10. Load “mygame.exe” in Resource Hacker
11. CAREFUL NOW! You're playing with power!
12. Search the directory tree and you'll eventually find the love icon
13. Remove it
14. Now implement you own icon
15. Save your executable and close Resource Hacker

And now your distribution should be ready. Pack this entire folder with zip or rar or 7z and upload it to wherever you want to post it!

8.4.2 Distributing a LÖVE game for Mac

This is by far easier than than in Windows, I tell ya, and most of this can just be done in the Finder.

1. Copy LÖVE anywhere on your system
2. Rename the love application to the name you want your game to have. For now “mygame” will suffice.
3. Right-click the application and chose “Show Package Contents” in the menu.
4. You will see directory only containing a folder named “Contents” open it
5. Now open “Resources”
6. Copy your .love file here. If you name your app “mygame” best is to name your love file “mygame.love”
7. You will also find an .icns file. That's the icon. Just create an icon for Mac in your favorite icon editor and replace the original icon with this one.

And that's it! Now your distribution is ready.

Now in Mac you can set up some extra meta-data if you like, but you don't need to, but it can make your game look more professional.

For this you can edit the file “info.plist” which is located in the Contents folder. It's just a text file in .xml syntax, and you can open it in your favorite text editor.

Here you can alter some data, to which MacOS can react and also your “about” screen can be altered by this. On the wiki you can file all details about this.

8.4.3 Linux notes

I will not go into the depths of Linux, simply because Linux and I are enemies. However the wiki has this globally covered. Now the best option is to go for an AppImage, as this will take away all the needs for dependencies and merge your game into one big file, and it's said that all Linux distros SHOULD be able to run your game then (I put emphasis on “SHOULD” as in Linux, you never know). Since I don't have a well-running Linux system right now (my VM is not that kind on Linux either), I cannot test this, and therefore I cannot fully cover this, sorry about that.

8.5 Final words on LÖVE

LÖVE is just one out of the many engines that support Lua, and the fun part is that there are loads of experiments to make LÖVE compatible with more languages than just C. My very own LAURA II engine which I used for Star Story and The Fairy Tale REVAMPED, were completely coded in BlitzMax and not in C, and LAURA II uses Lua as scripting language as well.

As long as you do not have professional ambitions, LÖVE will mostly suit all your needs, and even for professionals LÖVE can cut the bill on many fronts. Of course, if you think LÖVE can no longer help you and you feel an engine that can do stuff LÖVE doesn't support, then mastering LÖVE will by no means be a waste of time. All engines supporting Lua, have basically much of he same logical background since it's all Lua.

And if you really go into the hard core stuff in Lua, you'd be surprised what you can do to step outside the boundaries LÖVE has technically set up, without having to dismiss LÖVE.

A. A few extra things

This entire “chapter” will be devoted to a few things you may not “need” to do basic stuff in Lua, but which can help you greatly on the way to “greatness”. Some things can be hard to understand, and some are pretty easy.

I may not really give much assignments anymore unless things are getting complicated. You are at a point now that you should be able to make up some practices yourself. Most chapters can be rather short as a result.

We've also come to a point that the playground may also not take you very far. Using a Lua cli tool can help, and some examples may also work in LÖVE.

My way of explaining may also differ, as you should understand some core fundametals of Lua programming now!

A1. Modules

(NOTE! None of the examples in this section work in the playground. Best is to use the CLI tool)

Perhaps when you've been looking through a source file written in C you've seen many “#include <blahblah.h>” lines, haven't you? That is where you can include the “headers” of a library into your own program so C knows you are using code from an external source. Most other programming languages have similar systems.

Lua is no exception.

This is done out of basically two basic philosophies, which are very close.

a. Why re-invent the wheel when you already did this in previous projects?

b. Why re-invent the wheel when somebody else already did it for you?

Now the exact approach for this, can differ per programming languages, but Lua has a pretty modular approach to this. An external file is being loaded by Lua and Lua handles that entire fire as a function, and yes that function can return stuff.

Let's first do the simple “dirty” approach:

```
-- mymodule.lua
function Hello()
    print("Hello World")
end
```

```
-- main.lua
require("mymodule")
Hello()
```

You'll need to put these two files in the same folder and now when you run main.lua then the “require” instruction will execute mymodule.lua as if it were a function. Since functions can create global functions too, this will simply happen and thus the function Hello() will be created and thus “Hello World” will be the output of the program in general.

This is considered “dirty” and is discouraged, but it works, and you need to decide for yourself

now what is “dirty” and what is not.

Since modules are in Lua just functions which are loaded when the “require” instruction falls, they can as such also return values. They mostly happen in an object oriented syntax (by convention), but it's not strictly needed, but that's your choice ;)

Example in OOP notation:

```
-- importme.lua
local module = {}

function module:ExHello()
     print("Hello World")
end

function module:Hello()
    self:ExHello()
end

return module
```

```
-- main.lua
m = require("importme")
m:Hello()
```

Of course, the example above is put in a more complicated way than strictly needed, but I just wanted to show you that inside the module methods can be called the same way is in regular OOP calls. This notation is considered the “official” way, and is if you have full understanding of OOP also the recommended way to prevent conflicts when things really get complicated.

A non-OOP notation is (of course) also possible.

```
-- importme.lua
local module = {}

function module.ExHello()
     print("Hello World")
end

function module.Hello()
    module.ExHello()
end

return module
```

```
-- main.lua
m = require("importme")
m.Hello()
```

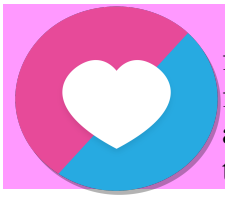
This may not really matter if you only write modules for yourself, but when distributing modules for other people to use, they will OOP notation, but of course you can document your module to use the non-OOP notation, but you must be clear on that one. Also note that the tables I used to create a module are all defined

as locals. Although working with non-locals works (as demonstrated in the “dirty” method), this may lead to conflicts if people unknowing us the same name for their variables as you do, and therefore it is heavily recommended to only use locals in a module, and to put all functions and even variables the program calling the module inside a table and to return that table. Better programmers than you managed to create a total mess by neglecting to properly take these things into account!

You may have seen that I never included “.lua” in my 'require' requests. That is because doing is is FORBIDDEN and your program will crash if you include “.lua”.

The period character has a different meaning in a require request and it will be replaced by “\” in Windows and by “/” on Unix based systems, and thus a directory separator. So when I say “m = require('module.lua')” Lua will try to look up “module/lua.lua”, and since that file doesn't exist, well, there we go!

And I guess with this note, I also taught you how to do modules in other directories. It is not possible to “go back” in the directory tree, unless the underlying engine as support for doing this.



The LÖVE engine has full support for importing modules using the 'require' function, and yes even when you have packed all your Lua script into one bit .love file, and in the latter case it will just look everything up in the .love file accordingly. The LÖVE engine has a completely adapted version all included, and this should even work on files stored in the save folder.

Very important is to note that Lua was designed to eliminate the differences between platforms and the file systems they use! Now it's not really the platform itself that decides this but rather the file systems used in them, but the formats common on Unix systems are case sensitive, and you should treat your module files just as such, even when you are working on Windows.
(Now Linux loses its case sensitive behavior when reading formats meant for Windows, such as FAT32, NTFS, ExFAT etc, but even then case sensitivy should be taken in order!)

A2. Strings as code

Yeah... Believe it or not, but Lua can consider string values to be code. Yes, really!

```
a = "print(\"Hello World\")"
b = load(a)
b() -- outputs "Hello World"
```

Well important to note is that I had to write \” in stead of just ”. This is because Lua would elseway end the string after print(causing errors in the process, and by using the backslash I've told Lua not to end the string, but just to put a quote in the string in stead (this is code an escape code).

The load function loads the string and compiles it into a function and returns the compiled function and thus *b* became a function with the instruction to output “Hello World”.

In Lua programming it's rather popular to save the data of a program (like savegames or configuration data) just as Lua code by use of serializers, and to load these just as strings and to compile them with the load instruction and tadaaaa... there you go!

A3. String escape codes

Lua works with escape codes in order to allow you to put stuff in strings that would normally not be allowed. In the previous section you could see me use \" in order to make the quotes part of the string. Here's a list of the others:

- `\a`: Bell
- `\b`: Backspace
- `\f`: Form feed
- `\n`: Newline
- `\r`: Carriage return
- `\t`: Tab
- `\v`: Vertical tab
- `\\`: Backslash
- `\"`: Double quote
- `\'`: Single quote
- `\nnn`: Octal value (nnn is 3 octal digits)
- `\xNN`: Hex value (Lua5.2/LuaJIT, NN is two hex digits)

Some of them may require more knowledge than others, and I won't go into the deep of all of them. It's just handy to have an overview. And you may try them out with the print command if you like.

A4. Hexadecimals in code

When you get more advanced into coding you will may need of the hexadecimal system. This system just works like the decimal system, in stead that 6 extra "numbers" have been added making a total of the digits: 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f

In Lua you can create a hexadecimal number by prefixing it with "0x"

"print(0x3f)" will output 63 as 3f is the hexadecimal counterpart of 63.

A5. Anything but "false" and "nil" is true

Yeah, that's how checkups frankly work in Lua.

```
a = "Hello"
if a then print(" true: a") else print("false: a") end
if b then print(" true: b") else print("false: b") end
```

When you run this you will see that a is deemed true due to it containing a string, and since b is still "nil" it is deemed "false". Now if you put in a number into *a* it will still be true, and a table will also make it true. Only "nil" and "false" are deemed false.

In C the number 0 is considered false, but in Lua the number 0 counts as a value and is as such true. There are also programming languages in which empty strings or arrays/dictionaries (brought together as tables in Lua) also count as false, but Lua will also deem these true. Anything but "nil" or "false" is true in Lua. PERIOD! Never forget this, as this has confused many people who do more than just Lua!

A6. Define me if I wasn't defined before

This is a trick, many programmers speak very ill of, but hey it works.

```
function h()
    a = a or "Hello World"
    print(a)
end

h()
a = "Hallo Welt" -- Means "Hello World" in German
h()

-- Output
-- Hello World
-- Hallo Welt
```

This is dirty code straight from Hell, but it works!

In order to explain this, we must get a bit into the way Lua works “under da hood”

The first time h() is called the global *a* was undefined, and we do know that all undefined variables in Lua contain the value *nil*. I told you in section A5 that anything but *nil* and *false* is *true*. The “or” keywords will stop scanning for anything coming after it when it already reached a *true* value. So the first time a is checked it's *nil* and thus considered as *false*, and the second value being a true value is thus being put in in this case “Hello World”.

Now I changed *a* afterward, but that doesn't matter, as German or English, *a* does contain a value so h() will define *a* with itself and when “or” comes in it already has a *true* value and this the rest is no longer relevant and thus ignored.

You could consider this a “glitch” (which is often confused with the word “bug”, but a glitch is not the same as a bug) as this was not really what the creators of Lua intended to be possible, but it is possible and even considered official by now.

This is often used in order to wait with creating a variable until the very moment it's needed (prevents waste of RAM after all), and it is also a safeguard against unwanted “nil-value” errors.

A7. “if”? “if” is not good!

The title of this section is a parody of Pain and Panic in the Disney movie “Hercules”.

The “if” command comes back in many programming languages, and there is a kind of debate about them. Some programmers deem the “if” command as “evil”, and that is should be avoided whenever possible, or even stronger that there is no valid excuse why it should be used at all. Now this debate does not go as far as the debate about the “goto”-command (which Lua does not even support), and there's it's proved you don't need it and that it mostly creates only dirty code (which is why Lua doesn't support it), but what is true is that many “if” commands can even in Lua (which does not support casing, for reasons beyond me) be avoided.

By using (or faking) OOP for example you can do a great deal, but there are more ways to do this.

Tables, are a very good way to avoid needless “if” commands. In the chapter about tables some very good examples have been shown in for example “a['cat']='meow'; a['dog']='woof'”, etc, but sometimes you need more than just a number of a string. But why bother since you do know you

can also put functions into a table.

Now I challenge ya to decipher this code, and if you can fully tell how it works, you are on the way to fully do dirty, yet great, things with Lua.

```
function unknown()
    print("I don't know that animal")
end

animal = {}
function animal.dog() print("woof!") end
function animal.cat() print("meow!") end

function speakanimal(a)
     (animal[a] or unknown)()
end

speakanimal("dog")
speakanimal("cat")
speakanimal("bird")
```

Everything you need to know to explain how this code works has been discussed before, so try to figure this one out, eh?

Okay, and now something the opposers of the “if” instruction should really think about, as THIS is really extremely dirty... Personally I'd prefer an “if” instruction :P

```
FU = {}
FU[true] = function()
      print("This is all cool! Oh yeah!")
end

FU[false] = function()
      print("This is not good! Oh no!")
end


a = 1

FU[a==1]()
```

Of course, this DOES work without a single “if” instruction, but I think it's a little bit cumbersome, but that's my opinion :P

So yeah, if you can avoid “if” commands and produce even clear code in the process, go for it, but there *is* something as overkill, and creating cumbersome code just to avoid “if” commands is in my humble opinion not the way to go.

A8. Optional parameters

Many programming languages do have support for optional parameters. Unfortunately Lua is officially not one of them.

An example in C# to show what I mean:

```
using System;
public class Program
{
        public static void Hello(string name="Jeroen")
        {
                Console.WriteLine($"Hello {name}");
        }
        public static void Main()
        {
                Hello();
                Hello("Mr. President");
        }
}
```

The output will be:

```
Hello Jeroen
Hello Mr. President
```

Now since this is about Lua and not about C# I will not go to break everything down, except for that the "name="Jeroen"" definition in the declaration of the function "Hello()" makes that if no parameter is given when calling "Hello()" the string variable "name" will automatically contain "Jeroen", and contain the string given if a parameter is given.

Now Lua does NOT support this entire getup at all, but that doesn't mean it's impossible to find a way around it. You remember that any parameter that is not given in Lua is automatically "nil"? And you also remember what I said in the "Define me when I'm undefined" section? Let's combine these too.

```
function Hello(name)
    name = name or "Jeroen"
    print("Hello "..name)
end


Hello()
Hello("Mr. President")
```

And so I faked the effect I did before in C# into Lua.

Cool, huh?

fd

dfdf

s

d